

C语言工作原理

1.编译器：

微软阵营：[MSVC](#)

开源组织阵营GUN:

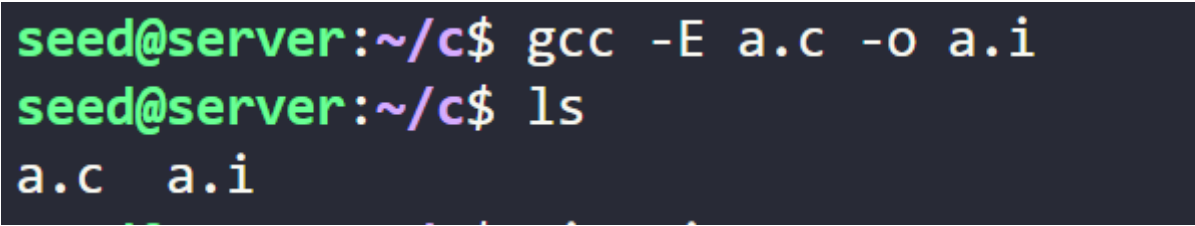
- linux: gcc
- Windows: minGW

编译器工作流程：

预处理----编译---汇编----链接

- 预处理

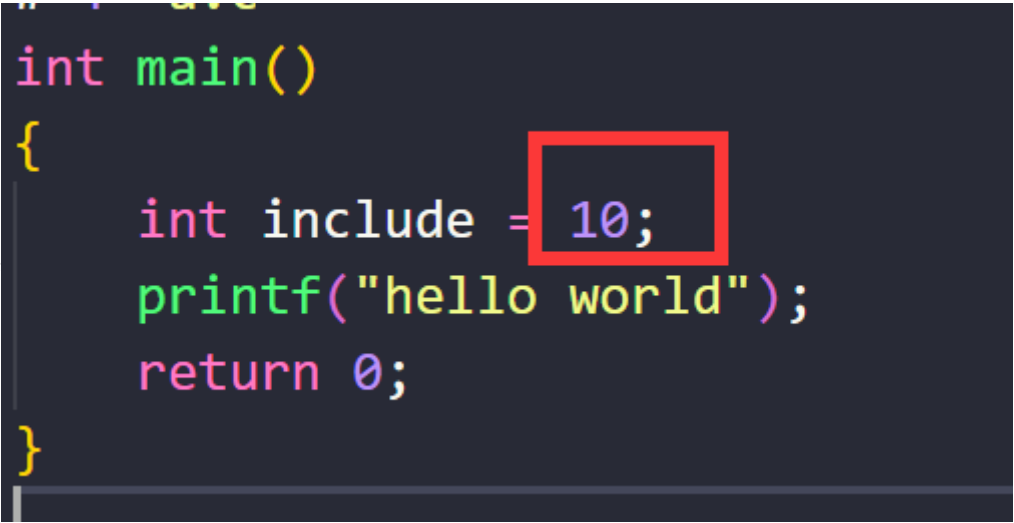
```
1 gcc -E a.c -o a.i
```



预处理的工作：将#所有的东西进行解释

```
1 //1_1
2 #include <stdio.h>
3 #define ABC 10
4
5 int main()
6 {
7     int include = ABC;
8     printf("hello world");
9     return 0;
10 }
```

预处理后.i文件的ABC被替换



注意:只有#后面的include才会被预处理或者说被预处理翻译

以后工程会用到的调试技巧

```
1 //1_2
2 #include <stdio.h>
3 #define ABC 10
4
5 int main()
6 {
7     int include = ABC;
8     #ifdef DEBUG
9         printf("Debug\n");
10    #else
11        printf("No Debug\n");
12    #endif
13    return 0;
14 }
```

如果编译加上 `gcc 1_2.c -o -D DEBUG a` -D加上你定义的宏就可以输出你调试的语句了

```
seed@server:~/c$ gcc -D DEBUG 1_2.c -o a
seed@server:~/c$ ./a
Debug
```

- 编译器

```
1 gcc -S a.i -o a.s
```

```
seed@server:~/c$ gcc -S a.i -o a.s
seed@server:~/c$ vi a.s
seed@server:~/c$ ls
a.c a.i a.s
```

- 汇编器

```
1 gcc -c a.s -o a.o
```

```
seed@server:~/c$ gcc -c a.s -o a.o
seed@server:~/c$ vi a.o
seed@server:~/c$ ls
a.c a.i a.o a.s
```

- 链接

```
1 gcc a.o -o a
```

```
seed@server:~/c$ gcc a.o -o a
seed@server:~/c$ la
a a.c a.i a.o a.s
```

总结：以后编译C/C++文件用两条命令就行

```
1 gcc -c a.c -o a.o
2 gcc a.c -o a
3 #或者一条也行
4 gcc a.c -o a
```

