Reference answer for exercise 1:

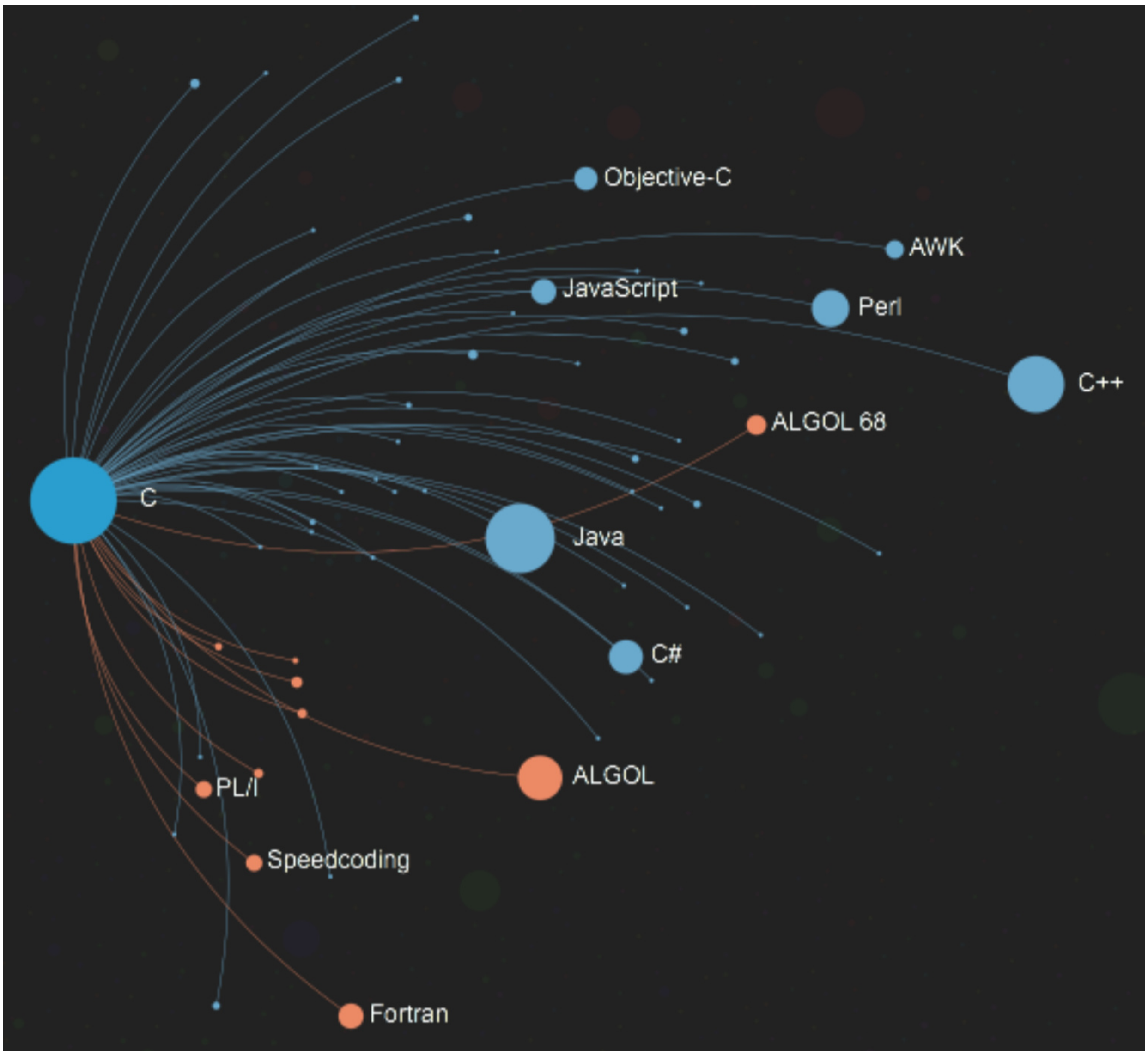
Introduction:

1. Before "C", there ever existed other languages called *A* and *B*.

   Here is a brief history about C language: In 1969, Project *Multics*, conducted by General Electric, MIT, and Bell Laboratories failed. Like other researchers, Ken Thompson shifted his interests to another OS. While waiting for approbation, he tried to transplant Game *Space Travel* to PDP-7(a minicomputer produced by Digital Equipment Corporation). Meanwhile, Thomspon was also developing a light OS for PDP-7, which was programmed by assembly language. The OS was later called *UNIX* by Brian Kernighan in 1970. Thomspon wasn't willing to encounter the difficulties appeared in Multics, so, he considered to use a language combined the advantages of the high-level language with high efficiency. After failed by Fortran, he modified *BCPL*(Basic Combined Programming Language) and created *B* Language, which is typeless language. Later Thomspon re-implemented UNIX on PDP-11 and Dennis Richard created *New B*, introducing data type and declaration. The *New B* quickly became the early *C*.

