

C 语言高级用法

作者：向仔州

数据类型的特殊定义.....	2
Void 的使用问题.....	9
extern 可以让 C++ 编译器编译 C 代码的时候按照标准 C 语言来编译.....	11
const 到底是不是常量.....	13
Volatile 就是告诉编译器老老实实去内存取变量的值，不要搞什么优化.....	15
Struct 结构体的一些 bug.....	15
计算结构体里面成员的偏移量.....	20
union 问题.....	21
enum 的使用方法.....	22
引号和单引号的问题.....	23
#define 的一些问题.....	24
#if #else 和 if else 的区别.....	29
编译器编译 C 代码的时候打印出的 warning , error , 是怎么实现的 ?	31
强制类型转换 , 转换的是数据还是地址 ?	33
指针使用注意事项.....	34
sizeof 使用注意.....	35
struct 结构体高级用法.....	36
数组指针 , 用来操作二维数组.....	40
指针数组 , 数组里面全部放的是指针变量.....	41
函数指针.....	43
typedef 修饰结构体的好处.....	48
定义结构体指针.....	49
二级指针一般用来指向指针数组.....	50
C 语言各种程序在内存分布的情况分析.....	52
函数递归调用.....57	C 语言字符串处理函数汇总.....59
Linux 系统下 C 文件制作静态库.....66	Linux 系统下 C 文件制作动态库.....67

数据类型的特殊定义

```
1 #include<stdio.h>
2
3 typedef int INT32;           // INT32 就是把 int 这种数据类型换
4 typedef unsigned char BYTE; // 一个好记的方式定义了一道
5
6 typedef struct _demo
7 {
8     short s;
9     BYTE b1;
0     BYTE b2;
1     INT32 i;
2 }DEMO;
3
4 void main(void)
5 {
6     INT32 i32;
7     BYTE byte;
8     DEMO d;
9
0     printf("%ld %ld\n", sizeof(INT32), sizeof(i32)); // sizeof 返回的是 long
1     printf("%ld %ld\n", sizeof(BYTE), sizeof(byte));
2     printf("%ld %ld\n", sizeof(DEMO), sizeof(d));
3
4 }
```

typedef 就是把 int 这种数据类型换
一个好记的方式定义了一道

这个 INT32 就代表了 int

```
root@ubuntu:/home/xiang/C_code# ./type
4 4
1 1
8 8
```

INT32 也就是 int 占用 4 个字节 32 位

BYTE 就是 char 占用 1 个字节 8 位

结构体 short 是 2 字节 BYTE 是 1 字节 , INT32 是 4 字节 , 也就是 short+BYTE+BYTE+INT32 = 8 字节

```
#include<stdio.h>

void main(void)
{
    auto int i=0; //auto 声明该变量在栈空间，其实正常的定义都是默认在栈空间的，所以auto多余
    register int j=0;//正常变量都试保存在栈空间的，register是该变量保存在寄存器中
    static int k=0; //static 使该变量只能在本文件下使用，其它c文件无法使用这个变量，这样其它文件定义相同变量名也不会受本文件影响
}
```

Register 定义变量要特别注意，就是你的系统是 32 位的，那么 register 最多定义 int，不能定义 long

如果你的系统是 64 位的，那么 register 可以定义 long。Register 定义的变量必须符合系统位宽

```
1 #include<stdio.h>
2
3 auto int g = 0;//我用auto把全局变量g声明在栈上面
4
5 void main(void)
6 {
7     auto int i=0; //auto 声明该变量在栈空间，其实正常的定义都是默认在栈空间的，所以auto多余
8     register int j=0;//正常变量都试保存在栈空间的，register是该变量保存在寄存器中
9     static int k=0; //static 使该变量只能在本文件下使用，其它c文件无法使用这个变量，这样其它文件定义相同变量名也不会受本文件影响
10 }
```

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
type.c:3:10: error: file-scope declaration of 'g' specifies 'auto'
    auto int g = 0; //我用auto把全局变量g声明在栈上面
               ^
root@ubuntu:/home/xiang/C_code#
```

我发现用 auto 定义全局变量报错

这是因为全局变量是放在全局区的，auto 声明的变量是放在栈上面的，全局区没有栈这个东西，你让 g 怎么放在栈上面？

```
1 #include<stdio.h>
2
3
4 void test()
5 {
6     auto int g = 0; //我用auto把全局变量g声明在栈上面
7 }
8
9 void main(void)
10{
11    auto int i=0; //auto 声明该变量在栈空间，其实正常的定义都是默认在栈空间的，所以auto多余
12    register int j=0; //正常变量都试保存在栈空间的，register是该变量保存在寄存器中
13    static int k=0; //static 使该变量只能在本文件下使用，其它c文件无法使用这个变量，这样其它文件定义相同变量名也不会受本文件影响
14}
```

因为子函数是申请在栈上面的，我在栈上用 auto 申请个变量是很正常的

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
root@ubuntu:/home/xiang/C_code#
```

你看成功编译

其实 auto 是不需要用的，系统会根据你的变量是在子函数还是在全局变量区域，来自动为你的变量进行分配，auto 只是锦上添花多余。

```
#include<stdio.h>

register int j = 0;

void main(void)
{
```

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
type.c:3:14: error: register name not specified for 'j'
register int j = 0;
               ^
root@ubuntu:/home/xiang/C_code#
```

register 变量不能放在全局区，寄存器变量只能放在栈里面

```
1 #include<stdio.h>
2
3 void test()
4 {
5 register int j = 0;
6 }
7
8 void main(void)
9 {
10}
11
12}
```

register 变量要比在栈上申请的变量读写速度快，如果我们的一个子函数要求运行很快，就可以在子函数里面多申请几个 register 变量，register 申请变量一般用上实时系统上

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
root@ubuntu:/home/xiang/C_code#
```

```
1 #include<stdio.h>
2
3
4 void main(void)
5 {
6     int z = 0;
7     register int j = 0;
8
9     printf("%p \n",&z);
10    printf("%p \n",&j);
11 }
12 }
```

register 寄存器地址是在 CPU 上面，不是在内存栈上面，在 X86 平台是不能获取地址的，在单片机里面可以，在 linux 平台下不清楚

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
type.c: In function 'main':
type.c:10:2: error: address of register variable 'j' requested
    printf("%p \n",&j);
    ^
root@ubuntu:/home/xiang/C_code#
```

```
1 #include<stdio.h>
2
3
4 void main(void)
5 {
6     int z = 0;
7     register int j = 0;
8
9     printf("%p \n",&z); //%p是输出变量的地址
10    printf("%p \n",&j);
11 }
12 }
```

Int 直接定义的变量在内存栈上

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
root@ubuntu:/home/xiang/C_code# ./type
0x7ffea1f567f4
root@ubuntu:/home/xiang/C_code#
```

只有普通变量是可以访问地址的。因为普通变量在内存栈上。

```

1 #include<stdio.h>
2
3
4 void test1()
5 {
6     int i=0;
7     i++;
8     printf("i = %d\n",i);
9 }
10
11 void test2()
12 {
13
14     static int i=0;
15     i++;
16     printf("i = %d\n",i);
17 }
18
19
20 void main(void)
21 {
22     for(int z=0;z<5;z++)
23     {
24         test1();
25     }
26
27     for(int z=0;z<5;z++)
28     {
29         test2();
30     }
31 }

```

为什么 test1 每次输出都是 1

而 test2 能累加到 5

```

root@ubuntu:/home/xiang/C_code# ./type
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 2
i = 3
i = 4
i = 5

```

test1 的输出

Test2 的输出

```

1 #include<stdio.h>
2
3
4 void test1()
5 {
6     int i=0;
7     i++;
8     printf("i = %d\n",i);
9 }
10
11 void test2()
12 {
13
14     static int i=0;
15     i++;
16     printf("i = %d\n",i);
17 }
18
19
20 void main(void)
21 {
22     for(int z=0;z<5;z++)
23     {
24         test1();
25     }
26
27     for(int z=0;z<5;z++)
28     {
29         test2();
30     }
31 }

```

那是因为 int i 是放在栈空间里面的，每次函数执行完了都会给 i 清 0

而 static int i 用 static 将 i 放在了全局区，(注意 static 静态变量和全局变量都放在全局区只是功能不一样而已。一个是局部使用，一个是全局使用)

每次进入函数发现 i 是静态变量，所以没有对 i 进行再次清 0，而是让 i 接着用

虽然 static 定义的变量作用范围在子函数内部，但是子函数运行结束后，变量是没有被释放的。虽然你在主函数可以定义同一个 i 而且不会被子函数的 i 影响，但是子函数 i 占着内存这个事实是一直存在的，只是 i 的作用位置不同，但是 i 占着内存是没有释放的。

static 还有一个非常重要的关键的作用

```
root@ubuntu:/home/xiang/C_code# ls  
type type2.c type.c  
root@ubuntu:/home/xiang/C_code#
```

我有两个 C 文件，type.c 和 type2.c

type2.c 里面有个 int 类型的变量

```
2 int z=5;
```

因为没有定义在函数里面，所以该变量是全局的

type.c 文件去调用这个全局变量

```
1 #include<stdio.h>  
2  
3 void main(void)  
4 {  
5     extern int z;  
6     printf("z = %d\n",z);  
7 }
```

这里用 extern 才能声明其它 C
文件里面的全局变量

```
root@ubuntu:/home/xiang/C_code# ./type  
z = 5  
root@ubuntu:/home/xiang/C_code#
```

我们 type.c 里面输出的 z 值是 type2.c 的值。

所以这就是全局变量的作用，它不仅在一个 C 文件里面全局，它在整个系统里面都是全局的，我根本不需要知道这个全局变量在哪个 C 文件，我只要知道有这个变量，然后用 extern 导入进我现在用的 C 文件，那么这个变量就可以给我用。所以为什么很多人说少用全局变量，就是因为这种全局变量任何文件都可以去篡改，很危险。

但是有个办法可以解决这个问题

type2.c

```
1  
2 static int z=5;
```

type.c

```
#include<stdio.h>  
  
void main(void)  
{  
    extern int z;  
    printf("z = %d\n",z);  
}
```

在 type2.c 文件里面给全局变量加 static

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c type2.c  
/tmp/ccdIJ4cG.o: In function `main':  
type.c:(.text+0x6): undefined reference to `z'  
collect2: error: ld returned 1 exit status  
root@ubuntu:/home/xiang/C_code#
```

你看编译就直接报错，说你的 type.c 文件用 extern 声明了一个不存在的全局变量。

但是我还是想使用这个全局变量怎么办？你可以用全局函数来承载这个文件内的全局变量

The image shows two terminal windows side-by-side. The left window is titled 'type.c' and contains the following code:

```
1 #include<stdio.h>  
2  
3 void main(void)  
4 {  
5     extern int test2_c();  
6     int m = test2_c();  
7     printf("m = %d\n",m);  
8 }
```

The right window is titled 'type2.c' and contains the following code:

```
1  
2 static int z=5;  
3  
4 int test2_c()  
5 {  
6     return z;  
7 }
```

```
root@ubuntu:/home/xiang/C_code# ./type  
m = 5
```

type.c 文件成功调用了 type2

文件里面的代码，从这里可以看出来函数和变量是一样的可以被局部使用，也可全局使用。所以以后写程序如果函数不是被其它文件需要调用，就尽量在函数前加 static

The image shows two terminal windows side-by-side. The left window is titled 'type.c' and contains the following code:

```
1 #include<stdio.h>  
2  
3 void main(void)  
4 {  
5     extern int test21();  
6     extern int test22();  
7     int x = test21();  
8     int y = test22();  
9     printf("x = %d\n",x);  
10    printf("y = %d\n",y);  
11 }  
12 }
```

The right window is titled 'type2.c' and contains the following code:

```
1  
2 static int z=5;  
3  
4 static int test21()  
5 {  
6     return z;  
7 }  
8  
9 int test22()  
10 {  
11     return z;  
12 }
```

A green triangle points from the 'static' keyword in the first code block to the 'static' keyword in the second code block.

在子函数前面加 static，这样该函数的作用范围就是本文件

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c type2.c  
/tmp/ccve67Vu.o: In function `main':  
type.c:(.text+0xe): undefined reference to `test21'  
collect2: error: ld returned 1 exit status  
root@ubuntu:/home/xiang/C_code#
```

你看其他文件调用 test21() 函数就会编译报错，和前面我们讲变量作用范围一个意思

```
type.c
1 #include<stdio.h>
2
3 void main(void)
4 {
5     extern int test21();
6     extern int test22();
7
8     int x = test21();
9     int y = test22();
10    printf("x = %d\n",x);
11    printf("y = %d\n",y);
12 }
```

```
type2.c
1
2 static int z=5; ←
3
4 int test21()
5 {
6     return z;
7 }
8
9 int test22()
10 {
11     return z;
12 }
```

```
root@ubuntu:/home/xiang/C_code# ./type
x = 5
y = 5
```

加了 static 的变量虽然不会成为整个系统的全局变量，但是可以成为本文件里面所有函数的全局变量

至于 extern 和 include 有什么区别呢？

extern 是提供本文件调用其它 C 文件函数的通道，比如我要调用 test21() 我就在本文件写 extern test21()，这样本文件才能调用其它文件的 test21() 函数

include 里面的函数全部是声明，不具备 extern 的功能，你发现刚才文件之间没有 include 一样的可以相互调用，所以 include 只是为了让用代码的人知道用了哪些文件里面的函数。没有什么实际的意义。

Void 的使用问题

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4 void main(void)
5 {
6     int *pI = (int *)malloc(sizeof(int));
7     char *pC = (char *)malloc(sizeof(char));
8
9     void p;
10
11 }
```

C 语言不允许有 void 类型变量

```
root@ubuntu:/home/xiang/C_code# gcc -o type type.c
type.c: In function ‘main’:
type.c:9:7: error: variable or field ‘p’ declared void
    void p;
          ^
root@ubuntu:/home/xiang/C_code#
```

C 语言中不允许有 void 类型的变量

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

void main(void)
{
    int *pI = (int *)malloc(sizeof(int));
    char *pC = (char *)malloc(sizeof(char));

    void *p = NULL;
    p=pI;
}
```

但是 C 语言运行有 void 类型的指针

我们知道 C 语言规定只有相同类型的指针才可以相互赋值，比如 `int *value = int *temp` 两边都是 int 指针，如果 `char *value = int *temp` 这样就不行

但是 void 指针可以接受任何类型的数据地址，比如 char , int , long

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4 void main(void)
5 {
6     int *pI = (int *)malloc(sizeof(int));
7     char *pC = (char *)malloc(sizeof(char));
8
9     void *p;
10
11     p = pI;
12     p = pC;
```

你看 int 赋地址给 void p
也可以 char 赋地址给 p

这样去操作 p 给*p 赋值是可以的。

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4 void main(void)
5 {
6     int *pI = (int *)malloc(sizeof(int));
7     char *pC = (char *)malloc(sizeof(char));
8
9     void *p = NULL;
10
11    pI = p; //把void *放在等号右边
12    pC = p; //把void *放在等号右边
13
14 }
```

如果是这样 void 放在等号右边，虽然 gcc 编译器让你编译通过了，但是运行会出现 bug

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

void main(void)
{
    int *pI = (int *)malloc(sizeof(int));
    char *pC = (char *)malloc(sizeof(char));
    void *p = NULL;
    pI = (int *)p; //为了解决bug右边的void类型指针必须经过强制转换，才能付给左边的数据类型
    pC = (char *)p; //等号左边是什么数据类型，就强制转换什么类型
}
```

(类型 *)地址 = 强转

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

void main(void)
{
    int *pI = (int *)malloc(sizeof(int)); //为什么这里malloc也要进行强转呢？
    char *pC = (char *)malloc(sizeof(char));
}
```

因为 malloc 原形是 extern void *malloc(unsigned int num_bytes);

malloc 返回值是 void * 也就和上面一样，void*在等号右边

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4 void main(void)
5 {
6     int *pI = (int *)malloc(sizeof(int)); //所以用int *进行强转
7     char *pC = (char *)malloc(sizeof(char)); //所以用char *进行强转
8
9 }
```

我们下面用 void 来做一个 memset 内存清 0 的函数功能

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4
5 void *my_memset(void *p,char v,int size) /*为什么用void *p来作为函数接受地址类型, */
6 {                                         /*是因为我这个函数想处理任何数据类型的值, */
7     void* ret = p;                      /*比如char , short,int都可以使用我这个函数来清0*/
8
9     char *dest = (char *)p; //我用char * 1个字节1个字节的去清0, 这样不管你是long , int, short我都给你拆分成1个字节1个字节的清0
10    //如果我用short * 那么我就是2个字节2个字节的去清0, 那么遇到传入进来的char地址, char是一个字节的,
11    //那么用short *去清0就不可能
12
13    for(int i=0;i<size;i++)
14    {
15        dest[i] = v; //v就是要清的是0还是1还是2, 我们说了是为了清0, 所以传入的v要写0
16    }
17
18    return ret;//ret指向的内存地址已经被dest代劳清0了
19 }
20
21
22
23
24 void main(void)
25 {
26     int a[5]={1,2,3,4,5};
27
28     for(int i=0;i<5;i++)
29     {
30         printf(" %d ",a[i]);
31     }
32
33     my_memset(a,0,sizeof(a));
34     printf("\n");
35
36     for(int i=0;i<5;i++)
37     {
38         printf(" %d ",a[i]);
39     }
40     printf("\n");
41 }
42
43 }
```

在 linux 驱动中 void* 就是两个 C 文件 , A.c 随便定义个结构体 , 传给 B.c , B.c 用 void* 来接收这个 A.c 传过来的数据 , 然后 B.c 再定义一个和 A.c 一样的结构体去装载 void* 接收的数据 , 这样 b.c 就可以操作 A.c 传过来的数据了 , 这样的好处是 A.c 可以定义不同的变量和数据结构来传给 B.c , 比较灵活

```
root@ubuntu:/home/xiang/C_code# ./type
 1  2  3  4  5
 0  0  0  0  0
```

