

C_Beginning

Editor:YiMingLi

C语言中常见的未定义行为 (C11)

什么叫做未定义

未定义行为(undefined behavior)(简称 UB)指编译器允许编译,但是语言标准中没有定义的行为.这种现象的出现并不是语言标准不完善,而是因为这种行为在编译时无法检查其错误或者受限于具体的 cpu 指令以及操作系统优化等使其会有在编译时不可预知的运行结果.

最简单的就是数组越界,这种错误在 c 语言编译时无法检查,而 c 标准也没有定义越界后会访问到什么.

未定义行为在不同编译器上或者不同操作系统或者不同架构 cpu 上会产生不同的结果,所以在任何代码中都应当避免未定义行为.

常见的未定义行为

数组越界(array index outof bounds)

```
void func()
{
    int array[5] = {0};
    printf("%d", array[5]);
}
```

而至于 `array[5]` 到底会访问到什么取决于操作系统和编译器.例如 windows 平台的 msvc 编译器的 debug 模式会将所有未初始化内存都用特定值来充填,在未初始化内存的边缘会用另一个值来充填(详见烫烫烫屯屯问题).但是在 linux 平台的 gcc 中此时只会访问到一个普通的未初始化内存,因此其值是不可预期的.

和数组越界一样的还存在**定义字符串的时候没有存在结束符**的时候,在打印的时候会一直打印,直到看到结束符,所以如果不服你在结束符的时候就会产生越界地情况.

修改字符串字面量

字面量(literal)是写在源代码中的表示固定值的符号(token).平时俗称的硬编码(hard coding)大多数时候就是在说字面量(有时指常量).举个例子

```
char *string = "Hello";
string[0] = 'h';
```

这个地方修改了未指定区域的内存,是未定义的行为.

除以零

```
#include <stdio.h>

int main()
{
    int a = 1, b = 0;
    int result = a / b;
    printf("%d", result);
    return 0;
}
```

有返回值的函数没有return/没有返回值的函数返回了一个值

```
#include <stdio.h>

int func() {

int main() {
    printf("Hello\n");
    printf("%d\n", func());
    return 0;
}
```

```
void no_return()
{
    return 10; //UB
}
```

存在副作用的子表达式

副作用(side effect)的意思是函数会对其调用者的上下文中的某些东西产生改变, 比如函数传入值的表达式本身会改变其调用者所在的上下文中的变量.

```
#include<stdio.h>
int main()
{
    int i = 1;
    printf("%d %d ", i++, i);
    retrun 0;
}
```

(输出 "1 2")(以 linux 平台的 gcc 编译器为例, 下同)

然后开始盲目分析: "i++" 是用完了再加, 所以输出 1, 而到了第二个参数时 i 已经被加过了, 所以输出 2. 谭浩强直呼闭门大弟子.

那我们来试试给第二个 i 再加 1

```
printf("%d %d\n", i++, i + 1);
```

此时同样输出 1 和 2

```
printf("%d %d\n", i++, i++);
```

这样甚至输出 2 和 1

```
printf("%d %d\n", i++, ++i);
```

这更是输出了不可理喻的 2 和 3

简单地说, 如果一个很长的表达式有多个子表达式, 并且子表达式存在副作用, 那么其运行顺序就是未定义的. 到底是哪个先运行取决于编译器. 所以不要分析了, 这是没有道理的.

```
int array[5] = {0};  
int i = 1;  
array[i] = i++;
```

像这种因为先运行哪一个函数后运行那个函数而导致结果不同的情况一般都是UD

解引用空指针或者解引用野指针 (or 悬垂指针)

当我们尝试对空指针进行解引用操作的时候编译器无法确定要访问的内存空间中储存的内容。例如:

```
int *ptr = NULL;  
printf("%d\n", *ptr); //UB  
int *ptr_2;  
printf("%d\n", (int)*ptr_2); //UB
```

指针转换成为 int 类型是可以的, 但是如果是 NULL 转换成为整数, 那么这个指针没有明确的规定

未初始化的局部变量

当我们使用未初始化的局部变量时, 其值是未定义的, 因此会导致未定义行为。例如:

```
int x;
printf("%d\n", x);
```

signed整数的溢出

```
int a = INT_MAX; // 假设 INT_MAX 是 2147483647
a++;           // 未定义行为
```

```
unsigned int b = UINT_MAX; // 假设 UINT_MAX 是 4294967295
b++;                  // b 变为 0
```

- 当使用`unsigned`的情况的时候，如果你超过范围了就会从最小的开始算起。

错误的类型转换以及错误的格式输出的匹配

```
int *ptr = (int *)malloc(sizeof(int));
float *fptr = (float *)ptr; // 错误的类型转换，结果未定义
```

```
int i = 42;
float *fp = &i;
++*fp; // UB; this is NOT equivalent to ++i.
```

什么时候的指针转换是合法的？

- 通过`void`类型指针搭桥完成的操作时合法的。
- 如果`T2`是与`T1`类型加上`const`, `volatile`或`restrict`限定的版本，那么转换时合法的。
- 如果`T2`是`T1`先容类型的有符号或者是无符号的版本，转换也是合法的。
- 如果`T2`是`char`, `signed char`, `unsigned char`类型的时候，也可以合法的转化。

函数参数数量并不匹配

调用函数时提供的参数数量与函数定义不匹配，如`printf("%s %d", "Name");`

练习 (From SHTU)

- 假设`int`类型是32位的。选择设计未定义行为的代码片段

```
unsigned uval = -111;
printf("%u%%\n", uval);
```

- 这个是正确的，因为这是`unsigned`的内容所以不会超过循环而是从头开始。

```
int x = 96;
printf("%f\n", x/100);
```

- 这里是未定义行为，这里的`%f`适用于`float`的数值，而这里属于错误匹配格式化输出符。

```
int i = 42;
printf("%d%d\n", ++i, i);
```

- 这是经典的副作用的未定义的情况，这里的`i`和`++i`无法确定哪一个会先被调用

```
int helper(int value) {
    printf("value is %d\n", value);
}
int main(void) {
    int x = helper(42);
    int y = helper(x);
    printf("%d\n", x + y);
}
```

- 这里涉及设计了没有返回值的情况，也是未定义行为

```
int random(void) {
    int x;
    return x;
}
int main(void) {
    printf("%d\n", random());
}
```

- 这里涉及没有定义自变量的情况

```
double lim(int condition, double formula){
    double d = 0.000618, dx = 1.0;
    return condition ? 0 : d*formula/dx;
}
int main(void){
    int _x = 1, x = 4399, inf = 2147483647;
    double e = lim(_x --> inf, (1 + 1/x)^x);
}
```

- 这并不是一个未定义的代码，而是一个语法错误的代码

了解scanf()