   As for *A*, it can be dated back to further earlier days. Before *BCPL*, *ALGOL* was proposed at a meeting in 1958 at ETH Zurich, called *ALGOL-58*. Later appeared *ALGOL-60*, which is considered as a beginning of Programming Language.

The relation between C and other languages can be showed in the following picture(https://exploring-data.com/vis/programming-languages-influence-network/#C):



Data type:

1. Primitive type in C:
    - `char`
    - signed:
        - `short int`
        - `int`
        - `long int`
        - `long long int` (C99)
    - unsigned:
        - `unsigned int`
        - `unsigned short int` (C90)
        - `unsigned long int` (C90)
        - `unsigned long long int` (99)
    - `_Bool` (C99, `stdbool.h`)
    - `float` , `double` , `long double`

- `_Complex`, `_Imaginary` (C99)
-

2. Code:

```c
#include <stdio.h>;
#include <limits.h>;

int main(void)
{
    printf("The minimum value of short int = %d\n", SHRT_MIN);
    printf("The maximum value of short int = %d\n", SHRT_MAX);

    printf("The minimum value of int = %d\n", INT_MIN);
    printf("The maximum value of int = %d\n", INT_MAX);

    printf("The minimum value of long int = %ld\n", LONG_MIN);
    printf("The maximum value of long int = %ld\n", LONG_MAX);

    printf("The minimum value of long long int = %lld\n", LLONG_MIN);
    printf("The maximum value of long long int = %lld\n", LLONG_MAX);

    printf("The maximum value of unsigned short int = %d\n",
USHRT_MAX);

    printf("The minimum value of unsigned int = %u\n", UINT_MAX);

    printf("The maximum value of unsigned long int = %lu\n",
ULONG_MAX);

    printf("The maximum value of unsigned long long int = %llu\n",
ULLONG_MAX);

    printf("The minimum value of char = %d\n", CHAR_MIN);
    printf("The maximum value of char = %d\n", CHAR_MAX);

    printf("The maximum value of unsigend char = %d\n", UCHAR_MAX);

    return 0;
}
```

Output:

- Xcode 7.3.1:

```
The minimum value of short int = -32768
The maximum value of short int = 32767
The minimum value of int = -2147483648
The maximum value of int = 2147483647
The minimum value of long int = -9223372036854775808
The maximum value of long int = 9223372036854775807
The minimum value of long long int = -9223372036854775808
The maximum value of long long int = 9223372036854775807
The maximum value of unsigned short int = 65535
The maximum value of unsigned int = 4294967295
The maximum value of unsigned long int = 18446744073709551615
The maximum value of unsigned long long int = 18446744073709551615
The minimum value of char = -128
The maximum value of char = 127
The maximum value of unsigend char = 255
```

- Windows 8.1 上的 Codeblocks 13.12:

```
The minimum value of short int = -32768
The maximum value of short int = 32767
The minimum value of int = -2147483648
The maximum value of int = 2147483647
The minimum value of long int = -2147483648
The maximum value of long int = 2147483647
The minimum value of long long int = -9223372036854775808
The maximum value of long long int = 9223372036854775807
The maximum value of unsigned short int = 65535
The maximum value of unsigned int = 4294967295
The maximum value of unsigned long int = 4294967295
The maximum value of unsigned long long int = 18446744073709551615
The minimum value of char = -128
The maximum value of char = 127
The maximum value of unsigend char = 255
```

- Ubuntu Kylin 14.04 的 gcc 4.8.4：

```
The minimum value of short int = -32768
The maximum value of short int = 32767
The minimum value of int = -2147483648
The maximum value of int = 2147483647
The minimum value of long int = -9223372036854775808
The maximum value of long int = 9223372036854775807
The minimum value of long long int = -9223372036854775808
The maximum value of long long int = 9223372036854775807
The maximum value of unsigned short int = 65535
The maximum value of unsigned int = 4294967295
The maximum value of unsigned long int = 18446744073709551615
The maximum value of unsigned long long int = 18446744073709551615
The minimum value of char = -128
The maximum value of char = 127
The maximum value of unsigend char = 255
```

Result: On different OS and compilers, the value range of some types varies.

