# Topic 1: C First Acquaintance

## Guidance

1. Why do we use C in embedded system? Why don't we use other high-level Programming Language, like Java/Python/C++? Why don't use assembly language?
2. What is a compiled language? What is an interpreted language? Which languages are compiled? Which languages are interpreted? What's the difference between them? What are the advantages and disadvantages?
3. What is syntax checking in compilation and What is semantic checking? What's the difference between them?
4. What are the steps and tools needed to go from a C program to an executable program? What do these tools do? Why are these steps needed?

## Practice

1. Use different options in gcc to see the input, output and role of different compilation tools

| tool name | pre-processor | Compiler | Assembler | Linker |
|---|---|---|---|---|
| gcc tool | cpp | cc1 | as | ld |
| 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 executable file |
| input | source file | expand file | assemble file | object file |
| output | expand file | assemble file | object file | executable file |

2. Use readelf to view the composition of an executable file. What sections are made up of an executable file? Draw a picture. What do these sections do?

| The major composition of an executable file | |
|---|---|
| ELF Header | Records the basic properties of current file.<br>Contains pointers to the locations of Section Header Table and Program Header Table within the ELF file |
| Section Headers | Describes the information for each section. Be used by the linker |
| Program Headers | Be used by the loader<br>Only exist in executable File |
| Others | We don't have to pay attention right now |

| The type of ELF Header | |
|---|---|
| Relocatable File | Contains code and data that can be used to link to executable files or shared object files, and static libraries can also be classified as such |
| Executable File | Files that have been relocated |
| Shared Object File | Contains code and data.<br>in two cases:<br>• a static linker that can be used for relocation to produce new object files,<br>• a dynamic link library such as.so and.lib. |
| Core Dump File | The dump file is generated after a program crash. It saves various crash scenes and can be used to analyze and debug the cause of the crash. |

| Description of major sections | | |
|---|---|---|
| section | description | data |
| .rodata | read-only data | leave it out for now, follow up later |
| .bss | uninitialized global and static variables | |
| .data | initialized global and static variables | |

| | | |
|---|---|---|
| .text | machine instructions, only 1 copy, read only | |

3. Is it possible to mix C and assembly programming? How to do it? (You can find examples on the Internet to verify.)

Only need to know the usage of keyword asm

# Topic 2: Basic Concept

## Constant

## Guidance

1. What is constant?
2. Which type of constant do you know?
    1. number. different base
    2. character. storage, representation, special characters
    3. enumeration.
    4. string. storage, representation
3. keyword const. global and local variable

## Practice

1. How to representation characters? How to store 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 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
printf("D - 2 %d\n", 'D' + 'A'); // D - 2 is 18, There is not any meaning here
```

The 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 |

2. What are the special characters? What do they do? How do I print 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)

3. How to define and represent different base 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
```

4. How to define and use enumeration? Write code to describe

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

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

- The usage is very flexible

```
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 a typedef to define a type (leave it out for now, follow up later)

5. Why do we say string is constant? Illustration

- Compare the output of a and b with objdump -h to see which part is 7 bytes larger

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

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

- To check the address. Leave it for you

6. Can we change the value of a const variable? How to do?

It's different for global and local variables.
global variables declare 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 pointer.

# Variable

# Guidance

1. Composition of variable
2. Data Type
3. local and global variable

# Practice

The Linux tools objdump and readelf can help you know more about that how a variable be stored in memory

1. What are the rules for constitute the variables?

- The variable name consists of letters, digits and underscores, starting with a letter or an underscore. Normally a variable starting with a letter used for normal user, and starting with underscore used for system base functions.

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

You can find here, There are really many functions which name is start with underscore
https://github.com/torvalds/linux/blob/master/include/linux/socket.h

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

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

- Variable names are case sensitive.

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

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

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

2. What are the types of variables? What's the difference between them?

Misunderstanding

3. How to define and use variables? Assign and Use

- The variable need be define before to use it
- We can give the value when define it, The format is = ;
- We also can define it without value assign. The format is ;
- We need set the variable on left side and value on right side when do the assignment. =
- The variable can be used after it be assigned a value

4. What is the nature of a variable name? Illustration

Variable Name is only exist when the program was a C code. Then it will become a address.

5. Where do computer store local and global variable in memory? Topic 1->Practice->question

Refer to program "The address of constant and variable.c"

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

- When it used as a global variable, the scope of the variable changes to the file which define it, that is mean, the variable cannot be used in a source file other than the current file.

- When it used as a local variable, the variable is stored in the global variable table. The lifetime of the variable does not depend on the call stack.

7. What is different between local and global variable when it declare by static? And Where the computer store them in memory? Topic 1->Practice->question

Refer to program "The address of constant and variable.c"

# Operators and Expressions

## Guidance

Pointers On C.pdf -> Chapter 5

## Practice

1. What are the rules of expression operation?

You can understand it in your own way and don't need to memorize it. In particular, the priority problem can be solved by parenthesis if you are unsure when programming.
This helps to improve the readability of the code, but it is necessary to know some priorities, otherwise adding too many brackets will affect the readability of the code.
This question is more subjective, you can judge for yourself, I give a few of my habits, as an example. It's not mandatory here.

