05. 数组（Array）补充

1. 警惕数组的访问越界

先来分析一下这段 C 语言代码的运行结果

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

    int i = 0;
    int arr[3] = {0};

    for(; i<=3; i++){
        arr[i] = 0;
        printf("hello world\n");
    }

    return 0;
}
```

这段代码的运行结果并非是打印三行“hello word”，而是会无限打印“hello world”，这是为什么呢？

因为，数组大小为 3，a[0]，a[1]，a[2]，而我们的代码因为书写错误，导致 for 循环的结束条件错写为了 i<=3 而非 i<3，所以当 i=3 时，数组 a[3]访问越界.

在 C 语言中，只要不是访问受限的内存，所有的内存空间都是可以自由访问的。根据前面讲的数组寻址公式，a[3]也会被定位到某块不属于数组的内存地址上，**而这个地址正好是存储变量 i 的内存地址，那么 a[3]=0 就相当于 i=0，所以就会导致代码无限循环。**

数组越界在 C 语言中是一种未决行为，并没有规定数组访问越界时编译器应该如何处理。因为，访问数组的本质就是访问一段连续内存，只要数组通过偏移计算得到的内存地址是可用的，那么程序就可能不会报任何错误。

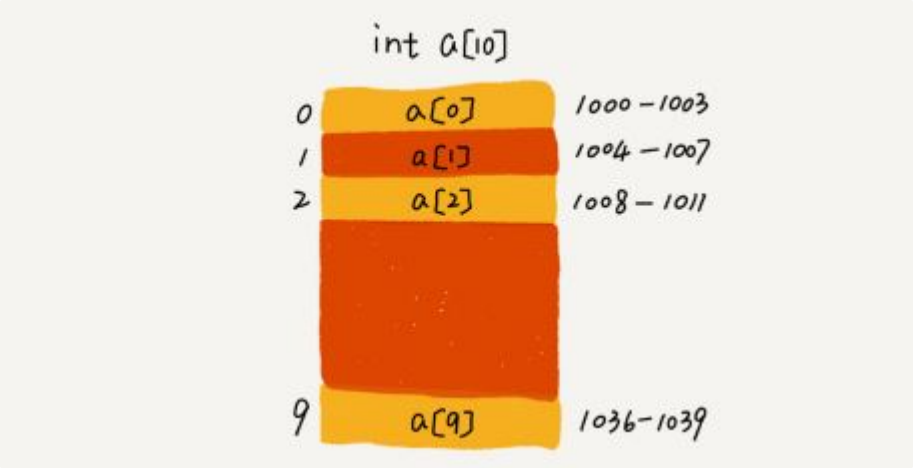
这种情况下，一般都会出现莫名其妙的逻辑错误，就像刚刚举的那个例子，debug 的难度非常的大。而且，很多计算机病毒也正是利用到了代码中的数组越界可以访问非法地址的漏洞，来攻击系统，所以写代码的时候一定要警惕数组越界。

2. 随机访问

> 数组是如何实现根据下标随机访问数组元素的？

拿一个长度为 10 的 int 类型的数组 int[] = new int[10] 来举例.

在下面这个图中，计算机给数组 a[10]，分配了一块连续内存空间 1000 ~ 1039，其中内存块儿的首地址为 base_address = 100.



算机会给每个内存单元分配一个地址，计算机通过地址来访问内存中的数据。当计算机需要随机访问数组中的某个元素时，它会首先通过下面的寻址公式，计算出该元素存储的内存地址

一维数组的寻址公式

$$a[i]\_address = base\_address + i * data\_type\_size$$

> - data _ type _ size 表示数组中每个元素的大小
> - 例子中存储的是 int 数据类型，所以 data _ type _ size 就是 4 个字节.

base _ address = 1000

a[0] = 1000 + 0 * 4 = 1000

a[1] = 1000 + 1 * 4 = 1004

a[2] = 1000 + 2 * 4 = 1008

a[3] = 1000 + 3 * 4 = 1012

.....

a[9] = 1000 + 9 * 4 = 1036

二维数组的寻址公式

对于 m * n 的数组，的地址为：

$$a[i][j]\_address = base\_address + (i * n + j) * type\_size$$

3. 数组编号从0始

> 为什么大多数编程语言，数组要从 0 开始编号，而不是从 1 开始呢？

从数组存储的内存模型上来看，**“下标”最确切的定义应该是“偏移（offset）”** 。

前面也讲到，如果用 a 来表示数组的首地址，a[0]就是偏移为 0 的位置，也就是首地址，a[k]就表示偏移 k 个 type _ size 的位置，所以计算 a[k]的内存地址只需要用这个公式：

