C语言深度剖析 石虎

第一章 关键字

绪

定义和声明最重要的区别是； 定义创建了对象并为这个对象分配了内存

声明没有分配内存

1.1 auto

1.2 register

1.3 static

作用一： 修饰变量。

1.静态全局变量

作用域仅限于变量被定义的文件中。其他文件即使用extern声明也没法使用他。在定义前面的代码行也不能使用它，想要使用就得在前面加extern。介意直接在文件顶端定义静态全局变量。

2.静态局部变量

在函数体内部定义的变量，只能在该函数里使用。和静态全局变量一样，总是存在内存的静态区。变量的值在函数运行结束也不会被销毁。函数下次使用仍然是这个值。



作用二： 修饰函数。

函数前加static使得函数成为静态函数。此处的static不是指存储方式，而是指函数的作用域仅限于本文件（又称为内部函数）。不用担心自己定义的函数是否会与其他文件中的函数同名。

1.4.1 基本数据类型



1.4.2 数据命名规则

标识符的一般规则：

1.命名只管可拼读，望文生义，便于记忆。

2.命名长度应符合最小长度和最大信息的原则。英文尽量不缩写。

3.标识符由多个词组成时，每个词的第一个字母大写，其余全部小写。

4.除了驱动开发的管脚命名等，尽量避免出现数字编号。

5.多文件时全局变量或函数要加范围限定符。

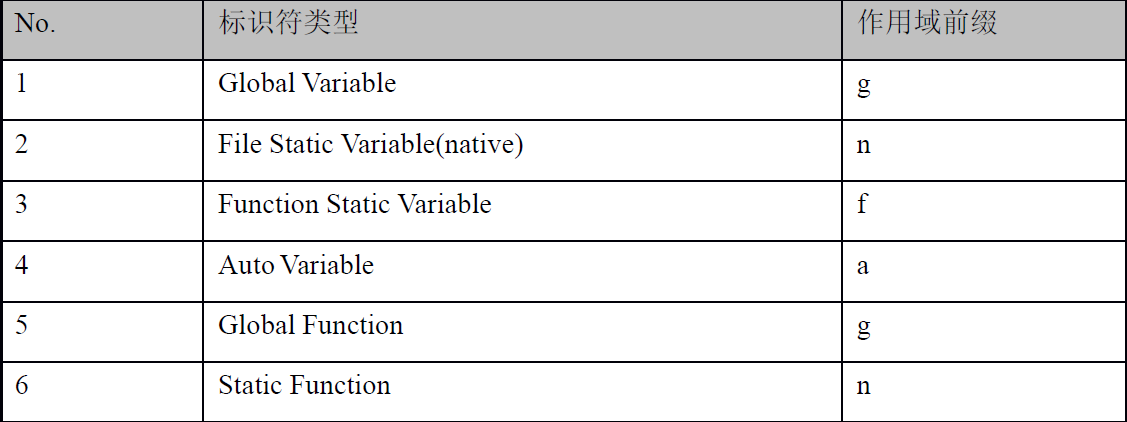
标识符的命名规则：

6.标识符分为两部分：规范标识符前缀+含义标识。非全局变量可以不使用范围限定符前缀。

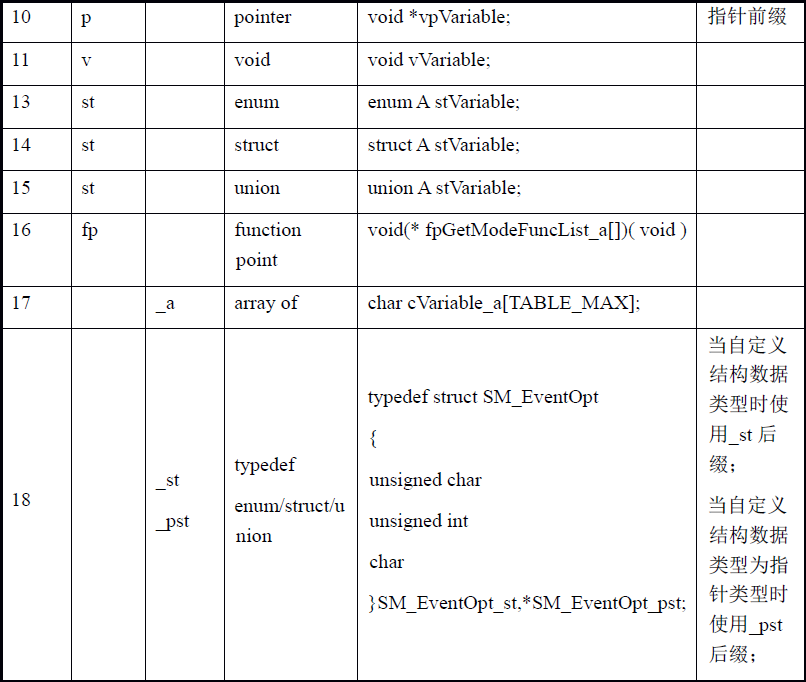
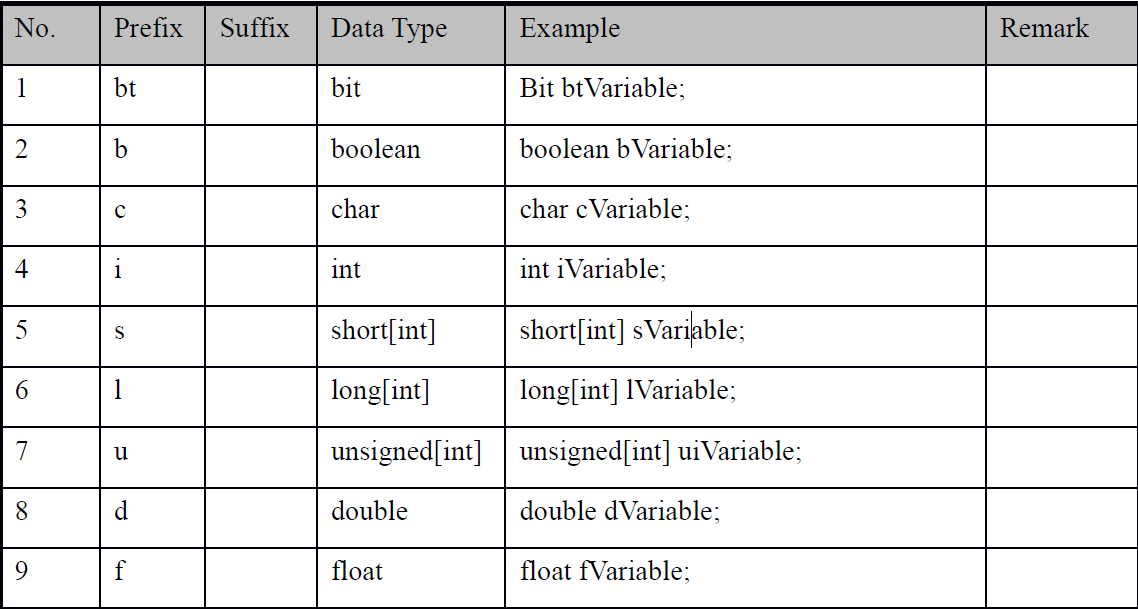
|-标识符前缀-|

模块名缩写 \_ 作用域前缀 数据类型前缀 [指针前缀] 含义标识 数组/结构后缀

作用域前缀



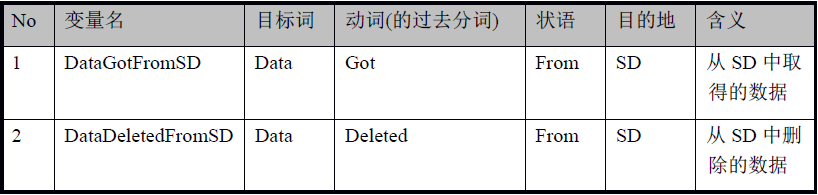
数据类型前缀



含义标识

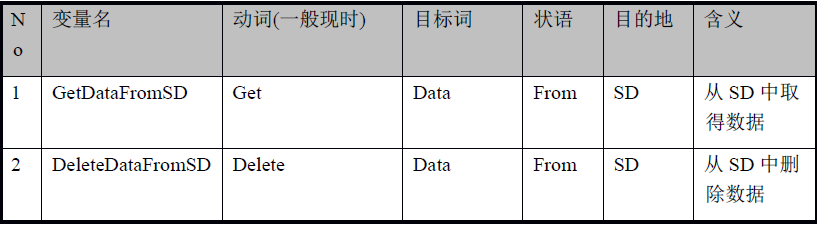
1）变量命名使用名词性词组

变量含义标识符构成：目标词 + 动词（的过去分词）+ [状语] + [目的地]



2）函数命名使用动词性词组

函数含义标识符构成： 动词（一般现在时）+ 目标词 + [状语] + [目的地]



7.程序中不得出现仅靠大小写区分的相似标识符。

8.一个函数名禁止被用于定义其他变量名等。

9.所有宏定义、枚举常数、只读变量全用大写字母命名，用下划线分割单词。

10.定义变量的同时千万千万别忘了初始化。定义变量时编译器并不一定清空了这块内存，它的值可能是无效数据。

11.不同类型的数据之间的运算要注意精度扩展问题。

1.5 sizeof关键字



1.6 signed unsigned

1. 一个32位的signed int类型整数其值表示法范围为：- 231～231 -1；

2. 8位的char类型数其值表示的范围为- 27～27 -1;

3 一个 32位的 unsigned int类型整数其值表示法范围为：0～ 232 -1；

4. 8位的 char类型数其值表示的范围为 0～28 -1。

5. 同样我们的 signed 关键字也很宽恒大量，你也可以完全当它不存在，编译器缺省默认情况下数据为signed 类型的。

数值一律用补码来表示（存储）。主要原因是使用补码，可以将符号位和其它位统一处理；同时，减法也可按加法来处理。正数的补码与原码相同，求负整数的补码，将其对应正数二进制表示所有位取反（0变1，1变0）后加1 。这种方式最后需要加上最前面的符号位。

如果是char型数据

负数 正数二进制 取反 +1 补码（2的8次-负数的绝对值）

-1 00000001 11111110 11111111 11111111(0xff)

-2 00000010 11111101 11111110 11111110(0xfe)

.

.