- `scanf("%d %d", &a, &b);` 和 `scanf("%d%d", &a, &b);` 之间到底是什么关系?
 1. 在 C 中 ' ', '\n', '\f', '\t', '\v' 这几个都叫做空白字符，而在 `scanf` 中的 `%d` 和空白字符可以跳过任意的空白字符
 2. 当且仅当遇到下一个非空字符的时候才会去对应到下一个值，或推出退出语句
- `scanf()` 的返回值
 - 如果 `scanf()` 返回值是合法的，`scanf()` 的返回值和预期输入数量匹配。
 - 如果 `scanf()` 返回值是非法的，`scanf()` 的返回值是零。示例

```
int num;
float fnum;
int result = scanf("%d %f", &num, &fnum);
if (result == 2) {
    // 输入合法
} else {
    // 输入不合法
}
```

```
int num;
int result = scanf("%d", &num);
if (result == EOF) {
    // 输入结束或错误
}
```

数据类型

不同类型的字节大小

数据类型	32位	64位	说明
<code>char</code>	1	1	始终为 1 字节 ，表示单个字符。
<code>short int</code>	2	2	通常为 2 字节，表示短整数。
<code>int</code>	4	4	通常为 4 字节，表示整数。
<code>long int</code>	4	8	在 32 位系统中通常为 4 字节，在 64 位系统中通常为 8 字节，表示长整数。
<code>long long int</code>	8	8	始终为 8 字节，表示长长整数。
<code>float</code>	4	4	始终为 4 字节，表示单精度浮点数。
<code>double</code>	8	8	始终为 8 字节，表示双精度浮点数。
<code>long double</code>	12 或 16	16	在 32 位系统中通常为 12 或 16 字节，在 64 位系统中通常为 16 字节，表示扩展精度浮点数。

数据类型	32位	64位	说明
unsigned char	1	1	始终为 1 字节，表示无符号字符。
unsigned short	2	2	通常为 2 字节，表示无符号短整数。
unsigned int	4	4	通常为 4 字节，表示无符号整数。
unsigned long	4	8	在 32 位系统中通常为 4 字节，在 64 位系统中通常为 8 字节，表示无符号长整数。
unsigned long long	8	8	始终为 8 字节，表示无符号长长整数。
void*	4	8	在 32 位系统中通常为 4 字节，在 64 位系统中通常为 8 字节，表示指针。
_Bool	1	1	始终为 1 字节，表示布尔值（0 或 1）。
enum	通常与 int 相同	通常与 int 相同	枚举类型的大小通常与 int 类型相同。

不同类型的格式转换符（占位符）

数据类型	printf 占位符	scanf 占位符	示例值	示例代码 (printf)	示例代码 (scanf)
char	%c	%c	'A'	printf("%c", 'A');	scanf("%c", &ch);
int	%d 或 %i	%d 或 %i	42	printf("%d", 42);	scanf("%d", &num);
unsigned int	%u	%u	42	printf("%u", 42);	scanf("%u", &unum);
short int	%hd	%hd	42	printf("%hd", (short)42);	scanf("%hd", &snum);
unsigned short	%hu	%hu	42	printf("%hu", (unsigned short)42);	scanf("%hu", &usnum);
long int	%ld	%ld	42L	printf("%ld", 42L);	scanf("%ld", &lnum);
unsigned long	%lu	%lu	42L	printf("%lu", 42L);	scanf("%lu", &ulnum);
long long int	%lld	%lld	42LL	printf("%lld", 42LL);	scanf("%lld", &llnum);

数据类型	printf 占位符	scanf 占位符	示例值	示例代码 (printf)	示例代码 (scanf)
unsigned long long	%llu	%llu	42LL	printf("%llu", 42LL);	scanf("%llu", &ullnum);
float	%f	%f	3.14f	printf("%f", 3.14f);	scanf("%f", &fnum);
double	%f 或 %lf	%lf	3.14	printf("%f", 3.14);	scanf("%lf", &dnum);
long double	%Lf	%Lf	3.14L	printf("%Lf", 3.14L);	scanf("%Lf", &ldnum);
char[] 或 char*	%s	%s	"Hello"	printf("%s", "Hello");	scanf("%s", str);
void*	%p	N/A	&var	printf("%p", &var);	N/A
_Bool	%d	%d	true 或 false	printf("%d", true);	scanf("%d", &boolvar);

C语言逻辑运算结果类型及与int的关系

- C语言逻辑运算结果类型及与int的关系: 在C语言中, 逻辑运算的结果类型和值有明确的定义。即使在C99标准引入布尔类型后, 逻辑运算的结果仍然可以被看作是int类型。以下是详细解释。
 1. C99标准引入的布尔类型
 2. 逻辑运算的结果类型
 3. 逻辑运算结果的值 逻辑运算的结果值仍然是0或1:
- C99标准引入了_Bool类型, 表示布尔值 (true或false), 并通过stdbool.h头文件提供了bool类型别名, 以及true和false宏定义。例如:

```
#include <stdbool.h>
bool flag = true;
```

- _Bool类型是一个独立的类型, 其大小通常为1字节, 且只有两个可能的值: true (非零值) false (零值)
- 尽管引入了布尔类型, 但C语言的逻辑运算 (&&、 ||、 !) 的结果类型仍然是int, 而不是_Bool。原因如下:
 - 历史兼容性: C语言的逻辑运算结果一直是int类型, 为了保持与早期代码的兼容性, C99标准没有改变这一行为。
 - 隐式转换: 在C语言中, _Bool类型可以隐式转换为int类型, 反之亦然。因此, 即使逻辑运算的结果是int类型, 它也可以被用在布尔上下文中。
- 逻辑运算结果的值 逻辑运算的结果值仍然是0或1:
 - 0表示false (假)。

- 1表示true（真）。

这意味着，即使你使用bool类型来存储逻辑运算的结果，它也会被隐式转换为int类型。例如：

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

int main() {
    bool result = (5 > 3); // 逻辑运算结果为 1 (true)
    printf("result = %d\n", result); // 输出 1
    printf("sizeof(result) = %zu\n", sizeof(result)); // 输出 1 (_Bool类型大小)

    int int_result = (5 > 3); // 逻辑运算结果为 1 (int类型)
    printf("int_result = %d\n", int_result); // 输出 1
    printf("sizeof(int_result) = %zu\n", sizeof(int_result)); // 输出 4 (int类型大小)

    return 0;
}
```

4. 为什么可以将结果看作int类型

- 类型兼容性：_Bool类型和int类型之间可以隐式转换。_Bool类型在存储时通常会被提升为int类型，因此逻辑运算的结果（int类型）可以直接赋值给_Bool类型变量。
- 值兼容性：逻辑运算的结果只有0和1，这两个值在int类型和_Bool类型中都具有相同的语义。

计算时的隐式转换

- char和short类型在运算的时候发生隐式转换为int。但是只会在计算或者比较大小的时候发生转变，在计算完成之后就还是原来的类型。

