关于sizeof

sizeof以字节形式给出操作数的存储大小。通过本文我们可以了解sizeof的使用和计算方法。

主要内容：

一、功能

二、使用语法

三、计算方法

四、举例

五、其他

一、功能

===============

sizeof以字节形式给出操作数的存储大小。

sizeof是C语言的一种单目操作符，如C语言的其他操作符++、--等。它并不是函数。sizeof操作符以字节形式给出了其操作数的存储大小。操作数可以是一个表达式或括在括号内的类型名。操作数的存储大小由操作数的类型决定。

二、使用语法

===============

sizeof有如下两种使用方式：

用于数据类型的sizeof使用形式："sizeof（type）"。这里，数据类型必须用括号括住。如sizeof（int）。

用于变量的sizeof使用形式："sizeof（var\_name）"或"sizeof　var\_name"。变量名可以不用括号括住。如"sizeof　(var\_name)"，"sizeof　var\_name"等都是正确形式。带括号的用法更普遍，大多数程序员采用这种形式。

注意：sizeof操作符不能用于函数类型，不完全类型或位字段。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、void类型等。如"sizeof(max)"而此时变量max定义为int　max(), sizeof(char\_v)而此时char\_v定义为char　char\_v　[MAX]且MAX未知，sizeof(void)都不是正确形式。

三、计算方法

===============

1,基本数据类型大小

sizeof操作符的结果类型是size\_t，它在头文件中typedef为unsigned　int类型。该类型保证能容纳实现所建立的最大对象的字节大小。对于int, unsigned int, short int, unsigned short, long int, unsigned long, float, double, long　double等基本类型的大小，没有明确的规定，但是一般应当记住如下原则：

a ANSI C规定char类型一定是8位。

b long类型的长度和cpu字长一样。

c int长度没有规定，但是不比short短不比long长，并且linux上支持的所有体系中int长度目前都是32位。

d short和int类似，目前linux上长度都是16位。

例如，一般常见的基本类型大小信息如下（32位机Linux平台）

char为1byte;

int为4byte;

unsigned int为4byte；

short int为2byte；

unsigned short为2byte；

long int为4byte；

unsigned long为4byte；

float为4byte；

double为8byte；

long double为12byte；

2,指针类型大小

当操作数是指针时，sizeof依赖于编译器。例如Microsoft　C/C++7.0中，near类指针字节数为2，far、huge类指针字节数为4。一般Unix的指针字节数为4。

3,数组类型大小

当操作数具数组类型时，其结果是数组的总字节数。如果操作数是函数中的数组形参或函数类型的形参，sizeof给出其指针的大小。

4,结构和联合

联合类型操作数的sizeof是其最大字节成员的字节数。结构类型操作数的sizeof是其对象包括任何填充在内的总字节数。

关于结构大小和结构变量存放地址，本文后面例子中将重点进行介绍，确定方法依照编译器所不同，大致规则如下：

a) 成员变量在结构中被声明的顺序和其在内存中的存储顺序。第一个结构成员变量相对整个结构的地址偏移为0。

b) 结构成员变量的偏移地址由编译器中设置的"最大对齐参数"，以及该结构变量的类型大小决定。整个结构的大小由最大结构成员变量类型或编译器中设置的"最大对齐参数"决定。这里的最大对齐方式可以由不同的编译器的特定关键字来设置。默认一般是cpu的字长，例如32位机器为4字节。

结构大小和地址计算方法如下：

a）结构成员变量的偏移地址计算：偏移地址为该成员类型大小和最大对齐字节大小两者较小者的整数倍。如果当前偏移位置不满足条件，那么就填充至满足，才开始存放该结构变量。尺寸计算：就是该成员类型的大小。

b）结构变量偏移地址计算：偏移地址为该结构中最大成员大小和最大对齐字节大小两者较小者的整数倍。尺寸计算：整个结构大小为结构中最大成员变量与最大对齐方式两者较小者的整数倍，如果当前所有成员大小之和小于这个整数倍就会添充直至达到要求。

