Topic 1: C Programming Language

The First C Program

Guidance

- Chapter: C-Primer-Plus.pdf Chapter 1-2
- Spend time: 1 day
- Learn suggestion: Write canonical C code
- Key points:
  - Why do we need C Programming Language? What can it be used for? What are the advantages and disadvantages?
  - How many C language standards are in C?
  - How to make the binary file for different standards in gcc? What other options can be used in gcc?

Practice

1. How to add a comment for a function?

The single-line comment isn't recommended because some compilers do not support it.

```
/*****************************************************************************
 *
 *      Name:rt_bfd_register
 *
 *      Input:
 *              proto   - protocol code
 *              process - protocol process information
 *              func    - callback function
 *
 *      Return: TRUE -- Success
 *              FALSE -- Failed
 *
 *      Description:resigter the protocol information include callback function
 *
 *****************************************************************************/
bool rt_bfd_register(uint32 proto, uint32 process, int (*func) (uint32, uint32, struct
rt_msg *))
{
    // function body
}
```

2. What should we pay attention to when declaring a variable?

- The variable name consists of letters, digits, and underscores, starting with a letter or an underscore. Commonly, a variable starting with a letter

is used for regular users, and beginning with an underscore is used for system base functions.

```
int a, a_1, a1, a1_;    // valid
int 2a, 3_a, a#, a?;    // invalid
```

You can find here that there are really many functions whose names start with an underscore

https://github.com/torvalds/linux/blob/master/include/linux/socket.h

- There are 32 reserved words in the C programming language; the variable name defined cannot be the same as the reserved word.

```
int for = 2, case = 3, if = 0;    // invalid, for/case/if are reserved words
```

- Variable names are case-sensitive.

```
int abc = 2, ABC = 3;           // variable abc and ABC are different
```

- Variable names should be as meaningful as possible, Which can improve code readability

```
// We can got more information from ip_bitmap and arp_timeout than aa and b
int ip_bitmap, arp_timeout, aa, b;
```

3. Can a C program exist without the main function? How is this possible? Why?

There are 3 concepts
User-level, all functions written by the general programmer in the program
System-level, file system, elf in Linux
libc-level, the function between the user and the file system, in order to help users using C, such as libc_start_main

- main() is the default function name of user-level
  - Step 1, there is a program named test.c, get an executable file a.out by command "gcc test.c"

```
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

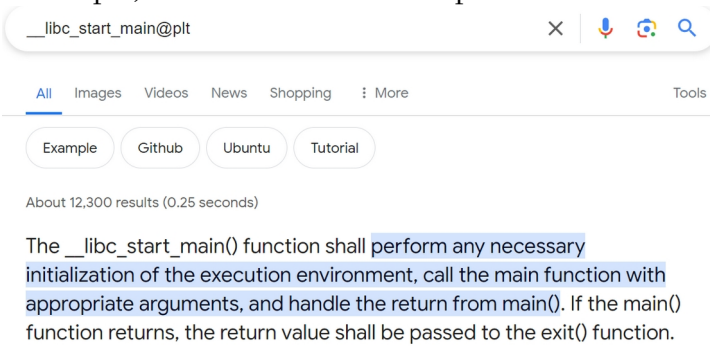
- Step 2, get the Entry point address by command "readelf -h a.out | grep Entry"

```
[xiaohei@localhost share]$ readelf -h a.out | grep Entry
  Entry point address:               0x8048310
```

- Step 3, find address 0x8048310 from the output of the command "objdump -d a.out"

```
08048310 <_start>:
 8048310:       31 ed                   xor    %ebp,%ebp
 8048312:       5e                      pop    %esi
 8048313:       89 e1                   mov    %esp,%ecx
 8048315:       83 e4 f0                and    $0xfffffff0,%esp
 8048318:       50                      push   %eax
 8048319:       54                      push   %esp
 804831a:       52                      push   %edx
 804831b:       68 a0 84 04 08          push   $0x80484a0
 8048320:       68 30 84 04 08          push   $0x8048430
 8048325:       51                      push   %ecx
 8048326:       56                      push   %esi
 8048327:       68 0d 84 04 08          push   $0x804840d
 804832c:       e8 bf ff ff ff          call   80482f0 <__libc_start_main@plt>
 8048331:       f4                      hlt
 8048332:       66 90                   xchg   %ax,%ax
 8048334:       66 90                   xchg   %ax,%ax
 8048336:       66 90                   xchg   %ax,%ax
 8048338:       66 90                   xchg   %ax,%ax
 804833a:       66 90                   xchg   %ax,%ax
 804833c:       66 90                   xchg   %ax,%ax
 804833e:       66 90                   xchg   %ax,%ax
```

- Step 4, what is the relationship between start_main and main



- How to change the default entry of user-level

As you can see from these four steps, the main is just the default function entry at the user level, which is called by libc_start_main. Because the libc_start_main call to main is fixed and cannot be modified, you cannot use libc_start_main if you want to achieve your purpose.

- Step 1, option -nostartfiles can be used

`-nostartfiles`

Do not use the standard system startup files when linking. The standard system libraries are used normally, unless `-nostdlib`, `-nolibc`, or `-nodefaultlibs` is used.

- Step 2, change test.c to blew

```c
int func1(void)
{
    printf("Hello World 1\n");
    return 0;
}

int func2(void)
{
    printf("Hello World 2\n");
    return 0;
}
```

- Step 3, get a new executable file a.out and run it by command "gcc test.c -nostartfiles -efunc2 && ./a.out"

```
[xiaohei@localhost share]$ gcc test.c -nostartfiles -efunc2 && ./a.out
Hello World 2
Segmentation fault (core dumped)
```

- Why is there a segmentation here occurrence

exit() will be called in libc_start_main to perform some clearing actions When main() returns normally. However, the link to the startup file of libc is closed, libc_start_main is not enabled, and no clearing action is performed, resulting in an exception. So, the issue can be fixed by changing the return to exit.

4. How does a C source code become an executable file, Fill in the table below

Tool Name				
gcc tool				
gcc option				
function				
input				
ouput				

Tool Name	pre-processor	Compiler	Assembler	Linker
gcc tool	gcc built-in	ccl	as	collect2
gcc option	-E	-S	-c	-o
function	Expand header file, macro definition, conditional compilation and so on	Change C code to assemble code. It includes lexical analysis, grammatical analysis and semantic analysis	Change assemble code to object file. Which is binary machine instruction	Link different object files and library files to an execalbe file
input	source file	expand file	assemble file	object file
ouput	expand file	assemble file	object file	execalbe file

Functions

Guidance

Functions
Guidance

- Chapter: Pointers-On-C.pdf 7.1-7.3; The-C-Programming-Language-2nd.pdf 4.1-4.2
- Spend time: 1 day
- Learn suggestion: Compare with different types of functions
- Key points:
  - Can variables defined in one function be accessed in another function?
  - What will happen if you define a function with the static keyword?

- Write a simple program and read the assembly language

Practice

1. Why do we need functions? How to define a function? What does a function consist of? What are the naming rules for function names?
   a) The reason for using functions are

   - Structurized, Separate a big logic into smaller logic. Make your code more organized
   - Uncoupled， Each function is independent of the other， Enhancing the code maintainability
   - Reused， It can be called multiple times in different places

   b) The syntax of a function definition is

```
return-value function-name(parameter list)
{
    // body
    return value;
}
```

   c) The syntax of function declaration is

```
data-type function-name(parameter list);
```

   d) The naming rule of the function is totally the same as a variable name
2. Why do we need function declaration? If there is no function declaration, sometimes the compiler will report an error, sometimes report a warning, sometimes not any error or warning. Why is this? Illustration separately
   a) Why do we need function declaration?

This is a historical legacy issue because the C language was born 40 years ago, When CPU and memory resources were very limited. If there were no function declarations, the compiler would need to read the code multiple times to find the definition of the called function, consuming unnecessary CPU and memory resources.

Now that hardware resources are sufficient, why do we still need function declarations?

- With function declarations, the compiler only needs to read the code once, which can also improve compilation speed.
- Due to function declarations are short and usually concentrated at the beginning of header files or source files, they can help readers quickly understand the function. This improves the program's readability and maintainability.

- Improve program compatibility and increase code consistency. Because the code written now needs to be consistent with old code and needs to be used on compilers with different capabilities.

b) The compiler creates an implicit declaration for a function when the compiler finds it is undefined. The data type of implicit function declaration is int.

```c
#include <stdio.h>

int main(void)
{
    fun();
    return 0;
}

int fun(void)
{
    printf("Hello World!\n");
    return 0;
}
```

```
Terminal ×  +

root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c
main.c: In function 'main':
main.c:5:5: warning: implicit declaration of function 'fun' [-Wimplicit-function-declaration]
    5 |     fun();
      |     ^~~
root@dcs-6b203f0c-0:/workspace/CProject# ls
a.out  main.c
root@dcs-6b203f0c-0:/workspace/CProject#
```

c) If the actual return type of the function is not, an error will occur

```c
#include <stdio.h>

int main(void)
{
    fun();
    return 0;
}

short fun(void)
{
    printf("Hello World!\n");
    return 0;
}
```

```
Terminal ×  +

root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c
main.c: In function 'main':
main.c:5:5: warning: implicit declaration of function 'fun' [-Wimplicit-function-declaration]
    5 |     fun();
      |     ^~~
main.c: At top level:
main.c:9:7: error: conflicting types for 'fun'
    9 | short fun(void)
      |       ^~~
main.c:5:5: note: previous implicit declaration of 'fun' was here
    5 |     fun();
      |     ^~~
root@dcs-6b203f0c-0:/workspace/CProject# ls
main.c
root@dcs-6b203f0c-0:/workspace/CProject#
```

d) The compiler can work well when it has already found the definition of the be called function

```c
#include <stdio.h>

short fun(void)
{
    printf("Hello World!\n");
    return 0;
}

int main(void)
{
    fun();
    return 0;
}
```

```
Terminal ×  +
root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c
root@dcs-6b203f0c-0:/workspace/CProject# ls
a.out  main.c
```

3. What are the matters that need attention for function return values?

- The return value must match with return type

```c
unsigned int fun(void)
{
    return -1;
}
```

- Never return an address that is stored on a stack

```c
char* fun(void)
{
    char str[] = "Hello World!";
    return str;
}
```

- The return value is the right value, It can not do operator operations

```c
int test_fun(void)
{
    return 222;
}

int main(void)
{
    test_fun() = 2;
    return 0;
}
```

4. What is an inline function? What is the form of it?

A function declared by the keyword inline is an inline function. It will be expanded in the calling function in the compile stage. Which can save calling overhead

There are 2 forms of inline function

```
// The compiler will not be expanded the funcion in compile stage
static inline __attribute__((noinline)) int fun();

// The compiler will be expanded the funcion in compile stage
static inline __attribute__((always_inline)) int fun();
```

5. What is the difference between inline function, normal function, and macro function? What are their advantages and disadvantages?

	macro function	normal function	inline function
essence	replace text	function call stack	be expanded in the calling function
timing	pre-processing stage	running stage	compile stage
syntax check	No	Yes	Yes
advantage	Do not need new stack	No effect on the length of the calling function	Do not need new stack
disadvantage	Expanding makes the call function become larger, increasing the cost of calling the function	Need new stack and register	Expanding makes the call function become larger, increasing the cost of calling the function
applicable scene	constant or the function do not need syntax check	normally	The length of function is small and be called frequently

Function Call Stack

Guidance

- Chapter: Computer-Systems-A-Programmers-Perspective.pdf 3.4 3.7.1-3.7.4
- Spend time: 2 days
- Learn suggestion: Read the assembly language
- Key points:
  - When are local variables allocated space and when are they released space?
  - Why the value of the function argument did not change when we changed the value of formal parameters
  - How many ways can a computer store stack data? Which approach does Linux take?
- Reference Code

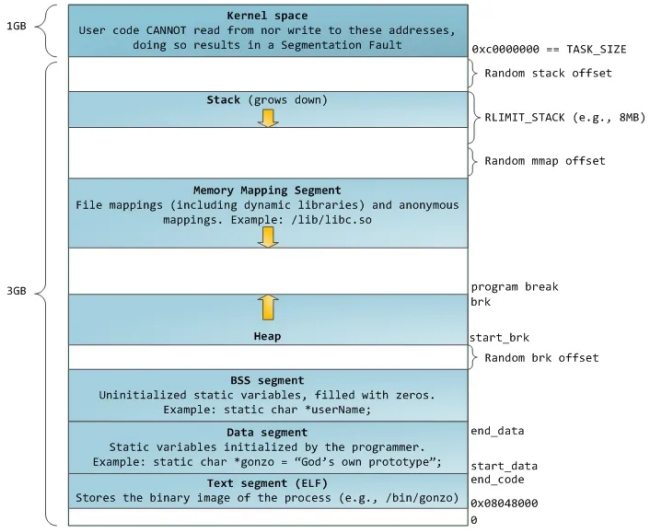
```c
#include <stdio.h>

int fun1(void);
int fun2(void);

int fun2(void)
{
    int aaa = 111;
    int bbb = 222;
    int ccc = 333;

    return 444;
}

int fun1(void)
{
    int aa = 11;
    int bb = 22;
    int cc = 33;
    fun2();
    return 44;
}

int main(void)
{
    int a = 1;
    int b = 2;
    int c = 3;
    fun1();

    return 4;
}
```

```
/*
fun2:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $111, -4(%ebp)
        movl    $222, -8(%ebp)
        movl    $333, -12(%ebp)
        movl    $444, %eax
        leave
        ret
fun1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $11, -4(%ebp)
        movl    $22, -8(%ebp)
        movl    $33, -12(%ebp)
        call    fun2
        movl    $44, %eax
        leave
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    $3, -12(%ebp)
        call    fun1
        movl    $4, %eax
        leave
        ret
*/
```

Practice

Suppose, The initial address of %ebp is 0x1100, %esp 0x1000 when the main function is called. The program runs on a computer with a decrement stack

1. Draw an image for the virtual address space of the Linux process in 4G memory



The top most region of the address space is reserved for code and data in the operating system that is common to all processes

The lower region of the address space holds the code and data defined by the user's process.

2. What is the difference between a, b and c? From the stack perspective

```c
void test_fun(int b)
{
    int c = 2;
    printf("%d\n", b + c)
    return;
}

int main(void)
{
    int a = 1;
    test_fun(a);
    return 0;
}
```

```c
void test_fun(int b) // b is a formal parameter
{
    int c = 2;           // c is a local variable
    printf("%d\n", b + c)
    return;
}

int main(void)
{
    int a = 1;
    test_fun(a);    // a is a actual parameter
    return 0;
}
```

The stack does not create any space for the formal parameter. It just a representation of function argument in C language level

The actual parameter is stored on the stack of the function who do call， and the variable in the function is stored on the stack of the function being called.

For more detail, refer to the program "Function call stack.c"

3. The practice is based on the program "Function call stack.c"
    a. Write the corresponding C code next to the assembly statement

    return address, is the following instruction address of the called function. It'll be stored in the PC register when the called function is returned.
    Refer to the program "Function call stack.c"

    b. Draw some pictures or tables to illustrate the memory layout and content of the register after each assembly instruction.

    Refer to "Function call stack.xlsx"

Topic 2: Basic Concept

Constant & Variable

Guidance

- Chapter: C-Primer-Plus.pdf chapter 3; Pointers-On-C.pdf chapter 3
- Spend time: 1 day
- Learn suggestion: The Linux tools objdump and readelf can be used to help you know more about how a variable is stored in memory.
- Key points
    - Where are the different types of variables stored in virtual memory?
    - What keywords can be used to declare a variable? What are the differences between them?
    - What kinds of constants are there in C?

Practice

1. How is C represent and store characters? What are the special characters? Write a program to describe the relationship between ASCII and characters.

In C, characters are printed with %c and numbers are printed with %d. Here we can take a look at the difference between characters and numbers by printing in.

