# C++中 类 和 结构体所占内存大小



[张念](https://me.csdn.net/qq_25867649" \t "https://blog.csdn.net/qq_25867649/article/details/_blank) 2017-02-16 09:39:52 IMG_2571413 IMG_258收藏 3

分类专栏： [C++](https://blog.csdn.net/qq_25867649/category_6103743.html" \t "https://blog.csdn.net/qq_25867649/article/details/_blank) 文章标签： [C++](https://so.csdn.net/so/search/s.do?q=C++&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/qq_25867649/article/details/_blank) [类](https://so.csdn.net/so/search/s.do?q=类&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/qq_25867649/article/details/_blank) [结构体](https://www.csdn.net/gather_28/MtTaEg0sMTA4NDItYmxvZwO0O0OO0O0O.html" \t "https://blog.csdn.net/qq_25867649/article/details/_blank) [内存](https://www.csdn.net/gather_27/MtTaEg0sNDk0NjQtYmxvZwO0O0OO0O0O.html" \t "https://blog.csdn.net/qq_25867649/article/details/_blank)

版权

类所占内存：

类所占内存的大小是由成员变量（静态变量除外）决定的，成员函数（这是笼统的说，后面会细说）是不计算在内的。

摘抄部分：

成员函数还是以一般的函数一样的存在。a.fun()是通过fun(a.this)来调用的。所谓成员函数只是在名义上是类里的。其实成员函数的大小不在类的对象里面，同一个类的多个对象共享函数代码。而我们访问类的成员函数是通过类里面的一个指针实现，而这个指针指向的是一个table，table里面记录的各个成员函数的地址（当然不同的编译可能略有不同的实现）。所以我们访问成员函数是间接获得地址的。所以这样也就增加了一定的时间开销，这也就是为什么我们提倡把一些简短的，调用频率高的函数声明为inline形式（内联函数）。

(一)  
class CBase   
{   
};   
sizeof(CBase)=1；

为什么空的什么都没有是1呢？  
c++要求每个实例在内存中都有独一无二的地址。//注意这句话！！！！！！！！！！  
空类也会被实例化，所以编译器会给空类隐含的添加一个字节，这样空类实例化之后就有了独一无二的地址了。所以空类的sizeof为1。

(二)

class CBase   
{   
int a;   
char p;   
};   
sizeof(CBase)=8;  
记得对齐的问题。int 占4字节//注意这点和struct的对齐原则很像！！！！！  
char占一字节，补齐3字节

(三)  
class CBase   
{   
public:   
CBase(void);   
virtual ~CBase(void);   
private:   
int  a;   
char \*p;   
};   
再运行：sizeof(CBase)=12

C++ 类中有虚函数的时候有一个指向虚函数的指针（vptr），在32位系统分配指针大小为4字节。无论多少个虚函数，只有这一个指针，4字节。//注意一般的函数是没有这个指针的，而且也不占类的内存。

(四)  
class CChild : public CBase   
{   
public:   
CChild(void);   
~CChild(void);

virtual void test();  
private:   
int b;   
};   
输出：sizeof(CChild)=16；  
可见子类的大小是本身成员变量的大小加上父类的大小。//其中有一部分是虚拟函数表的原因，一定要知道

父类子类共享一个虚函数指针

(五)

#include<iostream.h>

class a {};

class b{};

class c:public a{

virtual void fun()=0;

};

class d:public b,public c{};

int main()

{

cout<<"sizeof(a)"<<sizeof(a)<<endl;

cout<<"sizeof(b)"<<sizeof(b)<<endl;

cout<<"sizeof(c)"<<sizeof(c)<<endl;

cout<<"sizeof(d)"<<sizeof(d)<<endl;

return 0;}

程序执行的输出结果为：

sizeof(a) =1

sizeof(b)=1

sizeof(c)=4

sizeof(d)=8

前三种情况比较常见，注意第四种情况。类d的大小更让初学者疑惑吧，类d是由类b，c派生迩来的，它的大小应该为二者之和5，为什么却是8 呢？这是因为为了提高实例在内存中的存取效率．类的大小往往被调整到系统的整数倍．并采取就近的法则，里哪个最近的倍数，就是该类的大小，所以类d的大小为8个字节．

总结：

空的类是会占用内存空间的，而且大小是1，原因是C++要求每个实例在内存中都有独一无二的地址。

（一）类内部的成员变量：

* 普通的变量：是要占用内存的，但是要注意对齐原则（这点和struct类型很相似）。
* static修饰的静态变量：不占用内容，原因是编译器将其放在全局变量区。

（二）类内部的成员函数：

* 普通函数：不占用内存。
* 虚函数：要占用4个字节，用来指定虚函数的虚拟函数表的入口地址。所以一个类的虚函数所占用的地址是不变的，和虚函数的个数是没有关系的

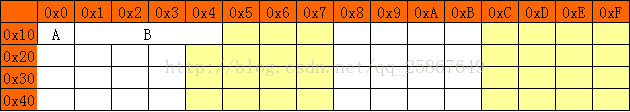
结构体所占内存：

结构体存在内存对齐，类(对象)也如此，甚至于所有变量在内存中的存储也有对齐一说(只是这些对程序员是透明的，不需要关心)。实际上，这种对齐是为了在空间与复杂度上达到平衡的一种技术手段，简单的讲，是为了在可接受的空间浪费的前提下，尽可能的提高对相同运算过程的最少(快)处理。先举个例子：

