47 Dynamic Arrays in C++ (std - -vector)

现在我们终于开始写一些C++标准库的东西了。*Standard Template Library*（标准模板库）本质上是一个库，里面装满了*containers*容器，容器类型。这些容器包含特定的数据，之所以被称为标准模板库是因为它可以模板化任何东西。
你只需要知道，*templates*（模板）可以处理你提供的底层数据类型，这意味着你不需要编写自己的数据结构或者类似的东西。

1. C++的动态数组（std::vector）

所以C++提供给我们一个叫做`Vector`的类，这个Vector在*std namespace*（std命名空间中）。

为什么叫Vector？可以在链接中了解到背后的故事：
它被称为向量是因为标准模板库的设计者 Alex Stepanov 在寻找一个名称以区分它与内置数组时采用了这个名字。他现在承认这是一个错误，因为数学已经使用术语 "向量" 来表示一组固定长度的数字序列。而 C++11 则进一步加重了这个错误，引入了一个名为 'array' 的类，它的行为类似于数学上的向量。
Alex 给我们的教训是：在给事物命名时要非常小心谨慎。

所以它其实不应该被叫做*Vector*（向量），而是应该被称为类似*ArrayList*，这样更有意义，因为它本质上是一个动态数组。它有点像一个集合，一个不强制其实际元素具有唯一性的集合。
换句话说，它基本上就是一个*array*（数组），不过与C++普通数组类型（原始数组或标准数组类31 Arrays in C++）不同的是，它可以调整数组大小，这意味着当你创建这个vector的时候，这个动态数组的时候，它并没有固定大小。你可以给它一个固定大小，如果你想用一个特定的大小初始化它。但一般情况下我们不给它设置一个size。
你只需要创建这个Vector然后把元素放进去，每次你往里面放一个元素，数组大小会增长。
原理：当你超过分配的内存大小时，它会在内存中创建一个比第一个大的新数组，把所有东西都复制到这里，然后删除旧的那个，这样你就拥有了更多存储空间的新数组。（所以可以猜测Alex当时是觉得动态数组可以像向量一样无限延长……）

C++

```
struct Vertex
{
    float x, y, z;
};

std::ostream& operator<<(std::ostream& stream, const Vertex& vertex)
{
    stream << vertex.x << "," << vertex.y << "," << vertex.z;
    return  stream;
}


int main()
{
    Vertex* vertices = new Vertex[5];

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

我们可能有一个程序，允许用户输入数据，我们肯定不想让用户在超过分配的内存量后显示你不能输入更多了。我们需要一种方式，当你到达它们的最大容量时，重新调整容量使容量变得更大以存储更多的数据。另一个解决方案就是分配变态大小的数组，当然这也是不理想的，因为你要用掉那么多内存，会造成浪费。

这里就可以用到Vector 类：

```C++
#include <vector>

std::vector<Vertex> vertices; //不像java，之类可以指定原始类型如int，float等
```

这里存储的对象而不是指针，主要考虑的是存储Vertex对象比存储指针在技术上更优。如果分配的是对象，你的内存分配将是在一条直线上的，动态数组是内存连续的数组，这意味着它在内存中不是*fragment*（碎片），你会得到一个顶点，然后后面跟着一个接一个的（x，y，z）。这是最优的，因为如果你想要遍历它们、设置它们、改变它们还是读取它们，不惯你想对它们做什么，因为在某种意义上它们都在同一条高速缓存线上。

C++

```cpp
#include <iostream>
#include <string>
#include <vector>

struct Vertex
{
    float x, y, z;
};

std::ostream& operator<<(std::ostream& stream, const Vertex& vertex)
{
    stream << vertex.x << "," << vertex.y << "," << vertex.z;
    return stream;
}


int main()
{
    std::vector<Vertex> vertices;
    vertices.push_back({1, 2, 3});
    vertices.push_back({4, 5, 6});

    //两种方法
    for (int i = 0; i < vertices.size(); i++)
    {
        std::cout << vertices[i] << std::endl;
    }

    for (Vertex& v : vertices)
    //for (Vertex v : vertices)
    //这里实际上是将每个vertex复制到这个for范围循环中，我们希望避免复制而加上&
    {
        std::cout << v << std::endl;
    }
    /*for (Vertex v : vertices) 是 C++11 引入的范围基于范围的循环语法，也称为范围循环或foreach循环。

在这种循环中，每次迭代都会将容器 vertices 中的元素赋值给循环变量 v。循环会遍历整个容器，依次处理每个元素。

vertices 是一个存储了 Vertex 结构体对象的 std::vector 容器。通过使用范围基于范围的循环，你可以逐个访问容器中的每个元素，并将其赋值给变量 v。

在循环体内部，你可以像处理普通的 Vertex 对象一样使用变量 v。在这里，使用了重载的 operator<< 函数将每个 Vertex 对象输出到 std::cout 流中。

总而言之，for (Vertex v : vertices) 这个范围基于范围的循环语法使得遍历容器元素变得更加简洁和直观，减少了手动索引迭代的繁琐。它是 C++ 中常用的循环形式之一。*/
    std::cin.get();
}

//Output:
//1,2,3
```

```
//4,5,6

//1,2,3
//4,5,6
```

如果我们要清除verties列表：

```C++
verticies.clear();  //会将数组大小设回0
```

我们也可以单独移除某个元素：

```C++
verticies.erase() // 传参应该是一个iterator（迭代器）

//如果我们想移除第二个元素，也就是索引1
vertices.erase(vertices.begin() + 1);  // 1,2,3
```

2. 注意事项

要提到的另一件事是，当你将这些vector传递给函数或者类或其它东西时，你要确保你是通过引用传递它们的：

```C++
void Function(const std::vector<Vertex>& vertices){}
```

这样做可以确保你没有把整个数组复制到这个函数中，这点非常重要。