```c
// Characters and numbers can be converted to each other,
// and the character corresponding to the number is the ASCII code
printf("A is %d, 65 is %c\n", 'A', 65);    // A is 65, 65 is A,

// Characters can be added and subtracted just like numbers
printf("A + 3 is %c, corresponding to %d\n", 'A' + 3, 'A' + 3); // A + 3 is D, corresponding to 68
printf("D - A %d\n", 'D' - 'A'); // D - A is 3, This means that there are 3 gaps between character A and character D
printf("D + A %d\n", 'D' + 'A'); // D + A is 133, There is not any meaning here
```

There are 128 characters in ASCII code

0-31	control	non displayable
127	control	non displayable
48-57	0-9	digit
65-90	A-Z	uppercase
97-122	a-z	lowercase
others	Do not need to remember	displayable

Special Characters

Broad definition：All characters except numeric uppercase and lowercase are special characters

Narrow definition：Control character(0-31,127) + escape character(some of here is also control character)

2. How to define and represent different bases in C? Write code to illustrate

```c
int binary_number = 0b1111;
int octal_number = 017;
int hex_number = 0xf;

printf("%d %d %d %d\n", binary_number, octal_number, decimal_number, hex_number); // 15 15 15 15
printf("%o %x %X\n", decimal_number, decimal_number, decimal_number);             // 17 f F
```

3. How to define and use enumeration? Write a piece of code to describe

- Enumeration can help the coder better understand code and enhance the readability of code

```c
switch(day)
{
    case 1:    // it is confusing
        printf("Today is Monday\n");
        break;
    default:
        break;
}
```

- The usage is very flexible

```c
enum weeks
{
    TEST_1,
    TEST_2,
    TEST_3 = 5,
    TEST_4,
    TEST_5 = TEST_2
};

printf("The default start value is %d, The next one increases by 1\n", TEST_1, TEST_2);
printf("The value can be changed %d, The next one increases by 1\n", TEST_3, TEST_4);
printf("The value can be changed %d like this form\n", TEST_5);
```

- It can also be combined with the typedef to define a new alias, Improve code readability

```c
typedef enum weeks
{
    MONDAY = 1,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
} week_t;

void func(week_t wt)
{
    printf("%d\n", wt);
    return;
}

int main(void)
{
    week_t wt = WEDNESDAY;

    func(wt);
    return 0;
}
```

4. Why do we sometimes say that string is constant? Illustration

- Compare the output of a and b with objdump -h to see which part is 5 bytes larger

```
a.
char *p;
p = "jerry";
printf("%s\n", p);

b.
char *p;
p = "jerry12345";
printf("%s\n", p);
```

- To check the address. Refer to the program "The address of constant and variable.c"

5. Is it possible to change the value of a const variable? How to do?

It's different for global and local variables.
global variables are declared by const are stored in .rodata, they cannot be modified anyway.
The corresponding local variable is stored on the stack and can be modified by the pointer.

6. Why do we need keyword static to declare a variable?

When it is used as a global variable, the scope of the variable changes to the file that defines it, that means, the variable cannot be used in a source file other than the current file.
It will be stored in the global variable table when it is used as a local variable . The variable’ s lifetime does not depend on the call stack.

7. Where are the following variables stored in the memory layout? Provide the evidence

```
int a = 1;
int b;
const int c = 3;
static int d = 4;
char *str1 = "Hello, World!";

int main(void)
{
    int aa = 1;
    int bb;
    const int cc = 3;
    static int dd = 4;
    char *str2 = "Hello, World!";

    return 0;
}
```

The address of stack and heap can be checked by command "cat /proc//maps"
Refer to the program "The address of constant and variable.c"

Operators & Expressions

Guidance

- Chapter: Pointers-On-C.pdf chapter 5
- Spend time: 1 day
- Learn suggestion: Thinking like a computer rather than a human.
- Key points
    - What is the relationship between expressions and statements?
    - What is the meaning of the operator's side effect?
    - When does short-circuit occur in expressions?
    - How a complex expression is executed?
    - Memorizing the priority of some operators is necessary.

Practice

1. What are the rules of expression operation?

You can understand it in your own way and don't need to memorize all of it. In particular, the operator precedence can be resolved with parentheses if you're unsure about it while programming. This enhances the code's readability; however, it may be necessary to memorize some common operators, otherwise, if there are a lot of brackets may impact the code's conciseness
I'd like to show you an example that I wrote here. It's not mandatory to follow.

```
/*We know that the monadic operator takes precedence over the binocular operator,
so we don't need parentheses here*/
if(!condition_a && *condition_b);

 /* The same is the monadic operator here, you can not add, but add parentheses,
 can make the code more readable */
 if((condition_a & condition_b) && (condition_c == NUM) || condition_d);
```

2. Pointers-On-C.pdf 5.8 question 5

```
int is_leap_year = 0, year = 0;
// A way to get the value of year
is_leap_year = ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

3. Pointers-On-C.pdf 5.8 question 6

Side Effect: There are some operators change the behavior of operands, and we call this change a side effect

The operators who have side effect: increment/decrement operator, assignment and compound assignment operator

Attention：Increment/Decrement operator is not recommended to use unless there is only one statement in the expression. Refer to next question.

4. What are the outputs of the following program? Try in different compilers

```c
#include <stdio.h>

int main(void)
{
    int a, b;

    a = 3;
    b = (a++) + (a++) + (a++);
    printf("%d, %d\n", a, b);

    a = 3;
    b = (++a) + (++a) + (++a);
    printf("%d, %d\n", a, b);
    return 0;
}
```

This the result of these 3 compilers. I want you know here is the result of experssions including the side effect operator is very depend of the compiler.

Compiler	Output
GCC	6, 12 6, 16
Turbo C	6, 9 6, 18
Pelles C	6, 13 6, 16

5. Pointers-On-C.pdf 5.8 question 7

Computers can't think as flexibly as humans and need to follow operators' rules. Both the 2 operators have the same priority and are executed from left to right.

```c
1 <= a <= 10;        // Be equivalent to b = 1 <= a; b <= 10; b is non 0
1 <= a && a <= 10    // It should be like this, If you want to implement the original logic
```

6. Please explain the output of the following program

```c
a.
int a[3][2] = {(1,2),(3,4),(5,6)};
printf("%d\n", a[0][1]);

b.
int x = 3, y = 0, z = 0;
if (x = y + z)
    printf("111\n");
else
    printf("222\n");

c.
int a = 0, b = 0, c = 0, d = 0;
d = (c = (a = 11, b = 22)) == 11;
printf("%d %d\n", c, d);

d.
int x = 3, y = 2, z = 3;
z = x < y ? !x : !((x = 2) || (x = 3)) ? (x = 1): (y = 2);
printf("%d %d %d\n", x, y, z);
```

```c
a.
int a[3][2] = {(1,2),(3,4),(5,6)}; // Be equivalent to {2, 4, 6};
printf("%d\n", a[0][1]);           // 4
// The value of multi expression separated by comma operator is just the value of the last ex

b.
int x = 3, y = 0, z = 0;
if (x = y + z)
    printf("111\n");
else
    printf("222\n");        // output
// The value of an assignment expression is the new value of the left operand.

c.
int a = 0, b = 0, c = 0, d = 0;
d = (c = (a = 11, b = 22)) == 11;
printf("%d %d\n", c, d);    // 22 0
// A combination of concepts comma expression and assignment expression.
/* steps
step 1:d = (c = 22) == 11  a is 11, b is 22
step 2:d = 22 == 11        a is 11, b is 22, c is 22
step 3:d = 0               a is 11, b is 22, c is 22, d is 0, == has a higher priority than =
*/

d.
int x = 3, y = 2, z = 3;
z = x < y ? !x : !((x = 2) || (x = 3)) ? (x = 1): (y = 2);
printf("%d %d %d\n", x, y, z);    // 2 2 2
// The short circuit in a logical judgment
/* steps
step 1: x < y is false, !((x = 2) || (x = 3)) ? (x = 1): (y = 2)
step 2: "x = 2" is an assignment statement its always true,
        In logic or expression, as long as one statement is true the whole expression is true
        Take the reverse as false, so z = (y = 2), x is 2
step 3: x is 2, y is 2, z is 2
*/
```

The Preprocessor & Typedef

Guidance

- Chapter: Pointers-On-C.pdf chapter 14; C-Primer-Plus.pdf p653 typedef: A

Quick Look; The-C-Programming-Language-2nd.pdf chapter 6.7

- Spend time: 1 day
- Learn suggestion: Thinking from the perspective of the compiler
- Key points
  - What is the difference between macro definition and typedef?
  - What is the difference between macro definition functions and general functions?
  - Is the typedef used to create a new type?
  - How do we use conditional compilation to improve code portability?
  - Which steps do the macro definition and typedef work on in the compiling process?

Practice

1. How to define a macro function? What are the points that need to pay attention?

```
Syntax:#define MACRO_NAME(parameters list) macro_body
```

- Do not forget to put every expression enclosed in parenthesis

```
#define S(a,b) a*b
int value = S(3 + 1, 3 + 4);
printf("%d %d %d\n", value);
```

- Avoid to use the operator that may cause side effect

```
#define SQUARE(a) (a)*(a)
int main(void)
{
    int a = 1;

    printf("%d\n", SQUARE(a++) + SQUARE(++a));
    return 0;
}
```

- The character "\" and do/while can be used for multi-line macros function

```
#define RT_LOOKUP_ACTIVE_LIST(vrfid, rth, protocol) \
    do{\
        struct _rt_list *list; \
        struct _rt_entry * t_rt; \
        rt_open(1); \
        list = rthlist_active(vrfid, TRUE); \
        RT_LIST(rth, list, rt_head) { \
            t_rt = rth->rth_active; \
            if(protocol != RTPROTO_ANY && protocol != t_rt->rt_proto) { \
                continue; \
            } \
        } RT_LIST_END(rth, list, rt_head); \
        if (list) { \
            RTLIST_RESET(list); \
        } \
        rt_close(1); \
    }while(0);
```

- Do not end with semicolon, it's not a statement
- Macro definition don't do any syntax check

```c
#define MIN(x,y) ((x)<(y)?(x):(y))

int main(void)
{
    MIN(int, char);
    return 0;
}
```

2. Write a macro MIN that accepts 3 parameters and returns the smallest one

There are 3 kinds of method

```c
#define MIN0(x,y) ((x)<(y)?(x):(y))
#define MIN1(x,y,z) (MIN(x,y)<(z)?MIN(x,y):(z))
#define MIN2(x,y,z) MIN(x, MIN(y, z))

#define MIN3(x,y,z) ((x)<(y)?((x)<(z)?(x):(z)):((y)<(z)?(y):(z)))
```

3. There is a debug function that has no output when the macro DEBUG is not defined. When the macro DEBUG is defined, debug outputs the file name, function name, line number, and content. The call and output are given.

```c
int main(void)
{
    debug("The value of 2 + 3 is %d\n", 2 + 3);
    return 0;
}
```

```
output: test.c, main, 3: The value of 2 + 3 is 5
```

a. Please write the macro definition

```c
#ifdef DEBUG
 #define debug(fmt, args...) printf("%s, %s, %d: " fmt , __FILE__, __FUNCTION__, __LINE__,
##args)
#else
 #define debug(fmt, args...)
#endif
```

b. How to open the macro by command line?

then the value for `ARRAY_SIZE` can be given on the command line when the program is compiled.

On UNK compilers, the `-D` option does this job. There are two ways to use this option.

```
-Dname
-Dname=stuff
```

The first form defines the symbol *name* to have the value one. The second form defines the symbol's value to be the *stuff* after the equal sign. The Borland C compilers for MS-DOS provide the same capability with the same syntax. Consult your compiler's documentation for information about your system.