2.数字的表示

1. 进制

- 二进制：0~1
- 八进制：0~8
- 十进制：0~9
- 十六进制：0~F

2. 二进制和十进制转化

记住2的1~10次方的数

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1024$$

$$(127)_{10} = 1+2+4+8+16+32+64=(01111111)_2$$

3. 八进制和十六进制

八进制三位二进制进行缩写

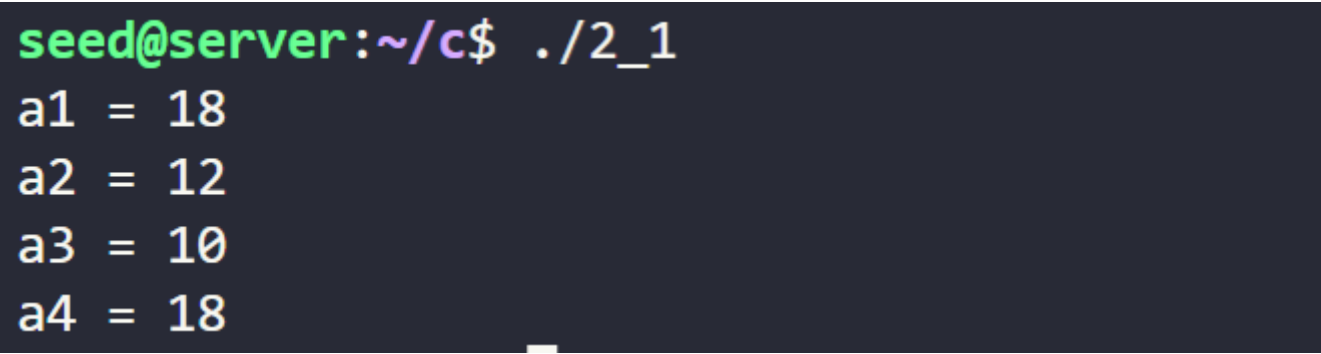
十六进制四位二进制进行缩写

二进制	十六进制
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

$(1111)_2 = (17)_8 = (F)_{16}$

4. C语言中表示2, 8, 10, 16进制

```
1 //2_1
2 #include <stdio.h>
3
4 int main()
5 {
6     char a1 = 0x12; //十六进制表示 0x
7     char a2 = 12; //十进制表示
8     char a3 = 012; //八进制表示 0
9     char a4 = 0b00010010; //二进制表示 0b
10    printf("a1 = %d\n",a1);
11    printf("a2 = %d\n",a2);
12    printf("a3 = %d\n",a3);
13    printf("a4 = %d\n",a4);
14    return 0;
15 }
```



打印表是各个进制数

%d打印十进制数

%o打印八进制数

%x打印十六进制数

```
1 //2_2
2 #include <stdio.h>
3
4 int main()
5 {
6     char a = 12;
7     printf("a = %d, %o, %x\n",a,a,a);
8     return 0;
9 }
```

```
seed@server:~/c$ ./2_2
a = 12, 14, c
```

3.C语言关键字

1. C语言ASCII码

ASCII TABLE														
Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	~
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

技巧：字符数字转化为数字 字符数字 - '0'

- \r :回车
- \n :换行
- \t:用tab键控制占位符
- \数字: 默认为八进制

```
1 //3_1
2 #include <stdio.h>
3
4 int main()
5 {
6     char a1 = 0x12;
7     char b1 = '9';//表示字符9
8     char c1 = '\12';//默认表示8进制数
9     //char c1 = '\x12';//表示十六进制数
10
11     printf("a1 = %d\t%o\t%x\n",a1,a1,a1);
12     printf("a1 = %d, %o, %x, %c\n",b1,b1,b1,b1);
13     printf("a1 = %d,%o,%x\n",c1,c1,c1);
14     return 0;
15 }
```

```
seed@server:~/c$ ./3_1
a1 = 18 22      12
a1 = 57, 71, 39, 9
a1 = 18,22,12
```

2. 内存分配行为关键字

- 内存分配行为单词
 - 基础数据类型关键字

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

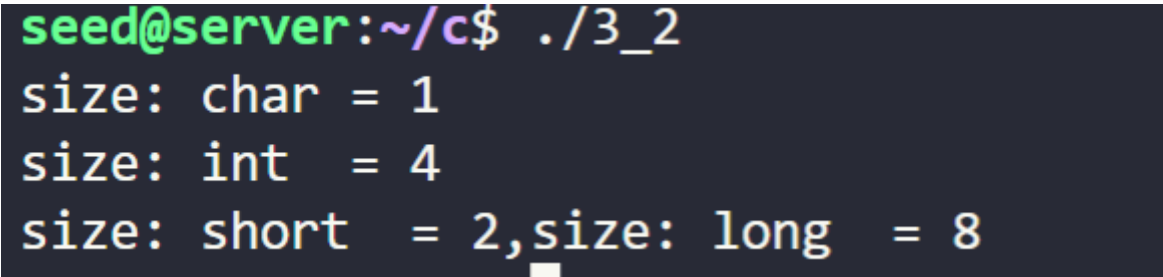
char: 最小单位 1B = 8bit

short: 2B

int: <=4B

long: >=4B

```
1 //3_2
2 #include <stdio.h>
3
4 int main()
5 {
6     char a = 12;
7     printf("size: char = %ld\n",sizeof a);
8     printf("size: int  = %ld\n",sizeof(int));
9     printf("size: short = %ld,size: long  = %ld\n",sizeof(short),sizeof(long long));
10    return 0;
11 }
12
```



注意: long long a = 1e10L float = 1.1f

C编译器默认整数数字空间是 int 大小字节的，浮点数默认为 double 的,为了防止1e10在默认空间大小下不能填充，一般是长整形long一般在后面加一个L,1.1默认是double空间大小，如果是float是一般在后面加个f

