信息学奥赛笔记vector专题

vector

一维动态数组

在之前的课程中，我们学习了静态数组`int a[1000010]`。静态数组可以为我们提供基本使用数组的需求，但是，有些时候，需要对数组实现一些功能，这是一个基础数组完全不可以做到的，比如：想要获取目前数组中保存值的元素个数，给数组动态扩容，或者将数组删除一些元素。

那么我们就需要使用C++库中为我们提供的一个标准模板库`STL--Vector`。

vector的创建

默认创建

```
vector<typename> 数组名;
```

和普通的数组创建方式不一样，vector的数据类型是写在**尖括号**内的。

比如说，我们想要创建一个$double$类型的数组`a`，那么我们该这么写。

```
vector<double> a;
```

这样创建的动态数组会产生一个问题，比如说我们想要访问`nums[0]`的时候，IDE并不会给我们报错，而是会在运行时$return$不是`0`，也就是发生了`Running Error`——**运行时错误**，归根结底产生问题的根源是**下标越界**，访问到非法地址，我们在创建数组的时候其实是没有赋予它任何空间的，上述的例子当中，相当于静态数组的`double a[0]`。这样怎么能访问下标呢?

带参创建——仅长度

```
vector<typename> 数组名(数组的初始长度)
```

如果我们希望一个动态数组，在创建完后立马可以获得初始的长度，那么我们应该在创建完变量后在后方加入括号，然后在括号里写上数组的长度的方式，我们来举个例子:

```
vector<int> a(10);
这局代码等效于
int a[10];
这两个数组在基本情况下用法完全相同。
```

带参创建——长度 + 初值

对于一般的静态数组，如果我希望给数组的值全部赋为`0`是怎么操作呢?

第一个方法：我们可以把数组定义在`main`函数外，在**全局变量区**定义数组，**全局变量区定义的变量是具有初始值0的！！**

第二个方法，我们可以使用`memset`函数：`memset(a, 0, sizeof(a));`

$sizeof$函数表示获取数组所占用的**字节**，这个函数的作用就是我们把数组`a`的每一块地址的值都修改为**0**。

问题来了：如果我希望把数组的初始值赋为`1`，我可以这样写吗?

```
int a[10];
memset(a, 1 sizeof(a));
```

此时老师告诉大家，数组的值都被修改成了`16843009`，我们从二进制的角度来看它，这个数是`0000 0001 0000 0001 0000 0001 0000 0001`刚刚我们也提到了，`memset`函数是**逐字节**的修改，一个字节含有**8**个二进制位，而一个`int`占用**4**个字节，也就是**32**个二进制位，`memset`会修改`4`次而不是`1`次，而我们希望把这个数组的值修改为的`0000 0000 0000 0000 0000 0000 0000 0001`就无法完成了，所以我们**没办法使用`memset`来对数组初始化除了0以外的值。**

那么我们只能对数组进行一次遍历然后将它的初始值修改为`1`了

```
int a[10];
for (int i = 0; i < 10; i++) {
    a[i] = 1;
}
```

动态数组给我们提供了另外一个带参数创建的方法，这个方法允许我们在创建出数组的时候就自动初始化好初值。

```
vector<数据类型> 数组名(数组长度, 数组初值);
vector<int> a(10, 5);
表示我们创建了一个int类型的数组a，他的长度为10，并且从a[0]~a[9]都被赋予初始值5
```

带值创建

有时候，我们希望创建数组的时候它本身就具有不相同的初值，而带参数创建出的动态数组它的初值是一样的，所以我们需要用和静态数组一样的大括号填值的方式来创建，这种方法广泛适用于方向数组(和`bfs`有关)，下标数组(和绑定排序有关)

```
vector<int> a = {1, 2, 3, 4, 5};
这与静态数组
int a[] = {1, 2, 3, 4, 5};
结果相同
```

