**[sizeof()用法汇总](http://www.cnblogs.com/chengxin1982/archive/2009/01/13/1374575.html)**

sizeof()功能：计算数据空间的字节数  
1.与strlen()比较  
      strlen()计算字符数组的字符数，以"\0"为结束判断，不计算为'\0'的数组元素。  
      而sizeof计算数据（包括数组、变量、类型、结构体等）所占内存空间，用字节数表示。  
2.指针与静态数组的sizeof操作  
      指针均可看为变量类型的一种。所有指针变量的sizeof 操作结果均为4。  
**注意**：int \*p; sizeof(p)=4;  
                  但sizeof(\*p)相当于sizeof(int);        
      对于静态数组，sizeof可直接计算数组大小；  
      例：int a[10];char b[]="hello";  
              sizeof(a)等于4\*10=40;  
              sizeof(b)等于6;  
 **注意**：数组做型参时，数组名称当作指针使用！！  
               void  fun(char p[])  
               {sizeof(p)等于4}

**经典问题：**  
      double\* (\*a)[3][6];   
      cout<<sizeof(a)<<endl; // 4 a为指针  
      cout<<sizeof(\*a)<<endl; // 72 \*a为一个有3\*6个指针元素的数组  
      cout<<sizeof(\*\*a)<<endl; // 24 \*\*a为数组一维的6个指针  
      cout<<sizeof(\*\*\*a)<<endl; // 4 \*\*\*a为一维的第一个指针  
      cout<<sizeof(\*\*\*\*a)<<endl; // 8 \*\*\*\*a为一个double变量

**问题解析：**a是一个很奇怪的定义，他表示一个指向double\*[3][6]类型数组的指针。既然是指针，所以sizeof(a)就是4。   
      既然a是执行double\*[3][6]类型的指针，\*a就表示一个double\*[3][6]的多维数组类型，因此sizeof(\*a)=3\*6\*sizeof(double\*)=72。同样的，\*\*a表示一个double\*[6]类型的数组，所以sizeof(\*\*a)=6\*sizeof  (double\*)=24。\*\*\*a就表示其中的一个元素，也就是double\*了，所以sizeof(\*\*\*a)=4。至于\*\*\*\*a，就是一个double了，所以sizeof(\*\*\*\*a)=sizeof(double)=8。   
3.格式的写法  
   sizeof操作符，对变量或对象可以不加括号，但若是类型，须加括号。  
4.使用sizeof时string的注意事项  
   string s="hello";  
   sizeof(s)等于string类的大小，sizeof(s.c\_str())得到的是与字符串长度。  
5.union 与struct的空间计算  
   总体上遵循两个原则：  
   (1)整体空间是 占用空间最大的成员（的类型）所占字节数的整倍数  
   (2)数据对齐原则----内存按结构成员的先后顺序排列，当排到该成员变量时，其前面已摆放的空间大小必须是该成员类型大小的整倍数，如果不够则补齐，以此向后类推。。。。。  
   注意：数组按照单个变量一个一个的摆放，而不是看成整体。如果成员中有自定义的类、结构体，也要注意数组问题。  
**例：[**引用其他帖子的内容]  
因为对齐问题使结构体的sizeof变得比较复杂，看下面的例子：(默认对齐方式下)  
struct s1  
{  
char a;  
double b;  
int c;  
char d;   
};  
  
struct s2  
{  
char a;  
char b;  
int c;  
double d;  
};  
  
cout<<sizeof(s1)<<endl; // 24  
cout<<sizeof(s2)<<endl; // 16  
  
  同样是两个char类型，一个int类型，一个double类型，但是因为对齐问题，导致他们的大小不同。计算结构体大小可以采用元素摆放法，我举例子说明一下：首先，CPU判断结构体的对界，根据上一节的结论，s1和s2的对界都取最大的元素类型，也就是double类型的对界8。然后开始摆放每个元素。  
  对于s1，首先把a放到8的对界，假定是0，此时下一个空闲的地址是1，但是下一个元素d是double类型，要放到8的对界上，离1最接近的地址是8了，所以d被放在了8，此时下一个空闲地址变成了16，下一个元素c的对界是4，16可以满足，所以c放在了16，此时下一个空闲地址变成了20，下一个元素d需要对界1，也正好落在对界上，所以d放在了20，结构体在地址21处结束。由于s1的大小需要是8的倍数，所以21-23的空间被保留，s1的大小变成了24。  
  对于s2，首先把a放到8的对界，假定是0，此时下一个空闲地址是1，下一个元素的对界也是1，所以b摆放在1，下一个空闲地址变成了2；下一个元素c的对界是4，所以取离2最近的地址4摆放c，下一个空闲地址变成了8，下一个元素d的对界是8，所以d摆放在8，所有元素摆放完毕，结构体在15处结束，占用总空间为16，正好是8的倍数。  
  
  这里有个陷阱，对于结构体中的结构体成员，不要认为它的对齐方式就是他的大小，看下面的例子：  
struct s1  
{  
char a[8];  
};  
  
struct s2  
{  
double d;  
};  
  
struct s3  
{  
s1 s;  
char a;  
};  
  
struct s4  
{  
s2 s;  
char a;   
};  
cout<<sizeof(s1)<<endl; // 8  
cout<<sizeof(s2)<<endl; // 8  
cout<<sizeof(s3)<<endl; // 9  
cout<<sizeof(s4)<<endl; // 16;  
  s1和s2大小虽然都是8，但是s1的对齐方式是1，s2是8（double），所以在s3和s4中才有这样的差异。  
  所以，在自己定义结构体的时候，如果空间紧张的话，最好考虑对齐因素来排列结构体里的元素。

**补充：**不要让double干扰你的位域   
　　在结构体和类中，可以使用位域来规定某个成员所能占用的空间，所以使用位域能在一定程度上节省结构体占用的空间。不过考虑下面的代码：   
  
struct s1   
{   
　int i: 8;   
　int j: 4;   
　double b;   
　int a:3;   
};   
  
struct s2   
{   
　int i;   
　int j;   
　double b;   
　int a;   
};   
  
struct s3   
{   
　int i;   
　int j;   
　int a;   
　double b;   
};   
  
struct s4   
{   
　int i: 8;   
　int j: 4;   
　int a:3;   
　double b;   
};   
  
cout<<sizeof(s1)<<endl; // 24   
cout<<sizeof(s2)<<endl; // 24   
cout<<sizeof(s3)<<endl; // 24   
cout<<sizeof(s4)<<endl; // 16   
　　可以看到，有double存在会干涉到位域（sizeof的算法参考上一节），所以使用位域的的时候，最好把float类型和double类型放在程序的开始或者最后。

**相关常数：**

sizeof int:4  
sizeof short:2  
sizeof long:4  
sizeof float:4  
sizeof double:8  
sizeof char:1  
sizeof p:4  
sizeof WORD:2  
sizeof DWORD:4