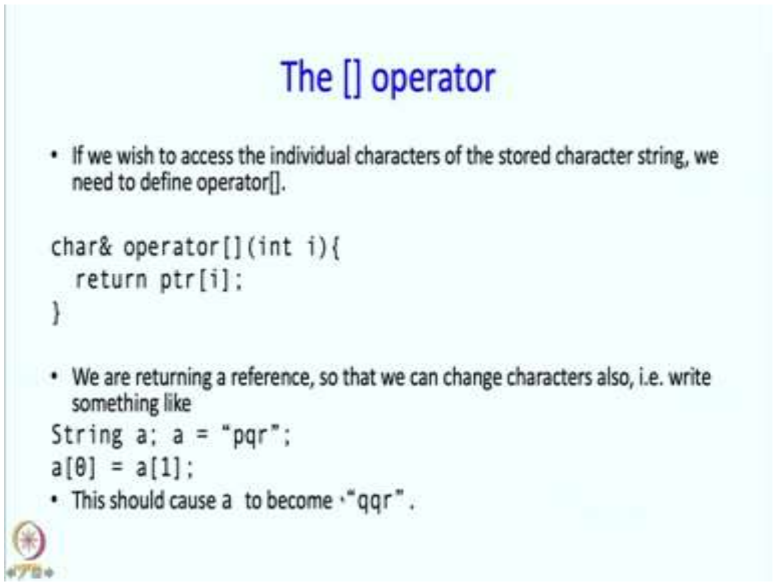
An Introduction to Programming Through C++

Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Lecture-22 Part-06
Representing variable length entities
Implementing a class with automated memory management 2

Welcome back! In the last segment we saw implementations of two assignment operators for our class string. In this segment, we are going to look at implementation of other operators. So, first the Square bracket or the Indexing operator.

(Refer Slide Time: 00:37)



The [] operator

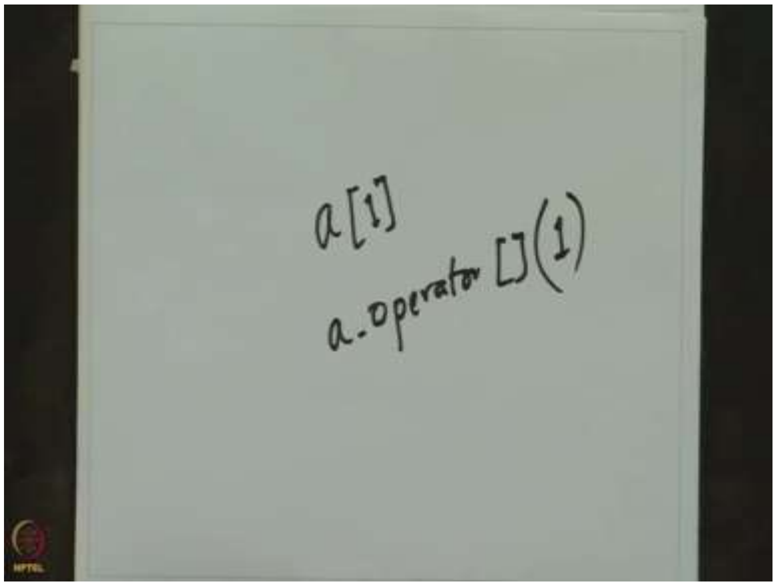
- If we wish to access the individual characters of the stored character string, we need to define operator[].

```
char& operator[](int i){
  return ptr[i];
}
```

- We are returning a reference, so that we can change characters also, i.e. write something like

```
String a; a = "pqr";
a[0] = a[1];
```

- This should cause a to become "qqr".