这就是 void 的应用实例。

extern 可以让 C++ 编译器编译 C 代码的时候按照标准 C 语言来编译

我们前面知道 extern 可以用来声明全局变量和全局函数 , 但是 extern 还有个用处就是在 C++ 代码里面添加 C 语言代码时 , 它可以让 C++ 编译器再编译 C 代码这部分按照标准 C 代码来编译 , 避免出现 bug

```
1 #include<stdio.h>
2
3 extern "C"
4 {
5     int add(int x,int y)
6     {
7         int num=x+y;
8         return num;
9     }
10 }
11
12 void main()
13 {
14     int i=0;
15     i = add(1,2);
16     printf("i = %d\n",i);
17     return 0;
18 }
```

这样用 gcc 编译是不行的 ,
因为 extern "C" {}
这种结构是给 C++ 编译器用
的 , 在 C++ 代码里面嵌入 C
代码用这种方式包裹

```
root@ubuntu:/home/xiang/C_code# gcc -o exter1 exter1.c
exter1.c:3:8: error: expected identifier or '(' before string constant
extern "C"
^
exter1.c: In function 'main':
exter1.c:15:6: warning: implicit declaration of function 'add' [-Wimplicit-function-declaration]
    i = add(1,2);
^
```

你看 gcc 编译报错了，那么我们用 g++ 编译器来编译

```
root@ubuntu:/home/xiang/C_code# g++ -o exter1 exter1.c
exter1.c:12:11: error: '::main' must return 'int'
void main()
^
root@ubuntu:/home/xiang/C_code# g++ -o exter1 exter1.c
```

这个错误是因为 C++ 代码不允许用 void，void 是一个不确定的数据类型，C++ 对这些东西做了严格要求。就是不准你用这种不确定的东西。

```
#include<stdio.h>

extern "C"
{
    int add(int x,int y)
    {
        int num=x+y;
        return num;
    }

    int main()
    {
        int i=0;
        i = add(1,2);
        printf("i = %d\n",i);
        return 0;
    }
}
```

将 void 改成 int

```
root@ubuntu:/home/xiang/C_code# g++ -o exter1 exter1.o
root@ubuntu:/home/xiang/C_code# ./exter1
i = 3
root@ubuntu:/home/xiang/C_code#
```

这样就编译过了，可以正常使用了。这就是 extern 的另一种用法，在 C++ 里面用

const 到底是不是常量

```
1 #include<stdio.h>
2
3
4 void main()
5 {
6     const int cc = 1;
7     printf("cc = %d\n",cc);
8
9     cc = 3;
10    printf("cc = %d\n",cc);
11 }
```

我们来修改下常亮的值

发现报错，因为 cc 是常量所以是不能修改的

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
const.c: In function ‘main’:
const.c:9:5: error: assignment of read-only variable ‘cc’
  cc = 3;
  ^
root@ubuntu:/home/xiang/C_code#
```

给变量加 const 就是告诉编译器变量是只读的

```
1 #include<stdio.h>
2
3
4 void main()
5 {
6     const int cc = 1;
7     printf("cc = %d\n",cc);
8
9     int *p = (int *)&cc;
10    *p = 3;
11    printf("cc = %d\n",cc);
12 }
```

我们发现直接用指针去操作 cc 的地址，用指针赋值给 cc 是可行的

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
root@ubuntu:/home/xiang/C_code# ./const
cc = 1
cc = 3
```

从以上我们看出来 const 对编译器有用对运行程序是没有用的，所以只要绕开编译器这个关卡，其实 const 变量的值还是可以被修改的，所以我们之前去 const 变量的地址上修改数值

```
1 #include<stdio.h>
2
3
4 void main()
5 {
6     int s1=1;
7     int s2=2;
8     const int *p1 = &s1;
9     int const *p2 = &s2;
10
11    *p1 = 10;
12    *p2 = 20;
```

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
const.c: In function ‘main’:
const.c:11:6: error: assignment of read-only location ‘*p1’
  *p1 = 10;
  ^
const.c:12:6: error: assignment of read-only location ‘*p2’
  *p2 = 20;
  ^
root@ubuntu:/home/xiang/C_code#
```

我们给 const 在*左边的变量赋值

编译出现了错误，const 指针口诀，左数右值，上面就是 const 在*左边，就是左数

```
#include<stdio.h>

void main()
{
    int s1=1;
    int s2=2;
    const int *p1 = &s1;
    int const *p2 = &s2;

    p1 = NULL;
    p2 = NULL;
}
```

左数就是地址上存放的
数值不能改，但是地址
本身是可以修改的

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
root@ubuntu:/home/xiang/C_code#
```

你看我把 p1 , p2 指向 s1 , s2 的地址修改成 NULL 是没有问题的。

下面我们来看看右指是什么情况

```
1 #include<stdio.h>
2
3
4 void main()
5 {
6     int s1=1;
7     int *const p1 = &s1;
8
9     p1 = NULL;
10 }
```

const 在*号右边，就是
不能修改变量地址，但是
可以修改变量值

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
const.c: In function ‘main’:
const.c:9:5: error: assignment of read-only variable ‘p1’
    p1 = NULL;
```

```
#include<stdio.h>

void main()
{
    int s1=1;
    int *const p1 = &s1;

    *p1 = 5;

    printf("*p1 = %d\n",*p1);
}
```

const 在*号右边，你看
我不能修改地址，但是可
以修改变量值

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
root@ubuntu:/home/xiang/C_code# ./const
*p1 = 5
```

```
1 #include<stdio.h>
2
3
4 void main()
5 {
6     int s1=1;
7     const int *const p1 = &s1;
8
9     *p1 = 5;
10
11     printf("*p1 = %d\n",*p1);
12 }
```

如果一个变量*左右都有 const，那么这个
变量地址和值都不能被改变

```
root@ubuntu:/home/xiang/C_code# gcc -o const const.c
const.c: In function ‘main’:
const.c:9:6: error: assignment of read-only location ‘*p1’
    *p1 = 5;
```

Volatile 就是告诉编译器老老实实去内存取变量的值，不要搞什么优化

```
#include<stdio.h>
void main()
{
    int s1=10;
    int p2,p3;

    p2 = s1;
    p3 = s1;
    p2 = s1;
    p3 = s1;

    s1 = 5;
}
```

这么一段代码，如果 S1 一直是在给其它变量赋值的话，编译器优化的时候会认为 s1 是个永远不变的值 10

但是哪晓得一句代码改变了 S1 的值，导致编译器自作聪明让 S1 的值始终为 10，所以 S1 还是 10

这种情况有些编译器会优化成问题，有些编译器不会，看自己运气。

```
3
4 void main()
5 {
6     volatile int s1=10;
7     int p2,p3;

8     p2 = s1;
9     p3 = s1;
10    p2 = s1;
11    p3 = s1;

12
13    s1 = 5;
14 }
```

加了 volatile 之后，编译器就不会去优化自作聪明把 S1 认为一直是 10，编译器按照传统方法编译

所以加 volatile 是为了让变量保险一点。

Struct 结构体的一些 bug

```
#include<stdio.h>

struct null
{
};

void main()
{
    struct null n1;
    struct null n2;
    printf("null = %d\n", sizeof(struct null));
    printf("n1 = %d n1_addr = %0x\n", sizeof(n1), &n1);
    printf("n2 = %d n2_addr = %0x\n", sizeof(n2), &n2);
}
```

两个空结构体初始地址不一样，这是因为我们这个版本的 gcc 编译器做了改进，以前的 gcc 编译器两个空结构体地址是一样的

```
root@ubuntu:/home/xiang/C_code# ./struc
null = 0
n1 = 0 n1_addr = d7632e6
n2 = 0 n2_addr = d7632e7
```

我们发现空结构体是不占用内存空间的

所以以上定义空结构体问题在 C 语言里面是有 bug 的

```
1 #include<stdio.h>
2
3 struct null
4 {
5 };
6
7
8 int main()
9 {
10     struct null n1;
11     struct null n2;
12     printf("null = %d\n", sizeof(struct null));
13     printf("n1 = %d n1_addr = %0x\n", sizeof(n1), &n1);
14     printf("n2 = %d n2_addr = %0x\n", sizeof(n2), &n2);
15     return 0;
16 }
```

我们将 main 返回值改为 int , 这样就可以用 g++ 来编译了

```
root@ubuntu:/home/xiang/C_code# g++ -o struc struc.c
```

```
root@ubuntu:/home/xiang/C_code# ./struc
null = 1
n1 = 1 n1_addr = 8431f8e6
n2 = 1 n2_addr = 8431f8e7
root@ubuntu:/home/xiang/C_code#
```

我们发现 g++ 编译器对 C 语言进行了严格控制，给空结构体也分配了 1 个字节内存

这样就让空结构体代码安全性更高

柔性数组

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

typedef struct _soft_array
{
    int len;
    int array[];
}softarray;

int main()
{
    softarray *sa = malloc(sizeof(softarray)+sizeof(int)*10);
    sa->len = 10;

    for(int i=0;i<sa->len;i++)
    {
        sa->array[i] = i+1;
    }
    for(int i=0;i<sa->len;i++)
    {
        printf("array = %d\n",sa->array[i]);
    }

    return 0;
}
```

这就是柔性数组 , 一般数组在初始化的时候要写数组长度 , 这里不写数组长度就是柔性数组

这里分配的长度是多少呢 ?
其实我们在分配之前先看看
结构体长度到底是多大

```

typedef struct _soft_array
{
    int len;
    int array[];
}softarray;

int main()
{
    printf("softarray size = %d\n", sizeof(softarray));

    /*
    softarray *sa = malloc(sizeof(softarray)+sizeof(int)*10);
    sa->len = 10;

```

root@ubuntu:/home/xiang/C_code# ./struc
softarray size = 4

softarray 结构体长度是 4 字节

4 字节	
------	--

意思就是结构体初始化只有变量 len 的长度占了内存，array 并没有占用内存

```

typedef struct _soft_array
{
    int len;
    int array[];
}softarray;

int main()
{

    softarray *sa = (softarray *)malloc(sizeof(softarray)+sizeof(int)*10);
    printf("softarray size = %d\n", sizeof(sa));

```

root@ubuntu:/home/xiang/C_code# ./struc
softarray size = 8

malloc 已经分配内存了为什么只打印 8

这是因为 sa 是地址，相当于打印了 softarray 结构体本身长度+指针的长度
但是实际上是多分配了 40 个字节长度的

4 字节	40 字节
------	-------

你看这 40 个字节是接在 len 长度后面的。

我们打印的地址是 sa 指针自己地址的长度。

这种柔性数组有个好处就是，你可以重复调用该函数接口，每次调用都可以分配不同长度的内存空间

```

#include<stdio.h>
#include<malloc.h>

typedef struct _soft_array
{
    int len;
    int array[];
}softarray;

int main()
{
    softarray *sa = malloc(sizeof(softarray)+sizeof(int)*10);
    sa->len = 10;

    for(int i=0;i<sa->len;i++)
    {
        sa->array[i] = i+1;
    }
    for(int i=0;i<sa->len;i++)
    {
        printf("array = %d\n",sa->array[i]);
    }

    return 0;
}

```

```

root@ubuntu:/home/xiang/C_code# gcc -o struc struc.c
root@ubuntu:/home/xiang/C_code# ./struc
array = 1
array = 2
array = 3
array = 4
array = 5
array = 6
array = 7
array = 8
array = 9
array = 10

```

柔性数组实际应用，用柔性数组来做斐波拉茨数列

斐波拉茨数列数列原理 = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 //意思就是数组前两项为 1，第三项开始，每一项都是前两项的和

定义柔性数组结构 `typedef struct`

创建柔性数组 `creat_soft_array`

生成斐波拉茨数列

释放柔性数组 `delete_soft_array`

```

1 #include<stdio.h>
2 #include<malloc.h>
3
4 typedef struct _soft_array
5 {
6     int len;
7     int array[];
8 }softarray;
9
10 softarray* create_soft_array(int size) //创建柔性数组
11 {
12     softarray* ret = NULL;
13     if(size>0)
14     {
15         ret = (softarray*)malloc(sizeof(*ret)+sizeof(*(ret->array))*size); //在堆空间申请柔性数组
16         ret->len = size;
17     }
18     return ret;
19 }
20
21 void fac(softarray *sa) //生成斐波拉茨数列
22 {
23     if(sa!=NULL)
24     {
25         if(1 == sa->len)
26         {
27             sa->array[0] = 1;
28         }
29         else if(2 == sa->len)
30         {
31             sa->array[0] = 1;
32             sa->array[1] = 1;
33         }
34         else
35         {
36             sa->array[0] = 1;
37             sa->array[1] = 1;
38             for(int i=2;i<sa->len;i++)
39             {
40                 sa->array[i] = sa->array[i-1] + sa->array[i-2];
41             }
42         }
43     }
44 }

```

这种*方法就是直接分配
结构体里面数据类型的大小，
如果结构体数据类型变了，
这个函数还是可以继续给
结构体分配

```

void delete_soft_array(softarray *sa) //释放柔性数组
{
    free(sa);
}

int main()
{
    softarray *sa =create_soft_array(10);
    fac(sa);

    for(int i=0;i<sa->len;i++)
    {
        printf(" %d",sa->array[i]);
    }
    printf("\n");
    delete_soft_array(sa);

    return 0;
}

```

```

root@ubuntu:/home/xiang/C_code# ./fac
1 1 2 3 5 8 13 21 34 55

```

这就是斐波拉茨数列数列

计算结构体里面成员的偏移量

```
4
5 #define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)
6
7 struct mystruct
8 {
9     char c; //c偏移量为0
10    int i; //i偏移量为4
11    short s; //s偏移量为8
12 };
13 int main()
14 {
15     struct mystruct str;
16
17     int offsetnum = offsetof(struct mystruct,c);
18     printf("c offsetnum = %d\n",offsetnum);
19     offsetnum = offsetof(struct mystruct,i);
20     printf("i offsetnum = %d\n",offsetnum);
21     offsetnum = offsetof(struct mystruct,s);
22     printf("s offsetnum = %d\n",offsetnum);
23
24     return 0;
25 }
26
```

offsetof 内核函数，用来计算
结构体每个成员的偏移量

```
root@ubuntu:/home/xiang/C_code# ./struc
c offsetnum = 0
i offsetnum = 4
s offsetnum = 8
root@ubuntu:/home/xiang/C_code#
```

#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER) //这句代码如何分析

第一步 (TYPE *)0 : 类似(int *) 0 , 我们这里是(struct mystruct *)0 , 就是把这个 0 强制转换成存放 struct mystruct 类型结构体的地址。

第 2 步 ((TYPE *)0)->MEMBER : 这句是取结构体里面的成员名 , 类似 p->data ; 这个 p 的地址是 0. 因为我们用(TYPE *)0 先把结构体地址写成 0 了。

第 3 步 &((TYPE *)0)->MEMBER : 类似&(((TYPE *)0)->MEMBER) 把结构体某个成员名的地址取出来

```
#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)

struct mystruct
{
    char c; //c偏移量为0
    int i; //i偏移量为4
    short s; //s偏移量为8
};

int main()
{
    struct mystruct str;
    str.i = 20;

    printf("整个结构体变量的地址 = %p\n",&str); //取结构体首地址, 也就是第1个成员地址
    printf("str.i 首地址 = %p\n",&(str.i)); //取结构体某个成员地址
    printf("计算s1.i相对于整个结构体基地址的偏移量 = %d\n", (char *)&(str.i)-(char *)&str); //偏移量相差8个char
    printf("计算s1.i相对于整个结构体基地址的偏移量 = %d\n", (int *)&(str.i)-(int *)&str); //偏移量相差1个int

    return 0;
}
```

这个和用 offsetof 宏一样 , 只是为了让大家理解这个原理

union 问题

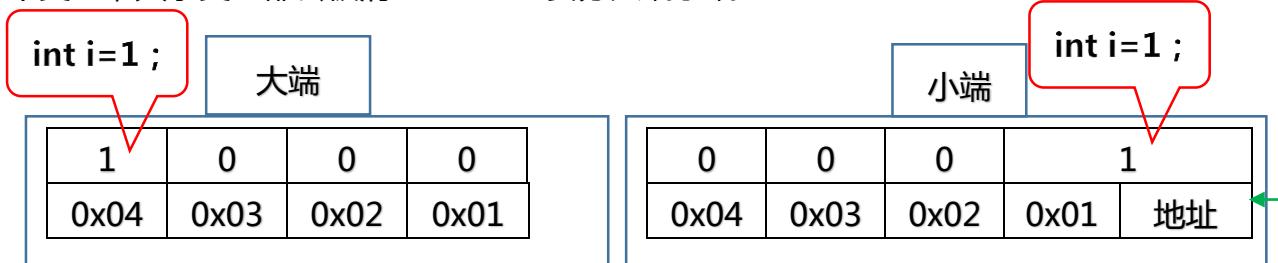
```
4 struct A
5 {
6     char a;
7     short b;
8     int c;
9 }A;
10
11 union B
12 {
13     char a;
14     short b;
15     int c;
16 }
17
18 int main()
19 {
20     struct A a;
21     union B b;
22
23     printf("A memory = %d\n", sizeof(a));
24     printf("B memory = %d\n", sizeof(b));
25
26     return 0;
27 }
```

Struct 里面的成员都要占用内存

union 里面的成员只有最大的那个变量占用内存

```
root@ubuntu:/home/xiang/C_code# ./union
A memory = 8
B memory = 4
```

union 联合体一般用在内存很小或者节省内存的系统中，因为你不管使用 union 里面的哪一个变量，其余变量都会被清 0。union 要分大端小端。



这就是大端，数据从地址高位开始向低位排列

这是小端，数据从低位向高位排列

```
4 union B
5 {
6     int i;
7     char c;
8 }B;
9
10 int main()
11 {
12     union B b;
13     b.i = 1;
14     printf("b.c = %d\n", b.c);
15
16     return 0;
17 }
```

给联合体 i 赋值

我赋值的是 i 所以 c 应该是 0

证明我们
ubuntu
系统是小
端模式

但是实际是 1