注意: 如何用到无符号 一般都加unsigned

char默认是有符号还是无符号与编译器有关

C语言空间操作篇

4. 如何访问和描述空间

1. 访问一个空间

- 有名访问

通过定义变量，以变量名来访问

```
1 int a;
2 char b;
3 struct buffer data;
```

变量定义在内存上，内存资源为了能够让CPU访问到，必须编制，通过名字访问时，对CPU来说，变量名只是一个地址的一个代号而已

- 无名访问

空间的本质：用一个地址进行编址，使CPU可以访问

- 如何保存地址的值

- 1. 数字概念：足够存储地址的大小

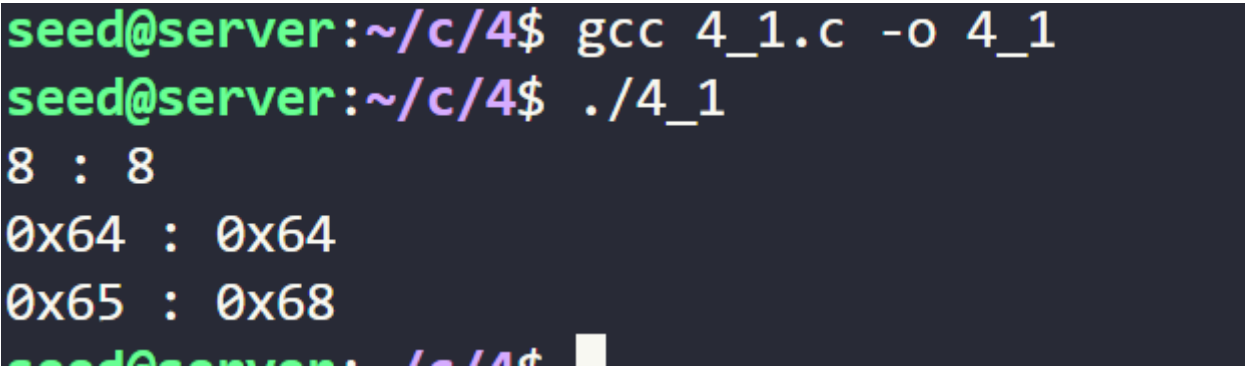
2. 物理含义：和普通的数字没有区别

2. 描述一个空间

一个地址描述空间这个地址要满足的要素

- 具备存储地址大小的容量：一般为32位或者64位（多少位系统有关）
- 地址的下一个单位如何访问？
 - 硬件单位大小是：1B
 - C语言定义不同单位大小：`int,short,char,struct XXX`
- 如何识别二要素（例子：`int *p`）
 - 存放地址容量：当前操作系统决定
 - 下一个地址：加4B

```
1 //4_1
2 #include <stdio.h>
3
4 int main()
5 {
6     char *p1 = (char *)100;
7     int *p2 = (int *)100;
8
9     //要素1 存储地址大小
10    printf("%ld : %ld\n",sizeof(p1),sizeof(p2));
11
12    //要素2 访问的下一个地址是多少
13    printf("%p : %p\n",p1,p2);
14    printf("%p : %p\n",p1 + 1,p2 + 1);
15 }
```



5.描述空间的强化（多维空间）

技巧：定位、先右看、再左看

- `int a1`
定位a1,右看没有,左看为int 则为一个int的变量
- `int a2[5]`
定位a2,右看为[5],左看为int 则为有5个int的变量
- `int *a3`
定位a3,右看没有,左看为*,为指针，右看没有,再左看为int 则为下一个地址为int长度的指针

高级变形

- `int *a4[5]`
定位a4,右看为[5],左看为*,为指针，右看没有,再左看为int 则有5个下一个地址为int长度的指针
- `int (*a5)[5]`
定位a5,右看没有(括号里面没有),左看为*,为指针,右看[5],再左看为int 则下一个地址为5个int长度的指针
- `int a6[3][4]`
定位a6,右看[3] [4],左看为int,则为有3行4列个int的变量
- `int *a7[3][4]`
定位a7,右看为[3] [4],左看为*,为指针右看没有,再左看为int 则有3行4列下一个地址为int长度的指针
- `int (*a8)[3][4]`
定位a8,右看没有(括号里面没有),左看为*,为指针,右看[3] [4],再左看为int 则下一个地址为3行4列个int长度的指针

设计一个指针，可以存储二维空间或三维空间的首地址

```
1 /5_1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a[3][4];
7     int *k1;
8
9     k1 = a;
10    printf("a = %p, a + 1 = %p\n",a,a + 1);
11    printf("k = %p, k + 1 = %p\n",k1,k1 + 1);
12    return 0;
13 }
```

```
seed@server:~/c/5$ ./5_1
a = 0x7ffe251206d0, a + 1 = 0x7ffe251206e0
k = 0x7ffe251206d0, k + 1 = 0x7ffe251206d4
```

解释：a为二维空间**3行4列** 应该下一个地址为**每一行的下一个地址**，所以下一个地址长度为4*4B=16B,而赋值给k1为下一个地址为4B,所以为上述的结果

正确写法：

```
1 //5_1
2 #include <stdio.h>
3
4 int main()
5 {
6     int a[3][4];
7     int (*k1)[4];
8     //int *k1;
9
10    k1 = a;
11    printf("a = %p, a + 1 = %p\n",a,a + 1);
12    printf("k = %p, k + 1 = %p\n",k1,k1 + 1);
13    return 0;
14 }
```

```
seed@server:~/c/5$ ./5_1
a = 0x7ffc4ac34b80, a + 1 = 0x7ffc4ac34b90
k = 0x7ffc4ac34b80, k + 1 = 0x7ffc4ac34b90
```

三维空间：

```
1 //5_2
2 #include <stdio.h>
3
4 int main()
5 {
6     int a[3][4][5];
7     int (*k)[4][5];
8
9     k = a;
10    printf("a = %p, a + 1 = %p\n",a,a + 1);
11    printf("k = %p, k + 1 = %p\n",k,k + 1);
12    return 0;
13 }
```

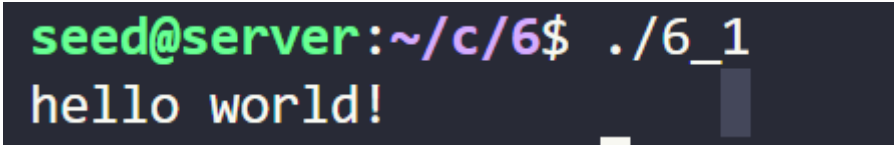
```
seed@server:~/c/5$ ./5_2
a = 0x7fffabafb470, a + 1 = 0x7fffabafb4c0
k = 0x7fffabafb470, k + 1 = 0x7fffabafb4c0
```

解释：a为二维空间**3排4行5列（三维）** 应该下一个地址为**每一排的下一个地址**，所以下一个地址长度为 4*4*5B=80B

6.如何描述和访问一个函数空间

1. 函数是空间
 - 函数名就是这个空间地址的常量的代号
2. 如何用一个变量保存这个代号
 - 保存这个代号，首先必须是一个地址

```
1 //6_1
2 #include <stdio.h>
3
4 int main()
5 {
6     //int printf (const char *__restrict __fmt, ...)
7     int (*myshow)(const char *__restrict __fmt, ...);
8     myshow = printf;
9     myshow("hello world!\n");
10    return 0;
11 }
```



```
seed@server:~/c/6$ ./6_1
hello world!
```

解释：跟翻译一个变量一样 **技巧：定位、先右看、再左看**，定位myshow,右看没有(括号里面没有),左看为*,为指针,右看(),再左看为int 则参数为 (const char *__restrict __fmt, ...) 返回值 int 的函数指针

案例分析

看如下代码,怎么实现case里面不需要改变，按照用户输入时候3天小计划来改变呢？

```
1 //6_2
2 #include <stdio.h>
3
4 void do_music()
5 {
6     printf("play music!\n");
7 }
8
9 void do_game()
10 {
11     printf("play game!\n");
12 }
13
14 void do_book()
15 {
16     printf("play book!\n");
17 }
18
19
20 int main()
21 {
22     int day;
23     printf("input day:");
24     scanf("%d",&day);
25
26     switch (day){
27         case 1:
28             do_music();
29             break;
30         case 2:
31             do_game();
32             break;
33         case 3:
34             do_music();
35             break;
36         default:
37             break;
38     }
39
40     return 0;
41 }
```

解答：由前面函数指针结合数组，可以想到用函数指针数组来实现

```
1 //6_3
2 #include <stdio.h>
3
4 void do_music()
5 {
6     printf("play music!\n");
7 }
8
9 void do_game()
```



```

10 {
11     printf("play game!\n");
12 }
13
14 void do_book()
15 {
16     printf("play book!\n");
17 }
18
19
20 int main()
21 {
22     //定义一个数组空间，保存key,每把钥匙都是函数行为
23     void (*events[3])(void);
24     //用户负责输入
25     events[0] = do_game;
26     events[1] = do_book;
27     events[2] = do_music;
28
29     int day;
30     printf("input day:");
31     scanf("%d",&day);
32
33     //执行day天做的事
34     events[day%3]();
35
36     return 0;
37 }

```

```

seed@server:~/c/6$ ./6_3
input day:1
play book!