$a[1]$

$a.\text{operator}[](1)$

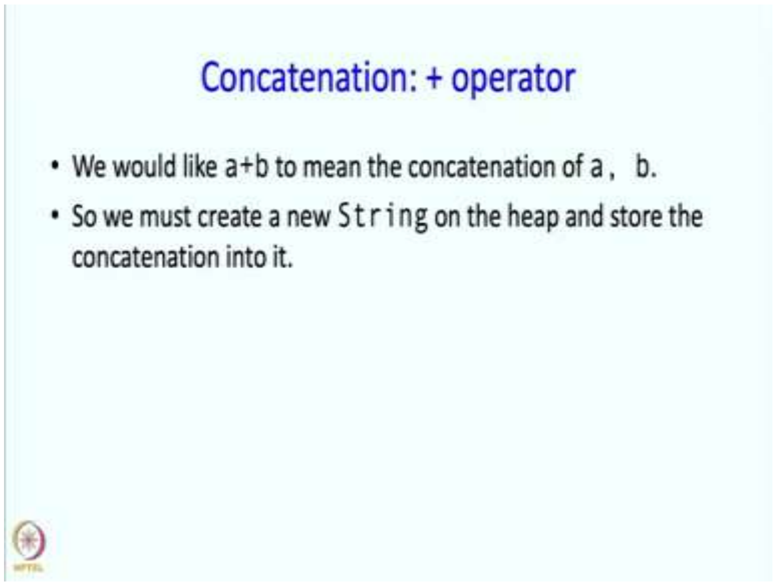
So, if we wish to access the individual characters of the stored character string, we need to define this. So, as we know if what has been written in the program is a[1], this is read by C++ as a.operator[], operator square bracket operating on 1.

So, this operator must be overloaded and that overloading is quite simple. So, the receiver has a certain string and we want character 1 from it and where is the string of the receiver? It is at a.ptr, so we just have to get character one from a.ptr, that is that is about it. So, we do not have to write a in this because the receiver is implicit. So, this is going to be the receiver's ptr and we are going to take a displacement of i from it.

So, that is about it, and notice that we are returning a reference. So, we are returning a reference because that way, we can change the character as well. So, we would be able to write something like, suppose we have a and a is pqr then we can write a[0]=a[1]. So, this a of 0 will also be translated or will also be interpreted by this operator. So, this will return reference to a[0] , so a[0] is a variable and it will be a reference so it will mean that variable itself, it will not mean the value. So, that variable itself will appear here so to say and therefore we can store into it.

So, that is why we are returning a reference rather than a value. So, if we do this, then if we have this defined and if we write this and we are allowed to write this, then this will cause a to become pqr.

(Refer Slide Time: 03:05)

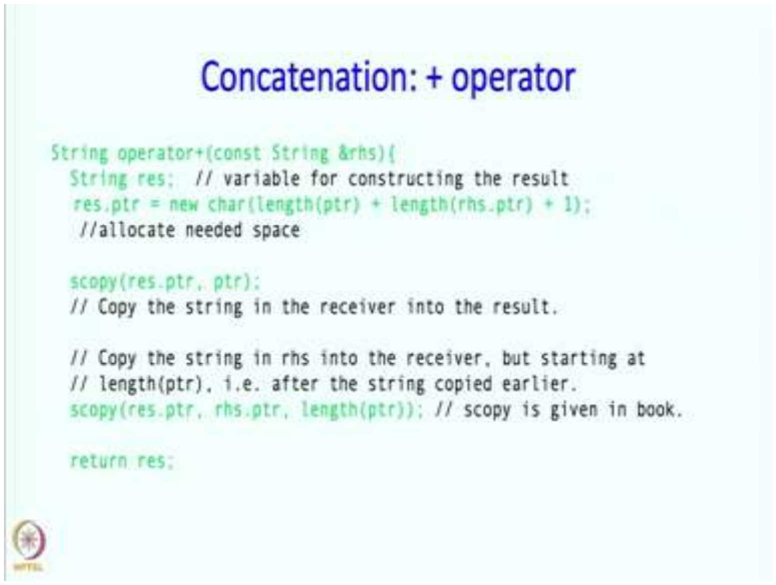


Concatenation: + operator

- We would like a+b to mean the concatenation of a , b.
- So we must create a new String on the heap and store the concatenation into it.

Now, there is an interesting operator Concatenation so for which we are going to overload our plus. So, we would like a+b to mean the Concatenation of a and b. So, you must create a new string first of all on the heap and store the Concatenation into it.

(Refer Slide Time: 03:12)