-127 01111111 10000000 10000001 10000001(0x81)

-128 10000000 01111111 10000000 10000000(0x80)

-129 10000001 01111110 101111111 01111111(0x7f)

-256 100000000 011111111 1100000000 00000000(0x00)

注；红色的是符号位

-129因为正数取反+1 后为01111111，八位有符号数不能是-129，发生了溢出，最高位符号位1丢失，剩下的8位为0x7f。

（-1）+（-127）

11111111 （-1）

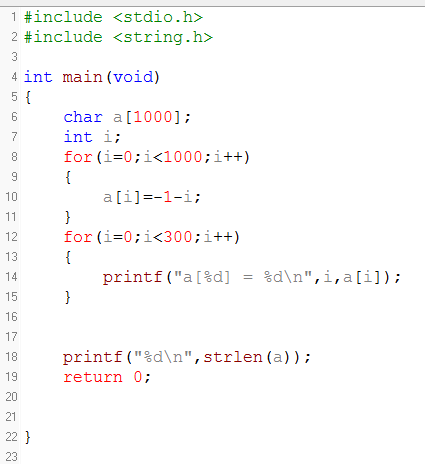
10000001 （-127）

------------------

1(舍弃)10000000 （-128） 存储仍然是补码形式

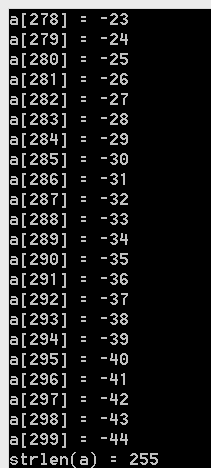
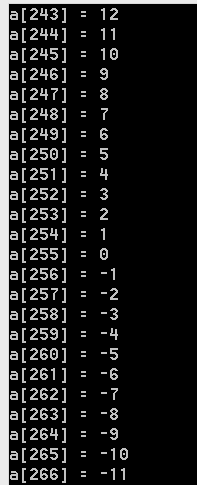
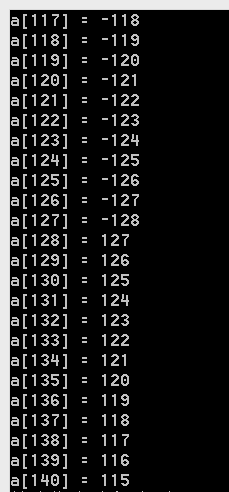
0：0000 0000  
-1：1111 1111（总之-1加1溢出后是0）  
-2：1111 1110（总之-2加1是-1）

-127：1000 0001(总之-127+127溢出后是0)  
-128：1000 0000（总之-128+1是-127）  
即-128加上127再加1溢出后是0——C/C++里存放整数的统一性。



分析：a[i] 从 -1 --------> -128

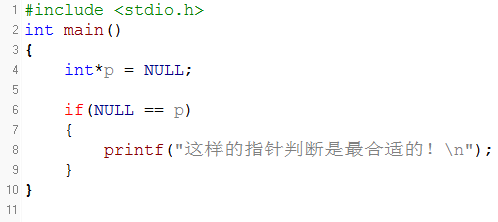
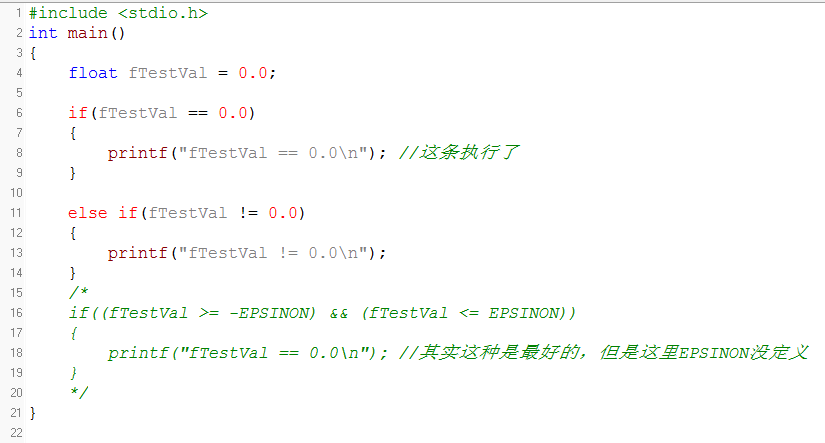
-129溢出，溢出以后的值是127，所以又从 127 -------> 0



strlen 函数是计算字符串长度的，并不包含字符串最后的‘\0’。而判断一个字符串是否结束的标志就是看是否遇到‘\0’。如果遇到‘\0’，则认为本字符串结束。



1.7 if else



1.7 switch case

1.if、else 一般表示两个分支或是嵌套表示少量的分支，但如果分支很多的话……还是用switch、case 组合。

2.最后必须使用default 分支。即使程序真的不需要default 处理，也应该保留语句。

3. case 后面只能是整型或字符型的常量或常量表达式（想想字符型数据在内存里是怎么存的）。

4.按字母或数字顺序排列各条case 语句。

5.把正常情况放在前面，而把异常情况放在后面。

6.按执行频率排列case语句。

7.简化每种情况对应的操作。一般来说case语句后面的代码尽量不要超过20行。

8.不要为了使用case 语句而刻意制造一个变量。

9.把default 子句只用于检查真正的默认情况。

1.8 do while for

1.在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数。

2.半开半闭区间写法和闭区间写法虽然功能是相同，但相比之下，半开半闭区间写法更加直观。

3.不能在for循环体内修改循环变量，防止循环失控。

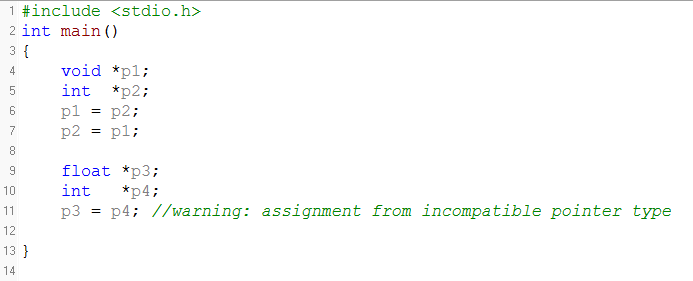
4.循环要尽可能的短，要使代码清晰，一目了然。

5.把循环嵌套控制在3层以内。

1.9 void 和 void\*

1. void 的字面意思是“空类型”，void \*则为“空类型指针”，void \*可以指向任何类型的数据。

2.任何类型的指针都可以直接赋值给void\*，无需进行强制类型转换。



3.如果函数没有返回值，那么应声明为void类型。

4.千万小心又小心使用void指针类型。

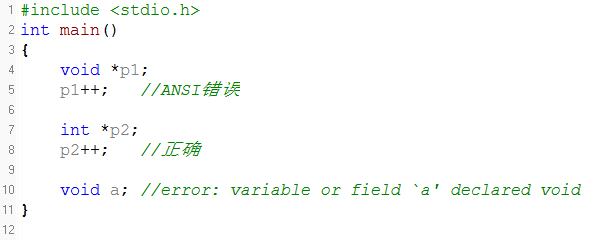
5.如果函数的参数可以是任意类型指针，那么应声明其参数为void \*。

例如：void \* memcpy(void \*dest, const void \*src, size\_t len);

void \* memset ( void \* buffer, int c, size\_t num );

这样，任何类型的指针都可以传入memcpy 和memset 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果memcpy和memset 的参数类型不是void \*，而是char \*，那才叫真的奇怪了！这样的memcpy 和memset明显不是一个“纯粹的，脱离低级趣味的”函数！memcpy 和memset 函数返回的也是void \*类型，标准库函数的编写者都不是一般人。

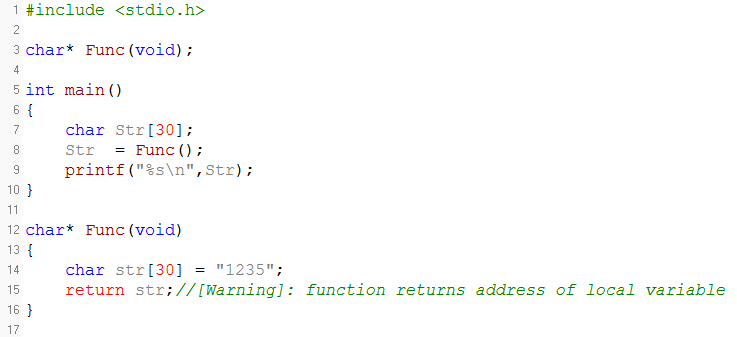
6. void 不能代表一个真实的变量。



1.10 return

1. return 语句不可返回指向“栈内存”的“指针”，因为该内存在函数体结束时

被自动销毁。



1.11 const

1.const修饰的只读变量

//精确的说应该是只读的变量，其值在编译时不能被使用

//因为编译器在编译时不知道其存储的内容。

//const推出的初始目的正是为了取代预编译指令，消除它的缺点，同时继承它的优点。

//编译器通常不为普通const 只读变量分配存储空间，而是将它们保存在符号表中

//这使得它成为一个编译期间的值，没有了存储与读内存的操作，使得它的效率也很高。

2.节省空间，避免不必要的内存分配，同时提高效率

#define M 3 //宏常量

const int N=5; //此时并未将N放入内存中

int i=N; //此时为N分配内存，以后不再分配！

int I=M; //预编译期间进行宏替换，分配内存

int j=N; //没有内存分配

int J=M; //再进行宏替换，又一次分配内存！

const定义的只读变量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define

一样给出的是立即数，所以，const 定义的只读变量在程序运行过程中只有一份拷贝（因为它是全局的只读变量，存放在静态区），而#define 定义的宏常量在内存中有若干个拷贝。#define 宏是在预编译阶段进行替换，而const 修饰的只读变量是在编译的时候确定其值。#define 宏没有类型，而const 修饰的只读变量具有特定的类型。

3.修饰变量

int const k = 2;

const int m = 3; //两种写法都可以

int const a[5]={1, 2, 3, 4, 5};

const int b[5]={1, 2, 3, 4, 5};

const int \*p; // p可变，p指向的对象不可变

int const \*p1; // p可变，p指向的对象不可变

int \*const p2; // p不可变，p指向的对象可变

const int \*const p3; //指针p和p指向的对象都不可变

先忽略类型名（编译器解析的时候也是忽略类型名），我们看const 离哪个近。“近水楼

台先得月”，离谁近就修饰谁。

const \*p; //const 修饰\*p,p 是指针，\*p 是指针指向的对象，不可变

const \*p; //const修饰\*p,p 是指针，\*p 是指针指向的对象，不可变

\*const p; //const修饰p，p 不可变，p 指向的对象可变

const \*const p; //前一个const 修饰\*p,后一个const 修饰p，指针p 和p 指向的对象

都不可变

4.修饰函数

const 修饰符也可以修饰函数的参数，当不希望这个参数值被函数体内意外改变时使用。例如：void Fun(const int i);告诉编译器i 在函数体中的不能改变，从而防止了使用者的一些无意的或错误的修改。

5.修饰函数的返回值

const 修饰符也可以修饰函数的返回值，返回值不可被改变。例如：

const int Fun (void);

在另一连接文件中引用const 只读变量：

extern const int i; //正确的声明

extern const int j=10; //错误！只读变量的值不能改变。

注意这里是声明不是定义，关于声明和定义的区别，请看本章开始处。

1.12 volatile

volatile关键字和const一样是一种类型修饰符，用它修饰的变量表示可以被某些编译器未知的因素更改，比如操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

1.编译器优化

int i=10;

int j = i；//(1)语句

int k = i；//(2)语句

这时候编译器对代码进行优化，因为在（1）、（2）两条语句中，i 没有被用作左值。这时候编译器认为i 的值没有发生改变，所以在（1）语句时从内存中取出i 的值赋给j 之后，这个值并没有被丢掉，而是在（2）语句时继续用这个值给k 赋值。编译器不会生成出汇编代码重新从内存里取i 的值，这样提高了效率。但要注意：（1）、（2）语句之间i 没有被用作左值才行。