```
// 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

```c
int is_leap_year = 0, year = 0;

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

```c
#include <stdio.h>
#define SQUARE(a) (a) * (a)

int main(void)
{
    int a, b;

    a = b = 1;
    printf("%d\n", a++ + ++b);                  // hard to memorize

    a = b = 1;
    printf("%d\n", ++a + ++b);                  // hard to memorize

    a = b = 1;
    printf("%d %d\n", SQUARE(a++), SQUARE(++b));    // Depending on the compiler, the result is uncertain

    return 0;
}
```

4. Pointers On C.pdf 5.8 question 7

Computers can't think as flexibly as people, and they need to follow the rules of operators.
Both 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
```

5. Pointers On C.pdf 5.8 question 12

There are 2 kinds of way used for right shift in C, arithmetic or logical shift. The rules 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 Programmer's Perspective.pdf -> 2.1.10

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

```
int a = -1;

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

6. Pointers On C.pdf 5.9 question 3

There are 2 points need to note:

- To apply what you've learned, it's easier to use the shifts you've just learned.
- Integer type is not always 32 bits. The program should be portable

7. What is the output of the following program? Why?

```
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 expression

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.

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);
```

This answer for c and d get from Mormo

7(c)

The output is **22 0.** The reason is given in the below table:

| No | Expression Procedure | Reason | a | b | c | d |
|---|---|---|---|---|---|---|
| 1 | | Initial | 0 | 0 | 0 | 0 |
| 2 | d = (c = (a = 11, b = 22)) == 11 | Assignment of a, b | 11 | 22 | 0 | 0 |
| 3 | d = (c = (b = 22)) == 11 | Due to comma operator C is assigned 22 | 11 | 22 | 22 | 0 |
| 4 | d = (c = 22) == 11 | Due to assignment operator d is assigned to 22 | 11 | 22 | 22 | 22 |
| 5 | d == 11 | Relational Operator Returns 0 or 1 | 11 | 22 | 22 | 0 |

7(d)

The output is **2 2 2**

| No | Expression Procedure | Reason | x | y | z |
|---|---|---|---|---|---|
| 1 | z = x < y ? !x : | Initial | 3 | 2 | 3 |
| 2 | Z = !((x=2) \|\| (x=3)) ? (x = 1) : (y = 2) | Conditional operator goes false | 3 | 2 | 3 |
| 3 | Z = !(x) ? (x = 1) : (y = 2) | x=2 assignment is executed | 2 | 2 | 3 |
| 4 | Z = !(x) ? (x = 1) : (y = 2) | Conditional Operator goes false | 2 | 2 | 3 |
| 5 | Z = (y = 2) | y gets assigned, then z gets assigned | 2 | 2 | 2 |

# Macro & Typedef

# Guidance

1. What is typedef? Is it used for create a new type?
2. Why do we need typedef?
3. Pointers On C.pdf 14.2
4. The name of micro & typedef(reference the Programming specification doc)

## 3.1 Naming

- Function names and variable names use lowercase with underscores.
- Macro definitions use uppercase with underscores.
- The name of typedef use lowercase end with _t.

# Practice

1. What is the different between define and typedef? Illustration

| | define | typedef |
|---|---|---|
| nature | replace text | type alias |
| timing | pre-processing stage | compile stage |
| mode | replace text only | a part of compile, including type check |
| function | Used to define constants or long string replace | Simplify complex declarations |

Both define and typedef can help simplify code and improve its readability. But sometimes they are different

```
#define U_INT32_1 unsigned int*;
#typedef U_int32_2 unsigned int*;

U_INT32_1 a, b;
U_INT32_2 c, d;
```

2. How to define a macro function? What are the points for attention?

- 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

```c
#include <stdio.h>
#define SQUARE(a) (a) * (a)

int main(void)
{
    int a = 1, b = 1;
    printf("%d %d\n", SQUARE(a++), SQUARE(++b));

    return 0;
}
```

- Do not end with semicolon, it's not a statement

3. Pointers On C.pdf 14.9 question 2

- Improve program readability. We can know the meaning of the constant from macro name easily
- Increase program maintainability. All it takes is one change if we need to change the value of macro

4. What is the type of T1?

