

# C++ Tricks 2.5 I386平台的边界对齐(Align)

从 [farseerfc.wordpress.com](http://farseerfc.wordpress.com) 导入

## 2.5 I386平台的边界对齐(Align)

首先提问，既然I386上`sizeof(int)==4`、`sizeof(char)==1`，那么如下结构(struct)A的sizeof是多少？

```
struct A{int i;char c;};
```

答案是`sizeof(A)==8.....1+5=8`？

呵呵，这就是I386上的边界对齐问题。我们知道，I386上有整整4GB的地址空间，不过并不是每一个字节上都可以放置任何东西的。由于内存总线带宽等等的技术原因，很多体系结构都要求内存中的变量被放置于某一个边界的地址上。如果违反这个要求，重则导致停机出错，轻则减慢运行速度。对于I386平台而言，类型为T的变量必须放置在`sizeof(T)`的整数倍的地址上，char可以随便放置，short必须放在2的整数倍的地址上，int必须放在4的整数倍的地址上，double必须放在8的整数倍的地址上。如果违反边界对齐要求，从内存中读取数据必须进行两次，然后将独到的两半数据拼接起来，这会严重影响效率。

由于边界对齐问题的要求，在计算struct的sizeof的时候，编译器必须算入额外的字节填充，以保证每一个变量都能自然对齐。比如如下声明的struct:

```
struct WASTE
```

```
{
char c1;
int i;
char c2;
}
```

实际上相当于声明了这样一个结构：

```
struct WASTE
{
char c1;
char_filling1 [3];//三个字节填充，保证下一个int的对齐
int i;
char c2 ;
char_filling2 [3];//又三个字节填充
}
```

值得注意的是尾部的3个字节填充，这是为了可以在一个数组中声明WASTE变量，并且每一个都自然对齐。因为有了这些填充，所以sizeof(WASTE)==12。这是一种浪费，因为只要我们重新安排变量的声明，就可以减少sizeof：

```
struct WASTE
{
int i;
char c1,c2;
}
```

像这样的安排，sizeof就减少到8，只有2个字节的额外填充。为了与汇编代码相兼容，C语言语法规则规定，编译器无权擅自安排结构体内变量的布局顺序，必须从左向右逐一排列。所以，妥当安排成员顺序以避免内存空间的浪费，就成了我们程序员的责任之一。一般的，总是将结构体的成员按照其sizeof从大到小排列，double在最前，char在最后，这样总可以将结构的字节填充降至最小。

C++继承了C语言关于结构体布局的规定，所以以上的布局准则也适用于C++的class的成员变量。C++进一步扩展了布局规定，同一访问区段(private、public、protected)中的变量，编译器无权重新排列，不过编译器有权排列访问区段的前后顺序。基于这个规则，C++中有的程序员建议给每一个成员变量放在单独区段，在每一个成员声明之前都加上private:、public:、protected:标志，这可以最大限度的利用编译器的决策优势。

在栈中按顺序分配的变量，其边界也受到对齐要求的限制。与在结构中不同的是，栈中的变量还必须保证其后续变量无论是何种类型都可以自由对齐，所以在栈中的变量通常都有平台相关的对齐最小值。在MSVC编译器上，这个最小值可以由宏 `_INTSIZEOF(T)` 查询：

```
#define _INTSIZEOF(T) ( (sizeof(T) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
```

`_INTSIZEOF(T)`会将`sizeof(T)`进位到`sizeof(int)`的整数倍。

由于在栈中分配变量使用`_INTSIZEOF`而不是`sizeof`，在栈上连续分配多个小变量(`sizeof`小于`int`的变量)会造成内存浪费，不如使用结构(struct)或数组。也就是说：

```
char c1,c2,c3,c4;//使用16字节
```

```
char c[4];//使用4字节
```

当然，使用数组的方法在访问数组变量(比如`c[1]`)时有一次额外的指针运算和提领(dereference)操作，这会有执行效率的损失。这又是一种空间(内存占用)vs时间(执行效率)的折中，需要程序员自己根据情况权衡利弊。

`sizeof`的大小可能比我们预期的大，也可能比我们预期的小。对于空类：

```
class Empty {};
```

在通常情况下，`sizeof(Empty)`至少为1。这是因为C++语法规定，对于任何实体类型的两个变量，都必须具有不同的地址。为了符合语法要求，编译器会给`Empty`加入1字节的填充。所以`sizeof()`的值不可能出现0的情况。可是对于以下的类声明：

```
class A:public Empty{virtual ~A(){}};
```

`sizeof(A)`有可能是6，也有可能是5，也有可能是4！必不可少的四个字节是一个指向虚函数表的指针。一个可能有的字节是`Empty`的大小，这是因为编译器在特定情况下会将`Empty`视作一个“空基类”，从而实施“空基类优化”，省掉那毫无作用的一字节填充。另一个字节是`A`的一字节填充，因为从语法上讲，`A`没有成员声明，理应有1字节填充，而从语义上讲，编译器给`A`的声明加入了一个指向虚函数表的指针，从而`A`就不再是一个“空类”，是否实施这个优化，要看编译器作者对语法措词的理解。也就是说，`sizeof`也会出现 $4+1+1=6$ 的情况。具体要看编译器有没有实施“空基类优化”和“含虚函数表的空类优化”。

结构和类的空间中可能有填充的字节，这意味着填充字节中可能有数值，虽然这数值并不影响结构的逻辑状态，但是它也可能不知不觉中影响到你。比如说，你手头正好有一组依赖于底层硬件(比如多处理器)的函数，他们在操纵连续字节时比手动编码要快很多，而你想充分利用这种硬件优势：

```
bool BitCompare(void* begin,void* end,void* another);
```

这个函数将区间`[begin,end)`之间的字节与`another`开始的字节相比较，如果有一位不同就返回`false`，否则返回`true`。

比如你想将这个函数用于你自己的类的`operator==`中，这样可以利用硬件加快速度。不过你在动手前要充分考虑，你的class是否真的要比较每一位。如果在类的成员中

存在编译器填充的字节数，那么应用以上的函数就是不正确的，因为填充的字节中可以有不同的值。为了保证你可以用Bitwise Compare，你必须确保填充的字节中的值也是相同的。这不仅要求你在类的构造函数中初始化类的每一bit而不是每一个成员，也要求你在复制初始化和复制赋值函数中也同时保证bitwise copy语义，而不是编译器默认产生的memberwise语义。当然，你可以通过与BitCompare一同提供的BitCopy来完成这个艰巨的任务。