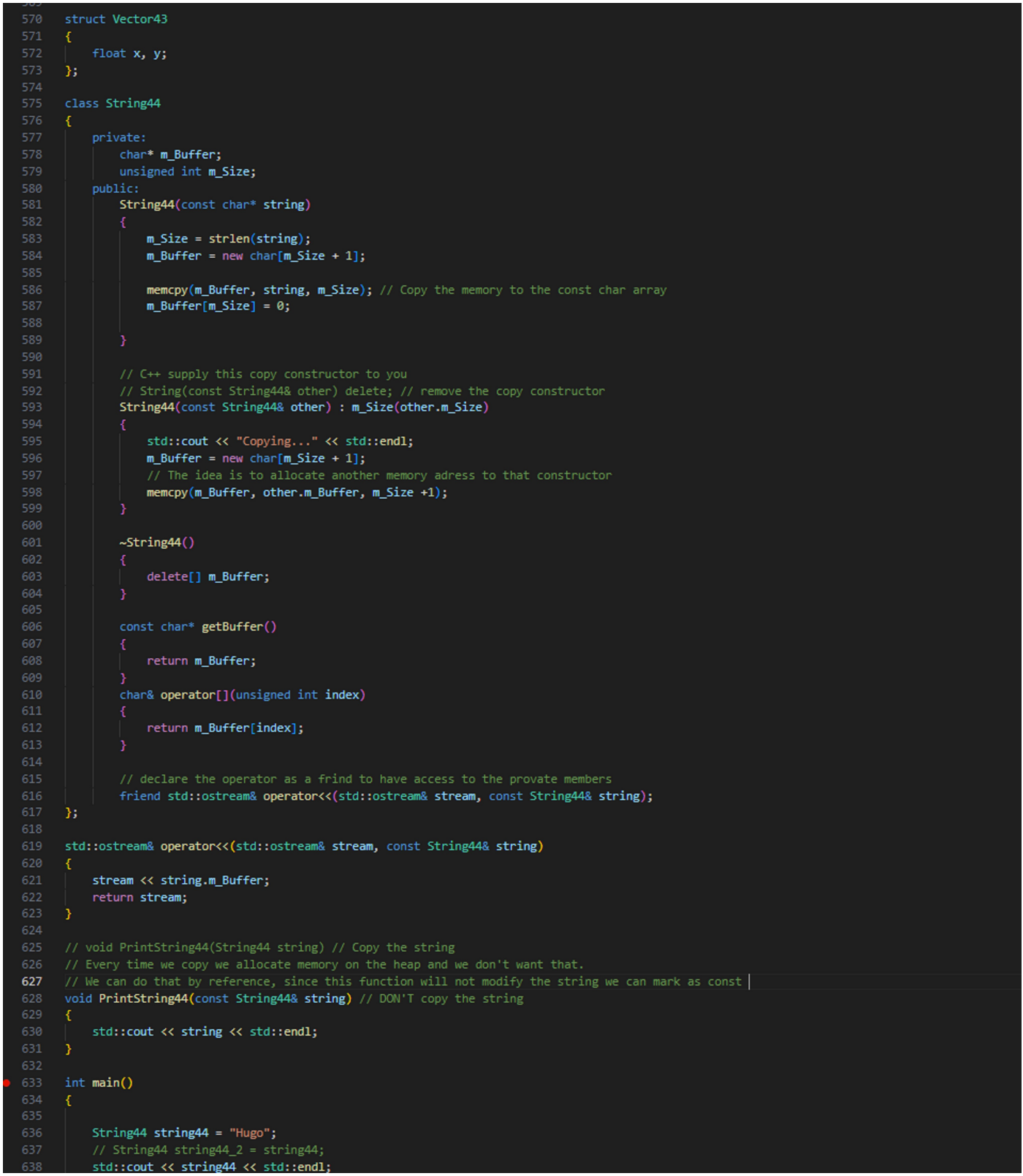
44 - Copying and Copy Constructors in C++