    假设机器字长是32位的(即4字节，下面示例均按此字长)，也就是说处理任何内存中的数据，其实都是按32位的单位进行的。现在有2个变量：

1. char A;
2. int B;

     假设这2个变量是从内存地址0开始分配的，如果不考虑对齐，应该是这样存储的(见下图,以intel上的little endian为例，为了形象，每16个字节分做一行，后同)：



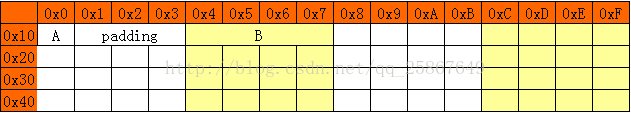
    因为计算机的字长是4字节的，所以在处理变量A与B时的过程可能大致为：

    A:将0x00-0x03共32位读入寄存器，再通过左移24位再右移24位运算得到a的值(或与0x000000FF做与运算)

    B:将0x00-0x03这32位读入寄存器，通过位运算得到低24位的值；再将0x04-0x07这32位读入寄存器，通过位运算得到高8位的值；再与最先得到的24位做位运算，才可得到整个32位的值。

    上面叙述可知，对a的处理是最简处理，可对b的处理，本身是个32位数，处理的时候却得折成2部分，之后再合并，效率上就有些低了。

    想解决这个问题，就需要付出几个字节浪费的代价，改为下图的分配方式：



    按上面的分配方式，A的处理过程不变；B却简单得多了：只需将0x04-0x07这32位读入寄存器就OK了。

    我们可以具体谈结构体或类成员的对齐了：

    结构体在编译成机器代码后，其实就没有本身的集合概念了，而类，实际上是个加强版的结构体，类的对象在实例化时，内存中申请的就是一些变量的空间集合(类似于结构体，同时也不包含函数指针)。这些集合中的每个变量，在使用中，都需要涉及上述的加工原则，自然也就需要在效率与空间之间做出权衡。

    为了便捷加工连续多个相同类型原始变量，同时简化原始变量寻址，再汇总上述最少处理原则，通常可以将原始变量的长度做为针对此变量的分配单位，比如内存可用64个单元，如果某原始变量长度为8字节，即使机器字长为4字节，分配的时候也以8字节对齐(看似IO次数是相同的)，这样，寻址、分配时，均可以按每8字节为单位进行，简化了操作，也可以更高效。

    系统默认的对齐规则，追求的至少两点：1、变量的最高效加工 2、达到目的1的最少空间

    举个例子，一个结构体如下：

1. //by www.datahf.net zhangyu
2. typedef struct T
3. {
4. char c; //本身长度1字节
5. \_\_int64 d;  //本身长度8字节
6. int e;  //本身长度4字节
7. short f;  //本身长度2字节
8. char g;  //本身长度1字节
9. **short** h;  //本身长度2字节
10. };

    假设定义了一个结构体变量C，在内存中分配到了0x00的位置，显然：

    对于成员C.c  无论如何，也是一次寄存器读入，所以先占一个字节。

    对于成员C.d  是个64位的变量，如果紧跟着C.c存储，则读入寄存器至少需要3次，为了实现最少的2次读入，至少需要以4字节对齐；同时对于8字节的原始变量，为了在寻址单位上统一，则需要按8字节对齐，所以，应该分配到0x08-0xF的位置。

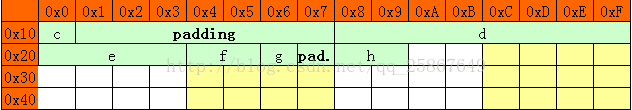
    对于成员C.e  是个32位的变量，自然只需满足分配起始为整数个32位即可，所以分配至0x10-0x13。

    对于成员C.f  是个16位的变量，直接分配在0x14-0x16上，这样，反正只需一次读入寄存器后加工，边界也与16位对齐。

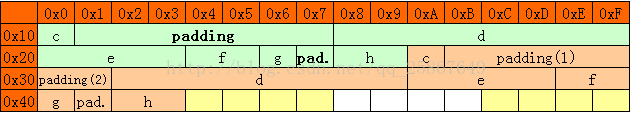
    对于成员C.g  是个8位的变量，本身也得一次读入寄存器后加工，同时对于1个字节的变量，存储在任何字节开始都是对齐，所以，分配到0x17的位置。

