问题

为什么要进行内存对齐？如何对齐？这个问题其实困扰我很久了，之前做过一些笔试题，经常在sizeof()的问题上出错，但一直没有充分地去理解过，所以这次想好好梳理一下。

什么是内存对齐？

所谓的内存对齐，就是为了让内存存取更加有效率而采取的一种优化手段，对齐的结果是**使得内存中数据的首地址是CPU单次获取数据大小的整数倍**。

比如，CPU单次获取数据的大小是4个字节，对于 int x 而言，如果 x 的地址是0x00000000、0x00000004...等4的倍数，就是内存对齐。

此外，这里说的内存对齐，一般就是针对结构体来进行探讨的，所以这就可以理解在本文后面提到的对整体和成员有不同的对齐方式了。

为什么要内存对齐？

1.硬件因素

经过内存对齐之后，CPU对内存访问的效率会大大提高。

举个例子：

> • 对于int变量 x 占用4个字节的内存大小，假设它存放在 $0x00000003 \sim 0x00000006$ 的位置上，此时 $0x00000003$ 不是4的整数倍。因此，对于每次只取4个字节的CPU而言，对 x 的读取就必须分两次进行，第一次读取 $0x00000000 \sim 0x00000003$，第二次读取 $0x00000004 \sim 0x00000007$，然后再进行拼接处理，才能得到我们想要的数据，可见这样的效率会很低下。
>
> • 倘若经过对齐，即数据的首地址是CPU单次获取数据大小的整数倍，假设 x 存放在 $0x00000004 \sim 0x00000007$ 的位置，那么CPU只需要访问一次内存就可以读取出 x 的值了。

2.可移植性

不是所有的硬件平台都能访问任意地址上的任意数据的，例如有些平台上CPU在内存非对齐的情况下执行二进制代码会崩溃。为了代码的可移植性，进行内存对齐是很有必要的。

如何进行内存对齐？

对齐方式

方式一： 编译器提供了一种手动指定对齐值的方式，**只要在代码前添加关键字 `#pragma pack(n)` 即可**，其中 `n` 是手动指定的内存对齐的字节数。比如 `#pragma pack(4)` 表示以4个字节进行对齐。

方式二： 倘若没有手动设置对齐值，**或者手动设置的对齐值 `n` 大于成员变量中最大的类型的字节数**（注意这一点！），编译器则会**默认将成员变量中最大的类型的字节数设置为对齐值（假设为 m）**。

对齐规则

- **成员对齐：** ① 第一个成员的首地址为0

  ② 假设某成员的类型所占字节数为 `k`，则该成员的首地址为 **`min(n,k)` 的整数倍**。

- **整体对齐：** 结构体总的大小，应该为 **`min(n,m)` 的整数倍**，如果不够就在后面填补占位。

> 补充：如果不能理解上面说的 `min(n,k)` 和 `min(n,m)`，可以看下面的解释：
>
> - 对于 `min(n,k)` 的理解：若手动设置了对齐值 `n`，且 `n<=k`，那么首地址就是 `n` 的倍数，也就是上面的对齐方式一；若 `k<n`，根据对齐方式二可知，编译器不会将 `n` 作为对齐值，而是会选择成员中最大类型的字节数（即 m）作为对齐值，由于 `m>=k`，则该值必然也是 `k` 的整数倍，因此 `min(n,k)` 就可以理解啦。
>
> - 对于 `min(n,m)` 的理解：根据对齐方式一和二，其实系统的对齐值就是 `n` 和 `m` 中最小的那个。当然，整体对齐的意思是整个结构体的总大小要对齐，不够就填补占位。比如，假设对齐值为8，结构体各个成员对齐之后的大小为12，由于12不是8的整数倍，所以编译器会继续填补4个空位，最终结构体的总大小为16。

代码解释

对于方式一，手动设置对齐值 `#pragma pack(n)`，且 `n` 不大于成员变量的最大类型，此时编译器的对齐值就是 `n`。

```
#include<iostream>
#pragma pack(4)                          //对齐值为4
using namespace std;
struct MyStruct
{
    char c;
    double b;
    int a;
};

int main() {
    MyStruct data;
    cout << sizeof(data.a) << endl;      //结果为4
    cout << sizeof(data.b) << endl;      //结果为8
    cout << sizeof(data.c) << endl;      //结果为1，自动填充3个字节
    cout << sizeof(data) << endl;        //结果为16，如果对齐值设置为8，这里结果就是24
    //system("pause");
    return 0;
}

```

对于方式二，先看不进行手动设置对齐值的情况，编译器默认将成员中最大类型的字节数作为对齐值，即double的类型大小，为 8，具体看代码：

```
#include<iostream>
using namespace std;
struct MyStruct
{
```

```
    char c;
    double b;
    int a;
};

int main() {
    MyStruct data;                          //没有手动设置对齐值，编译器默认为最大类型字节数，即8
    cout << sizeof(data.a) << endl;         //结果为4，自动填充4个字节
    cout << sizeof(data.b) << endl;         //结果为8
    cout << sizeof(data.c) << endl;         //结果为1，自动填充7个字节
    cout << sizeof(data) << endl;           //结果为24
    //system("pause");
    return 0;
}
```

对于方式二，手动设置对齐值`n`，且n大于成员变量中的最大类型的字节数`m`，则编译器采用`m`作为对齐值。

```
#include<iostream>
#pragma pack(16)                            //设置对齐值为16,实际对齐值为sizeof(double)=8
using namespace std;
struct MyStruct
{
    int a;
    double b;
    char c;
};

int main() {
    MyStruct data;
    cout << sizeof(data.a) << endl;         //结果为4，自动填充4个字节
    cout << sizeof(data.b) << endl;         //结果为8
    cout << sizeof(data.c) << endl;         //结果为1，自动填充7个字节
    cout << sizeof(data) << endl;           //结果为24
    //system("pause");
    return 0;
}
```

总结

用两句话来总结一下内存的对齐方式：

① 若没有手动设置对齐值，则编译器默认使用成员变量中最大的类型的字节数作为对齐值；

② 若手动设置了对齐值，则编译器会在默认对齐值和手动设置的对齐值之间选择最小的那个作为最终对齐值。

参考资料

【C/C++】内存对齐 到底怎么回事？