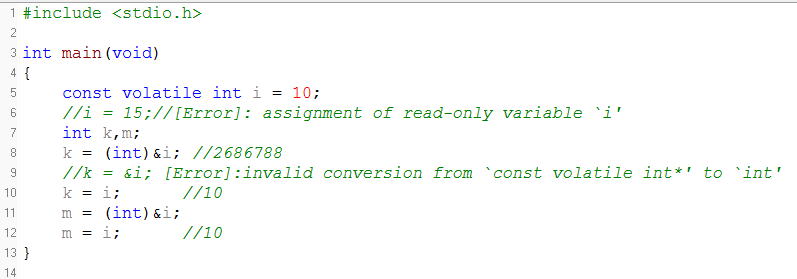
2.编译器没有优化

volatile int i=10;

int j = i；//(3)语句

int k = i；//(4)语句

volatile 关键字告诉编译器i是随时可能发生变化的，每次使用它的时候必须从内存中取出i的值，因而编译器生成的汇编代码会重新从i的地址处读取数据放在k中。这样看来，如果i是一个寄存器变量或者表示一个端口数据或者是多个线程的共享数据，就容易出错，所以说volatile可以保证对特殊地址的稳定访问。



1.13 extern

1.14 struct

1.可以压缩函数传入的参数个数，以及扩展函数的返回个数。

2.空结构体的大小是1byte。

int main(void)

{

struct student

{

}stu;

printf("%d\n",sizeof(stu)); //1

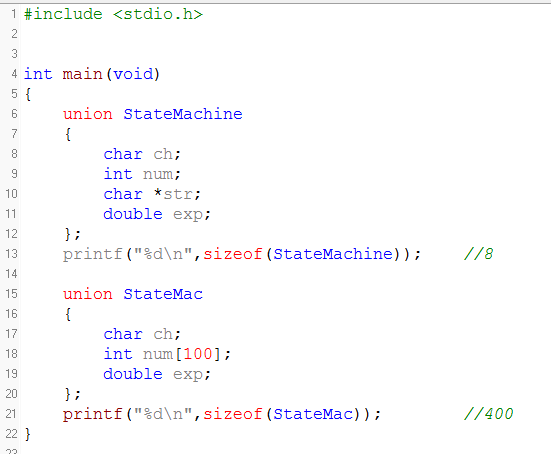
}

编译器认为任何一种数据类型都有其大小，用它来定义一个变量能够分配确定大小的空间。既然如此，编译器就理所当然的认为任何一个结构体都是有大小的，哪怕这个结构体为空。那万一结构体真的为空，它的大小为什么值比较合适呢？假设结构体内只有一个char 型的数据成员，那其大小为1byte（这里先不考虑内存对齐的情况）.也就是说非空结构体类型数据最少需要占一个字节的空间，而空结构体类型数据总不能比最小的非空结构体类型数据所占的空间大吧。这就麻烦了，空结构体的大小既不能为0，也不能大于1，怎么办？定义为0.5个byte？但是内存地址的最小单位是1 个byte，0.5 个byte 怎么处理？解决这个问题的最好办法就是折中，编译器理所当然的认为你构造一个结构体数据类型是用来打包一些数据成员的，而最小的数据成员需要1 个byte，编译器为每个结构体类型数据至少预留1 个byte的空间。所以，空结构体的大小就定位1 个byte。

1.15 union

1. union 维护足够的空间来置放多个数据成员中的“一种”，而不是为每一个数据成员配置

空间，在union 中所有的数据成员共用一个空间，同一时间只能储存其中一个数据成员，所有的数据成员具有相同的起始地址。如果一些数据不可能在同一时间同时被用到，则可以使用union。一个union 只配置一个足够大的空间以来容纳最大长度的数据成员，以下例而言，最大长度是double型态，所以StateMachine 的空间大小就是double数据类型的大小。

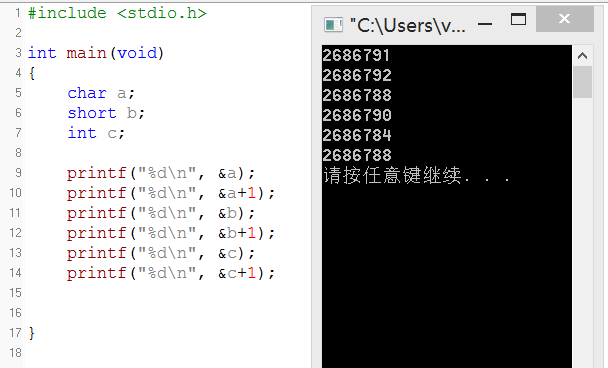


2.大小端模式对union的影响

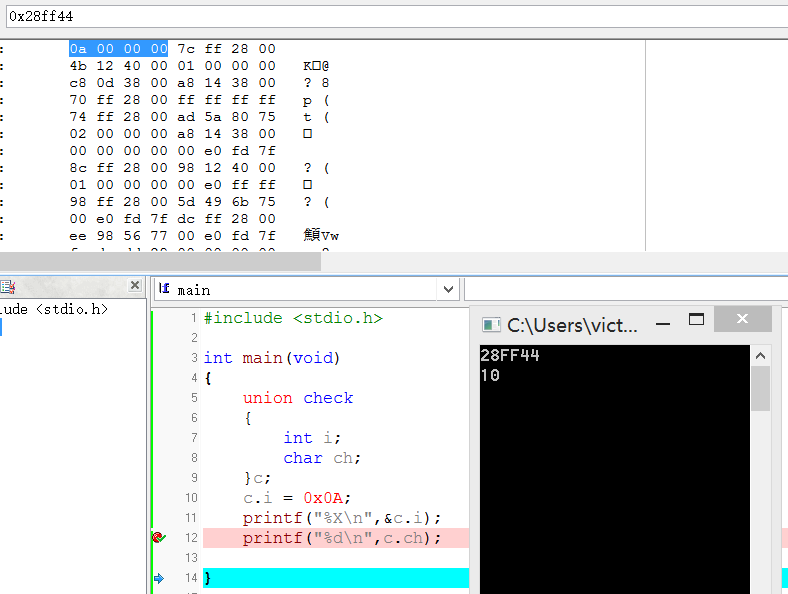
大端模式（Big\_endian）：字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中。

小端模式（Little\_endian）：字数据的高字节存储在高地址中，而字数据的低字节则存放在低地址中。

union 型数据所占的空间等于其最大的成员所占的空间。对union 型的成员的存取都是相对于该联合体基地址的偏移量为0 处开始，也就是联合体的访问不论对哪个变量的存取都是从union 的首地址位置开始。



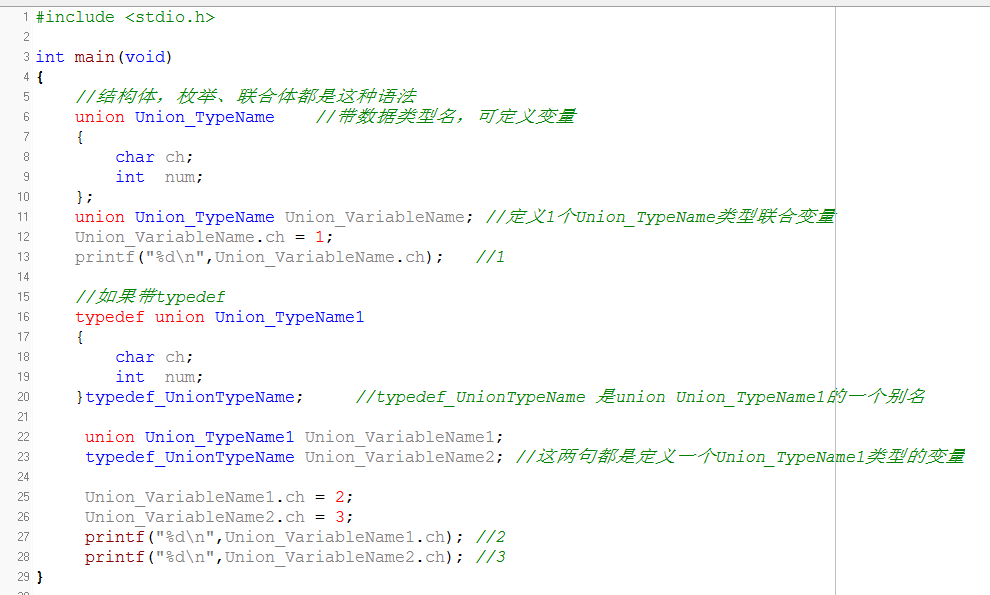
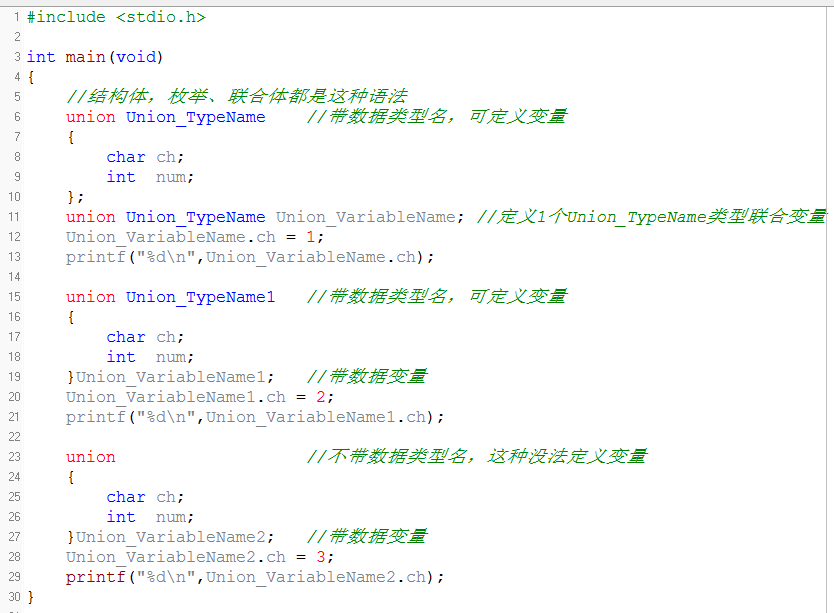
3.确定系统的存储模式



从内存地址可以看到，Ox0A存放在地址的低地址，所以是小端模式存储。



4.注意区分数据类型名和类型的变量。



1.16 enum

1.枚举类型特点

enum Enum\_TypeName //带数据类型名

{

ENUM\_CONST\_1,

ENUM\_CONST\_2,

ENUM\_CONST\_3

}Enum\_VaribalName; //带变量类型名

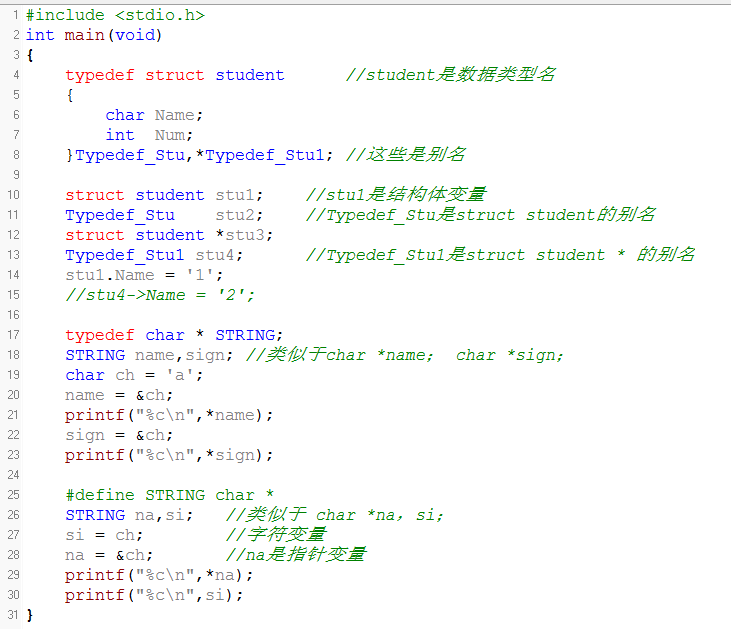
Enum\_TypeName 是自定义的一种数据数据类型名，而Enum\_VaribalName为Enum\_TypeName类型的一个变量，也就是我们平时常说的枚举变量。实际上Enum\_TypeName类型是对一个变量取值范围的限定，而花括号内是它的取值范围，即Enum\_TypeName类型的变量Enum\_VaribalName 只能取值为花括号内的任何一个值，如果赋给该类型变量的值不在列表中，则会报错或者警告。ENUM\_CONST\_1、ENUM\_CONST\_2、...、ENUM\_CONST\_n，这些成员都是常量，也就是我们平时所说的枚举常量（常量一般用大写）。enum 变量类型还可以给其中的常量符号赋值，如果不赋值则会从被赋初值的那个常量开始依次加1，如果都没有赋值，它们的值从0 开始依次递增1。