Explanation: The C language specification doesn't define any exact range for each data type. It only defines a *minimum* value that a particular type should be able to hold. More details can be referred the C standards.

3. In switch/case block, integer and character can be keywords.

   Code:

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

enum color_type
{
    GREEN,
    BLUE,
    YELLOW
};

int main(int argc, char* argv[])
{
    short kw1 = 10;
    int kw2 = 1;
    long kw3 = 1131231;
    char kw4 = 'a';
    float kw5 = 1.33534;
    double kw6 = 1.323436461;
    char kw7[3] = "ab";
    enum color_type kw8;
    kw8 = GREEN;

    switch (kw1)
    {
        case 10:
        {
            printf("short is OK\n");
            break;
        }
        default:
        {
            printf("Cannot be short!\n");
        }
    }

    switch (kw2)
    {
        case 1:
        {
            printf("int is OK\n");
            break;
```

```c
        }
        default:
        {
            printf("Cannot be int\n");
        }
    }

    switch (kw3)
        {
        case 1131231:
        {
            printf("long is OK\n");
            break;
        }
        default:
        {
            printf("Cannot be long\n");
        }
    }

    switch (kw4)
    {
        case 'a':
        {
            printf("char is OK\n");
            break;
        }
        default:
        {
            printf("Cannot be char\n");
        }
    }

    // kw5, kw6
    // switch (kw5)
    // {
    //     case 1.33534: // syntax error, so as double
    //     {
    //         printf("float is OK\n");
    //         break;
    //     }
    //     default:
    //     {
    //         printf("Cannot be float\n");
    //     }
    // }

    // switch (kw7)  // syntax error
    // {
```

```
    //     case "ab":
    //     {
    //         printf("arrray is OK\n");
    //         break;
    //     }
    //     default:
    //     {
    //         printf("Cannot be array\n");
    //     }
    // }

    switch (kw8)
    {
        case GREEN:
        {
            printf("enum is OK\n");
            break;
        }
        default:
        {
            printf("Cannot be enum\n");
        }
    }
    return 0;
}
```

We can see from the code, int(short, long) and char can be used as keywords, as well as enum. Actually, the reason behind the code is that in C, char is also integer in memory(such as ASCII). Enum is also the integer. See the code below:

```
int A = 65;
printf("In char, it is %c\nIn int, it is %d\n", A, A);
```

The output is:

```
In char, it is A
In int, it is 65
```

If referred in ASCII, we can know that the integer value of char 'A' is 65. The output is different since we use different format placeholder(`%d` for int, `%c` for char).

You can try to print enum for what will happen.

4. (Tested on Mac with clang-900.0.39.2) The size of the data type:
    - `char`: 1 bytes
    - `long`: 8 bytes
    - `double`: 8 bytes

While empty struct takes no memory space, the size of the given struct `STU` is 51 in byte via calculation. However, the real size in memory is 64. This difference is caused by the mechanism called **word align** (More details: Alignment in C). The variable `name` is aligned with 6 bytes and `sex` is laid after `student_id` with 7 bytes aligned.

Please run the following code to see the exact address and try to figure out what happen in memory.

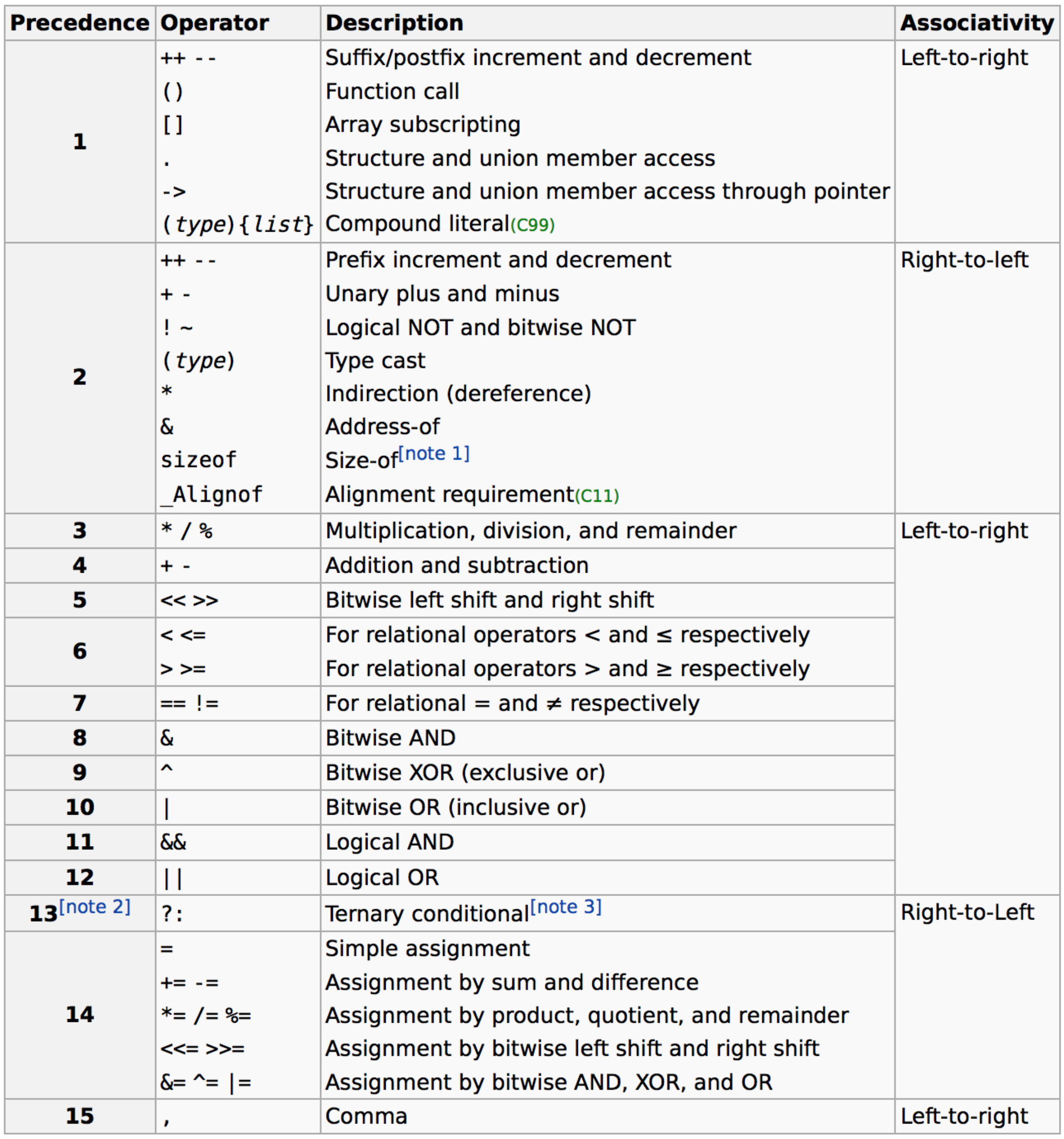
```c
#include <stdio.h>

typedef struct student
{
    char name[10]; //10 bytes
    long student_id; //8 bytes
    char sex; //1 byte
    double score[4]; //32 bytes
} STU;

int main ()
{
    STU b;
    printf("%ld\n", sizeof(b));
    printf("%p\n", &b.name)
    printf("%p\n", &(b.student_id));
    printf("%p\n", &(b.sex));
    printf("%p\n", &(b.score));
    return 0;
}
```

Operators

1. Operators:



Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(`*type*`){`*list*`}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of[note 1]	
	`_Alignof`	Alignment requirement(C11)	
3	`* / %`	Multiplication, division, and remainder	Left-to-right
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13[note 2]	`?:`	Ternary conditional[note 3]	Right-to-Left
14	`=`	Simple assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Comma	Left-to-right

Example code:

```
int a = 1, b = 2, c = 12, d = 2, e = 5, f = 2;
a = b += c++ - d + --e/-f; // (a = (b += (((c++) − d) + ((−−e)/(−f ))))) 
=> a = 10, b = 10
-a + (c + b * (c + a) / c - b / a) + a - b / 2; // (((-a) + ((c + ((b * (c
+ a)) / c)) - (b / a))) - (b/2)) => 11
```

2. The difference between `i++` and `++i` is the order of increment. `i++` will perform increment after `i` is used while `++i` does before. In memory, `i++` will require a temporary space for storing `i` and the increase 1 on the temporary value before re-assigning to the original value. Compared to `i++`, `++i` just increase 1 on itself. Otherwise, either `i++` and `++i` result in an increment on `i`.
3. Code:

