

**Design and Pedagogy of the Introductory Programming Course**  
**Prof. Abhiram G. Ranade**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology – Bombay**

**Lecture – 18**

**Advanced Programming Topics.1: Advanced memory management, Standard Library**

Welcome back, we are talking about advanced programming topics and the topic for this sub lecture is advanced memory management and also the second topic; a different topic standard library.

**(Refer Slide Time: 00:34)**

**Advanced memory management**

Core curriculum: notion of a heap, operations new and delete.

- ▶ Preventing memory leaks/dangling pointers requires some thought. T2C

Pointer is being deleted/ going out of scope. Should we delete pointed object?

Should delete: If the object is on heap and has no other pointers.  
No other pointer, not deleted => memory leak.

Should not delete: If the object is not on heap or has other pointers.  
Has other pointers, deleted => Dangling pointers

Hard to know for the programmer. May also "forget"

Some automatic protocols for memory management T2C

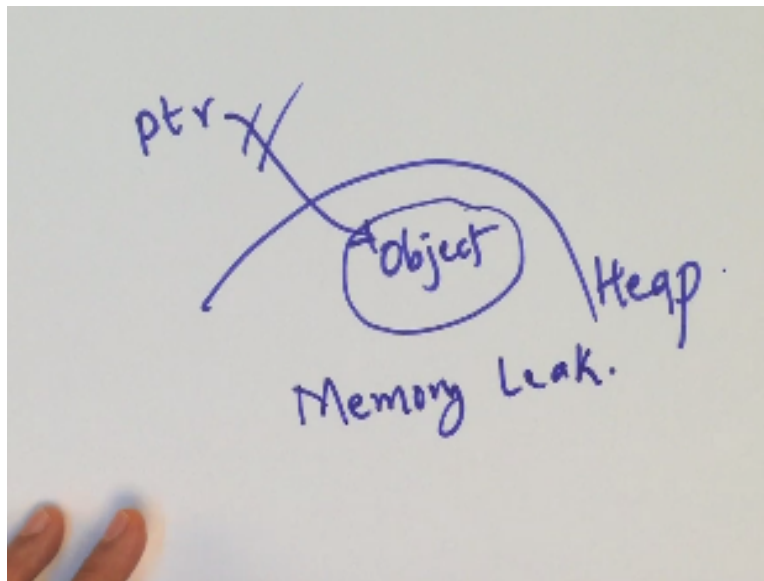
- ▶ Deep copying protocol Used in C++ standard library  
Detailed discussion Ch 21, [Ran14]
- ▶ Reference counting protocol Standard library class `shared_ptr`  
Appendix G, [Ran14]
- ▶ Garbage collection Not relevant for C++, but give brief idea.

So, on advanced memory management technique begin by saying that in the core curriculum, the notion of the heap and the operations new and delete should be discussed. Now, to use these operations is a little bit tricky because they can lead to memory leaks and dangling pointers and preventing these things require some thought, so how to do this is a part of the TL2 core not the basic core.

In the basic core, you can just say that there is a heap and you can allocate and deallocate using new and delete, okay but if you want to become good professional programmers, you need to know these things and even in the introductory programming course, you can give some introduction to it. Now, the typical problem is this; suppose, we have a pointer, so suppose that pointer is part of an object which is being deleted.

Or the pointer itself going out of scope, what should we do? What should we do if the pointer is pointing to some object, okay, so we have some pointer which is going to go away and it is pointing to some object, what should we do, Should we delete the pointed object or should we leave it alone, okay, so we should delete the pointed object if the object is on the heap and it has no other pointers, okay.

