

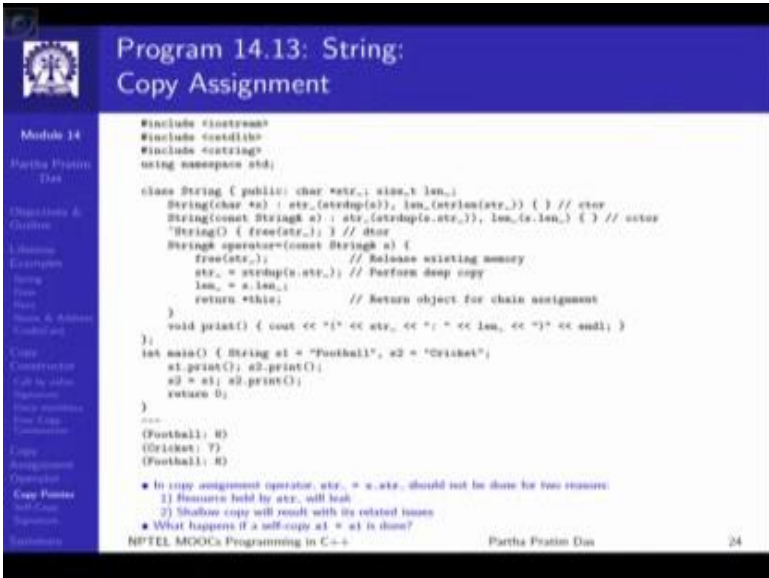
**Programming in C++**  
**Prof. Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 28**  
**Copy Constructor and Copy Assignment Operator (Contd.)**

Welcome back to module 14 of Programming in C++. We have been discussing about copy, we have discussed about copy construction at depth and we have introduced what is copy assignment operator.

To quickly recap, we do a copy construction when you want to make a clone of an object which does not exist and we do copy assignment, when we have an existing object and want to copy in other object of the same type into this existing object. We have seen that a copy assignment can be defined as a function operator function and it takes the parameter as a constant reference of the class and it returns a non-constant reference of the same class, it may return a constant reference of the class as well.

(Refer Slide Time: 01:19)



**Program 14.13: String: Copy Assignment**

```
#include <iostream>
#include <string>
using namespace std;

class String { public: char *str_; size_t len_;
String(char *s) : str_(stdup(s)), len_(strlen(str_)) {} // ctor
String(const String& s) : str_(stdup(s.str_)), len_(s.len_) {} // copy
String() { free(str_); } // dtor
String& operator=(const String& s) {
    free(str_); // Release existing memory
    str_ = stdup(s.str_); // Perform deep copy
    len_ = s.len_;
    return *this; // Return object for chain assignment
}
void print() { cout << "s" << str_ << " : " << len_ << " " << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
s1.print(); s2.print();
s2 = s1; s2.print();
return 0;
}

//
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- In copy assignment operator `str_ = s.str_;` should not be done for two reasons:
  - 1) Memory held by `str_;` will leak
  - 2) Shallow copy will result with its related issues
- What happens if a self-copy `s1 = s1` is done?

NPTEL MOOCs Programming in C++ Partha Pratim Das 24

Now, we will look into some of the more tricky areas of copy assignment. So, you recall that in terms of copy construction, you have already explained the notion of shallow

copy and deep copy and now we will have; we will see that the consequences of shallow copy and deep copy also percolates into copy assignment. So, particularly focus this is a string example and focus on the copy assignment operator.

Now, what you are trying to do, you are trying to. So, this is your, let me just draw the two objects. This is s 1, this is s 2. So, this is football. So, whatever it will have, it will have a length 8. This is cricket; this will have length 7. Now, I am trying to copy. So, I am trying to do s 2 assign s 1. So, if I copy this, naturally I will need to; while I do copying I know that this string will have to be copied here and we know there are two options to that. One is to copy the pointer; other is to actually copy the object. So, we would like to do deep copy, in this case we are doing a deep copy. So, you have making strdup of a parameter s, which is basically strdup of s 1.

Now, when you assign that to str then what will happen? Simply, so another football has got created, strdup we have done duplicate. So, another football has got created. Now, if I put this pointer into str, naturally I will lose this pointer and there will be no way to retrieve this string any further. So, before I can do this, I have to free up, otherwise the resource will leak, memory will leak. So, this is a critical point. I will first have to free this up and then I am doing a string copy. So, as I free this up this is gone, I do a strdup. Now, I have a new football pointed to here, this will get copied. So, this will become 8 as in here at the objective is returned as it was done in the last case.

Just note that this is basically returns a starts this is a current object. So, it returns that because it has to return the object to which the assignment is happening. So, this object can now be used for chain assignment, as I have explained. This can be used in the chain assignment as I have explained already. So, this is how a copy can be done for a string with deep copy that is similar strategy can be used whenever we have pointer members in the object.

(Refer Slide Time: 05:01)

