Overloading Copy Constructor

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

November 6, 2024

Shallow Copying and Associated Problems

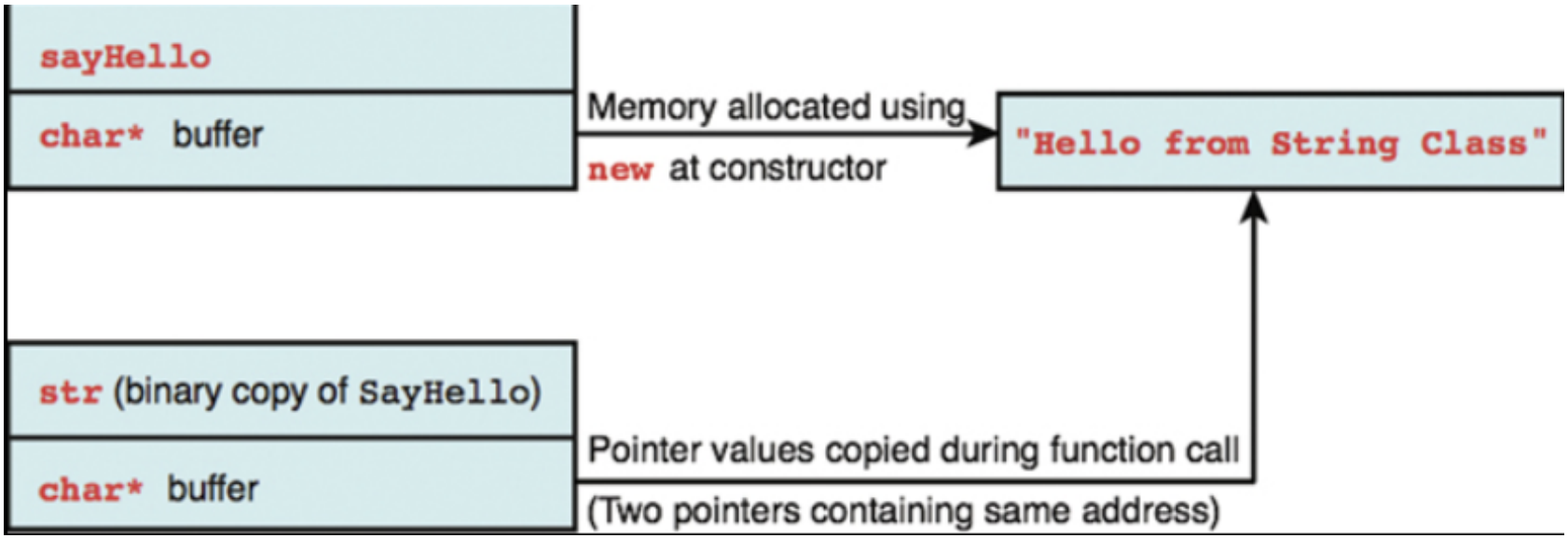
- The `MyString` contains a pointer member `buffer`.
- see ”mystring.cpp”
- This `buffer` points to a dynamically allocated memory.
- The allocation is done in the constructor using `new` and the deallocation is carried out in the destructor using `delete[]`.
- When an object of this class is copied the ”pointer member” is copied not the ”pointed memory”
- resulting in two members pointing to the same dynamically allocated buffer in the memory.
- When an object is destructed, `delete[]` deallocates the memory, thereby invalidating the pointer copy held by the other object.
- Such copies are shallow and are a threat to the stability of the program

Failure of Shallow Copying

- see ”mystring_cp.cpp”
- Why does class `MyString` that worked just fine in ”mystring.cpp” failed here?
- The only difference is that the use of the object `sayHello` of class `MyString` created in `main()` has been delegated to function `UseMyString()`
- This has resulted in object `sayHello` in `main()` to be copied into parameter `str` used in `UseMyString()`
- This is a copy generated by the compiler as the function has been declared to take `str` as a parameter by value and not by reference
- The compiler performs a binary copy of Plain Old Data such as integers, characters, and pointers to the same.

Failure of Shallow Copying

- So the pointer value contained in `sayHello.buffer` has simply been copied to `str` that is, `sayHello.buffer` points to the same memory location as `str.buffer`.
- The binary copy did not perform a deep copy of the pointed memory location, and you now have two objects of class `MyString` pointing to the same location in memory.



- Thus, when the function `UseMyString()` ends, variable `str` goes out of scope and is destroyed.

