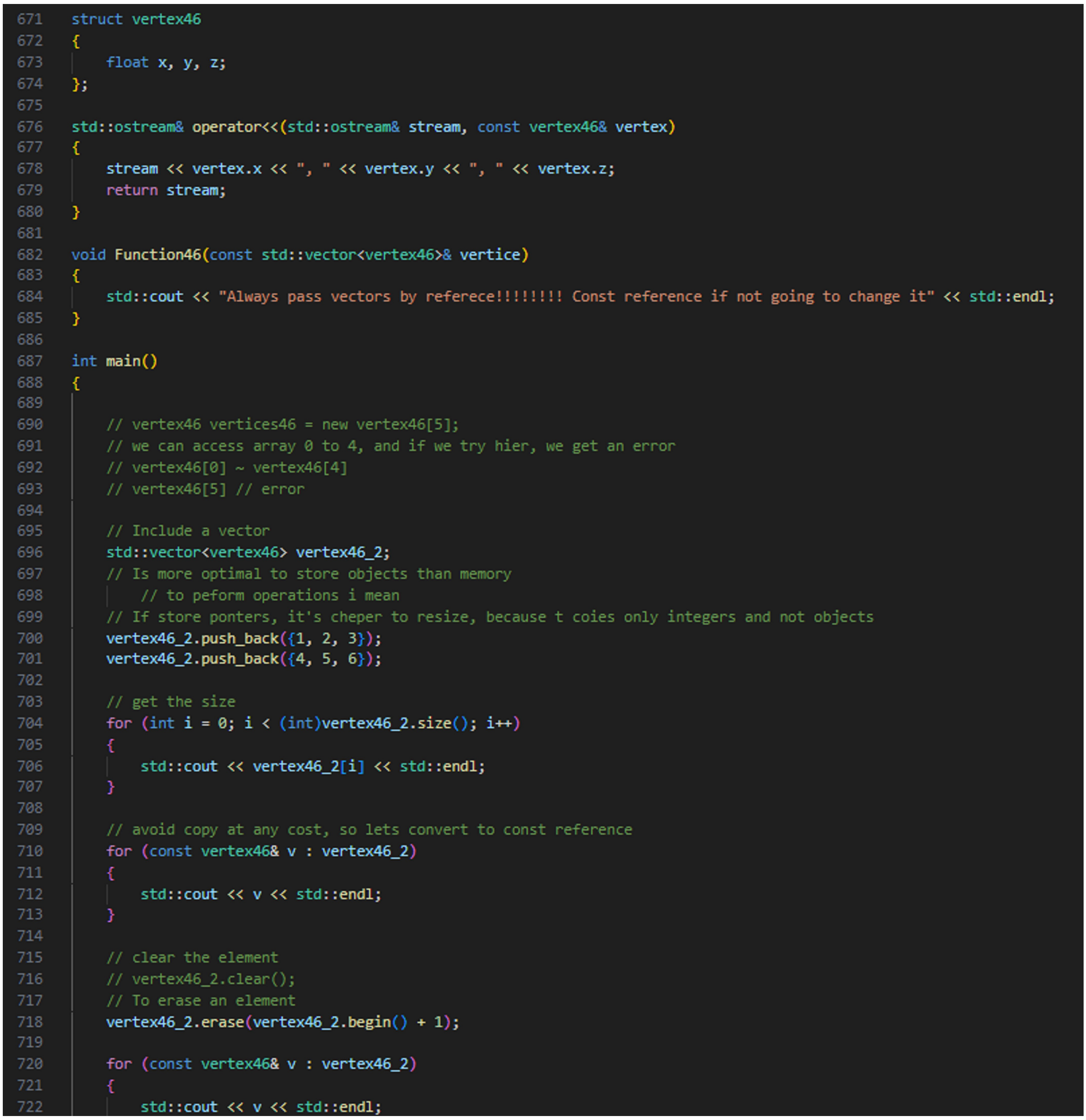
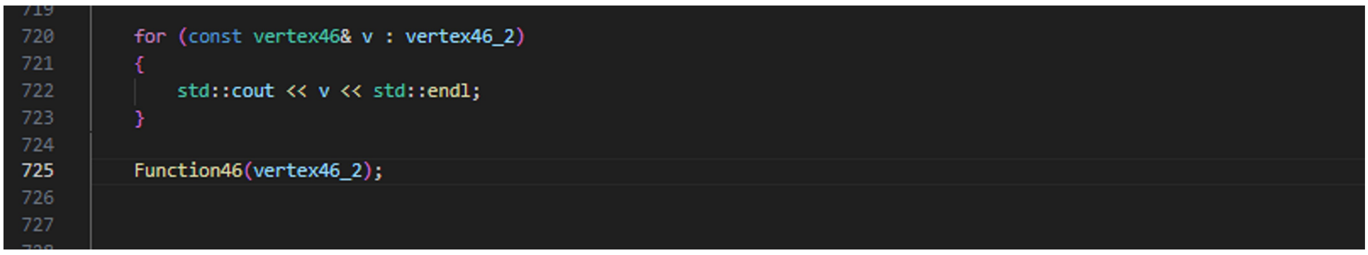
46 - Dynamic Arrays in C++ (std::vector)

