wanghetao

sizeof和sizeof(string)的问题

今天看《程序员面试宝典》一书（为了应付将要到来的微软笔试），看到了sizeof(string)这个问题。在Dev C++上测试的结果是4，很不明白。上网搜了一下，得到如下结果：

string strArr1[]={"Trend", "Micro", "Soft"};

sizeof(strArr1)=12

转自：http://apps.hi.baidu.com/share/detail/30398570

关于sizeof(string)，今天看那本面试宝典的时候看到这个表达式，有点吃惊，书上写着sizeof(string)=4;当时很纳闷，难道分配4个字节大小的内存给string吗?查阅了相关资料得出结论：string的实现在各库中可能有所不同，但是在同一库中相同一点是，无论你的string里放多长的字符串，它的sizeof()都是固定的，字符串所占的空间是从堆中动态分配的，与sizeof()无关。

sizeof(string)=4可能是最典型的实现之一，不过也有sizeof()为12、32字节的库实现。 但是VC6.0测试后sizeof(string)=16.还是跟编译器有关

```
#include<iostream>
using namespace std;
void main(void)
{
string a[] = {"aaaaa","bbbb","ccc"};
int x = sizeof(a);
int y = sizeof(string);
cout << x << endl;
cout << y << endl;
}
```

运行结果：

Qt(4)
Redis(1)
Shell与命令(4)
SVN(2)
UML与设计模式(5)
UNIX/LINUX 编程(3)
Vim(1)
Yii Framework(2)
其他(2)
网络编程(4)
正则表达式(1)

随笔档案

2015年7月(1)
2015年5月(3)
2015年4月(1)
2015年3月(1)
2015年2月(3)
2015年1月(3)
2014年12月(2)
2014年11月(1)
2014年9月(8)
2014年8月(2)
2014年7月(12)
2014年6月(9)
2014年5月(1)
2014年4月(2)
2014年3月(2)
2014年1月(1)
2013年11月(8)
2013年10月(5)
2013年8月(1)
2013年7月(2)
2013年6月(2)
2013年5月(5)
2013年4月(4)
2013年2月(1)
2013年1月(2)
2012年12月(1)
2012年11月(3)
2012年8月(1)
2012年7月(2)
2012年6月(3)
2012年5月(15)
2012年4月(8)
2012年3月(7)
2012年2月(5)
2012年1月(1)
2011年12月(4)
2011年11月(21)
2011年10月(16)

最新评论

1. Re:MySQL添加用户、删除用户与授权
删除用户是这么删的🐂
--Moment_s

关于sizeof更多的用法摘
自：http://hi.baidu.com/haijiaoshu/blog/item/a269f527706b910a908f9d5b.html

1、什么是sizeof

首先看一下sizeof在msdn上的定义：

The sizeof keyword gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types). This keyword returns a value of type size_t.

看到return这个字眼，是不是想到了函数？错了，sizeof不是一个函数，你见过给一个函数传参数，而不加括号的吗？sizeof可以，所以sizeof不是函数。网上有人说sizeof是一元操作符，但是我并不这么认为，因为sizeof更像一个特殊的宏，它是在编译阶段求值的。举个例子：

```
cout<<sizeof(int)<<endl; // 32位机上int长度为4
cout<<sizeof(1==2)<<endl; // == 操作符返回bool类型，相当于 cout<<sizeof(bool)<<endl;
```

在编译阶段已经被翻译为：

```
cout<<4<<endl;
cout<<1<<endl;
```

这里有个陷阱，看下面的程序：

```
int a = 0;
cout<<sizeof(a=3)<<endl;
cout<<a<<endl;
```

输出为什么是4，0而不是期望中的4，3？？？就在于sizeof在编译阶段处理的特性。由于sizeof不能被编译成机器码，所以sizeof作用范围内，也就是()里面的内容也不能被编译，而是被替换成类型。=操作符返回左操作数的类型，所以a=3相当于int，而代码也被替换为：

```
int a = 0;
cout<<4<<endl;
cout<<a<<endl;
```

所以，sizeof是不可能支持链式表达式的，这也是和一元操作符不一样的地方。

结论：不要把sizeof当成函数，也不要看作一元操作符，把他当成一个特殊的编译预处理。

2、sizeof的用法

sizeof有两种用法：

（1）sizeof(object)

也就是对对象使用sizeof，也可以写成sizeof object 的形式。例如：

（2）sizeof(typename)

也就是对类型使用sizeof，注意这种情况下写成sizeof typename是非法的。下面举几个例子说明一下：

```
int i = 2;
cout<<sizeof(i)<<endl; // sizeof(object)的用法，合理
```

