



D2DS | COURSES | 2024

动手写动态数组Vector

Vector<T, Alloc>容器核心原理

@sunrisepeak | 2024.5

- 内存管理 - 手动VS自动

- 自定义分配器支持

- Vector核心原理

- dslings - Vector代码练习

- 基本介绍
 - 定长数据
 - 变长数组
- Vector核心实现
 - 类型定义和数据初始化 - 自定义分配器支持
 - BigFive - 行为控制
 - 常用函数和数据访问
 - 数据增删和扩容机制 - resize
 - 迭代器支持 - 范围for
 - 功能扩展 - 向量加减法
- 总结

手动管理 VS 自动管理



输出结果

```
1 2 3 4  
1 2
```

```
int main() {  
    d2ds::Vector<int> intArr = { 1, 2 };  
  
    intArr.push_back(3);  
    intArr.push_back(4);  
  
    for (int i = 0; i < 4; i++) {  
        std::cout << intArr[i] << " ";  
    }  
    std::cout << std::endl;  
  
    intArr.pop_back();  
    intArr.pop_back();  
  
    for (int i = 0; i < intArr.size() /* 2 */; i++)  
        std::cout << intArr[i] << " ";  
}  
  
return 0;  
};
```

<https://sunrisepeak.github.io/d2ds-courses>

```
int main() {  
    int *intArr = (int *)malloc(sizeof(int) * 2);  
    intArr[0] = 1; intArr[1] = 2; // init  
  
    // do something  
  
    int *oldIntArr = intArr;  
    intArr = (int *)malloc(sizeof(int) * 4);  
  
    intArr[0] = oldIntArr[0]; intArr[1] = oldIntArr[1];  
    free(oldIntArr);  
  
    intArr[2] = 3;  
    intArr[3] = 4;  
  
    for (int i = 0; i < 4; i++) {  
        std::cout << intArr[i] << " ";  
    }  
    std::cout << std::endl;  
    for (int i = 0; i < 2; i++) {  
        std::cout << intArr[i] << " ".
```



自定义分配器支持 - 接口标准化

使用一个分配器类型作为作用域标识, 类型中包含两个静态成员函数用于内存的分配和释放

```
struct Allocator {  
    static void * allocate(int bytes);  
    static void deallocate(void *addr, int bytes);  
};
```

其中 `allocate` 用于分配内存, 它的参数为请求的内存字节数; `deallocate` 用于内存的释放, `addr` 为内存块地址, `bytes` 为内存块的大小

自定义分配器支持 - 接口实现示例



```
struct DefaultAllocator {  
  
    static void * allocate(int bytes) {  
        allocate_counter()++;  
        if (debug())  
            HONLY_LOGI("DefaultAllocator: try to allocate %d bytes", bytes);  
        return malloc(bytes);  
    }  
  
    static void deallocate(void *addr, int bytes) {  
        deallocate_counter()++;  
        if (debug())  
            HONLY_LOGI("DefaultAllocator: free addr %p, bytes %d", addr, bytes);  
        assert(addr != nullptr);  
        free(addr);  
    }  
};
```

<https://sunrisepeak.github.io/d2ds-courses>

内存分配日志打印
内存分配/释放统计
常用大小内存块缓存?

public: // config & status



Vector核心原理 - 类型定义

```
d2ds::Vector<int, d2ds::DefaultAllocator> intVec;
d2ds::Vector<char, StackMemAllocator> charVecByStack(10);
d2ds::Vector<double> doubleVec = { 1.1, 2.2, 3.3 };
```

第一个模板参数用于接收数据类型, 第二个参数用于接收一个满足上面标准的分配器类型。为了方便使用, 使用 `DefaultAllocator` 作为分配器模板参数的默认类型, 这样开发者在不明确指定分配器的时候就会使用默认的分配器进行内存分配

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {

};
```

Vector核心原理 - 数据定义和初始化



```
d2ds::Vector<int> vec1;
d2ds::Vector<int> vec2(10);
d2ds::Vector<int> vec3 = { 1, 2, 3 };
```

- 数据成员定义
- 默认初始化
- 指定长度初始化
- 列表初始化

<https://sunrisepeak.github.io/d2ds-courses>

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:

    Vector() : mSize_e { 0 }, mDataPtr_e { nullptr } { }

    Vector(int size) : mSize_e { size } { 分配内存
        1 mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        for (int i = 0; i < mSize_e; i++) {
            2 new (mDataPtr_e + i) T();
        }
    } 定位new - placement new - 只构造对象
```