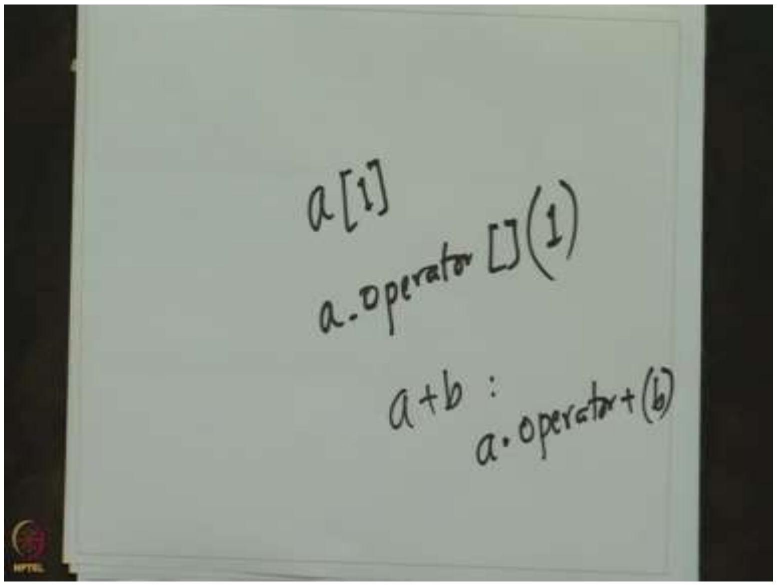
Concatenation: + operator

```
String operator+(const String &rhs){
  String res;  // variable for constructing the result
  res.ptr = new char(length(ptr) + length(rhs.ptr) + 1);
   //allocate needed space

  scopy(res.ptr, ptr);
  // Copy the string in the receiver into the result.

  // Copy the string in rhs into the receiver, but starting at
  // length(ptr), i.e. after the string copied earlier.
  scopy(res.ptr, rhs.ptr, length(ptr)); // scopy is given in book.

  return res;
```



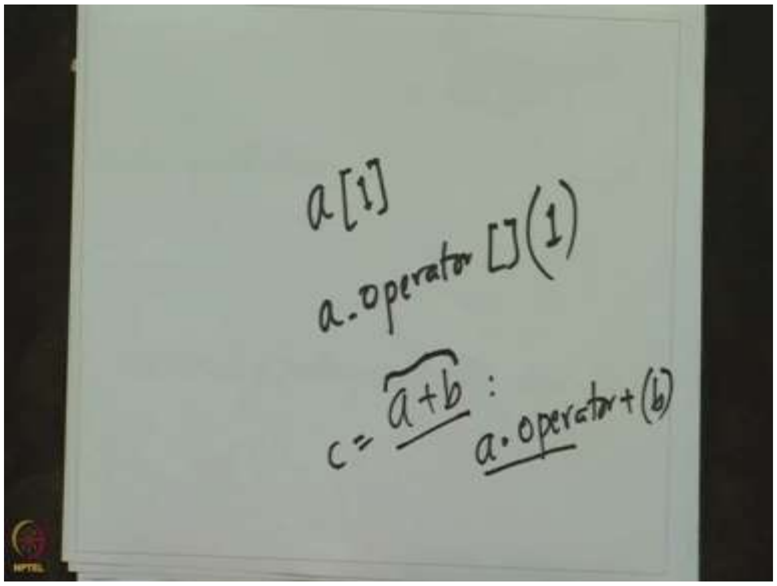
So, how does that work? So, we are going to return a string this time, so the result is going to be a string and our operator plus is going to be overloaded. So again, let me just remind you, a+b becomes a.operator+(b), so therefore we need to redefine or we need to overload this function operator plus. So, first we will create this string which is which is going to eventually contain the result and into that string we are going to store the result but this string has its own pointer and that pointer must point to the heap and how many elements, how long an array does it need on the heap.

This array must contain the string from a as well as the string from b or looking at it here it should contain the string from the receiver as well as the string from the right hand side. So, the space needed must be the length of the string pointed to by ptr, as well as the length of the string pointed to by the rhs.ptr and a 1 so that we can append the null character to it. So, we have allocated the space needed by writing this new thing and now we just have to copy.

So, this s copy that we had earlier is going to copy whatever is in the receiver into this result, the result area that we have created. This is created on the heap ofcourse and then we have to copy the rhs string as well so the same scopy is going to be used but we have to tell scopy that, look I do not want it copied from the very beginning as what this would be doing, but we want to copy it from index length(ptr). So, we want to copy from res.ptr index length of the ptr, length of the string that we just copied. We are going to call s copy but this time we are going to give three arguments.

So, this is the destination , this is the source , what we want copied and this is the displacement in the destination from where the copying has to begin and you can write this but this function is also given in the book in that same chapter 15. and finally res will be returned.