quarta-feira, 12 de março de 2025 07:14

- Template array, the data type the container contains, is up to you to decide
- Don't need to use templates to use, just need to rovide the type
- Class called std::vector
  - Should be called array list, not vector... but ok
  - It's a set that doesn't enforce the type
- Unlike array, this can actually resize itself
  - Create the array and put element into it
  - We can start witout knowing how many elements are in there
- We usually get creating our own types
- Make a vector that allocates 10 elements, and we violate this size
  - It copy all elements taht is already there, and put in another place in memory, larged
  - But the thng is... it grows and copy things a lot
- We nees a way to grow, that is the motivation with vectors
  - When reached certain aamount of data, groww



```cpp
struct vertex46
{
    float x, y, z;
};

std::ostream& operator<<(std::ostream& stream, const vertex46& vertex)
{
    stream << vertex.x << ", " << vertex.y << ", " << vertex.z;
    return stream;
}

void Function46(const std::vector<vertex46>& vertice)
{
    std::cout << "Always pass vectors by referece!!!!!!!! Const reference if not going to change it" << std::endl;
}

int main()
{

    // vertex46 vertices46 = new vertex46[5];
    // we can access array 0 to 4, and if we try hier, we get an error
    // vertex46[0] ~ vertex46[4]
    // vertex46[5] // error

    // Include a vector
    std::vector<vertex46> vertex46_2;
    // Is more optimal to store objects than memory
        // to peform operations i mean
    // If store ponters, it's cheper to resize, because t coies only integers and not objects
    vertex46_2.push_back({1, 2, 3});
    vertex46_2.push_back({4, 5, 6});

    // get the size
    for (int i = 0; i < (int)vertex46_2.size(); i++)
    {
        std::cout << vertex46_2[i] << std::endl;
    }

    // avoid copy at any cost, so lets convert to const reference
    for (const vertex46& v : vertex46_2)
    {
        std::cout << v << std::endl;
    }

    // clear the element
    // vertex46_2.clear();
    // To erase an element
    vertex46_2.erase(vertex46_2.begin() + 1);

    for (const vertex46& v : vertex46_2)
    {
        std::cout << v << std::endl;
```

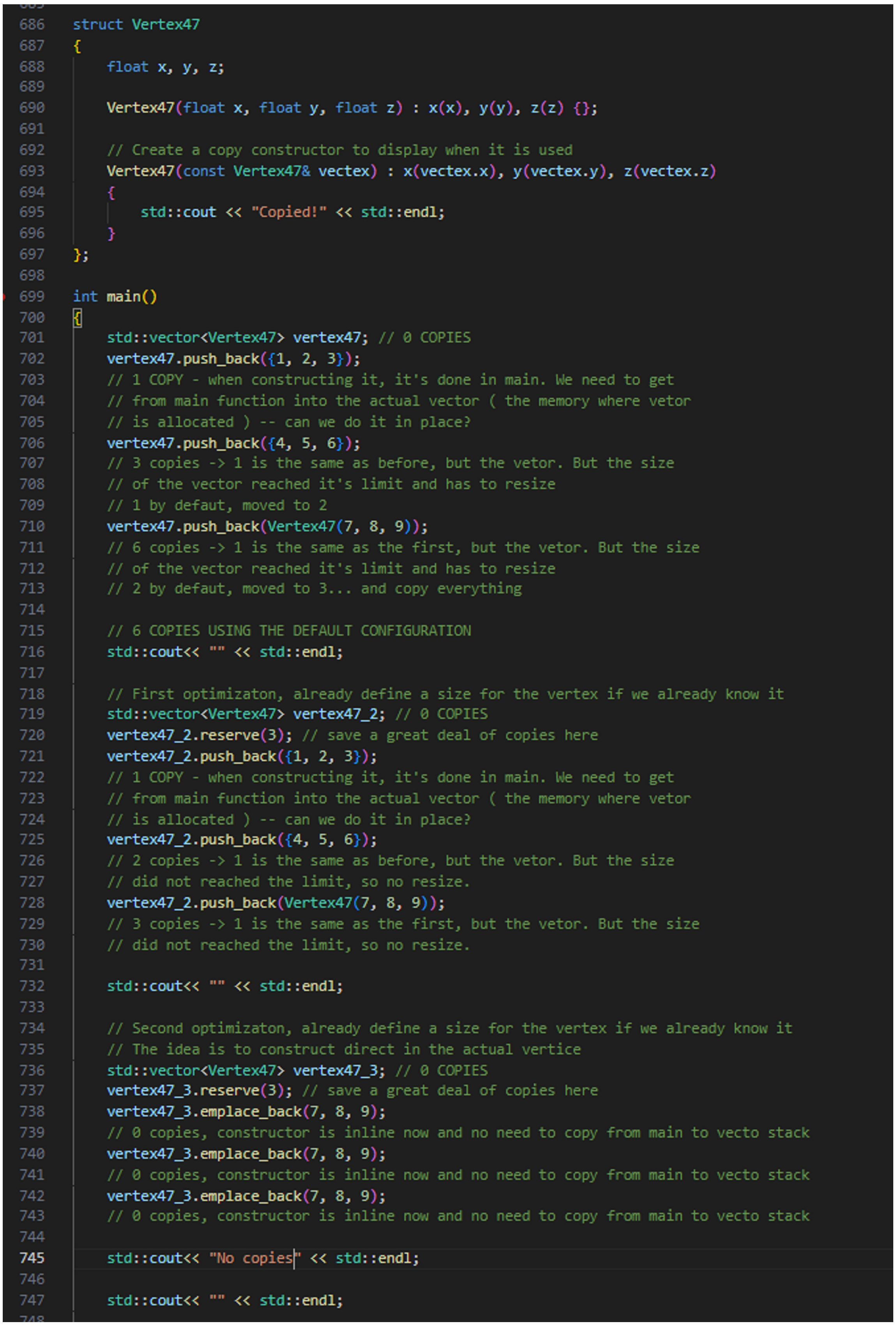


```
    for (const vertex46& v : vertex46_2)
    {
        std::cout << v << std::endl;
    }

    Function46(vertex46_2);
```