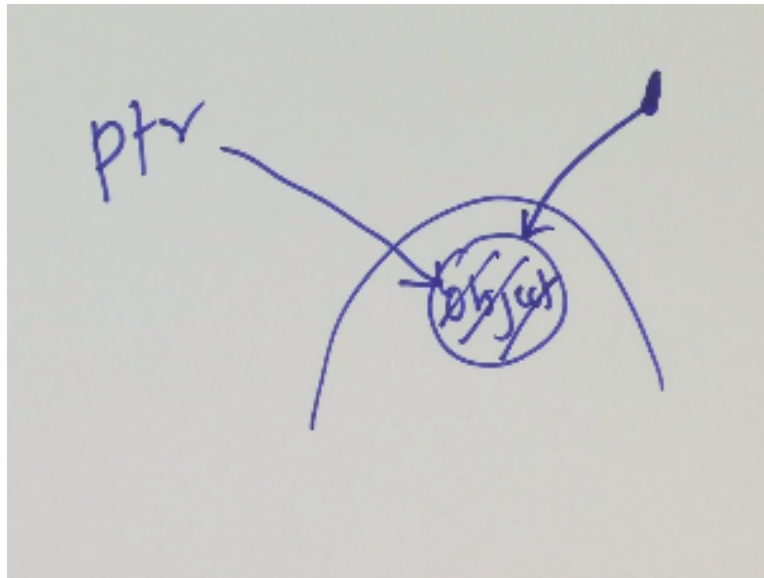
**(Refer Slide Time: 02:29)**



Why is that? Well, if it has no other pointers and if we do not delete, what will happen? So, let me take a picture, here is the pointer which is going out of scope say and it is pointing to some object and this is on the heap, if you delete this, then this object has been given to us and we do not have any other pointer to it, which means that we are not going to be able to use this memory and it is still marked as; used by the heap management functions.

So, this is what is typically, this is what is called a memory leak, so here allow memory leaks, your heap will start getting more and more such dead cell and you may eventually, loose the entire heap because everything in it is dead cells, okay, so this is not something that you want to do and therefore, if there is no other pointers, then we should delete this object because you know that that is not going to be used by anyone, you do not need it.

**(Refer Slide Time: 03:48)**



But there is also a reason to say we should not delete it, if the object is not on the heap, then we certainly should not delete it but even if the object is on the heap, okay, so here is the object on the heap, it is pointed to by a pointer which we are; which is about to go out of scope or which is about to be deleted itself, then if there is an additional pointer and if we delete this object, what will happen?

If we dereference this pointer, when we are going to get to; we are going to point to a region of the heap, which has not been given to us and that is called a dangling pointer. So, if there is another pointer and if we deleted, then we are going to get dangling pointers. Now, the programmer may not know whether there is exactly one pointer or whether there are many pointers.

And even if the programmer knows, the programmer may forget, I mean after all the programmer is human, we can forget to initialise, we can forget to delete but we should not but still we do, so what is needed is some automatic protocols for memory management. So, things which will take care of these things for you, okay, so course this is TL2 core, okay, now there are 3 things that happen, we are going to discuss 2 of those.

So, of the first is the deep copying protocol, okay and this is used in the C++ standard library which we are going to talk about next and there is a detailed discussion of this in the book, then

there is also a reference counting protocol, so this is implemented using the standard library class `shared_ptr` and then there is the garbage collection protocol, which is not really relevant for C++ but you should mention it and explain it in a little bit at least, okay.

**(Refer Slide Time: 05:44)**


### Deep copy protocol

- ▶ Every object allocated on the heap is pointed to by a single pointer.
- ▶ If a pointer to heap memory is copied, then the pointed object must also be copied.
- ▶ If a pointer is deleted or goes out of scope, the pointed object must also be deleted and memory reclaimed.

If copied object has pointer which must also satisfy uniqueness, then its pointed object must also be copied, hence "deep"

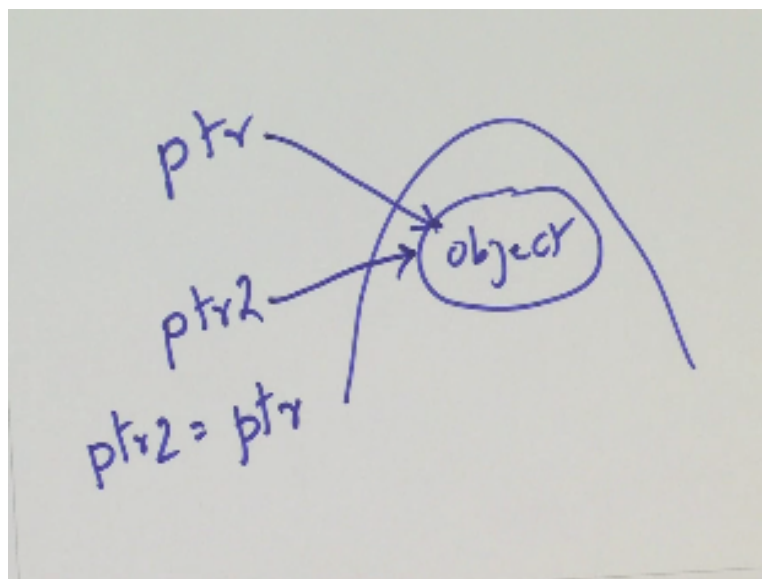
**Remark:** Ch 21, [Ran14] shows how to use protocol to implement a `String` class, which is a baby version of standard library `string` class.