    对于成员C.h  是个16位的变量，为了保证与16位边界对齐，所以，分配到0x18-0x1A的位置。

    分配图如下(还不正确，耐心读下去)：

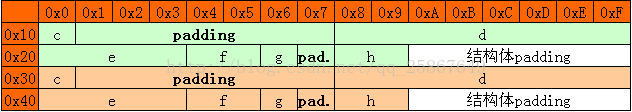


    结构体C的占用空间到h结束就可以了吗?我们找个示例：如果定义一个结构体数组 CA[2]，按变量分配的原则，这2个结构体应该是在内存中连续存储的，分配应该如下图：



    分析一下上图，明显可知，CA[1]的很多成员都不再对齐了，究其原因，是结构体的开始边界不对齐。

    那结构体的开始偏移满足什么条件才可以使其成员全部对齐呢。想一想就明白了：很简单，保证结构体长度是原始成员最长分配的整数倍即可。     上述结构体应该按最长的.d成员对齐，即与8字节对齐，这样正确的分配图如下：

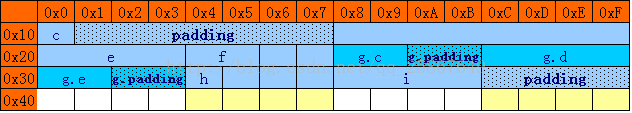


    当然结构体T的长度:sizeof(T)==0x20;

     再举个例子，看看在默认对齐规则下，各结构体成员的对齐规则：

1. //by www.datahf.net zhangyu
2. typedef struct A
3. {
4. char c;  //1个字节
5. int d;  //4个字节，要与4字节对齐，所以分配至第4个字节处
6. short e;  //2个字节， 上述两个成员过后，本身就是与2对齐的，所以之前无填充
7. }; //整个结构体，最长的成员为4个字节，需要总长度与4字节对齐，所以， sizeof(A)==12
8. typedef struct B
9. {
10. char c;  //1个字节
11. \_\_int64 d;  //8个字节，位置要与8字节对齐，所以分配到第8个字节处
12. int e;  //4个字节，成员d结束于15字节，紧跟的16字节对齐于4字节，所以分配到16-19
13. short f;  //2个字节，成员e结束于19字节，紧跟的20字节对齐于2字节，所以分配到20-21
14. A g;  //结构体长为12字节，最长成员为4字节，需按4字节对齐，所以前面跳过2个字节,
15. //到24-35字节处
16. char h;  //1个字节，分配到36字节处
17. int i;  //4个字节，要对齐4字节，跳过3字节，分配到40-43 字节
18. }; //整个结构体的最大分配成员为8字节，所以结构体后面加5字节填充，被到48字节。故：
19. //sizeof(B)==48;

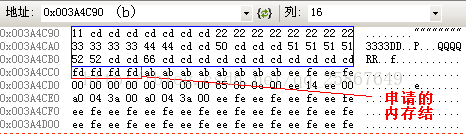
    具体的分配图如下：



 上述全部测试代码如下：

1. //by www.datahf.net zhangyu
2. #include "stdio.h"
3. typedef struct A
4. {
5. char c;
6. int d;
7. short e;
9. };
10. typedef struct B
11. {
12. char c;
13. \_\_int64 d;
14. int e;
15. short f;
16. A g;
17. char h;
18. int i;
19. };
20. typedef struct C
21. {
22. char c;
23. \_\_int64 d;
24. int e;
25. short f;
26. char g;
27. short h;
28. };
29. typedef struct D
30. {
31. char a;
32. short b;
33. char c;
34. };
35. int main()
36. {
38. B \*b=new B;
39. void \*s[32];
40. s[0]=b;
41. s[1]=&b->c;
42. s[2]=&b->d;
43. s[3]=&b->e;
44. s[4]=&b->f;
45. s[5]=&b->g;
46. s[6]=&b->h;
47. s[7]=&b->g.c;
48. s[8]=&b->g.d;
49. s[9]=&b->g.e;
50. s[10]=&b->i;
51. b->c= 0x11;
52. b->d= 0x2222222222222222;
53. b->e= 0x33333333;
54. b->f=0x4444;
55. b->g.c=0x50;
56. b->g.d=0x51515151;
57. b->g.e=0x5252;
58. b->h=0x66;
59. int i1=sizeof(A);
60. int i2=sizeof(B);
61. int i3=sizeof(C);
62. int i4=sizeof(D);
63. printf("i1:%d\ni2:%d\ni3:%d\ni4:%d\n",i1,i2,i3,i4);//12 48 32 6
64. }

运行时的内存情况如下图：



最后，简单加工一下转载过来的内存对齐正式原则：

  先介绍四个概念：

1)数据类型自身的对齐值：基本数据类型的自身对齐值，等于sizeof(基本数据类型)。

2)指定对齐值：#pragma pack (value)时的指定对齐值value。

3)结构体或者类的自身对齐值：其成员中自身对齐值最大的那个值。

4)数据成员、结构体和类的有效对齐值：自身对齐值和指定对齐值中较小的那个值。

  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)

#pragma pack (value)来告诉编译器，使用我们指定的对齐值来取代缺省的。

如#pragma pack (1)  /\*指定按2字节对齐\*/

#pragma pack () /\*取消指定对齐，恢复缺省对齐\*/