↑
objPtr = new Obj()

```
    Vector(std::initializer_list<T> list) {
        mSize_e = list.end() - list.begin();
        mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        auto it = list.begin();
        T *dataPtr = mDataPtr_e;
        while (it != list.end()) {
            new (dataPtr) T(*it);
            it++; dataPtr++;
        }
    }

private:
    int mSize_e;
    T * mDataPtr_e;
};
```

Vector核心原理 - 行为控制 - 析构



由于使用了内存分配和对象构造分离的模式,所以在析构函数中需要对数据结构中的元素要先析构,最后再释放内存。即需要满足如下构造/析构链,让对象的创建和释放步骤对称:

- 分配对象内存A
- 基于内存A构造对象B
- 析构对象B
- 释放B对应的内存A

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    ~Vector() {
        if (mSize_e) {
            for (int i = 0; i < mSize_e; i++) {
                (mDataPtr_e + i)->~T();
            }
        }
        Alloc::deallocate(mDataPtr_e, mSize_e * sizeof(T));
    }
}
```

Vector核心原理 - 行为控制 - 拷贝语义



在拷贝构造函数中, 使用 `new (addr) T(const T &)` 把拷贝构造语义传递给数据结构中存储的元素

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector(const Vector &dsObj) : mSize_e { dsObj.mSize_e } {
        mDataPtr_e = (T *) Alloc::allocate(sizeof(T) * mSize_e);
        for (int i = 0; i < mSize_e; i++) {
            new (mDataPtr_e + i) T(dsObj.mDataPtr_e[i]);
        }
    }
}
```

数据深拷贝

Vector核心原理 - 行为控制 - 拷贝语义



在拷贝赋值函数中, 先调用析构函数进行数据清理, 同时也使用 `operator=` 进行语义传递

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector & operator=(const Vector &dsObj) {
        D2DS_SELF_ASSIGNMENT_CHECKER
        this->~Vector();
        mSize_e = dsObj.mSize_e;
        mDataPtr_e = static_cast<T *>(Alloc::allocate(sizeof(T) * mSize_e));
        for (int i = 0; i < mSize_e; i++) {
            mDataPtr_e[i] = dsObj.mDataPtr_e[i];
        }
        return *this;
    }
}
```

Vector核心原理 - 行为控制 - 移动语义



在移动构造函数中, 只需要把要目标对象的资源移动到该对象, 然后对被移动的对象做重置操作即可。对于 Vector来说, 只需进行浅拷贝数据成员, 并对被移动的对象置空

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    Vector(Vector &&ds0bj) : mSize_e { ds0bj.mSize_e } {
        mDataPtr_e = ds0bj.mDataPtr_e;
        // reset
        ds0bj.mSize_e = 0;
        ds0bj.mDataPtr_e = nullptr;
    }
}
```

```
Vector & operator=(Vector &&ds0bj) {
    D2DS_SELF_ASSIGNMENT_CHECKER
    this->~Vector();
    mSize_e = ds0bj.mSize_e;
    mDataPtr_e = ds0bj.mDataPtr_e;
    // reset
    ds0bj.mSize_e = 0;
    ds0bj.mDataPtr_e = nullptr;
    return *this;
}
```

数据浅拷贝

Vector核心原理 - 数据增删 - 扩容/缓存机制



当动态数组Vector执行push操作进行添加元素时, 如果每次都需要重新分配内存这会极大的影响效率

```
void push(const int &obj) {
    newDataPtr = malloc(sizeof(int) * (size + 1)); // 分配内存
    copy(newDataPtr, oldDataPtr); // 复制数据
    free(oldDataPtr); // 释放内存
    newDataPtr[size + 1] = obj; // 添加新元素
    size++; // 数量加1
}
```

Vector核心原理 - 数据增删 - 扩容/缓存机制



通过引入内存容量的缓存或者说预分配机制, 来避免过多的内存分配释放, 可以有效的降低它的影响。所以就需要引入另外一个标识 `mCapacity_e` 来标识当前内存最大容量, 而 `mSize_e` 用来标识当前数据结构中的实际元素数量, 所以 `mCapacity_e` 是大于等于 `mSize_e` 的