```
#include <stdio.h>

int main() {
    char a = 5;
    char b = 10;
    char c = a + b; // 先变成int，再变成char

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

```
#include <stdio.h>

int main() {
    short a = 5;
    short b = 10;
    short c = a + b; // 会先转换为int然后再变成short
```

```

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

```

未定义的情况

1. 超过范围的情况：例如int超过范围
2. 未使用正确的占位符
3. `i++ + ++i`的情况

```

#include <stdio.h>

int main()
{
    int a = 64;
    char b = 64;
    long long c = 64LL;
    float d = 64.00;
    double e = 64.00;
    unsigned int f = 64;
    short g = 64;
    printf("%f,%lld,%c,%hd,%u\n", a, a, a, a, a);
    printf("%f,%lld,%d,%hd,%u\n", b, b, b, b, b);
    printf("%f,%d,%c,%hd,%u\n", c, c, c, c, c);
    printf("%d,%lld,%c,%hd,%u\n", d, d, d, d, d);
    printf("%d,%lld,%c,%hd,%u\n", e, e, e, e, e);
}

```

```

0.000000,64,@,64,64
0.000000,64,64,64,64
0.000000,64,@,64,64
0,4634204016564240384,,0,0
0,4634204016564240384,,0,0

```

运算符

运算符的优先级和结合性

以下是C语言中不同运算符的优先级和结合性：

优先级	运算符	描述	结合性
1	()	圆括号	从左到右
2	[]	数组下标	从左到右
2	.	结构体成员访问	从左到右

优先级	运算符	描述	结合性
2	<code>-></code>	结构体指针成员访问	从左到右
2	<code>++</code>	后置递增	从左到右
2	<code>--</code>	后置递减	从左到右
3	<code>++</code>	前置递增	从右到左
3	<code>--</code>	前置递减	从右到左
3	<code>+</code>	一元加	从右到左
3	<code>-</code>	一元减	从右到左
3	<code>!</code>	逻辑非	从右到左
3	<code>~</code>	按位取反	从右到左
3	<code>*</code>	指针解引用	从右到左
3	<code>&</code>	取地址	从右到左
3	<code>sizeof</code>	求大小	从右到左
3	<code>(type)</code>	类型转换	从右到左
4	<code>*</code>	乘法	从左到右
4	<code>/</code>	除法	从左到右
4	<code>%</code>	取模	从左到右
5	<code>+</code>	加法	从左到右
5	<code>-</code>	减法	从左到右
6	<code><<</code>	左移	从左到右
6	<code>>></code>	右移	从左到右
7	<code><</code>	小于	从左到右
7	<code><=</code>	小于等于	从左到右
7	<code>></code>	大于	从左到右
7	<code>>=</code>	大于等于	从左到右
8	<code>==</code>	等于	从左到右
8	<code>!=</code>	不等于	从左到右
9	<code>&</code>	按位与	从左到右
10	<code>^</code>	按位异或	从左到右
11	<code>\ </code>	按位或	从左到右
12	<code>&&</code>	逻辑与	从左到右

优先级	运算符	描述	结合性
13	<code>\ \ </code>	逻辑或	从左到右
14	<code>? :</code>	条件运算符	从右到左
15	<code>=</code>	赋值	从右到左
15	<code>+ = - = * = / = % =</code>	复合赋值运算符	从右到左
15	<code><<= >>= &= ^= \ =</code>	复合赋值运算符	从右到左
16	<code>,</code>	逗号运算符	从左到右

一个使用位运算符的例子——转换二进制

```
#include <stdio.h>
#define SIZEOF_UINT 32
void trans(unsigned x, char **s)
{
    if (x > 1)
    {
        trans(x >> 1, s);
        (*s)++;
    }
    *s = (x & 1u) + '0';
}
int main(void)
{
    char str[SIZEOF_UINT + 1] = {'\0'};
    char *ptr = str;
    trans(148, &ptr);
    printf("%s", str);
    return 0;
}
```

算数运算与算数运算的左连接问题

```
#include <stdio.h>

int main()
{
    int ival = 100000;
    long long llval = ival;
    int res1 = ival * ival;
    long long res2 = ival * ival;//运算结果是int再显示转换为long long还是会溢出
    long long res3 = llval * ival;//在运算的时候ival隐式转换成long long在进行计算
    long long res4 = llval * ival * ival;//注意是从左向右的方式进行两个ival依次被转换
    //成long long类型
    printf("%d\n%lld\n%lld\n%lld\n", res1, res2, res3, res4);
    /*
```

```
1410065408  
1410065408  
100000000000  
1000000000000000  
*/  
}
```

赋值运算的右连接问题

```
#include <stdio.h>  
  
int main()  
{  
    int a = 0, b = 1, c = 2;  
    a = b = c;  
    printf("%d %d %d", a, b, c); // 2 2 2  
    // 赋值运算从右往左进行  
}
```

未定义的函数调用顺序

```
#include <stdio.h>  
int func1(void)  
{  
    printf("Func1 is called\n");  
    return 2;  
}  
int func2(void)  
{  
    printf("Func2 is called\n");  
    return 3;  
}  
int func3(void)  
{  
    printf("Func3 is called\n");  
    return 4;  
}  
int main()  
{  
    int c = func1() + (func2() * func3());  
    return 0;  
}
```

```
Func1 is called  
Func2 is called  
Func3 is called
```

- Operator precedence does not determine evaluation order:`f() + g() * h()` is interpreted as `f() + (g() * h())`, but the order in which `f,g,h` are called is unspecified.
- 也就是说在对于不同函数的调用`a() + b() + c()`中，由于operator+的原因从左到右的结合性分析成为`(a() + b()) + c()`，但是在运行时可以首先，最后或者在中间运行`c()`
- 个人的情况，其实我觉得函数的调用应该还是按照从左往右的顺序进行，但是如果是遇到逻辑判断则会出现短路现象（只要能够判断是否是真或者是假就会跳过相关的函数的调用）

static local变量和global变量

- `static local`变量表示的是在局部声明的整体变量，他们和`global`变量一样如果没有进行初始化就会被默认为初始化为0（不同类型的0模式，同样的如果是指针都是表示的是空指针）
- 同样的`global`变量，在全局作用域外部定义的变量，这一些变量在整个程序中都可以使用
- `static local`和`global`变量的声明都是在整个程序开始运行的时候进行声明的。同时也是在整个程序结束之后在消除的。
- 例子：使用全局变量完成的只能调用一次的函数的书写

```
bool start_game_called = false;
void start_game(Player *p1, Player *p2, int difficulty, GameWorld *world) {
    if (start_game_called)
        report_an_error("You cannot start the game twice!");
    start_game_called = true;
    // ...
}
```