```

7.空间的属性

空间是可以任意访问的吗?

```

1 //7_1
2 #include <stdio.h>
3
4 int main()
5 {
6     char *s = "Aello";
7
8     s[0] = 'H';
9     printf("the s is %s\n",s);
10
11     return 0;
12 }

```

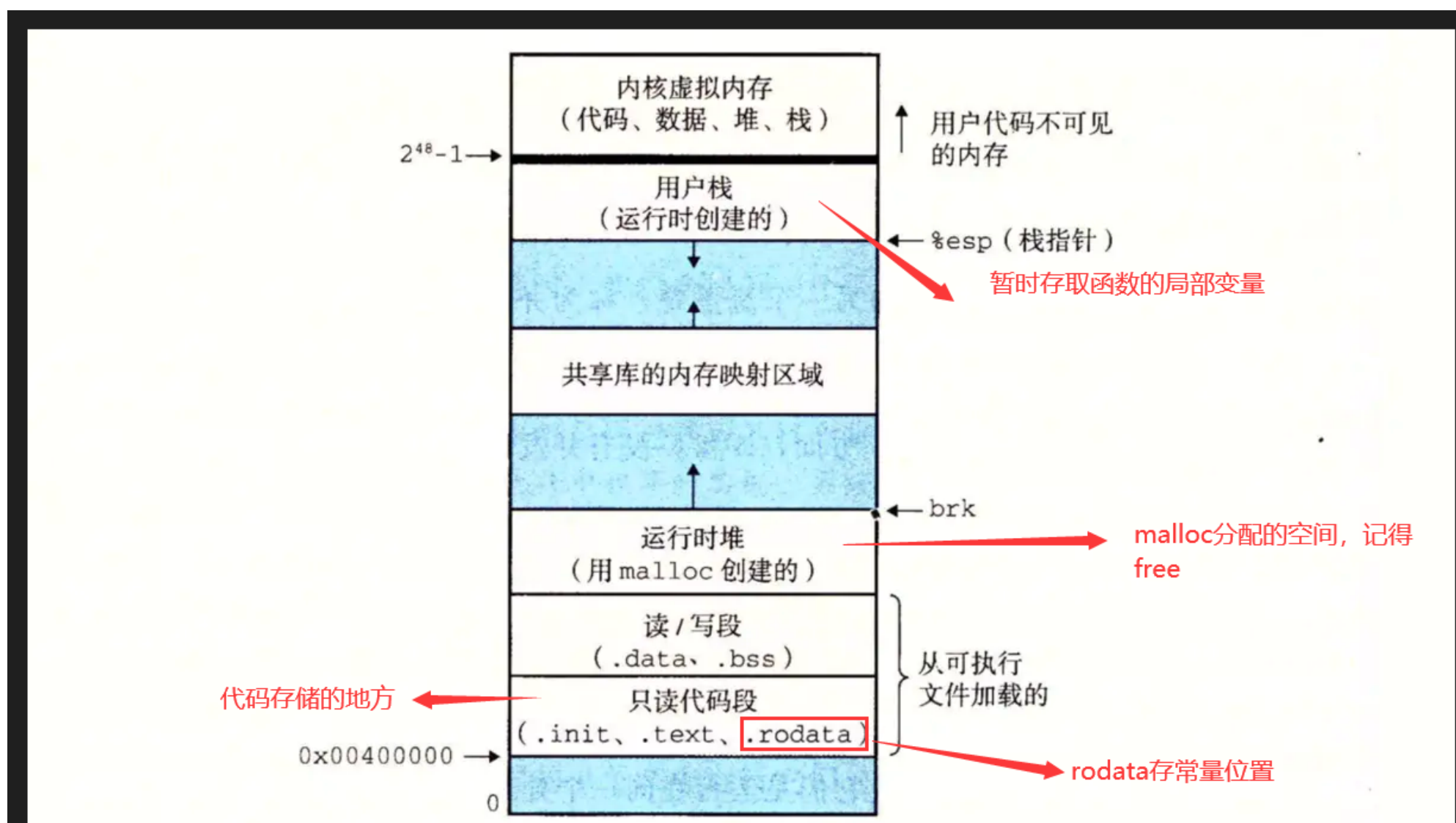
```

seed@server:~/c/7$ ./7_1
Segmentation fault

```

为什么会报段错误?

"Aello"为字符串常量，由于常量放在.rodata段为只读段，s指针指向了只读代码段，因此修改会发生段错误。段的读写权限为操作系统管理



不同的变量，默认定义在不同的段内

- 函数代码：代码段 (.text段)
- 字符串、常量值：只读数据段 (.rodata段)
- 全局的变量：数据段 .data段
- 堆段：malloc申请的，必须通过free释放
- 栈段：函数的局部变量，函数返回后，出栈释放
- static的变量：静态数据段 (.data段)

```

1 //7_2
2 #include <stdio.h>
3
4 int main()
5 {
6     char s[] = "Aello";
7
8     s[0] = 'H';
9     printf("the s is %s\n",s);
10
11     return 0;
12 }

```

```

seed@server:~/c/7$ ./7_2
the s is Hello
seed@server:~/c/7$ 

```

为什么成功了?