1.17 typedef

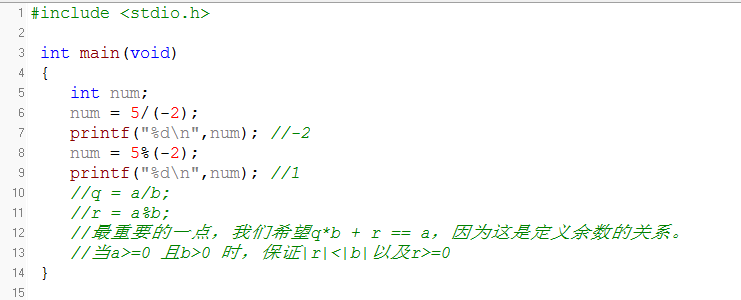
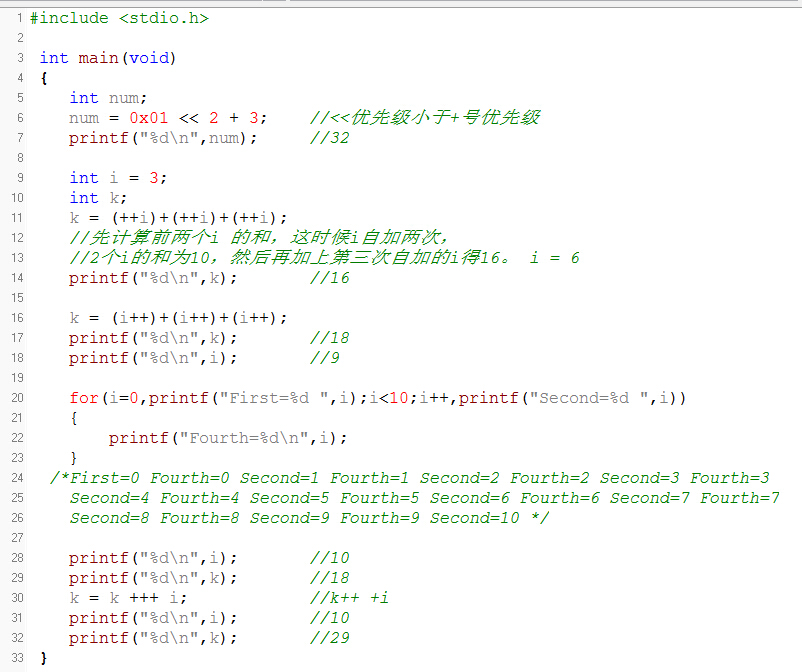
typedef 的真正意思是给一个已经存在的数据类型（注意：是类型不是变量）取一个别

名，而非定义一个新的数据类型。

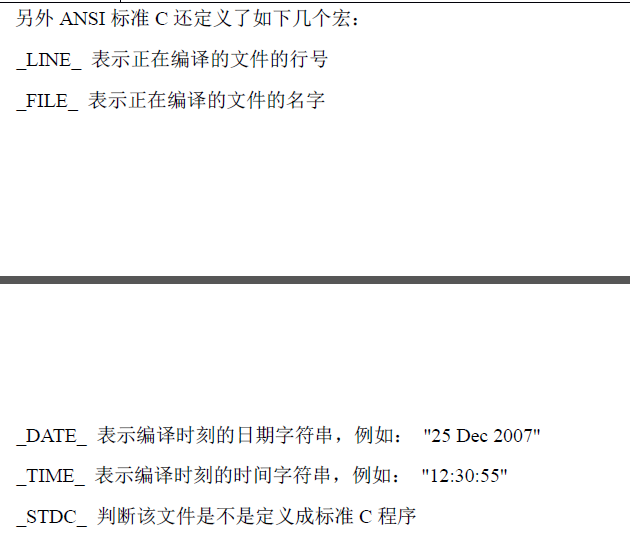
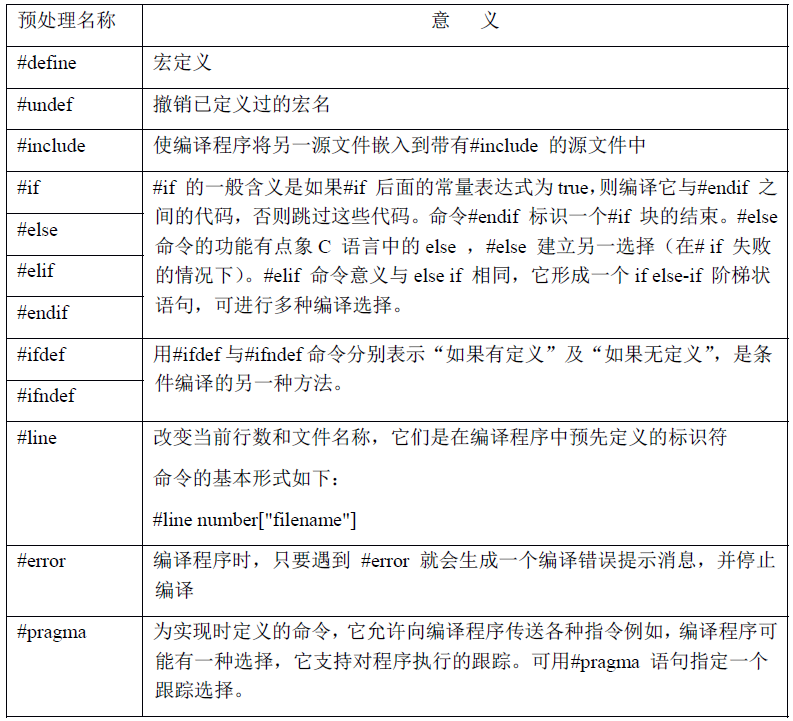


第二章 符号

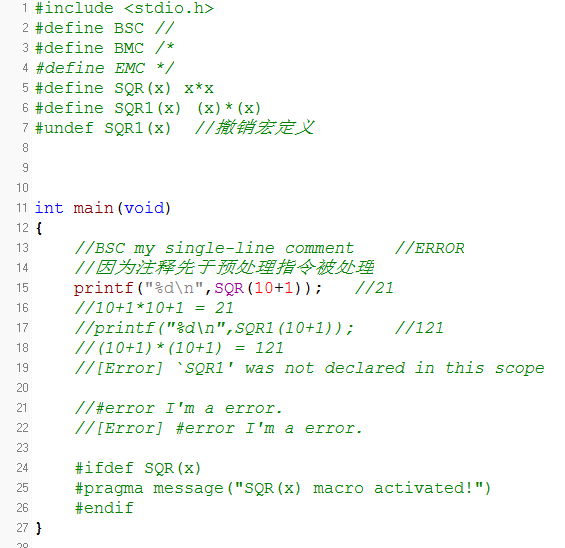
2.1



第三章 预处理



3.1 #define



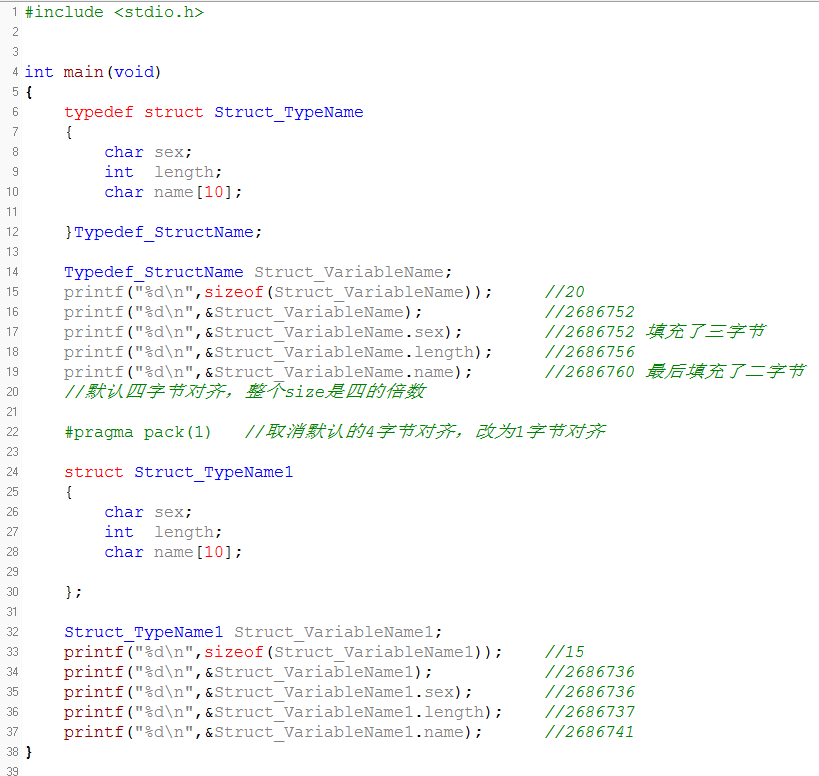
3.2 内存对齐

字，双字，和四字在自然边界上不需要在内存中对齐。（对字，双字，和四字来说，自然边界分别是偶数地址，可以被4整除的地址，和可以被8整除的地址。）无论如何，为了提高程序的性能，数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；然而，对齐的内存访问仅需要一次访问。

如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。比如在32位cpu下，假设一个整型变量的地址为0x00000004，那它就是自然对齐的。

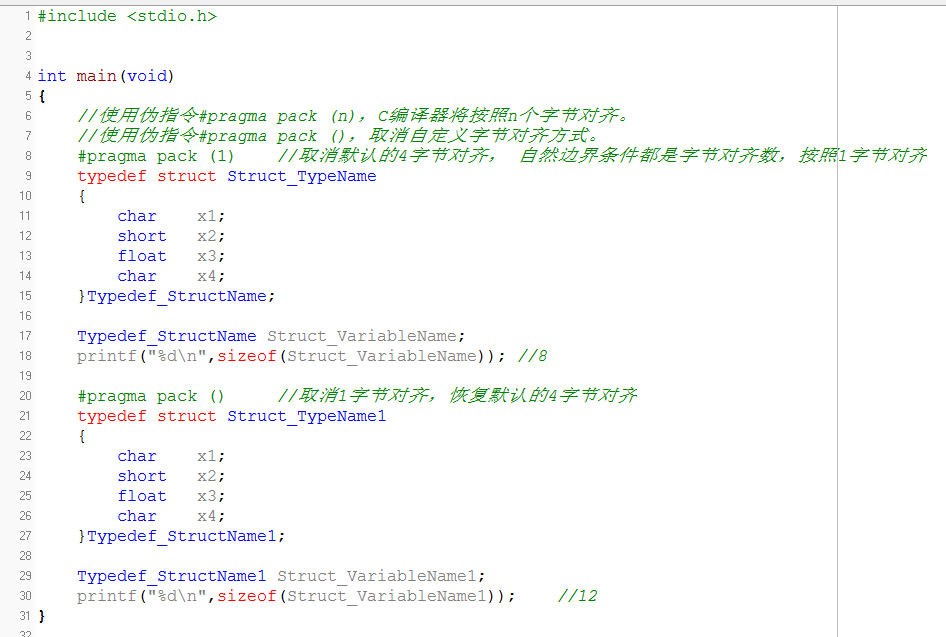
需要字节对齐的根本原因在于CPU访问数据的效率问题。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。而如果变量在自然对齐位置上，则只要一次就可以取出数据。