```
root@ubuntu:~/C_code# ./union
b.c = 1
```

enum 的使用方法

```
5 enum Color
6 {
7     GREEN,
8     RED,
9     BLUE,
10};
11
12 int main()
13 {
14
15     printf("GREEN = %d\n",GREEN);
16     printf("RED = %d\n",RED);
17     printf("BLUE = %d\n",BLUE);
18
19     return 0;
20 }
```

enum 类型里面的成员是从 0
开始，多一个成员自动+1

```
root@ubuntu:/home/xiang/C_code# ./enumm
GREEN = 0
RED = 1
BLUE = 2
```

enum 类型里面的成员是从 0
开始，多一个成员自动+1

```
enum Color
{
    GREEN=2,
    RED,
    BLUE,
};

int main()
{
    printf("GREEN = %d\n",GREEN);
    printf("RED = %d\n",RED);
    printf("BLUE = %d\n",BLUE);

    return 0;
}
```

如果我将没有定义数字的成
员，前一个成员定义位 2，那
么后面的成员根据前面的成
员挨着+1

```
root@ubuntu:/home/xiang/C_code# ./enumm
GREEN = 2
RED = 3
BLUE = 4
```

你看后面的成员
从 2 开始+1

```
enum Color
{
    GREEN=2,
    RED=5,
    BLUE,
};

int main()
{
    printf("GREEN = %d\n",GREEN);
    printf("RED = %d\n",RED);
    printf("BLUE = %d\n",BLUE);

    return 0;
}
```

你看第三个成员是根据第二个
成员的数向后加 1 的

```
root@ubuntu:/home/xiang/C_code# ./enumm
GREEN = 2
RED = 5
BLUE = 6
```

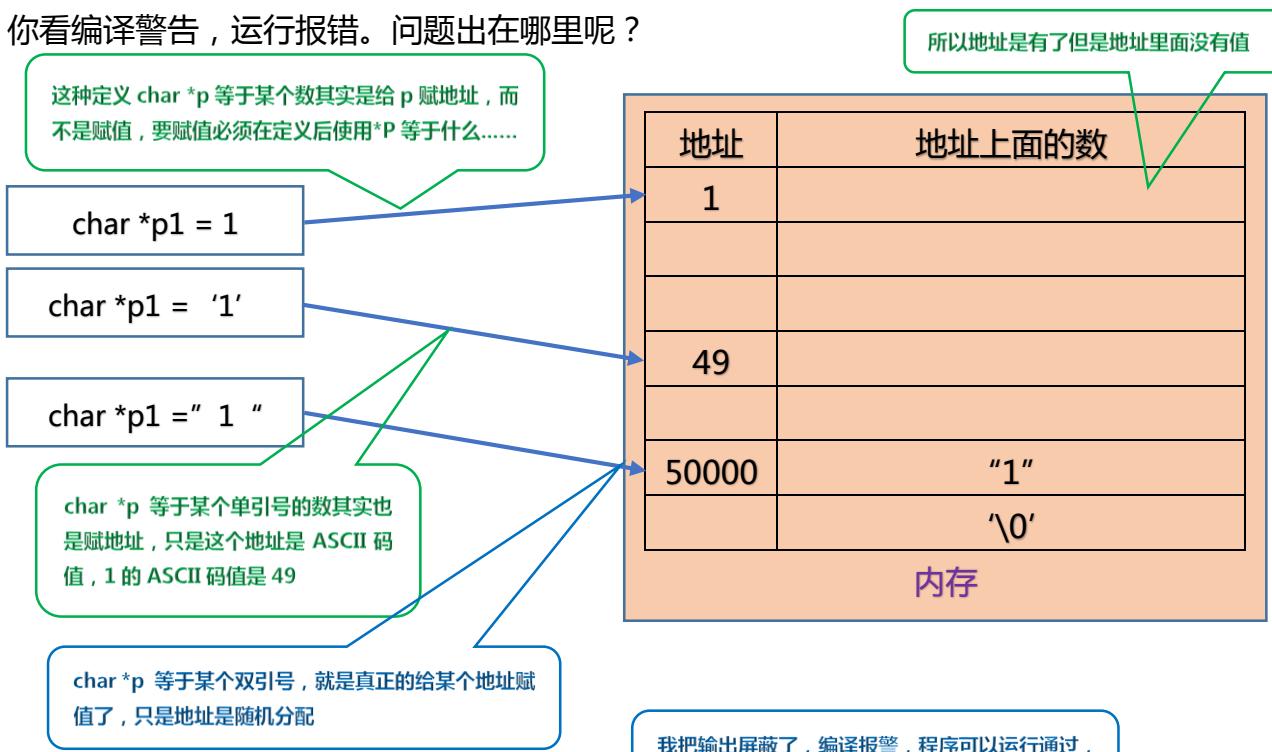
这就是枚举 enum 类型的用法

引号和单引号的问题：

```
5 int main()
6 {
7     char *p1 = 1;
8     char *p2 = '1';
9     char *p3 = "1";
10
11    printf("*p1 = %s\n",p1);
12    printf("*p2 = %s\n",p2);
13    printf("*p3 = %s\n",p3);
14
15    return 0;
16 }
17
```

```
root@ubuntu:/home/xiang/C_code# gcc -o zifu zifu.c
zifu.c: In function ‘main’:
zifu.c:7:13: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  char *p1 = 1;
             ^
zifu.c:8:13: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  char *p2 = '1';
             ^
root@ubuntu:/home/xiang/C_code# ./zifu
Segmentation fault (core dumped)
root@ubuntu:/home/xiang/C_code#
```

你看编译警告，运行报错。问题出在哪里呢？



```
5 int main()
6 {
7     char *p1 = 1;
8     char *p2 = '1';
9     char *p3 = "1";
10
11 //    printf("*p1 = %s\n",p1);
12 //    printf("*p2 = %s\n",p2);
13 //    printf("*p3 = %s\n",p3);
14
15    return 0;
16 }
```

```
root@ubuntu:/home/xiang/C_code# gcc -o zifu zifu.c
zifu.c: In function ‘main’:
zifu.c:7:13: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  char *p1 = 1;
             ^
zifu.c:8:13: warning: initialization makes pointer from integer without a cast [-Wint-conversion]
  char *p2 = '1';
             ^
root@ubuntu:/home/xiang/C_code# ./zifu
Segmentation fault (core dumped)
root@ubuntu:/home/xiang/C_code#
```

#define 的一些问题

```
4 #define pi 3.14
5 #define SUM(a,b) (a)+(b) //求和
6 #define MAX(a,b) ((a)>(b)?a:b) //拿出最大数
7 #define DIM(array) (sizeof(a)/sizeof(*a)) //求数组元素个数
8
9 int main()
0 {
1
2     printf("pi = %f\n",pi);
3     printf("SUM = %d\n",SUM(5,3));
4     printf("MAX = %d\n",MAX(5,3));
5     return 0;
6 }
```

root@ubuntu:/home/xiang/C_code# ./def
pi = 3.140000
SUM = 8
MAX = 5

这样是没有问题的

```
#define pi 3.14
#define SUM(a,b) (a)+(b) //求和
#define MAX(a,b) ((a)>(b)?a:b) //拿出最大数
#define DIM(array) (sizeof(a)/sizeof(*a)) //求数组元素个数

int main()
{
    printf("pi = %f\n",pi);
    printf("SUM = %d\n",SUM(5,3)*SUM(5,3));//按道理应该是 8*8=64
    printf("MAX = %d\n",MAX(5,3));
    return 0;
}
```

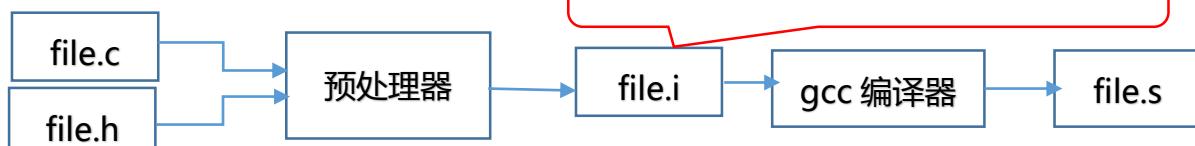
root@ubuntu:/home/xiang/C_code# ./def
pi = 3.140000
SUM = 23
MAX = 5

怎么不是 64 而是 23

这是编译器在编译过程造成的 ,

我们来看看编译器编译原理

问题出在编译到中间文件.i 文件时候造成的



我们以往都是 gcc -o file file.c 直接编译成执行文件 , 我们这次先编译-E 中间文件看看

```
root@ubuntu:/home/xiang/C_code# gcc -E def.c -o def.i
```

81 # 9 "def.c"
82 int main()
83 {
84
85 printf("pi = %f\n",3.14);
86 printf("SUM = %d\n", (5)+(3)*(5)+(3));
87 printf("MAX = %d\n",((5)>(3)?5:3));
88 return 0;
89 }

你看宏展开后是 $(5)+(3)*(5)+(3)$
很明显 $(3)*(5)$ 优先级高所以就是 23

编译出的中间文件

所以宏最好是在定义的时候直接运算出结果。

宏的第二个运算问题

```
#define pi 3.14
#define SUM(a,b) (a)+(b) //求和
#define MIX(a,b) ((a)<(b)?a:b) //求最小值
#define DIM(array) (sizeof(a)/sizeof(*a)) //求数组元素个数

int main()
{
    int i=1;
    int j=5;
    printf("MIX = %d\n",MIX(i,j));
    return 0;
}
```

求最小值

```
root@ubuntu:/home/xiang/C_code# ./def
MIX = 1
```

没有问题，如果我下面修改一下呢？

```
1 #define pi 3.14
2 #define SUM(a,b) (a)+(b) //求和
3 #define MIX(a,b) ((a)<(b)?a:b) //求最小值
4 #define DIM(array) (sizeof(a)/sizeof(*a)) //求数组元素个数
5
6 int main()
7 {
8     int i=1;
9     int j=5;
10    printf("MIX = %d\n",MIX(++i,j));//按道理说应该返回2，因为我是i先加1然后传值进宏
11    return 0;
12 }
```

```
root@ubuntu:/home/xiang/C_code# ./def
MIX = 3
```

为什么结果是 3，我们编译成.i 文件看看

```
1 # 9 "def.c"
2 int main()
3 {
4     int i=1;
5     int j=5;
6     printf("MIX = %d\n",((++i)<(j)?++i:j));
7     return 0;
8 }

4 #define pi 3.14
5 #define SUM(a,b) (a)+(b) //求和
6 #define MIX(a,b) ((a)<(b)?a:b) //求最小值
7 #define DIM(array) (sizeof(a)/sizeof(*a)) //求数组元素个数
8
9 int main()
10 {
11     int i=1;
12     int j=5;
13     printf("MIX = %d\n",MIX(i++,j));//按道理说应该返回1，因为我先获取i值然后再++
14     return 0;
15 }
```

```
root@ubuntu:/home/xiang/C_code# ./def
MIX = 2
```

为什么结果是 2

你编译成中间.i 文件就知道了。所以当宏定义放进主函数执行的时候，最好直接得到宏运算后的值，不要在主函数里面运算两个带有运算功能的宏。

宏计算数组长度的问题

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

#define pi 3.14
#define SUM(a,b) (a)+(b) //求和
#define MIN(a,b) ((a)<(b)?a:b) //求最小值
#define DIM(array) (sizeof(array)/sizeof(*array)) //求数组元素个数

int main()
{
    int arr[]={1,2,3,4,5};
    printf("array = %ld\n",DIM(arr)); //计算数组长度
    return 0;
}
```

为什么是 long，因为 sizeof 返回值是 long 型

```
root@ubuntu:/home/xiang/C_code# ./def
array = 5
计算结果没有问题，

#define DIM(array) (sizeof(array)/sizeof(*array)) //求数组元素个数
```

这个计算原理是什么？

```
int main()
{
    int arr[]={1,2,3,4,5};
    printf("sizeof(arr) = %ld\n",sizeof(arr));
    printf("sizeof(*arr)= %ld\n",sizeof(*arr));
    printf("array = %ld\n",DIM(arr)); //计算数组长度
    return 0;
}

root@ubuntu:/home/xiang/C_code# ./def
sizeof(arr) = 20
sizeof(*arr)= 4
array = 5
```

sizeof 取数组地址，就是计算整个数组占用多少个字节



#define DIM(array) (sizeof(array)/sizeof(*array))

一个数组占用内存字节数/一个数组元素占用内存的字节数 = 数组长度

```

#define DIM(array) ((array)/sizeof(*array)) //求数组元素个数

int main()
{
    char arr[]={1,2,3,4,5};
    printf("sizeof(arr) = %ld\n",sizeof(arr));
    printf("sizeof(*arr)= %ld\n",sizeof(*arr));
    printf("array = %ld\n",DIM(arr));//计算数组长度
    return 0;
}

```

```

root@ubuntu:/home/xiang/C_code# ./def
sizeof(arr) = 5
sizeof(*arr)= 1
array = 5

```

我改成 char , sizeof 取数组指针一个元素就是一个字节

内存

1	2	3	4	5
---	---	---	---	---

1 格代表一个字节

所以明显计算 1 , 2,3,4,5,这种数组用 char 比用 int 节省空间。但是在 linux 用 char 不一定节省空间，因为一个 char 就霸占一个内存地址，一个内存地址是 32 位，所以霸占了 8 位，后面 24 位只有空着，然后在申请一个 char 就从新的内存地址开始分配。所以我们分配数组最好是声明的时候就直接连续定义长度，或者用 malloc 来连续分配动态内存。

```

int dim(int arrayzz[])
{
    long arraylong = sizeof(arrayzz);
    return arraylong;
}

int main()
{
    int arr[]={1,2,3,4,5};
    printf("array = %d\n",dim(arr));//我不用宏我用函数来计算数组长度呢
    return 0;
}

root@ubuntu:/home/xiang/C_code# gcc -o def def.c
def.c: In function 'dim':
def.c:12:25: warning: 'sizeof' on array function parameter 'arrayzz' will return size of 'int *' [-Wsizeof-array-argument]
  long arraylong = sizeof(arrayzz);
                           ^
def.c:10:13: note: declared here
  int dim(int arrayzz[])
                           ^

```

这样写有问题

为什么会有警告，在子函数里面计算数组长度有问题，因为 arrayzz[] 形参自动退化成了指针，所以你 sizeof(arrayzz) 计算出来的数组长度就是一个指针的 4 字节长度，和我们前面使用宏计算数组长度的结果不一样，所以这里用宏来计算数组长度比用子函数来计算数组长度优势大多了。

#undef 用法

```
int f1(int a,int b)
{
    #define MIX(a,b) ((a)<(b)?a:b) //求最小值
    return MIX(a,b);
}

int f2(int a,int b)
{
    return MIX(a,b);
}

int main()
{
    printf("f1 = %d\n",f1(2,5));
    printf("f2 = %d\n",f1(3,5));
    return 0;
}
```

我们发现在子函数 f1 定义的宏，子函数 f2 可以使用

这是因为宏在代码里面定义之后，文件里面任何一段代码都可以调用它
这就是宏的特性

```
root@ubuntu:/home/xiang/C_code# gcc -o def def.c
root@ubuntu:/home/xiang/C_code# ./def
f1 = 2
f2 = 3
```

输出结果正常

如果我们只想让宏在某个函数里面执行，其它函数不能调用这个宏怎么办呢？

```
int f1(int a,int b)
{
    #define MIX(a,b) ((a)<(b)?a:b) //求最小值
    return MIX(a,b);
    #undef MIX
}

int f2(int a,int b)
{
    return MIX(a,b);
}

int main()
{
    printf("f1 = %d\n",f1(2,5));
    printf("f2 = %d\n",f1(3,5));
    return 0;
}
```

在使用这个宏的函数结束位置加#define就可以了，这个宏就只限制于本函数

```
root@ubuntu:/home/xiang/C_code# gcc -o def def.c
def.c: In function 'f2':
def.c:14:9: warning: implicit declaration of function 'MIX' [-Wimplicit-function-declaration]
    return MIX(a,b);
    ^
/tmp/ccn7vzeo.o: In function `f2':
def.c:(.text+0x34): undefined reference to `MIX'
collect2: error: ld returned 1 exit status
root@ubuntu:/home/xiang/C_code#
```

你看 f2 想去调用 f1 的宏就报错

#if #else 和 if else 的区别

```
#define C 1
int main()
{
    if(C == 1)
        printf("执行第1行输出\n");
    else
        printf("执行第2行输出\n");
    return 0;
}

root@ubuntu:/home/xiang/C_code# gcc -o ifel
root@ubuntu:/home/xiang/C_code# ./ifelse
执行第1行输出
```

这是我们常用的
if else

输出正常

```
3 # 6 "ifelse.c"
4 int main()
5 {
6     if(1 == 1)
7         printf("执行第1行输出\n");
8     else
9         printf("执行第2行输出\n");
10    return 0;
11 }
```

在 i 文件里面编
译器把 if 和 else
的输出都编译进
去了

下面这是#if #else

```
5 #define C 1
6 int main()
7 {
8     #if(C == 1)
9         printf("执行第1行输出\n");
10    #else
11        printf("执行第2行输出\n");
12    #endif
13    return 0;
14 }
15

root@ubuntu:/home/xiang/C_code# gcc -o ifel
root@ubuntu:/home/xiang/C_code# ./ifelse
执行第1行输出
```

输出结果一样

但是你看看 gcc -E 的中间文件 ,

```
# 6 "ifelse.c"
int main()
{
    printf("执行第1行输出\n");

    return 0;
}
```

只编译进#if 成
立的执行段代
码。不成立的就
不编译进二级制

```
int main()
{
    #if(C == 1)
        printf("执行第1行输出\n");
    #else
        printf("执行第2行输出\n");
    #endif
    return 0;
}
```

我如果不定义宏，那么就输出
#else 代码段

```
root@ubuntu:/home/xiang/C_code# ./ifelse
执行第2行输出
```

还有一种不用给 C 赋值的条件判断语句，#ifdef #else #endif

```
4 #define DEBUG
5 int main()
6 {
7     #ifdef DEBUG
8         printf("执行第1行输出\n");
9     #else
10        printf("执行第2行输出\n");
11    #endif
12    return 0;
13 }
14
```

用 ifdef 直接在
宏这里定义名字
就是了

```
root@ubuntu:/home/xiang/C_code# ./ifelse
执行第1行输出
```

```
3
4 int main()
5 {
6     #ifdef DEBUG
7         printf("执行第1行输出\n");
8     #else
9         printf("执行第2行输出\n");
10    #endif
11    return 0;
12 }
13
```

这是没有在宏定
义名字的

```
root@ubuntu:/home/xiang/C_code# ./ifelse
执行第2行输出
```

其实这种条件编译语句适合用来做代码调试，比如我写了很多 printf 和调试程序，正式发布程序的时候不需要这些，但是调试的时候需要，但是太多的调试代码一个一个去删除很麻烦，如果删除了那天又想拿回来调试又要重新写，所以用条件编译时最好的解决方法。

```
# 4 "ifelse.c"
int main()
{
    printf("执行第2行输出\n");
    return 0;
}
```

#ifdef 和#if一样的都是选择性编译。

编译器编译 C 代码的时候打印出的 warning , error , 是怎么实现的 ?