terça-feira, 11 de março de 2025 06:28

- When we want to copy to change something in there
- we can avoid coyng because eit takes time
- unecessary copy is bad because it wasts performance
- Allways pass objects as const reference, to avoid copies. The unction it'self can decide if we want to copy it there or not



```
570  struct Vector43
571  {
572      float x, y;
573  };
574
575  class String44
576  {
577      private:
578          char* m_Buffer;
579          unsigned int m_Size;
580      public:
581          String44(const char* string)
582          {
583              m_Size = strlen(string);
584              m_Buffer = new char[m_Size + 1];
585
586              memcpy(m_Buffer, string, m_Size); // Copy the memory to the const char array
587              m_Buffer[m_Size] = 0;
588
589          }
590
591          // C++ supply this copy constructor to you
592          // String(const String44& other) delete; // remove the copy constructor
593          String44(const String44& other) : m_Size(other.m_Size)
594          {
595              std::cout << "Copying..." << std::endl;
596              m_Buffer = new char[m_Size + 1];
597              // The idea is to allocate another memory adress to that constructor
598              memcpy(m_Buffer, other.m_Buffer, m_Size +1);
599          }
600
601          ~String44()
602          {
603              delete[] m_Buffer;
604          }
605
606          const char* getBuffer()
607          {
608              return m_Buffer;
609          }
610          char& operator[](unsigned int index)
611          {
612              return m_Buffer[index];
613          }
614
615          // declare the operator as a frind to have access to the provate members
616          friend std::ostream& operator<<(std::ostream& stream, const String44& string);
617  };
618
619  std::ostream& operator<<(std::ostream& stream, const String44& string)
620  {
621      stream << string.m_Buffer;
622      return stream;
623  }
624
625  // void PrintString44(String44 string) // Copy the string
626  // Every time we copy we allocate memory on the heap and we don't want that.
627  // We can do that by reference, since this function will not modify the string we can mark as const
628  void PrintString44(const String44& string) // DON'T copy the string
629  {
630      std::cout << string << std::endl;
631  }
632
633  int main()
634  {
635
636      String44 string44 = "Hugo";
637      // String44 string44_2 = string44;
638      std::cout << string44 << std::endl;
```

```cpp
{

    String44 string44 = "Hugo";
    // String44 string44_2 = string44;
    std::cout << string44 << std::endl;
    // std::cout << string44_2 << std::endl;
    // crashed
    // free(): double free detected in tcache 2 Aborted
    // Copyg the values to a new memory adress in memory, a direct copy.
    // the memory adress of the buffer is the same for the strings, and we are tring to release the same buffer twice and it's a problem

    // we need to allocate a new char aarray to fix the crash
    // we wANT THE SECOND MEMORY TO HAVE IT'S OWN BLOCK OF MEMORY
    // WE NEED TO PEORM A DEEP COPY, copy the whole object and not just a shallow copy
    // Write a copy constructor, which is called when coying the object. when assignng to a variable of the same type
    // Adter defining the copying constructor, no crash anymore
    String44 string44_3 = "Hugo";
    // Copy
    String44 string44_4 = string44;
    string44_4[2] = 'a';
    PrintString44(string44_3);
    std::cout << string44_3 << std::endl;
    std::cout << string44_4 << std::endl;


    // This is diferent, they are pointers and we are not copyng the vector itself
    // we are copyng the ointer, ending up with 2 pointers to the same variable
    // If accessing the pointer and change the vlue ther, both variables are affected
    // Vector43* a43_3 = new Vector43();
    // Vector43* b43_3 = a43_3;

    Vector43 a43_2 = {2, 3};
    Vector43 b43_2 = a43_2; // a copy of a43_2, bcause they are 2 variables that ocupy 2 diferent locations in memory

    b43_2.x = 5;

    int a43 = 2;

    int b43 = a43; // Creating a copy of a43, they are 2 diferent variables

    b43 = 3;

    // std::cout << a43_3 << std::endl;
    // std::cout << b43_3 << std::endl;
    std::cout << a43_2.x << std::endl;
    std::cout << b43_2.x << std::endl;
    std::cout << a43 << std::endl;
    std::cout << b43 << std::endl;

```