```c
struct foo
{
 int x;
 char y;
};
typedef struct foo *T1;
```

The type of T1 is struct foo *

5. Define a macro swap(t, x, y) that interchanges two arguments of type t.

```c
#define SWAP(data_type, a, b) {data_type temp = a; a = b; b = temp;}
```

6. Write a macro MIN that accepts three parameters and returns the smallest one

```
#define MIN(x,y) ((x)<(y)?(x):(y))
#define MIN2(x,y,z) (MIN(x,y)<(z)?MIN(x,y):(z))
#define MIN3(x,y,z) MIN(x, MIN(y, z))

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

7. When do we need typedef? Illustration

I think it is better to use typedef here

- used for pointer

```
// normal pointer
typedef char * pchar_t;
int main (void)
{
  pchar_t str = "hello world!";
  printf("str: %s\n", str);
  return 0;
}
// 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;
}
```

- Increased portability of code.

```
#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];
```

I do not think it is better to use typedef here

- Used for struct, It can save a keyword struct, but loses the information that it is a struct

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

struct student stu1;
student_t stu2;
```

- Used for array, Do not conducive to code readability, need to look at the definition

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

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

- Used for enumeration, It can save a keyword enum, but lose the information that it is a enumeration

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

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

8. What is the different between macro function and normal function?

| | macro function | normal function |
|---|---|---|
| nature | replace text | function call stack |
| timing | pre-processing stage | running stage |
| advantage | Do not need new stack | Call function are not affected |
| disadvantage | Expanding makes the call function become larger, increasing the cost of calling the function | Need new stack and register |

# Topic 3:Control Flow

## Selection Statement

## Guidance

Pointers On C.pdf Chapter 4.1-4.4 4.8
The C Programming Language 2nd.pdf 3.1-3.4

1. What is the relationship between expressions and statement?
2. bool type
3. Why do we need to indent if the compile allow us write code like this?

```
if (A) if (B) C; else D;

if (A)
     if (B)
 C;
else
     D;
```

## Practice

1. What is the function of control statements? What kind of form it can be?

Control statement is a statement used to control the selection, loop, turn, and return of a program flow.

Control expression can be any form

```
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 uses of if can also be used to while/for/switch/return

2. How many control statements can C code have? How many categories can we divide them into?

There are 9 control statements be devided into 3 categories

| category | control statements |
| --- | --- |
| Selection | if, switch |
| Loop | while, do while, for |
| Turn to | break, continue, return, goto |

3. 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

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

```
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 convenience to use if in multiple condition
- 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)

4. How many forms of an if statement?

There are 3 forms of if statements

| | if | if-else | if-elseif-else |
|---|---|---|---|
| description | only need logical true | logical true or false | Multiple logic or need arithmetic results |
| syntax | if (control expression)<br>; | if (control expression)<br>;<br>else<br>; | if (control expression 1)<br>;<br>else if (control expression 2)<br>;<br>else if (control expression ...)<br>;<br>else<br>; |

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

- The value of case must be a inter
- The case lable must be unique
- Do not forget to add after case statement otherwise a fall-though will happen
- Do not forget to add default statement end with all of the case, which can help you catch the case that you thoughtless

- The Statements that do not be put in a case statement will not be executed

```c
#include <stdio.h>

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;
}
```

6. What is Dangling-else? Illustration

The rule of an else match with if is that it will match with the if closest with it.
Therefore, Something will happen that out of our expect if we wrote if without brace

```
// 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;
```

7. What is switch fall-though? What are its advantages and disadvantages?

Advantages: Avoid writing duplicate code. Make the code look cleaner
Disadvantages: Multiple cases reuse the same code. If the logic of one branch need to change, it's easy to forget about fall-though, so that the logic of the other cases is also modified

8. 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
if (!condition1)
    statement1
```