$$a[k]\_address = base\_address + k * type\_size$$

但是，如果数组从 1 开始计数，那我们计算数组元素 a[k]的内存地址就会变为：

$$a[k]\_address = base\_address + (k - 1) * type\_size$$

对比两个公式，不难发现，从 1 开始编号，每次随机访问数组元素都多了一次减法运算，对于 CPU 来说，就是多了一次减法指令。

不过我，上面解释得再多其实都算不上压倒性的证明，说数组起始编号非 0 开始不可。所以最主要的原因可能是历史原因。

C 语言设计者用 0 开始计数数组下标，之后的 Java、JavaScript 等高级语言都效仿了 C 语言，或者说，为了在一定程度上减少 C 语言程序员学习 Java 的学习成本，因此继续沿用了从 0 开始计数的习惯。实际上，很多语言中数组也并不是从 0 开始计数的，比如 Matlab。甚至还有一些语言支持负数下标，比如 Python。

4. 容器能否完全代替数组

> 对数组类型，很多语言都提供了容器类，比如 Java 中的 ArrayList、C++ STL 中的 vector。在项目开发中，什么时候适合用数组，什么时候适合用容器呢？

ArrayList 最大的优势就是可以将很多数组操作的细节封装起来。比如前面提到的数组插入、删除数据时需要搬移其他数据等。另外，**它还有一个优势，就是支持动态扩容。**

数组本身在定义的时候需要预先指定大小，因为需要分配连续的内存空间。如果申请了大小为 10 的数组，当第 11 个数据需要存储到数组中时，就需要重新分配一块更大的空间，将原来的数据复制过去，然后再将新的数据插入。

如果使用 ArrayList，就完全不需要关心底层的扩容逻辑，ArrayList 已经实现好了。每次存储空间不够的时候，它都会将空间自动扩容为 1.5 倍大小。

不过，这里需要注意一点，因为扩容操作涉及内存申请和数据搬移，是比较耗时的。所以，如果事先能确定需要存储的数据大小，最好**在创建 ArrayList 的时候事先指定数据大小。**

比如要从数据库中取出 10000 条数据放入 ArrayList。看下面这几行代码，会发现，相比之下，事先指定数据大小可以省掉很多次内存申请和数据搬移操作。

```
1  ArrayList<User> users = new ArrayList(10000);
2  for (int i = 0; i < 10000; ++i) {
3    users.add(xxx);
```

}

```
1
```

> 那数组是否就没有用武之地了呢?

当然不是，有些时候，用数组会更合适些：

1. Java ArrayList **无法存储基本类型**，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。

2. 如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。
3. 还有一个是个人的喜好，当要表示多维数组时，用数组往往会更加直观：

```
// 数组
Object[][] array;

// 而用容器的话则需要这样定义
ArrayList<ArrayList<object> > array。
```

对于业务开发，直接使用容器就足够了，省时省力。毕竟损耗一丢丢性能，完全不会影响到系统整体的性能。但如果是做一些非常底层的开发，比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。

注脚：

笔记时间：2021-02-23 二次总结《算法与数据结构之美》王争专栏