指针

指针的定义

```
int a = 0;
int *ptr1 = &a;
int* ptr1 = &a; //两种写法都是正确的
int      *  ptr4 = &a; //空格不影响
int*ptr5 = &a; //没有空格也可以
int *ptr2 = 0; //空指针定义
int *ptr3 = NULL; //空指针定义
int *ptr6 = &2; //错误的定义方式
int *ptr7 = &(2*a); //错误的定义方式
```

这里的指针指向的内容必须要是一个左值（表示能够写在左侧的值）`int c = a+b`这里的`c`就是左值。

空指针是正确的但是空指针不能够解引用（这样也就导致了空指针是不能够赋值的）打印空指针的内容可以看到`0000000000000000`,因此为了避免空指针的使用，我们常常这么写：

```
if (ptr != NULL && *ptr == 42) { /* ... */ }
```

这样如果指针式空的，那么后面解引用的过程就不会发生

- 野指针：野指针表示的就是没有初始值的指针，这一些指针因为不知道具体值以及地址所以几乎不能够使用，这也是未定义的行为中的一种。
- 悬垂指针：表示的是已经释放内存地址的指针，同样的这个也是不好的

例子：

```
#include <stdio.h>
int * method(void);
int main()
{
    int *p = method();
    return 0;
}
int * method(void)
{
    int p = 10;
    int *p = &p;
    return *p;
}
```

解释：因为在函数中的元素在函数外部将会释放它的内存地址，因此method传出的参数在函数的外部会将所储存的元素释放。

指针传入函数

指针传入函数中是指针的副本，如果在函数中通过指针对变量进行改变，这是正确的，但是如果在函数中通过指针本身对指针进行改变（例如交换）这个只在函数内部有效，如果要实现这个功能必须要使用双重指针来传入swap(&p1,&p2),void swap(int **p1,int **p2) int *temp = p1;p1 = p2;p2 = temp;这样才行

const指针

```
#include <stdio.h>
int main()
{
    int a = 20;
    int b = 10;
    int const *p1 = &a; // const int *p1 = &a;
    p1 = &b;
    // *p1 = 10;      Error! the pointer is read-only
    // a = 10; Correct!
    printf("%d", *p1); // Prints 10
}
```

const int *ptr表示的是这个指针受到保护，不能够通过这个指针来修改变量的数值。但是可以通过改变指针所指向的对象来进行修改（可以指向另外一个变量）或者是直接改变变量a来进行修改这个和int const*p1 = &a;的结果是一样的。

```
#include <stdio.h>
int main()
{
    const int a = 20;
    int b = 10;
    int *p1 = &a;
    p1 = &b; // Warning
    *p1 = 10; // Warning
    printf("%d", *p1); // Prints 10
}
```

可以通过指针来改变`const int`的内容，这里编译器报警，但是不会导致程序不运行，这里相当于使用`*p1`对`const int`所做的操作都是检查不出问题的（这是`UNdefined Behaviors`）

如果我们希望这个指针只能指向这一个变量而不能够指向其他的变量，那么我们应该使用`int *const pc`来确定一个`Top-locked pointer`

```
int x = 42;
int *const pc = &x;
++*pc; // OK.
int y = 30;
pc = &y; // Error
```

在这个例子中我们注意到，这里的指针不能够通过赋值其他的变量的地址来进行对指针内容进行修改，但是可以通过指针来修改变量的数值，也可以直接对变量的内容进行修改（变量没有受到`const`的限制）

所以这样的话我们有最高的上锁的指针 `const int *const cipc = &x` 也就是说不能够对他赋值，也不能够转换指针指向的对象。

`void*`和`nullptr`

任何其他的指针能够隐式转换为`void*`类型的指针，但是从`void*`类型转换成为其他的指针类型通常需要显示转化。

- 常见运用
 - `int *p = malloc(sizeof(int));` `malloc`返回值是`void*`但是允许调用者将内存解释成为任意的对象类型
 - 空指针与任何指向对象的指针比较的时候均不相同
 - 空指针转换成为`int`类型的时候为`char *ptr = NULL; int x = (int)ptr;`
 - 空指针和其他的空指针通过比较操作符进行比较之后，两个空指针总是相同的。（不论是否是相同类型的）

函数指针

- 函数指针是一种指针，即 变量，它 指向一个函数的地址。
- 运行中的程序在主内存中占据一定空间。无论是可执行的编译代码还是所使用的变量，都存放在这块内存中，因此程序代码中的函数也具有唯一的地址。

- 定义: `return_type (* pointer_name) (datatype_arg_1, datatype_arg_1, ...);`
 - 例如: `float foo(int,int)->float (*foo_ptr)(int,int);`
 - 实例: `void* (*p)(int *,int *);`
- 传递:

```
void f(int);
void (*ptr1)(int) = &f;
void (*ptr2)(int) = f;//和上面一样
```

- 使用

```
int (*pointer)(int);
int areaSquare (int);
pointer = areaSquare;
```

```
int length = 5;
int area = areaSquare(length);
int area = (*pointer)(length);//使用指针1
int area = pointer(length);//使用指针2
```

宏与NULL

- `#define PI 3.1415926`,相当于将后文的PI全部替换成为3.1415926,这仅仅理解为一个字符替换的过程。
- `NULL`也是一个实现定义的空指针常量,在不同的类型的变量接受的时候,常量也会发生变化。

数组与指针

数组的基本知识

- 数组只能够通过对其中的元素赋值来修改,不能够在main函数中使用`b = {1,2,3}`这样的方式进行赋值
- 数组的类型是`int [M]`而不是`int *`虽然在使用的时候他会转换成为首地址的形式
- 数组在函数中表示的首地址只能够获取不能够修改,相当于`const int *p`,但是与之不同的是,可以通过解引用的方式对arr的首元素进行修改,即`*arr`相当于`arr[0]`.

```
#include <stdio.h>
int main()
{
    int arr_1[] = {1, 2, 4, 5, 6};
    int arr_2[] = {7, 8, 9};
    // arr_1 = arr_2;这样是错误的, 因为数组的首地址是不能够修改的
    int *p1 = arr_1;
    int *p2 = arr_2;
    void *temp;
```

```

    temp = p1;
    p1 = p2;
    p2 = (int *)temp;
    for (int i = 0; i < 3; i++)
        printf("%d", p1[i]); //prints 789
}

```

- 如果要通过指针的形式来遍历数组，请使用指针变量来接受数组的地址。
- 数组的初始化
 - `int arr[100] = {1}`: 表示的是第一个元素为1，后面的元素都是0。
 - `char arr[100] = "1"`: 表示的是第一个元素为'1'后面的元素是'\0'，因为'\0'的ASCII码为0。