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
private:
    int mSize_e, mCapacity_e;
    T * mDataPtr_e;
}
```

这里需要先说明一下, 扩容(缩容)机制通常是包含两个概念或步骤:

- 第一个, 扩容(缩容)的条件, 也是执行实际操作的时机。通常扩容发生在数据增加操作, 缩容发生在数据删除操作中
- 第二个, 具体的扩容(缩容)规则。最简单的就是二倍扩容(缩容)

Vector核心原理 - push_back - 扩容



在每次扩容的时候，可以选择基于当前容量的二倍进行扩容。例如：当 `mCapacity_e` 等于4时，做扩容时应该分配可以容纳8个元素的内存

```
d2ds::Vector<int> intArr = {0, 1, 2, 3};  
intArr.push_back(4);  
/*  
old: mCapacity_e == 4, mSize_e == 4  
      +-----+  
mDataPtr_e -> | 0 | 1 | 2 | 3 |  
      +-----+  
new: mCapacity_e == 8, mSize_e == 5  
      +-----+  
mDataPtr_e -> | 0 | 1 | 2 | 3 | 4 |   |   |  
      +-----+  
*/
```

Vector核心原理 - push_back - 扩容



什么时候扩容? 最直观的是增加元素, 但容量又不够的时候。执行push_back时, 当 `mSize_e + 1 > mCapacity_e` 时就需要扩容来获取更大的空间用于新数据/元素的存放, 既是否扩容需要在存储新元素操作之前

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    void push_back(const T &element) {
        if (mSize_e + 1 > mCapacity_e) { 扩容条件/时机
            resize(mCapacity_e == 0 ? 2 : 2 * mCapacity_e); 扩容规则
        }
        new (mDataPtr_e + mSize_e) T(element);
        mSize_e++;
    }
}
```