- ▶ Deep copy used in copy constructor, assignment operator, destructor.
- ▶ Implementation consistent with deep copy discussed also for other operations such as `+`.



So, what is the deep copying protocol? So, basically the idea over here is that every object allocated on the heap is always pointed to by a single pointer, so this is our invariant so to say, okay. Now, what is this mean; if a pointer to a heap memory is copied then what should we do; so let us take a picture.

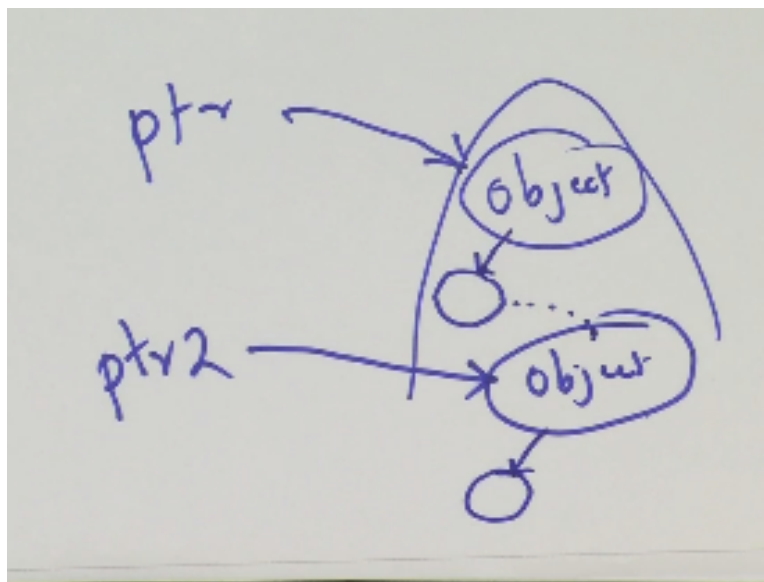
**(Refer Slide Time: 06:06)**



So, we have the heap memory, there is the object inside it which has been allocated and we have a pointer, well the pointer itself could be inside the heap but I am just showing it outside of that because I have drawn these heaps slightly small, so it is pointing to this object, now suppose we make a copy of this, so this is some ptr, then we make a copy let us call it ptr2 and we make an assignment  $\text{ptr2} = \text{ptr1}$ ; ptr, what does that mean?

Now, this will also start pointing to this object but now our invariant will be violated if we just do this copy and live with it, so what should we do, if you want to have the invariant and yet the program behaviour, the way we wanted, so what we should do is; that instead of just making this assignment in addition to the assignment, we should make a copy of this, so the new picture should not be what I have shown.

**(Refer Slide Time: 07:11)**



But instead it should be that we have the object over here, we have the ptr, then the moment we assign  $\text{ptr2} = \text{ptr1}$ , the assignment somehow should be changed or modified, so that we make a copy of object and ptr2 actually points to this, okay. So, why is this a good thing, well if we dereference ptr, we are going to go to object, if we dereference ptr2, we are going to go to object also, well a copy of it but the data unit is going to be the same.

So, our program is not going to know the difference, okay on the other hand our invariant has now been maintained, now this is a little tricky if this object itself contain some pointers, what

will happen? Well, if this object was just a copy of this, then we would get a pointer to this lower object but actually, the way this works is; we really should make a copy of this object as well and this pointer should really be pointing over here.