9. Pointers On C.pdf 4.13 question 16

```
if (precipitating)
{
    if (temperature < 32)
        printf("snowing\n");
    else
        printf("raining\n");
}
else
{
    if (temperature < 60)
        printf("cold\n");
    else
        printf("warm\n");
}
```

# Loop & Goto Statement

## Guidance

Pointers On C.pdf Chapter 4.5-4.7 4.9
The C Programming Language 2nd.pdf 3.5-3.8

1. What is the execute order of the 3 parts of the for control expression?
2. How to convert between while and for?

## Practice

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

There are 3 type of loop statements in C.
There is no any difference between for and while in essence, only look different in form, they 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)<br>statement | init; condition; change condition,When these 3 parts are necessary together, using for makes the code look more intuitive<br><br>for(i = 0; i < 3; i++)<br>statement |
| while | init;<br>while (condition)<br>{<br>statement<br>change condition<br>} | When we do not need all the 3 parts appear at the same time, using while makes the code look more intuitive<br>while(q = (singly linked list)->next); // point p to the end of singly linked list |

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

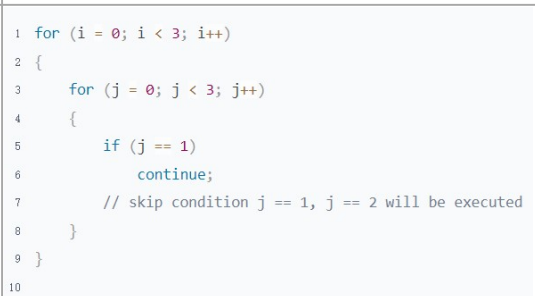
2. What is the different between following 2 programs? Give me an example to illustrate the impact of these two methods on the actual results

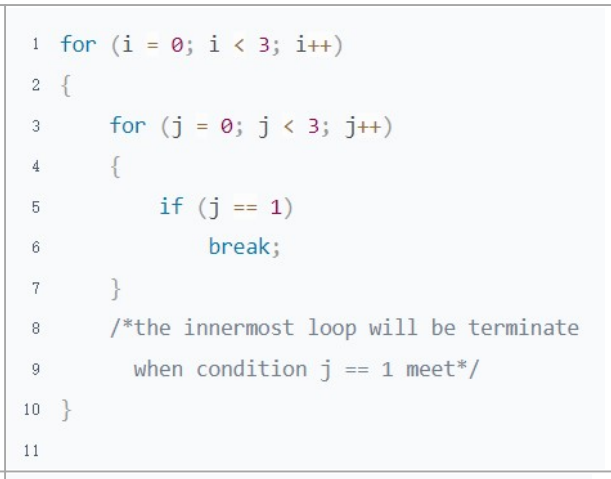
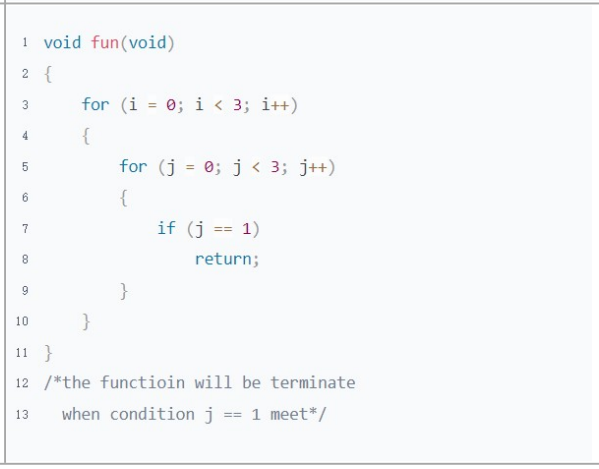
```
a.
while(expression_1)
{
    statement_1
}
b.
while(1)
{
    if(expression_1)
        statement_1
}
```

```
unsigned int bitmap = 0x5e3b;   // How many bit has been set to 1
```

| a | b |
| --- | --- |
| Loop terminate depend on loop condition. Loop will terminate if expression_1 is false at one time | Loop terminate depend on if condition. Loop do not terminate until some condition met of if statement |
|  |  |

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

| | | |
| --- | --- | --- |
| continue | skip this time loop of the innermost loop |  |