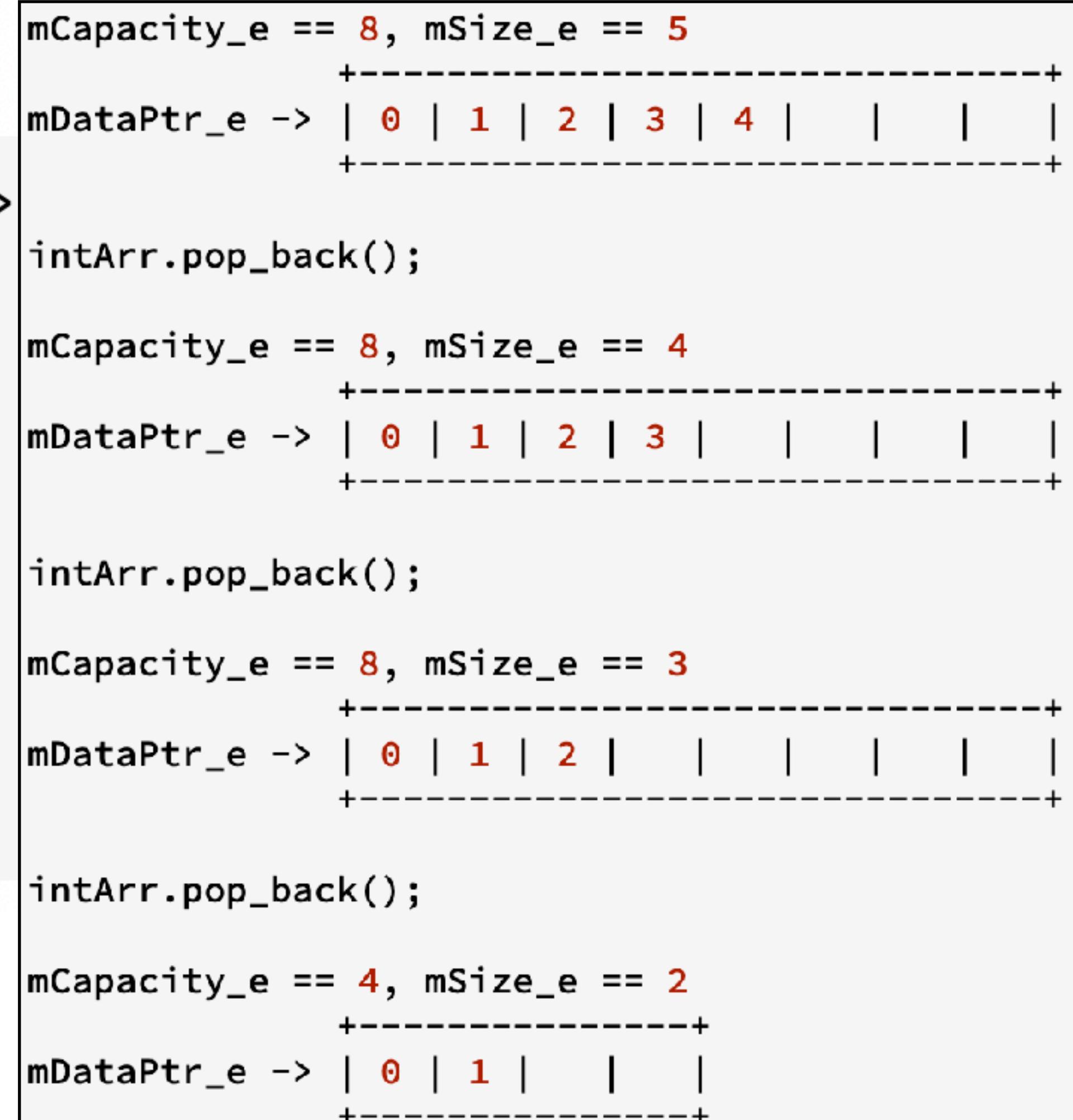
Vector核心原理 - pop_back - 缩容



当 `mSize_e <= mCapacity_e / 3` 时就触发一次二倍扩容机制的执行, 把容量从8缩小一半到4, 此时实际存储的数据量 `mSize_e == 2`。这里需要注意的是, 虽然 `pop_back` 不一定会释放Vector管理的内存, 但依然需要去调用被删除元素的析构函数去释放它额外管理的资源(如果存在)

```
template <typename T, typename Alloc = DefaultAllocator>
class Vector {
public:
    void pop_back() {
        mSize_e--;
        (mDataPtr_e + mSize_e)->~T();
        if (mSize_e <= mCapacity_e / 3) { 缩容条件
            resize(mCapacity_e / 2); 缩容规则
        }
    }
}
```

<https://sunrisepeak.github.io/d2ds-courses>



Vector核心原理 - resize实现

```
void resize(int n) { // only mSize_e <= n
    auto newDataPtr = n == 0 ? nullptr : static_cast<T *>(Alloc::allocate(n * sizeof(T)))
    for (int i = 0; i < mSize_e; i++) {
        new (newDataPtr + i) T(mDataPtr_e[i]);
        mDataPtr_e[i]~T();
    }
    if (mDataPtr_e) {
        // Note: // memory-size is mCapacity_e * sizeof(T) rather than mSize_e * sizeof(T)
        Alloc::deallocate(mDataPtr_e, mCapacity_e * sizeof(T));
    }
    mCapacity_e = n;
    mDataPtr_e = newDataPtr;
}
```

