C 中 sizeof 用法 _phenixyf 的专栏 -CSDN 博客

一、sizeof的概念

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、sizeof的使用方法

1、用于数据类型

sizeof使用形式：sizeof（type）
数据类型必须用括号括住。如sizeof（int）。

2、用于变量

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。sizeof(void)都不是正确形式。

三、sizeof的常见结果

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。
1、若操作数具有类型char、unsigned char或signed char，其结果等于1。ANSI C正式规定字符类型为1字节。例如int类型在16位系统中占2个字节，在32位系统中占4个字节。
2、int、unsigned int 、short int、unsigned short 、long int 、unsigned long 、float、double、long double类型的sizeof 在ANSI C中没有具体规定，大小依赖于实现，一般可能分别为2、2、2、2、4、4、4、8、10。
3、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。
4、当操作数具有数组类型时，其结果是数组的总字节数。
5、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是这种类型对象的总字节数，包括任何填充在内。
让我们看如下结构：
struct {char b; double x;} a;
在某些机器上sizeof（a）=12，而一般sizeof（char）+ sizeof（double）=9。这是因为编译器在考虑对齐问题时，在结构中插入空位以控制各成员对象的地址对齐。如double类型的结构成员x要放在被4整除的地址。具体分析见第五节。

6、unsigned影响的只是最高位bit的正负，数据长度是不会被改变的，即sizeof(unsigned int) = sizeof(int)

7、自定义类型的sizeof取值等用于它的类型原型。如typeof short TT;sizeof(TT) = sizeof(short)。

8、参数为结构或类。Sizeof应用在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。
举例说明如下：

```
1: Class Test{int a;static double c};//sizeof(Test)=4.

2: Test *s;//sizeof(s)=4,s为一个指针。

3: Class test1{ };//sizeof(test1)=1;
```

四、Sizeof与Strlen的区别

1、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。
2、sizeof是算符，strlen是函数。
3、sizeof可以用类型做参数，strlen只能用char*做参数，且必须是以"\0"结尾的。
sizeof还可以用函数做参数，比如：

```
1: short f();

2: printf("%d\n", sizeof(f()));
```

输出的结果是sizeof(short)，即2。因为f的返回类型为short，所以相当于求sizeof(short)。
4、数组做sizeof的参数不退化，传递给strlen就退化为指针了。
5、大部分编译程序在编译的时候就把sizeof计算过了，是类型或是变量的长度这就是sizeof(x)可以用来定义数组维数的原因。

```
1: char str[20]="0123456789";

2: int a=strlen(str); //a=10;

3: int b=sizeof(str); //而b=20;
```

6、strlen的结果要在运行的时候才能计算出来，时用来计算字符串的长度，不是类型占内存的大小。
7、sizeof后如果是类型必须加括弧，如果是变量名可以不加括弧。这是因为sizeof是个操作符不是个函数。
8、当适用了于一个结构类型时或变量， sizeof 返回实际的大小，当适用一静态地空间数组， sizeof 归还全部数组的尺寸。
sizeof 操作符不能返回动态地被分派了的数组或外部的数组的尺寸。
9、数组作为参数传给函数时传的是指针而不是数组，传递的是数组的首地址，如：
fun(char [8])
fun(char [])
都等价于 fun(char *)
在C++里参数传递数组永远都是传递指向数组首元素的指针，编译器不知道数组的大小。如果想在函数内知道数组的大小， 需要这样做：进入函数后用memcpy拷贝出来，长度由另一个形参传进去。

```
1: fun(unsiged char *p1, int len)

2: {

3: unsigned char* buf = new unsigned char[len+1]

4: memcpy(buf, p1, len);

5: }
```

我们能常在用到 sizeof 和 strlen 的时候，通常是计算字符串数组的长度。看了上面的详细解释，发现两者的使用还是有区别的，从这个例子可以看得很清楚：

```
1: char str[20]="0123456789";

2: int a=strlen(str); //a=10; >>>> strlen 计算字符串的长度，以结束符 0x00 为字符串结束。

3: int b=sizeof(str); //而b=20; >>>> sizeof 计算的则是分配的数组 str[20] 所占的内存空间的大小，不受里面存储的内容改变。
```

上面是对静态数组处理的结果，如果是对指针，结果就不一样了

```
1: char* ss = "0123456789";

2: sizeof(ss) 结果 4 ===》ss是指向字符串常量的字符指针，sizeof 获得的是一个指针的之所占的空间,应该是

3: 长整型的，所以是4。

4: sizeof(*ss) 结果 1 ===》*ss是第一个字符 其实就是获得了字符串的第一位'0' 所占的内存空间，是char类

5: 型的，占了 1 位。
```