2. Re:MySQL添加用户、删除用户与授权
in 5 part, it should be:**show databases**;
--西军电__哈哈
3. Re:MySQL添加用户、删除用户与授权
1
--liurwei
4. Re:CGI与FastCGI
春上风格啊。
--h1nson
5. Re:CGI与FastCGI
但是有缺点，于是PHP-fpm就是针对于PHP的，Fastcgi的一种实现，他负责管理一个进程池，来处理来自Web服务器的请求。目前，PHP-fpm是内置于PHP的 这里有错误 php-fpm 全程 ...
--第一缕阳光

阅读排行榜
1. MySQL添加用户、删除用户与授权(467708)
2. C语言 gets（）和scanf（）函数的区别(149964)
3. Linux下ffmpeg的完整安装(93178)
4. Nginx搭建flv视频点播服务器(67782)
5. node.js JS对象和JSON字符串之间的转换(67211)

评论排行榜
1. C语言 gets（）和scanf（）函数的区别(7)
2. sizeof和sizeof(string)的问题(6)
3. MySQL添加用户、删除用户与授权(5)
4. CGI与FastCGI(5)
5. char *p="abc"与char p[]="abc"的不同(4)

推荐排行榜
1. MySQL添加用户、删除用户与授权(16)
2. CGI与FastCGI(11)
3. C语言 gets（）和scanf（）函数的区别(11)
4. sizeof和sizeof(string)的问题(7)
5. 逆波兰表达式(5)

cout<<sizeof i<<endl; // sizeof object的用法，合理

cout<<sizeof 2<<endl; // 2被解析成int类型的object, sizeof object的用法，合理

cout<<sizeof(2)<<endl; // 2被解析成int类型的object, sizeof(object)的用法，合理

cout<<sizeof(int)<<endl;// sizeof(typename)的用法，合理

cout<<sizeof int<<endl; // 错误！对于操作符，一定要加()

可以看出，加()是永远正确的选择。

结论：不论sizeof要对谁取值，最好都加上()。

3、数据类型的sizeof

（1）C++固有数据类型

32位C++中的基本数据类型，也就char,short int(short),int,long int(long),float,double, long double

大小分别是：1，2，4，4，4，8, 10。

考虑下面的代码：

cout<<sizeof(unsigned int) == sizeof(int)<<endl; // 相等，输出 1

unsigned影响的只是最高位bit的意义，数据长度不会被改变的。

结论：unsigned不能影响sizeof的取值。

（2）自定义数据类型

typedef可以用来定义C++自定义类型。考虑下面的问题：

typedef short WORD;

typedef long DWORD;

cout<<(sizeof(short) == sizeof(WORD))<<endl; // 相等，输出1

cout<<(sizeof(long) == sizeof(DWORD))<<endl; // 相等，输出1

结论：自定义类型的sizeof取值等同于它的类型原形。

（3）函数类型

考虑下面的问题：

int f1(){return 0;};

double f2(){return 0.0;}

void f3(){}

cout<<sizeof(f1())<<endl; // f1()返回值为int，因此被认为是int

cout<<sizeof(f2())<<endl; // f2()返回值为double，因此被认为是double

cout<<sizeof(f3())<<endl; // 错误！无法对void类型使用sizeof

cout<<sizeof(f1)<<endl; // 错误！无法对函数指针使用sizeof

cout<<sizeof*f2<<endl; // *f2，和f2()等价，因为可以看作object，所以括号不是必要的。被认为是double

结论：对函数使用sizeof，在编译阶段会被函数返回值的类型取代，

4、指针问题

考虑下面问题：

cout<<sizeof(string*)<<endl; // 4

```
cout<<sizeof(int*)<<endl; // 4

cout<<sizof(char****)<<endl; // 4
```

可以看到，不管是什么类型的指针，大小都是4的，因为指针就是32位的物理地址。

结论：只要是指针，大小就是4。（64位机上要变成8也不一定）。

顺便唧唧歪歪几句，C++中的指针表示实际内存的地址。和C不一样的是，C++中取消了模式之分，也就是不再有small,middle,big,取而代之的是统一的flat。flat模式采用32位实地址寻址，而不再是c中的 segment:offset模式。举个例子，假如有一个指向地址 f000:8888的指针，如果是C类型则是8888(16位, 只存储位移，省略段)，far类型的C指针是f0008888(32位，高位保留段地址，地位保留位移),C++类型的指针是f8888(32位，相当于段地址*16 + 位移，但寻址范围要更大)。

5、数组问题

考虑下面问题：

```
char a[] = "abcdef";

int b[20] = {3, 4};

char c[2][3] = {"aa", "bb"};


cout<<sizeof(a)<<endl; // 7

cout<<sizeof(b)<<endl; // 20*4=80

cout<<sizeof(c)<<endl; // 6
```