解释：因为是s[]相当于给一个数组，然后“Aello”赋值给s[]这个数组了，为局部变量，可读可写

```

1 //7_3
2 #include <stdio.h>
3 int abc;
4 int func(){
5     int *k = malloc(1024);
6
7     free(k);
8     abc;
9 }
10
11 int main()
12 {
13     fun();
14     abc;
15 }

```

k是fun函数malloc的然后要free abc是全局变量在.data段

```

1 //7_4
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int fun(){
6     static int a = 10;
7     a++;
8     printf("%d\n",a);
9     return 0;
10 }
11
12 int main()
13 {
14     //这里能访问a吗?
15     //printf("%d\n",a); 这里编译报错了
16     //访问a必须可以拥有他的地址
17     fun();
18
19     //这里能访问a吗?
20     //printf("%d\n",a);
21     fun();
22     fun();
23     return 0;
24 }

```

 QQ截图20220721180538

解释：这里相当于a只初始化一遍，static在静态数据段；但是主函数访问不到a,然后呢可以通过地址访问到

空间访问权限要考虑那些？

- 读写权限
 - 定义空间时候，定义在哪里（RW） ---操作系统
- 边界标志
 - 无越界检查 ---程序员来检查
 - 如何定义边界 （数量和特殊结束标志）

```

1 //7_5
2 #include <stdio.h>
3
4 int main(){
5     int a = 0x12345678;
6     char buf[4];
7
8
9     buf[0] = 0x11;
10    buf[1] = 0x22;
11    buf[2] = 0x33;
12    buf[3] = 0x44;
13    buf[4] = 0x99;
14
15    printf("the a is %x\n",a);
16    return 0;
17 }

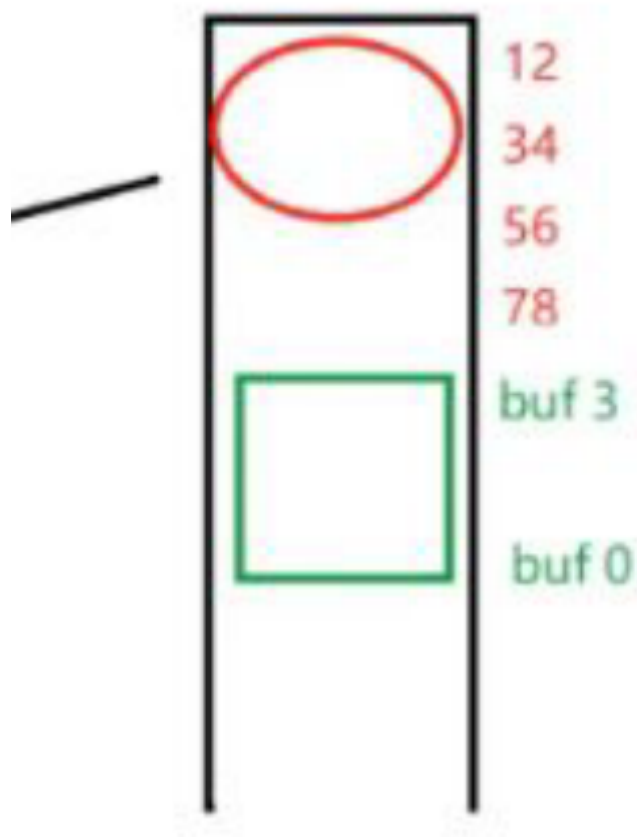
```

```

seed@server:~/c/7$ ./7_5
the a is 12345678
*** stack smashing detected ***: terminated
Aborted

```

linux报错了缓冲区溢出，数组越界了,可能其他操作系统会改变a的值



因为都是局部变量在栈上如果溢出数组的话可能会改变a的值

注：c中const与c++中的const

```
1 #include <stdio.h>
2
3 int main(){
4     const int a = 0x12345678;
5     char buf[4];
6
7
8     buf[0] = 0x11;
9     buf[1] = 0x22;
10    buf[2] = 0x33;
11    buf[3] = 0x44;
12    buf[4] = 0x99;
13
14    printf("the a is %x\n",a);
15    return 0;
16 }
```

可能其他操作系统会改变a的值,因为c语言 建议 const是常量（建议所以可以改变），但是c++一定是常量

8.C语言特殊的空间行为：字符串空间

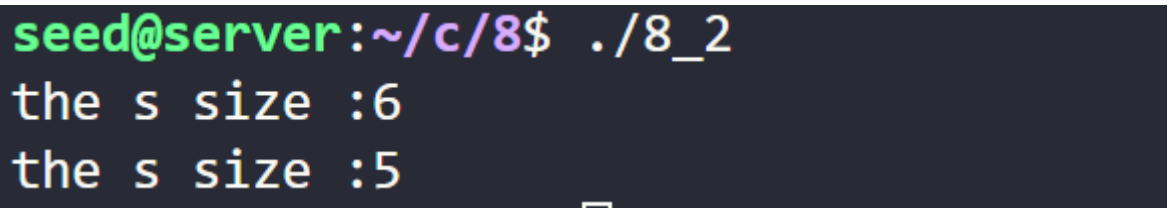
字符串是以 `\0` 未结束标记符的

```
1 //8_1
2 #include <stdio.h>
3
4 int main()
5 {
6     char *s1 = "12345";
7
8     printf("the s1 size %ld; the str size:%ld\n",
9           sizeof(s1),sizeof("12345"));
10
11    return 0;
12 }
```

```
seed@server:~/c/8$ ./8_1
the s1 size 8; the str size:6
```

s1存储的空间大小为8B,字符串大小为6B(有一个结束符'\0'占一个字节)

```
1 //8_2
2 #include <stdio.h>
3 #include <string.h>
4 int main()
5 {
6     char s[] = "12345";
7
8     printf("the s size :%ld\n",sizeof(s)/sizeof(s[0]));
9     printf("the s size :%ld\n",strlen(s));
10    return 0;
11 }
```



求数组的长度: sizeof(s)/sizeof(s[0])

求字符串长度(不算'\0'): strlen(s)

一般的遍历字符串手段

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char s[] = "12345";
6
7     char *temp = s;
8
9
10    int cnt = 0,i = 0;
11    //遍历字符串空间
12    while(temp[i]){
13        //每一个元素处理方式
14        if(temp[i] == '4') cnt++;
15        //遍历下一个位置
16        i++;
17    }
18    //处理完,后续事件
19    printf("the cnt is %d\n",cnt);
20    return 0;
21 }
```

函数的设计

9.函数的本质

- 1. 函数名
 - 程序员编写代码空间内的名称，本质是一个地址（常量）
 - int fun(int a,int b); 声明一个地址常量
 - fun(10,20);使用这个地址，访问代码空间
 - 非常特殊的地址
 - 可接受地址 (几个？ 每个都是什么类型)
 - 可以返回信息（返回1个信息，什么类型？）
 - 定义一个变量来保存不同的变量
 - int (*p)(int,int)
 - 利用typedef便于程序员进行阅读