So that is why there is this name deep copy, okay, what if a pointer is deleted or goes out of scope, well if this pointer goes out of scope, we know that look nothing else can be pointing to this because our invariant guarantees that and therefore, we can delete this but of course, if we delete this and if our invariant is applicable to this pointer as well then this object will also have to be deleted, so we will need to set up that chain of deletions, okay.

Now, yeah, so if the copied object has a pointer which must also satisfy uniqueness, then its pointed object must also be copied and hence this is called deep, deep as opposed to superficial, superficial but say something like this, if I write `ptr2 = ptr`, then this is just this object being copied into this, okay but even here, the lower object is also copied and if there are further low down object that they will also be copied.

So that is why this deep opposed to superficial, so how is this protocol to be implemented, I means this seems like really complicated, okay but actually it is not, if you use the copy constructor and assignment operator and if you define them suitably and again the definitions are not too complicated and also the destructor, then all things happen behind a scenes for you, you do not have to worry about that, the user does not have to worry about that.

And again, the recursive nature of all of this will sort of make sure that exactly the right things happen, okay, so this is discussed in the book and as a T2C topic or TL2 core topic, it can be discussed, okay. Now, in the book the implementations of the concatenation operator, concatenation operator is also given and that will also require some amount of deep copy, okay.

**(Refer Slide Time: 10:35)**

## Reference counting protocol

Implemented by `shared_ptr` class

header file `<memory>`

- ▶ Objects allocated on heap can be pointed to by many pointers
- ▶ Keep track of number of pointers pointing. "Reference count"
- ▶ If `shared_ptr` is copied, reference count is incremented.
- ▶ If `shared_ptr` is destroyed, reference count is decremented
- ▶ If reference count becomes zero, heap object is deallocated.
- ▶ Works only if there are no circular pointer chains. Weak pointers

How to use: Instead of using ordinary pointers use `shared_ptr` throughout.

`shared_ptr` supports dereferencing etc. like normal pointers

Remark: Discussed in detail in Appendix G, [Ran14]

Also weak pointers

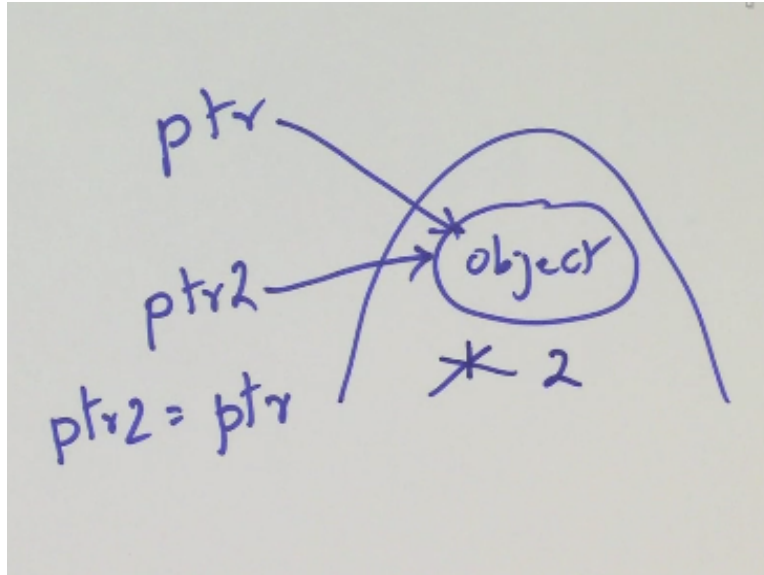


Navigation icons: back, forward, search, etc.

So, the second protocol that I am going to talk about is the reference counting protocol and this is also something that you should potentially consider and teaching as TL2 core, so this is implemented by the shared ptr class and for this, you need to use the header file memory, this is a standard library class, okay. Now, in this case instead of having the requirement that every object be pointed to by exactly one pointer, we are going to allow multiple pointers.

So, we will allow a situation like this and of course, this situation is better in some sense than allowing multiple copies, why it is better? Because well this is going to potentially use memory more efficiently, okay, all right, now here you still need to know when to delete the pointed object and so what we are going to do is; we are going to keep track of how many pointers are pointing to object.