数组：按照基本数据类型对齐，第一个对齐了后面的自然也就对齐了。   
　　联合：按其包含的长度最大的数据类型对齐。   
　　结构体：结构体中每个数据类型都要对齐。



在设计不同CPU下的通信协议时，或者编写硬件驱动程序时寄存器的结构这两个地方都需要按一字节对齐。即使看起来本来就自然对齐的也要使其对齐，以免不同的编译器生成的代码不一样。





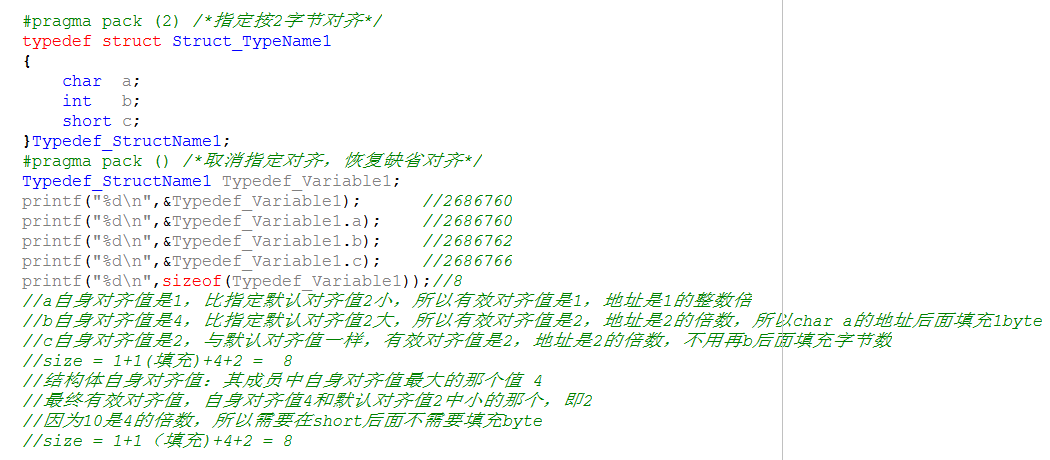
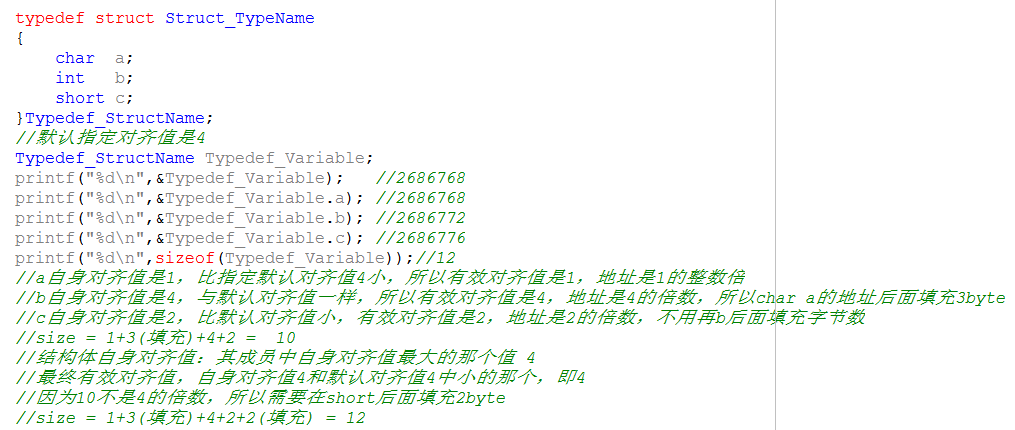
# 正式分析结构体大小

对齐规则：

1.数据类型自身的对齐值：  
对于char型数据，其自身对齐值为1，对于short型为2，对于int,float,double类型，其自身对齐值为4，单位字节。  
2.结构体或者类的自身对齐值（自然边界条件）：其成员中自身对齐值最大的那个值。  
3.指定对齐值：#pragma pack (value)时的指定对齐值value。  
4.结构体中的每个数据成员、结构体和类的最终有效对齐值：自身对齐值和指定对齐值中小的那个值。

整个结构体的大小就是有效对齐值的整数倍。

有了这些值，我们就可以很方便的来讨论具体数据结构的成员和其自身的对齐方式。有效对齐值N是最终用来决定数据存放地址方式的值，最重要。有效对齐N，就是表示“对齐在N上”，也就是说该数据的"存放起始地址%N=0".而数据结构中的数据变量都是按定义的先后顺序来排放的。第一个数据变量的起始地址就是数据结构的起始地址。结构体的成员变量要对齐排放，结构体本身也要根据自身的有效对齐值圆整(就是结构体成员变量占用总长度需要是对结构体有效对齐值的整数倍，结合下面例子理解)。



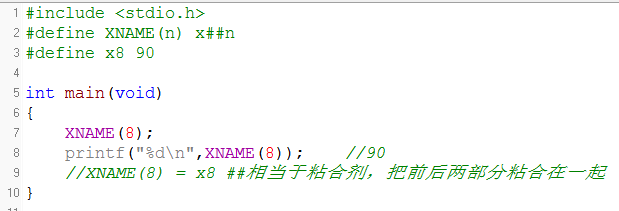
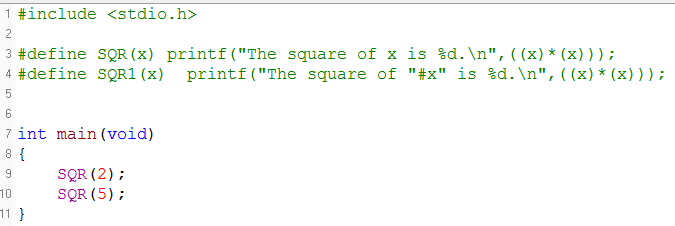
   如果在编程的时候要考虑节约空间的话,那么我们只需要假定结构的首地址是0,然后各个变量按照上面的原则进行排列即可,基本的原则就是把结构中的变量按照类型大小从小到大声明,尽量减少中间的填补空间.还有一种就是为了以空间换取时间的效率,我们显示的进行填补空间进行对齐,比如:有一种使用空间换时间做法是显式的插入reserved成员：  
             structA{  
              char a;  
              char reserved[3];//使用空间换时间  
              int b;  
}

reserved成员对我们的程序没有什么意义,它只是起到填补空间以达到字节对齐的目的,当然即使不加这个成员通常编译器也会给我们自动填补对齐,我们自己加上它只是起到显式的提醒作用.

代码中关于对齐的隐患，很多是隐式的。比如在强制类型转换的时候。例如：  
unsigned int i = 0x12345678;  
unsigned char \*p=NULL;  
unsigned short \*p1=NULL;

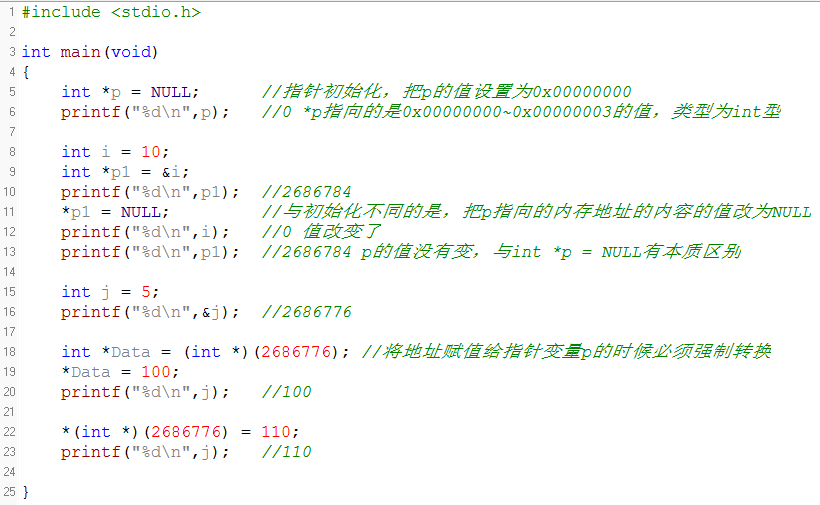
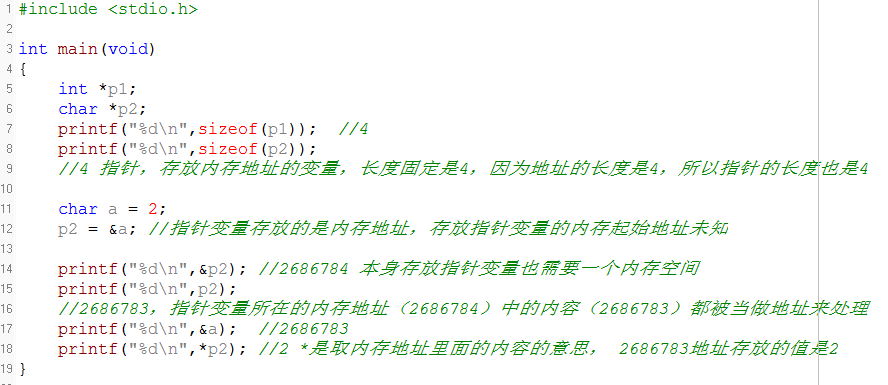
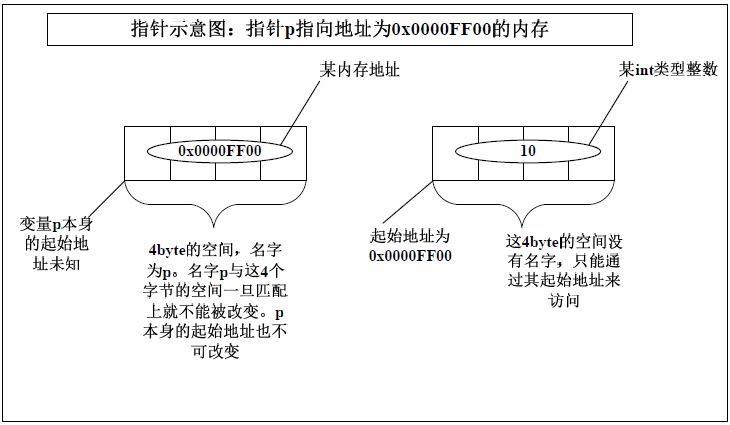
p=&i;  
\*p=0x00;  
p1=(unsigned short \*)(p+1);  
\*p1=0x0000;  
最后两句代码，从奇数边界去访问unsignedshort型变量，显然不符合对齐的规定。  
在x86上，类似的操作只会影响效率，但是在MIPS或者sparc上，可能就是一个error,因为它们要求必须字节对齐.