二维数组的定义

```

#include <stdio.h>
int main()
{
    arr[2][4] =
    {
        {1, 2, 3, 4},
        {3, 4, 5, 6}
    };
}

```

```

#include <stdio.h>
{
    int arr1[4] = {1, 2, 3, 4};
    int arr2[4] = {3, 4, 5, 6};
    int *arr[2] = {arr1, arr2};
    // 解释:int* 表示存放的内容是指针,arr 表示的是这个指针数组的名称叫做arr.
}

```

区别，第一种规定的两个数组的长度应该相同，但是第二种对两个数组的长度没有限制。注意，这个时候的占用空间就变成有其中的指针的占用空间了。

使用指针来遍历二维数组

定义`int arr[3][5]`的指针：`int (*p)[5] = arr;`解释：因为`int [5]`表示的是指针所指的是一个长度为5的一个类型(也就是所指的是数组的首地址)，`*p`表示的是这是一个指针，并且指针的名称为`p`。

其中这里的`arr + 1`表示的是移动`int [5]`个单位，也就是说直接移动的长度是`arr`中整个子数组的长度

```

int (*p)[5] = arr;
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 5; j++)

```

```

    printf("%d ", (*p)++);
    p++;
}

```

如果是按照`int *arr[2] = {arr1, arr2};`来定义的二维数组的话`int **p = arr;`这里指向的类型是`arr`的首地址，也就是`arr1`，即指向`int`的指针。

遍历的过程当中要注意

```

#include <stdio.h>
void operation(int **arr)
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 4; j++)
            printf("%d ", *(*arr + j));
        putchar('\n');
        arr++;
    }
}
int main()
{
    int arr_1[] = {1, 2, 3, 4};
    int arr_2[] = {11, 22, 33, 44};
    int *arr[2] = {arr_1, arr_2};
    operation(arr);
    return 0;
}

```

首先应当先解引用`*p`，现在`*p`表示的是`arr1`的首地址，也就是第一个元素，然后通过指针的运算获得后面的元素的地址，随后在进行解引用得到他所拥有的数据，在内层循环外部通过`p++`表示加上一个指针的长度，这样就可以有效的过渡到第二个数组`arr2`，因此这么写是✓

为什么使用指针接受数组可以`p++`操作，但是如果是直接`arr++`就不行？

因为如果是`int *p = arr`，这里的`p`是一个指针的变量，可以进行操作，但是`arr`是一个固定的地址，表示的是指向数组首地址的指针，这个是不能够操作的。

- 数组指针和指针数组

1. 数组指针：指向数组的指针

1. 一维数组：`int *p = arr`步长为`sizeof(int)`

2. 二维数组：`int (*p)[5] = &arr(arr)`步长为`sizeof(int arr[5])`

2. 指针数组：存放指针的数组`int *arr = {arr1, arr2};` ` ` `int *p[5]`

数组作为指针传递到函数中

```

void fun(int *a);
void fun(int a[]);

```

```
void fun(int a[10]);
void fun(int a[2]);
```

这一些都是正确的这里的a都是作为指针int *

同样的，二维数组也可以作为参数传递到函数中

```
void fun(int (*a)[N]);
void fun(int a[][N]);
void fun(int a[2][N]);
void fun(int a[10][N]);
// 因为这个地方的a[M]都会被隐式转换成为(*a)
```

这一些都是可以的

数组传递到函数中会退化成为首地址，这样的话，指针可以移动，这个地方不算修改函数首地址。（血的教训）

指向首地址与指向整个数组

```
int arr[5];
int *p = &arr;
int (*p_arr)[5] = &arr;
```

这个地方int (*p_arr)[5]就是指向整个数组的一个指针，int *p就是指向数组的首地址的一个指针。

- 指向整个数组的指针和指向首地址的指针的区别？
 1. 对int (*p_arr)[5]的指针进行操作的时候相当于对于整个数组进行操作，例如要选取其中的某个元素(*p_arr)[i]表示获得其中某个元素的内容
 2. 同样的对于int (*p_arr)[5]进行数学操作,p_arr+1表示的就是往后移动整个数组的长度
- 同样的我们可以定义一个指向一个二维数组整体的指针int (*p_arr)[5][10]

动态内存

计算机中存在多个内存的区域，其中最常见的区域是栈(stack)和堆(heap)

stack的内存由编译器自动管理。stack内存的分配和释放速度都非常快，因为他遵循后进先出的原则，但是他的内存通常较小，而且是固定的

heap内存通过程序员手动管理，使用malloc,calloc,new等函数的分配，使用free,delete进行删除。

malloc函数

在<stdlib.h>中的定义void * malloc(size_t size)

其中的size_t的变量就相当于unsigned int.

```
T *ptr = malloc(sizeof(T)*n);
for (int i = 0; i!=n; i++)
{
    ptr[i] = ....
}
free(ptr);
```

使用malloc函数申请一个数组

- 一维数组

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *arr = malloc(sizeof(int) * 5);
    for (int i = 0; i < 5; i++)
    {
        arr[i] = i;
    }
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

- 二维数组

- 使用连续申请的方式来进行申请

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int **arr = malloc(sizeof(void *) * 3);
    for (int i = 0; i <= 3; i++)
        arr[i] = malloc(sizeof(int) * 5);
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 5; j++)
            arr[i][j] = 3 * i * j;
    }
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 5; j++)
            printf("%d ", arr[i][j]);
        putchar('\n');
    }
    for (int i = 0; i < 3; i++)
        free(arr[i]);
    free(arr);
}
```

```

    }
    for(int i = 0;i<3;i++)
        free(arr[i]); //释放每一行内存
    free(arr); //释放储存指针的数组
    return 0;
}

```

- 使用计算的方式来申请

```

int *p = malloc(sizeof(int) * n * m);
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        p[i * m + j] = /* ... */ // This is the (i, j)-th entry.
// ...
free(p);

```

malloc()的返回值

- 当malloc尝试分配一块非常大的内存的时候，如果系统没有足够的(heap)来存放内存，malloc就会返回一个空指针，可以通过确定返回的指针是否是空指针来确定是否存在充足的内存可以使用

