31 Arrays in C++

数组，需要16 POINTERS in C++作为前置知识

1. 什么是数组

数组是一些元素的集合，是一些按照特定顺序排列的东西。我们的例子中，数组基本上就是用来表示存放一堆变量的集合，通常来说存放的都是相同的类型。

数组非常重要和有用，因为有非常多的场景中我们想要去表示一大堆数据的整个集合。而创建一大堆变量对我们来说没意义，这些数据应该被放在一个*data set*(数据集)中。

因为变量是需要手动创建的，我们要到代码中设置变量然后给它们一个变量名。
假设我们想要去存储代表某种数据的50个整数，我们肯定不想手动指定五十个，很恐怖且不能维护：要去写50行代码手动把它们设置为0。
这种场景下我们需要用一个数组同时去包含所有这些50个相同类型的元素，会轻松很多。

请记住一个数组就像在一个变量里有很多变量。

```C++
int main()
{
    int example[5];  //分配了足够内存可以存储五个integer
    example[0] = 2;
    example[4] = 4;

    std::cout << example[0] << std::endl;    // 2
    std::cout << example << std::endl;       // 0064FC30(内存地址)，因为它实际一个指针

    std::cin.get();
}
```

内存访问冲突

Memory access violation

```C++
    int example[5];  //分配了足够内存可以存储五个integer

    example[-1] = 5;
    example[5] = 2;
```

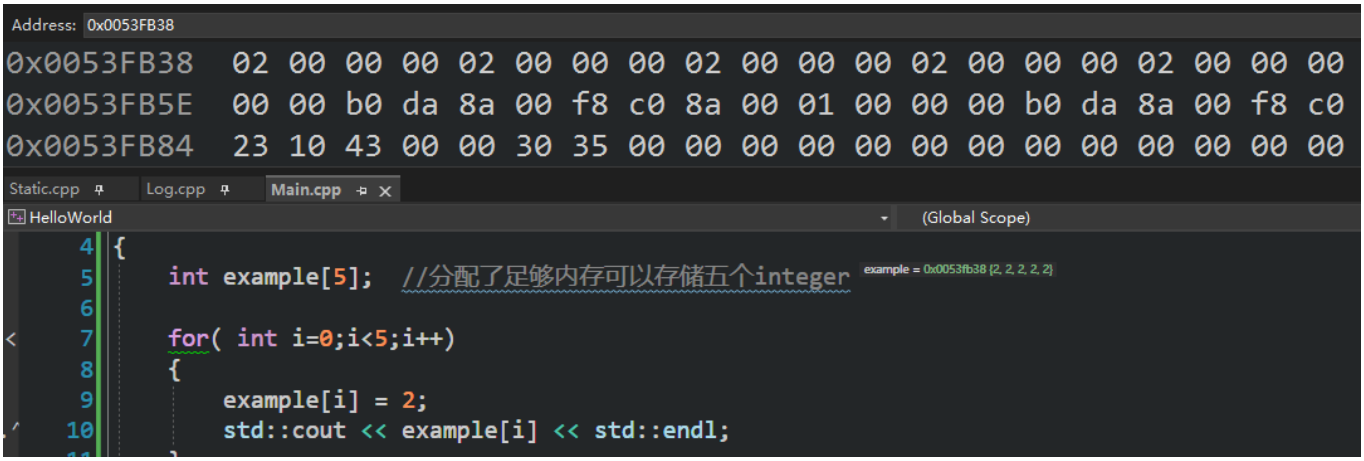
*Debug*模式下，你会得到一个程序崩溃的错误来帮助你调试这些问题
*Release*模式下，你可能不会受到错误，意味着你已经改动了不属于你的内存。
要非常小心这个问题，并且确保总是在数组边界内操作。如果不这样，会产生很难调试的问题，因为你修改了不属于这个数组的内存，但这个内存属于代码中其他变量的一部分，相当于你在代码里无意识地修改了其它变量。

2. For loops

*Array*总是伴随着For loops,因为for循环可以在特定范围内使用索引遍历。
所以如果我们想要设置数组中的每个值，一个很好的方式就是通过for循环实现。

```C++
    for( int i=0;i<5;i++)  // i≤4有性能问题，因为你在做和小于和等于比较
    {
        example[i] = 2;
        std::cout << example[i] << std::endl;
    }
```

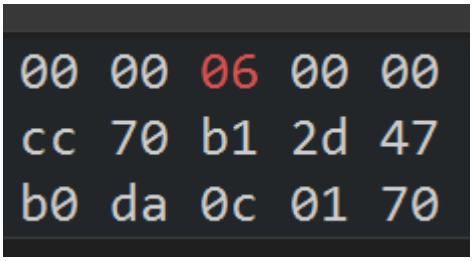
有关数组很重要的一点就是数组中的数据是连续的，也就是说数据都放在了一行
内存分配：每个*integer*都是4 bytes，所以我们这里得到了一行被分成几个4 bytes的20 bytes的内存（不是字面意义分割）