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

4. What are the difference between break and return when we use them in a case statement?

break: Terminate current case
return: Terminate current function

5. Pointers On C.pdf 5.8 question 8

```
// 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;
}
```

6. What are advantages and disadvantages of goto statement?

Advantages: Make the code can flexibly jump within the function
Disadvantages: goto statement breaks the C code execution structure, making the readability and maintainability of the code greatly reduced. And affect the execution efficiency of machine instructions

```
// 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;
}
```

There is one situation in which the use of goto is recommended. When a function exits and needs to do some work on resource cleaning classes, 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
}
```

There is another 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:
}
```

# Topic 4: Functions

## Basic Knowledge

## Guidance

Pointers On C.pdf 3.5
Pointers On C.pdf 7.1-7.3 7.7-7.9

1. Can variables defined in one function be accessed in another function?
2. What will happen if you define a function with the static keyword?
3. Write a simple program and read the according assembly language
4. Function define/declaration
5. Inline function

## 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 each other, Enhanced the code maintainability
- Reused, It can be called multiple times in different places

   b) The syntax of 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 function is totally same as 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 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. This would consume 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 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 find the define of 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 is the different between a, b and c? From a stack perspective

```c
void test_fun(int b) // b is a formal parameter
{
    int c;           // c is a local variable
    return;
}

int main(void)
{
    test_fun(2);    // 2 is a actual parameter
    return 0;
}
```

The stack does not create any space for 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, the variable in function is stored on the stack of the function being called.
More detail refer to program "Function call stack-Practice.c"

4. What are the matters 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 who is stored on stack

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

- Return value is a right value, It can not do operators operations

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

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

5. What is 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 compile stage. Which can save calling overhead
There are 2 forms of inline function

```c
// 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();
```

6. 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 funcion do not need syntax check | normally | The length of function is small and be called frequently |

# Function Call Stack

## Guidance

Advanced Programming in the UNIX Environment-> 7.6
Computer Systems A Programmer's Perspective.pdf ->3.4 3.7.1 3.7.4

1. When do local variables be allocated space and when do they be released space?
2. Why the value of function argument did not change when we change the value of formal parameters
3. How many ways are there to store stack data in computer? Which approach does Linux take?

Here is a very simple program and according assembly language. Try to understand the process of Function Call Stack via it. (Function call stack-Guidance.c)

## Practice

Suppose, The start address of main stack is 0x1000.In a computer what uses a full decrement stack

1. Draw a picture for virtual address space of Linux process
2. Write the corresponding C code next to the assembly instruction. (Function call stack-Practice.c)

   Refer to program "Function call stack-Practice.c"

3. Draw some pictures or tables to illustrate the memory layout and content of register after each assembly instruction for "Function call stack-Practice.c",

   Refer to "Function call stack-ReferenceAnswer.xlsx"

# Recursion

## Guidance

Grokking Algorithms -> chapter 3
Pointers On C.pdf -> 7.5

1. How can you crash a stack?
2. What is recursion? How to use recursion?

## Practice

1. How many phases does recursion have?

Recursive function comes in two phases:

- Winding phase, When the recursive function calls itself, and this phase ends when the condition is reached.
- Unwinding phase, Unwinding phase starts when the condition is reached, and the control returns to the original call.

2. Grokking Algorithms 3.2

- The default stack size of Linux process is 8M, the program will crash if you exceed the size

```c
#include <stdio.h>

int main(void)
{
    int arr[10 * 1024 * 1024];
    arr[1] = 1;
    return 0;
}
```

```
Terminal ×  +

root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c -O0 && ./a.out
Segmentation fault (core dumped)
root@dcs-6b203f0c-0:/workspace/CProject# ulimit -s
8192
```

- The stack size can be change via command ulimit

```c
#include <stdio.h>

int main(void)
{
    int arr[10 * 1024 * 1024];
    arr[1] = 1;
    return 0;
}
```

```
Terminal ×  +

root@dcs-6b203f0c-0:/workspace/CProject# ulimit -s 81920
root@dcs-6b203f0c-0:/workspace/CProject# ulimit -s
81920
root@dcs-6b203f0c-0:/workspace/CProject# gcc main.c -O0 && ./a.out
root@dcs-6b203f0c-0:/workspace/CProject#
```

3. Refer to the program "Recursion-Practice.c" to answer the following questions

    a. What is the difference between recursion and loop?

Fibonacci_1 will jump to the location .L2 call itself when the condition n>2 met, and the process of assigning the stack and the release stack will be repeated multiple times. And there is a need for a variety of instructions, all of which waste CPU and memory resources.