(Refer Slide Time: 06:04)



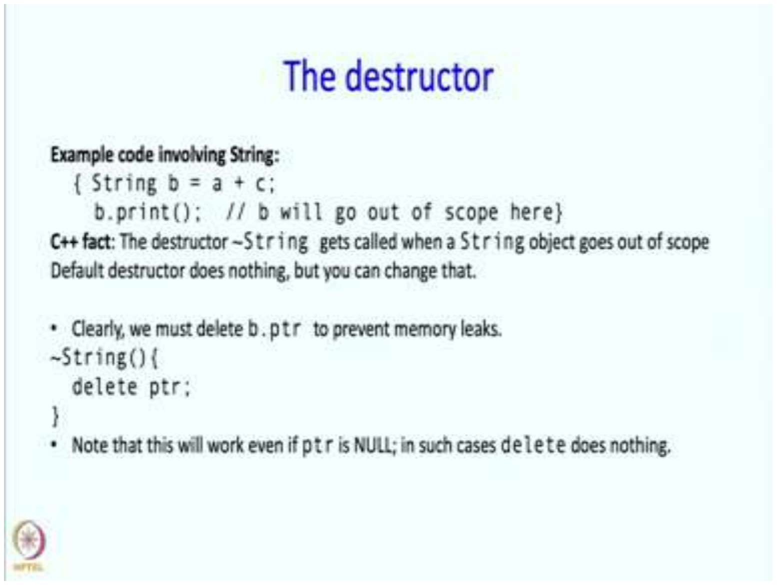
a[1]

a.operator[](1)

$c = \underline{\overset{\frown}{a+b}}$: $\underline{a.\text{operator}}+(b)$

So, the result of this will effectively be replacing this expression, so there will be a temporary which will be replacing this and you can do whatever you want. So, you might have c=a+b so that then the temporary will be assigned to c but this evaluation will result in a temporary which contains the Concatenation, so that is it. So, that is the Concatenation operator.

(Refer Slide Time: 06:39)



The destructor

Example code involving String:

```
{ String b = a + c;
  b.print();  // b will go out of scope here}
```

C++ fact: The destructor ~String gets called when a String object goes out of scope
Default destructor does nothing, but you can change that.

- Clearly, we must delete b.ptr to prevent memory leaks.

```
~String(){
  delete ptr;
}
```

- Note that this will work even if ptr is NULL; in such cases delete does nothing.

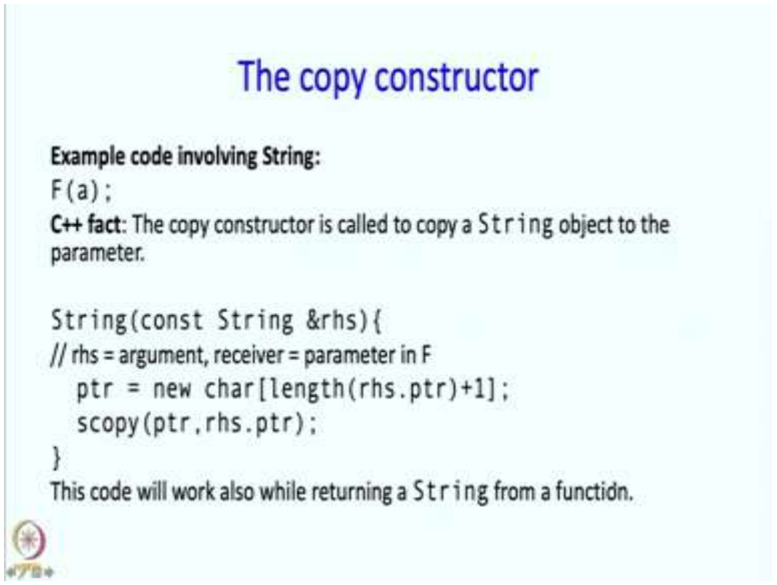
Now, we need the destructor as well and we talked about it but let us just do this again. So, suppose we have this code which we had in the program that we have written. So, at this point string b is created and at this point string b goes out of scope. So, when string b goes out of scope, C++ is automatically going to call the destructor ~String , so I should really say the destructor in the string class , so that will get called and it will be called on the receiver so it will be ~b actually the call will look like ~b.

So, what do we want to happening over here. So, by the way there is a default destructor and the default destructor does nothing and we are going to change it. So, we should delete b.ptr to prevent memory leaks, so how do we do this? So, our class is going to contain this , the our destructor is going to have delete ptr inside it, that is about it. So, this is what will happen if b goes out of the scope over here then it is as good as calling ~b and this is the call that is going to happen.

So, this is the body of the call and b is going to be the receiver and so ptr, the ptr of b is going to be deleted. So, whatever memory b was pointing to will get deleted exactly as we wanted and as I said earlier this works even if ptr is null and in this case the delete does nothing.