To return to our example, the command line to compile this program on a UNIX system might look like this:

```
cc -DARRAY_SIZE=100 prog.c
```

4. What is the difference between macro definition and typedef?

	Macro Definition	Typedef
Essence	text replacement	type alias
Timing	pre-processing stage	compile stage
Mode	replace text only	a part of compile, including syntax check
Function	Used to define constants or long string replace	Simplify complex declarations

5. Both macro definition and typedef can help simplify code and improve its readability. However, sometimes they are different. What's the difference between them for the variables in the following program?

```c
#define U_INT32_1 unsigned int*
typedef unsigned int* U_INT32_2;

int main()
{
    U_INT32_1 a, b;
    U_INT32_2 c, d;

    return 0;
}
```

```c
//extended form, It can be check by command ptype under gdb(gcc debuging tool)
unsigned int *a, b;
unsigned int *c, *d;
```

6. When do we need to use typedef? Illustration

- Used for pointer to improve program readability

```c
// pointer to function
typedef int (*func_t)(int a, int b);
int sum (int a, int b)
{
  return a + b;
}
int main (void)
{
  func_t fp = sum;
  printf ("%d\n", fp(1,2));
  return 0;
}
```

- Improve portability of program

```c
#ifdef PIC_16
    typedef unsigned long U32
#else
    typedef unsigned int U32
#endif
```

- Simplify complex definitions

```
int *(*array[10])(int *p, int len, char name[]);

typedef int *(*func_ptr_t)(int *p, int len, char name[]);
func_ptr_t array[10];
```

- Used to struct

```
struct student
{
  char name[20];
  int  age;
};
typedef struct student student_t;

struct student stu1;
student_t stu2;
```

- Used to array

```
typedef int array_t[10];
array_t array;

int main (void)
{
  array[9] = 100;
  printf ("array[9] = %d\n", array[9]);
  return 0;
}
```

- Used to enumeration

```
enum color
{
  red,
  white,
  black,
  green,
  color_num,
};
typedef enum color color_t;

enum color color1 = black;
color_t    color2 = black;
```

7. How to use conditional compilation to improve code portability?

Here is an example of conditional compilation from Linux about how to define machine width.

```
199     /* Host-dependent types and defines for user-space ACPICA */
200
201     #define ACPI_FLUSH_CPU_CACHE()
202     #define ACPI_CAST_PTHREAD_T(pthread) ((acpi_thread_id) (pthread))
203
204     #if defined(__ia64__)     || (defined(__x86_64__) && !defined(__ILP32__)) ||\
205             defined(__aarch64__) || defined(__PPC64__) ||\
206             defined(__s390x__)   || defined(__loongarch__) ||\
207             (defined(__riscv) && (defined(__LP64__) || defined(_LP64)))
208     #define ACPI_MACHINE_WIDTH          64
209     #define COMPILER_DEPENDENT_INT64    long
210     #define COMPILER_DEPENDENT_UINT64   unsigned long
211     #else
212     #define ACPI_MACHINE_WIDTH          32
213     #define COMPILER_DEPENDENT_INT64    long long
214     #define COMPILER_DEPENDENT_UINT64   unsigned long long
215     #define ACPI_USE_NATIVE_DIVIDE
216     #define ACPI_USE_NATIVE_MATH64
217     #endif
```

Control Flow

Guidance

- Chapter: Pointers-On-C.pdf Chapter 4; The-C-Programming-Language-2nd.pdf 3
- Spend time: 1 day
- Learn suggestion: Don't arbitrarily use the goto statement if you want to keep the program structure clear.
- Key points
  - What's the fall-through in the case statement?
  - What's the dangling in the else statement?
  - Why do we need control statements?
  - What is the execution order of the 3 parts of the for statement?
  - How can you convert between while and for?

Practice

1. What is the function of control statements? What form can a control expression be?

A control statement is a statement used to control the selection, loop, turn, and return of a program flow. Control expression can be any form

```c
int fun(void){return 1;}

int main(void)
{
    int a, b, c;

    if (2 + 3);
    if (a % b);
    if (c = a / b);
    if (a = 1, b = 2);
    if (a > b? a : b);
    if (fun());
    if (a = fun());
    if (a == fun());
    if (a | fun());
    if (a || fun());

    return 0;
}
```

All used by if can also be used to while/for/switch/return.

2. How many kinds of selection statements are there? What are their application scenarios?

- It is simpler to use if when the value of an expression is range and to use switch when the value of an expression is definite

For example: A class will be graded according to the requirements of 0-59: E-level 60-69: D-level 70-79: C-level 80-89: B-level 90-100: A-level

```c
if (score >=0 && score < 60)
    ;
else if (score >=60 && score < 69)
    ;
else if (score >= 70 && score < 79)
    ;
else if (score >= 80 && score < 89)
    ;
else if (score >= 90 && score <= 100)
    ;
else
    ;

/* Need to find a way to change
the range to a definite value */
switch(score / 10)
{
    case 10:
    case 9:
        break;
    case 8:
        break;
    case 7:
        break;
    case 6:
        break;
    default:
        break;
}
```

- It is easy to use “if” in multiple conditions
- The control expression of if use logical expression and switch use arithmetic expression. Because the result of a logical expression is 0 or non-0

```
if (condition_1)
    ;
else
    ;

switch(condition_1)
{
    case v1:
        break;
    default:
        break;
}
```

- Some compilers optimize the switch statement to make instruction execution more efficient.(Need more research)

3. What should we pay attention to in the switch statement?

- The value of the case must be an inter
- The case label must be unique
- Do not forget to add a break after the case statement otherwise a fall-through will happen
- Do not forget to add a default statement ending with all of the case, which can help you catch the case that you thoughtless
- The Statements aren't put in a case statement will not be executed

```
int main(void)
{
    int a = 1, b = 1;

    switch(a)
    {
        b = 2;
        case 1:
            printf("111\n");
            break;
        case 2:
            printf("222\n");
            break;
        default:
            printf("333\n");
            break;
        b = 3;
    }

    printf("%d\n", b);        // 1
    return 0;
}
```

4. What is dangling-else? Illustration

The rule of an else match with it is that it will match with the if closest to it. Therefore, Something will happen that is out of our expectations if we write if without braces

```
// original intent:
if A is true then:
    if B is ture then:
        statement1
    else
        nothing to do
else
    statement2

// The code out of our expect
if (A)
    if (B)
        statement1;
    else
        statement2;
if A is true then:
    if B is ture:
        statement1
    else
        statement2
else
    nothing to do

// Correct code
if (A)
{
    if (B)
        statement1;
}
else
    statement2;
```

5. Pointers-On-C.pdf 4.13 question 4

There are 2 approaches can be taken in this case

- empty statements
- get logical not of the control expression

```
// emply statements
if (condition1)
    ;
else
    statement1

// logical not, recommand
if (!condition1)
    statement1
```

6. How many kinds of loop statements are there? What are their application scenarios?

There are 3 types of loop statements in C.

There isn’t any difference between for and while; they only look different in form and can be converted to each other.

In a particular case, they are better used in the following situations

	Syntax	Peculiarity

for	for(init; condition; change condition)
	    statement
		init; condition; change condition,When these 3 parts are necessary together, using for makes the code look more intuitive
		
		for(i = 0; i < 3; i++)
		    statement

while	init;
	while (conditiohen)
	{
	    statement
	    change condition
	}
		When we do not need all the 3 parts appear at the same time, using while makes the code look more intuitive
		while(q = (singly linked list)->next); // point p to the end of singly linked list

The only difference between the statement while and do while is the block code of do while will be executed at least once

7. What is the difference between break, continue and return when we use them in a nested loop? Illustration

Suppose, they all run in the innermost loop

- continue, skip this time loop of the innermost loop

```
for (i = 0; i < 3; i++)
{
    for (j = 0; j < 3; j++)
    {
        if (j == 1)
            continue;
        // skip condition j == 1, j == 2 will be executed
    }
}

```

- break, skip out of the innermost loop to the second innermost loop

```
for (i = 0; i < 3; i++)
{
    for (j = 0; j < 3; j++)
    {
        if (j == 1)
            break;
    }
    /*the innermost loop will be terminate
      when condition j == 1 meet*/
}
```

- return, skip out of all loop, there is a side effect to use return in nested. It causes the function to exit

```c
void fun(void)
{
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            if (j == 1)
                return;
        }
    }
}
/*the functioin will be terminate
  when condition j == 1 meet*/
```

8. Pointers-On-C.pdf 5.8 question 8

```c
// redundant code
a = f1(x);
b = f2(x + a);

for(c = f3(a, b); c > 0; c = f3(a, b))
{
    statements
    a = f1(++x);
    b = f2(x + a);
}

// simplified code
while (a = f1(x), b = f2(x + a), (c = f3(a, b)) > 0)
{
    statements
    ++x;
}
```

9. What are the advantages and disadvantages of the goto statement?

Advantages: Make the code flexibly jump within the function

Disadvantages: The goto statement breaks the C code execution structure, reducing the readability and maintainability of the code greatly. And affect the execution efficiency of machine instructions

```c
// If the function is large, there are more than three lable functions,
// which makes the code difficult to read.
void fun (void)
{
    int i = 1;
AAA:
    statements

    if (condition_1)
        goto AAA;

BBB:
    if (condition_2)
        goto CCC;

CCC:
    if (condition_3)
        goto BBB;
    return;
}
```

- The first situation in which the use of goto is recommended. When a function exits and needs to do some work on resource cleaning classes and Don't break the code structure, using goto makes the code leaner

```c
void fun(void)
{
    if (condition_1)
    {
        // error occurs,need to close file/socket and release memory before exit function
        goto Resource_cleaning;
    }

    if (condition_2)
    {
        // error occurs,need to close file/socket and release memory before exit function
        goto Resource_cleaning;
    }

    if (condition_3)
    {
        // error occurs,need to close file/socket and release memory before exit function
        goto Resource_cleaning;
    }

Resource_cleaning:
    // close file/socket and release memory
}
```

- The second situation in which the use of goto is recommended. Goto can be used to help code jump out of nested loop.

```c
void fun(void)
{
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
        {
            if (j == 1)
                goto finish_loop;
        }
    }
finish_loop:
}
```

Input & Output

Guidance

- Chapter: Pointers-On-C.pdf 15; The-C-Programming-Language-2nd.pdf 7
- Spend Time: 1 day
- Learn Suggestion: Try to build a strong awareness of buffer
- Key Points
    - What are the standard input/output/error?
    - What's the input/output buffer? Why do we need it?

- How to format string/digital when Input/output? Figure 15.1-15.2, Table 15.3-15.7 of Pointers-On-C.pdf(Important)
- The basic operation of file operation

Practice

1. Why do we need to know the output buffer? How many ways to flush the output buffer?

The output buffer is used to temporarily store output data to prevent frequent generate interruptions and improve cpu efficiency. But there is one caveat when using output buffer. If it isn't refreshed before some fatal errors. You may not get the output.

There are several ways to flush the output buffer

- The buffer will be flushed when the process is exit

```c
int main(void)
{
    printf("Hello BDCOM");
    return 0;
}
```

- The buffer can be flushed by a newline character

```c
int main(void)
{
    printf("Hello \n BDCOM");
    pause();
    return 0;
}
```

- The buffer can be flushed by the library function fflush

```c
int main(void)
{
    printf("Hello BDCOM");
    fflush(stdout);
    pause();
    return 0;
}
```

2. Which library functions may cause overflow in Table 15.1 of Pointers-On-C.pdf? How can we prevent it from occurring?

```c
int main(void)
{
    char buf[8] = {0};

    scanf("%8s", buf);    // using format to prevent
    puts(buf);

    gets(buf);            // fgets can be used to replace it
    puts(buf);
    return 0;
}
```

3. Get the output based on the given data

```c
#define BUF_LEN 128

int main(void)
{
    unsigned int sip = 0, dip = 0;
    int len = 0, sport = 0, dport = 0;
    char buf[BUF_LEN] = {0};

    sip = 0x01010101;
    dip = 0x02020202;
    len = 44;
    sport = 23;
    dst = 54177;

    return 0;
}
```

Requirements

- Get the time by command date. Hint,Using the knowledge what we learned in this chapter to redirect
- Put all the output data into the variable buf
- Show as below

```
[xiaohei@localhost share]$ gcc test.c && ./a.out
Fri Feb 14 02:16:09 EST 2025, IP src=1.1.1.1, dst=2.2.2.2, len=44, TCP sport=23, dport=54177
```

```c
int main(void)
{
    unsigned int sip = 0, dip = 0;
    int len = 0, gataway = 0, sport = 0, dport = 0, tell_len = 0;
    FILE *fd_sto2fd = NULL, *fd_sto2nor = NULL;
    struct in_addr addr;
    char buf[BUF_LEN] = {0}, *pb = buf;
    char sip_str[INET_ADDRSTRLEN] = {0}, dip_str[INET_ADDRSTRLEN] = {0};

    sip = 0x01010101;
    dip = 0x02020202;
    len = 44;
    sport = 23;
    dport = 54177;

    if((fd_sto2fd = freopen("output.txt", "w+", stdout)) == NULL)
    {
        perror("FD stdout to file opened failure\n");
        return;
    }

    system("date");
    tell_len = ftell(fd_sto2fd);
    rewind(fd_sto2fd);
    fgets(buf, BUF_LEN, fd_sto2fd);

    if((fd_sto2nor = freopen("/dev/tty", "w", stdout)) == NULL)
    {
        fclose(fd_sto2fd);
        perror("FD stdout recover opened failure\n");
        return;
    }

    addr.s_addr = sip;
    inet_ntop(AF_INET, &addr, sip_str, INET_ADDRSTRLEN);
    addr.s_addr = dip;
    inet_ntop(AF_INET, &addr, dip_str, INET_ADDRSTRLEN);

    pb += tell_len - 1;
    pb += sprintf(pb, ", ");
    pb += sprintf(pb, "IP src=%s, dst=%s, len=%d, ", sip_str, dip_str, len);
    pb += sprintf(pb, "TCP sport=%d, dport=%d", sport, dport);

    puts(buf);

    fclose(fd_sto2fd);
    fclose(fd_sto2nor);
```

4. There is a file whose name is input.txt; it is calculated that the file length is n. Copy the part n/2-n to file output.txt. As efficient as possible

Suppose the file length is n, copy n/2-n to file output.txt.

```c
int main(int argc, char *argv[], char *envp[])
{
    int fsize;
    FILE *fr = NULL, *fw = NULL;

    fr = fopen("input.txt", "r");
    fw = fopen("output.txt", "w");

    if (fr == NULL || fw == NULL)
        return -1;

    fseek(fr, 0, SEEK_END);
    fsize = ftell(fr);
    fseek(fr, fsize / 2, SEEK_SET);
    /* copy the content from fr to fw */

    fclose(fr);
    fclose(fw);

    return 0;
}
```

5. List out all options of the fopen that you know and their descriptions

mode	read/write	position indicator	What happens when the file exists	What happens when the file doesn't exists
r				
r+				
w				
w+				
a				
a+				

mode	read/write	position indicator	What happens when the file exists	What happens when the file doesn't exists
r	read only	beginning of the file	read from beginning of the file	return null
r+	read/write	beginning of the file	read or write from/to beginning of the file	return null
w	write only	beginning of the file	truncated the file then write to the beginning of it	create the file
w+	read/write	beginning of the file	truncated the file then read or write from/to the beginning of it	create the file
a	write only	end of the file	write to the end of the file	create the file
a+	read/write	end of the file	read or write from/to the end of the file	create the file

Topic 3: Representation Of Data

Byte Order

Guidance

- Chapter: Computer-Systems-A-Programmers-Perspective.pdf 2.1.3 2.1.4;https://teaching.idallen.com/cst8281/10w/notes/110_byte_order_endian.html
- Spend Time: 1 day
- Learn Suggestion: Do the experiments in both little/big endian environments
- Key Points
  - How is the data stored in different byte order platform?
  - What's the relationship between big-endian/little-endian/host order/network order?
  - How does the string storage on different byte order

Practice

1. What are the rules of big-endian and little-endian? Illustration

Answer: In big endian system the Most Significant Byte (MSB) of a word is stored at the smallest memory address and the Least Significant Byte (LSB) is stored at the highest memory address.

