48 Optimizing the usage of std - -vector in C++

1. C++的stdvector使用优化

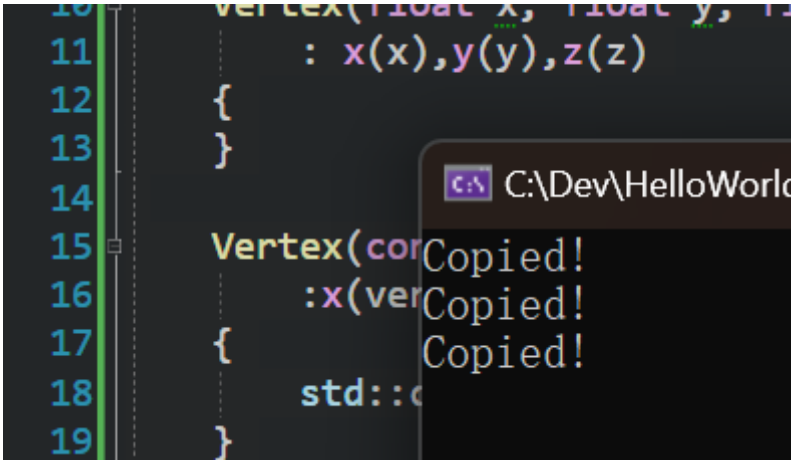
想要优化vector的使用，我们应该先知道vector是怎么工作的，以及如何改变它使之能更好地工作。

`std::vector class`基本上是这样工作的：
你创建一个vector，然后你开始`push_back`元素，也就是向数组中添加元素。如果vector的容量不够大，不能容纳你想要的新元素。vector需要分配新的内存，至少足够容纳这些想要加入的新元素和当前已有的内容，从内存中的旧位置复制到内存中的新位置，然后删除旧位置的内存。
所以当我们尝试`push_back`一个元素的时候，如果容量用完，它就会调整大小,重新进行分配——这就是将代码拖慢的原因之一。事实上，我们需要不断地重新分配，which is a缓慢的操作，我们需要重新分配当我们要复制所有的现有元素的时候，这是我们要避免的。
事实上，这就是我们现在对于复制的优化策略：我们如何避免复制对象，如果我们处理的是vector，特别是基于vector的对象（我们没有存储vector指针，我们存储的是vector对象）。

了解自己的环境，是优化过程中最重要的事情之一。
现在我们正试图优化复制，所以我们需要知道，复制是什么时候发生的，为什么会发生。

```C++
struct Vertex
{
    float x, y, z;

    Vertex(float x, float y, float z)
        : x(x),y(y),z(z)
    {
    }

    Vertex(const Vertex& vertex)
        :x(vertex.x),y(vertex.y),z(vertex.z)
    {
        std::cout << "Copied!" << std::endl;
    }
};

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});

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

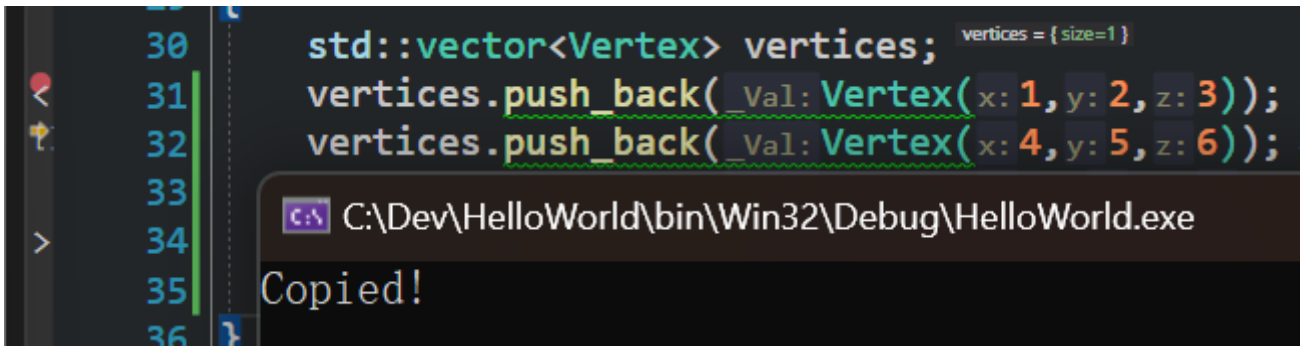
发现复制发生了3次：



改为默认构造函数的方式，并增加一个元素：

