

Copy Constructor and Assignment Operator

```
class String
{
    public:
        String(const char* value = "");
        ~String();

    private:
        //Class invariant:
        //    m_text points to a dynamically allocated array of m_len+1 chars
        //    m_len >= 0
        //    m_text[m_len] == '\0'
        char* m_text;
        int m_len;
};

String::String(const char* value)
{
    m_len = strlen(value);
    m_text = new char[m_len + 1];
    strcpy(m_text, value);
}

String::~~String()
{
    delete [] m_text;
}
```

```
void f(String t)
{
```

```

}

void g()
{
    String s("Hello");

    f(s);
} //when we leave g(), s goes away, but "Hello" is still there

char* somefunc();

int main()
{
    String x(somefunc());

    String y; //String y("");

    g();
}

```

What happens if `somefunc()` returns a null pointer so that the `String()` constructor is initialized with it?

The first thing that goes "wrong" is `strlen()`.

If we enter the constructor, and the parameter value is `NULLPTR`, it sets the value instead to an empty string.

We can make `value` itself point to another pointer. `const char* value` simply means that `value` points to a constant, so the pointer stored in `value` can't be changed. We can change `value`, to be clear.

```

String::String(const char* value)
{
    if(value == nullptr)
        value = "";

    m_len = strlen(value);
    m_text = new char[m_len + 1];
    strcpy(m_text, value);
}

```

How an object is constructed:

1. -----(We'll see this one later) -----
2. Construct the data members, consulting the member initialization list. If a member is not listed, if it is of
 - builtin type, it is left uninitialized
 - class type, it is default-constructed (if no default ctor, error)
3. Execute the body of the constructor

How an object is destroyed:

1. Execute the body of the destructor
2. Destroy the data members
 - builtin type, do nothing
 - class type, call its destructor
3. -----(We'll see this one later)-----

Copy constructor

```
class String
{
    public:
        String(const String& other); //copy constructor
};
```

```
String::String(const String& other)
{
    //m_len = other.m_len;
    //m_text = other.m_text; //wrong!!!!!!
    //m_text = new char[m_len+1];
    //strcpy(m_text, other.m_text);
    //-----

    m_len = other.size();
    m_text = new char[m_len+1];
    for (int k = 0; k != m_len; k++)
        m_text[k] = other.charAt(k);
    m_text[m_len] = '\0';
```

```
}
```

Isn't `m_text` private?

Well, what does it mean to be private?

One possible meaning would be that if a member is declared private, only member functions of the member object that the `this` pointer points to can use it.

The actual meaning would be that even if a member is declared private, other member objects can use it.

```
class X
{<-----private
    ...
};
struct X
{<-----public
    ...
}
```

Ordinary C structure behave like they do in C.

```
void f(String t)
{
    String u("Wow");
    ...
    u = t;
    ...
}
```

Initialization is not the same thing as assignment.

```
String& String::operator=(const String& rhs)
{
    if(this != &rhs)
        delete [] m_text;
    m_len = rhs.m_len;
```

```
        m_text = new char[m_len+1];

        strcpy(m_text, rhs.m_text);
    }

    return *this;
}
```

```
if(this != &rhs)
{
    String temp(rhs);    //copy and swap
    swap(temp);
}
```