**(Refer Slide Time: 11:49)**



So, after you write this assignment, somehow in our bookkeeping, the count for the number of pointers which might originally have been 1, should become 2 instead, okay, so the assignment operator for the pointers has been redefined by the header file; the standard library and that is available to you through the header file memory and that redefined operator will actually change the reference count to the count of how many references are there to that pointer, okay.

So, because there is this pointer, which is counting how many references are there, this protocol is called the reference counting protocol, okay, so if the shared ptr is copied, then the reference count is incremented, if shared pointer is destroyed, then the reference count is decremented, not only that if the reference count becomes 0, so now if I; if ptr2 goes out of scope as well as ptr goes out of scope, then this reference count will become 0.

And now, the code for destroying a pointer will itself called the destructor of this object as well and it will be deleted from the heap, so all this can be set up but in this case, you do not have to, it has already been done for you and it is available to you in the header file memory, okay. Now, there are some caveats here, mainly that if you have circular chain of pointers, then this idea will not work.

So, if you are sure that you are not going to have circular chains of pointers and most of the time we do not have circular chains, often we do not have circular chains of pointers, then the



reference counting protocol will work, if you do have circular chains of pointers, then there is something called weak pointer which can be used which also has been provided, okay. So, basically instead of using ordinary pointers, you essentially use shared pointer throughout and that is really the only change you want to make.


**(Refer Slide Time: 14:03)**

## Standard (Template) Library

The major advantage of C++ over C is that it provides a powerful library of algorithms and data structures.

- ▶ All basic data structures like dynamic arrays, lists, balanced search trees, and priority queue are elegantly supported.
- ▶ Because of template arguments, each data structure can be instantiated for any type. Impossible in C
- ▶ The notion of *iterators* enables data structures to be accessed without pointers.
- ▶ Dynamic memory allocation happens behind the scenes. Deep copy.
- ▶ Very safe.

It is often not necessary to build any data structures in your program, the library is often enough!

 Next: Some examples: vector, string, map

Also discussed in Ch 22, [Ran14]

And so, it is a really friendly library, okay, so you definitely should use it, okay and this has been discussed in an appendix of the book including a discussion of weak pointers. Our second topic in this lecture is the standard template library, okay. Now, the major advantage of C++ over C is that it provides a powerful library of algorithms and data structures, all the basic data structures like dynamic arrays, lists, balanced search trees and priority queue are elegantly supported.

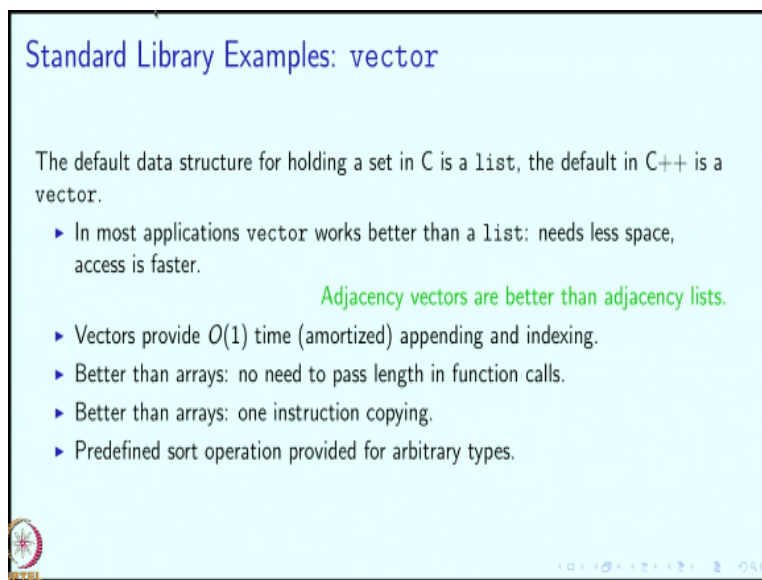
Because of template arguments, each data structures can be instantiated for any type and something like this is impossible in C, in C, you would have to write a separate code for each type, so if I have to want list of integers, I will need separate code from if I want to list of circles or some other structure, then there is a beautiful notion of iterators, which enables data structures to be accessed really without worrying about pointers, okay.