#error , #warning , #pragma 这些都是预处理指令 , 就是靠这些实现的

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

#define version1
    #if defined(version1)
        #pragma message("compile version111....")
        #define VERSION "VERSION11111"
    #elif defined(version2)
        #pragma message("compile version222....")
        #define VERSION "VERSION2222"
    #elif defined(version3)
        #pragma message("compile version333....")
        #define VERSION "VERSION3333"
    #else
        #error Compile not failed!
    #endif

int main()
{
    printf("%s\n",VERSION);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# gcc -o progam progam.c
progam.c:6:11: note: #pragma message: compile version111...
    #pragma message("compile version111....")
          ^
root@ubuntu:/home/xiang/C_code# ./progam
VERSION11111
```

比如我把宏#define version1 改成 version2

```
#define version2
    #if defined(version1)           我把宏改成了版本 2
        #pragma message("compile version111....")
        #define VERSION "VERSION11111"
    #elif defined(version2)
        #pragma message("compile version222....")
        #define VERSION "VERSION2222"
    #elif defined(version3)
        #pragma message("compile version333....")
        #define VERSION "VERSION3333"
    #else
        #error Compile not failed!
    #endif

int main()
{
    printf("%s\n",VERSION);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# gcc -o progam progam.c
progam.c:9:11: note: #pragma message: compile version222....
    #pragma message("compile version222....")
    ^
root@ubuntu:/home/xiang/C_code# ./progam
VERSION222
```

你看编译的时候执行 elif 里面的输出，主程序也是执行 VERSION222 输出
如果我取消宏呢？

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

#if defined(version1)
    #pragma message("compile version111....")
    #define VERSION "VERSION11111"
#elif defined(version2)
    #pragma message("compile version222....")
    #define VERSION "VERSION22222"
#elif defined(version3)
    #pragma message("compile version333....")
    #define VERSION "VERSION33333"
#else
    #error Compile not failed!
#endif

int main()
{
    printf("%s\n",VERSION);
    return 0;
}
```

我取消了#define version1 或者#define version2 这一段

输出直接打印报错
error 里面的字符串

```
root@ubuntu:/home/xiang/C_code# gcc -o progam progam.c
progam.c:14:4: error: #error Compile not failed!
    #error Compile not failed!
    ^
progam.c: In function 'main':
progam.c:19:16: error: 'VERSION' undeclared (first use in this function)
    printf("%s\n",VERSION);
    ^
progam.c:19:16: note: each undeclared identifier is reported only once for each function it appears in
```

所以编译程序输出信息用#pragma , #error , #warning , 执行程序输出信息用 printf

#progma 这个东西根据编译器的不同，是有很大区别的，所以在不同编译器使用#progma 的时候，最好先实验下#progma 的功能。

强制类型转换，转换的是数据还是地址？

```
void main()
{
    int *p = (int *)0;
    float *fp = (float *)0;
    int p2 = (int)0;
    float fp2 = (float)0;
    short s = (short)0;
}
```

这种方式就是我们把 0 当做一个地址赋值给 p , p 指向内存 0 这个地址

内存	
地址	数据
0x01	
0x00	

类似与下面这样

```
int *p;
p=(int *)0x00; //把0x00地址赋值给p
p=0x00; //把0x00地址赋值给p
```

	地址	数据
	0x01	
p=	0x00	

```
*p = 5; //把数据5写入p指向的内存地址上面
```

	地址	数据
	0x01	
*p=	0x00	5

```
int p2 = (int)0;
float fp2 = (float)0;
short s = (short)0;
```

这种括号里面不带*的写法又是什么意思呢

我们知道内存的最小单元是 char 字节，所以内存一个地址只能存放一个字节 8 位的数，而不是 16 位，32 位

内存(32 位系统)

数据				
地址	0x00000007	0x00000006	0x00000005	0x00000004

数据				
地址	0x00000003	0x00000002	0x00000001	0x00000000

指针使用注意事项

野指针处理方法

```
int main()
{
    int *p;
    *p = 4;

    return 0;
}
```

指针 p 被赋值前，没有给指针赋地址

```
root@ubuntu:/home/xiang/C_code# gcc -o pointt pointt.c
root@ubuntu:/home/xiang/C_code# ./pointt
Segmentation fault (core dumped)
root@ubuntu:/home/xiang/C_code#
```

因为指针的地址不确定，所以这个 4 也不知道放在内存哪个地址上，就会出现段错误

这种就是野指针的段错误。

```
int main()
{
    int *p = NULL;

    /*.....这里就是给指针赋值地址,
     * 分配指针指向的内存什么的.....*/

    if(p != NULL)//然后在给指针赋值之前，先用NULL判断指针是不是已经获得了一个明确的地址
    {
        *p = 4;//然后再给这明确的地址赋值
    }

    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# gcc -o pointt pointt.c
root@ubuntu:/home/xiang/C_code# ./pointt
root@ubuntu:/home/xiang/C_code#
```

用 NULL 判断指针是否有地址，这样可以防止段错误

```
int main()
{
    int *p = NULL;
    int a;

    p=&a;//正确的使用方法就是给指针赋值前，先给指针一个地址

    if(p != NULL)//然后在给指针赋值之前，先用NULL判断指针是不是已经获得了一个明确的地址
    {
        *p = 4;//然后再给这明确的地址赋值
    }
    printf("*p = %d\n",*p);
    p=NULL;//指针使用完了，如果以后不使用这个指针，就让p和&a这个地址解绑，这样其它指针就可以再次指向a地址
    //如果不给p=NULL，这个p会成为野指针
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
*p = 4
root@ubuntu:/home/xiang/C_code#
```

指针用完后，给指针地址改成 NULL，这样可以防止野指针。

你如果水平可以也可以在给*p 赋值之前不用判断 p 是不是为 NULL，自己把控指针使用方法。

sizeof 使用注意

```
int main()
{
    char str[] = "hello";
    int addrsiz = sizeof(str); //因为sizeof的返回值是size_t无类型，我们只有用int来接收返回值
    int charsize = sizeof(str[0]);
    int string = strlen(str);
    printf("sizeof(str) = %d\n", addrsiz); //返回str数组里面的元素占用多少个地址
    printf("sizeof(str[0]) = %d\n", charsize); //计算str里面一个元素需要几个地址
    printf("strlen(str) = %d\n", string); //计算str里面有多少个字符

    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
sizeof(str) = 6
sizeof(str[0]) = 1
strlen(str) = 5
```

为什么是 6 ,因为计算的是 str 里面所有字符串加起来占多少个地址 , 我们知道 1 个字符占用 1 个字节 这个字符串除了 hello 之外还有个\0 结束符 , 所以加起来是 6

这是计算数组里面我们看得见的字符个数 , 所以 hello 是 5 个字符 , 而不是前面使用 sizeof 计算数组占用地址大小

这是计算 1 个字符的大小 , 1 个字符 8 位 1 个字节

```
4 int main()
5 {
6     char str[] = "hello";
7     char *p = str;
8
9     printf("sizeof(p) = %d\n", sizeof(p)); //计算指针自己占用内存地址大小 , 32位系统指针本身占用内存4个字节 , 8位系统不清楚...
10    printf("sizeof(*p) = %d\n", sizeof(*p)); //sizeof(*p)计算的是p指向的str自己占用地址大小 , str是char的 , 所以是1个字节
11    printf("strlen(p) = %d\n", strlen(p)); //这是取出p指向的str地址 , 然后计算str地址上字符串长度 , 类似strlen(str)
12
13    return 0;
14 }
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
sizeof(p) = 8
sizeof(*p) = 1
strlen(p) = 5
root@ubuntu:/home/xiana/C_code# ■
```

这里指针怎么占用 8 个字节 , 因为我们这里用的是 ubuntu 64 位操作系统

```
4 int main()
5 {
6     char c[100];
7     short s[100];
8     int i[100];
9
10    printf("sizeof(c[100]) = %d\n", sizeof(c)); //这种写法就是sizeof直接计算数组开辟多少个字节的空间
11    printf("sizeof(s[100]) = %d\n", sizeof(s));
12    printf("sizeof(i[100]) = %d\n", sizeof(i));
13
14    return 0;
15 }
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
sizeof(c[100]) = 100
sizeof(s[100]) = 200
sizeof(i[100]) = 400
```

struct 结构体高级用法

```
struct A
{
    char a;
    int b;
};

void fun(struct A a1)
{
    printf("sizeof(a1) = %d\n", sizeof(a1));
    printf("&a1 = %p\n", &a1);
}

int main()
{
    struct A a =
    {
        .a = 4,
        .b = 555,
    };
    printf("main sizeof(a) = %d\n", sizeof(a));
    printf("main &a = %p\n", &a);
    fun(a);
    return 0;
}
```

函数用结构体做形参和函数用 int
形参是一样的，都是拷贝数据

在 gcc 编译器中结构体可以这样
初始化，这样初始化的好处是如果
我不想初始化 a 我可以直接
Struct A a =
{
 .b = 555,
};

```
root@ubuntu:/home/xiang/C_code# ./pointt
main sizeof(a) = 8
main &a = 0x7fff03f84280
sizeof(a1) = 8
&a1 = 0x7fff03f84260
root@ubuntu:/home/xiang/C_code#
```

主函数和子函数地址都
不一样，所以主函数只
是拷贝数据给子函数，
子函数运行结束，子函
数的数据都会被释放

像这种结构体传参数给子函数的方法效率很低，因为结构体一般情况下会定义很多变量和数据，所以在传递形参过程中相当于拷贝数据，你说慢不慢。解决这种效率低下的方法就是用指针来接收结构体地址。

```
struct A
{
    char a;
    int b;
    int c[10];
};

void fun(struct A *a1)
{
    printf("sizeof(a1) = %d\n", sizeof(a1)); //计算形参自己占用内存大小
    printf("sizeof(*a1) = %d\n", sizeof(*a1)); //计算传入形参这个变量的大小，也就是a的大小，a的大小就是整个结构体大小
    printf("&a1 = 0x%llx\n", &a1); //形参自己的地址
    printf("a1 = 0x%llx\n", a1); //传入形参a的地址
    printf("a1 ->b %d\n", a1->b); //因为形参是结构体指针所以用->来获取结构体成员数据
}

int main()
{
    struct A a =
    {
        .a = 4,
        .b = 555,
    };
    printf("sizeof(a) = %d\n", sizeof(a)); //计算a变量大小，也就是整个结构体大小
    printf("&a = 0x%llx\n", &a); //获取结构体变量a的首地址
    printf("a.b %d\n", a.b); //因为结构体a是变量所以用.a来获取结构体成员数据
    fun(&a); //因为子函数形参是结构体指针，所以这里要传入结构体地址
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
sizeof(a) = 48
&a = 0xb76ab2d0
a.b 555
sizeof(a1) = 8
sizeof(*a1) = 48
&a1 = 0xb76ab2b8
a1 = 0xb76ab2d0
a1 ->b 555
```

如何取结构体指针成员里面的地址

```
1 #include<stdio.h>
2 #include<malloc.h>
3 #include<string.h>
4
5 struct ssuid
6 {
7     int age;
8     char name[20];
9 };
0
1 void input(struct ssuid *ssuidp)
2 {
3     printf("获取指针指向的age地址 : %x\n",&ssuidp->age); //箭头前面取地址&是获取指针指向的那个结构体变量的地址
4     printf("获取指针指向的age数据 : %d\n",ssuidp->age); //箭头前面不取地址，就是直接获取指针指向的那个结构体数据
5     printf("获取指针指向的name数据 : %x\n",ssuidp->name); //name虽然箭头前面没加&，但是因为它是数组所以这里也是取地址
6 }
7
8 int main()
9 {
0     struct ssuid std;
1     std.age = 20;
2     std.name = "abcdefg";
3     input(&std);
4     return 0;
5 }
```

这里为什么给字符数组赋值会报错？

C 语言只有在定义字符数组的时候才能用“=”来初始化变量，其它情况下是不能直接用“=”来为字符数组赋值的
char a[10] = "123"; /*正确，在定义的时候初始化*/

char a[10];
a = "123"; /*错误，不能用“=”直接为字符数组赋值*/

strcpy(a, "123"); /*正确，使用 strcpy 函数复制字符串*

```
struct ssuid
{
    int age;
    char name[20];
};

void input(struct ssuid *ssuidp)
{
    printf("获取指针指向的age地址 : %x\n",&ssuidp->age); //箭头前面取地址&是获取指针指向的那个结构体变量的地址
    printf("获取指针指向的age数据 : %d\n",ssuidp->age); //箭头前面不取地址，就是直接获取指针指向的那个结构体数据
    printf("获取指针指向的name数据 : %x\n",ssuidp->name); //name虽然箭头前面没加&，但是因为它是数组所以这里也是取地址
}

int main()
{
    struct ssuid std;
    std.age = 20;
    strcpy(std.name, "abcdefg");
    input(&std);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./struc
获取指针指向的age地址 : cfa675c0
获取指针指向的age数据 : 20
获取指针指向的name数据 : cfa675c4
```

多级结构体使用

创建结构体 1

创建结构体 2

结构体 2

结构体 1

(结构体 1 包含结构体 2)

```

4
5 typedef struct student
6 {
7     int ID;
8     char name[20];
9 }stu;
10
11
12 typedef struct school
13 {
14     stu st;
15     char schoolname[20];
16 }school_T;
17
18 int main()
19 {
20
21     school_T mschool;
22
23     mschool.st.ID = 125;
24     strcpy(mschool.st.name,"student xzz");
25     strcpy(mschool.schoolname,"hafuschool");
26
27     printf("student name = %s\n",mschool.st.name);
28     printf("student ID = %d\n",mschool.st.ID);
29     printf("school name = %s\n",mschool.schoolname);
30
31     return 0;
32 }
33

```

用 2 个点访问
school 结构体
里面的 stu 结
构体

用 1 个点访问
school 结构体

```

root@ubuntu:/home/xiang/C_code# ./struc
student name = student xzz
student ID = 125
school name = hafuschool

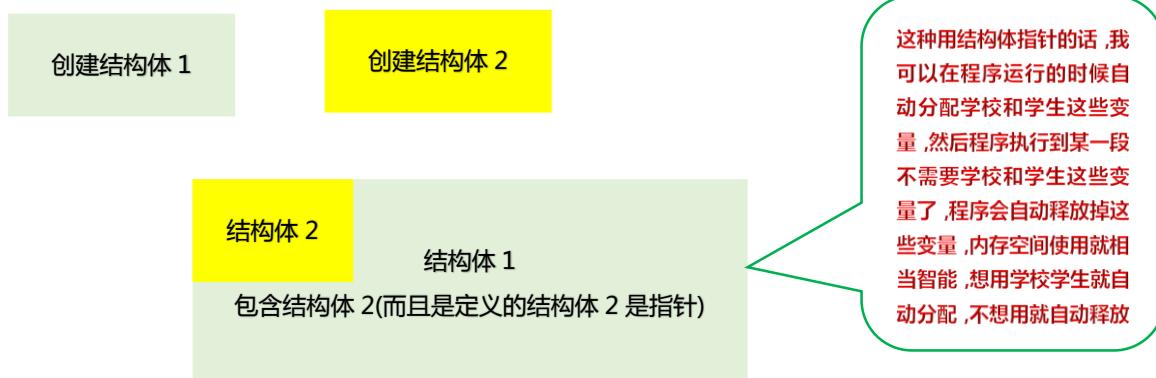
```

这种多级结构体都是用来管理变量和内存的，如果变量少用这种方法没有意义反而麻烦。那么用多级结构体有什么用？比如我要定义一个学校，学校这个变量就包含了树木，房子，老师，操场，学生，这些变量。在学生这个变量里面还包含学生，年龄，成绩，健康这些变量，所以我们就用了一级结构体来定义学校的成员，二级结构体来定义学生自己本身的内容。当然你也可以不这么做，你也可以直接写 n 多个变量来做。只是看起不是很直观。

其实这种直接访问多级结构体还是没有看出它有什么优势，但是我们用多级结构体指针就能看出它的优势了

结构体多级指针使用

结构体多级指针一般使用在多个结构体指针里面



```

4
5 typedef struct student
6 {
7     int ID;
8     char name[20];
9 }stu;
10
11
12 typedef struct school
13 {
14     stu* st;
15     char schoolname[20];
16 }school_T;
17
18 int main()
19 {
20     school_T *mschool = (school_T *)malloc(sizeof(school_T));
21     if(mschool == NULL)
22     {
23         printf("school malloc failed\n");
24     }
25     mschool->st = (stu *)malloc(sizeof(stu));
26     if(mschool->st == NULL)
27     {
28
29         printf("student malloc failed\n");
30     }
31
32     mschool->st->ID = 125;
33     strcpy(mschool->st->name, "student xzz");
34     strcpy(mschool->schoolname, "hafuschool");
35
36     printf("student name = %s\n", mschool->st->name);
37     printf("student ID = %d\n", mschool->st->ID);
38     printf("school name = %s\n", mschool->schoolname);
39
40     free(mschool->st);
41     free(mschool);
42     return 0;
43 }