However, The fibonacci_2 logic processing focuses on the .L6 section, with only the most basic addition and subtracting operations. Compared to fibonacci_1, it can complete the functionality with a small amount of CPU resources.

    b. Why chapter 7.5.2 on "Pointers On C.pdf", Do not recommend you Compute fibonacci by recursion?

Each fibonacci calls itself two times, and its 2 children call another two times separatly. This is a disaster

Here is a table get from Dipto

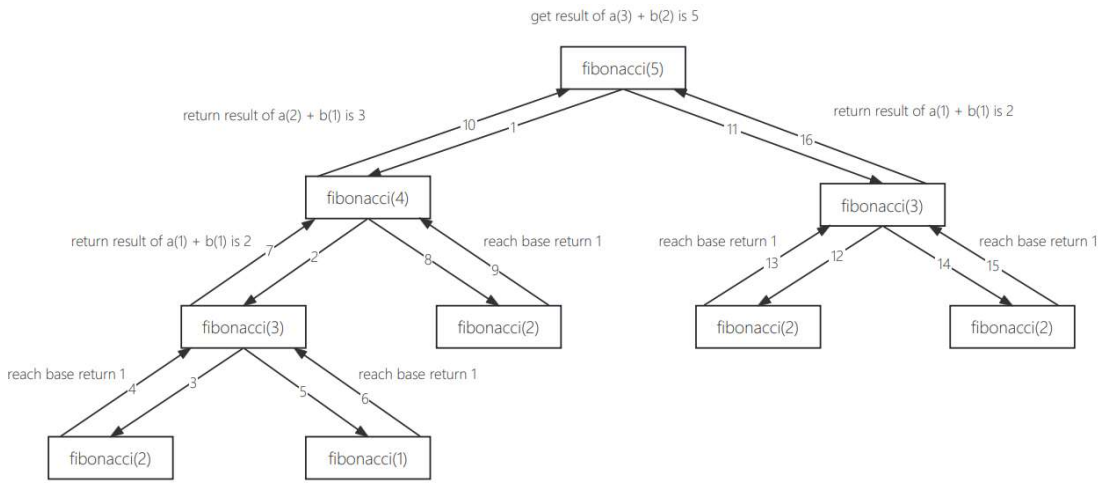
| n | No of time fibonacci(3) is called |
|---|---|
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 5 |
| 8 | 8 |
| 9 | 13 |
| **10** | **21** |
| 15 | 233 |
| **20** | **2, 584** |
| 25 | 46, 368 |
| **30** | **317, 811** |
| 35 | 3, 524, 578 |
| **40** | **39, 088, 169** |
| 45 | 433, 494, 437 |
| **50** | **4, 807, 526, 976** |

This is a description of book "Pointers On C.pdf".

Many problems are explained recursively only because it is clearer than a non-recursive explanation. However, an iterative implementation of these problems is often more efficient than a recursive implementation, even though the resulting code is less clear. When a problem is complex enough that an iterative implementation is difficult for someone else to follow, then the clarity of the recursive implementation may justify the runtime overhead that it will incur.

In general, We should avoid using recursion in the code. We only consider using recursion in one situation that is the recursive code readable can offset the efficiency loss that it brings

c. Draw a call tree when you Compute fibonacci by recursion



# Topic 5: Representation Of Data

## Binary Number

## Guidance

Digital Systems Principles and Applications.pdf 1-3 1-6 2-1 2-2 2-3 2-4 6-1
Computer Systems A Programmer's Perspective.pdf 2.1-overview 2.1.1

## Practice

1. Why do computer store data in binary and not in other bases?

The computer is composed of a large number of electronic devices, and most of the electronic devices have two stable states, such as transistor on and off, high and low voltage, magnetic and non-magnetic and so on

- Improve system stability: The use of electronic devices with fewer states can enhance the stability of the entire system
- High reliability: only two numbers 0 and 1 are used in binary, which is not easy to make mistakes when transmitting and processing, so it can ensure that the computer has high reliability
- High transmission efficiency: the binary operation rules are simple, which simplifies the structure of the arithmetic machine, and is conducive to improving the operation speed

2. What is the difference between bcd code and binary?

| BCD | Binary |
| --- | --- |
| Converts each decimal digit to a 4-digit 0/1 number | Base-2 representation of number |
| 123(decimal) = 0001 0010 0011 | 123(decimal) = 0111 1011 |
| Used for encoding/decoding according real data | used for computer store and representation |

