Chris Culling

Lab for **assignment 5**

19 Nov 2024

Constructors

`String(const char* value)`

First goal was to make the `String(const char* value)` and printing it to the terminal using the provided `operator<<` function.

> [!NOTE] It clicked implementing `operator<<` that in order to implement a `friend` function declared in a class in a header file, you need to drop the `MyClass::` prefix because the friend is *not* a member of the class.

1. I used `strlen()` to get the length of the `char* value` and assigned that value to the `String`'s private `size` variable—representing the amount of characters in the string (excluding the `\0` null byte)
2. I simply assigned `String`'s private `capacity` variable to be `= size + 10`
3. I then used `capacity` to initialize `text(new char[capacity] {})` and then used `std::memcpy` and `size` to copy over the content from `char* value` to my `char* text` valuable.

> [!NOTE] I've learned from **Twitch** chat that curly brackets at the end of array declarations *calls the array constructor*, which also means they won't contain garbage content lying around in memory.

Before discovering `std::memcpy` (cppreference, 2024) I tried using `std::strcpy` or iterating through the `char* value` and assigning each character in `text` individually which at the time became very non-functional when I wrote poor implementations of the copy-constructor and the assignment operator.

main.cpp (for personal testing and debugging)

```
String foo("FOO"), bar("BAR"), pie("PIE");
cout << "foo: " << foo << endl;
cout << "bar: " << bar << endl;
cout << "pie: " << pie << endl;
```

output

```
foo: FOO
bar: BAR
pie: PIE
```

Copy-constructor and Assignment Operator

The copy-constructor and assignment operator works similarly. Due to my misunderstanding some inner workings of C++, my copy-constructor at one point consisted of just the line `*this = other`; The thinking was that this would call the custom assignment operator and do everything I'd intended there instead of (almost) duplicating code.

Assignment operator

1. Delete `text` (if it's not already capable of holding the right hand side value `other.text` within its own capacity) and allocate a new, bigger spot on the heap.
2. Re-initialize the `String` variables using `other`'s variables and call `std::memcpy`

```
if(capacity < other.size)
{
    delete[] text;
    this->capacity = other.capacity;
    text = new char[capacity];
}

this->size = other.size;
std::memcpy(text, other.text, size);
```

The copy-constructor is not much different aside from not having anything to delete; It always initializes the new `String` variables with `other`'s variables.

main.cpp (for personal testing and debugging)

```
String foo("FOO"), bar("BAR"), pie("PIE");
cout << "foo: " << foo << endl;
cout << "bar: " << bar << endl;
cout << "pie: " << pie << endl;
String foo2(foo);
cout << "foo2: " << foo2 << endl;
foo = pie;
pie = bar;
cout << "foo: " << foo << endl;
cout << "bar: " << bar << endl;
cout << "pie: " << pie << endl;
cout << "foo2: " << foo2 << endl;
```

output

```
foo: FOO
bar: BAR
pie: PIE
foo2: FOO
foo: PIE
bar: BAR
```

```
pie: BAR
foo2: FOO
```

`push_back` and `reserve`

`push_back` simply compares `size` and `capacity`, calls `reserve` whenever needed, and adds the new character while incrementing `size`; `text[size++] = character;`.

`reserve` is similar to our assignment operator in the sense that it **deletes** our `text` and allocates space for a new, bigger array on the heap. The key difference is that `reserve` allocates the space in a `char* newData` variable *first*, and copies the `text` content over before deleting so as to preserve the data. This is obviously not an issue in the assignment operator logic where the data is meant to be replaced.

main.cpp (for personal testing and debugging)

```
// test push_back
for (size_t i = 0; i < 20; i++)
    foo2.push_back('A');

cout << "foo2: " << foo2 << endl;
```

output

```
foo2: FOOAAAAAAAAAAAAAAAAAA
```