总之，首先假定好最大字节对齐方式，然后确定每个成员起始地址(该成员类型大小以及最大字节对齐的较小者的倍数)，最后确定结构总大小(最大成员大小和最大字节对齐较小者的倍数)。(先每个成员，再整体，不满足则填充)

四、举例

===============

下面是整理的说明VC到底怎么样来存放结构的(可以参见后面参考资料，这里没有对其进行严格的实践)，其它编译器处理方式有所不同，但是大体都一致。

\*结构定义如下：

struct MyStruct

{

double dda1;

char dda;

int type

}；

大小计算方式如下:

为上面的结构分配空间的时候，VC根据成员变量出现的顺序和对齐方式，先为第一个成员dda1分配空间，该成员类型为double，其起始地址跟结构的起始地址相同（偏移量 0 为sizeof(double)的倍数），占用sizeof(double)=8个字节；接下来为第二个成员dda分配空间，该成员类型为char，其想对于结构的起始地址的偏移量为8(偏移量8为sizeof(char)的倍数)，该成员变量占用 sizeof(char)=1个字节；接下来为第三个成员type分配空间，该成员类型为int，相对于结构的起始地址的偏移量为9，不是sizeof (int)=4的倍数，为了满足对齐方式对偏移量的约束问题，VC自动填充3个字节（这三个字节没有放什么东西），这时下一个可以分配的地址对于结构的起始地址的偏移量为12，刚好是sizeof(int)=4的倍数，所以把type存放在偏移量为12的地方，该成员变量占用sizeof(int)=4个字节；这时整个结构的成员变量已经都分配了空间，总的占用的空间大小为：8+1+3+4=16，刚好为结构的字节边界数（即结构中占用最大空间的类型所占用的字节数sizeof(double)=8）的倍数，所以没有空缺的字节需要填充。所以整个结构的大小为：sizeof(MyStruct)=8+1+ 3+4=16，其中有3个字节是VC自动填充的，没有放任何有意义的东西。

\*交换一下上面的MyStruct的成员变量的位置，使它变成下面的情况：

struct MyStruct

{

char dda;

double dda1;

int type

}；

这时候大小计算方式如下：

为上面的结构分配空间的时候，VC根据成员变量出现的顺序和对齐方式，先为第一个成员dda分配空间，该成员类型为char，其起始地址跟结构的起始地址相同（偏移量 0 为sizeof(char)的倍数），占用sizeof(char)=1个字节；接下来为第二个成员dda1分配空间，该成员类型为double，其想对于结构的起始地址的偏移量为1，不是sizeof (double)=7的倍数，为了满足对齐方式对偏移量的约束问题，VC自动填充7个字节（这7个字节没有放什么东西），这时下一个可以分配的地址对于结构的起始地址的偏移量为8，刚好是sizeof(double)=8的倍数，所以把dda1存放在偏移量为8的地方，该成员变量占用sizeof(double)=8个字节；接下来为第三个成员type分配空间，该成员类型为int，相对于结构的起始地址的偏移量为16，刚好是sizeof(int)=4的倍数，所以把type存放在偏移量为16的地方，该成员变量占用sizeof(int)=4个字节；这时整个结构的成员变量已经都分配了空间，总的占用的空间大小为：1+7+8+4=20，不满足结构的字节边界数（即结构中占用最大空间的类型所占用的字节数sizeof(double)=8）的倍数，所以VC自动填充4个字节。综上，整个结构的大小为：sizeof(MyStruc)为1+7+8+4+4=24。其中总的有7+4=11个字节是VC自动填充的，没有放任何有意义的东西。