Failure of Shallow Copying

- The destructor of class `MyString` is invoked.
- It releases the memory allocated to buffer via `delete[]`
- This call to `delete[]` invalidates the memory being pointed to in copy `sayHello` contained in `main()`
- When `main()` ends, `sayHello` goes out of scope and is destroyed.
- Now a call is made to `delete[]`, on a memory address that is no longer valid
- This double delete is what results in a crash.

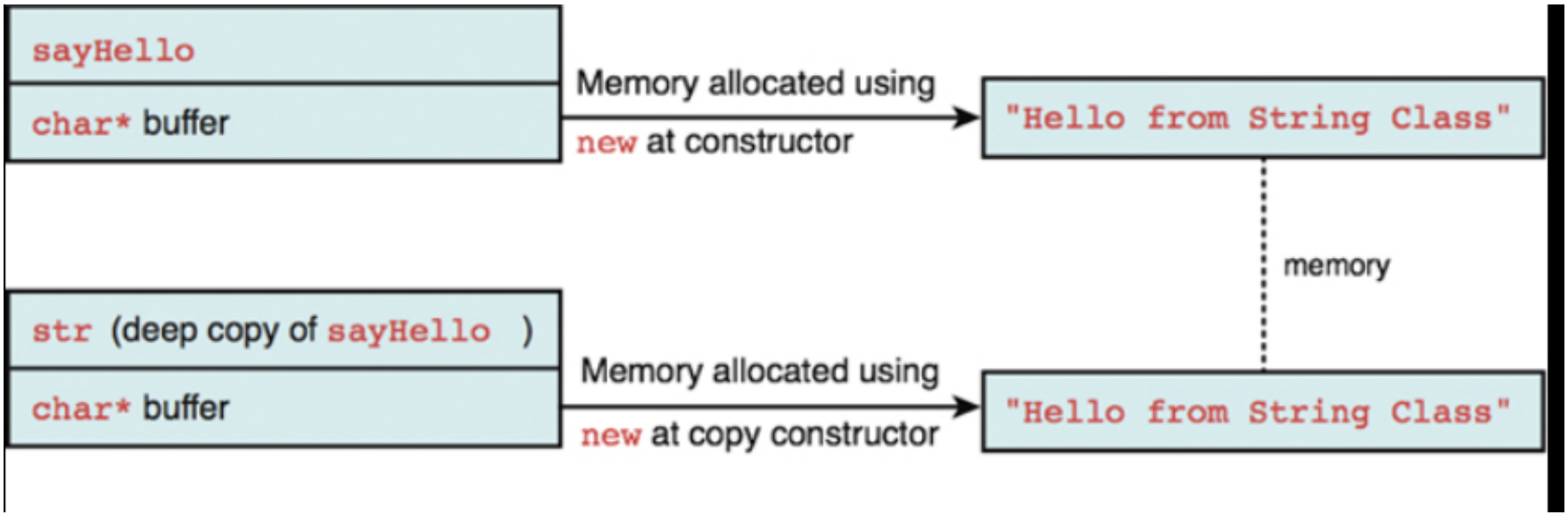
Deep Copy Using a Copy Constructor

- The copy constructor can be overloaded and supplied by the designer, although it is by default available from the compiler.
- A copy constructor takes an obejct of the same class by reference as a parameter.
- This parameter is an alias of the source object and is the handle we have in writing out custom copy code.
- We would use copy constructor to ensure a deep copy of all buffers in the source
- see ”deepcopy.cpp”

How Deep Copy Works

- Most of the code is similar to the previous `MyString` code, except for the new addition of a copy constructor.
- In main Creating `sayHello` results in the first line of output that comes from the constructor of `MyString`.
- It also displays the memory address of that `buffer` points to.
- `main` then passes `sayHello` by value to the function `UseMyString()`
- apparently, this results in automatic invocation of the copy constructor.
- In the copy constructor it is a deep copy that is performed, where the content being pointed to is copied to a newly allocated buffer that belongs to this object

How Deep Copy Works



- Now both the objects don't point to the same address, unlike the previous case.
- that is, two objects don't point to the same dynamically allocated memory address
- As a result when `UseMyString()` returns, it destroys the object in the memory address that was created by the copy constructor

Why Deep Copy Works

- In doing so, it does not touch memory that is being pointed to by `sayHello` in `main()`.
- So, both functions end and their respective objects are destroyed successfully and peacefully without the application crashing.

DOs

- DO always program a copy constructor and copy assignment operator when your class contains raw pointer members (`char*` and the like).
- DO always program the copy constructor with a `const` reference source parameter.

Destructor

- A destructor is also a special member function like a constructor.
- Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde $(\sim)$ symbol.
- It is not possible to define more than one destructor.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.