```
void swap(int *a, int*b)
{
    *a = (*a) ^ (*b);
    *b = (*a) ^ (*b);
    *a = (*a) ^ (*b);
}
```

Explanation: This is very interesting for bit operators. The operator `^` conduct **exclusive or**(1 ^ 0 = 1, 1 ^ 1 = 0, 0 ^ 0 = 0) on two bits. Since there are only two values(0 and 1) for a bit, so the difference between the two values are decided by the different bits. `^` can flip all different bits which result in a swap for two values.

Aarray and Pointer

1. `char*` is a pointer and `char []` indicates an array. See codes below:

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

   int main(int argc, char* argv[])
   {
       char* pointer = "Hello";
       char array[] = "world";
       printf("%s", pointer);
       printf(" %s\n", array);
       printf("%c", pointer[0]);
       printf(" %c\n", array[0]);

       printf("%s", pointer++);
       // printf(" %s\n", array++); // syntax error

       // *pointer = 'd'; // syntax error
       array[1] = 'd';
       printf("%s", pointer);
       printf(" %s\n", array);

       printf("%ld", sizeof(pointer)); //size of pointer
       printf(" %ld\n", sizeof(array)); // size of the array
       return 0;
   }
   ```

   PS: Here the `char []` by itself has no meaning except in prototype declaration like in function prototype.

   `pointer` is a constant, static array, which cannot be modified.

   `array` is variable and its elements can be changed. The length of the array, though not explicitly declared, is implicitly initialize when the string is assigned to.

2. The operation which can be conducted on pointers includes **addition**, **subtraction**, **increment**, **decrement**.

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

int main(int argc, char const *argv[])
{
    char str[] = "This is a string.";
    char* p = &(str[0]);
    printf("%c\n", *p);
    printf("%c\n", *(++p));
    printf("%c\n", *(p++));
    printf("%c\n", *p);
    printf("%c\n", *(--p));
    printf("%c\n", *(p+7));
    return 0;
}
```

Others

1. There some types of writing a main function.

```
// the following three are fine in anywhere.
int main()
int main(void)
int main(int argc, char* argv[])
int main(int argc, char** argv)
int main(int argc, char** argv, char **envp) // Unix

void main() // some embedded system, some microcontrollers
```

The main with return values are strongly recommended and either one of them is compiled normally. While the main without return values are used when no return is expected. This type in some cases can be compiled but most modern compilers do not support it.

Example error with void main() (Tested on Mac with clang-900.0.39.2):