```

我在 school 结构体里面
创建了一个结构体指针

我分配第一级的 school
结构体变量就是用指针，
所以访问 school 的时候
用 1 个箭头

我分配第二级，也就是
school 里面的结构体变
量时也是用的指针，所以
我要用 2 个箭头

访问结构体里
面的结构体

访问本身结构体

释放学生和学校变量的时候一定要先释放结构体
里面包含的结构体

然后再释放本身的结构体

```

root@ubuntu:/home/xiang/C_code# ./struc
student name = student xzz
student ID = 125
school name = hafuschool

```

结构体指针虽然比上面直接访问结构体多了几行代码，但是这样就可以做到自动申请自动释放。

如果你使用 3 级结构体，就多写几个箭头， ->->->

如果你使用 4 级结构体，就多写几个箭头， ->->->->

还有结构体指针访问比直接访问结构体运行效率高。

数组指针，用来操作二维数组

定义 `int (*p)[n];`

()优先级高，首先说明 p 是一个指针，指向一个整型的一维数组，这个一维数组的长度是 n

```
int main()
{
    int a[3][4] = { {1,2,3,4},
                    {10,20,30,40},
                    {100,200,300,400},
                };
    int (*p2)[4];//定义一个接受二维数组(*p2)[4] 4列的指针，该指针不管二维数组有多少行，只关心二维数组列必须和指针要求的列对应
    p2 = a; //将二维数组a的首行首列地址赋值给p2，也就是a[0]或&a[0][0]
    printf("*p2[0] = %d\n", *p2[0]);//*p2[0]取地址0行的第0个元素
    printf("*p2[1]+1 = %d\n", *p2[1]+1);//*p2[1]取地址1行的第0个元素

    printf("*(p2[1]+1) = %d\n", *(p2[1]+1));//*(p2[1]+1) 取数组地址1行的第一个元素

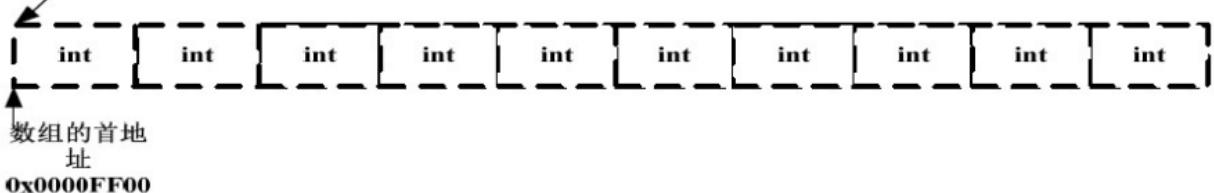
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
*p2[0] = 1
*p2[1]+1 = 11
*(p2[1]+1) = 20
```

`int (*p2)[10];`

P2 = 0x0000FF00

p2为指针变量名



```
void p2array(int (*array)[4])
{
    printf("*array[0] = %d\n", *array[0]);
    printf("*(array[1]+1) = %d\n", *(array[1]+1));
    array[1][1] = 2000;
}
int main()
{
    int a[3][4] = { {1,2,3,4},
                    {10,20,30,40},
                    {100,200,300,400},
                };
    p2array(a);

    printf(" main a[1][1] = %d\n", a[1][1]);

    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
*array[0] = 1
*(array[1]+1) = 20
main a[1][1] = 2000
```

这就是数组指针用在子函数接口封装，修改

原本二维数组值的好处。

指针数组，数组里面全部放的是指针变量

定义 int *p[n];

[]优先级高,先与 p 结合成为一个数组,再由 int*说明这是一个整型指针数组,它有 n 个指针类型的数组元素。

```
int main()
{
    int a=10,b=20,c=30,d=40;
    int *array[4];
    array[0]=&a;
    array[1]=&b;
    array[2]=&c;
    array[3]=&d;

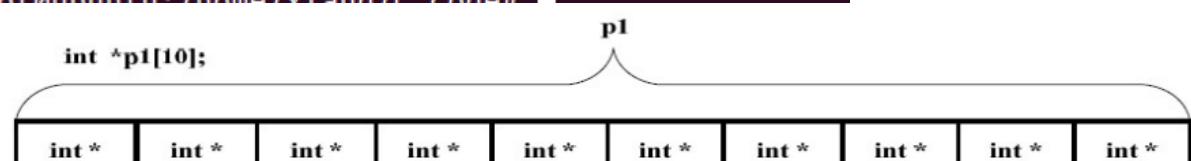
    printf("-----打印数组元素自己的地址-----\n");
    printf("&array[0] = %x\n",&array[0]);
    printf("&array[1] = %x\n",&array[1]);
    printf("&array[2] = %x\n",&array[2]);
    printf("&array[3] = %x\n",&array[3]);
    printf("数组元素存放的地址是不是和abcd一样\n");
    printf("*array[0] = %x\n",*array[0]);//因为数组元素是指针,所以每个元素存放的是a, b, c, d的地址
    printf("*array[1] = %x\n",*array[1]);
    printf("*array[2] = %x\n",*array[2]);
    printf("*array[3] = %x\n",*array[3]);
    printf("-----对比分隔符-----\n");
    printf("&a = %x\n",&a);
    printf("&b = %x\n",&b);
    printf("&c = %x\n",&c);
    printf("&d = %x\n",&d);
    printf("abcd值输出\n");
    printf("*array[0] = %d\n",*array[0]);//array[0]是取数组元素里面存放的地址,*array[0]是在数组元素里面存放的地址上,再去取地址上的值
    printf("*array[1] = %d\n",*array[1]);
    printf("*array[2] = %d\n",*array[2]);
    printf("*array[3] = %d\n",*array[3]);

    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
-----打印数组元素自己的地址-----
&array[0] = 8906ae60
&array[1] = 8906ae68
&array[2] = 8906ae70
&array[3] = 8906ae78
数组元素存放的地址是不是和abcd一样
array[0] = 8906ae50
array[1] = 8906ae54
array[2] = 8906ae58
array[3] = 8906ae5c
-----对比分隔符-----
&a = 8906ae50
&b = 8906ae54
&c = 8906ae58
&d = 8906ae5c
abcd值输出
*array[0] = 10
*array[1] = 20
*array[2] = 30
*array[3] = 40
root@ubuntu:/home/xiang/C_code#
```

指针数组里面每一个元素不是用来放值的,而是用来放地址

因为 array[0] = &a,
所以指针数组不用*号
直接用下标取出来的是
地址



`p1`为数组名

```
int main()
{
    char *word[4] =
    {
        "do",
        "for",
        "structstatic",
        "makemenuconfig",
    };
    char *table[4];

    table[0] = word[0];
    table[1] = word[1];
    table[2] = word[2];
    table[3] = word[3];
    printf("table[0] = %s\n",table[0]);
    printf("table[1] = %s\n",table[1]);
    printf("table[2] = %s\n",table[2]);
    printf("table[3] = %s\n",table[3]);

    return 0;
}
```

如果指针数组不做函数形参，那么相互之间传递地址要用下标[]位置

```
root@ubuntu:/home/xiang/C_code# ./pointt
table[0] = do
table[1] = for
table[2] = structstatic
table[3] = makemenuconfig
```

```
void keywold(char *table[])
{
    printf("table[0] = %s\n",table[0]);
    printf("table[1] = %s\n",table[1]);
    printf("table[2] = %s\n",table[2]);
    printf("table[3] = %s\n",table[3]);
}
```

如果指针数组做函数形参，那么相互之间传递地址就不用指定下标位置

```
int main()
{
    char *word[4] =
    {
        "do",
        "for",
        "structstatic",
        "makemenuconfig",
    };

    keywold(word);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./pointt
table[0] = do
table[1] = for
table[2] = structstatic
table[3] = makemenuconfig
```

指针数组比较适合用来做字符串传递，比如 main 函数的 argv

函数指针

```
3 void func1(void)
4 {
5     printf("func1.....\n");
6 }
7
8 int main()
9 {
10     void (*pfunc)(void); //函数指针pfunc就是指针变量，func1函数的形参是什么类型，那么函数指针也必须是什么类型，func1的返回值是什么类型，那么函数指针的返回值也必须是什么类型
11     pfunc = func1;
12     pfunc();
13     return 0;
14 }
```

root@ubuntu:/home/xiang/C_code# ./mainpoint
func1.....
root@ubuntu:/home/xiang/C_code#

```
3 char *strcpy1(char *dest, const char *src)
4 {
5     for(int i=0;i<strlen(src);i++)
6     {
7         dest[i] = src[i];
8     }
9 }
10
11 int main()
12 {
13     char* (*pfunc)(char *dest, const char *src); //函数指针pfunc就是指针变量，func1函数的形参是什么类型，那么函数指针也必须是什么类型，func1的返回值是什么类型，那么函数指针的返回值也必须是什么类型
14     pfunc = strcpy1; //pfunc = &strcpy;如果函数指针用&来获得函数地址真实和不用是一样的，函数指针不像数组a[0]和&a[0]，虽然得到的首地址一样，但是意义不一样
15     char a[10];
16     pfunc(a,"abcdef");
17     printf("%s\n",a);
18     return 0;
19 }
```

root@ubuntu:/home/xiang/C_code# ./mainpoint
abcdef

这就是函数指针多参数用法。

Func1	0x0c
Func1	0x0b
Func1	0x0a
Func1	0x09
	0x08
	0x07
pfunc	0x06
	0x05
数据	地址

这是 func1 子函数占用的内存空间大小，每个子函数在内存都是连续的

函数指针变量存放在 0x06 地址这个位置

这个 pfunc 变量存放的是 func1 函数的首地址，所以执行 pfunc 就是执行 func1

typedef 修饰函数指针的好处

```
int main()
{
    char* (*p1)(char *dest, const char *src);
    char* (*p2)(char *dest, const char *src);
    char* (*p3)(char *dest, const char *src);
    char* (*p4)(char *dest, const char *src);
    return 0;
}
```

如果我想多定义几个函数指针，指向 func1 函数，我就要写很多个，因为 func1 的形参太多，所以我要每个函数指针都要重复写很多形参，导致我太累。

```
char* func1(char *dest, const char *src)
{
    printf("func1.....\n");
}

typedef char* (*typefunc)(char *dest, const char *src);

int main()
{
    typefunc p1;
    typefunc p2;
    p1 = func1;
    p2 = func1;

    p1(NULL,NULL);
    p2(NULL,NULL);
    return 0;
}
```

我用 `typedef` 定义一个函数
指针的类型

然后我不管想创建多少个函
数指针，都不用每个都去写
形式参数

以后看到这种定义，先搞清楚是定义的结构体，
还是函数指针

```
root@ubuntu:/home/xiang/C_code# ./mainpoint
func1.....
func1.....
```

函数指针模拟函数重载使用

```
#include<stdio.h>
int add(int a,int b)
{
    printf(".....a + b.....\n");
    return a+b;
}
int sub(int a,int b)
{
    printf(".....a - b.....\n");
    return a-b;
}
int multiply(int a,int b)
{
    printf(".....a * b.....\n");
    return a*b;
}
int divide(int a,int b)
{
    printf(".....a / b.....\n");
    return a/b;
}
typedef int (*pfunc)(int a,int b);
int main()
{
    pfunc p1 = NULL;
    char c = 0;
    int a = 0,b = 0;
    printf("输入数字和符号\n");
    scanf("%d %d %c",&a,&b,&c);
    switch(c)
    {
        case '+':p1=add;break;
        case '-':p1=sub;break;
        case '*':p1=multiply;break;
        case '/':p1=divide;break;
        default:
            p1 = NULL;break;
    }
    int num = p1(a,b); //我们用函数指针做了一个类似面向对象的函数重载功能,
                        //就是我的函数可以重复使用，虽然函数名一样，但是每次重复使用功能可以不一样
                        //像C++/JAVA是根据函数参数的个数或者类型来决定函数的功能
                        //我们这里是用函数指针和switch来模拟一个类似的功能，但是不是完整的函数重载
    printf("num = %d\n",num);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./mainpoint
输入数字和符号
```

```
10 5 +
.....a + b.....
```

```
num = 15
root@ubuntu:/home/xiang/C_code# ./mainpoint
输入数字和符号
```

```
10 5 -
.....a - b.....
```

```
num = 5
root@ubuntu:/home/xiang/C_code# ./mainpoint
输入数字和符号
```

```
10 5 *
.....a * b.....
```

```
num = 50
root@ubuntu:/home/xiang/C_code# ./mainpoint
输入数字和符号
```

```
10 5 /
.....a / b.....
```

```
num = 2
```

其实我觉得上面用函数指针

做子函数调用，和直接调用子函数一样，我为什么还要这么麻烦用函数指针去调用子函数呢？

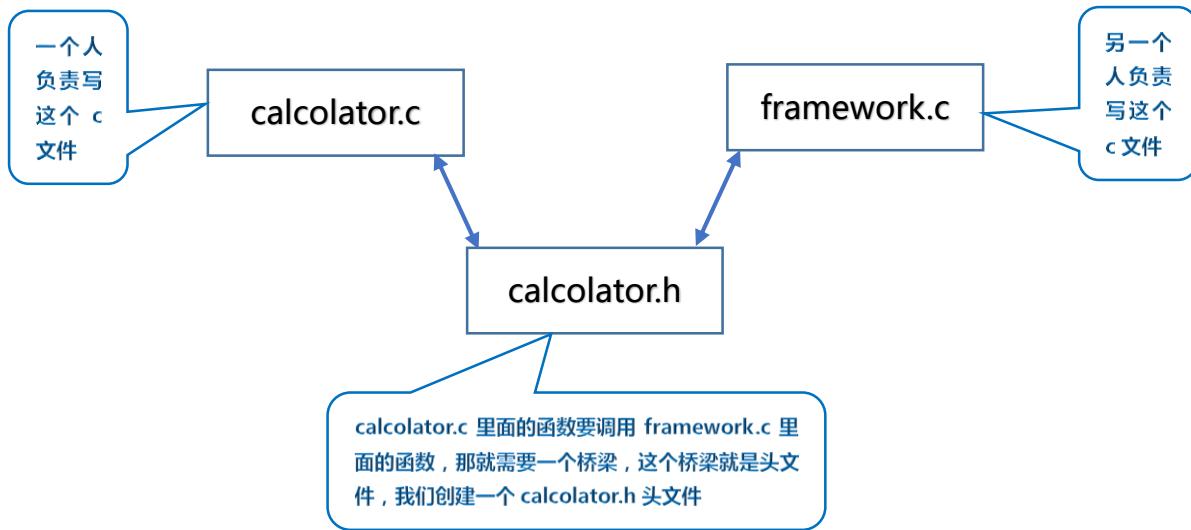
如果是用函数指针去做调用子函数确实没有什么意义，反而麻烦，但是我下面准备做分层编程，那么函数指针就有用了。

我们把上面计算器这个程序，进行分层，让多人来完成这一个计算器功能。

```
root@ubuntu:/home/xiang/C_code/plat# ls  
calcolator.c  framework.c
```

我创建了两个文件，framework.c 是架构文件。

Calcolator.c 是具体执行的文件，一个人负责一个文件，所以就需要两个人来做这个事情。



```
root@ubuntu:/home/xiang/C_code/plat# ls  
calcolator.c  calcolator.h  framework.c
```

我们创建了一个 h 文件当桥梁

我们先创建桥梁 **calcolator.h**

```
#ifndef __CALCOLATOR_H__  
#define __CALCOLATOR_H__  
  
//连接框架和具体实现函数的结构体，要根据实际功能来定义，  
//我这里是要做计算器，那么就要有两个值的输入，和计算后的输出  
  
struct calcolator_framework  
{  
    int a;  
    int b;  
    int (*pfunc)(int,int); //这里就要用函数指针  
};  
  
int calculator(struct calcolator_framework* p); //这就是framework层的人提供的函数，  
//至于函数怎么实现，写calcolator的人不用关心，只要写的结构体和函数能塞进framework给的calculator(struct calcolator_framework* p)函数就行  
#endif
```

Framework 层的人负责头文件的内容编写

```
1 #include<stdio.h>  
2 #include"calcolator.h"  
3  
4 int calculator(struct calcolator_framework* p) //这就是h文件里面的结构体，用来连接两个c文件里面的内容  
5 {  
6     int ret = p->pfunc(p->a,p->b);  
7     return ret;  
8 }  
9  
//其实framework层提供的函数只是指挥其他c文件要做什么事情，但是framework自己不去实现，只提供函数接口
```

在 **framework.c** 里面 Framework 层的人还要实现给下层人的函数接口

```

#include<stdio.h>
#include"calculator.h"

int add(int a,int b)
{
    printf(".....a + b.....\n");
    return a+b;
}

int sub(int a,int b)
{
    printf(".....a - b.....\n");
    return a-b;
}
int multiply(int a,int b)
{
    printf(".....a * b.....\n");
    return a*b;
}
int divide(int a,int b)
{
    printf(".....a / b.....\n");
    return a/b;
}

int main(void)
{
    struct calcolator_framework mycal;//定义一个结构体，这个结构体要符合framework层函数的要求
    mycal.a = 12;//填充要做运算的数
    mycal.b = 4;//填充要做运算的数
    mycal.pfunc = add;//加法

    int ret = calculator(&mycal);//调用framework实现的calculator函数，将mycal塞进去
    printf("结果 = %d\n",ret);
    return 0;
}