3.2 #运算符



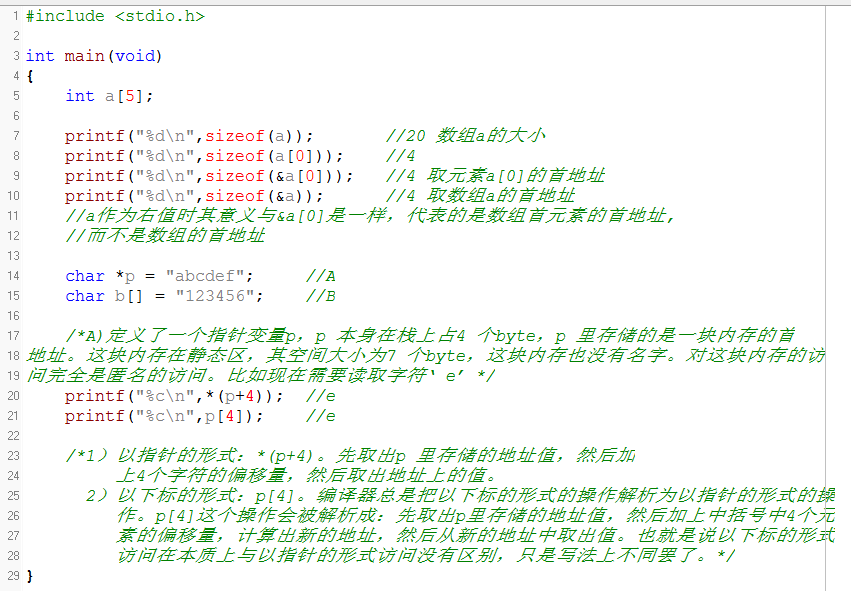
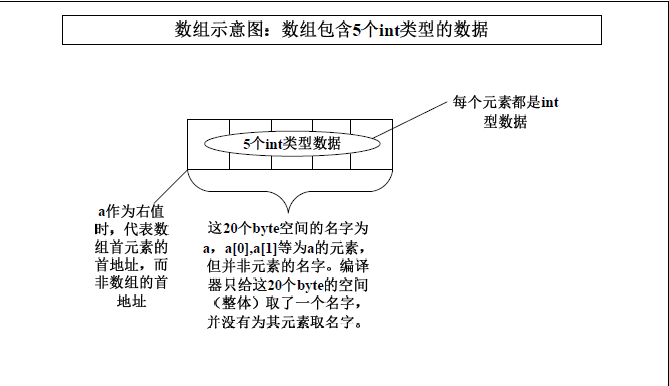
第四章 指针和数组

4.1 指针的内存布局

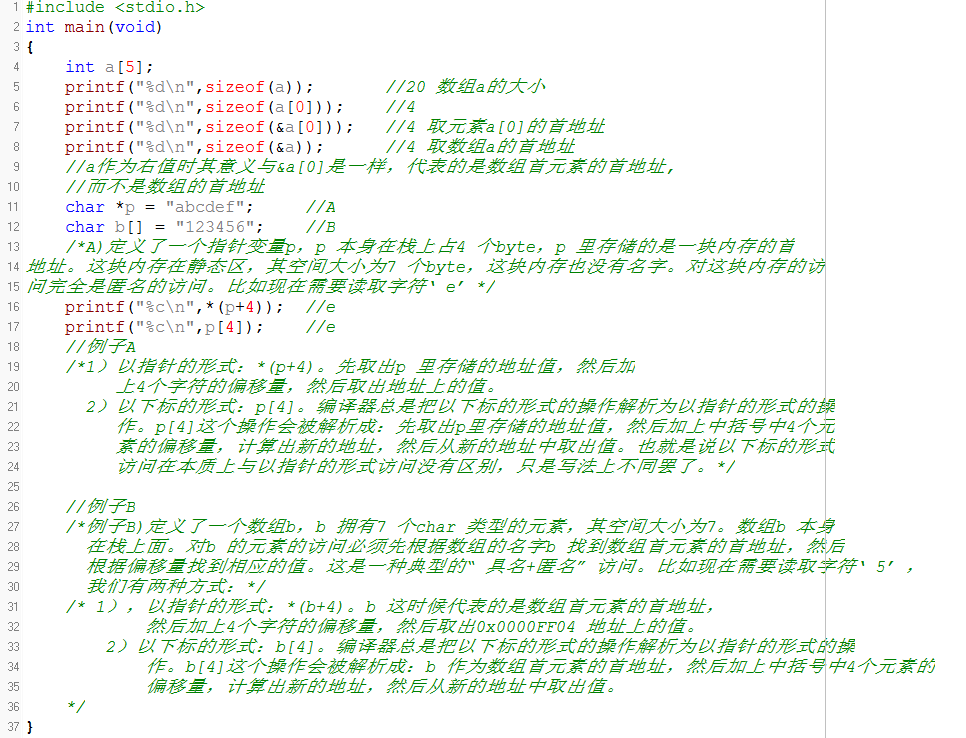


4.2 数组

Int a[5]

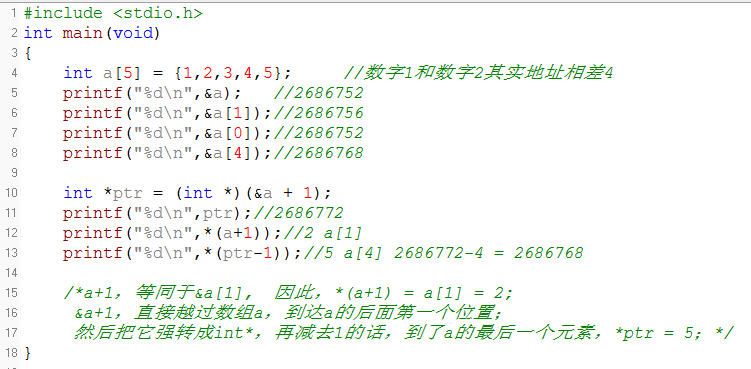


4.3 指针与数组的恩恩怨怨



由上面的分析，我们可以看到，指针和数组根本就是两个完全不一样的东西。只是它们都可以“以指针形式”或“以下标形式”进行访问。一个是完全的匿名访问，一个是典型的具名+匿名访问。一定要注意的是这个“以XXX 的形式的访问”这种表达方式。

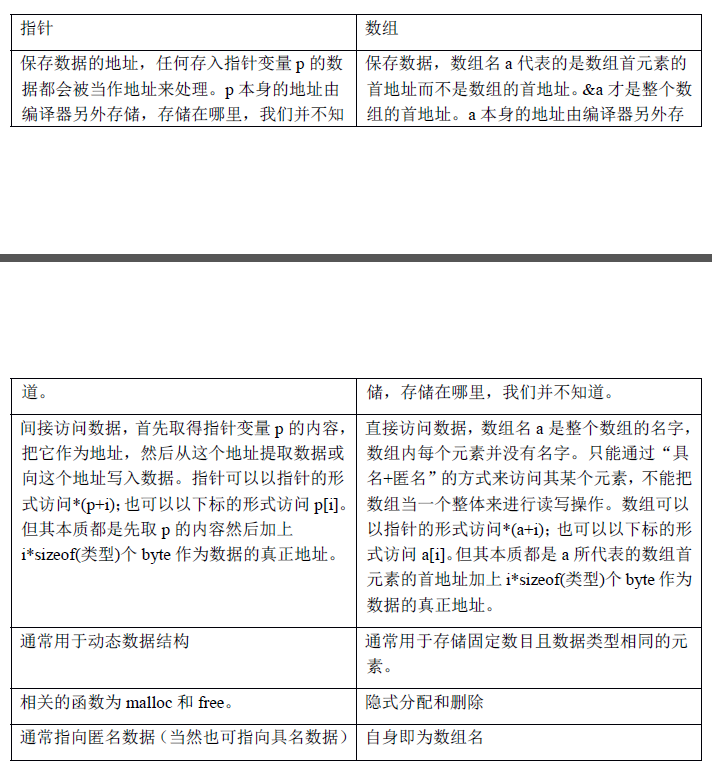
另外一个需要强调的是：上面所说的偏移量4 代表的是4 个元素，而不是4 个byte。只不过这里刚好是char 类型数据1 个字符的大小就为1 个byte。记住这个偏移量的单位是元素的个数而不是byte 数，在计算新地址时千万别弄错了。



对指针进行加**1** 操作，得到的是下一个元素的地址，而不是原有地址值直接加1。所以，一个类型为**T** 的指针的移动，以**sizeof(T)** 为移动单位。因此，对上题来说，a 是一个一维数组，数组中有5 个元素； ptr 是一个int 型的指针。&a + 1: 取数组a 的首地址，该地址的值加上sizeof(a) 的值，即&a + 5\*sizeof(int)，也就是下一个数组的首地址，显然当前指针已经越过了数组的界限。

(int \*)(&a+1): 则是把上一步计算出来的地址，强制转换为int \* 类型，赋值给ptr。\*(a+1): a,&a 的值是一样的，但意思不一样，a 是数组首元素的首地址，也就是a[0]的首地址，&a 是数组的首地址，a+1 是数组下一元素的首地址，即a[1]的首地址,&a+1 是下一个数组的首地址。所以输出2\*(ptr-1): 因为ptr 是指向a[5]，并且ptr 是int \* 类型，所以\*(ptr-1) 是指向a[4] ，输出5。

以后一定要确认你的代码在一个地方定义为指针，在别的地方也只能声明为指针；在一个的地方定义为数组，在别的地方也只能声明为数组。



4.4 指针数组和数组指针

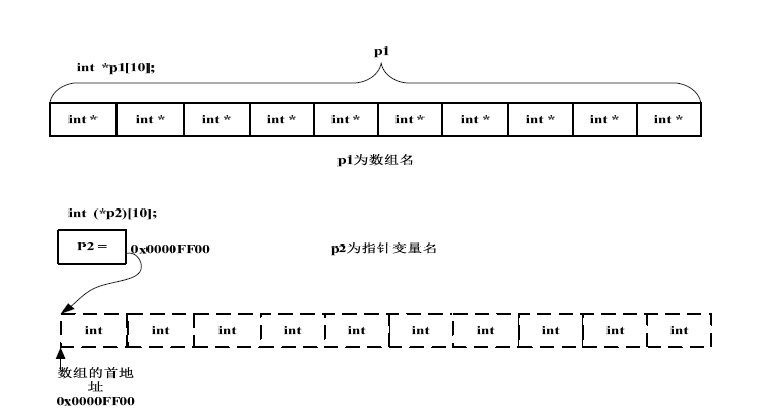
指针数组：首先它是一个数组，数组的元素都是指针，数组占多少个字节由数组本身决定。它是“储存指针的数组”的简称。

数组指针：首先它是一个指针，它指向一个数组。在32 位系统下永远是占4 个字节，至于它指向的数组占多少字节，不知道。它是“指向数组的指针”的简称。

1) int \*p1[10]; —>类似于 (int \*) p[10] 这个数组是一个存储了10个int类型的指针变量的数组

2) int(\*p2)[10]; —>类似于 int (\*p2) [10] (\*p2)是一个指针变量,指向一个int [10]类型的数组,其实更可以看成 int (\*)[10] p2 因为通常是数据类型名加上指针变量