```
1 int time;
2 //time 是 int 数据类型的变量
3 typedef int time_t;
4 //time_t 是 int 数据类型的别名
5
6 //函数别名
7 typedef int (*show_handler)(const char *__format,...);
8 show_handler myshow;
```

使用函数别名

```

1 //9_1
2 #include <stdio.h>
3
4 typedef int(*show_t)(const char *__format, ...);
5
6 int main()
7 {
8     //int (*myshow)(const char *__format, ...);
9     show_t myshow;
10    myshow = printf;
11    myshow("hello world\n");
12    return 0;
13 }

```

2. 函数承上启下的作用

- 从调用者处拷贝到函数空间（就这一种）
- 函数参数
 - 值拷贝
 - 地址拷贝
- 传递地址的含义
 - 反向更新
 - 连续空间的查看
 - 修改连续空间的值

交换两个数

```

1 //9_2
2 #include<stdio.h>
3
4 void myswap(int a1,int a2)
5 {
6     int temp;
7     temp = a1;
8     a1 = a2;
9     a2 = temp;
10    printf("swap: a1 = %d,the a2 = %d\n",a1,a2);
11 }
12
13 int main()
14 {
15     int a = 20;
16     int b = 30;
17
18    myswap(a,b);
19    printf("the a = %d,the b = %d\n",a,b);
20    return 0;
21 }

```

```

seed@server:~/c/9$ ./9_2
swap: a1 = 30,the a2 = 20
the a = 20,the b = 30

```

这里是值传递，相等与在myswap的函数空间重新拷贝了a,b两个值然后只是交换了myswap的函数空间的a和b，没有改变mian函数空间的

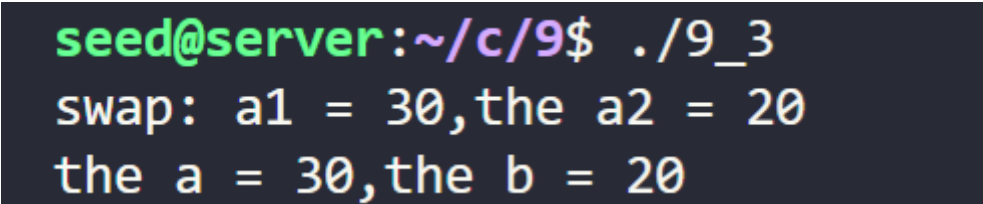
```

1 //9_3
2 #include<stdio.h>
3
4 void myswap(int *a1,int *a2)
5 {
6     int temp;
7     temp = *a1;
8     *a1 = *a2;
9     *a2 = temp;
10    printf("swap: a1 = %d,the a2 = %d\n",*a1,*a2);
11 }
12
13 int main()
14 {
15     int a = 20;
16     int b = 30;
17
18    myswap(&a,&b);
19    printf("the a = %d,the b = %d\n",a,b);

```



```
20     return 0;
21 }
```



这里为啥交换了,可以发现传递的是地址，而这个地址指向的是main函数空间的a和b,可以通过地址来改变a,b的值

思考：可以想成其实是地址是值拷贝，这里常说的是地址拷贝；想想scanf()是不是也是地址拷贝

```
1 int a;
2 scanf("%d",&a);
```

总结：

	基础数据类型	连续空间
看	值拷贝	const
改	地址拷贝	地址和大小

以后就可以看参数就知道函数对参数干啥的

```
1 int fun(int); //看基础数据类型
2 int fun(int *);//改基础数据类型
3 int fun(const int *);//看连续的空间
4 int fun(int *,int num);//改连续空间（一定要传空间的大小！！）
```

有趣的考题

```
1 int fun(int a[1000])
2 {
3     printf("%d",sizeof(a)); //8个字节
4 }
```

这里为什么是8个字节（64位操作系统）,传参的时候已经退化位地址了，1000只是告诉空间的大小

空间		
字符空间	结束有标志 '\0'	可以不限定长度
非字符空间	没有结束标志	限定长度

3. 字符串：

- 读 `const char *`
- 改 `char *`

写一个strcpy函数的声明

从一个源地址 拷贝到 目的地址

源地址只用读

目的地址要改写

```
char buf[1024];
strcpy(buf,"hello world");
```

```
1 void strcpy(char* dst,const char* src);
```

如果考虑到src的长度比dst长度要长，可能复制不了

也可以在参数加一个size

```
1 void strncpy(char *dst,const char* src,size_t len);
```

可以看到string.h的strcpy和strncpy的设计思想就是这样的

4. void的说明

希望函数能一个Byte一个Byte或者一个int一个int的访问地址，但是c语言没有函数重载，那只能写两个函数

但是void传递的参数不知道是什么类型，再函数里面可以强转

`void *p` p不合法，没有明确指定操作行为，作为形参，接受地址用的

```
1 void fun(void *addr)
2 {
3     char *p = (char *)a;//一个Byte一个Byte
4     int *p = (int *)a;//一个int一个int
5 }
6 int main()
7 {
8     struct adc data[10];
9     fun(data);
10 }
```

总结:

```
1 //连续数值空间
2 int fun(int); //看基础数据类型
3 int fun(int *);//改基础数据类型
4 int fun(const int *);//看连续的空间
5 int fun(int *,int num);//改连续空间（一定要传空间的大小!!!）
6 //连续的字符空间
7 int fun(const char *);//读字符空间
8 int fun(const *,[int num]);//改字符空间
9 //连续的数据空间
10 int fun(const void*,int num);//读数据空间
11 int fun(void*,int num);//改数据空间
12 void *memcpy(void * dst,const void* src,size_t count);
13 void *memset(void * dst,int val,size_t count)
```

5. malloc和free配套函数

```
1 char *p = malloc(1024);
2 free(p);
```

6. 函数返回值

乘上启下

- 输入:
- 输出: 输入的参数的方向更新 返回值

返回值

- 基础类型 int
- 地址 信息量变大
- 结构体

返回后，函数的空间 消失