If the iterators are some kind of generalised pointers but they are given to you without you having to worry about memory allocation and things like that so, they allow you to step through the data structure in a very nice elegant manner and the dynamic memory allocation happens

behind the scenes, so things are really clean and safe, okay and the dynamic memory allocation in case of the standard library implements deep copy.

So, this really means that unless you really want a copy to happen, do not make copies instead use references to the same pointer, okay, same name, this is very safe that has been tested, okay and I would say that when you write C++ programs, the standard library is so good that you do not usually, you may not have to write any data structure code at all, the library is usually just adequate, okay.

**(Refer Slide Time: 16:14)**



**Standard Library Examples: vector**

The default data structure for holding a set in C is a list, the default in C++ is a vector.

- ▶ In most applications vector works better than a list: needs less space, access is faster.

Adjacency vectors are better than adjacency lists.

- ▶ Vectors provide  $O(1)$  time (amortized) appending and indexing.
- ▶ Better than arrays: no need to pass length in function calls.
- ▶ Better than arrays: one instruction copying.
- ▶ Predefined sort operation provided for arbitrary types.

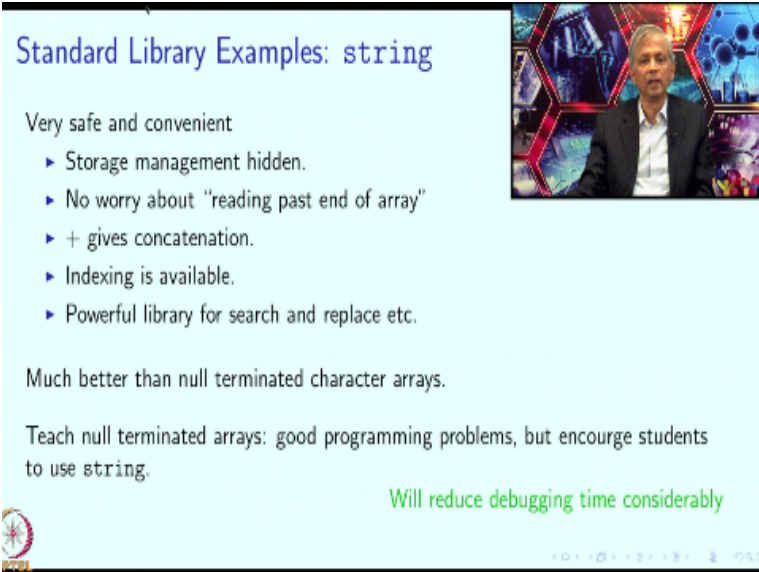
So, I am going to talk about 3 examples, okay but there are more, okay, so these are discussed in the book, as well. So, vector, we have talked about it a little bit, so I should say that the vector class is a better class than the list of C, okay, so but it serves sort of like what you would do with list. In C, if you want to implement a list, you would actually have to write code in C++, it is given to you.

And in most applications, it is better the list, it needs less space and it is much faster and for example, graphs are often represented by adjacency list that is what, in fact there is a name that graph has an adjacency list of presentation but in C++, you should use an adjacency vector representation because that will be much more compact and it will be; it will allow you to do things more efficiently.

The beautiful point about vectors, which are implemented by the data structure called dynamic array is that it provides  $O(1)$  time appending as well as indexing, okay and they are better than arrays because I can pass a vector and a vector has a member function called `size`, which will tell me what the length of the vector is or the size of the vector is and I do not have to pass the length in addition as I need to pass for an array.

So, it is actually better than array but it looks like it does everything that an array does and this does more, okay, so I can copy a vector, if I want and I can do this in a single instruction, in the single assignment statement without having to write a loop and then predefined sort operations are also available and this is also something which makes vectors very convenient.

**(Refer Slide Time: 18:00)**



**Standard Library Examples: string**

Very safe and convenient

- ▶ Storage management hidden.
- ▶ No worry about "reading past end of array"
- ▶ + gives concatenation.
- ▶ Indexing is available.
- ▶ Powerful library for search and replace etc.

Much better than null terminated character arrays.