In little endian system the Least Significant Byte (LSB) of a word is stored at the smallest memory address and the Most Significant Byte (MSB) is stored at the highest memory address.

Example: Consider a integer number **0x12345678** (hexadecimal) and it requires 4 byte to represent an integer, So

Big endian	Address	0x1000	0x1001	0x1002	0x1003
	Value	12	34	56	78

Little endian	Address	0x1000	0x1001	0x1002	0x1003
	Value	78	56	34	12

2. How many ways to determine the byte order of your PC? Illustration

```
union
{
    unsigned int a;
    unsigned char b[4];
} u;

int main()
{
    unsigned int a = 0x12345678;
    unsigned char *p = NULL;
    int i;

    p = (unsigned char *)&a;

    // Print all content from low address to high addres.
    // If the content is 78563412 is little endian. if it is 12345678 is big endian
    for (i = 0; i < sizeof(a); i++, p++)
        printf("%p %x\n", p, *p);

    // Truncate 4 bytes to 1 byte the lowest byte is reserved.
    // if it is 78 is little endian. if it is 12 is big endian
    printf("%x\n", *p);

    // Get the lowest byte of a. if it is 78 is little endian. if it is 12 is big endian
    u.a = 0x12345678;
    printf("%x\n", u.b[0]);

    // Get the hightest byte of a. if it is 12 is little endian. if it is 78 is big endian
    printf("%x\n", a >> 24);

    // If equal, then big-endian; since the function htonl does nothing in big-endian.
    htonl(a) == a;
    return 0;
}
```

3. Computer-Systems-A-Programmers-Perspective.pdf Problem 2.6

```c
int number_of_zero(unsigned int m)
{
    int i, conter = 0;

    for (i = 0; i < 32; i++)
    {
        if (!(m & 0x1))
            conter++;

        m = m >> 1;
    }

    return conter;
}

int main()
{
    int i;
    unsigned int a = 0x0027C8F8, b = 0x4A1F23E0, max_num = 0, a_l, a_r, b_l, b_r;

    for (i = 1; i < 32; i++)
    {
        a_l = number_of_zero((a << i) ^ b);
        a_r = number_of_zero((a >> i) ^ b);
        b_l = number_of_zero((b << i) ^ a);
        b_r = number_of_zero((b >> i) ^ a);

        if (max_num < a_l)
        {
            max_num = a_l;
            printf("al %x %x %d %d\n", a << i, b, a_l, i);
        }
        else if (max_num < a_r)
        {
            max_num = a_r;
            printf("ar %x %x %d %d\n", a >> i, b, a_r, i);
        }
        else if (max_num < b_l)
        {
            max_num = b_l;
            printf("bl %x %x %d %d\n", a, b << i, b_l, i);
        }
        else if (max_num < b_r)
        {
            max_num = b_r;
            printf("br %x %x %d %d\n", a, b >> i, b_r, i);
        }
    }

    return 0;
}
```

4. Here is a bunch of data from the network(big-endian). Parse and print it following the format as the image blew to ensure it can run properly.

```
void main(void)
{
    //data order smac/dmac/sip/dip/sport/dport
    char raw_data[BUF_LEN] = {0x98, 0x45, 0x62, 0xd6, 0xa1, 0x6c, 0x20, 0x7b, 0xd2, 0x51,
0x19, 0x05, 0x01, 0x02, 0x03, 0x04, 0x11, 0x22, 0x33, 0x44, 0x12, 0x34, 0x56, 0x78};

    return;
}
```

```
[xiaohei@localhost share]$ gcc test.c && ./a.out
MAC src=98:45:62:d6:a1:6c, dst=20:7b:d2:51:19:05, IP src=1.2.3.4, dst=17.34.51.68, TCP sport=1234, dport=5678
```

```
Switch#os-demo test1
MAC src=98:45:62:d6:a1:6c, dst=20:7b:d2:51:19:05, IP src=1.2.3.4, dst=17.34.51.68, TCP sport=1234, dport=5678
```

```
#ifdef ROUTING_MODULE
#include <ip/in.h>
#endif

#define BUF_LEN     128
#define MAC_LEN     6
#define IP_LEN      4
#define PORT_LEN    2

char *os_demo_mac2str(unsigned char *mac)
{
    static char str[20] = {0};

    sprintf(str, "%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);

    return str;
}

void main(void)
{
    //data order smac/dmac/sip/dip/sport/dport
    char raw_data[BUF_LEN] = {0x98, 0x45, 0x62, 0xd6, 0xa1, 0x6c, 0x20, 0x7b, 0xd2, 0x51, 0x19, 0x05,
0x01, 0x02, 0x03, 0x04, 0x11, 0x22, 0x33, 0x44, 0x12, 0x34, 0x56, 0x78};
    char buf[BUF_LEN] = {0}, *pb = buf;
    char smac[MAC_LEN] = {0}, dmac[MAC_LEN] = {0};
    unsigned int sip = 0, dip = 0;
    unsigned short sport = 0, dport = 0;

    memcpy(smac, raw_data, MAC_LEN);
    memcpy(dmac, raw_data + MAC_LEN, MAC_LEN);
    memcpy(&sip, raw_data + MAC_LEN * 2, IP_LEN);
    memcpy(&dip, raw_data + MAC_LEN * 2 + IP_LEN, IP_LEN);
    memcpy(&sport, raw_data + MAC_LEN * 2  + IP_LEN * 2, PORT_LEN);
    memcpy(&dport, raw_data + MAC_LEN * 2  + IP_LEN * 2 + PORT_LEN, PORT_LEN);


    pb += sprintf(pb, "MAC src=%s", os_demo_mac2str(smac));
    pb += sprintf(pb, ", dst=%s", os_demo_mac2str(dmac));
    pb += sprintf(pb, ", IP src=%s", inet_ntoa(*(struct in_addr *)&sip));
    pb += sprintf(pb, ", dst=%s", inet_ntoa(*(struct in_addr *)&dip));
    pb += sprintf(pb, ", TCP sport=%x, dport=%x", ntohs(sport), ntohs(dport));

    printf("%s\n", buf);

    return;
}
```

Data Storage & Representation

Guidance

- Chapter: Pointers-On-C.pdf 3.1.1; Computer-Systems-A-Programmers-Perspective.pdf 2.1.2 2.2.1-3 2.2.4-7

- Spend Time: 1 day
- Learn Suggestion: To know exactly how the data is stored and represented in the memory.
- Key Points
  - What are the extension and truncation of data store?
  - What is the difference between implicit type conversion and explicit type conversion?
  - How does the computer store negative numbers?
  - What is the rule of 2's complement operation?

Practice

1. What is the difference between the following 2 values in memory? Are they the same or not? Why?

```
1  char a = -1;
2  unsigned char b = 255;
```

In memory, basically there is no difference between the given 2 value. Because, both the values are stored in memory as bit. We know that, character data type is 8 bit long (1 byte). In signed char, 7 bit is used for storing value and 1 bit is used for the sign. And for unsigned char, whole 8 bit is used to store the value.

For char a = -1, the representation of the value will be-

1 1 1 1 1 1 1 1

The leftmost bit is used to represent the sign of the value, which is negative. The rest of the bit is represent the value, which is the 2's complement of positive 1.

For unsigned char b = 255, the representation of the value is-

1 1 1 1 1 1 1 1

Since there is no sign bit, all the bit is represent a number, which is 255.
So, there is no difference between the given 2 value in memory.

2. Try to explain why the result of the following program is that way. What happened?

```
1  int main(void)
2  {
3      char a = 128;
4      unsigned char b = 256;
5
6      printf("%d %d\n", a, b);    // -128 0
7      return 0;
8  }
```

Char data type is 1 byte long. For signed char, 1 bit is used to represent the sign of the number, and other 7 bit is used to represent the value. So, the range of signed char is -128 to + 127. For char a = 128, if we represent it in binary form, then this will look like

1 0 0 0 0 0 0 0

As we know that, the leftmost bit is used to represent the sign bit and 1 is represent negative number, that's why the compiler assumes the number as negative number. The value of the number will be-

-2^7 + (0*2^6 + 0*2^5 + …… + 0*2^0) = -128

Using the following formula.

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

That's why the value a will be -128.

For unsigned char b = 256, there is no sign bit in unsigned char, all the bits are used to represent the value. If we convert the value 256 to binary, this will look like-

1	0 0 0 0 0 0 0 0

This requires 9 bit to represent 256. We know that char is 8 bit long. So, the leftmost bit (9th bit) will be discarded. After discarding the 9th bit, the value will be - 00000000, which decimal equivalent number is 0. That's why the value of second variable is 0.

3. What is an extension when storing a number? When does it occur?

When the length of the converted original type < the length of the destination type, the destination storage space cannot be filled, then it needs to be extended. There are two cases of extension

a) For signed types, the highest bit copy needs to be filled in the remaining space, which is called symbol extension.

```c
int main()
{
    char a = 2, b = -2;

    printf("%x %x\n", a & 0Xff, (short)a & 0Xffff);  // 2 2
    printf("%x %x\n", b & 0Xff, (short)b & 0Xffff);  // fe fffe
    return 0;
}
```

b) For unsigned types, fill the remaining space with 0, which is called 0 extension

```c
int main()
{
    unsigned char c = 2, d = -2;

    printf("%x %x\n", c & 0Xff, (short)c & 0Xffff);         // 2 2
    printf("%x %x\n", d & 0Xff, (short)d & 0Xffff);         // fe fe
    return 0;
}
```

4. What's an integer promotion? What's the rule of it? What scenarios does it occur?

Integer Promotion：In an expression, Any value range of the original type that can be represented by type int is raised to int. else raised to unsigned int

long long->long->int

In an arithmetic operation, as long as the type of one side is the above type, the other side is also promoted to this type. If not any side is of any of the above types, it will be promoted to int according to the principle of integer promotion.

To know more, search for "Integer Conversion Rank" on the google

- Default Promotion

  The default type is an integer to store a data even though it's only a character.

```c
printf("%c", 'A') // the 'A' is actually raised to int before being passed to printf
```

- Usual Arithmetic Conversion

  promote the type of c1,c2 and c3 to int. then do arithmetic

```c
int main()
{
    unsigned char c1 = 255, c2 = 2, c3 = 3, n;
    n = c1 + c2 - c3;
    printf("%d\n", n);    // 254
    return 0;
}
```

- Function calling procedure

```c
unsigned int test_fun(int a)    /* Formal and actual parameter are different types */
{
    return a;                   /* return value and type are different types */
}

int main()
{
    char a = -1;

    test_fun(a);
    return 0;
}
```

5. What happened in the following program when the argument length is 0? Why?

```c
unsigned int sum_elements(unsigned int a[], unsigned int length)
{
    int i;
    unsigned int result = 0;

    for(i = 0; i <= length - 1; i++)
        result += a[i];

    return result;
}
```

The process is crashed

- Why did the process crash?

Invalid memory space may be accessed due to an infinite loop causing the array to overflow.

- Why does the infinite loop happen?

i <= length - 1,The unsigned int type participates in this expression, and according to the rule of integer promotion, all data type become unsigned int. When length is 0， 0-1 equal to -1， It is 0xffffffff in unsigned int. The range of i is 0-0xffffffff.
Therefore the expression i <= length - 1 is true forever

Bitwise Operation

Guidance

- Chapter: Computer-Systems-A-Programmers-Perspective.pdf 2.1.6-2.1.9;C-Primer-Plus.pdf chapter 15 C’ s Bitwise Operators
- Spend Time: 1 day
- Learn Suggestion: Build awareness to save the memory in the embedded system as much as possible.
- Key Points
  - What's the difference between logical shift and arithmetic shift?
  - What's the difference between bitwise and logical AND/OR/NOT?
  - The truth table for Bitwise AND/OR/NOT/XOR

Practice

1. Pointers-On-C.pdf 5.8 question 12

There are 2 kinds of ways used for a right shift in C, arithmetic or logical shift. The rules are as following

unsigned	logical shift	fill the left always end with 0
singed	arithmetic shift	fill the left end with repetitions of the most significant bit

More detail to reference Computer-Systems-A-Programmers-Perspective.pdf -> 2.1.10

For the current question, we can determine to use the program as following

```
int a = -1;

printf("%s\n", a == (a >> 3) ? "arithmetic" : "logical");
```

2. Computer-Systems-A-Programmers-Perspective.pdf Practice Problem 2.16

a		a << 2	
Hex	Binary	Binary	Hex
0xD4	1101 0100	0101 0000	0x50
0x64	0110 0100	1001 0000	0x90
0x72	0111 0010	1100 1000	0xC8
0x44	0100 0100	0001 0000	0x10

Logical a >> 3		Arithmetic a >> 3	
Hex	**Binary**	**Binary**	**Hex**
0x1A	0001 1010	1111 1010	0xFA
0x0C	0000 1100	0000 1100	0x0C
0x0E	0000 1110	0000 1110	0x0E
0x08	0000 1000	0000 1000	0x08