1.分配新内存块

2.老数据迁移和释放

3.释放老内存块

4.数据成员更新

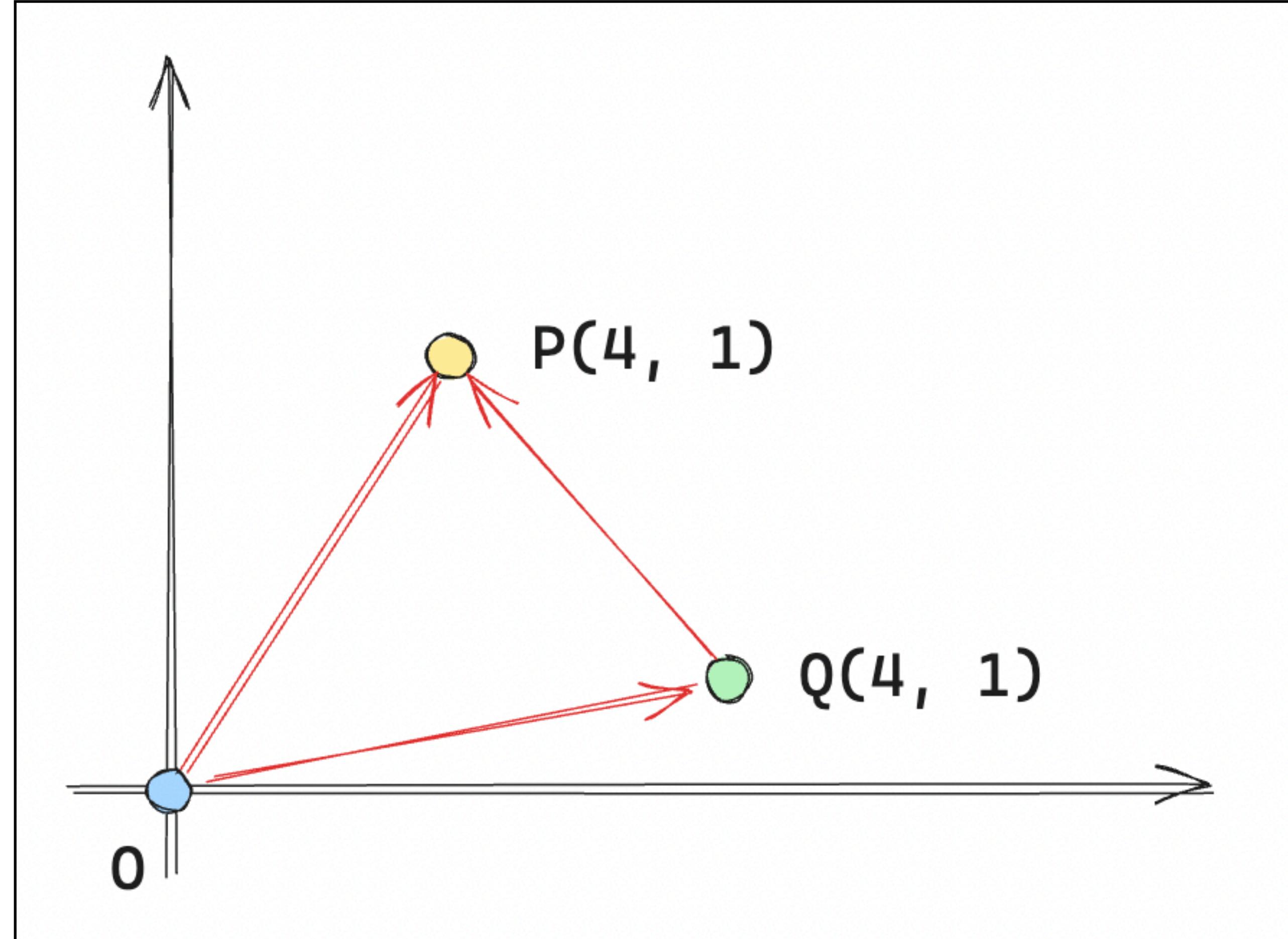
Vector功能扩展 - 向量加减法



假设有如下**OQ**、**OP**、**QP**三个向量

```
^  
| * P(2, 4)  
| | *Q(4, 1)  
*----->  
O(0, 0)
```

```
d2ds::Vector<int> OQ = { 4, 1 };  
d2ds::Vector<int> OP = { 2, 4 };  
d2ds::Vector<int> QP = { -2, 3 };  
d2ds_assert(OQ + QP == OP);  
d2ds_assert(OP - OQ == QP);
```



Vector功能扩展 - 向量加减法



- 重载==运算符
- 重载+/-运算符

```
template <typename T>
bool operator==(const Vector<T> &v1, const Vector<T> &v2) {
    bool equal = v1.size() == v2.size;
    if (equal) {
        for (int i = 0; i < v1.size()); {
            if (v1[i] != v2[i]) {
                equal = false;
                break;
            }
        }
    }
    return equal;
}
```

```
template <typename T>
Vector<T> operator+(const Vector<T> &v1, const Vector<T> &v2) {
    Vector<T> v(2);
    v[0] = v1[0] + v2[0];
    v[1] = v1[1] + v2[1];
    return std::move(v);
}
```

for ?

总结



- 变长数组内存: 手动管理 VS 自动管理
- 分配器定义 和 自定义支持
- 扩容机制的简单实现 - 两倍扩容
- 小扩展 - 向量的加减法实现

dslings - 代码练习



Vector代码练习

<https://sunrisepeak.github.io/d2ds-courses>