```
1 int *fun(){
2     int a = 100;
3     return &a;
4 }
5 int main()
6 {
7     int *res;
8
9     res = fun();
10    *res = xx;
11 }
```

 a的生命周期在fun，fun调用完后就释放掉a，返回a的地址相当于指向是一个无效地址，相当于指向是一个僵尸空间

返回地址：地址那些是有效的 调用者如何使用

- 代码区
- 常量区
- 数据区 (全局变量，static变量)：有效
- 堆区： malloc 靠free :有效
- 栈区： 局部变量 有效

```
1 //9_4
2 #include <stdio.h>
3 #include <stdlib.h>
4 char * fun1(int flags)
5 {
6     //static char buf[32]; //static有效  不用free
7     char *buf;
8     buf = (char *)malloc(32);
9
10    snprintf(buf,32,"====%d====",flags);
11    return buf;
```

```
12 }
13
14 void free_fun(void *p)
15 {
16     free(p);
17 }
18 int main()
19 {
20     char *res;
21
22     printf("=====\n");
23     res = fun1(10);
24     printf("%s\n",res);
25     free_fun(res);
26
27     res = fun1(20);
28     printf("%s\n",res);
29     free_fun(res);
30     return 0;
31 }
```

static的返回地址有效，位于data段

malloc函数分配的地址一定free函数

如:没有传入指针但是，返回指针一般为类molloc函数，一般有一个配套的分类free函数

```
FILE *fopen(const char *_Filename,const char*_Mode)

int fclose(FILE *_File)
```

面试题

指针声明相关题目

数组 有多少个 每个元素是什么类型

地址 占用多大空间 4B/8B

空间如何操作

1. int (*s[10])(int) 表示的是什么意思

规则:右看然后左看

右看是数组，左看是指针，则为数组指针，再右看为函数指针，传入参数为int,传出参数为int

即 为函数指针传入参数为int,传出参数为int的10个函数指针数组

2. 用变量a给出下面的定义

- 一个有10个指针的数组，该指针是指向一个整型数的

```
int *a[10];
```

- 一个指向有10个整型数数组组的指针

```
int (*a)[10];
```

- 一个指向函数的指针，该函数有一个整型参数并返回一个整型数

```
int (*a)(int)
```

- 一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数

```
int (*a[10])(int)
```

3. char *const p1;const char *p;const char const *p3 上述三个变量有什么区别？

```
char *const p1
```

p1是 **只读**的地址 这个地址指向的内容 **可读可写** 一个char一个char的操作内存

```
const char * p2 或 char const *p2
```

p1是 **可读可写**的地址 这个地址指向的内容 **只读** 一个char一个char的操作内存

```
const char const *p3
```

p1是 **只读**的地址 这个地址指向的内容 **只读** 一个char一个char的操作内存

指针概念相关题目

1. 在32bit编译器下，运行结果：

```
1 char str[] = "Hello";
2 char *p = str;
3 int n = 10;
4 //sizeof(str) = ( 6 ); //字符串空间大小,加上'\0'的长度
5 //sizeof(p) = ( 4 ); //指针的空间
6 //sizeof(n) = ( 4 ); //32位int
7 void func(char str[100])
8 {
9     // sizeof(str) = ( 4 );//出入时候已经退化为指针了
10 }
```

2. 已知代码如下，填空：

```
1 unsigned char *p1;
2 unsigned char *p2;
3 p1 = (unsigned char *)0x801000;
4 p2 = (unsigned long *)0x810000;
5 //注意是16进制加减 (20) 10 = (14)16
6 //p1 + 5 = (0x801005),p2 + 5 = (0x810014);
```

字符空间操作类题目

1. 如下代码运行后产生什么结果？为什么？

```
1 char str[10];
2 strcpy(str,"0123456789");
```

有一个字节'\0'写入str+10的地址，这个地址不知道会是代表什么，可能会让程序崩溃！！

2. 请写出以下str变量的sizeof、strlen的大小：

```
1 char str[] = “12345”;
2 //大小为6,字符串天生带'\0'
3 char str[10] = {'1','2','3','4','5'};
4 //大小为10
5 char str[] = {'1','2','3','4','5'};
6 //大小为5
```

3. 以下代码运行后会出现什么问题？打印结果是什么？

```
1 void main() {
2     char abc[10];
3     printf(“%d”, strlen(abc));
4 }
```

由于局部变量是放在栈上的，栈上的数据初始化可能不知道是初始化这10个字节的数据，因此什么长度都可能发生(strlen这个函数只要遇到'\0'统计结束)；
如果是全局变量初始化，则这10个字节为全0即'\0'，则返回长度为0

结构体对齐类题目

有关结构体对齐原因：**提高cpu取数的有效性**

地址为直接的整数倍，存储的数据不能换行（cpu一次32位或者64位）

下面的题目假设为32bit系统

1. 给定结构体，请问他的sizeof大小？

```
1 struct data {
2     char t1; //一个字节
3     char t2; //一个字节
4     unsigned short t3;//两个字节
5     unsigned long t4; //4个字节
6 };
7 //sizeof(data)=(8)
```

t1	t2	t3	t3
t4	t4	t4	t4

2. 给定结构体，请问他的sizeof大小？

```
1 struct data {
2     char t1;//一个字节
3     int t2;//四个字节
4     short t3;//两个字节
5 };
6 //sizeof(data) = (12)
```

t1	x	x	x
t2	t2	t2	t2
t3	t3	x	x

最后一行的两个字节也算，因为这个结构体结束了，这一行也结束了下一行给下一个结构体用

指针访问类题目

1. 下面程序的运行结果是：

```
1 int main() {
2     int a[5] = {1, 2, 3, 4, 5};
3     int *ptr = (int *)(&a+1);
4     printf("%d %d\n", *(a+1), *(ptr-1));
5 }
```

int a[5] ;

int *k = a ;

int (*k2)[5] = &a

难点：a是指向一个int一个int的地址，而&a是一个指向5个int5个地址的指针，相当于一个二维指针

则(&a + 1)为下一地址为5个int地址然强制转化为下一个地址为一个int的指针， **所以这里的 + 1为加了指向5个int的地址**

ptr的减1， ptr为一个int一个int的指针， **减了一个指向int的指针**

*(ptr - 1) = 5

*(a + 1) = 1

2. 请问下列代码有什么问题？

```
1 int main() {
2     char a;
3     char *str = &a;
4     strcpy(str, "hello");
5     printf("%s",str);
6     return 0;
7 }
```

str为指向a的指针，但a的空间为为一个字节，越界了， 如果没写入到危险的越界的区域，那么可以打印 hello ,返回时候可能会发生段错位

3. 下面程序的运行结果是：