```

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

int main() {
    // 尝试分配一个非常大的内存块
    int *ptr = (int *)malloc(1ull << 60);
    if (!ptr) {
        // 检查是否分配失败
        fprintf(stderr, "Out of memory.\n");
    }
    if (ptr) {
        free(ptr); // 如果分配成功，释放内存
    }
    return 0;
}

```

calloc函数

- void *calloc(size_t num, size_t each_size); 和malloc(num*each_size)相同的含义，但是他将所有的内容都初始化为相应的空元素 (int : 0 ;char:'\0')

free函数

free函数的作用是释放内存，但是需要注意的是，free函数只能释放由malloc, calloc, new等函数申请的内存，不能释放其他的内存。

- 注意`free`不能够重复释放内存（未定义行为）
- `free`不能够一个一个地释放`malloc`出来的相应的地址
- 在调用`free`函数之前不需要进行检查，因为`free(NULL)`是安全的，它不会导致任何错误，然而这并不意味着可以重复调用`free`函数。

传入0个需要分配的空间会怎样？

- `malloc(0)`的行为是由编译器决定的。这意味着不同的编译器可能存在不同的处理方式。它可能返回一个空指针，表示没有分配任何内存；也可能返回一个非空的指针。这样也会导致内存泄露的问题，因为如果返回了一个非空的指针，而这个时候你没有使用`free`函数将这一块空间释放，这样就会导致内存泄漏。
- 和`malloc`一样`calloc(0, N)`也是根据编译器来选择不同行为的。
- `free(NULL)`是安全的，不会产生任何副作用，但是不能够重复释放同一个内存。

realloc函数重新分配内存地址

- 参数`ptr`可以为`null`，或之前由`malloc`, `calloc`, `realloc`分配但是没有被`free`的指针或者是`NULL`
- 如果`ptr`是`NULL`，行为等同于`malloc(new_size)`
- 如果`ptr`不是`NULL`，则内存分配通过以下方式进行
 - 扩展或者收缩`ptr`现有内存
 - 分配一个大小为`new_size`的新内存地址，将新的内存地址中较小的部分的数据复制过去，然后释放旧内存块
 - 如果内存不足，则旧内存块不会被释放，并返回空指针**

malloc(0)的情况

`malloc(0), calloc(0, N), calloc(N, 0), realloc(ptr, 0)`这些结果都是实现定义的

- 这些函数可能不分配任何内存地址而返回空指针
- 然而，对这些指针进行解引用也是未定义行为
- 为了避免内存泄漏，这样分配的内存也要释放

memset函数

- `void *memset(void *s, int c, size_t n);` (定义在`<string.h>`中)
 - 参数解释：
 - `s`指向的是待填充的内存块的指针
 - `c`填充值，会被转换成为`unsigned char`类型
 - `n`填充的字节数
- 返回值：返回填充后内存块指针`s`
- 例如：

```
#include <string.h>
int arr[10];
memset(arr, 0, sizeof(arr)); // 将 arr 所有字节置为 0
```

一个经典的例子——读取一个长度不限的字符串

- 使用malloc和free来完成这个任务

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define INITIAL_SIZE 10
char *read_string(void)
{
    char c = getchar();
    while (isspace(c))
        c = getchar();
    char *buffer = malloc(INITIAL_SIZE);
    int capacity = INITIAL_SIZE;
    int cur_pos = 0;
    while (c != '\n')
    {
        if (cur_pos == capacity - 1)
        {
            char *new_buffer = malloc(capacity * 2);
            if (new_buffer == NULL)
            {
                printf("WRONG!");
                buffer[cur_pos] = '\0';
                return buffer;
            }
            memcpy(new_buffer, buffer, cur_pos);
            free(buffer);
            capacity *= 2;
            buffer = new_buffer; //指针的赋值
        }
        buffer[cur_pos++] = c;
        c = getchar();
    }
    ungetc(c, stdin);
    buffer[cur_pos] = '\0';
    return buffer;
}
int main()
{
    char *str = read_string();
    printf("%s", str);
    return 0;
}
```

- 使用realloc()写的

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

```
#include <ctype.h>
#define INITIAL_SIZE 10
char *read_string(void)
{
    char c = getchar();
    while (isspace(c))
        c = getchar();
    char *buffer = malloc(INITIAL_SIZE);
    int capacity = INITIAL_SIZE;
    int cur_pos = 0;
    while (c != '\n')
    {
        if (cur_pos == capacity - 1)
        {
            char *new_buffer = realloc(buffer, 2 * capacity);
            if (new_buffer == NULL)
            {
                printf("Wrong!\n");
                buffer[cur_pos++] = c;
                buffer[cur_pos] = '\0';
                return buffer;
            }
            capacity *= 2;
            buffer = new_buffer;
        }
        buffer[cur_pos++] = c;
        c = getchar();
    }
    ungetc(c, stdin);
    buffer[cur_pos] = '\0';
    return buffer;
}
int main()
{
    char *str = read_string();
    printf("%s", str);
    return 0;
}
```

字符串

字符串与\0

- \0表示的是字符串的结束，在使用char str[15] = "This is me"来储存字符的时候一定要考虑到\0算作其中的一个元素（如果少了会导致在输出的时候存在未定义行为）。
- 同样的printf("%s", str);的时候会在\0的地方停下来，所以如果没有\0的时候就会导致未定义行为（数组越界）
- char empty[] = "";这里empty的大小是1，因为要包括\0.
- 字符串的读入和修改

```
char str[100] = "abcdef";
scanf("%s",str);//reads "123" ,str becomes {'1','2','3','\0','e','f'}
printf("%s",str);//prints "123"
```

`scanf()`函数并不安全，`scanf()`在读取字符串的时候仅仅是将它转换成为了一个指针来传递，并不知道这个字符串的大小，所以不能够检查时候会超过`char str[10]`的内部的空间。

gets与fgets

`gets`没有边界检查，所以不推荐使用，和`gets`一样的有`get_s`但是这个不是在所有的编译器中都能使用的。

`fgets`是一个更加安全的函数，它的使用方法是

```
char str[100];
fgets(str,100,stdin);
```

这里的`fgets`就只会最多读取99位字符。

According to the cppreference documentation for `fgets`:

It reads at most `size - 1` characters. Here `size` is the number of elements specified as an argument (typically the size of the buffer). This leaves space for the null terminator (\0).

