**字节对齐**

**一 什么是字节对齐**

现代计算机中，内存空间按照字节划分，理论上可以从任何起始地址访问任意类型的变量。但实际中在访问特定类型变量时经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序一个接一个地存放，这就是对齐。

**二、对齐的原因和作用**

不同硬件平台对存储空间的处理上存在很大的不同。某些平台对特定类型的数据只能从特定地址开始存取，而不允许其在内存中任意存放。例如Motorola 68000 处理器不允许16位的字存放在奇地址，否则会触发异常，因此在这种架构下编程必须保证字节对齐。

但最常见的情况是，如果不按照平台要求对数据存放进行对齐，会带来存取效率上的损失。比如32位的Intel处理器通过总线访问(包括读和写)内存数据。每个总线周期从偶地址开始访问32位内存数据，***内存数据以字节为单位存放。***如果一个32位的数据没有存放在4字节整除的内存地址处，那么处理器就需要2个总线周期对其进行访问，显然访问效率下降很多。

因此，通过合理的内存对齐可以提高访问效率。为使CPU能够对数据进行快速访问，数据的起始地址应具有“对齐”特性。比如4字节数据的起始地址应位于4字节边界上，即起始地址能够被4整除。

此外，合理利用字节对齐还可以有效地节省存储空间。但要注意，在32位机中使用1字节或2字节对齐，反而会降低变量访问速度。因此需要考虑处理器类型。还应考虑编译器的类型。在VC/C++和GNU GCC中都是默认是4字节对齐。

**三 对齐的分类和准则**

主要基于Intel X86架构介绍结构体对齐和栈内存对齐，位域本质上为结构体类型。

对于Intel X86平台，每次分配内存应该是从4的整数倍地址开始分配，无论是对结构体变量还是简单类型的变量。

四、不同系统下的C语言类型长度

| **Data Type** | **ILP32** | **ILP64** | **LP64** | **LLP64** |
| --- | --- | --- | --- | --- |
| char | 8 | 8 | 8 | 8 |
| short | 16 | 16 | 16 | 16 |
| int | 32 | 64 | 32 | 32 |
| long | 32 | 64 | 64 | 32 |
| long long | 64 | 64 | 64 | 64 |
| pointer | 32 | 64 | 64 | 64 |

**五、Linux和Microsoft Windows的对齐方式**

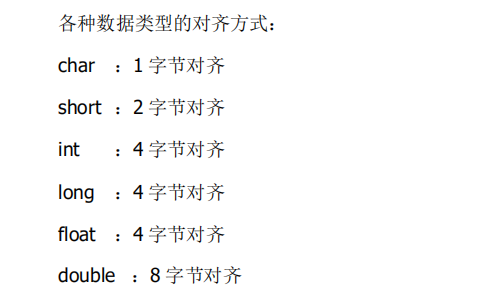
1、Linux的对齐策略:

在Linux中2字节数据类型(例如short)的**地址**必须是2的倍数，而较大的数据类型(例如int，int \*，float和double)的**地址**必须是4的倍数。也就是说Linux下要么2字节对齐，要么4字节对齐，没有其他格式的对齐。

2、Microsoft Windows的对齐策略:

在Windows中对齐要求更严–任何K字节基本对象的地址都必须是K的倍数，K=2，4，或者8.特别地，double或者long long类型数据的地址应该是8的倍数。可以看出Windows的对齐策略和Linux还是不同的。

**六、各种数据类型的对齐方式**



**七、对齐准则**

1、四个重要的基本概念：

1. 数据类型自身的对齐值：char型数据自身对齐值为1字节，short型数据为2字节，int/float型为4字节，double型为8字节。
2. 结构体或类的自身对齐值：其成员中自身对齐值最大的那个值。
3. 指定对齐值：#pragma pack (value)时的指定对齐值value。
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者，即有效对齐值=min{自身对齐值，当前指定的pack值}。

基于上面这些值，就可以方便地讨论具体数据结构的成员和其自身的对齐方式。

其中，有效对齐值N是最终用来决定数据存放地址方式的值。有效对齐N表示“对齐在N上”，即该数据的“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存放。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存放，结构体本身也要根据自身的有效对齐值圆整(即结构体成员变量占用总长度为结构体有效对齐值的整数倍)。

2、对齐准则也可以理解为

1. 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；
2. 结构体每个成员相对结构体首地址的偏移量(offset)都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节(internal adding)；
3. 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在最末一个成员之后加上填充字节{trailing padding}。

对于以上规则的说明如下：

* 第一条：编译器在给结构体开辟空间时，首先找到结构体中最宽的基本数据类型，然后寻找内存地址能被该基本数据类型所整除的位置，作为结构体的首地址。将这个最宽的基本数据类型的大小作为上面介绍的对齐模数。
* 第二条：为结构体的一个成员开辟空间之前，编译器首先检查预开辟空间的首地址相对于结构体首地址的偏移是否是本成员大小的整数倍，若是，则存放本成员，反之，则在本成员和上一个成员之间填充一定的字节，以达到整数倍的要求，也就是将预开辟空间的首地址后移几个字节。
* 第三条：结构体总大小是包括填充字节，最后一个成员满足上面两条以外，还必须满足第三条，否则就必须在最后填充几个字节以达到本条要求。

**八、对齐的隐患**

1、数据类型转换

2、处理器间数据通信

3、排查对齐问题

**九、更改对齐方式**

1、主要是更改C编译器的缺省字节对齐方式。

在缺省情况下，C编译器为每一个变量或是数据单元按其自然对界条件分配空间。一般地，可以通过下面的方法来改变缺省的对界条件：

使用伪指令#pragma pack(n)：C编译器将按照n个字节对齐；

使用伪指令#pragma pack()： 取消自定义字节对齐方式。

2、还有如下的一种方式(GCC特有语法)：

\_\_attribute((aligned (n)))： 让所作用的结构成员对齐在n字节自然边界上。如果结构体中有成员的长度大于n，则按照最大成员的长度来对齐。

\_\_attribute\_\_ ((packed))： 取消结构在编译过程中的优化对齐，按照实际占用字节数进行对齐。

【注】\_\_attribute\_\_机制是GCC的一大特色，可以设置函数属性(Function Attribute)、变量属性(Variable Attribute)和类型属性(Type Attribute)。