```
1 char *s = "AAA";
2 printf("%s\n", s);
3 s[0] = 'B';
4 printf("%s\n", s);
```

由于s指向的常量字符串在只读区域，操作系统会保护，修改s[0]会发生段错误；

如果printf("%s\n", s);加了 换行符就会输出否则不会输出这一行字符串

由于pritf是先输出到一个缓冲区里面，然后最后输出到屏幕上的，如果遇到了换行符也会直接输出到屏幕上

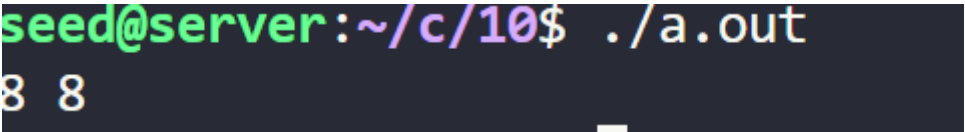
4. 下面的程序运行结果是（ ）

```
1 01234720.pngint arr[] = {6, 7, 8, 9, 10};
2 int *ptr = arr;
3 *(ptr++) += 123;
4 printf("%d %d\n", *ptr, *(++ptr));
```

由于栈的缘故，可能会倒着执行，与人的从左往右执行不一样

先执行*(++ptr),后执行 *ptr

所以打印 8 8



函数承上启下功能类题目

1. 如下代码运行后会产生什么结果？为什么？

```
1 void GetMemory(char *p) {
2     p = (char *)malloc(100);
3 }
4 void Test(void) {
5     char *str = NULL;
6     GetMemory(str);
7     strcpy(str, "hello world");
8     printf("%s\n", str);
9 }
```

str还是NULL

记住修改某个值需要传递这个变量的地址，代码形式如下

```
1 int func(xxx *x) //这里取*
2 {
3     x = ;
4 }
5 int main()
6 {
7     xxx x;
8     func(&x) //这里取地址&
9 }
```

所以上面程序的xxx换成char,然后传入时候取地址

```
1 void GetMemory(char* *p) {
2     p = (char *)malloc(100);
3 }
4 void Test(void) {
5     char *str = NULL;
6     GetMemory(str);
7     strcpy(str, "hello world");
8     printf("%s\n", str);
9 }
```

记住在调用函数时不传变量的地址不可能改变这个变量的值

2. 下面的代码有什么问题

```
1 char *GetMemory(void) {
2     char p[] = "hello world";
3     return p;
4 }
5 void Test(void) {
6     char *str = NULL;
7     str = GetMemory();
8     printf("%s\n", str);
9 }
```

编译时候都会报错误，因为p字符串数组为局部变量，GetMemory函数生命周期一完就收回了；

```
1 #include <stdio.h>
2 #include <string.h>
3 char *GetMemory(void) {
4     char *p = "hello world";
5     return p;
6 }
7 int main(void) {
8     char *str = NULL;
9     str = GetMemory();
10    printf("%s\n", str);
11    return 0;
12 }
```

想一想为什么这个可以打印hello world？

3. 如下代码运行后会产生什么结果？为什么？


```
1 void GetMemory(char **p, int num){
2     *p = (char *)malloc(num);
3     printf("*p=%p\n", *p);
4 }
5 void Test(void) {
6     char *str = NULL;
7     GetMemory(&str, 100);
8     printf("str=%p\n", str);
9     strcpy(str, "hello");
10    printf("%s\n", str);
11 }
```

```
seed@server:~/c/10$ ./a.out
*p=0x55f8d46592a0
str=0x55f8d46592a0
hello
```

malloc分配的空间在堆上，作用空间为全局的，但是要记得有malloc就有free

4. 下面的代码有什么问题

```
1 void Test(void){
2     char *str = (char *) malloc(100);
3     strcpy(str, "hello");
4     free(str);
5     if(str != NULL){
6         strcpy(str, "world");
7         printf("%s\n", str);
8     }
9 }
```

free这个函数不会把str指针的值清空，但是会把申请的那块空间回收，但是这个指针还是指向那个空间，所以一般free后接一条str=NULL,strcpy会执行

```
seed@server:~/c/10$ ./a.out
world
```

虽然还了那个空间，还是可以根据地址来修改，可能后来有的申请到这个空间，也会修改这个空间，造成不可避免的麻烦

游程编码

已知数据 0x11 0x11 0x11 0x11 0x22 0x22 0x33 0x33 0x33，使用游程编码 后，数据变为 0x11 0x04 0x22 0x02 0x33 0x03 设计一个编码函数，实现上述功能：

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 char *rlc_encode(char *data,int *len)
5 {
6     char *res = NULL;
7     int i;
8     int data_len = *len;
9     int cnt = 1; //计数器，统计有多少前后有多少个不一样的元素
10    char temp = data[0];
11    //计算原始数据的特定,决定分配多少空间
12    for(int i = 1;i < data_len;i++){
13        if(data[i] != temp){
14            cnt++;
15            temp = data[i];
16        }
17    }
18    //申请编码后的空间
19    res = (char *)malloc(cnt*2);
20    *len = cnt*2;
21    //填充编码空间
22    res[0] = data[0];
23    int res_index = 1;
24    temp = data[0];
25    cnt = 1;
26    for(i = 1;i < data_len;i++){
27        if(data[i] != temp){
28            res[res_index] = cnt;
29            res_index++;
```

```
30         res[res_index] = data[i];
31         res_index++;
32         cnt = 1;
33         temp = data[i];
34     }else{
35         cnt++;
36     }
37 }
38 res[res_index] = cnt; //最后一个计数填充
39 return res;
40 }
41 void rlc_release(char *data)
42 {
43     free(data);
44 }
45 int main(void) {
46     //原始数据
47     char raw_data[] = {0x11,0x11,0x11,0x11,
48                        0x22,0x22,
49                        0x33,0x33,0x33
50                        };
51     //将原始数据进行编码
52     int data_len = sizeof(raw_data)/sizeof(raw_data[0]);
53     int i;
54     //将原始数据传入编码函数 data_len传入传出参数
55     char *result = rlc_encode(raw_data,&data_len);
56     //查看结果
57     for(i = 0;i < data_len;i++){
58         printf("0x%02x ",result[i]);
59     }
60     printf("\n");
61     rlc_release(result);//记得释放
62     return 0;
63
64 }
```