It stops under one of these conditions:

- When it reads a newline character
- when it reaches the end of the file
- When it has read `size - 1` characters

字符串的相关函数

在`<ctype.h>`中定义的

- `islower`: 检查是否是小写字符，如果是小写字母，返回非零值；否则返回 0。
- `isupper`: 检查是否是大写字符，若为大写字母，返回非零值；反之返回 0。
- `isspace`: 该函数用于检查字符是否为空白字符，空白字符包括空格、制表符、换行符等。如果是空白字符，返回非零值；否则返回 0。
- `ispunct`: 此函数用于检查字符是否为标点字符。若为标点字符，返回非零值；反之返回 0。
- `tolower`: 该函数用于将大写字母转换为小写字母。如果传入的字符是大写字母，则返回对应的小写字母；否则返回原字符。
- `toupper`: 此函数用于将小写字母转换为大写字母。若传入的字符是小写字母，则返回对应的大写字母；否则返回原字符。

在`<stdlib.h>`中定义

- `strto+`...:

- **strtol**: The function prototype is `Long strtol(const char *nptr, char **endptr, int base)`. It converts a string to a `Long Long` value in the specified `base`(radix, ranging from 2 to 36, or 0 for auto-detection). (字符串一开始0表示八进制, 字符串一开始0x表示16进制) It skips leading whitespace, starts conversion at digits/signs, and stops at non-digit characters or '\0'. If `endptr` isn't `NULL`, it stores the pointer to the character after the conversion end. (`endptr`表示的就是遇到第一个不满足条件的字符, 随后停止转换, 如果这里写`NULL`, 表示遇到不满足的条件的时候直接返回`NULL`, 没有指针指向它)
 - **strtoul**: `unsigned long strtoul(const char *nptr, char **endptr, int base)`
 - **strtoull**: `unsigned long long strtoull (const char*nptr, char**endptr, int base)`
 - **strof**: `double strtod(const char *nptr, char **endptr)` 这里就不存在转换进制的功能。
 - **strtold**: `long double strtole(const char *nptr, char **endptr)`
- **snprintf**: `snprintf(str, sizeof(str), "%f", num);` 根据字符串 `format` 格式将数据格式化输出到 `str` 中, 最多写入 `size-1` 个字符串, 并且在末尾添加上 '\0'.

在 `<string.h>` 中定义

- **strcpy(dest, src)** 将字符串 `src` 复制到 `dest` 中。包含结束符 '\0'.
- **strncpy(dest, src, n)**; 从源字符串 `src` 复制最多 `n` 个字符到目标字符串 `dest` 中。如果 `src` 的长度小于 `n`, 则用 '\0' 填充 `dest` 直到复制了 `n` 个字符; 如果 `src` 的长度大于等于 `n`, 则不会在 `dest` 末尾添加 '\0' *If dest and src point to the same memory address, strcpy will still work correctly because it copies the null terminator (\0) from src to dest. The result will be the same string in the same memory location.*
- **strcat(dest, src)** 将源字符串 `src` 连接到目标字符串 `dest` 的末尾, 覆盖 `dest` 末尾的 '\0', 并在连接后的字符串末尾添加新的 '\0'.
- **strncat(dest, src, n)** 将源字符串 `src` 的最多 `n` 个字符连接到目标字符串 `dest` 的末尾, 并在连接后的字符串末尾添加 '\0'. 如果 `src` 的长度小于 `n`, 则将整个 `src` 连接到 `dest` 末尾。
- `size_t len = strlen(str)` 返回给定的字符串的长度(不包括 '\0') 注意 `strlen()` 的运算速度很慢, 不要总是调用这个函数。
- **strcmp(s1, s2)** 比较两个字符串 `s1` 和 `s2` 的大小。如果 `s1` 小于 `s2`, 返回一个负数; 如果 `s1` 等于 `s2`, 返回 0; 如果 `s1` 大于 `s2`, 返回一个正数。 *(The result of the result is not limited to -1, 0, or 1.)*
- **strchr** 查找第一次出现的指定元素, 并返回 `char *` 类型的地址
- **strrchr** 查找最后一次出现的指定元素
- **strstr(haystack, needle)** 在字符串 `haystack` 中查找子字符串 `needle` 首次出现的位置, 并返回指向该位置的指针。如果未找到, 则返回 `NULL`.

Some tips: Overlapping memory:

- **strcpy** with overlapping memory:

This is undefined behavior. The C standard does not guarantee correct behavior when memory regions overlap. The function might overwrite parts of the source string before copying them.

- **strncpy** with overlapping memory:

This is also undefined behavior. While strncpy limits the number of characters copied, it still doesn't handle overlapping memory safely.

- **strcat** with overlapping memory:

*This is **undefined behavior**. strcat first finds the end of dest and then appends src. If the memory regions overlap, it might overwrite parts of src before copying them.*

Can or Cannot change?

```
#include <stdio.h>
int main()
{
    char *str_1 = "Liyiming";
    char str_2[] = "Liyiming";//It copies the whole string to the array `str_2`.
    str_1[4] = 'M'//No compile-error,but undefined behaviors.The string cannot
change
    str_2[4] = 'M'//Correct! The array can change.
    return 0;
}
```

The change of the string can lead to the severe runtime-error.

There are some ways to protect it:

```
const char *str = "abcde";
str[3] = 'a';//compile error
```

Array of strings

```
const char *translations[] = {
    "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine"};
```

*Notes that **translation** is an array of pointers, where each pointer points to a string literal.**translations** is not a 2-d array.*

Struct

- 结构体可以理解为自定义的数据类型，他是由一批数据组合成为的结构性数据。

```
struct 结构体名称
{
    成员1;
    成员2;
    ...
};
```

```

#include <stdio.h>
#include <string.h>
struct Girlfriend
{
    char name[100];
    int age;
    char gender;
    double height;
};

struct Student
{
    char name[30];
    int age;
    double height;
};

int main()
{
    /*
        结构体：
        自定义的数据类型
        就是由很多的数据组合成为一个整体
        每一个数据，都是结构体的成员
        书写的位罝：
        函数里面：局部位置只能在本函数中使用
        函数外面：在所有的函数中都可以使用
    */
    // 使用结构体
    // 定义一个类型的变量
    struct Girlfriend gf1;
    strcpy(gf1.name, "aaa");//字符串的赋值需要注意
    gf1.age = 21;
    gf1.gender = 'F';
    gf1.height = 1.63;
    //-----
    struct Student stu1 = {"Sam", 18, 175.26};
    struct Student stu2 = {"Lily", 17, 16.35};
    struct Student strArr[2] = {stu1, stu2};
    // 遍历每一个元素
    for(int i = 0; i < 2; i++)
    {
        struct Student temp = strArr[i];
        printf("%s %d %lf", temp.name, temp.age, temp.height);
    }
    return 0;
}

```

结构体的别名

```

typedef struct (Name)//name可以写也可以不写
{

```

```
char name[100];
int age;
char gender;
double height;
}GF;
```

```
#include <stdio.h>
#include <string.h>
typedef struct Ultram
{
    char name[100];
    int attack;
    int defense;
    int blood;
} M;

int main()
{
    M taro = {"Laitor", 100, 90, 500};
    M rem = {"rem", 90, 80, 450};
    M arr[2] = {taro, rem};
    for (int i = 0; i < 2; i++)
    {
        M temp = arr[i];
        printf("%s %d %d %d\n", temp.name, temp.attack, temp.defense, temp.blood);
    }
    return 0;
}
```

- 相当于在这里我们将 `struct Ultram` 改成了 `M`.