此时并不需要对数组的初始长度进行定义，编译器会根据我们大括号中填写值的数量，自动为数组开辟空间。

拷贝构造创建

有时候，我们不希望修改原数组的值，但是在处理的过程中，我们可能一定要修改值才好去做，这时我们可以将原数组先备份一份，那么按照静态数组的方法是，我们要先创建好备份数组，然后对原数组进行一次遍历，将值拷贝进我们的备份数组，vector也给我们提供了一个构造

```
vector<int> a = {1, 2, 3, 4, 5};
vector<int> b(a);
```

同学们会发现，这个括号内不仅仅可以填整数的参数，来表示这个数组的初始长度，甚至我们可以直接填另外一个数组，此时这个函数的功能就发生了变化，它表示的意思是直接沿用`a`数组的值，`a`数组的长度，具备`a`数组的各方面的属性。

这种方式是一种**拷贝(Copy)**行为，类似于我们键盘上的`Ctrl + C`。

vector的使用

我们在讲到函数的时候，各位同学一定要明白**函数接口**的概念，接口是什么意思？就是这个函数为我们提供了一种使用的方式。现在我给你一款手机，但是我不告诉你这手机怎么开机，怎么关机，怎么玩游戏，那么该怎么用它呢？正是因为我们知道了函数的接口，所以我们才能够直接使用函数。

比如说，C++库中有个函数叫`abs`绝对值函数，大家可以想想，我们该怎么使用它？是不是传入一个`int`类型的整数，传出一个`int`类型的整数。

那么，我们可以写出这个函数的定义式：

```
int abs(int& a);
```

这就是一个函数的接口，它没有实际上函数的内容，我们使用手机的时候也看不见芯片里到底写的是什么，到底有哪些电路板组成，我们只需要知道怎么用就行了。

那么一个函数的接口需要具备哪些条件呢？

- 函数名
- 函数的传入参数
- 函数的返回值类型

所以各位同学可以发现，函数的第一行定义式当中，一定会包含这三个信息，那么我们想使用一个函数的时候，需要哪些条件呢？

你是不是需要函数名才能调用函数？

你需不需要传入参数来告诉计算机你要丢进去哪些值，来调用函数？

你是不是需要把函数的结果保存下来，给另外一个变量，或者是void返回，那我怎么知道该把这个结果给哪个变量呢？这就是返回值类型。

那么大家会发现，我们真正在使用函数的时候，也只是需要这三个条件，所以我们只需要读懂函数的三个基本要素，以及知道函数的功能，我们就知道该怎么用了。

老师再给大家写个函数。

```
int max(int& a, int& b);
```

该怎么用这个函数呢？

调用`max`这个名字，传入两个整数，传出一个答案，它的作用是获取这两个传入参数的较大值。

看，这就是函数定义的作用，这就是函数接口。

`vector`最强大的地方，并不在于它的创建比静态数组更智能和优秀，跟关键的在于动态数组的内部为我们提供了非常多的接口，我们可以直接使用这些接口去实现一些功能，讲白了就是我们可以直接使用很多别人已经写好的代码，让我们在真正使用的时候直接调用，省去了我们手搓的成本。

由于`vector`的数据类型是不定的，我们接下来的数据类型都用`typename`来定义，也就是这个类型表示所有的数据类型，但是对于你在用的时候，肯定是一个固定的`int, double, char`这时`typename`就被这个数据类型给替换了。

添加函数push_back

```
void push_back(typename& _val);
```

这个函数的作用是把传入的`_val`这个值加入到`vector`的最后，让数组的长度加一。

```
vector<int> a = {1, 2, 3, 4, 5};
//此时数组的长度为5
a.push_back(6);
//此时数组的长度为6，数组内的值为{1, 2, 3, 4, 5, 6}
```

删除函数pop_back

```
void pop_back();
```

这个函数没有任何的传入参数和返回值，我们可以直接调用它，来把数组的最后一个元素删除并且数组的长度减一。

```
vector<int> a = {1, 2, 3, 4, 5, 6};
//此时数组的长度为6
a.pop_back();
//此时数组的长度为5，数组内的元素为{1, 2, 3, 4, 5}
```