五、sizeof中struct，class，union对齐分析

由于struct，class的类似性，这里主要对struct进行分析。而union和struct的主要区别在于：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是这种类型对象的总字节数，包括任何填充在内。

请看下面的结构：

```
1: struct   MyStruct

2: {

3: double   dda1;

4: char   dda;

5: int   type

6: };
```

对结构MyStruct采用sizeof会出现什么结果呢？sizeof(MyStruct)为多少呢？也许你会这样求：
sizeof(MyStruct)=sizeof(double)+sizeof(char)+sizeof(int)=13
但是当在VC中测试上面结构的大小时，你会发现sizeof(MyStruct)为16。你知道为什么在VC中会得出这样一个结果吗？
其实，这是VC对变量存储的一个特殊处理。为了提高CPU的存储速度，VC对一些变量的起始地址做了"对齐"处理。在默认情况下，VC规定各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。下面列出常用类型的对齐方式(vc6.0,32位系统)。

类型
对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）
Char
偏移量必须为sizeof(char)即1的倍数
int
偏移量必须为sizeof(int)即4的倍数
float
偏移量必须为sizeof(float)即4的倍数
double
偏移量必须为sizeof(double)即8的倍数
Short
偏移量必须为sizeof(short)即2的倍数

各成员变量在存放的时候根据在结构中出现的顺序依次申请空间，同时按照上面的对齐方式调整位置，空缺的字节VC会自动填充（***填充情况1***）。同时VC为了确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍数，所以在为最后一个成员变量申请空间后，还会根据需要自动填充空缺的字节（***填充情况2***）。
下面用前面的例子来说明VC到底怎么样来存放结构的。

```
1: struct   MyStruct

2: {

3: double   dda1;

4: char   dda;

5: int   type

6: };
```

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，其起始地址跟结构的起始地址相同（刚好偏移量0刚好为sizeof(double)的倍数），该成员变量占用sizeof(double)=8个字节；接下来为第二个成员dda分配空间，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，是sizeof(char)的倍数，所以把dda存放在偏移量为8的地方满足对齐方式，该成员

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，其起始地址跟结构的起始地址相同（刚好偏移量0刚好为sizeof(double)的倍数），该成员变量占用sizeof(double)=8个字节；接下来为第二个成员dda分配空间，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，是sizeof(char)的倍数，所以把dda存放在偏移量为8的地方满足对齐方式，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，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。
下面再举个例子，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：

```
1: struct   MyStruct

2: {

3: char   dda;

4: double   dda1;

5: int   type

6: };
```

这个结构占用的空间为多大呢？在VC6.0环境下，可以得到sizeof(MyStruc)为24。结合上面提到的分配空间的一些原则，分析下VC怎么样为上面的结构分配空间的。（简单说明）

```
1: struct   MyStruct

2: {

3:     char   dda;//偏移量为0，满足对齐方式，dda占用1个字节；

4: double   dda1;//下一个可用的地址的偏移量为1，不是sizeof(double)=8

5:                          //的倍数，需要补足7个字节才能使偏移量变为8（满足对齐

6:                          //方式），因此VC自动填充7个字节，dda1存放在偏移量为8

7:                          //的地址上，它占用8个字节。即满足填充情况1

8: int   type; //下一个可用的地址的偏移量为16，是sizeof(int)=4的倍

9:                      //数，满足int的对齐方式，所以不需要VC自动填充，type存

10:                     //放在偏移量为16的地址上，它占用4个字节。

11: }; //所有成员变量都分配了空间，空间总的大小为1+7+8+4=20，不是结构

12:       //的节边界数（即结构中占用最大空间的类型所占用的字节数sizeof

13:       //(double)=8）的倍数，所以需要填充4个字节，以满足结构的大小为

14:       //sizeof(double)=8的倍数。   即满足填充情况2
```

所以该结构总的大小为：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。

VC对结构的存储的特殊处理确实提高CPU存储变量的速度，但是有时候也带来了一些麻烦，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。n字节对齐就是说变量存放的起始地址的偏移量有两种情况：

第一、如果n大于等于该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；

第二、如果n小于该变量的类型所占用的字节数，那么偏移量为n的倍数，不用满足默认的对齐方式。结构的总大小也有个约束条件，分下面两种情况：如果n大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量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。