3. How to convert between different bases? Write a program to convert different bases(binary, octal, decimal, hexadecimal)

# Encoding & Decoding

## Guidance

Digital Systems Principles and Applications.pdf 6-2 6-3 6-15
Computer Systems A Programmer's Perspective.pdf 2.2.2 2.3.2

## Practice

1. How the computer to store negative number (here is not finger to data type)?

From Rakin

**Answer:** 2's Complement format is how computer store negative number. In binary the MSB (Most Significant Bit), that is the leftmost bit is used to indicate whether the number is positive or negative. In a positive number the MSB will be 0, In a negative number the MSB will be 1.

| **0** | 1 | 0 | 1 | 1 | 0 | 1 | $= (45)_{10}$ |
|---|---|---|---|---|---|---|---|
| **Sign bit** | Magnitude in True binary | | | | | | |

| **1** | 0 | 1 | 0 | 0 | 1 | 1 | $=(-45)_{10}$ |
|---|---|---|---|---|---|---|---|
| **Sign bit** | Magnitude in 2's complement form | | | | | | |

Here the Sign bit is the MSB. For positive number the magnitude is stored in its true binary form, but for negative numbers it is stored as 2's complement form.

2. What is the difference between following 2 value in memory? If they are different point out where is it, if they are the same point out why?

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

From Liakot

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.

3. Explain to me why the result of the following program is that way?

```c
int main(void)
{
    char a = 128;
    unsigned char b = 256;

    printf("%d %d\n", a, b);    // -128 0
    return 0;
}
```

From Liakot

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.

4. What are the rules of Sign-and-Magnitude, 1's Complement and 2's Complement? Which approach the computer used? Why didn't use others?

- First of all we need know the concept of machine number and real value. Let's take 3 and -3 for example

Computers use the highest bit as a sign bit to represent positive or negative numbers

| | machine number | |
|---|---|---|
| | sign | real value |
| 3 | 0 | 0000011 |
| -3 | 1 | 0000011 |

We call this encoding method the Sign-and-Magnitude, which is the most friendly encoding method for human memory, the range is [-127, 127],It can represent 255 numbers.

The operation rule of the Sign-and-Magnitude is that if the sign bit is the same, remove the symbol, operation, and add the symbol; Different sign bits. big number - small number

The biggest problem with computer Sign-and-Magnitude is that sign bit are involved in operations and subtraction is required. The computer should be

designed as simple as possible, otherwise the basic circuitry will become very complicated.
So we need a sign bit to participate in the operation, and do not do subtraction design. The 1's Complement is designed

- We can get a code number, based on Sign-and-Magnitude, the sign bit is unchanged, and the other bits are reversed. This is 1's Complement

Let's take 1-1 for example.

```
1 - 1 = 1 + (-1) = [0000 0001]SM + [1000 0001]SM= [0000 0001]1's + [1111 1110]1's
= [1111 1111]1's = [1000 0000]SM = -0
```

The problem is sign bit are involved in operations and subtraction is required has been solved with 1's Complement.
the range is [-127, 127],It can represent 255 numbers.
Now we have 2th 0, +0 and -0.In order to save all the resource of computer the 2's Complement is designed

- We can get a code number, based on 1 's Complement, via 1's Complement + 1.This is 2's Complement.

Let's take same example 1-1

```
1-1 = 1 + (-1) = [0000 0001]SM + [1000 0001]SM = [0000 0001]2's + [1111 1111]2's
= [0000 0000]2's=[0000 0000]SM
```

We have solved the +0/-0 problem and can use the extra bit [1000 0000] to store a number smaller than -127, which is -128
Note that since -128 is actually represented using the former -0 complement, -128 has no source and inverse code representation.