获取长度函数size

```
size_t size();
```

$size_t$是一种类似于$unsigned\quad int$类型的数据类型，同学们可以直接记忆为int类型，我们可以通过调用这个函数，获取目前数组的长度

```
vector<int> a = {1, 2, 3, 4, 5};
int ans;
ans = a.size(); //ans = 5;
a.push_back(6);
ans = a.size() // ans = 6;
```

重新设置长度函数resize

```
void resize(int& _size); //重设vector的长度

void resize(int& _size, int& _val) //重设vector的长度，并且将扩容部分的初始值设置为_val
```

*resize*有两个函数，他们可以根据传入参数数量的不同执行不同的功能。

如果只传入一个参数，则表示将数组的长度修改为这个值，如果改少了，原本数组内的值会直接丢失，如果改多了，默认情况会吧扩容的部分设置为`0`，那如果我们再传入第二个参数，则表示可以把扩容的部分设置为这个值

```
vector<int> a = {1, 2, 3, 4, 5};
a.resize(3);
//此时数组的长度为3，值为{1, 2, 3};
```

```
vector<int> a = {1, 2, 3, 4, 5};
a.resize(7);
//此时数组的长度为7，值为{1, 2, 3, 4, 5, 0, 0};
```

```
vector<int> a = {1, 2, 3};
a.resize(6, 3);
//此时数组的长度为6，值为{1, 2, 3, 3, 3, 3};
```

清空函数clear

```
void clear();
```

调用这个函数后，数组的长度设置为`0`，值全部丢失，将数组彻底归为初始状态。方便我们重新给数组填数，这个函数一般用于我们不想开辟额外数组，就只想使用单一数组来完成任务的情况。

```
vector<int> a = {1, 2, 3, 4, 5};
int n;
a.clear();
n = a.size(); // n = 0;
cout << a[0] << endl; //运行时错误，因为数组长度为0，没有下标0
```

起始迭代器begin

```
vector<typename>::iterator begin();
```

迭代器的思想我们在课上已经给大家分享了很多很多，其实就是一个指向数组的指针。

那么起始的迭代器，就是指向数组中下标`[0]`的那个位置。

和指针一样的是，迭代器可以做加减，就表示访问内存地址的前后。

```
vector<int> a = {1, 2, 3, 4, 5};
int ans = *a.begin();   // ans = 1;
ans = *(a.begin() + 1); // ans = 2;
ans = *(a.begin() + 2); // ans = 3;
```

终止迭代器end

```
vector<typename>::iterator end();
```

这个迭代器指向的是数组最后一个元素后虚拟的那个位置，并不完全是数组的最后一个元素。

因为一个容器为了使得它能够继续扩容，我们就要为这个扩容的区域先临时的准备好一个区域。

就比如说，明明说好了是`08:00`上课，那么各位同学会选择正好`08:00`来校吗?

是不是在我们出行的时候会习惯的提前早到，这是中华民族的传统美德，那么沿用进代码当中也就是提前考虑到所有的情况，这样能够提高代码的**复用性**。

既然我们已经学习了数组的起始终止迭代器，接下来我们就可以用迭代器来遍历数组了。

```
vector<int> a = {1,  2,  3,  4,  5};
for (auto it = a.begin(); it != a.end(); it++) {
    cout << (*it) << " ";
}
```

我们先让`it`指向`a`的起始位置，每次让`it++`，也就是指向下一个下标的位置，依次为`a[0], a[1], ... a[n - 1]`那最终，`it`指向了最后一个位置后的虚拟位置，也就是终止迭代器`end`的位置时，遍历就算结束了，所以我们可以拿`it`去与`a.end()`判断是否相等来判断遍历是否结束，这个方法也适用于`map, set`这一类关联迭代器。