3. Computer-Systems-A-Programmers-Perspective.pdf Practice Problem 2.13

```c
#include<stdio.h>
int bis(int x, int m);
int bic(int x, int m);

int bool_or(int x, int y){
    int result = bis(x, y);
    return result;
}
int bool_xor(int x, int y){
    int tmp1 = bic(x, y);
    int tmp2 = bic(y, x);
    int result = bis(tmp1, tmp2);
    return result;
}
int main()
{
    int x, y;
    x = 0x1234ffff;
    y = 0xffffedcb;
    printf("Logical XOR : %X\nLogical OR : %X\n", bool_xor(x, y), bool_or(x, y));
    return 0;
}

int bis(int x, int m){
    x = x | m;
    return x;
}
int bic(int x, int m){
    x = x & (~m);
    return x;
}
```

4. Pointers-On-C.pdf 5.9 question 4

```c
#define ARRAY_SIZE 80
static unsigned char_index, bit_position;
typedef enum{false = 0, true} bool;

bool get_idx_pos(unsigned bit_number)
{
    if (bit_number >= ARRAY_SIZE)
        return false;

    char_index = bit_number / 8;
    bit_position = bit_number % 8;
    return true;
}

void set_bit(char bit_array[], unsigned bit_number)
{
    if (!get_idx_pos(bit_number))
    return;

    bit_array[char_index] |= (1 << bit_position);
    return;
}

void clear_bit(char bit_array[], unsigned bit_number)
{
    if (!get_idx_pos(bit_number))
    return;

    bit_array[char_index] &= ~(1 << bit_position);
    return;
}

void assign_bit(char bit_array[], unsigned bit_number, int value)
{
    if (value)
        set_bit(bit_array, bit_number);
    else
        clear_bit(bit_array, bit_number);
    return;
}

int test_bit(char bit_array[], unsigned bit_number)
{
    if (!get_idx_pos(bit_number))
    return;

    return (bit_array[char_index] >> bit_position) & 1;
}
```

5. Pointers-On-C.pdf 5.9 question 5

```c
int store_bit_field(int original_value, int value_to_store,
    unsigned starting_bit, unsigned ending_bit)
{
    unsigned int mask = 0;

    mask = ((1 << (starting_bit - ending_bit + 1)) - 1) << ending_bit;  // step 1

    original_value &= ~mask;                                           // step 2

    value_to_store <<= ending_bit;                                     // step 3

    value_to_store &= mask;                                            // step 4

    original_value |= value_to_store;                                  // step 5

    return original_value;
}
```

Topic 4: Compound Type

Array

Guidance

- Chapter: C-Primer-Plus.pdf Chapter 10(Skip pointers section); Pointers-On-C.pdf chapter 8.2.1-8.2.3
- Spend Time: 1 day
- Learn Suggestion: Learning from the perspective of the memory
- Key Points
    - What is compound type? Why do we need compound type?
    - How is an array stored in memory? What about a multi-dimensional array?
    - What scenario does an overflow may occur? What might it lead to?

Practice

1. What is the physical structure of the array? How is the array stored in memory? Why is the index of the first element of the array [0] instead of [1]?

The index is the offset of the start address of the element in the array relative to the start address of the array

An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations. Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.
The value of array is stored in memory one after another. There is no gap or any empty space between two other elements of the same array.
Suppose we've an array

```
int arr[5] = {1, 2, 3, 4, 5}
```

The value will stored in the memory as the following-

1000	1004	1008	100C	1010
1	2	3	4	5

So, the value will stored in the consecutive address in the memory.
The array_name arr is the address of first element of the array or address of the 0^{th} index element of the array. So, address of next element in the array is arr+1 and further address will be arr + 2 and so on.
So, arr + i mean the address at i distance away from the starting element of the array.
Since the first element is at 0 distance away from the starting element of the array. that's why indexing of array starts from 0. offset

2. What happens if an array overflows? Illustration

It will overwrite the data of the neighbor when the array overflow occurs. resulting in incalculable consequences.
If the array is stored on the stack, it may destroy the key information on the stack, leading to a function call stack mess. For example, the return address may be overwritten.
If the array is stored in a global area, it may destroy the global variable table, leading to more serious effects.

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

#define NUM 5
int main(void)
{
    int a = 11;
    int arr[NUM] = {1, 2, 3, 4, 5};
    int b = 22;
    int i;

    printf("The address of a      is %p\n", &a);

    for (i = NUM - 1; i >= 0; i--)
        printf("The address of arr[%d] is %p\n", i, &arr[i]);

    printf("The address of b      is %p\n", &b);

    printf("\nThe address of arr[%d] is %p\n", NUM, &arr[NUM]);

    return 0;
}
```

```
Terminal ×  +

root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c -m32 && ./a.out
The address of a      is 0xffcf12f8
The address of arr[4] is 0xffcf12f4
The address of arr[3] is 0xffcf12f0
The address of arr[2] is 0xffcf12ec
The address of arr[1] is 0xffcf12e8
The address of arr[0] is 0xffcf12e4
The address of b      is 0xffcf12e0

The address of arr[5] is 0xffcf12f8
root@dcs-6b203f0c-0:/workspace/CProject#
```

3. What can you learn from the following program?

```c
int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    printf("%x\n", array[1]);
    printf("%x\n", array[0]);
    printf("%x\n", array[-1]);

    printf("%x\n", array[4]);
    printf("%x\n", array[5]);
    return 0;
}
```

We can find that the C compiler did not have any restrictions for the array index; you can access the array by any index. Because the designers of C thought that C programmers knew exactly what they were doing. There is no need to restrict him in this regard. So, as a C programmer, we need to know exactly what we are doing now

4. Briefly describe the storage and access of multi-dimensional arrays

A multidimensional array stores data in a row-major order in C. Based on this, there is no difference in memory between a one-dimensional and multidimensional array when they have the same number of elements.

The following program's multidimensional array can also be initialized using one-dimensional arrays. Multidimensional arrays are just a form of data representation that exists only to facilitate developers' better implementation of their business.

```c
int a[18] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};
int b[3][6] = {{1, 2, 3, 4, 5, 6}, {7, 8, 9, 10, 11, 12}, {13, 14, 15, 16, 17, 18}};
int c[3][2][3] = {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}}, {{13, 14, 15}, {16, 17, 18}}};

int bb[3][6] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};
int cc[3][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};

printf("%d %d %d %d %d\n", a[7], b[1][1], c[1][0][1], bb[1][1], cc[1][0][1]);   // 8 8 8 8 8
```

.

Here is an explanation of row-major order and column-major order

Answer: Multidimensional arrays are stored in memory in contiguous block, but the arrangement of elements differs. It can be -

i. **Row major order:** elements are stored row by row

ii. **Column major order:** elements are stored column by column

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

In row major it will be stored liked this

1	2	3	4	5	6

In column major it will be

1	4	2	5	3	6

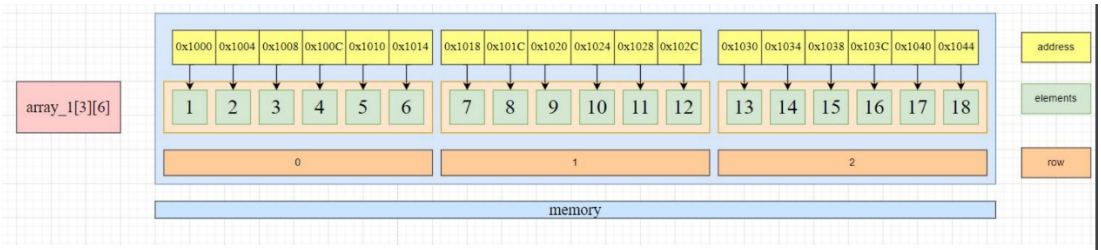
For accessing elements we have to provide indices for each dimension and a nested loop can traverse the multidimensional array

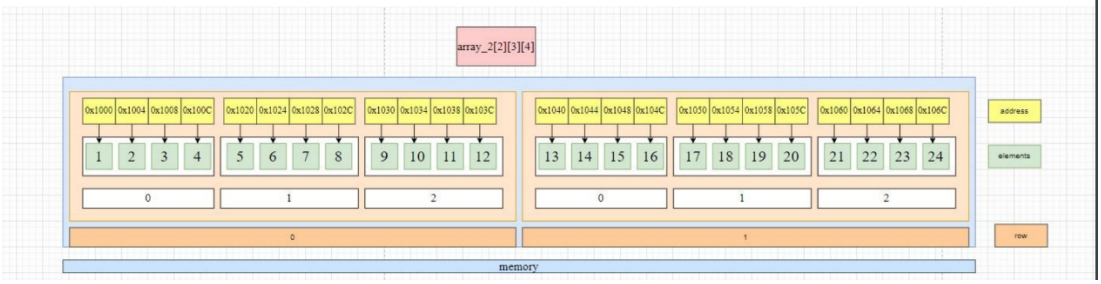
i. **Row major access:** address = base_address + (row_index * number_of_columns + column_index) * size_of_each_elements.

ii. **Column major access:** address = base_address + (column_index * number_of_rows + row_index) * size_of_each_elements.

5. Draw some images to demonstrate the memory structure of array1/array2 and the output in the following program. Which can help a person know well how the memory stores an array

```c
int main(void)
{
    int array1[3][6] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};
    int array2[2][3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24};

    printf("%d %d\n", array1[1][6], array1[0][15]);         // 13 16
    printf("%d %d\n", array2[0][3][4], array2[0][2][8]);    // 17 17

    return 0;
}
```





String

Guidance

- Chapter: Pointers-On-C.pdf chapter 9, 13.5
- Spend Time: 1 day
- Learn Suggestion: To know well, how the library functions that start with 'str' work in the header "string.h"
- Key Points
  - How to store a string in C? Is there a string type?
  - What is the difference between the c sizeof and strlen?
  - What is the meaning of 'n' in the library function strncat/strncmp and strncpy?

Practice

1. How many ways to store a string "Hello World!" in C? Where are they stored? Lists all the different memory storage ways in which this string can be stored

Refer to "String storage.c"

2. What is the difference between strcpy and memcpy? Illustration

- strcpy: Copy from source to destination character by character until 0 character is encountered
- memcpy: Copy a specified number of bytes of data from the source to the destination

```c
int main(void)
{
    char str_src[NUM] = "Hello \0 World!";
    char str_dest[NUM];
    int i;

    printf("%20s", "index");
    for(i = 0; i < NUM; i++)
        printf("%3d", i);
    printf("\n");

    printf("%20s", "raw data, src str");
    for(i = 0; i < NUM; i++)
        printf("%3c", str_src[i] == 0?' ': str_src[i]);
    printf("\n");

    memset(str_dest, 0, NUM);
    strcpy(str_dest, str_src);
    printf("%20s", "strcpy, dest str");
    for(i = 0; i < NUM; i++)
        printf("%3c", str_dest[i] == 0?' ': str_dest[i]);
    printf("\n");

    memset(str_dest, 0, NUM);
    strncpy(str_dest, str_src, strlen(str_src));
    printf("%20s", "strncpy, dest str");
    for(i = 0; i < NUM; i++)
        printf("%3c", str_dest[i] == 0?' ': str_dest[i]);
    printf("\n");

    memset(str_dest, 0, NUM);
    memcpy(str_dest, str_src, sizeof(str_src));
    printf("%20s", "memcpy, dest str");
    for(i = 0; i < NUM; i++)
        printf("%3c", str_dest[i] == 0?' ': str_dest[i]);
    printf("\n");

    return 0;
}
```

```
               index  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
 raw data, src str  H  e  l  l  o           W  o  r  l  d  !
  strcpy, dest str  H  e  l  l  o
 strncpy, dest str  H  e  l  l  o
  memcpy, dest str  H  e  l  l  o           W  o  r  l  d  !
```

3. Pointers-On-C.pdf 9.14 question 3

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

void my_strcpy(char *dest, const char *src, int dest_len) {
  int remaining_space = dest_len - 1;   // reserve 1 byte for NUL character
  int i = 0;

  // copying elements
  while (src[i] != '\0' && i < remaining_space) {
    dest[i] = src[i];
    i++;
  }

  // Adding NUL to the last position
  dest[i] = '\0';
}


int main() {
    char str1[] = "Hello World!";
    char str2[7] = "ab";

    my_strcpy(str2, str1, sizeof(str2));

    printf("str1: %s\n", str1); // Output: Hello World!
    printf("str2: %s\n", str2); // Output: Hello
}
```

4. Pointers-On-C.pdf 9.14 question 5

```c
#include<stdio.h>
void my_strncat(char *dest, char *src, int dest_len);
int main()
{
    char dest[10] = "Hello";
    char src[] = ", World";
    my_strncat(dest, src, sizeof(dest));
    printf("%s\n", dest);
}
void my_strncat(char *dest, char *src, int dest_len)
{
    int lend = strlen(dest);
    int lens = strlen(src);
    int i, j = 0;
    for(i=lend; i<dest_len && j<lens; i++){
        dest[i] = src[j++];
    }
    dest[i] = '\0';
}
```

The output is-

```
Hello, Wor

--------------------------------
Process exited after 0.04038 seconds with return value 0
Press any key to continue . . .
```

This only concatenate the equal size of the destination string. So, there is no chance to overflow the string.

5. Find the average

There are 8 people in the group, and the current C language test scores are 210, 327, 413, 57145, 9154, 163, 23172, and 4081. The current device is an 8-bit-based PC, the maximum size is 8 bits. Only uint8 is available. You are allowed to use arrays, structs, or other simple variables. Please provide the average score of these 8 people.

```c
void main(void)
{
    char *data[] = {"210", "327", "413", "57145", "9154", "163", "23172", "4081"};
    char result[COLUMN] = {0};
    char i, j, k, len, num;
    char *p;

    num = sizeof(data) / sizeof(char*);

    for(i = 0; i < num; i++)
    {
        len = strlen(data[i]) - 1;
        p = data[i] + len;

        for(j = COLUMN - 1, k = len; k >= 0; j--, k--, p--)
            result[j] += (*p - '0');
    }

    for (i = COLUMN - 1; i; i--)
    {
        result[i - 1] += (result[i] / 10);
        result[i] = (result[i] % 10);
    }

    printf("Sum:");
    for (i = 0; i < COLUMN; i++)
        printf("%d", result[i]);
    printf("\n");

    for (i = 0; i < COLUMN - 1; i++)
    {
        result[i + 1] += (result[i] % 8) * 10;
        result[i] = result[i] / 8;
    }

    result[i] = result[i] / 8;

    printf("Average:");
    for (i = 0; i < COLUMN; i++)
        printf("%d", result[i]);
    printf("\n");
```

Struct

Guidance

- Chapter: Pointers-On-C.pdf chapter 10; Align And BE_LE Endian.docx
- Spend Time: 2 days
- Learn Suggestion: Try to read some open-source codes. To learn how they are implemented.
- Key Points
    - Why do we need a struct? What is it used for?
    - How to use the bit field to save the memory? What's the difference is it in big/little endian device? How to write a program for a bit field that doesn't need to consider byte order?
    - What's the alignment? Why do we need it?

