C/C++拾遗之内存对齐(Memory Alignment)

2016-07-21 | 2020-07-19 | C++ | 0 Comments
2.4k | 2 mins.

有两篇很不错的文章，总结的已经非常好，所以就不再重复详细说明。

- C++ 内存对齐
- 失传的C结构体打包技艺

其他多数类似文章中没有提及的几个重要的内容（当然上面的文章有详细讲解）：

内存对齐原因:

- 平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
- 性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

为什么未对齐的数据需要两次访问内存呢？由于计算机在从内存中读取数据时，按块进行读取，例如假如一次读取4个字节的数据。当一个占用4个字节的int型变量紧跟在一个刚好自然对齐的short之后，那么如果不进行对齐，int型的数据就必须有2个字节在前面的4字节内存块的后半部分，然后再有2字节在后面4字节内存块的前半部分。如果此时要读取出int，那么就必须先读一次前面的2字节的内存，再读取后面内存块中2字节的内存再进行整合。

默认对齐

如果有指定对齐字节数目，则编译器会按 **类或结构中最大类型长度来对齐**。可以通过语句 `#pragma pack(i)` 来指定对齐字节数目，i的取值为1, 2, 4, 8, 16

对齐规则：

- 如果设置了内存对齐为 i 字节，类中最大成员对齐字节数为j，那么整体对齐字节n = min(i, j)（某个成员的对齐字节数定义：如果该成员是c++自带类型如int、char、double等，那么其对齐字节数=该类型在内存中所占的字节数；如果该成员是自定义类型如某个class或者struct，那个它的对齐字节数 = 该类型内最大的成员对齐字节数）
- 每个成员对齐规则：类中第一个数据成员放在offset为0的位置；对于其他的数据成员（假设该数据成员对齐字节数为k），他们放置的起始位置offset应该是 min(k, n) 的整数倍
- 整体对齐规则：最后整个类的大小应该是n的整数倍
- 当设置的对齐字节数大于类中最大成员对齐字节数时，这个设置实际上不产生任何效果；当设置对齐字节数为1时，类的大小就是简单的把所有成员大小相加

注意：当数据成员较多时，使用各成员的起始位置来分析更可靠

分析举例：

不指定对齐大小

```
#include <iostream>

//以起始位置分析
class node1 {
//未指定内存对齐，默认以类中占用最大的元素大小对齐，x64系统，所以p指针会占用8个字节
    char c;     //c的起始位置为0,占用1,占用位置为0
    char *p;    //p起始位置需要是8的倍数，所以占用位置为8~15
    int a;      //a的起始位置需要是4的倍数，即16~19
    short b;    //b的起始位置需要是2的倍数，即20~21
};
// 根据上面的分析，node1的占用应该是从0~21即22个字节，由于整个类需要以8字节对齐，即占用需要是

// 以占用分析
class node2 {
// 未指定内存对齐，默认以类中占用最大的元素大小对齐
// 如果里面含有结构体或类类型，则该类型的对齐大小以其中的占用最大元素大小来定，即8
    int a;      //a大小为4字节，按4字节对齐，占用8字节内存块的前4字节
    char b;     //b大小为1，按1字节对齐，由于后面的float类型的c需要4个字节
    float c;    //所以，而这个8字节的内存块放不下，需要将c存到下一个内存块
    node1 n;    //导致b与c之间会有3个字节的空闲。n占用24个字节，显然c后面剩余的那个
                //内存块的4节点放不下，n也需要存到下一个内存块
};
```

```
// 根据上面的分析node2占用应该为4+1+3+8+24=40

int main(void)
{
    std::cout << "node1: " << sizeof(node1) << std::endl;
    std::cout << "node2: " << sizeof(node2) << std::endl;
    return 0;
}
```

输出为:

```
node1: 24
node2: 40
```

通过#pragma pack()来指定对齐大小举例:

```

#include <iostream>
#pragma pack(2)
// 指定对齐大小为2

class node3 {
    char a;    // a按1字节对齐，占用为0
    int b;     //b按2字节对齐，占用为2~5
    short c;   //c按2字节对齐，占用为6~7

};
//整个类node3的大小应该是2的整数倍，即8

int main(void)
{
    std::cout << "node3: " << sizeof(node3) << std::endl;
    return 0;

}
```

输出为:

```
node3: 8
```

对于结构体中有数组的情况，该数组的对齐大小依然由数组的类型决定，它只表示多个相应类型的数据聚合在一起，如：

```
class node4 {  //整体按double的8个字节对齐
    int a[5];  //a按4字节对齐，占用0-19
    double d;   //d按8字节对齐，占用24-31
    char c;  //c按1字节对齐，占用32
    short s[3]; //s按2字节对齐，占用34-39
}
//整个node4按8字节对齐，需要是8的位数，即40字节
```

通过sizeof(node4)验证输出为

```
40
```

注意：#pragma另一个作用是保证头文件只被编译一次，用法是在头文件开头加上 `#pragma once`

C++拾遗

C/C++拾遗之关于固定大小的整型

C/C++拾遗之位字段(Bit Field)