重设函数assign

```
void assign(const _InputIterator& _first, const _Input_Iterator _last);
```

同学们可能会看不懂这个函数传入参数的类型，这个类型大家只需要知道它是一个迭代器就行，也就是说，这个重设函数的传入参数是两个迭代器，那么我们可以把另外一个容器的值放进当前这个容器，来初始化当前容器的值。

```
vector<int> a = {1, 2, 3, 4, 5};
vector<int> b;
b.assign(a.begin, a.end());
//上述代码等效于b = a; 或者vector<int> b(a);
```

$assign$的用法和拷贝非常相似，但是$assign$和拷贝最大的区别在于它可以部分拷贝，因为我们传入的迭代器是可以做临时调整的。

```
vector<int> a = {1, 2, 3, 4, 5};
vector<int> b;
b.assign(a.begin(), a.begin() + 3);
//此时b拷贝了a从begin开始，但是不到a.begin() + 3这个位置，所以目前b的长度为3，值为{1, 2, 3};
```

```
vector<int> a = {1, 2, 3, 4, 5};
vector<int> b;
b.assign(a.begin() + 3, a.end());
//从a.begin() + 3开始拷贝，拷贝到不到a.end()那个位置，也就是从4开始拷贝到数组的最后，b目前的长度为2，值为{4, 5};
```

`assign`最强大的地方仍然不在这！它可以做到容器到容器的转换

即使是把`set`内的值变成`vector`存储，`assign`都可以做到

```
set<int> s = {1, 2, 3};
vector<int> a;
a.assign(s.begin(), s.end());
//此时a的长度为3，值为{1, 2, 3};
```

vector与algorithm库内置函数的联用

在`algorithm`算法库，C++为我们提供了非常多的功能，各位同学之前使用过的`sort`，`swap`都是algorithm库的内置函数，对于`vector`，它也可以直接调用这些函数进行一些常用的操作。

排序函数sort

```
void sort(_RAIter& _first, _RAIter& _last); // 排序
void sort(_RAIter, _RAIter, _Compare);      // 自定义排序
```

想要使用`sort`函数，我们需要传入两个迭代器，也就是和普通静态数组用法相同。只不过传参部分替换成了迭代器。

```
vector<int> a = {1, 5, 2, 4, 3};
sort(a.begin(), a.end());
//此时a的值为{1, 2, 3, 4, 5};
```

`sort`函数内为我们提供了第三个参数是自定义函数，同学们之前学习过的`cmp`就是这么使用的。

只不过我们今天从接口的角度来思考一下这个函数。

现在我需要将数组从大到小排列，我就可以修改第三个参数的值。

```
vector<int> a = {1, 5, 2, 4, 3};
sort(a.begin(), a.end(), greater<int>());
//此时a的值为{5, 4, 3, 2, 1};
```

反转函数reverse

```
void reverse(_RAIter& _first, _RAIter& _last); // 反转
```

有时候我们需要将整个数组的值完全颠倒反转，我们就应该在使用这个函数来操作，不仅仅针对`vector`，字符串`string`也可以直接使用。

```
vector<int> a = {1, 3, 2, 4, 5};
reverse(a.begin(), a.end());
// 此时a的值为{5, 4, 2, 3, 1};
```

我们也可以修改传入的迭代器，来实现部分反转。

```
vector<int> a = {1, 2, 3, 4, 5};
reverse(a.begin(), a.begin() + 3);
// 此时a的值为{3, 2, 1, 4, 5};
```

交换函数swap

```
void swap(const _T& a, const _T& b);
```

传入两个任意类型但是是相同类型的变量，交换他们的值，这个类型甚至可以是`vector`数组。

```
vector<int> a = {1, 2, 3, 4, 5};
vector<int> b = {1, 2, 3};
swap(a, b);
// 此时a的值为{1, 2, 3}, b的值为{1,  2,  3,  4,  5};
```