Optimizing the usage of std::vector in C++

- How to optimize vectors
  - know your environment
  - what should happen
  - Important things to know when optimizing

- For vectors, it's important to know ow it works
  - Create a vector and start to push back elements. Until it reach its lmit and has to copy everything across to a nother larger location and delete the old one
  - This is a slow operation
- How can we avoid coping the objects when dealing with copy



```cpp
struct Vertex47
{
    float x, y, z;

    Vertex47(float x, float y, float z) : x(x), y(y), z(z) {};

    // Create a copy constructor to display when it is used
    Vertex47(const Vertex47& vectex) : x(vectex.x), y(vectex.y), z(vectex.z)
    {
        std::cout << "Copied!" << std::endl;
    }
};

int main()
{
    std::vector<Vertex47> vertex47; // 0 COPIES
    vertex47.push_back({1, 2, 3});
    // 1 COPY - when constructing it, it's done in main. We need to get
    // from main function into the actual vector ( the memory where vetor
    // is allocated ) -- can we do it in place?
    vertex47.push_back({4, 5, 6});
    // 3 copies -> 1 is the same as before, but the vetor. But the size
    // of the vector reached it's limit and has to resize
    // 1 by defaut, moved to 2
    vertex47.push_back(Vertex47(7, 8, 9));
    // 6 copies -> 1 is the same as the first, but the vetor. But the size
    // of the vector reached it's limit and has to resize
    // 2 by defaut, moved to 3... and copy everything

    // 6 COPIES USING THE DEFAULT CONFIGURATION
    std::cout<< "" << std::endl;

    // First optimizaton, already define a size for the vertex if we already know it
    std::vector<Vertex47> vertex47_2; // 0 COPIES
    vertex47_2.reserve(3); // save a great deal of copies here
    vertex47_2.push_back({1, 2, 3});
    // 1 COPY - when constructing it, it's done in main. We need to get
    // from main function into the actual vector ( the memory where vetor
    // is allocated ) -- can we do it in place?
    vertex47_2.push_back({4, 5, 6});
    // 2 copies -> 1 is the same as before, but the vetor. But the size
    // did not reached the limit, so no resize.
    vertex47_2.push_back(Vertex47(7, 8, 9));
    // 3 copies -> 1 is the same as the first, but the vetor. But the size
    // did not reached the limit, so no resize.

    std::cout<< "" << std::endl;

    // Second optimizaton, already define a size for the vertex if we already know it
    // The idea is to construct direct in the actual vertice
    std::vector<Vertex47> vertex47_3; // 0 COPIES
    vertex47_3.reserve(3); // save a great deal of copies here
    vertex47_3.emplace_back(7, 8, 9);
    // 0 copies, constructor is inline now and no need to copy from main to vecto stack
    vertex47_3.emplace_back(7, 8, 9);
    // 0 copies, constructor is inline now and no need to copy from main to vecto stack
    vertex47_3.emplace_back(7, 8, 9);
    // 0 copies, constructor is inline now and no need to copy from main to vecto stack

    std::cout<< "No copies" << std::endl;

    std::cout<< "" << std::endl;
```