数组a的大小在定义时未指定，编译时给它分配的空间是按照初始化的值确定的，也就是7。c是多维数组，占用的空间大小是各维数的乘积，也就是6。可以看出，数组的大小就是他在编译时被分配的空间，也就是各维数的乘积*数组元素的大小。

结论：数组的大小是各维数的乘积*数组元素的大小。

这里有一个陷阱：

```
int *d = new int[10];

cout<<sizeof(d)<<endl; // 4
```

d是我们常说的动态数组，但是他实质上还是一个指针，所以sizeof(d)的值是4。

再考虑下面的问题：

```
double* (*a)[3][6];


cout<<sizeof(a)<<endl;  // 4

cout<<sizeof(*a)<<endl;  // 72

cout<<sizeof(**a)<<endl; // 24

cout<<sizeof(***a)<<endl; // 4

cout<<sizeof(****a)<<endl; // 8
```

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。

6、向函数传递数组的问题。

考虑下面的问题：

```
#include <iostream>
using namespace std;
int Sum(int i[])
{
int sumofi = 0;
for (int j = 0; j < sizeof(i)/sizeof(int); j++) //实际上，sizeof(i) = 4
{
  sumofi += i[j];
}
return sumofi;
}
int main()
{
int allAges[6] = {21, 22, 22, 19, 34, 12};
cout<<Sum(allAges)<<endl;
system("pause");
return 0;
}
```

Sum的本意是用sizeof得到数组的大小，然后求和。但是实际上，传入自函数Sum的，只是一个int 类型的指针，所以sizeof(i)=4，而不是24，所以会产生错误的结果。解决这个问题的方法使是用指针或者引用。

使用指针的情况：

```
int Sum(int (*i)[6])
{
int sumofi = 0;
for (int j = 0; j < sizeof(*i)/sizeof(int); j++) //sizeof(*i) = 24
{
  sumofi += (*i)[j];
}
return sumofi;
}
int main()
{
int allAges[] = {21, 22, 22, 19, 34, 12};
cout<<Sum(&allAges)<<endl;
system("pause");
return 0;
}
```

在这个Sum里，i是一个指向i[6]类型的指针，注意，这里不能用int Sum(int (*i)[])声明函数，而是必须指明要传入的数组的大小，不然sizeof(*i)无法计算。但是在这种情况下，再通过sizeof来计算数组大小已经没有意义了，因为此时大小是指定为6的。

使用引用的情况和指针相似：

```
int Sum(int (&i)[6])
{
int sumofi = 0;
for (int j = 0; j < sizeof(i)/sizeof(int); j++)
{
  sumofi += i[j];
}
return sumofi;
}
int main()
{
int allAges[] = {21, 22, 22, 19, 34, 12};
cout<<Sum(allAges)<<endl;
system("pause");
return 0;
}
```

这种情况下sizeof的计算同样无意义，所以用数组做参数，而且需要遍历的时候，函数应该有一个参数来说明数组的大小，而数组的大小在数组定义的作用域内通过sizeof求值。因此上面的函数正确形式应该是：

```
#include <iostream>
using namespace std;
int Sum(int *i, unsigned int n)
{
int sumofi = 0;
for (int j = 0; j < n; j++)
{
  sumofi += i[j];
}
return sumofi;
}
int main()
{
int allAges[] = {21, 22, 22, 19, 34, 12};
cout<<Sum(i, sizeof(allAges)/sizeof(int))<<endl;
system("pause");
return 0;
```

```
}
```

7、字符串的sizeof和strlen

考虑下面的问题：

```
char a[] = "abcdef";
char b[20] = "abcdef";
string s = "abcdef";
cout<<strlen(a)<<endl;  // 6，字符串长度
cout<<sizeof(a)<<endl;  // 7，字符串容量
cout<<strlen(b)<<endl;  // 6，字符串长度
cout<<sizeof(b)<<endl;  // 20，字符串容量
cout<<sizeof(s)<<endl;  // 12, 这里不代表字符串的长度，而是string类的大小
cout<<strlen(s)<<endl;  // 错误！s不是一个字符指针。
a[1] = '\0';
cout<<strlen(a)<<endl;  // 1
cout<<sizeof(a)<<endl;  // 7，sizeof是恒定的
```

strlen是寻找从指定地址开始，到出现的第一个0之间的字符个数，他是在运行阶段执行的，而sizeof是得到数据的大小，在这里是得到字符串的容量。所以对同一个对象而言，sizeof的值是恒定的。string是C++类型的字符串，他是一个类，所以sizeof(s)表示的并不是字符串的长度，而是类string的大小。strlen(s)根本就是错误的，因为strlen的参数是一个字符指针，如果想用strlen得到s字符串的长度，应该使用sizeof(s.c_str())，因为string的成员函数c_str()返回的是字符串的首地址。实际上，string类提供了自己的成员函数来得到字符串的容量和长度，分别是Capacity()和Length()。string封装了常用了字符串操作，所以在C++开发过程中，最好使用string代替C类型的字符串。