Practice

1. What is the difference between aa, bb and cc?

```
struct
{
    int a;
    int b;
} aa;

struct bb
{
    int a;
    int b;
};

typedef struct
{
    int a;
    int b;
} cc;
```

aa	bb	cc
aa is a variable name.	It’ s a tag of struct	It’ s a alias of the
Itis of type **struct { int a; int b;}**	type **struct { int a; int b;}**	struct **struct { int a; int b;}**
We can use it as a variable of the structure.	It can be used to create structure of the same type. **struct bb my_struct;** Here my_struct is a vaiable. My_struct is similar as aa.	It can also be used to create variables. **cc my_struct;** my_struct is a variable of cc type, where cc is a struct.

2. How many ways to initialize a struct? Illustration

```c
struct point
{
    int x;
    int y;
} p1 = {2, 3};

int main(void)
{
    struct point p2 = {2, 3};
    struct point p3 = {.x = 2, .y = 3};
    struct point p4 = {.y = 3};
    struct point p5 = {.y = 3, .x = 2};
    struct point p6;
    struct point p7 = p5;               // Do not recommended

    p6.x = 2;
    p6.y = 3;

    printf("%d %d\n", p7.x, p7.y);
    return 0;
}
```

3. What is the problem using a struct as a function argument or return value?

Answer: We know struct is a compound data type. It may consist of members of different types. The size of a struct is the sum of total number of size of each members requires plus some padding (if required). When we pass a struct as a function argument we have to push the struct in stack and then pass a copy. Also when we return a struct from function it has to push again in the stack frame of the caller. Copying an large struct variable is time consuming and not efficient. It may cause stack overflow issue. Another issue arises when we need to change one of the member of struct. But it is not possible when we pass by value. So in short pass by value-

- ❖ Not efficient. Cosumes more memory and create function overhead
- ❖ Can't modify original members

We can use pass by reference, as pointers requires less memory than a whole struct. It is also convenient and faster approach. But we have to pay attention whenever we do not need to modify the original values, as pass by reference may override original members unexpectedly.

4. What is the difference between struct and union? Why do we need union?

The difference between struct and union is shown below-

Features	Struct	Union
Memory Allocation	Allocates separate memory for all its member variables.	Allocates same memory location for all of the member variables.
Total Size	Sum of the size of all member variables.	Size of the largest member variable.
Data Storage	It can store store values for all the members at a time.	It can store value for only one member at a time.
Accessing Member	All members can be accessed at any time.	Only the last stored member can be accessed.
Modification impact	Modifying a member doesn't affect other members.	Modifying one member will overwrite other members.

Unions are particularly useful for scenarios where we want to store different types of information in the same space and we need only one types of data at a time. We can do this type of work using struct also, but using struct it will store all the information in different memory location, which is waste of memory.
So, when we have different type of information but only one type of data is needed at time. In this situation we can use union to save of memory usage.

5. How to use bit fields in struct? Why do we need it?

```c
#include<stdio.h>
struct BitField {
    unsigned int bolean : 1; // Limited to 0-1
    unsigned int age : 8; // Limited to 0-255
    unsigned int date : 5; // Limited to 0-31
};
int main()
{
    struct BitField test;
    test.bolean = 1;
    test.age = 150;
    test.date = 20;
    printf("In limit: %u %u %u\n", test.bolean, test.age, test.date);

    test.bolean = 3;
    test.age = 300;
    test.date = 35;
    printf("Out of limit : %u %u %u\n", test.bolean, test.age, test.date);
    return 0;
}
```

Here, I set the bit field for member boolean is 1, which can only capture two values (0 and 1). So, We can use this variable to represent boolean value.
The member variable age bit field is 8, which can only represent 0 to 255. This member variable will only use 8 bit instead of 4 bytes in memory.
The member variable date, which will be used to represent 1-31 in some situation. So, 5 bit is perfect for the member variable. If the value is greater than 31, then it'll be truncated.

We need bit field to save memory. Normally when we declare a variable with a specific data type, the compiler allocates specific size of memory for the variable.(1 byte for char, 4 bytes for int and so on). Sometimes we don't need that much of memory to represent our data. Suppose, we need to represent boolean value using integer data type. The compiler will allocates 4 bytes (32 bits) for the variable, but we need only 1 bit to do the task. In this situation bit field is very important, which can save lot of memory.
In embedded systems, the memory size is very limited. In this type of devices memory optimization is very important. We have to use bit field in this type of devices.

6. What is the output of the program on big endian and little-endian platform?

```c
#include <stdio.h>

struct abc
{
     unsigned char a:2;
     unsigned char b:3;
     unsigned char c:3;
};

int main(void)
{
    unsigned char aa;
    struct abc v;
    v.a = 2;
    v.b = 3;
    v.c = 6;

    memcpy(&aa, &v, sizeof(v));
    printf("%x\n", aa);
    return 0;
}
```

	8	7	6	5	4	3	2	1
little endian Reverse order as definition	c			b			a	
	1	1	0	0	1	1	1	0
	c				e			
big endian Same order as definition	a			b			c	
	1	0	0	1	1	1	1	0
	9				e			

7. What's the output in the following program? Try to analyze

```
struct AA
{
    short a;
    int b;
    char c;
};

struct BB
{
    short a;
    struct AA aa;
    char c;
};

#pragma pack(1)
struct CC
{
    short a;
    struct AA aa;
    char c;
};

struct AAA
{
    short a;
    int b;
    char c;
};

struct BBB
{
    struct AAA aaa;
    char c;
};
#pragma pack()

int main()
{
    printf("%d %d %d %d %d\n", sizeof(struct AA), sizeof(struct AAA),
    sizeof(struct BB), sizeof(struct BBB), sizeof(struct CC));
    return 0;
```

struct AA [aligned by 4 bytes]					
short a (2 byte)	0x1000			padding	padding
int b (4 byte)	0x1004				
char c (1 byte)	0x1008		padding	padding	padding
Total size: **12 byte**			Padding: 2 + 3 = **5 byte**		

struct BB [aligned by 4 bytes]					
short a (2 byte)	0x1000			padding	padding
struct AA aa (12 byte)	0x1004				
	0x1008				
	0x100C				
char c (1 byte)	0x1010		padding	padding	padding

Total size: **20 byte**	Padding: 2 + 3 = **5 byte**

struct CC [aligned by 1 bytes] (*#pargma used*)			
short a (2)		struct AA aa (12)	
		char c (1)	unused
Total size: **15 bytes**		**No padding**	

struct AAA [aligned by 1 bytes] (*#pargma used*)			
short a (2)		int b (4)	
		char c (1)	unused
Total size: **7 bytes**		**No padding**	

struct BBB [aligned by 1 bytes] (*#pargma used*)			
struct AAA aaa (7)			
			char c (1)
Total size: **8 bytes**		**No padding**	

8. Based on the struct of the previous question, What are the outputs in the following program? Why?

```c
int main(voida)
{
    struct AA aa;
    struct AAA aaa;
    struct BB bb;
    struct BBB bbb;
    struct CC cc;

    char array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    memcpy(&aa, array, sizeof(aa));
    memcpy(&aaa, array, sizeof(aaa));
    memcpy(&bb, array, sizeof(bb));
    memcpy(&bbb, array, sizeof(bbb));
    memcpy(&cc, array, sizeof(cc));

    printf("%d %d %d %d %d\n", aa.c, aaa.c, bb.c, bbb.c, cc.c);
    return 0;
}
```

structure	size	Copied values	Offset of c	Value of c
struct AA	12	address[0] = 1 to Address[11]=12	**8**	**9**
struct AAA	7	address[0] = 1 to Address[6]=7	**6**	**7**
struct BB	20	address[0] = 1 to Address[19]=20	**16**	**17**
struct BBB	8	address[0] = 1 to Address[7]=8	**7**	**8**
struct CC	15	address[0] = 1 to Address[14]=15	**14**	**15**

Topic 5: Pointers

Basic Knowledge

Guidance

- Chapter: Pointers-On-C.pdf chapter 6.1-6.8
- Spend Time: 1 day
- Learn Suggestion: Learn from the assembly language
- Key Points
    - What's a pointer? Why is the C using it?
    - What's the difference between the different type of pointer?
    - What's a wild/dangling pointer?

Practice

1. What can you learn from the following program? (From the perspective of variable content). Write the corresponding C code next to the assembly instruction

```
int main(void)
{
    int a = 1;
    int *p = &a;
    int b = 2;
    char *str = "BDCOM";
    int array[3] = {1, 2, 3};
    int *q = array;

    return 0;
}
/*
LC0:
    .string    "BDCOM"
    .text

main:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp
    movl    $1, -4(%ebp)
    leal    -4(%ebp), %eax
    movl    %eax, -8(%ebp)
    movl    $2, -12(%ebp)
    movl    $.LC0, -16(%ebp)
    movl    $1, -32(%ebp)
    movl    $2, -28(%ebp)
    movl    $3, -24(%ebp)

    leal    -32(%ebp), %eax
    movl    %eax, -20(%ebp)
    movl    $0, %eax
    leave
    ret
*/
```

The content of normal variables is numbers that resemble an immediate value

The content of a pointer variable is the address of a normal variable or another address

An array is an ungapped space whose elements are pushed onto the stack one by one. The pointer only needs to hold the first address of the array if you want to manipulate the array through the pointer

```
LC0:
    .string    "BDCOM"
    .text

main:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp        // Alloc 32 bytes for new stack
    movl    $1, -4(%ebp)     // int a = 1, using 1st 4 bytes of current stack to store immediate value 1
    leal    -4(%ebp), %eax   // Move the address of variable a to register %eax, prepare for next instructions
    movl    %eax, -8(%ebp)   // int *p = &a, using 2nd 4 btypes of current stack to store the address of variable a
    movl    $2, -12(%ebp)    // int b = 2,using 3rd 4 bytes of current stack to store immediate value 2
    movl    $.LC0, -16(%ebp) // char *str = "BDCOM", using 4th 4 bytes of current stack to store the first address of constant
    string "BDCOM"
    movl    $1, -32(%ebp)    // array[0] = 1, using 8th 4 bytes of current stack to store 1st number of array
    movl    $2, -28(%ebp)    // array[1] = 2, using 7th 4 bytes of current stack to store 2nd number of array
    movl    $3, -24(%ebp)    // array[2] = 3, using 6th 4 bytes of current stack to store 3rd number of array

    leal    -32(%ebp), %eax  // Move the first address of array to register %eax, prepare for next instructions
    movl    %eax, -20(%ebp)  // int *q = array, using 5th 4 tytes of current stack to store the first address of array
    movl    $0, %eax
    leave
    ret
```

2. There is a program, p,*p and &p, Answer following questions

```c
int main(void)
{
    // Suppose the address of variable a is 0x1000, p is 0xFFC.
    int a = 1;
    int *p = &a;

    return 0;
}
```

a. Short comparison between p, *p, &p?

variable	a	p
address	0x1000	0xFFC
value	1	0x1000

p	A variable of type int *. Store address of an int variable. Here stores the address of variable a.
*p	It is called indirection. It is used to get the value of the address it stores. Here is 1.
&p	It is an address of the variable p.

b. Which can be used as the l-value, and which can not in *p, p and &p?

As I said above

p = <value>, means store a value to the address of p. The value of p changed from 0x1000 to <value>

*p = <value>, means store a value to the address of p point to. The value of a changed from 1 to <value>

&p is an address. It is a constant and can not be move.

c. Fill the table, after each of the statements is executed

```
1  *p = 2;
2  p = 1;
3  *p = 2;
```

variable, address	a, 0x1000	p, 0xFFC
raw value		
*p = 2		
p = 1		
*p = 2		

variable, address	a, 0x1000	p, 0xFFC
raw value	1	0x1000
*p = 2	2	0x1000
p = 1	2	1
*p = 2	2	Try to change the memory belongs to OS. Illegal.

3. What can you learn from the following program? (From the perspective of pointer data type)

```
int main(void)
{
    int a = 0x11223344;
    int b = 0x12345678;
    int *p = &b;
    char *q = &b;

    printf("%d %d\n", sizeof(p), sizeof(q));    // 4 4

    printf("%p %x\n", p, *p);                   // 0xbfb51508 12345678
    printf("%x\n", *(unsigned char*)p);         // 0xbfb51508 78
    p = (unsigned char*)p + 1;
    printf("%x\n", *(unsigned char*)p);
        // 0xbfb51509 56

    p = &b;
    p++;
    printf("%p %x\n", p, *p);    // 0xbfb5150c 11223344

    printf("%p %x\n", q, *q);    // 0xbfb51508 78
    q++;
    printf("%p %x\n", q, *q);    // 0xbfb51509 56
    q++;
    printf("%p %x\n", q, *q);    // 0xbfb5150a 34
    q++;
    printf("%p %x\n", q, *q);    // 0xbfb5150b 12
    q++;
    printf("%p %x\n", q, *q);    // 0xbfb5150c 44

    return 0;
}
```

The step size of a pointer depends on its type.
The operation of a pointer is not limited by its type and can be converted to any type at any time if desired.

