本文只为复习内存对齐的理解，试验只包括目前关心的内容。

对于一个C/C++原始类型的变量来说，它的内存地址正好是它长度的整数倍，它就被称作对齐的。

使用C/C++做偏底层软件、交互协议、数据存储时，经常会碰到结构体字节对齐的问题。默认情况下C/C++编译器针对常用的CPU架构，不是将结构体中的数据成员紧紧压在一起的，结构体中每一个数据成员都是对齐的，因此有些数据成员之间会留有一些空洞。编译器这么做的主要原因是：访问对齐的内存地址上的数据效率更高，少部分CPU架构下访问非对齐的内存地址是非法操作。

目前工作环境使用的C/C++编译器是GCC，接下来的demo程序都使用GCC进行试验。

首先看一个直观的对齐demo程序，因为结构体数据成员申明的位置不同，导致了结构体占用内存大小的不同。代码注释部分注明了内存的相对位置，结构体中每个数据成员都是对齐的。

```

#include <stdio.h>

typedef struct

{

char cChar; // 0x00000000

; // 三个字节的空洞

int iInt; // 0x00000004

short sShort; // 0x00000008

; // 两个字节的空洞，结构体整体长度是结构体内原占用空间最长的原始类型（这里是int）的整数倍

} ALIGN\_1\_S;

typedef struct

{

short sShort; // 0x00000000

char cChar; // 0x00000002

; // 一个字节的空洞

int iInt; // 0x00000004

; // 结构体整体长度是int长度的整数倍，所以这里没有多余的空洞

} ALIGN\_2\_S;

int main(int argc, char const \*argv[]) {

ALIGN\_1\_S stAlign1;

ALIGN\_2\_S stAlign2;

printf("sizeof stAlign1 = %d\n", (int)sizeof(stAlign1)); // 长度为12

printf("sizeof stAlign2 = %d\n", (int)sizeof(stAlign2)); // 长度为8

return 0;

}

```

为了让程序员能够对结构体内数据成员的内存对齐方式进行调整，通常C/C++编译器会提供一些工具。对于GCC来说，是通过申明结构体时附加如下属性实现的。

\_\_attribute\_\_((packed))

结构体内所有的数据按紧凑的方式排在一起，彼此之间没有空洞，结构体尾部也不会有多余的空洞。

\_\_attribute\_\_((aligned(n)))

按结构体内所有的数据成员都是对齐的，结构体的整体长度是“n”的整数倍。数据成员之间可能会有空洞，结构体末尾也可能会有空洞。

\_\_attribute\_\_((packed, aligned(n)))

结构体内所有的数据按紧凑的方式排在一起，彼此之间没有空洞，结构体的整体长度是“n”的整数倍。

如下程序显示这些属性对对齐方式的影响。

```

#include <stdio.h>

typedef struct

{

char cChar; // 0x00000000

; // 三个字节的空洞

int iInt; // 0x00000004

short sShort; // 0x00000008

; // 两个字节的空洞，结构体整体长度是结构体内原占用空间最长的原始类型（这里是int）的整数倍

} ALIGN\_1\_S;

typedef struct

{

char cChar; // 0x00000000

int iInt; // 0x00000001

short sShort; // 0x00000005

} \_\_attribute\_\_((packed)) ALIGN\_2\_S;

typedef struct

{

char cChar; // 0x00000000

; // 三个字节的空洞

int iInt; // 0x00000004

short sShort; // 0x00000008

; // 六个字节的空洞，结构体整体长度是8的整数倍

} \_\_attribute\_\_((aligned(8))) ALIGN\_3\_S;

typedef struct

{

char cChar; // 0x00000000

int iInt; // 0x00000001

short sShort; // 0x00000005

; // 一个字节的空洞，结构体整体长度是8的整数倍

} \_\_attribute\_\_((packed, aligned(8))) ALIGN\_4\_S;

int main(int argc, char const \*argv[]) {

ALIGN\_1\_S stAlign1;

ALIGN\_2\_S stAlign2;

ALIGN\_3\_S stAlign3;

ALIGN\_4\_S stAlign4;

printf("sizeof stAlign1 = %d\n", (int)sizeof(stAlign1)); // 长度为12

printf("sizeof stAlign2 = %d\n", (int)sizeof(stAlign2)); // 长度为7

printf("sizeof stAlign3 = %d\n", (int)sizeof(stAlign3)); // 长度为16

printf("sizeof stAlign4 = %d\n", (int)sizeof(stAlign4)); // 长度为8

return 0;

}

```

上面所试验的是C/C++中的原始类型，结构体中也可能出现原始数据类型数组、结构体、联合、结构体数据等这类数据成员。对齐的基本原则是几乎每一个数据成员都能按照内存对齐的方式访问到。

原始类型数组：按照数组的类型对齐；

结构体：结构体中的数据成员都要对齐；

联合：所包含的长度最大的数据类型对齐。如果最大的数据类型是结构体，则这个结构体内部的数据成员都要对齐；

对于结构体中包含不同类型对齐方式的需填坑，不过一般没人这么用吧。

参考资料：

1. C语言字节对齐，http://blog.csdn.net/21aspnet/article/details/6729724
2. 为什么要进行内存对齐及对齐规则，http://www.cnblogs.com/jijiji/p/4854581.html