```
pointer_operation.c:4:1: warning: return type of 'main' is not 'int'
[-Wmain-return-type]
void main(int argc, char const *argv[], char **envp)
^
pointer_operation.c:4:1: note: change return type to 'int'
void main(int argc, char const *argv[], char **envp)
^~~~
```

For all homework, please use the types with return values.

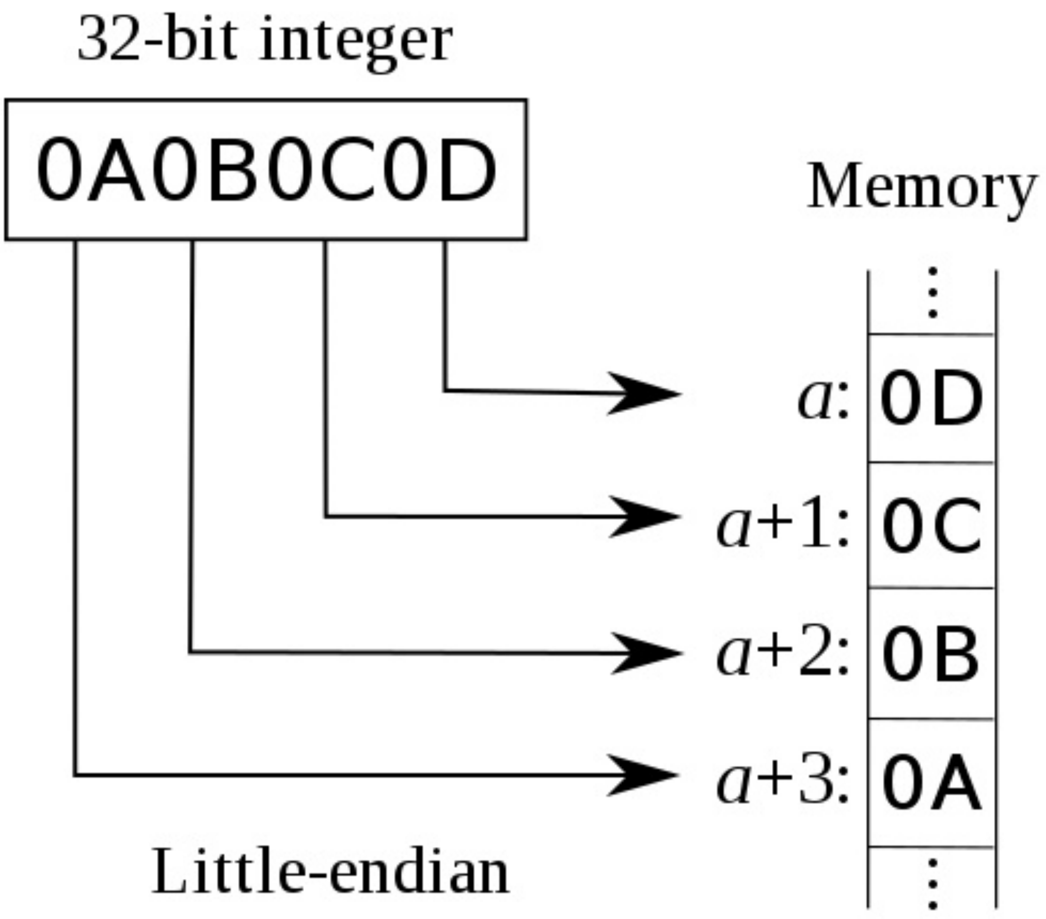
3. Code:

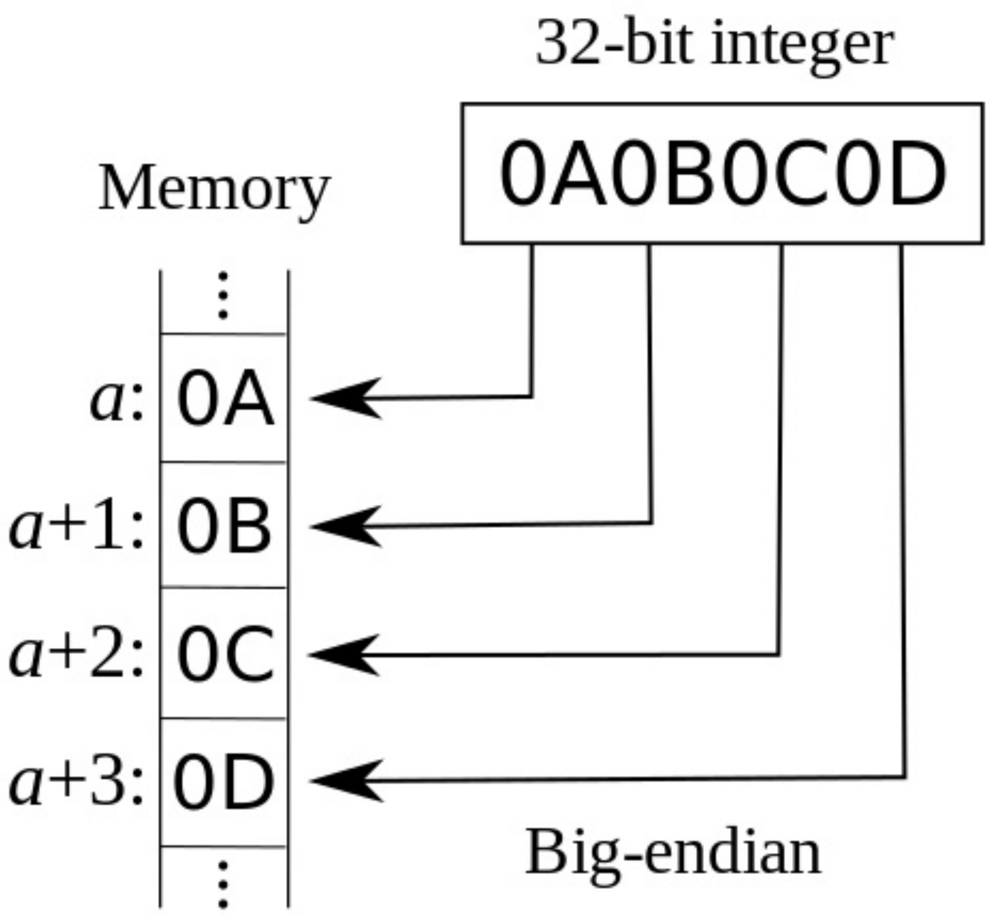
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x = 1;
    char *y = (char *)&x;
    printf("%c\n", *y + 48);
    // or
    unsigned int x = 1;
    printf("%d", (int)(((char *)&x)[0]));
    return 0;
}
```