```
#include <iostream>
#include <string>
#include <string>
using namespace std;

class String { public: char *str; size_t len;
String(char s) : str(&strdup(s)), len(strlen(s)) {} // ctor
String(const String& s) : str(&strdup(s.str)), len(s.len) {} // copy
String() { free(str); } // dtor
String operator=(const String& s) {
free(str); // Release existing memory
str = &strdup(s.str); // Perform deep copy
len = s.len; // Return object for chain assignment
return *this;
}
void print() { cout << "S" << str, << " " << len, << "\n"; }
};

int main() { String s1 = "Football", s2 = "Cricket";
s1.print(); s2.print();
s1 = s2; s1.print();
return 0;
}

//
(Football) 8
(Cricket) 7
(FFFFFF) 8 // Garbage is printed

• For self-copy str, and s.str, are the same pointers
• Hence, free(str) first releases the memory, and then strdup(s.str) tries to copy from released memory
• This may crash or produce garbage values
• A self-copy must be detected and protected

NETEL MOCCA Programming in C++ Partha Pratim Das 25
```

Now, let's look into a very small, but dangerous issue with the code that we have already seen. This is the code, exactly the code that you have seen, the only difference being earlier we were copying s 2 to s 1. Now, I have copied s 1 to s 1. Now, you can very legitimately ask me, as to why should somebody write this kind of a code that answers to that what, one is what if somebody writes. We have to know, what is going to happen. The other issue is that not always the code will look like this, for example, it could be that I have a string, I have a reference to s 1 that is done somewhere; I do not know where this is done.

This may have been done in some other function, in some other class whatever it is come and now I am doing s 1 assigned r. Syntactically, looking at the code it does not look like a self copy, but it actually the self copy. So, self copy is a potential situation that we must look into now certainly there are issues that the reason we are trying to look into this.

So, look into this is the self copy. So, this is what I have, this is my s 1, this is my string, my s 1 is football. So, I have football and a weight here. Now, I am doing s 1 assign this one. So, what will happen? This will execute first, this is my s 1. So, this will free this up. Now, this will try to do this, that is it will try to take this object s dot str, make a copy make a copy into something and then assign it here. Now, this object is already gone this

is been freed up. So, what you make copy of here is not known, is not question mark it is just not known, it is something invalid and then so on.

So, quiet expectedly what you get after the copy, when you print it after the copy you get a garbage, I got a garbage, while I was running it, but it is quite possible that instead of a garbage, it could be a crash because it just depends on what memory is getting violated. So, self copy with pointer type of data is something which is quite, which could prove to be quite difficult to deal with. So, we will have to do something about that.

(Refer Slide Time: 08:10)



The slide displays the following C++ code for a String class:

```
#include <iostream>
#include <string>
#include <string>
using namespace std;

class String { public: char *str; size_t len;
    String(char s) : str, (strlen(s)), len, (strlen(s)) {} // ctor
    String(const String& s) : str, (strlen(s.str)), len, (s.len) {} // ctor
    String() { free(str); } // dtor
    String operator=(const String& s) {
        if (this != &s) {
            free(str);
            str = strdup(s.str);
            len = s.len;
        }
        return *this;
    }
    void print() { cout << "(" << str, << ", " << len, << ")" << endl; }
};

int main() { String s1 = "Football", s2 = "Cricket";
    s1.print(); s2.print();
    s1 = s1; s1.print();
    return 0;
}

//
// (Football): 0)
// (Cricket): 1)
// (Football): 0)
```

Below the code, there are two bullet points:

- Check for self-copy (this != &s)
- In case of self-copy, do nothing

The slide footer includes the NPTEL MOOCs Programming in C++ logo, the name Partha Pratim Das, and the slide number 26.

So, the way we handle this and that is very typically is all that you want to say that if I am doing a self copy, if I am doing this, then all the time need to tell my copy assignment operator is that do not copy. If you are doing a self copy then all that I would like to tell is do not copy because it is bypass, it is a same object. So, the rest of the code remains same, but all that I add is check, if it is a same object. How do I check if it is a same object? Just understand s 1 is being assigned s 1. So, it is s 1 dot operator assignment s 1, this become s and this is the object on which the invocation has happen, so this is star this.





So, it can be, it is already existing initialized then it has to be replaced by the members of the object being copied from, and in copy assignment operator self copy could be a significant issue and needs to be taken care of and again please keep in mind that this is not explicitly written in the slide, but please keep in mind that if the user does not provide a copy assignment operator, but uses it in the program then the compiler will provide a free copy assignment operator, which again like the free copy constructor will again just do a bit wise copy without considering what specific requirements the copy may have.

So, it is always advised that like the constructor, you should also provide the copy constructor and the copy assignment operator whenever you are designing a class where copies are possible or where objects are likely to be passed to functions in call by value.

In specific terms, we have also seen here the notions of the deep and shallow copy with pointers. Please remember, shallow copy will just copy the pointer. So that after a shallow copy more than one pointer points to the same object and deep copy do not copy the pointer it copies the pointed object. Therefore, after deep copy the 2 pointers point to two different copies after possibly this originally same object, but they become different objects.

So, deep copy and shallow copy will have to be used naturally judiciously. Certainly, if it is not required, we will not try to do deep copy because it will involve the copying of the pointed data which may be costly because that will again need copy construction by recursive logic, but in terms of safety using deep copy is often more safe compared to using shallow copy.