这里需要明白一个符号之间的优先级问题。“[]”的优先级比“\*”要高。p1 先与“[]”结合，构成一个数组的定义，数组名为p1，int \*修饰的是数组的内容，即数组的每个元素。那现在我们清楚，这是一个数组，其包含10 个指向int 类型数据的指针，即指针数组。至于p2 就更好理解了，在这里“（）”的优先级比“[]”高，“\*”号和p2 构成一个指针的定义，指针变量名为p2，int 修饰的是数组的内容，即数组的每个元素。数组在这里并没有名字，是个匿名数组。那现在我们清楚p2 是一个指针，它指向一个包含10 个int 类型数据的数组，即数组指针。

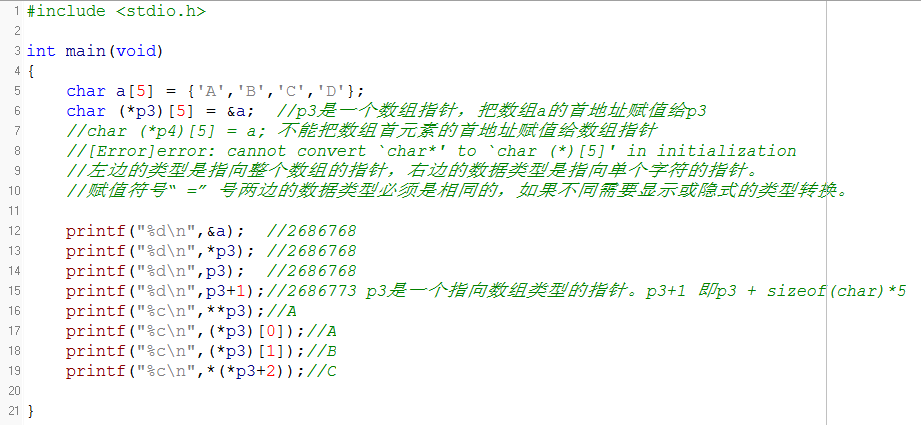


需要注意的是

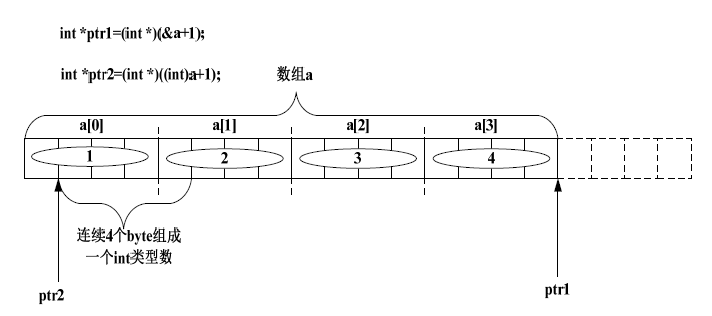
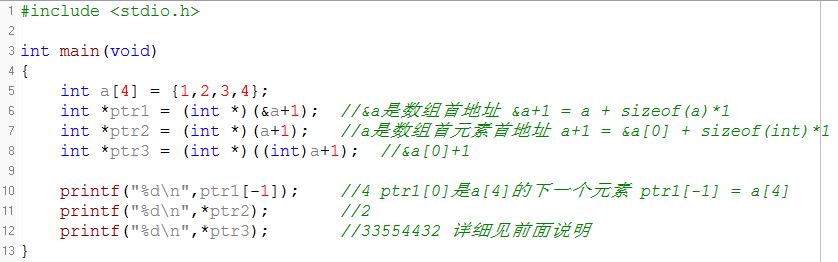
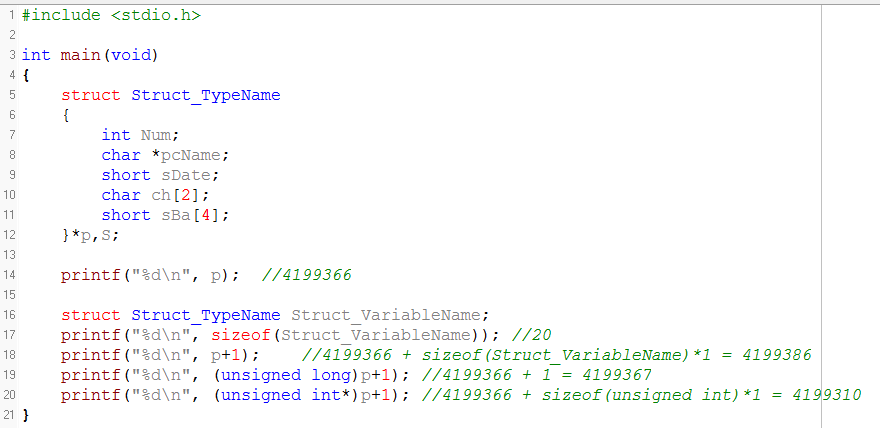
int a[10]

a -> 数组名a代表的是数组元素的首地址而不是数组的首地址

&a -> 整个数组的首地址

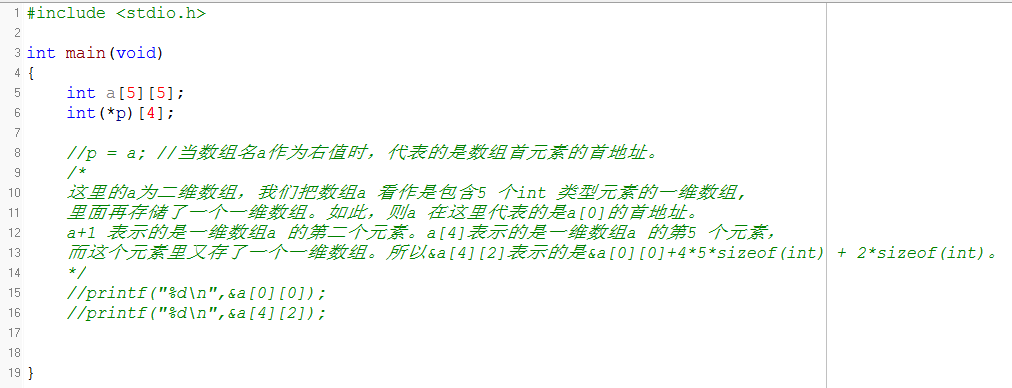
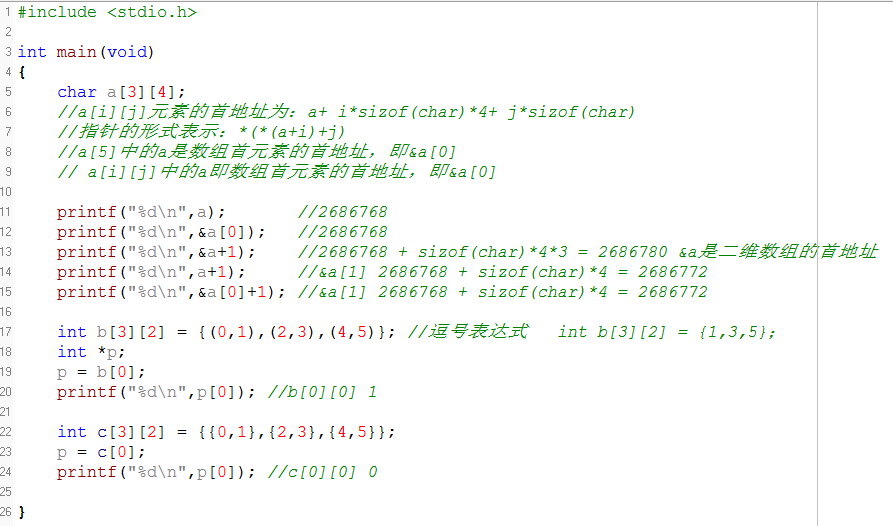


地址的强制转换

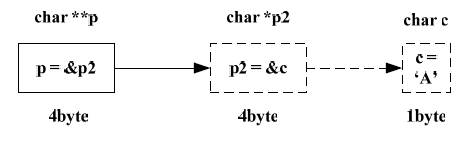


图中的ptr2在程序里是ptr3

4.5 多维数组和多级指针



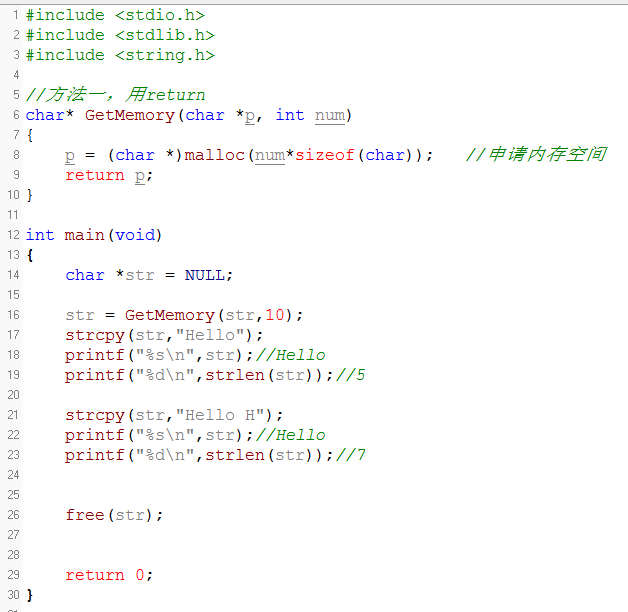
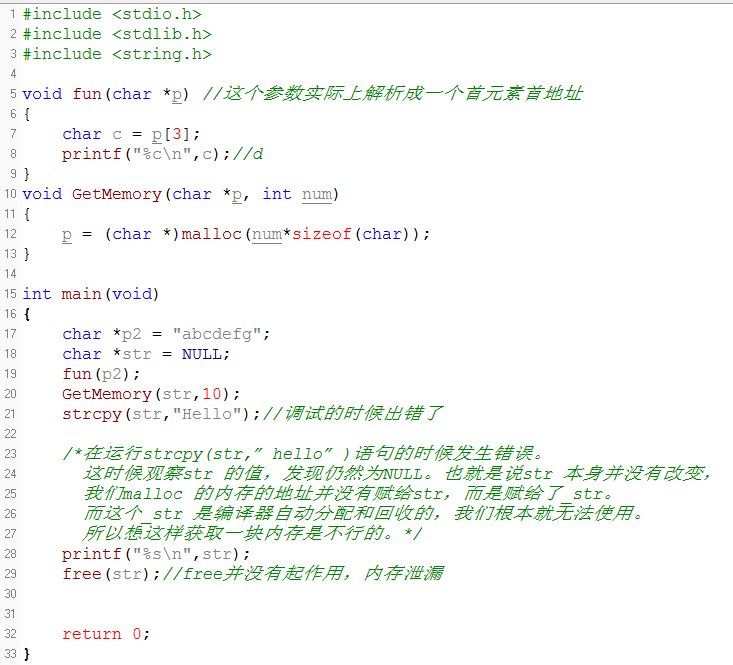
二级指针