If it is little endien, the printed result will be 1, otherwise it will print 0.

Explanation: Little endian and big endian are two types which represent the order of the bytes in memory. Little endian means that bytes are stored from larger memory address to smaller address while big endian does reverse.





The code stores a integer and then convert the `int` data to `char` which enable the access of byte. The use `&` to test the endian.

3. Code:

```
char word[10] = "where";
printf("%lu\n", sizeof(word)); // 10
printf("%lu\n", strlen(word)); // 5
```

Explanation: `strlen` is declared in `string.h` and count the length of the string when running, the result will be returned when encounters `NULL`. `sizeof` count the length of the memory to store the data. Since we explicitly declared an array of `char` of length 10, the size of the data is 10.

4. - strcmp`

```
int strcmp(const char *s1, const char *s2)
{
    const char* p = s1;
    const char* q = s2;
    while (*p == *q)
    {
        if (*p == '\0')
        {
            return 0;
        }
        ++p;
        ++q;
    }
    return *p - *q;
}
```

- `strcat`

```
char* strcat(char* dst, const char *src)
{
    char* p = dst;
    while(*p != '\0')
    {
        ++p;
    }
    while((*p++ = *src++));
    return dst;
}
```

- `strcpy`

```
char* stpcpy(char *dst, const char *src)
{
    while((*dst = *src))
    {
        dst++;
        src++;
    }
    return dst;
}
```

5. Code:

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

int main(void)
{
    char input[1024];

    scanf("%s", input);
    printf("Input: %s\n", input);

    gets(input);
    puts(input);

    fgets(input, sizeof(input), stdin);
    fputs(input, stdout);

    return 0;
}
```

function	ends when reads ...	safety	comment
scanf	' ', '\n', '\r', '\v', '\t'	unsafe	no limit in length, end character will be left in buffer
gets	'\n'	unsafe	no limit in length, end character will be read and dropped
fgets	'\n'	safe	limit length, end character will be read and appended to result

Please use scanf with printf, gets with puts and fget with fputs, otherwise, the result might be what you expected.

6. **goto** is a keyword which also exists in FORTRAN and BASIC. The use of `goto` will lead to spaghetti code sometimes. Kernighan and Ritchie suggested less or none use of it. In a paper written by Dijkstra in 1968, *Goto Considered Harmful*, he also presented the harm of using `goto`. During that time, many programmers used goto to replace the structured programming. Later these days, we rarely see it in code. However, like the discussion between kernel programmers, `goto` is usually used in kernel programming to keep the code clean and the flow straightforward. Here is an example from stackoverflow: if you have to handle some errors by conditions, the structured programming will be coded as

```
if (do_something() != ERR) {
    if (do_something2() != ERR) {
        if (do_something3() != ERR) {
            if (do_something4() != ERR) {
                ...
```

Compared to the `goto` version, the code will be

```
f (do_something() == ERR)  // Straight line
    goto error;             // |
if (do_something2() == ERR) // |
    goto error;             // |
if (do_something3() == ERR) // |
    goto error;             // V
if (do_something4() == ERR) // emphasizes normal control flow
    goto error;
```

which gives us different feelings in typeset. Notes that `goto`, if needed, should be used in forward instead of backward. And the case we discussed here is for kernel program, which has a huge difference compared to non-kernel programs.

7. `malloc` and `free` is common in `C`, while `new` and `delete` is provided in `C++`.

allocation & release	**type**	**allocated from**	**return value**	**usage**	**comment**
malloc/free	function	heap	void *, NULL if failed	size should be specified	realloacting larger chunk of memory
new/delete	operator	free store	full typed pointer, throw on failure	called with type-id and compiler calculates the size	new memory can be added