结构体与函数

- 如果将结构体直接传入函数中，这个时候传入的其实是结构体的副本，如果在函数中对结构体进行改变，这个时候也不会改变结构体的内部的内容（**哪怕结构体中存在数组**）
 - 但是如果是结构体中存在数组的情况，数组会拷贝，所以尽管数组不能够直接拷贝，但是结构体的数组元素可以直接拷贝

```
struct A{
    int arr[5];
};
```

```
int a[10];
int b[10] = a; //Error!
```

```
struct A a;
struct A b = a;//correct!
```

- 如果希望你能够传入结构体本身进入函数，那么需要传递结构体的地址，然后再使用指针来接受(**struct Student *stu1**)。

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
typedef struct Student//注意要写在函数声明的上面
{
    char name[30];
    int age;
} stu;
void change(stu *st); //类型是stu的一个指针
int main(void)
{
    stu stu1;
    strcpy(stu1.name, "Liyiming");
    stu1.age = 18;
    change(&stu1);
    printf("The name of stu1 is %s\n", stu1.name);
    printf("The age of stu1 is %d\n", stu1.age);
    return 0;
}
void change(stu *st)
{
    printf("The name of stu1 is %s\n", (*st).name); //(*st).name = st->name
    printf("The age of stu1 is %d\n", (*st).age);
    scanf("%s", (*st).name);
    scanf("%d", &(*st).age);
}
```

同样的，结构体同样也可以作为函数的返回值，和上面直接传入结构体（而不是它的地址）而言，返回值也是它的一个copy。

```
#include <stdio.h>

typedef struct Point {
    int x;
    int y;
} Point;

// 返回结构体
Point createPoint(int x, int y) {
    Point p = {x, y};
    return p;
}
```

```
// 返回结构体指针
Point* createPointPtr(int x, int y) {
    static Point p; // 使用static确保返回的指针有效
    p.x = x;
    p.y = y;
    return &p;
}

int main() {
    // 使用返回结构体的函数
    Point p1 = createPoint(10, 20);
    printf("p1: (%d, %d)\n", p1.x, p1.y);

    // 使用返回结构体指针的函数
    Point* p2 = createPointPtr(30, 40);
    printf("p2: (%d, %d)\n", p2->x, p2->y);

    return 0;
}
```

结构体的嵌套

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
typedef struct Message
{
    char phone[12];
    char school_mail[100];
} msg;
typedef struct Student
{
    char name[30];
    int age;
    struct Message msg;
} stu;

void change(stu *st);
int main(void)
{
    stu stu1;
    strcpy(stu1.name, "Liyiming");
    stu1.age = 18;
    strcpy(stu1.msg.phone, "17775756985");
    strcpy(stu1.msg.school_mail, "liym2024@shanghaitech.edu.cn");
    change(&stu1); //如果传递的是stu1的话意味着传递的是一个副本
    printf("%s", stu1.msg.school_mail);
    return 0;
/*
stu stu2 = {"Liyiming", 18, {"17775756985", "liym2024@shanghaitech.edu.cn"}};
*/
```

```

    */
}

void change(stu *st)
{
    scanf("%s", (*st).msg.school_mail);
}

```

结构体的内存对齐

- 确定变量的位置：
 1. 总体上还是按照定义的顺序从前到后的安排内存地址
 2. 每一个变量只能放在自己类型整数倍的内存地址上（中间空出来的字节会被补位空白字符）
- 最后一个补位：结构体的总大小，是最大类型的整数倍
- 补位并不会改变相应的类型的变量的大小
- 其实不只是在结构体中，只要是储存变量就会存在内存对齐的情况
- 综上：我们将小的数据类型写在上面，大的数据类型写在下面（节省空间）

结构体指针

```
struct Student *pstu = malloc(sizeof(struct Student));
```

*Member access through a pointer :`ptr->mem`(maybe better) or `(*ptr).mem` or `*ptr.mem`. For example: `ptr->num = 123;`. As usual, don't forget to `free` after use.*

结构体初始化

结构体在定义的时候，是不存在初始化的。当我们去调用实例的时候，如果是定义的一个全局的实例，那么就会是一个已经初始化的结果，同样的，如果在定义实例的时候定义为一个静态的实例那么也会初始化。

```

#include <stdio.h>
typedef struct Student
{
    int id;
    double height;
} stu;
stu stu1;//全局变量，自动初始化
int main()
{
    stu stu2={0,0.0};//显示初始化
    static stu stu3;//使用static关键字将它初始化
    printf("%d %f", stu1.id, stu1.height);
    return 0;
}

```

结构体数组

和其他的类型的数组一样，使用指针可以快速的定义一个数组，并采用循环的方式来赋值。

```
struct Student *student_list = malloc(sizeof(struct Student) * n);
for (int i = 0; i != n; ++i)
{
    // A, B, C and D are some functions
    student_list[i].id = B(i);
    student_list[i].entrance_year = C(i);
    student_list[i].dorm = D(i);
}
```

```
struct Student *student_list = malloc(sizeof(struct Student) * n);
for (int i = 0; i != n; ++i)
{
    student_list[i] = (struct Student){.name = A(i), .id = B(i),
        .entrance_year = C(i), .dorm = D(i)};
}
```

结构体中的指针数组

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

// 定义包含数组指针的结构体
typedef struct {
    int *arr;
    int size;
} MyStruct;

int main() {
    // 初始化一个结构体实例
    MyStruct original;
    original.size = 5;
    original.arr = (int *)malloc(original.size * sizeof(int));
    for (int i = 0; i < original.size; i++) {
        original.arr[i] = i;
    }

    // 复制结构体
    MyStruct copy = original;

    // 修改复制结构体中指针所指向的数组元素
    copy.arr[0] = 100;

    // 输出原结构体中指针所指向的数组元素
    printf("Original array: ");
    for (int i = 0; i < original.size; i++) {
        printf("%d ", original.arr[i]);
    }
    printf("\n");
}
```

```
// 释放内存  
free(original.arr);  
  
return 0;  
}
```

可见如果是使用的是指针来储存数组，那么在结构体复制的时候内部的指针也会进行复制，也就是说在两个结构体中的数组是同一个数组，在`free(ptr);`的时候会发生UB.

- Deep copy of the array

```
void vector_assign(struct Vector * to, const struct Vector *from)  
{  
    to->entries = malloc(from->dimension * sizeof(double));  
    memcpy(to->entries, from->entries, from->dimension * sizeof(double));  
    to->dimension = from->dimension;  
}
```

命令行输出

```
int main(int argc,char **argv){  
    //body  
}
```

注意如果输入`.\test liyiming 060514 123`, 那么`argc = 4, argv[0] = '.\test', argv[1] = 'liyiming'....` 我们如何简化这个复杂的过程?

The journey is to be continued