```

calculator.c 由下层的人负责实现具体功能代码，然后填充 framework 层人设计的结构体和函数。

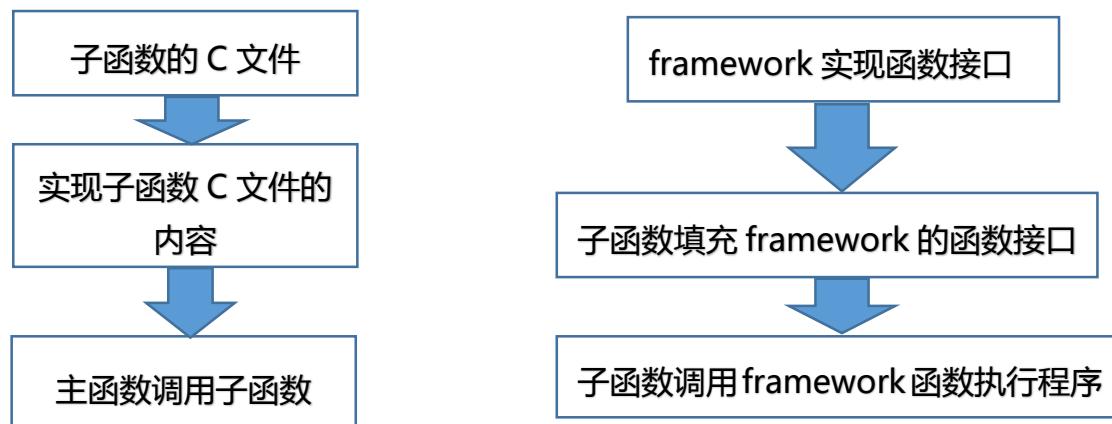
```

root@ubuntu:/home/xiang/C_code/plat# gcc -o plat calcolator.c framework.c
root@ubuntu:/home/xiang/C_code/plat# ls
calcolator.c  calcolator.h  framework.c  plat
root@ubuntu:/home/xiang/C_code/plat# ./plat
.....a + b.....结果 = 16
root@ubuntu:/home/xiang/C_code/plat#

```

将两个 C 文件编译起来，就可以使用了，我们计算的是 $12 + 4 = 16$

这就是函数指针的用处，我们以前都是先写调用函数，在调用函数里面实现代码。然后主函数调用子函数



这是以前的代码框架

这是分层后的代码框架

这样看分层貌似没有什么特别的用处，但是当你几十个人在完成一个项目的时候你就会发现很有用，只是我们现在这个例子太简单，所以反而感觉是杀鸡用牛刀，现在记住有这个思想就是了。

typedef 修饰结构体的好处

```
#include<stdio.h>

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

int main(void)
{
    struct student std;
    std.age = 10;
    printf("age = %d\n",std.age);
    return 0;
}

root@ubuntu:/home/xiang/C_code# gcc -o type_struct type_struct.c
root@ubuntu:/home/xiang/C_code# ./type_struct
age = 10

3 struct student
4 {
5     char name[20];
6     int age;
7 }stu;
8
9 int main(void)
0 {
1     stu std;
2     std.age = 10;
3     printf("age = %d\n",std.age);
4     return 0;
5 }

root@ubuntu:/home/xiang/C_code# gcc -o type_struct type_struct.c
type_struct.c: In function 'main':
type_struct.c:11:6: error: expected ';' before 'std'
  stu std;
      ^
type_struct.c:12:2: error: 'std' undeclared (first use in this function)
  std.age = 10;
  ^
type_struct.c:12:2: note: each undeclared identifier is reported only once for each function it appears in
```

发现编译出错，那么我们再来修改下。

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

int main(void)
{
    stu std;
    std.age = 10;
    printf("age = %d\n",std.age);
    return 0;
}
```

我感觉每次定义结构体变量前面都要加 struct 太累了

我在结构体分号前定义一个类型名

然后用这个类型名来定义结构体变量，这样就不用在前面加 struct 这么累了

我们在结构体前面加了 typedef，来让结构体可以在分号前定义结构体类型

这样就成功编译过了

```
root@ubuntu:/home/xiang/C_code# gcc -o type_struct type_struct.c
root@ubuntu:/home/xiang/C_code# ./type_struct
age = 10
root@ubuntu:/home/xiang/C_code#
```

定义结构体指针

```
3 typedef struct student
4 {
5     char name[20];
6     int age;
7 }student; 这个名字可以和类型名字一样
8
9 int main(void)
10 {
11     struct student *p1std;//结构体指针可以这样定义
12     student *p2std; //结构体指针也可以这样定义
13
14     student std = {"xxxxxxxx",10}//初始化结构体成员name和arg
15     p1std = &std;
16     p2std = p1std;
17     printf("name = %s\n",p1std->name);
18     printf("arg = %d\n",p2std->age);
19     return 0;
20 }
```

root@ubuntu:/home/xiang/C_code# gcc -o type_struct type_struct.c
root@ubuntu:/home/xiang/C_code# ./type_struct
name = xxxxxxxx
arg = 10

这就是结构体定义指针的基本方法

```
3 typedef struct student
4 {
5     char name[20];
6     int age;
7 }student,*Pstudent; 结构体可以定义类型，还可以定义指针类型
8
9 int main(void)
10 {
11
12     student std = {"xxxxxxxx",10}//初始化结构体成员name和arg
13     Pstudent p1std; //指针类型的结构体定义指针变量
14     p1std = &std;
15     printf("name = %s\n",p1std->name);
16     printf("arg = %d\n",p1std->age);
17     return 0;
18 }
```

root@ubuntu:/home/xiang/C_code# gcc -o type_struct type_struct.c
root@ubuntu:/home/xiang/C_code# ./type_struct
name = xxxxxxxx
arg = 10

结果一样

二级指针一般用来指向指针数组

```
int main()
{
    int a=10,b=20,c=30,d=40;
    int *array[4]={&a,&b,&c,&d};
    int **p;

    p = array;

    printf("获取指针数组第1个元素的地址和值\n");
    printf("p = %x\n",p); //二级指针p自己占用的地址
    printf("*p = %x &a = %x\n",*p,&a); /*p二级指针指向的变量地址
    printf("**p = %d\n",**p); //二级指针指向变量的值
    printf("获取指针数组第1个元素的地址和值\n");
    printf("p = %x\n",p);
    printf("*p = %x &a = %x\n",++*p,&b);
    printf("**p = %d\n",**p);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./twopoint
获取指针数组第1个元素的地址和值
p = 55825070
*p = 55825058 &a = 55825058
**p = 10
获取指针数组第1个元素的地址和值
p = 55825070
*p = 5582505c &a = 5582505c
**p = 20
root@ubuntu:/home/xiang/C_code#
```

这就是二级指针大部分都是用来指向指针数组的，用来获取指针数组里面存放的各个元素指向变量的地址。
但是这样有意义吗？我不知道直接用指针数组吗？

我们用二级指针来做子函数形参就有意义了

```
void twopoint(int **p)
{
    *p = (int *)0x12345678;
    p++;
    *p = (int *)0x87654321;
}

int main()
{
    int a=10,b=20,c=30,d=40;
    int *array[4]={&a,&b,&c,&d};

    printf("array[0] = %x  array[1] = %x\n",array[0],array[1]); //a变量的地址 b变量的地址
    twopoint(array);
    printf("array[0] = %x  array[1] = %x\n",array[0],array[1]); //a变量的地址 b变量的地址
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./twopoint
array[0] = 2ded0840  array[1] = 2ded0844
array[0] = 12345678  array[1] = 87654321
root@ubuntu:/home/xiang/C_code#
```

这就是二级指针，用子函数去修改主函数里面变量的地址。

```

void twopoint(int **p)
{
    *p = (int *)0x12345678;
    p++;
    *p = (int *)0x87654321;
}

int main()
{
    int a=10,b=20,c=30,d=40;
    int *array[4]={&a,&b,&c,&d};

    printf("array[0] = %x  array[1] = %x\n",array[0],array[1]); //a变量的地址 b变量的地址
    twopoint(array);
    printf("array[0] = %x  array[1] = %x\n",array[0],array[1]); //a变量的地址 b变量的地址
    return 0;
}

```

a	0x09
b	0x08
c	0x07
d	0x06
	0x05
0x06	0x04
0x07	0x03
0x08	0x02
0x09	0x01
数据	地址

内存

将 abcd 的变量地址分别给数组的每个元素，这样数组每个元素存放的就是变量的地址，而不是值，这就是指针数组

a	0x09
b	0x08
c	0x07
d	0x06
	0x05
0x06	0x04
0x07	0x03
0x08	0x02
0x09	0x01
数据	地址

内存

二级指针**p 里面 (*p 访问的是数组元素上的值，这值也是地址值)

a	0x09
b	0x08
c	0x07
d	0x06
	0x05
0x06	0x04
0x07	0x03
0x08	0x02
0x09	0x01
数据	地址

内存

二级指针**p 里面 (p 访问的是数组元素自己的地址)

```

void twopoint(int **p)
{
    **p = 100;
    p++;
    **p = 200;
}

int main()
{
    int a=10,b=20,c=30,d=40;
    int *array[4]={&a,&b,&c,&d};

    printf("*array[0] = %d  *array[1] = %d\n",*array[0],*array[1]);//a变量的地址 b变量的地址
    twopoint(array);
    printf("*array[0] = %d  *array[1] = %d\n",*array[0],*array[1]);//a变量的地址 b变量的地址
    return 0;
}

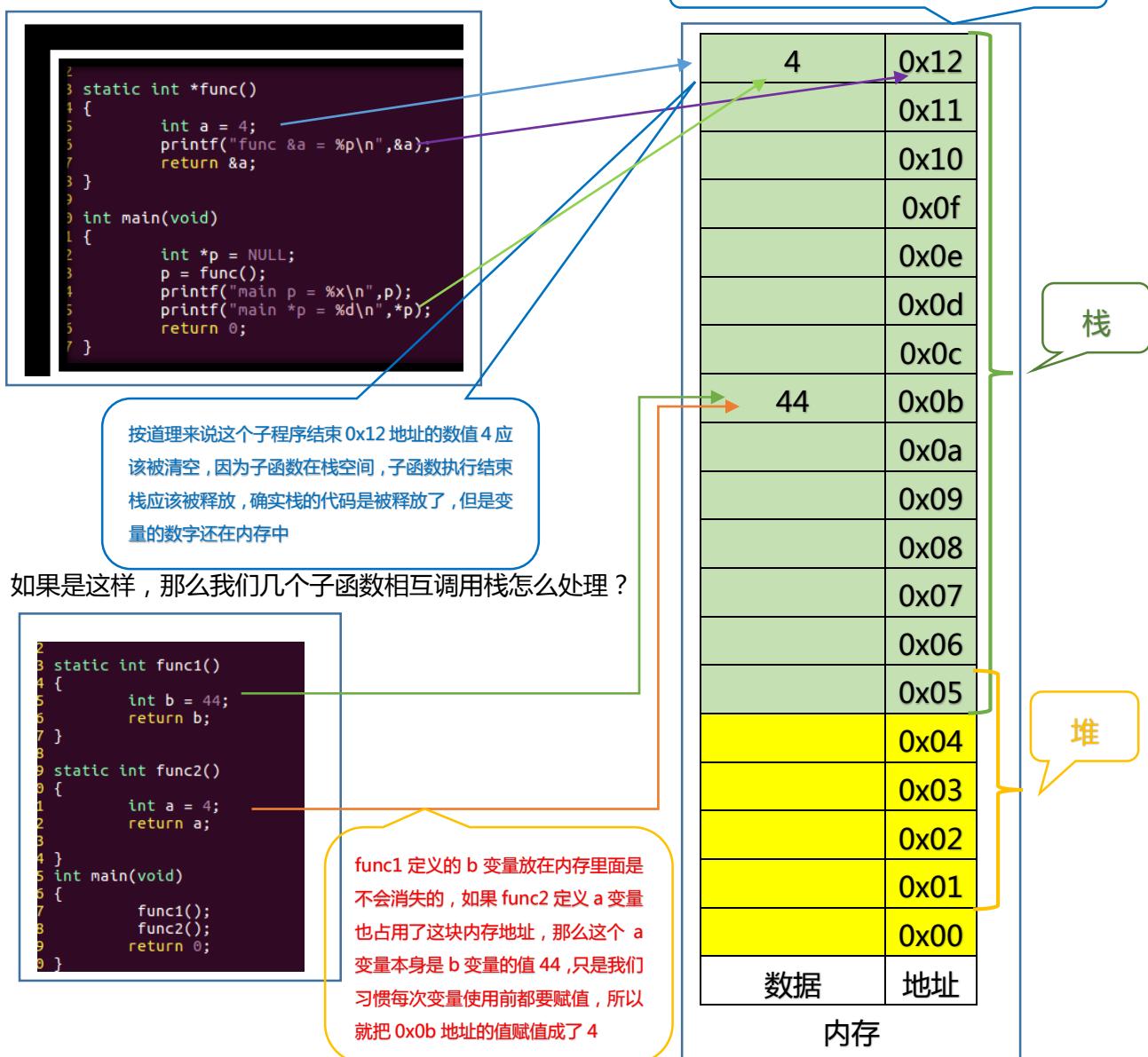
root@ubuntu:/home/xiang/C_code# ./twopoint
*array[0] = 10  *array[1] = 20
*array[0] = 100  *array[1] = 200

```

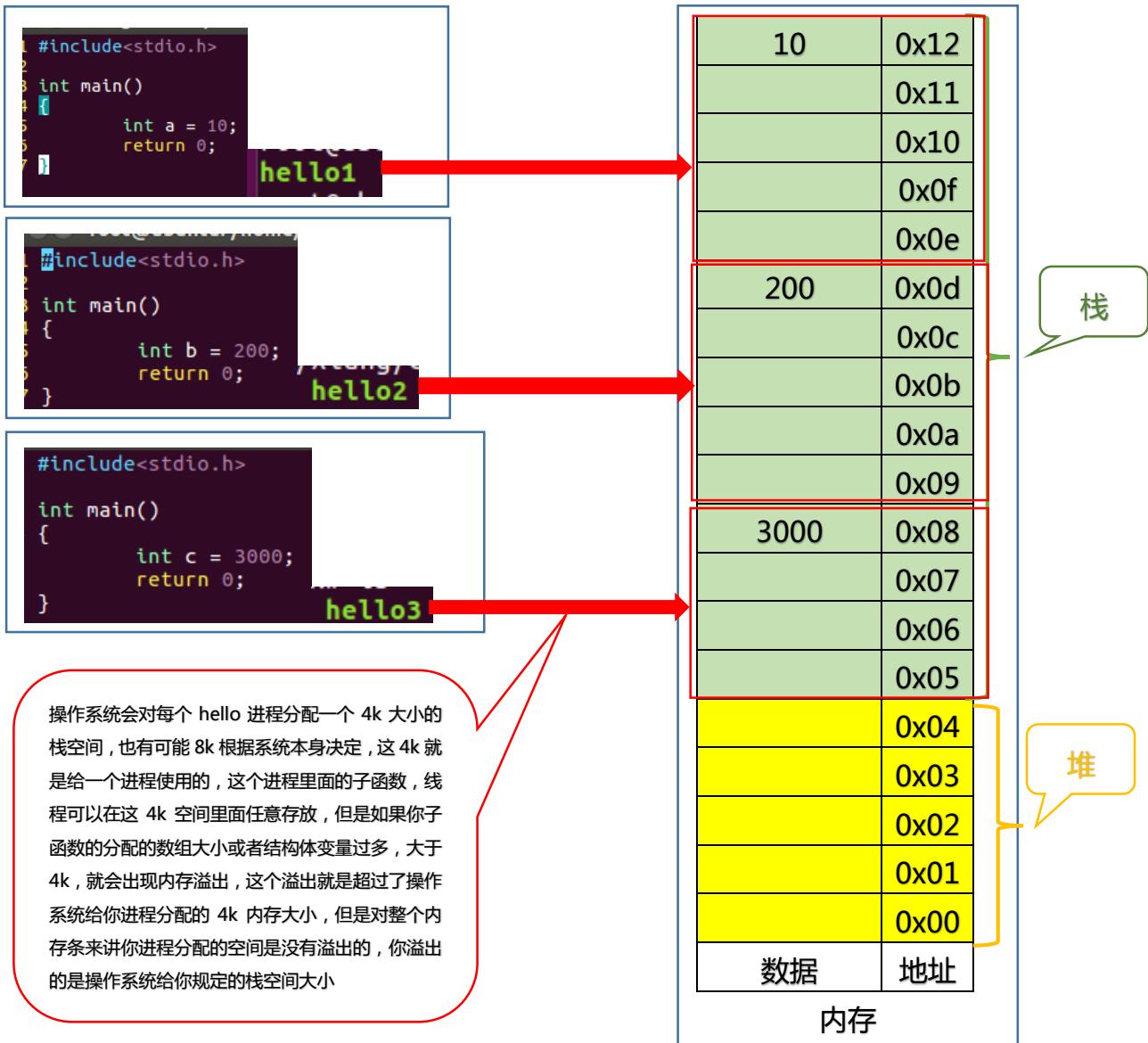
上面说我用 `p` 是取数组自己元素的地址，用`*p`是取数组元素上面存放的地址，但是这个值是地址，因为我是指针数组。然后`**p`就是取数组元素存放地址上存放的值。

C 语言各种程序在内存分布的情况分析

操作系统把一整块内存划分位堆，栈，.data 区域



所以同一块栈空间是可以被多个子函数反复使用的。一般操作系统是对每个进程分配一个栈空间

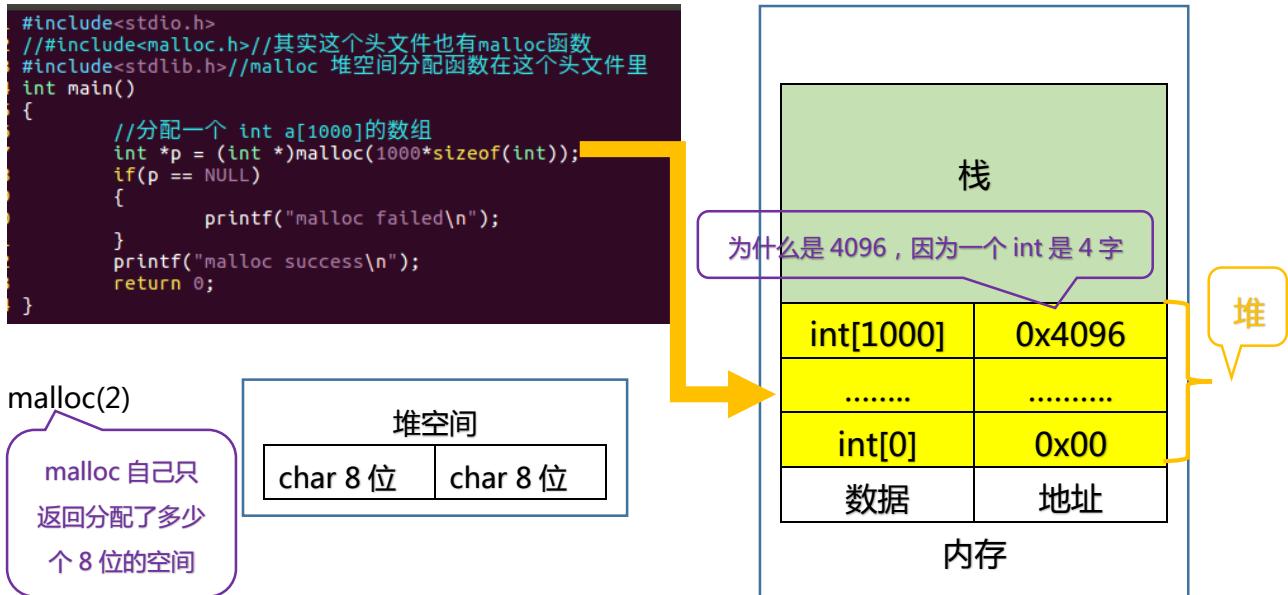


所以在定义变量存放大批量数据的时候不要用局部变量数组，全局变量数组。比如 `int a[1000000]`；或者结构体成员过多 `struct{int a[1000000000000] , int b[10000000000] ,.....} ;`；这些都会超过栈空间大小。

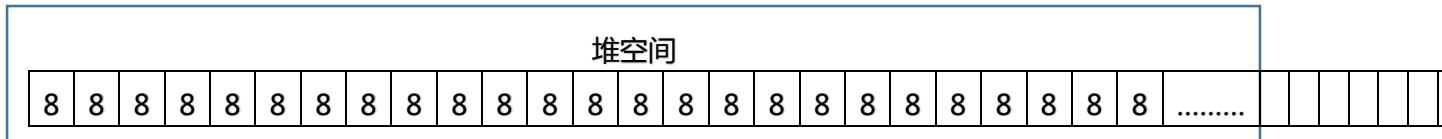
定义变量存放大批量数据我们要用 `malloc` 堆来分配，这样可以解决栈空间不够内存溢出问题。



就是 malloc 把 int a[1000]本来分配在栈的变量，放在了堆上。只不过把变量名改成了 p



malloc(10*sizeof (int))



malloc 分配了 40 个 8 位的空间

(int *)malloc(10*sizeof(int))



void *malloc(.....) //返回 void*的意思不是返回空 NULL , 而是 malloc 开辟了很多个字节的内存 , 但是它不知道怎么划分 , 返回个 void* , 你可以将这块内存用(int *) (char *)来定义用 (int *) , 将 malloc 分配的 40 个 8 位变成了 10 个 32 位的空间 , 那么每 1 个空间可以装 1 个 int 的值