4. What is the wild pointer? Illustration

A wild pointer is a pointer that is not initialized before using.

```c
int main(void)
{
    int array[3] = {1, 2, 3};
    int *p, *q;

    *p = 2;    // An unknown address, The pointer without being initialized
    q = array[3];
    *q = 2;    // An unknown address, The address is out of expected

    return 0;
}
```

5. What is dangling pointer? Illustration

A dangling pointer is a pointer that has been released or no longer valid but is still in use.

```c
int* test_fun(void)
{
    int a = 2;

    return &a;    // Returns the automatic variable allocated on the stack
}

int main(void)
{
    int *q, *p;
    q = test_fun();
    p = malloc(10);
    memcpy(p, "hello", sizeof("hello"));
    free(p);

    printf("The return value of test_fun is %d\n", *q);    // case 1
    printf("The string is %s\n", p);                       // case 2
    return 0;
}
```

Pointer & Array

Guidance

- Chapter: Pointers-On-C.pdf 6.13.1 8.1; C-Primer-Plus.pdf 10
- Spend Time: 1.5 days
- Learn Suggestion: Learn pointer compare with array
- Key Points
    - What is the difference between array, &array[0] and &array
    - What happens when an array is the parameter of a function
    - What's the direct reference and indirect reference to an array?

Practice

1. What can you learn from the following program?

```c
int main(void)
{
    int array[] = {1, 2, 3};
    printf("%p %p\n", array, array + 1);            // 0xbfec79f4, 0xbfec79f8
    printf("%p %p\n", &array[0], &array[0] + 1);    // 0xbfec79f4, 0xbfec79f8
    printf("%p %p\n", &array, &array + 1);          // 0xbfec79f4, 0xbfec7a00
    return 0;
}
```

The array and &array[0] have the same meaning; they both represent the first element address of the array. Its type is "int *", which size is equivalent to the length of the address bus

&array is equivalent to int (*array)[3], which is a pointer to an array. Its step size is the whole array.

2. Why does the following program have such an output? Is it reasonable?

```c
int main(void)
{
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *p = a;
    int *q = &a + 1;
    int m = 22;
    int n = 33;

    p += 3;

    printf("%d %d\n", *p, q - p);     // 4 6
    printf("%d\n", &m - &n);          // 1
    return 0;
}
```

Pointer arithmetic should only occur within array ranges or between two pointers pointing to the same array. The operations outside the scope of the array are meaningless.

3. Why does the following program have such an output?

```c
void fun(int array[])
{
    printf("The lengh of array in fun is %d\n", sizeof(array));
    return;
}

int main(void)
{
    int array[5] = {1, 2, 3, 4, 5};
    fun(array);
    printf("The lengh of array in main is %d\n", sizeof(array));

    return 0;
}
```

The array will auto convert to a pointer to the first element of the array when it as a r-value.

4. Why does the following program have such an output?

```c
int main(void)
{
    int array[5] = {1, 2, 3, 4, 5};
    int *p = array;

    printf("%p %p\n", &p, p);              // 0xbf84b118 0xbf84b11c
    printf("%p %p\n", &array, array);      // 0xbf84b11c 0xbf84b11c
    return 0;
}
```

The output of &p and p are different, but the output of &array and array are the same.

There are 2 reasons for this

- The value or address of the pointer is in a different memory, but the array has only one space. So we use the array name to represent the first element address
- p and array have the same meaning, they both represent the address they store. Array is a sequential memory. Represented by the address of the first element(&array[0]).

5. Why can not an array be assigned just like a pointer?

```c
int main(void)
{
    int array1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int array2[] = {1, 2, 3};
    int *p = array1;

    p = array2;
    array1 = array2;

    return 0;
}
```

The array name cannot be used as a l-value, otherwise the array will be accessed improperly and cannot be released.

6. What is the difference between a, b and c?

```c
void func(int a[], int *b, int c[5])
{
    return;
}

int main(void)
{
    int array1[5] = {0};
    int array2[5] = {0};
    int array3[5] = {0};

    func(array1, array2, array3);
    return 0;
}
```

The formal parameters a, b, and c are all pointers to the compiler; The difference is that a[] and c[5] code are more readable and help the programmer understand the meaning of parameters more quickly.

7. Pointers-On-C.pdf 8.7 1

```c
int ints[20] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100,110, 120, 130, 140, 150, 160, 170, 180, 190, 200};
int *ip = ints + 3;
```

Assume that the ints array begins at location 0x100, the location of ip is next to array.

that integers and pointers both occupy four bytes. Addresses are sorted from highest to lowest

Expression	Value	Expression	Value
ints	0x100	ip	0x10C
ints[4]	50	ip[4]	80
ints + 4	0x110	ip + 4	0x11C
*ints + 4	14	*ip + 4	44
*(ints + 4)	50	*(ip + 4)	80
ints[-2]	out of bound, unknown	ip[-2]	20
&ints	0x100	&ip	Unknow
&ints[4]	0x110	&ip[4]	0x11C
&ints + 4	0x240	&ip + 4	Unknow
&ints[-2]	0xF8	&ip[-2]	0x104

8. Here is a program, answer some questions

```
1 int main(void)
2 {
3     int a[] = {1, 2, 3, 4, 5};
4 
5     return 0;
6 }
```

```
movl    $1, -24(%ebp)
movl    $2, -20(%ebp)
movl    $3, -16(%ebp)
movl    $4, -12(%ebp)
movl    $5, -8(%ebp)
leal    -24(%ebp), %eax
addl    $8, %eax
movl    %eax, -4(%ebp)
movl    $11, -20(%ebp)
movl    -4(%ebp), %eax
addl    $4, %eax
movl    $22, (%eax)
movl    $33, -16(%ebp)
movl    -4(%ebp), %eax
addl    $8, %eax
movl    $44, (%eax)
movl    $0, %eax
```

a. Convert the following assembly language to C language

```c
int main(void)
{
    int a[] = {1, 2, 3, 4, 5};
    int *p = &a[2];

    *(a + 1) = 11;
    *(p + 1) = 22;
    a[2] = 33;
    p[2] = 44;

    return 0;
}
```

b. Which are the direct/indirect references in this program?

Direct reference	*(a + 1) = 11 a[2] = 33
Indirect reference	*(p + 1) = 22 p[2] = 44

9. Convert the following assembly language to C language, What can you learn from the following program?

```
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $32, %esp
        movl    $.LC0, -4(%ebp)
        movl    $1819043144, -17(%ebp)
        movl    $1867980911, -13(%ebp)
        movl    $560229490, -9(%ebp)
        movb    $0, -5(%ebp)
        movl    -4(%ebp), %eax
        addl    $2, %eax
        movb    $11, (%eax)
        movb    $22, -15(%ebp)
        movl    $0, %eax
        leave
        ret
```

```
int main(void)
{
    char *str1 = "Hello World!";
    char str2[] = "Hello World!";

    str1[2] = 11;
    str2[2] = 22;
    return 0;
}
```

The way for string literals and character array data may be different, pushed into the stack.

Pointer & Function

Guidance

- Chapter: The-C-Programming-Language-2nd.pdf 5.2; Computer-Systems-A-Programmers-Perspective.pdf 3.4.2 3.5.1
- Spend Time: 1.5 days
- Learn Suggestion: Learn the behavior of passing a parameter by address to functions through assembly language
- Key Points
  - How the value of the parameter is changed when passing an address to the function?
  - How many approaches to get a new memory from another function?
  - What's the difference between mov[l|q] and lea[l|q] instruction?

Practice

1. What can you learn from the following program?

```c
#include <stdio.h>

void test_fun(int a, void* b, char *c)
{
    *(int*)a = 11;
    *(int*)b = 22;
    *(int*)c = 33;

    return;
}

int main(void)
{
    int a = 1, b = 2, c = 3;

    test_fun(&a, &b, &c);
    printf("a=%d, b=%d, c=%d\n", a, b, c);    // a=11, b=22, c=33

    return 0;
}
```

Pointer usage isn’t influenced by its type. Any pointer type can be converted to other types whenever necessary.

2. There is a program in file "Pointer and Function.c", Answer the following questions
   a. Write the corresponding C code next to the assembly statement
   Refer to "Pointer and Function.c"

b. Draw some pictures to show the stack changes after each assembly instruction is executed

Refer to "Pointer and Function.xlsx"

3. What is the problem of the following program?

Rigorous inspections should be strengthened. such as, the process should be terminated when str is null.

a. P is a formal parameter whose address stores the value of str rather than its address. So the value of str is not changed.

```
void get_memory(char *p)
{
    p = (char *)malloc(100);
    return;
}
int main(void)
{
    char *str = NULL;
    get_memory(str);
    strcpy(str, "Hello World!");
    printf(str);
    return 0;
}
```

b. The address of p is allocated on the stack. It'll be invalid as the function get_memory exits.

```
char *get_memory(void)
{
    char p[] = "Hello World!";
    return p;
}
int main(void)
{
    char *str = NULL;
    str = get_memory();
    printf(str);
    return 0;
}
```

c. The Standard library functions malloc and free must be used in pairs

```c
void get_memory(char **p, int num)
{
    if (p == NULL)
        return;
    *p = (char *)malloc(num);
    return;
}
int main(void)
{
    char *str = NULL;
    get_memory(&str, 100);
    strcpy(str, "Hello World!");
    printf(str);
    return 0;
}
```

Pointer & Struct

Guidance

- Chapter: Pointers-On-C.pdf chapter 11 12
- Spend Time: 2 days
- Learn Suggestion: Structured the data
- Key Points
    - Why do we need the struct pointer? Why don't we use a struct variable directly? What are the benefits of a struct pointer?
    - How to maintain the new memory allocated by library functions malloc/calloc/realloc?
    - How to maintain a linked list? Append/Insert a new node to the linked list, Remove an existing node from the linked list.

Practice

- Reference
    - RFC Document: Ethernet/IP(v4), DIX frame/rfc791
    - Technic Point: Linked list, Data structuring, Data Alignment in the packet, Bit field in the struct
- Description
    - Capture at least 10 complete IP packets by Wireshark and store them to a file.
    - Read the packet from the file individually and build a linked list to store them.

    - Create a new node for the packet if it's nonexistent (same src/dest ip) in the structure, otherwise auto-increment 1 to the reference counter. print out the reminder information
    - Print out the key-information(smac/dmac/sip/dip) of a packet when handle it
- Requirements
    - The raw data need to be structured
    - Portability needs to be considered

Topic 6: Advanced Pointers

Pointer to Pointer & Array of Pointer

Guidance

- Chapter: Pointers-On-C.pdf chapter 6.10 8.3 13.1 13.4
- Spend Time: 1 day
- Learn Suggestion: Understanding the memory address and the relationship of each type of variable
- Key Points
    - Why do we need a pointer to pointer?
    - What is an array of pointer? Why do we need it?
    - What is the difference between an array of pointer and a normal array?
    - What is the relationship between an array of pointer and a pointer to pointer?

Practice

1. Here is a program, and answer some questions

```c
int main(void)
{
    int a = 2, *p = &a, **q = &p;

    printf("The value of a is 0x%-8x, Its address is %p\n", a, &a);
    printf("The value of p is 0x%-8x, Its address is %p\n", p, &p);
    printf("The value of q is 0x%-8x, Its address is %p\n", q, &q);

    return 0;
}
```

    a. Draw a picture to show the relationship between these 3 variables
    b. What does the following expression mean?

```
*a, **a;
*p, **p;
*q, **q;
```

***a**: means get the value of the location **a** is pointing at. This shows segment fault. Alternative way would be ***(int *)a**

****a**: get the value of the location a is pointing at, use that value as an address and then get the value from that address. This shows segment fault. Alternative would be ***(int *)*(int *)a**

***p**: get the value of the location p is pointing at which is 2.

****p**: get the value of the location p is pointing at (2), use it as an address and get the value of that address. This cause segment fault as we are accessing memory location 2.

***q**: get the value of the location q is pointing at. It is the address of a.

****q**: get the value of the location q is pointing at (&a), use it as an address and get the value of that address which is 2.

c. Based on the main program. What can you learn from the following program?

```
int **p1 = &a;
int *p2 = &p;

printf("%d %d\n", *p1, **(int **)p2);
```

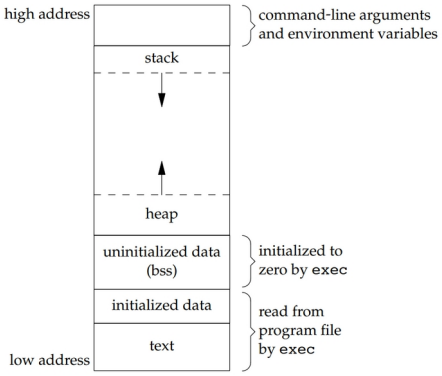
p1 is a **pointer to a pointer**. p1 points to **a** and considers **a** to be another pointer. So, ***p1** shows the value containing at the address of **a** which is **2**.
p2 is a **pointer to an integer**. It is pointing to **p** which is another pointer but p2 treats it as an integer. When explicit type cast **(int **)** is used, now p2 is a **pointer to pointer**. The first indirection gets the value of **p** which is the address of **a**. The second indirection goes to the address of **a** and gets it' s value which is **2**.

2. Read the program below, and answer some questions

```
int main(int argc, char *argv[], char *envp[])
{
    return 0;
}
```

a. What kind of data is stored in argv and envp? Where are they stored in memory?

argv and envp are both array of pointer. They store different kinds of strings. Get from parent process.
They are stored between the stack and the kernel memory.



b. How many ways to get the content of argv?

We can get the content of the argv using 2 ways

I. Using subscript notation

```
int i;
while(argv[i] != NULL)
{
    printf("%s ",argv[i]);
    i++;
}
```

II. Using pointer

```
while(*argv != NULL){
    printf("%s ",*argv);
    argv++;
}
```

c. What is the difference between char *argv[] and char **argv in here?

c) What is the difference between char * argv[ ] and char **argv be used in here?

Answer: There is no difference between this 2 declaration. We also know that each element of the array is a null terminated string. So char * argv[ ] defines an array of character pointer and each element of argv points a null terminated string. Here char **argv is a double pointer which we can also use in place of char * argv[ ].