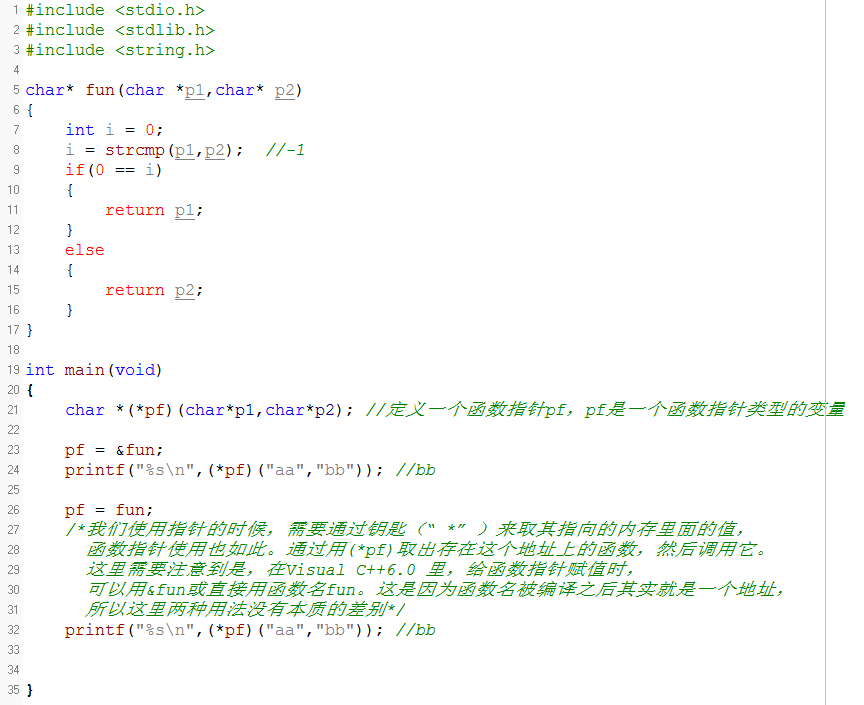
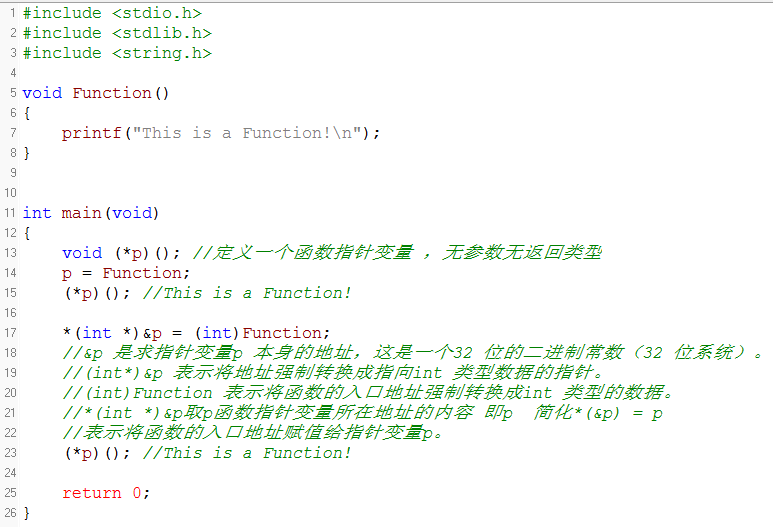
4.6 数组参数和指针参数

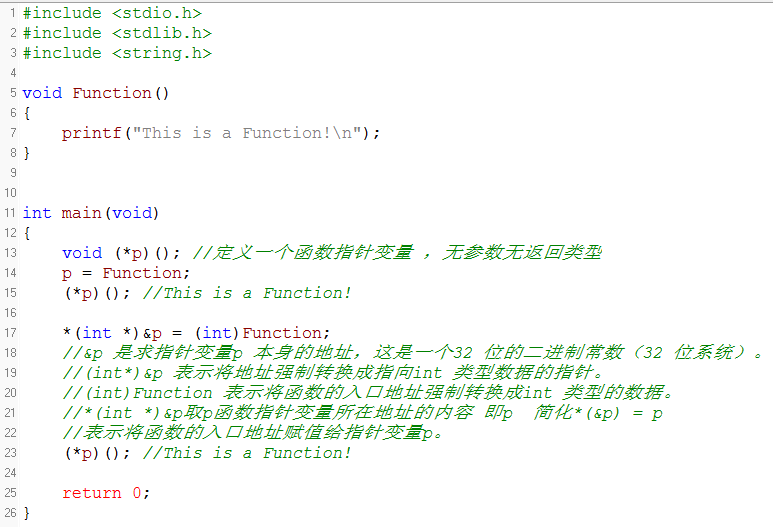
**C**语言中，当一维数组作为函数参数的时候，编译器总是把它解析成一个指向其首元素首地址的指针。

函数本身是没有类型的，只有函数的返回值才有类型。

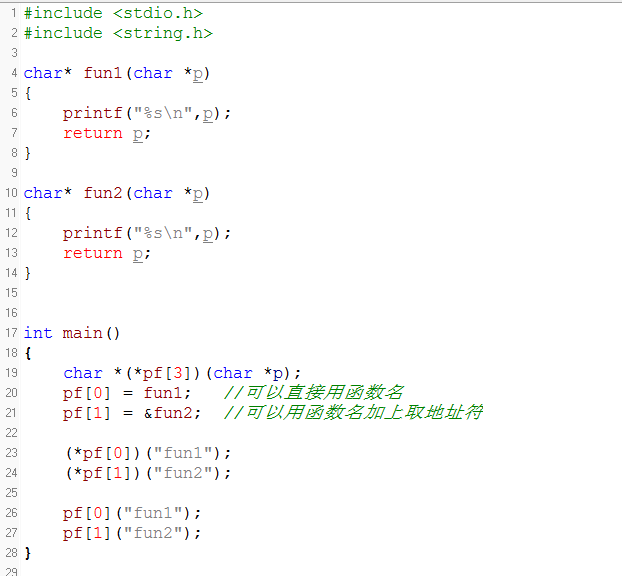


4.7 函数指针





函数指针数组



第五章 内存管理

5.1 野指针

定义指针变量的同时最好初始化为NULL，用完指针之后也将指针变量的值设置为NULL。也就是说除了在使用时，别的时间都把指针“栓到0 地址处。这样它就老实了。

5.2 栈、堆和静态区

静态区：保存自动全局变量和static 变量（包括static 全局和局部变量）。静态区的内容

在总个程序的生命周期内都存在，由编译器在编译的时候分配。

栈：保存局部变量。栈上的内容只在函数的范围内存在，当函数运行结束，这些内容

也会自动被销毁。其特点是效率高，但空间大小有限。

堆：由malloc 系列函数或new 操作符分配的内存。其生命周期由free 或delete 决定。

在没有释放之前一直存在，直到程序结束。其特点是使用灵活，空间比较大，但容易出错。

5.3 常见的内存错误及对策



函数入口校验

不管什么时候，我们使用指针之前一定要确保指针是有效的。

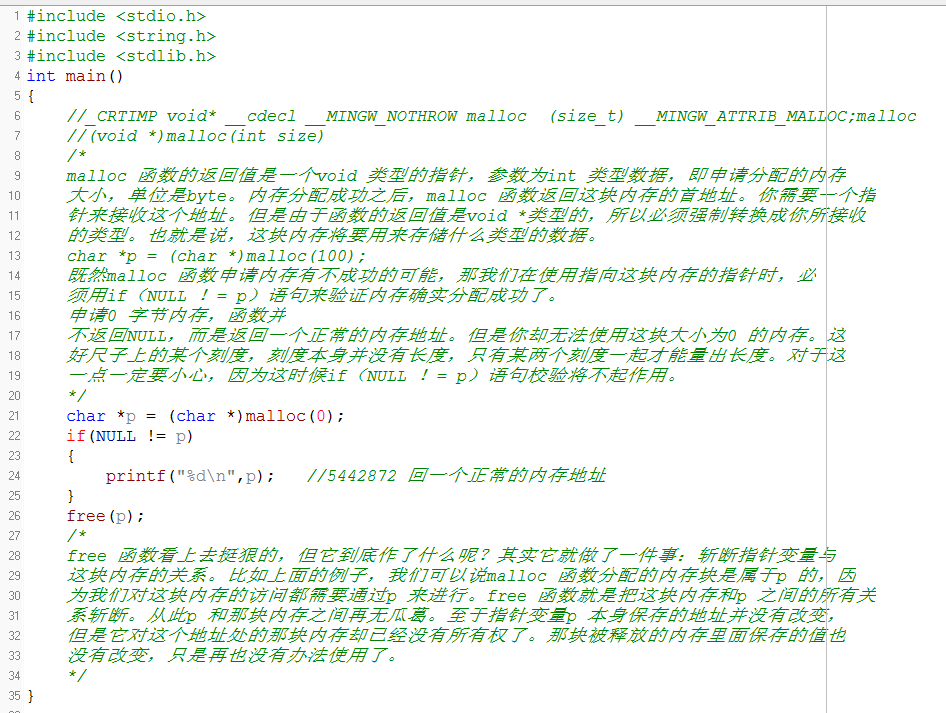
一般在函数入口处使用assert(NULL != p)对参数进行校验。在非参数的地方使用if（NULL != p）来校验。但这都有一个要求，即p 在定义的同时被初始化为NULL 了。比如上面的例子，即使用if（NULL != p）校验也起不了作用，因为name 指针并没有被初始化为NULL，其内部是一个非NULL 的乱码。

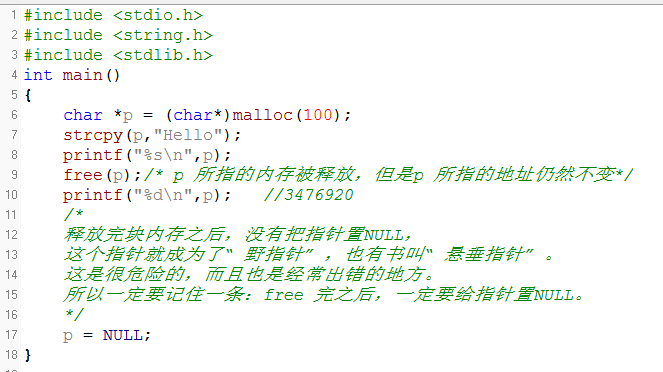
assert 是一个宏，而不是函数，包含在assert.h 头文件中。如果其后面括号里的值为假，则程序终止运行，并提示出错；如果后面括号里的值为真，则继续运行后面的代码。这个宏只在Debug 版本上起作用，而在Release 版本被编译器完全优化掉，这样就不会影响代码的性能。

有人也许会问，既然在Release 版本被编译器完全优化掉，那Release 版本是不是就完

全没有这个参数入口校验了呢？这样的话那不就跟不使用它效果一样吗？是的，使用assert 宏的地方在Release 版本里面确实没有了这些校验。但是我们要知道，

assert 宏只是帮助我们调试代码用的，它的一切作用就是让我们尽可能的在调试函数的时候把错误排除掉，而不是等到Release 之后。它本身并没有除错功能。再有一点就是，参数出现错误并非本函数有问题，而是调用者传过来的实参有问题。assert 宏可以帮助我们定位错误，而不是排除错误。





第一种：就是上面所说的，free（p）之后，继续通过p 指针来访问内存。解决的办法就是给p 置NULL。

第二种：函数返回栈内存。这是初学者最容易犯的错误。比如在函数内部定义了一个数组，却用return 语句返回指向该数组的指针。解决的办法就是弄明白栈上变量的生命周期。

第三种：内存使用太复杂，弄不清到底哪块内存被释放，哪块没有被释放。解决的办法是重新设计程序，改善对象之间的调用关系。