*index*索引其实是地址偏移，偏移=索引x每个元素大小
比如example[2] 的地址偏移就是2 x *sizeof*(int) = 8 bytes

这里它是一个指向包含5个整数的内存块的整形指针

```C++
int example[5];  //分配了足够内存可以存储五个integer
    int* ptr = example;

    for (int i = 0; i < 5; i++)
    {
        example[i] = 2;
        std::cout << example[i] << std::endl;
    }

    example[2] = 5;  // 可以用指针重写，如下行
    *(ptr + 2) = 6;
```



上面说偏移是8字节，但是处理指针时，只要在指针上加上像2这样的值，它会根据数据类型来计算实际的字节数，如上面计算的。

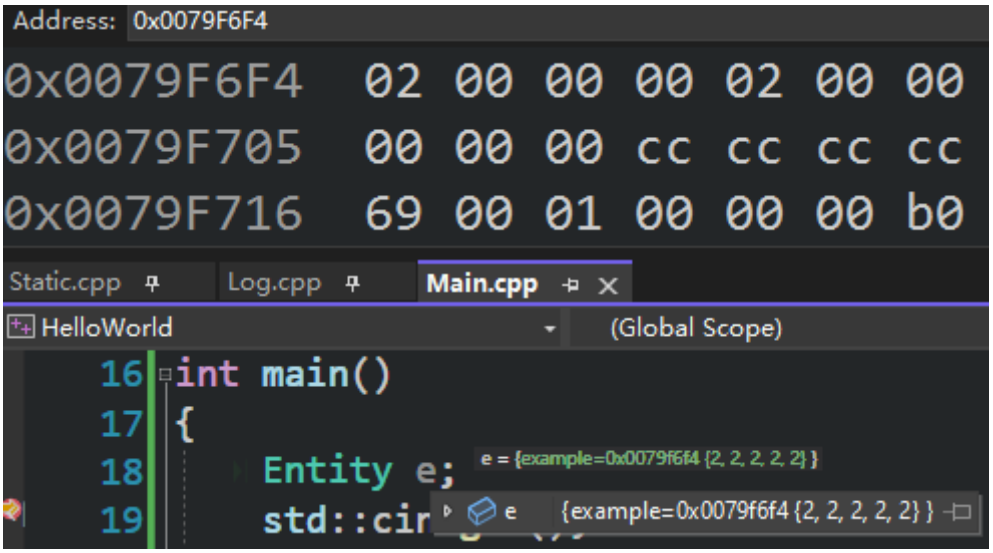
```C++
int main()
{
    int example[5];    // 创建在栈上，跳出作用域时销毁
    for (int i=0;i<5;i++)
    {
        example[i] = 2;
    }
    int* another = new int[5]; //创建在堆上，会存活到我们主动销毁或者程序结束
    for (int i=0;i<5;i++)
    {
        another[i] = 2;
    }
    delete[] another;

    std::cin.get();
}
```

如果你有一个函数要返回新创建的数组，那么你必须要用*new*来分配，除非你传入的参数是一个内存地址

另一件要考虑的是*memory indirection*(内存间接寻址)，意思是我们有一个指针，指针指向另一个保存着我们实际数组的内存块，这会产生一些内存碎片和缓存损失。

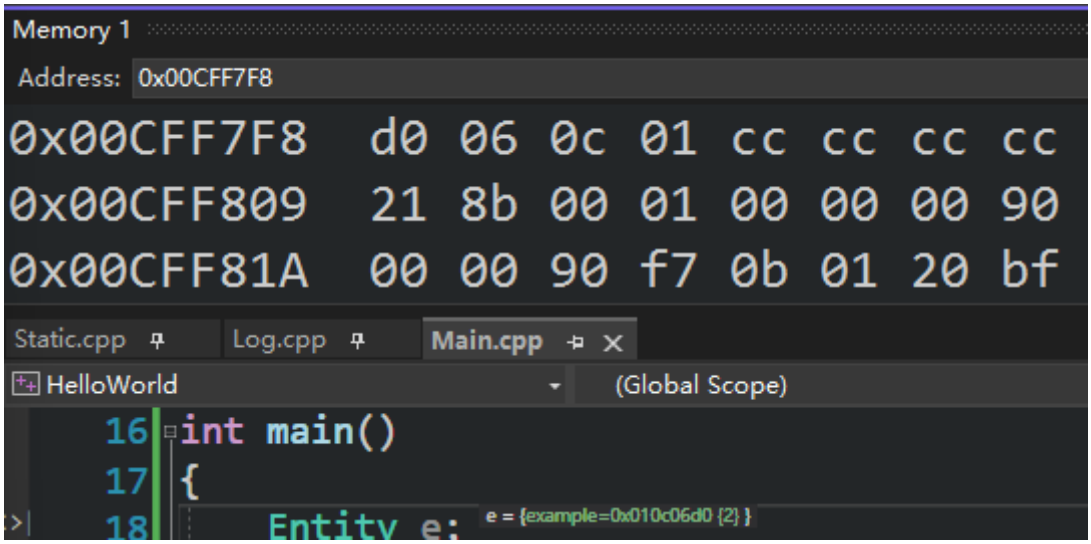
栈上

```C++
class Entity
{
public:
    int example[5];    // 创建在栈上，跳出作用域时销毁

    Entity()
    {
        for (int i = 0; i < 5; i++)
            example[i] = 2;
    }
};
```