3. In which scenario is a pointer to pointer used?

A. **Dynamic Memory Allocation using function:** When we want to allocated memory for specific data type using function, we need to use pointer to pointer.

```c
#include<stdio.h>
void get_memory(char **p)
{
    *p = (char *)malloc(100*sizeof(char));
    return;
}

int main()
{
    char *str = NULL;
    get_memory(&str);
    strcpy(str, "Hello World");
    printf(str);
    return 0;
}
```

B. **Dynamic Memory allocation for string array:** We need to use pointer to pointer to allocated memory for string array.

```c
char **strs;
strs = (char **)malloc(5*sizeof(char*));
strs[0] = "Hello";
strs[1] = "World";
strs[2] = "Goodbye";
int i =-1;
while(++i < 3)
    printf("%s\n", strs[i]);
return 0;
```

4. What is the relationship between an array of pointer and a pointer to pointer?

```c
void func(char **pptr)
{
    char *ptr = NULL;

    while(*pptr)
    {
        ptr = *pptr;

        while(*ptr)
        {
            putchar(*ptr);
            ptr++;
        }

        putchar('\n');
        pptr++;
    }

    return;
}

int main(void)
{
    char *str[] = {"hello", "bdcom", NULL};

    func(str);

    return 0;
}
```

There are two associations

- We can indirectly access each element of an array of pointer by a pointer to pointer
- They are the same in some situations, ex. As a formal parameter of a function

Pointer to Array & Multi-Dimensional Array

Guidance

- Chapter: Pointers-On-C.pdf 8.2; Understanding-And-Using-C-Pointers.pdf chapter 4
- Spend Time: 1 day
- Learn Suggestion: Row-major and step size
- Key Points
  - What's the meaning of the different forms of MD array?
  - What is the relationship between MD array and pointer to array?
  - In what scenario can a pointer to array be used?

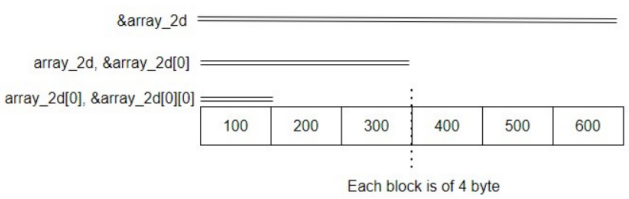
Practice

1. There is a 2D array, answer some questions

```
1 int array_2d[2][3] = {1, 2, 3, 4, 5, 6};
```

   a. What's the meaning of &array_2d、array_2d，&array_2d[0]，array_2d[0]，&array_2d[0][0]?

```
int array_2d[2][3] = {{100, 200, 300}, {400},{500},{600}};
```



&array_2d	Address of the full array.
array_2d	Address of the first block. Here the first block is a 1D array. So it is the address of an array of 3 elements.
&array_2d[0]	Same as array_2d
array_2d[0]	Address of the first block of array_2d[0]. in this case it is a int. So it is the address of array_2d[0][0] or pointer to a int.
&array_2d[0][0]	Same as array_2d[0]

   b. What is the relationship between a 2D array and a pointer to array?

Item	pointer to array(Be equivalent to)
&array_2d	int (*p1)[2][3];
array_2d	int (*p2)[3];

```c
int main(void)
{
    int array_2d[2][3] = {1, 2, 3, 4, 5, 6};
    int (*p1)[2][3] = &array_2d;
    int (*p2)[3] = &array_2d[0];

    printf("%p %p %p\n", &array_2d, &array_2d + 1, p1 + 1);
    printf("%p %p %p\n", &array_2d[0], &array_2d[0] + 1, p2 + 1);
    return 0;
}
```

c. How to access each element of the two-dimensional array through a pointer to array?

```c
int main(void)
{
    int array_2d[2][3] = {1, 2, 3, 4, 5, 6};
    int (*p)[3] = NULL;
    int *q = NULL;

    for(p = array_2d; p < (int (*)[3])(&array_2d + 1); p++)
    {
        for(q = (int*)p; q < (int*)(p + 1); q++)
            printf("%d\n", *q);
    }

    return 0;
}
```

2. Why do we need a pointer to array? What is it used for?

```c
void func_print(int (*p)[3], int end_mem_addr)    // case a, For passing multi-dimensional array to a function.
{
    int *q = NULL;

    // case c, Pointer arithmetic improves code efficiency
    for(; (int)p < end_mem_addr; p++)
    {
        for(q = (int*)p; q < (int*)(p + 1); q++)
            printf("%d ", *q);
    }

    putchar('\n');
}

void func_insert(int (*p)[3], int end_mem_addr)
{
    int *q = NULL;
    int counter = 1;

    for(; (int)p < end_mem_addr; p++)
    {
        for(q = (int*)p; q < (int*)(p + 1); q++, counter *= 2)
            *q = counter;
    }
}

void func_print_string(char (*p)[8], int num)
{
    int i;

    // case d, Easy string manipulation
    for(i = 0; i < num; i++, p++)
    {
        printf("%s ", p);
    }

    putchar('\n');
}

int main(void)
{
    int array_2d[2][3] = {1, 2, 3, 4, 5, 6};
    func_print(array_2d, (int)(&array_2d + 1));

    int (*p)[3] = malloc(2 * 3 * 4);        // case b, Creating dynamic array.
    func_insert(p, (int)p + 2 * 3 * 4 + 1);
    func_print(p, (int)p + 2 * 3 * 4 + 1);
    free(p);

    char strs[8][8] = {"hello", "world"};
    func_print_string(strs, 2);

    return 0;
}
```

3. Pointers-On-C.pdf 8.7 11

```
int array[4][2][3][6];
```

Item	Address Format	Type	Size
	&array	int (*)[4][2][3][6]	0x240 = 4 * 2 * 3 * 6 * 4
array	&array[0]	int (*)[2][3][6]	0x90 = 2 * 3 * 6 * 4
array[0]	&array[0][0]	int (*)[3][6]	0x48 = 3 * 6 * 4
array[0][0]	&array[0][0][0]	int (*)[6]	0x18 = 6 * 4
array[0][0][0]	&array[0][0][0][0]	int *	0x4 = 4

array	array[0]	0x1000
array + 1	array[1]	0x1090 = 0x1000 + 0x90
array + 2	array[2]	0x1120 = 0x1000 + 0x90 * 2
array + 3	array[3]	0x11B0 = 0x1000 + 0x90 * 3

Expression	Value	Type of x
array	0x1000	int (*)[2][3][6]
array + 2	0x1120 = 0x1000 + 0x90 * 2	int (*)[2][3][6]
array[3]	0x11B0 = 0x1000 + 0x90 * 3	int (*)[3][6]
array[2] - 1	0x10D8 = 0x1000 + 0x90 * 2 - 0x48	int (*)[3][6]
array[2][1]	0x1168 = 0x1000 + 0x90 * 2 + 0x48	int (*)[6]
array[1][0] + 1	0x10A8 = 0x1000 + 0x90 + 0x18	int (*)[6]
array[1][0][2]	0x10C0 = 0x1000 + 0x90 + 0x30	int *
array[0][1][0] + 2	0x1050 = 0x1000 + 0x48 + 0x8	int *
array[3][1][2][5]	value	int
&array[3][1][2][5]	0x123C = 0x1000 + 0x90 * 3 + 0x48 + 0x30 + 0x14	int *

4. Pointers-On-C.pdf 8.7 13

```
int array[4][5][3];
```

Item	Address Format	Type
	&array	int (*)[4][5][3]
array	&array[0]	int (*)[5][3]
array[0]	&array[0][0]	int (*)[3]
array[0][0]	&array[0][0][0]	int *

Expression	subscripts	Type of x
*array	array[0]	int (*)[3]
*(array + 2)	array[2]	int (*)[3]
*(array + 1) + 4	array[1] + 4	int (*)[3]
*(*(array + 1) + 4)	array[1][4]	int *
*(*(*(array + 3) + 1) + 2)	array[3][1][2]	int
*(*(*array + 1) + 2)	array[0][1][2]	int
*(**array + 2)	array[0][0][2]	int
**(*array + 1)	array[0][1][0]	int
***array	array[0][0][0]	int

Pointer to Function

Guidance

- Chapter: Understanding-And-Using-C Pointers.pdf chapter 3; Pointers-On-C.pdf 13.3
- Spend Time: 1 day
- Learn Suggestion: Passing a function just like a value between the different functions
- Key Points
  - Why do we need this technique? What can it bring to us?
  - How to use this technique to implement the idea of object orientation?
  - How to use this technique to implement enhanced code portability?

Practice

1. Implement the following program by pointer to function

```c
int main(void)
{
    int a = 7, b = 3;

    printf("%d\n", calculate(a, b, '-'));
    printf("%d\n", calculate(a, b, '/'));
    printf("%d\n", calculate(a, b, '%'));

    return 0;
}
```

Approach 1

```c
#include<stdio.h>
typedef int (*operationPointer) (int, int);

int subtruct(int a, int b)
{
    return (a) - (b);
}
int divide(int a, int b)
{
    return (a) / (b);
}
int mod(int a, int b)
{
    return (a) % (b);
}

int calculate(int a, int b, char code)
{
    operationPointer ptr;
    switch(code)
    {
        case '-':
            ptr = subtruct;
            break;
        case '/':
            ptr = divide;
            break;
        case '%':
            ptr = mod;
            break;
    }

    return ptr(a, b);
}

int main()
{
    int a = 7, b = 3;

    printf("%d\n", calculate(a, b, '-'));
    printf("%d\n", calculate(a, b, '/'));
    printf("%d\n", calculate(a, b, '%'));

    return 0;
}
```

Approach 2

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

#define PRINT_INT(x)    printf("%d ", x)
#define NL              printf("\n");

typedef int (*fptrOperation_t)(int, int);

int sub(int num1, int num2) {
    return num1 - num2;
}

int divide(int num1, int num2) {
    return num1 / num2;
}

int mod(int num1, int num2) {
    return num1 % num2;
}

fptrOperation_t operations[128] = {NULL};
void initializeOperations() {
    operations['-'] = sub;
    operations['/'] = divide;
    operations['%'] = mod;
}

int calculate(int num1, int num2, char opcode) {
    fptrOperation_t operation = operations[opcode];
    return operation(num1, num2);
}

int main(void)
{
    initializeOperations();

    int a = 7, b = 3;
    printf("%d\n", calculate(a, b, '-'));
    printf("%d\n", calculate(a, b, '/'));
    printf("%d\n", calculate(a, b, '%'));

    return 0;
}
```

2. Implement the Jump Tables on page 360 of the book Pointers-On-C.pdf

```c
int add(int x, int y)
{
    return (x + y);
}

int sub(int x, int y)
{
    return (x - y);
}

int mul(int x, int y)
{
    return (x * y);
}

int div(int x, int y)
{
    return (x / y);
}

int main(void)
{
    int x, y;
    int input = 1;
    int (*p[5])(int, int) = { 0, add, sub, mul, div };

    while (input)
    {
        printf("**********************\n");
        printf("         1: ADD      \n");
        printf("         2: SUB      \n");
        printf("         3: MUL      \n");
        printf("         4: DIV      \n");
        printf("         0: EXIT     \n");
        printf("**********************\n");

        printf("Enter a choice: ");
        scanf("%d", &input);

        if (input == 0)
        {
            break;

        }
        else if (input < 1 || input > 4)
        {
            printf("wrong input!\n");
        }
        else
        {
            printf("Enter two numbers: ");
            scanf("%d %d", &x, &y);
            printf("result = %d\n", (*p[input])(x, y));
        }
    }

    return 0;
```

3. The main function and output have already been provided, to complete the remaining parts.

```c
void main(void)
{
    int int_a = 10;
    char str1[] = "bdcom";
    person_t person = {"Tom", 18};

    memset(&infos, 0, sizeof(infos));

    register_info("int", print_int);
    register_info("string", print_string);
    register_info("person_t", print_person);

    print_text(&int_a, "int");
    print_text(&str1, "string");
    print_text(&person, "person_t");

    return;
}
```

```
[xiaohei@localhost share]$ gcc test.c && ./a.out
10
bdcom
name:Tom age:18
```

```c
#define NUM 10
typedef void (*my_print_t) (void*);

struct info
{
    char name[12];
    my_print_t my_print;
};

struct info infos[NUM];

typedef struct
{
    char name[64];
    int age;
} person_t;

void print_text(void* data, char *name)
{
    int i;

    for (i = 0; i < NUM; i++)
    {
        if (!strcmp(name, infos[i].name))
            infos[i].my_print(data);
    }
}

void print_int(void* data)
{
    int* num = data;
    printf("%d\n", *num);
}

void print_string(void* data)
{
    char* str1 = data;
    printf("%s\n", str1);
}

void print_person(void * data)
{
    person_t * p = (person_t *)data;
    printf("name:%s age:%d\n", p->name, p->age);
}

void register_info(char *name, my_print_t fun)
{
    static int counter = 0;

    strncpy(infos[counter].name, name, strlen(name) + 1);
    infos[counter].my_print = fun;

    counter++;

    return;
}
```

Pointer Enhancement

Guidance

- Chapter: The-C-Programming-Language-2nd.pdf chapter 5
- Spend Time: 1 day
- Learn Suggestion: Pointer knowledge integration and strengthening
- Key Points
    - How many methods can be used to declare a pointer by the keyword const?
    - How to simplify a complex pointer definition?
    - What is the most closely related to the step size?

Practice

1. How to prevent the variable a be changed?

```c
void func(int *p)
{
    *p += 1;
    return;
}

int main(void)
{
    int a = 2;
    func(&a);
    printf("%d\n", a);    // 3

    return 0;
}
```

2. Supplement the corresponding formal parameters and add the corresponding actual parameters. List all the forms that you know. Refer to the one-dimensional array a

```c
void funca1(int a[]){}
void funca2(int a[ARR_NUM]){}
void funca3(int *a){}

void funcb(){}
void funcc(){}
void funcd(){}
void funce(){}

int main(int argc, char *argv[], char *envp[])
{
    int a[ARR_NUM] = {1, 2, 3};
    int *b[5];
    int (*c)[5];
    int d[4][5];
    int **e;

    funca1(a);
    funca2(a);
    funca3(a);

    funcb(b);
    funcc(c);
    funcd(d);
    funce(e);

    return 0;
}
```

3. Write a program lists out all the types of pointer step size that you know
4. Complete the definition according to the description. using the variable ptr

description	type definition
A pointer to a char constant	const char *ptr;
A constant pointer to a char volatile	
A constant pointer to char	
An array of 10 pointers that point to an integer	
A pointer to an array of 10 integers	
A pointer to a function that takes an integer parameterand returns an integer	
An array of 10 pointers that point to a function thattakes an integer parameter and returns an integer	

5. Using typedef to simplify complicated declarations
    a. int (*(*AA)(void *))[10];
    b. void (*BB(int, void (*)(int)))(int);
    c. char (*(*CC[3])())[5]