```
1 #include<stdio.h>
2 #include<malloc.h>
3
4 typedef struct _soft_array
5 {
6     int len;
7     int array[10];
8 }softarray;
9
10 int main()
11 {
12
13     softarray *sa = (softarray *)malloc(sizeof(softarray)*10); //malloc分配什么类型, 就接受什么类型
14
15     int *i = (int *)malloc(10*sizeof(int)); //malloc分配什么类型, 就接受什么类型
16
17     char *c = (char *)malloc(10*sizeof(char)); //其实可以不用sizeof(char), 因为malloc最小分配单位就是char
18
19     long *l = (long *)malloc(10*sizeof(long)); //malloc分配什么类型, 就接受什么类型
20
21     return 0;
22 }
```

这就是 malloc 的好处 , 分配多少个空间就使用多少个空间 , 这些空间在堆里面 , 可以反复的使用。所以当你在开发新的 cpu 平台时 , 或者新版本 linux 操作系统时 , 要先查阅内存分配的堆 , 栈大小。

堆空间分配错误问题

```
1 #include<stdio.h>
2 #include<stdlib.h> //malloc 堆空间分配函数在这个头文件里
3 int main()
4 {
    //分配一个 int a[1000]的数组
    int *p = (int *)malloc(1000*sizeof(int));
    if(p == NULL)
    {
        printf("malloc failed\n");
    }
    else
    {
        printf("malloc success\n");
    }
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code/stack# ./mallooo
malloc success
```

堆分配成功

```
1 #include<stdio.h>
2 #include<stdlib.h> //malloc 堆空间分配函数在这个头文件里
3 int main()
4 {
    //分配一个 int a[1000000000000000]的数组
    int *p = (int *)malloc(1000000000000000*sizeof(int));
    if(p == NULL)
    {
        printf("malloc failed\n");
    }
    else
    {
        printf("malloc success\n");
    }
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code/stack# ./mallooo
malloc failed
```

堆分配失败

这是因为你分配的堆空间大小，已经超过了内存堆区域的大小了。

堆使用后一定要释放

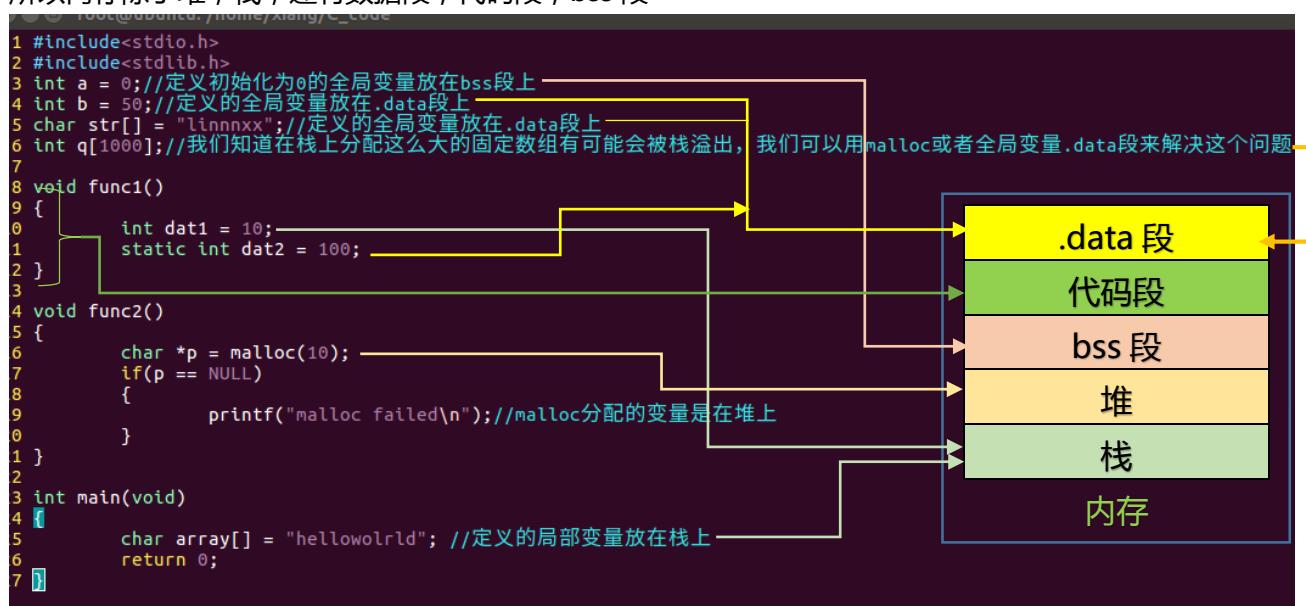
```
1 #include<stdio.h>
2 #include<stdlib.h> //malloc 堆空间分配函数在这个头文件里
3 int main()
4 {
    int *p = (int *)malloc(1000*sizeof(int));
    if(p == NULL) {printf("malloc failed\n");}
    else {printf("malloc success\n");}
    *p = 100;
    *(p+1) = 200;
    printf("*p = %d *(p+1) = %d\n", *p, *(p+1));
    free(p); //使用后一定要释放堆内存，方便其它程序使用这块堆区域
    return 0;
}
```

总结：不管是哪个进程，对变量进行堆分配，都在一个堆区域里面。不像栈区域，每个进程都有自己的4k栈，不管怎么定义变量，都在自己的栈里面，不会影响其它进程定义的变量。但是堆是公共厕所，所以某个进程使用完堆，一定要执行free，否则其它进程就无法再使用堆

```
root@ubuntu:/home/xiang/C_code/stack# ./mallooo
malloc success
*p = 100 *(p+1) = 200
```

数据段(.data) , 代码段 , bss 段

所以内存除了堆，栈，还有数据段，代码段，bss 段



所以最好在使用的开发板上搞清楚各区域的大小

代码段：就是放函数里面的 if ...else...for 之内的语句

.data 段：就是放定义的非 0 全局变量和加有 static 的局部变量，记住哦局部变量加了 static 那么就放在.data 段上，而不是栈上。

bss 段：就是放非 0 的全局变量，但是你给全局变量修改值后，就右放在.data 段了所以 bss 也算.data 段
局部变量放栈上

malloc 的变量指针和区域放堆上

还有一种特殊的情况

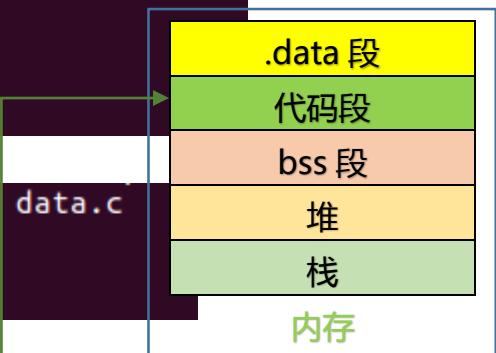
```
4
5 int main(void)
6 {
7     char *p = "linux";
8     *(p+0) = 'f'; //我们认为现在修改后应该是finux
9
10    return 0;
11 }
```

```
root@ubuntu:/home/xiang/C_code# gcc -o data data.c
root@ubuntu:/home/xiang/C_code# ./data
Segmentation fault (core dumped)
root@ubuntu:/home/xiang/C_code#
```

为什么是段错误？

```
int main(void)
{
    char *p = "linux"; //const char *p = "linux"
    //其实char *p = 指定字符串，就相当于const char *p。我们把该内容放在代码段。代码段是不允许被运行更改的
    *(p+0) = 'f';

    return 0;
}
```



函数递归调用

```
1 int jiecheng(int n)
2 {
3     if(n<1)
4         printf("值输入错误\n");
5
6     if(n==1)
7     {
8         return 1;
9     }
10    else
11    {
12        return n * jiecheng(n-1); //递归就是函数调用它自己
13    }
14 }
15
16 int main()
17 {
18     int a = 5;
19     printf("%d 的阶乘是 %d\n",a,jiecheng(a));
20     return 0;
21 }
```

```
root@ubuntu:/home/xiang/C_code# ./recursive
5 的阶乘是 120
```

我们用函数递归调用计算 5 的阶乘， $5 \times 4 \times 3 \times 2 \times 1 = 120$

阶乘就是一个数乘以比它小一个单位的数，一直乘到 1，最后得出来一个数。比如上面这个就是。

函数递归绝对不是函数循环。

递归的原理：

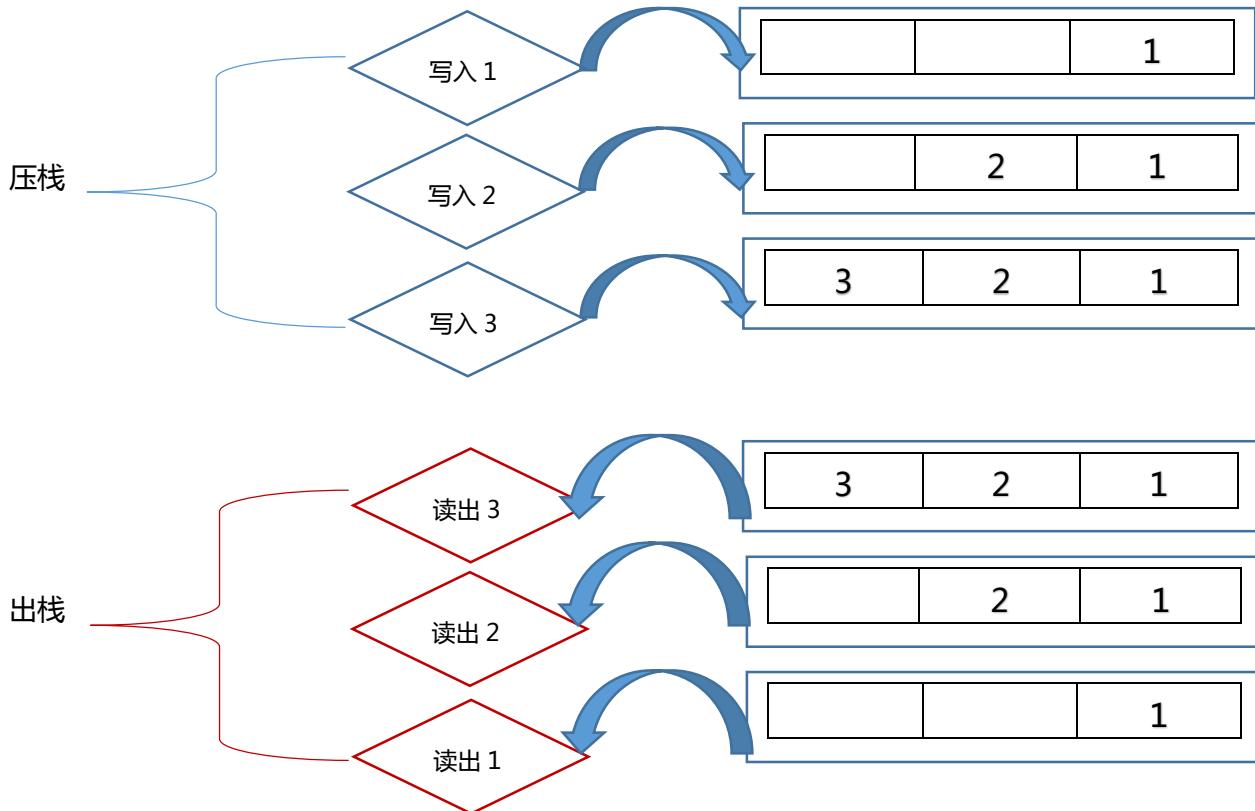
```
1 void digui(int n)
2 {
3     printf("递归前:n = %d\n",n);
4     if(n > 1)
5     {
6         digui(n-1);
7     }
8     else
9     {
10        printf("结束递归:n = %d\n",n);
11    }
12
13    printf("递归后:n = %d\n",n);
14 }
15
16 int main()
17 {
18     digui(3);
19     return 0;
20 }
```

root@ubuntu:/home/xiang/C_code# ./recursive
递归前:n = 3
递归前:n = 2
递归前:n = 1
结束递归:n = 1
递归后:n = 1
递归后:n = 2
递归后:n = 3

函数递归还可以用来求斐波拉茨数列

理解递归要用栈的概念

栈是一种先进后出的结构



所以入栈就类似向箱子里面放盘子，放很多个。出栈就是拿箱子的盘子，但是拿的这个盘子不是放入箱子的第一个盘子，而是放入箱子的最后一个盘子先拿出来。

C 语言字符串处理函数汇总

字符串处理函数在 linux 系统 C 库下面，其它操作系统就不清楚了

memcpy 函数

void *memcpy(void *dest, const void *src, size_t n);

字符串复制函数，从内存区域 src 拷贝 n 个字节到 dest，必须确保 src 和 dest 是内存不重复的区域

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

int main()
{
    char *a="abcdefg";
    char b[10];
    memcpy(b,a,sizeof(b));
    printf("%s\n",b);
    return 0;
}

int main()
{
    char *a="abcdefg";
    char b[10]="copy";
    memcpy(b,a,sizeof(b));
    printf("%s\n",b);
    return 0;
}

int main()
{
    char a[10]="abcdefg";
    char *b="copy";
    memcpy(b,a,sizeof(b));
    printf("%s\n",b);
    return 0;
}
```

将 a 指针的字符串全部拷贝给 b
数组存放

就算 b 数组里面有字符串也会被
memcpy 覆盖

如果是将数组的字符串拷贝给指
针是不行的，拷贝到的对象必须
是确定好的内存比如数组

```
root@ubuntu:/home/>
abcdefg
root@ubuntu:/home/>
```

```
root@ubuntu:/home/>
abcdefg
root@ubuntu:/home/>
```

```
root@ubuntu:/home/xiang/C_code# ./str
Segmentation fault (core dumped)
```

memmove 函数

memset 函数

```
extern void *memset(void *buffer, int c, int count)
```

buffer : 为指针或是数组

c : 是赋给 buffer 的值

count : 是 buffer 的长度.

```
int main()
{
    char a[10];
    memset(a,10,sizeof(a));
    printf("%d %d %d %d %d %d %d %d %d\n",
           a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9]);
    return 0;
}

root@ubuntu:/home/xiang/C_code# ./str
10 10 10 10 10 10 10 10 10 10
root@ubuntu:/home/xiang/C_code#
```

我们一般把数组或者内存全部填 0 就叫清空内存

```
int main()
{
    char a[10];
    memset(a,0,sizeof(a));
    printf("%d %d %d %d %d %d %d %d %d\n",
           a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9]);
    return 0;
}

root@ubuntu:/home/xiang/C_code# ./str
0 0 0 0 0 0 0 0 0 0
root@ubuntu:/home/xiang/C_code#
```

memcmp 函数

```
int memcmp(const void *buf1, const void *buf2, unsigned int count);
```