We also need to write the copy constructor. So, remember that in our program we were calling some function f with a string object a as the argument. So, how does C++ handle it?

(Refer Slide Time: 08:50)



The copy constructor

Example code involving String:

```
F(a);
```

C++ fact: The copy constructor is called to copy a `String` object to the parameter.

```
String(const String &rhs){
// rhs = argument, receiver = parameter in F
   ptr = new char[length(rhs.ptr)+1];
   scopy(ptr,rhs.ptr);
}
```

This code will work also while returning a `String` from a function.

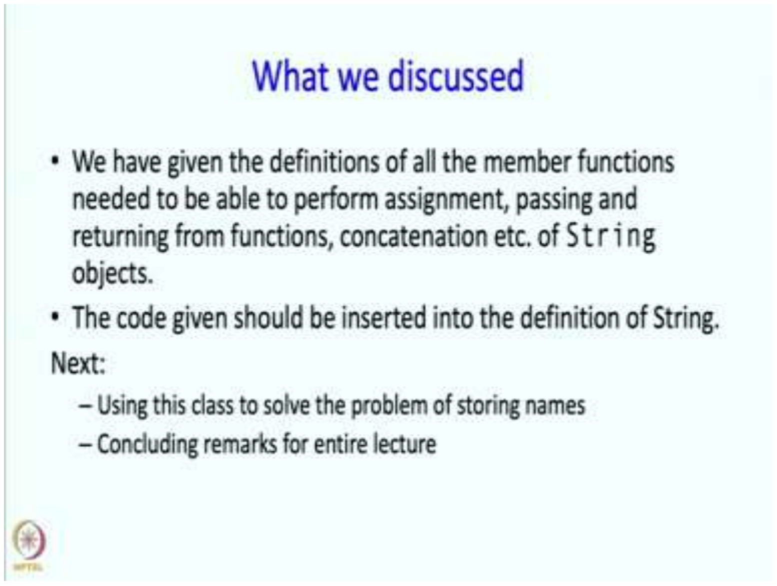
The copy constructor is called by C++ to copy a string object to the parameters. So, this string object is copied to the parameter in f and that copy is a is little bit of special copy, it is not just like an ordinary assignment it is a little bit more special than that and we will see in what way it is special and therefore C++ has a notion of a copy constructor. So, the copy constructor looks like this.

So, it is a constructor, and therefore there is no return type and it is constructing, so here we are going to use it to construct that parameter in f and so it is going to take one argument which is the object from which we are supposed to make the copy. So, in this case argument is going to be a and rhs is really going to be a when this code gets executed and the receiver will be the variable which is the corresponding parameter in f.

So, what happens? Well, ptr which is the receiver's ptr is going to be assigned, is going to be made to point to the string that rhs contains. So, first it must allocate some memory and that is what we are doing over here and its length should be adequate for getting a copy of this. So, rhs.ptr length plus 1 for the sentinel null. After that we are going to make a copy, the actual copy and that is about it.

Notice that we did not delete ptr as we deleted in the case of the other assignment operations because this ptr is the ptr of the parameter of f which is of type string , so that is why we are copying this argument to that parameter , but this parameter is just been constructed as we speak and therefore it is not pointing to anything at all, so we do not have to delete ptr. And likewise there is nothing to be returned because this is a constructor and so in the assignment we were returning something, so the code looked a little bit more complex but here there is no chaining that might be needed and therefore it is just going to be copying from copying from this a to the parameter inside f and just this is necessary. So, this is the core code of the assignment statement but before that we had to delete ptr in the assignment statement and then we have to return the object this object itself but both of these things are not necessary as far as this constructor is concerned. So, this is the code which is used to copy an argument to a parameter but C++ uses the same code to copy the result back from the called function to the calling function and that same copy constructor is used and this code will work for that purpose as well.

(Refer Slide Time: 12:22)



What we discussed

- We have given the definitions of all the member functions needed to be able to perform assignment, passing and returning from functions, concatenation etc. of `String` objects.
- The code given should be inserted into the definition of String.

Next:

- Using this class to solve the problem of storing names
- Concluding remarks for entire lecture

So, what have we discussed in this segment? so we have given the definitions of all member functions needed to perform assignment, passing and returning from functions, concatenation etc for our string objects. And these pieces of code should be inserted into the definition of the string. So, next we will use this class to solve the problem of storing names and we will have concluding remarks for our entire lecture. But before that we will take a short break.