```C++
vertices.push_back(Vertex(1,2,3));
vertices.push_back(Vertex(4,5,6));
vertices.push_back(Vertex(7, 8, 9));
```

为什么C++复制了我的vertex 6次？我们进一步调试它。



我们`push_back`了一个vertex元素，有了一个复制，为什么会这样？
原因是，当我们创建vertex时，我们实际上是在主函数的*current stack frame*（当前栈帧）中构造它，所以我们是在main的栈上创建它。然后我们需要把这个刚创建的vertex从main函数里放到实际的vector中，放到vector分配的内存中。

优化1

我们所做的就是将vertex从main函数**复制**到vector类中，这是我们可以优化的第一件事。如果我们可以再适当的位置构造那个vertex，在vector实际分配的内存中，这就是优化策略一号。

优化2

Name	Value	Type
vertices	{ size=3 }	std::vector<Vertex,std::...
[capacity]	3	unsigned int
[allocator]	allocator	std::_Compressed_pair...
[0]	{x=1.00000000 y=2.00000000 z=3.00000000 }	Vertex
[1]	{x=4.00000000 y=5.00000000 z=6.00000000 }	Vertex
[2]	{x=7.00000000 y=8.00000000 z=9.00000000 }	Vertex
[Raw View]	{_Mypair=allocator }	std::vector<Vertex,std::...

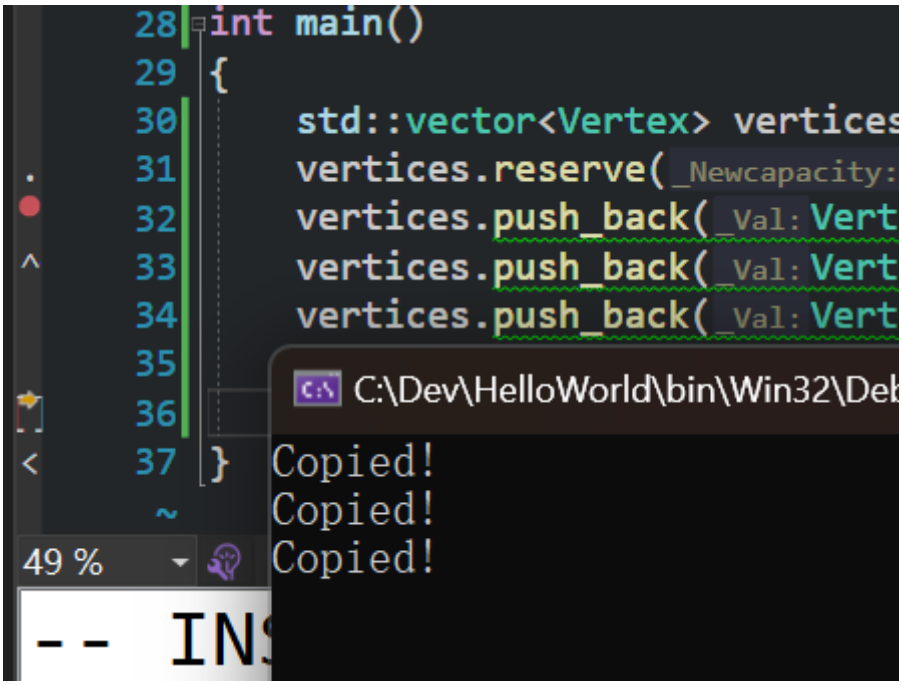
可以看到我们的vector在这里调整了两次大小，如果我们了解自己的“环境”，就是如果我们知道本身计划要放进3个vertex对象，为什么不让vector一开始就留下足够3个元素的内存，这样就不用调整两次大小了。这就是第二种优化策略。

```cpp
vertices.reserve(3);
```

这与调整大小（`resize`）或者在构造函数中传入3是不同的：

```
std::vector<Vertex> vertices( _Count: 3);
vertices.reserve( _Newcapacity: 3);
```

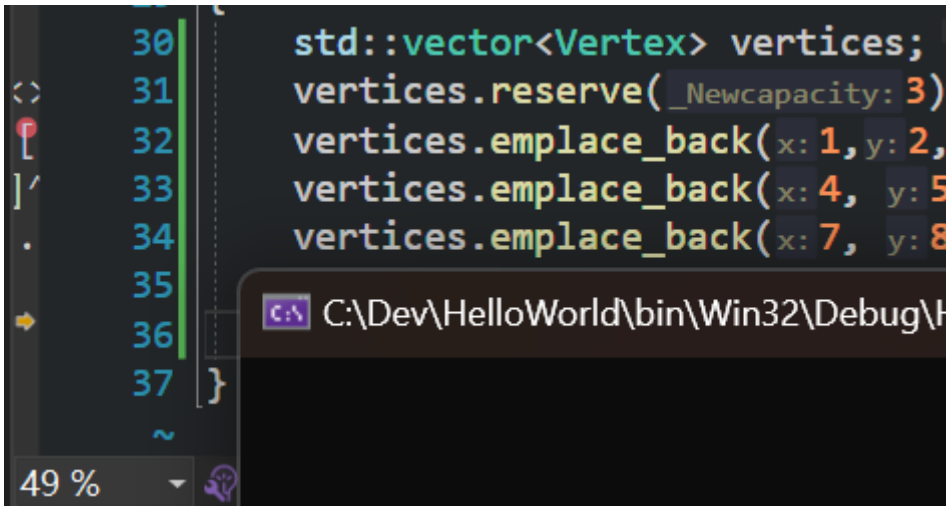
这甚至无法编译，因为这实际上不仅仅是分配足够的内存来存储3个vertex对象，它实际上会构造3个vertex对象，而我们只想有足够的内存来容纳它们。这就是`reserve`所做的，可以确保我们有足够的内存。



已经节省一半了，如果不用`reserve`的话复制次数会*exponentially*（指数倍地）增长。不过我们还可以做的更好：我们仍然得到了一次复制，因为这个vertex实际上是在main函数中构造的，然后复制到实际的vector中。我想在实际的vector中构造，我们可以使用`emplace_back`而不是`push_back`。

```cpp
std::vector<Vertex> vertices;
vertices.reserve(3);
vertices.emplace_back(1, 2, 3);
vertices.emplace_back(4, 5, 6);
vertices.emplace_back(7, 8, 9);
```

在这种情况下，不是传递我们已经构建的vertex对象，而是只是传递了构造函数的参数列表，它告诉我们的vector：在我们是的vector内存中，使用以下参数来构造一个vertex对象。



一个copies都没有了。

回顾一下我们是如何简单地优化它的：只需要知道它是如何工作的，意识到vertex对象实际上被复制了6次。写出这个优化代码也并不难，但修改后会比最初的代码运行快很多，而这甚至不需要花很长时间来优化，只需要知道实际上发生了什么、了解我们的环境、知道如何使用可用的工具来优化它。