当 buf1<buf2 时 , 返回值小于 0

当 buf1==buf2 时 , 返回值=0

当 buf1>buf2 时 , 返回值大于 0

count 是比较字符次数 , 如果两个字符串变量第一个相等就继续向下比 , 根据 count 大小决定向下比好多个。但是一旦发现字符串比较结果为大于或者小于就立即返回 , 不再向下比较。不管你写多少个 count 都不会继续向下比较。

```
int main()
{
    char *a = "hhaa";
    char *b = "hhhh";
    int ret = memcmp(a,b,2);
    printf("%d\n",ret);
    if(ret>0)
        printf("char a > b\n");
    if(ret == 0)
        printf("char a = b\n");
    if(ret < 0)
        printf("char a < b\n");
    return 0;
}

root@ubuntu:/home/xiang/C_code# ./str
0
char a = b
root@ubuntu:/home/xiang/C_code#
```

我写了 count 比较两个字符串变量的前两个字符 , 得到的结果是相等 , 返回 0

```

int main()
{
    char *a = "hhaa";
    char *b = "hhhh";
    int ret = memcmp(a,b,3);
    printf("%d\n",ret);
    if(ret>0)
        printf("char a > b\n");
    if(ret == 0)
        printf("char a = b\n");
    if(ret < 0)
        printf("char a < b\n");
    return 0;
}

```

我将 count 改成 3 , memcmp 函数发现比较到第三个字符串的时候 , 根据 ASCII 码 , h>a , 所以根据 buf1<buf2 返回负数

```

root@ubuntu:/home/xiang/C_code# ./str
-7
char a < b

```

```

int main()
{
    char *a = "hhhh";
    char *b = "ahhh";
    int ret = memcmp(a,b,4);
    printf("%d\n",ret);
    if(ret>0)
        printf("char a > b\n");
    if(ret == 0)
        printf("char a = b\n");
    if(ret < 0)
        printf("char a < b\n");
    return 0;
}

```

我将 count 改成 4 , 但是发现只比较了第一个字符就返回了 , 所以 count 成立条件必须是 buf1 和 buf2 的字符串相等 才按照 count 继续向下比较 , 在比较的过程中出现了大于或者小于就停止对比 , 直接返回

```

root@ubuntu:/home/xiang/C_code# ./str
7
char a > b
root@ubuntu:/home/xiang/C_code#

```

memchr 函数

```
void *memchr(const void *str, int c, size_t n)
```

str : 要处理的字符串数组 , 或者指针首地址

c : 我要寻找的字符 , 找到该字符 memchr 就返回该字符在数组中的地址 , 也就是元素下标

n : 这个字符串有多长

```

int main()
{
    char *web = "abcdefghijklmn";
    char *ret = memchr(web,'c',strlen(web));
    printf("%s\n",ret);

    return 0;
}

```

返回找到的字符串地址给 ret

从 ret 地址开始继续输出后面的字符串

```

int main()
{
    char *web = "aabbccddeefghijklmn";
    char *ret = memchr(web, 'c', strlen(web));
    printf("%s\n", ret);

    return 0;
}

```

root@ubuntu:/home/xiang/C_code# ./str

ccddeefghijklmn

memchar 函数就是不管你有多少个相同的字符串，都是以第一个找到的字符串为准，返回它的地址。

strcpy 函数

```

char* strcpy(char* des, const char* source)
//参数：des 为目标字符串，source 为原字符串。

```

把 source 空间的字符串以 '\0' 结尾全部复制到 des 空间内

```

1 #include<stdio.h>
2 #include<string.h>
3
4 int main()
5 {
6     char *src = "abcdefghijklm";
7     char dest[20];
8     strcpy(dest,src);
9     printf("%s\n",dest);
10    return 0;
11 }

```

root@ubuntu:/home/xiang/C_code# ./str

abcdefghijklm

复制成功

strncpy 函数

```

char *strncpy(char *dest,char *src,size_t n);

```

src 是源字符存放空间

dest 是要将 src 里面的字符要赋值到的位置

n 是要复制多少个

```

int main()
{
    char *src = "abcdefghijklm";
    char dest[20];
    strncpy(dest,src,3);
    dest[3] = '\0';
    printf("%s\n",dest);
    return 0;
}

```

strncpy 和 strcpy 相比 strncpy 是指定要
复制多少个字符，而 strcpy 是全部复制

Strncpy 复制后的字符不自
带'\0'，所以要自己在后面加

strcat 函数

extern char *strcat(char *dest, const char *src);

把 `src` 所指字符串添加到 `dest` 结尾处(覆盖 `dest` 结尾处的'\0')。

相当于就是合并两个字符串

```
int main()
{
    char *src = "abc";
    char dest[20] = "1234";
    strcat(dest, src);
    printf("%s\n", dest);
    return 0;
}
root@ubuntu:/home/xiang/C_code# ./str
1234abc
```

Strncat

dest 开辟的空间要足够存放自己的字符串和合并过来的字符串

strcmp 函数

extern int strcmp(const char *s1, const char *s2);

当 `s1 < s2` 时，返回为负数；

当 `s1 == s2` 时，返回值 = 0；

当 `s1 > s2` 时，返回正数。

两个字符串自左向右逐个字符相比 (按 ASCII 值大小相比较)，直到出现不同的字符或遇'\0'为止

```
int main()
{
    char *src = "abc";
    char dest[20] = "1234";
    int ret = strcmp(dest, src);
    if(ret == 0)
        printf("dset == src\n");
    if(ret < 0)
        printf("dset < src\n");
    if(ret > 0)
        printf("dset > src\n");
    return 0;
}
root@ubuntu:/home/xiang/C_code# ./str
dset < src
```

```
int main()
{
    char *src = "abc";
    char dest[20] = "abef";
    int ret = strcmp(dest, src);
    if(ret == 0)
        printf("dset == src\n");
    if(ret < 0)
        printf("dset < src\n");
    if(ret > 0)
        printf("dset > src\n");
    return 0;
}
```

比较两个字符串，前面 ab 都是一样的就继续向下比较，发现 e 和 c 不一样，就对比大小

```
root@ubuntu:/home/xiang/C_code# ./str
dset > src
```

strcmp

int strcmp (const char * str1, const char * str2, size_t n);

str1 , str2 比较 , 和 strcmp 不一样的是 , 指定比较的个数 , n 为个数

```
int main()
{
    char *src = "abc";
    char dest[20] = "abef";
    int ret = strncmp(dest, src, 2);
    if(ret == 0)
        printf("dset == src\n");
    if(ret < 0)
        printf("dset < src\n");
    if(ret > 0)
        printf("dset > src\n");
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
1234abc
```

这里写 2 就是只比较前面两个 , 两个字符串前面两个都是 ab 所以结果是相等

```
int main()
{
    char *src = "abc";
    char dest[20] = "abef";
    int ret = strncmp(dest, src, 3);
    if(ret == 0)
        printf("dset == src\n");
    if(ret < 0)
        printf("dset < src\n");
    if(ret > 0)
        printf("dset > src\n");
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
dset > src
```

这里 3 就是只比较前面 3 个 , 两个字符串第 3 个正好不相等 , 就比较出大小了

strchr 函数

char *strchr(const char* _Str, char _Val)

str 要查找的字符串

val 字符串里面的某个字符

返回查找到字符串某个字符的地址

```
int main()
{
    char *src = "abcdefg";
    char *p = strchr(src, 'b');
    printf("%s\n", p);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
bcdefg
```

我从左向右查找第 1 个 b ,
然后把 b 的地址返回给指
针 , 指针就从 b 这个地址
开始向后操作

strstr 函数

```
extern char *strstr(char *str1, const char *str2);
```

str1 字符串地址

str2 写一个字符串，然后查找 str1 字符串里面有没有包含 str2 写的字符，有的话就截取包含字符的地址给返回值，没有的话就返回 NULL

```
int main()
{
    char src1[] = "abcdefg";
    char *p = strstr(src1,"cd");
    if(p == NULL)
        printf("no !!\n");
    else
        printf("%s\n",p);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
cdefg
```

该 cd 字符串 src1 包含了的

```
int main()
{
    char src1[] = "abcdefg";
    char *p = strstr(src1,"12");
    if(p == NULL)
        printf("no !!\n");
    else
        printf("%s\n",p);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
no !!
```

该 12 字符串 src1 不包含

strtok 函数

```
char* strtok (char* str, constchar* delimiters );
```

str 字符串地址

delimiters 填入用字符串里面的什么字符来当做分割符

```
int main()
{
    char src1[] = "helloifclizmg";
    char *p = strtok(src1,"c");
    if(p == NULL)
        printf("no !!\n");
    else
        printf("%s\n",p);
    return 0;
}
```

```
root@ubuntu:/home/xiang/C_code# ./str
helloifu
root@ubuntu:/home/xiang/C_code#
```

Src1 字符串里面 c 开始后面的
字符我就不要了

Linux 系统下 C 文件制作静态库

```
1 #include<stdio.h> //我们用来做库的c文件也可以调用其它库的c文件
2
3 void xzz_func1()
4 {
5     printf("enter xzz_func1...\n");
6 }
7 int xzz_func2(int a,int b)
8 {
9     return a-b;
0 }
```

准备一份 C 文件 xzz.c

```
1 #ifndef __XZZ_H
2 #define __XZZ_H
3 void xzz_func1();
4 int xzz_func2(int a,int b);
5
6 #endif
```

准备一份和 C 对应的头文件 xzz.h

然后我有其它的 C 文件要调用 xzz.c 里面的函数，那么我先得把这个 xzz.c 文件作出静态库

```
root@ubuntu:/home/xiang/C_code/lib# gcc -o xzz.o xzz.c -c
root@ubuntu:/home/xiang/C_code/lib# ls
xzz.c xzz.h xzz.o
root@ubuntu:/home/xiang/C_code/lib#
```

把 c 文件制
作成.o

用 gcc -o 指定编译出来的文件名字，.o 后缀是中间文件，因为制作静态库不能用.c 文件直接编译成.a，必须经过中间文件.o 所以我们制作出了 xzz.o 文件

```
root@ubuntu:/home/xiang/C_code/lib# ar -rc libxzz.a xzz.o
root@ubuntu:/home/xiang/C_code/lib# ls
libxzz.a xzz.c xzz.h xzz.o
root@ubuntu:/home/xiang/C_code/lib#
```

ar -rc libxzz.a xzz.o 就是把 xzz.o 制作成静态库，静态库都必须要求在你的文件名字前面加 lib，所以叫 libxzz.a，然后其他程序就可以调用 libxzz.a 里面的函数了。

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ls
libxzz.a test.c xzz.h
```

我新建了一个 test.c 文件，在这个文件里面我调用了 xzz.c 里面的函数，那么我就需要把 libxzz.a 和 xzz.h 这两个文件放在和 test.c 同一个目录下

```
1 #include<stdio.h>
2 #include "xzz.h" //包含你需要调用函数的头文件
3
4 int main()
5 {
6     xzz_func1(); // 调用 libxzz.a 里的函数
7     int a = xzz_func2(5,4);
8     printf("a = %d\n",a);
9     return 0;
0 }
```

调用 libxzz.a 文件里
面的函数，也就是
xzz.c 文件里面的函数

```
root@ubuntu:/home/xiang/C_code/lib/testlib# gcc -o test test.c  
/tmp/cce5x6bu.o: In function `main':  
test.c:(.text+0xe): undefined reference to `xzz_func1'  
test.c:(.text+0x1d): undefined reference to `xzz_func2'  
collect2: error: ld returned 1 exit status
```

这种.text+....是属于
链接报错，编译是通
过了，链接出问题

但是链接怎么出错了？这是因为你 test.c 调用了 xzz 文件里面的函数，所以你编译的时候你要把你调用的文件都指定进来。

```
root@ubuntu:/home/xiang/C_code/lib/testlib# gcc -o test test.c -lxzz  
/usr/bin/ld: cannot find -lxzz  
collect2: error: ld returned 1 exit status
```

因为我们的 libxzz.a 不在 gcc 编译器默认寻找的路径库文件里面，所以我们要用-L 来指定链接库

Linux 系统就是很扯淡，编译库的时候在文件前面加 lib，-L 链接库的时候要取消库文件前面的 lib

但是为什么还是出错呢？，但是这次不是链接错误，是找不到库文件

```
collect2: error: ld returned 1 exit status  
root@ubuntu:/home/xiang/C_code/lib/testlib# gcc -o test test.c -lxzz -L./  
root@ubuntu:/home/xiang/C_code/lib/testlib# ls  
libxzz.a test test.c xzz.h
```

加-L 就可以了，这个-L 大写 L 是指定你的库文件在哪个路径里面，gcc 编译器它不知道你库文件在哪个路径下，所以要用-L 来指定。 -l(小写)指定库文件名，-L(大写)指定库文件存放路径

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ./test  
enter xzz_func1...  
a = 1
```

Linux 系统下 C 文件制作动态库

```
root@ubuntu:/home/xiang/C_code/lib# ls  
testlib xzz.c xzz.h
```

我们还是用前面制作静态库的文件来制作动态库

```
root@ubuntu:/home/xiang/C_code/lib# gcc xzz.c -o xzz.o -c -fPIC  
root@ubuntu:/home/xiang/C_code/lib# ls  
testlib xzz.c xzz.h xzz.o
```

先用-fPIC 把 xzz.c 编译成 xzz.o，这个.o 具备动态库制作的属性

```
root@ubuntu:/home/xiang/C_code/lib# gcc -o libxzz.so xzz.o -shared  
root@ubuntu:/home/xiang/C_code/lib# ls  
libxzz.so testlib xzz.c xzz.h xzz.o
```

再用-shared 把.o 文件编译成.so 动态库，libxzz.so 就是动态库

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ls  
libxzz.so test.c xzz.h
```

和静态库一样把动态库.so 和 h 文件放进调用他们函数的目录里面

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ls  
libxzz.so test.c xzz.h
```

编译失败，和静态库一样，需要指定库

```
root@ubuntu:/home/xiang/C_code/lib/testlib# gcc -o test test.c  
/tmp/cc1j4al8.o: In function `main':  
test.c:(.text+0xe): undefined reference to `xzz_func1'  
test.c:(.text+0x1d): undefined reference to `xzz_func2'  
collect2: error: ld returned 1 exit status
```

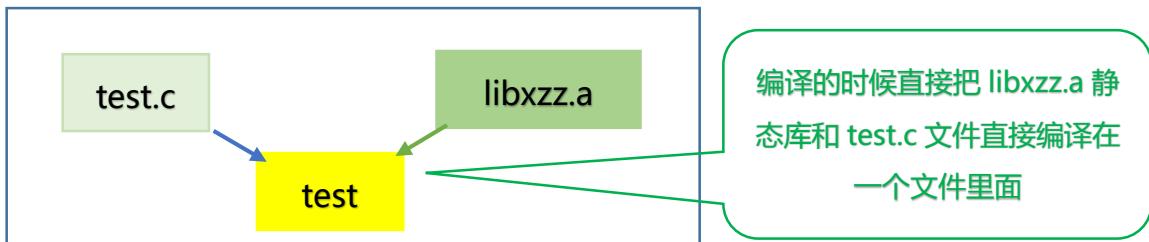
指定库后编译成功

生成可执行文件

运行失败

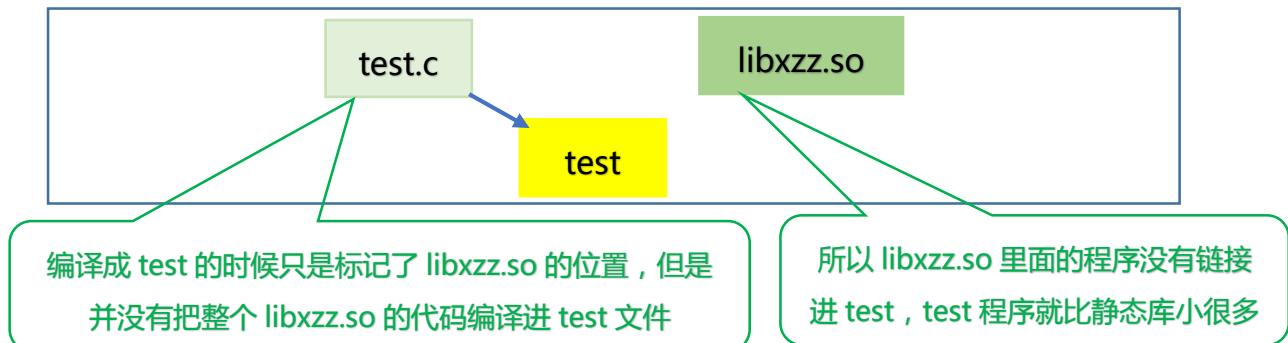
为什么会这样呢？，这就是静态库和动态库的区别了

静态库编译原理：



这样的话静态库编译出来的执行程序就比较大，因为里面包含了静态库整个代码，test.c 整个代码，所以你直接-I -L 指定就可以编译出来正常运行。

动态库编译原理：



当执行 test 程序的时候，test 会去默认标记的目录位置找 libxzz.so 文件，但是我们用-L 把 libxzz.so 标记到当前目录下，所以 test 执行的时候用默认链接库的方法找不到。因为默认链接库不是你的当前目录，而是/usr/lib 目录，这是 linux 系统规定的，你无法用-L 去修改动态库的链接位置。

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ls
libxzz.so  test  test.c  xzz.h
root@ubuntu:/home/xiang/C_code/lib/testlib# cp libxzz.so /usr/lib
root@ubuntu:/home/xiang/C_code/lib/testlib#
```

我们把 libxzz.so 放在/usr/lib 目录下

```
root@ubuntu:/home/xiang/C_code/lib/testlib# ./test
enter xzz_func1...
a = 1
root@ubuntu:/home/xiang/C_code/lib/testlib#
```

你看这样就能执行了。所以编译的方法没有错，错是错在忘记把.so 库文件放在/usr/lib 目录下

所以在 ARM 平台上，你 arm-linux-gcc 在 X86 上面编译的库一定要拷贝一份到 ARM 平台的文件系统下的/usr/lib 目录下，然后你在开发板上执行的程序，用到了动态库会自动去开发板的/usr/lib 目录下找

在 X86 平台上你编译的库可能要依赖其他库，所以这个其他库编译出来后要放在交叉编译器的/usr/lib 下