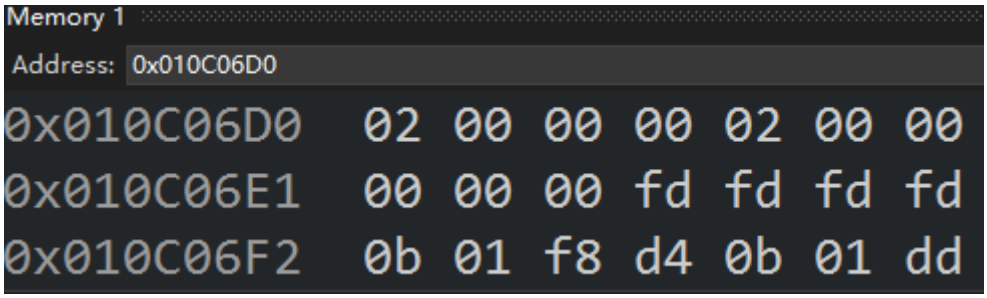


堆上

```C++
class Entity
{
public:
    int* example = new int[5];    // 创建在堆上

    Entity()
    {
        for (int i = 0; i < 5; i++)
            example[i] = 2;
    }
};
```



这里在内存视图中输入`&e`,我们看到另一个内存地址，其实这就是个指针。
由于*endian*(字节序)的原因，我们要做反转。



即`p → p → array`

3. C++ 11 std::array

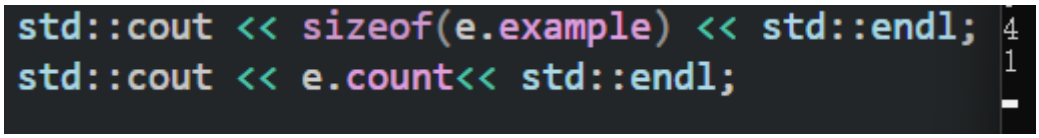
这是一个内置数据结构，定义在C++11的标准库中，很多人用来代替原生数组因为有诸多优点。
比如边界检查，有记录数组的大小。

1. 原生数组

我们实际上无法计算原生数组的大小，不过删除这个数组时，编译器要知道我们需要释放多少内存。不过很麻烦，而且不能相信它，永远不要再数组内存中访问数组的大小。

```C++
int a[5];
std::cout << sizeof(a) << std::endl;  // 20: 数组的大小
int count = sizeof(a) / sizeof(int);
std::cout << count << std::endl;      // 5:元素的数量
```

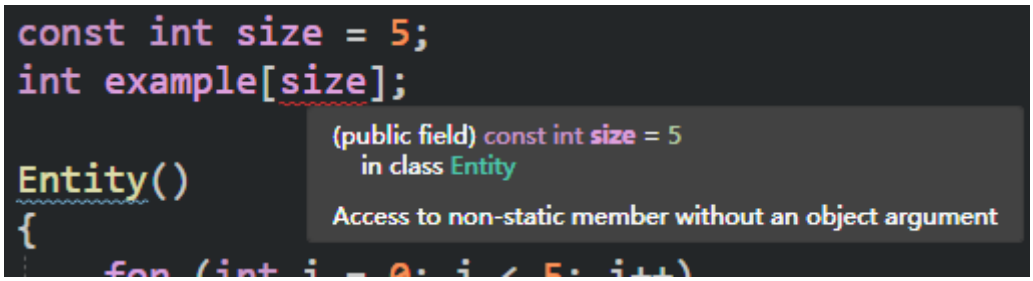
如果做`sizeof(example)`的话，会返回整形指针的大小（4），故你只能在栈分配的数组上用这个技巧



改回`int example[5];`也能工作，但是你真的不能相信这个方法

一个重点

当你在栈中为数组申请内存时，它必须*是一个编译时就需要知道的常量（以后会有别的方法解决）



```C++
static const int size = 5;
int example[size];

// 或者常量表达式 constexpr
// 类中的常量表达式必须是static静态的
```

像下面一样就可以得到数组的大小了

```C++
    static const int exampleSize = 5;
    int example[exampleSize];

    Entity()
    {
        for (int i = 0; i < exampleSize; i++)
            example[i] = 2;
    }

int main()
{
    std::cout << sizeof(Entity) << std::endl;  // 20
}
```

2. std::array

```C++
class Entity
{
public:
    static const int exampleSize = 5;
    int example[exampleSize];

    std::array<int, 5> another;  // std::array

    Entity()
    {
        for (int i = 0; i < another.size(); i++)
            example[i] = 2;
            }
};
```

会有额外开销，因为如果你想的话它会做边界检查，它实际也维持一个整数size，这些额外开销通常是值得的。
通常还是用原生的数组，会快一点。想非常安全的话可以用std::array。