5. Computer Systems A Programmer's Perspective.pdf Practice Problem 2.12

From Mormo

| $\vec{x}$ | $B2U_4(\vec{x})$ | $B2T_4(\vec{x})$ |
|---|---|---|
| 0 | 0 | 0 |
| 3 | 3 | 3 |
| 8 | 8 | -8 |
| A | 10 | -6 |
| F | 15 | -1 |

$B2U_4(\vec{x})$ Calculation:

$$B2U_4(\vec{x}) = \sum_{i=0}^{3} x_i 2^i$$

For $\vec{x}$=A; $B2U_4(1010) = 0*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 10$

$B2T_4(\vec{x})$ Calculation:

$$B2T_4(\vec{x}) = -x_3 * 2^3 + \sum_{i=0}^{2} x_i * 2^i$$

For $\vec{x}$=A; $B2T_4(1010) = -1*8 + (0*2^0 + 1*2^1 + 0*2^0) = -6$

6. Computer Systems A Programmer's Perspective.pdf practice problem 2.15

From Rakin

Here the value of k will be 3. So 2k will be 8. So (0 mod 8 = 0), (3 mod 8 = 3), (8mod8 = 0), (10 mod 8 = 2) and (15 mod 8 = 7). Which is exactly the same of unsigned truncated column. So unsigned truncated integers follows equation 2.7

| | Hex | | Unsigned | | Two's Complement | |
|---|---|---|---|---|---|---|
| Binary | Orignial | Truncated | Original | Truncated | Original | Truncated |
| 0000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0011 | 3 | 3 | 3 | 3 | 3 | 3 |
| 1000 | 8 | 0 | 8 | 0 | -8 | 0 |
| 1010 | A | 2 | 10 | 2 | -6 | 2 |
| 1111 | F | 7 | 15 | 7 | -1 | -1 |

7. What are the rules of 2's Complement operation(only addition need)?

From Liakot

Suppose our word size if w and we need to add x and y. Then the rules for 2's complement operation is -

If the value of x, y is (-2^(w-1)<= x, y <=2^(w-1)-1), then the following rules will apply for 2's compliment operation.

1. If x+y >= 2^(w-1), then this is known as positive overflow. Because, the original output is larger than maximum capacity. For this situation, we need to subtract 2^w from the original output. This also equivalent to (x+y) mod 2^w.

2. If x + y is (-2^(w-1)<= x + y <2^(w-1)), this value is in the range of the capacity. I this case, we can represent the original output in the variable.

3. If x + y < -2^(w-1), then this known as negative overflow. Because, the original output is smaller than the minimum value. We need to add 2^w with the original output.

All of these rules can be express as the following equation-

$$x +^{t}_{w} y = \begin{cases} x + y - 2^{w}, & 2^{w-1} \le x + y \quad \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} \quad \text{Normal} \\ x + y + 2^{w}, & x + y < -2^{w-1} \quad \text{Negative overflow} \end{cases}$$

Reference picture from Computer Systems: A programmer's perspective.pdf book.

| Positive overflow | Normal | Negative overflow |
|---|---|---|
| int a = 127;<br>a += 1; | int a = -123;<br>a += 1; | int a = -128;<br>a += -1; |
| 0111 1111<br>+ 0000 0001<br>= 1000 0000 = -128 | 1000 0101<br>+ 0000 0001<br>= 1000 0110 = -122 | 1000 0000<br>+ 1111 1111<br>= 0111 1111 = 127 |

# Byte Order

# Guidance

Computer Systems A Programmer's Perspective.pdf 2.1.3 2.1.4

1. big endian and little endian
2. host order and network order
3. How does the string storage on different byte order

## Practice

1. What are the rules of big endian and little endian? Illustration
2. Why do computer need different byte order?
3. How many ways to determine the byte order of your PC? Illustration
4. Computer Systems A Programmer's Perspective.pdf Practice Problem 2.5
5. Computer Systems A Programmer's Perspective.pdf Practice Problem 2.6
6. How to make you program work with both big endian and little endian?
7. To observate the start and end addresses of str1. What you can get?

```
unsigned int a = 0xFFFFFFFF;
char str1[] = "abcd";
unsigned int b = 0xFFFFFFFF;
char *p;

for (p = &b; p < &a + 1; p++)
    printf("%p %x\n", p, *p);
```

## *Data Type*

## Guidance

Computer Systems A Programmer's Perspective.pdf 2.1.2 2.2.1-2 2.2.4-6
Pointers On C.pdf 3.1.1

## Practice

1. What is expanding when store a number? When does it happen?
2. 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;
}
```

3. What is the difference between implicit type conversion and explicit type conversion?
4. What is integer promotion? In what scenarios does integer promotion occur?
5. List the rules of usual arithmetic conversion
6. When do type conversion induced by assignment. Description
7. Pointers On C.pdf 7.10 question 3
8. Pointers On C.pdf 7.10 question 6