\*自己设置对齐方式：

我们也屏蔽掉变量默认的对齐方式，自己可以设定变量的对齐方式。VC中提供了#pragma pack(n)来设定变量以n字节对齐方式。这里n字节对齐的意思是说:(1)结构成员变量存放的起始地址的偏移量按照两种情况确定。第一、如果设置的n大于等于该成员变量所占用的字节数，那么偏移量必须满足默认的对齐方式(也就是以该成员变量的类型大小对齐)，第二、如果设置的n小于该成员变量的类型所占用的字节数，那么成员变量的偏移量为n的倍数，而不用满足默认的对齐方式。(2)结构的总大小也按下面两种情况确定：如果n大于最大的成员变量类型所占用的字节数，那么结构的总大小为占用空间最大的成员变量所占用的空间数的倍数；否则，必须为n的倍数。

下面举例说明其用法。

#pragma pack(push) //保存对齐状态

#pragma pack(4)//设定为4字节对齐

struct test

{

char m1;

double m4;

int m3;

};

#pragma pack(pop)//恢复对齐状态

以上结构的大小为16，为此分配空间的时候，首先为m1分配空间，该成员类型为char，同时sizeof(char)为1,小于我们自己设定的4字节，所以起始地址按照sizeof(char)=1对齐，和结构起始地址一样，(偏移为0为sizeof(char)的倍数)，m1占用1个字节;接着开始为m4分配空间，该成员类型为double,同时sizeof(double)为8，大于我们自己设定的4字节，所以按照我们设置的4字节对齐，这时其偏移量为1，需要补足3个字节，使其偏移量为4，满足为n=4的倍数,m4占用8个字节；接着为m3分配空间，该成员类型为int，sizeof(int)为4，和设定的4字节一样，所以按照4字节对齐，这时其偏移量为12，正好满足为4的倍数，m3占用4个字节。这时已经为所有成员变量分配了空间，对于整个结构变量，其中最大类型为double为8字节，大于我们设定的4字节，所以结构总大小采用n=4的倍数，这里目前共分配了16个字节正好是4的倍数，所以结构大小为1+3+8+4=16。同理，如果把上面的#pragma pack(4)改为#pragma pack(16)，那么我们可以得到结构的大小为24。

从上面的例子我们可以看出，调整结构成员变量的位置，或者设置对齐方式，都会影响该结构变量的存储效率。同时设置对齐方式可能会影响速度。

五、其他

===============

几个额外需要注意的地方：

1，sizeof应用在C++中的类和结构的处理情况是相同的。但有两点需要注意，第一、结构或者类中的静态成员不对结构或者类的大小产生影响，因为静态变量的存储位置与结构或者类的实例地址无关。 第二、没有成员变量的结构或类的大小为1，因为必须保证结构或类的每一个实例在内存中都有唯一的地址。

2，经过实践对于Linux如下代码输出分别如下：

1)c语言源文件main.c内容如下：

  1 #include <stdio.h>

  2 struct mystruct

  3 {

  4 };

  5 int main(int argc, char \*argv[])

  6 {

  7     printf("%d\n",sizeof(struct mystruct));

  8     return 0;

  9 }

采用"gcc main.c"编译之后，输出结果显示"0"，即空结构体大小为0。

采用"g++ main.c"编译之后，输出结果显示"1"，即空结构体大小为1。

2)c++语言源文件main.cpp如下：

  1 #include <cstdio>

  2 class myclass

  3 {

  4 };

  5

  6 struct mystruct

  7 {

  8 };

  9 int main(int argc, char \*argv[])

10 {

11     printf("%d,%d\n",sizeof(myclass),sizeof(struct mystruct));

12     return 0;

13 }

采用"gcc main.cpp"无法编译。

采用"g++ main.cpp"编译之后，输出结果显示"1,1"即空类和结构体大小均为1。

参考资料：

http://hi.baidu.com/ciw\_blue/blog/item/28d74c3f6b134cea54e7239e.html

作者：QuietHeart

Email：quiet\_heart000@126.com

日期：2011年8月2日