8、从union的sizeof问题看cpu的对界

考虑下面问题：（默认对齐方式）

```
union u
{
  double a;
  int b;
};
union u2
{
  char a[13];
  int b;
};
union u3
{
  char a[13];
  char b;
};
cout<<sizeof(u)<<endl;  // 8
```

```
cout<<sizeof(u2)<<endl;  // 16
cout<<sizeof(u3)<<endl;  // 13
```

都知道union的大小取决于它所有的成员中，占用空间最大的一个成员的大小。所以对于u来说，大小就是最大的double类型成员a了，所以sizeof(u)=sizeof(double)=8。但是对于u2和u3，最大的空间都是char[13]类型的数组，为什么u3的大小是13，而u2是16呢？关键在于u2中的成员int b。由于int类型成员的存在，使u2的对齐方式变成4，也就是说，u2的大小必须在4的对界上，所以占用的空间变成了16（最接近13的对界）。

结论：复合数据类型，如union，struct，class的对齐方式为成员中对齐方式最大的成员的对齐方式。

顺便提一下CPU对界问题，32的C++采用8位对界来提高运行速度，所以编译器会尽量把数据放在它的对界上以提高内存命中率。对界是可以更改的，使用#pragma pack(x)宏可以改变编译器的对界方式，默认是8。C++固有类型的对界取编译器对界方式与自身大小中较小的一个。例如，指定编译器按2对界，int类型的大小是4，则int的对界为2和4中较小的2。在默认的对界方式下，因为几乎所有的数据类型都不大于默认的对界方式8（除了long double），所以所有的固有类型的对界方式可以认为就是类型自身的大小。更改一下上面的程序：

```
#pragma pack(2)
union u2
{
  char a[13];
  int b;
};
union u3
{
  char a[13];
  char b;
};
#pragma pack(8)
cout<<sizeof(u2)<<endl;  // 14
cout<<sizeof(u3)<<endl;  // 13
```

由于手动更改对界方式为2，所以int的对界也变成了2，u2的对界取成员中最大的对界，也是2了，所以此时sizeof(u2)=14。

结论：C++固有类型的对界取编译器对界方式与自身大小中较小的一个。

9、struct的sizeof问题

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，但是因为对界问题，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。

10、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。

wanghetao
关注 - 2
粉丝 - 175
+加关注

7 0

« 上一篇： char *p="abc"与char p[]="abc"的不同
» 下一篇： C++中const总结

posted on 2012-04-04 10:27 wanghetao 阅读(39617) 评论(6) 编辑 收藏

FeedBack:

#1楼 2012-04-26 21:22 | 金山大游侠

受教了

支持(0) 反对(0)

#2楼 2012-06-18 20:59 | Micheal菜菜

不错，很全面，分享一下！

支持(0) 反对(0)

#3楼 2012-06-24 20:12 | 守望者上弦月

很不错，解答了很多困惑...

支持(0) 反对(0)

#4楼 2014-01-09 20:16 | fly~~~

3.（3）中"cout<<sizeof*f2<<endl; // *f2，和f2()等价，因为可以看作object，所以括号不是必要的。被认为是double"

这个试了一下不对，vc中如果f2()定义为double，则sizeof(f2())为8，而sizeof(*f2)总为4的。

6。中倒数第四行"”cout<<Sum(i, sizeof(allAges)/sizeof(int))<<endl;"应该为”cout<<Sum(allAges, sizeof(allAges)/sizeof(int))<<endl;"

支持(1) 反对(0)

#5楼 2016-02-11 19:46 | zg.diligence

不错，已收藏、分享

支持(0) 反对(0)

#6楼 2016-02-11 19:52 | zg.diligence

四楼的建议值得考虑

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

相关博文：

· sizeof()用法汇总
· C/C++中sizeof()的用法——32位和64位下的sizeof（）
· sizeof()用法汇总
· union与struct 的 sizeof 问题
· sizeof()用法汇总
» 更多推荐...

最新 IT 新闻:

· 网易被裁员工再发声：双方已和解 网易会全力协助我的治疗
· 中国科学家发现70倍太阳质量黑洞 郭守敬望远镜立功
· 华为手机推特飙脏话骂苹果？华为：账号被盗
· IDC 武连峰：数字化转型2.0驱动企业服务大变革
· 全面领跑中国DevOps云服务市场，为什么是华为云？
» 更多新闻...