Teach null terminated arrays: good programming problems, but encourage students to use `string`.

Will reduce debugging time considerably

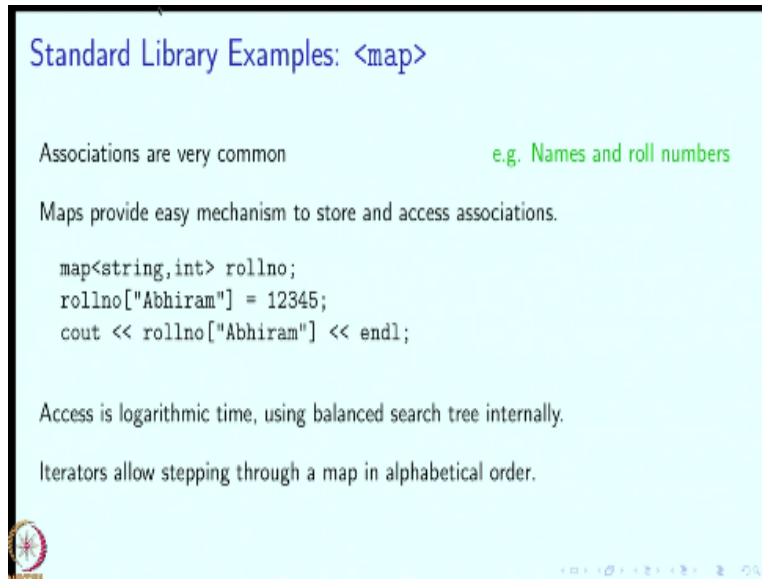
The slide includes a video inset of a man speaking and a small logo in the bottom left corner.

Then, there is the standard string library which we are talked about it but there are more member functions and again, this should be emphasised, so it is very safe and convenient, the storage management is hidden, no need to worry about reading past the end of the array plus gives concatenation, indexing is available, powerful library of search and replace are given to you and it is much better than null terminated arrays, safer and convenient, okay.

Now, you could teach null terminated array because say for example, the as a programming exercise, concatenating to strings given a null terminated array, it is a good exercise but if

somebody is writing code to do something other than just manipulate strings just for the purpose of a programming exercise, somebody is writing production code so to say that they must use the string class other.

**(Refer Slide Time: 19:10)**



Standard Library Examples: `<map>`

Associations are very common e.g. Names and roll numbers

Maps provide easy mechanism to store and access associations.

```
map<string,int> rollno;
rollno["Abhiram"] = 12345;
cout << rollno["Abhiram"] << endl;
```

Access is logarithmic time, using balanced search tree internally.

Iterators allow stepping through a map in alphabetical order.

So, they will spare themselves a lot of agony and debugging, okay, all right, the standard library also contain something called the map, so a map is used for implementing associations, okay. So, there are 2 maps actually, there is a map and there is also something called an unordered map, which is implemented by hash table, so a map implements associations using a balanced search tree, okay.

So, what is an association? So, for example I might have an association between the names and roll numbers, so a map provides an easy mechanism to store and access associations, so here is code which is doing that. So, for example, I might have an association called roll number, so what is roll number; a roll number takes as sort of a generalised index, a name or a string and returns the roll number associated with it.

So, I can write roll number of Abhiram to = 12345, to equal the number 12345, so this is going to store 12345 in the association roll number and associate the index Abhiram with 12345, now I can print roll number of Abhiram and it will pull out what was stored at this index so to say, so a

map really looks like an array but instead of restricting the indices to be integers, we can have arbitrary strings or even structures or we can have anything, pretty much anything as an index.

So, long as it satisfies some ordinary properties, so let me not go into detail of it but a map really is a very useful mechanism as you can see you would need you would often need to know the name, the roll number given the name and vice versa as well, so this is a very convenient mechanism and it is given to you and it has been debugged and tested and it uses the most efficient data structures.

So, it is really just a perfect arrangement, so the access is logarithmic time, using balanced search trees internally. Now, using iterators you can step through a map, so I can say that will give me the first element of this roll number, so it will give me that roll number of that person whose name is smallest or comes first in the alphabetical order, so I can say give me the first pair in your alphabetical order.

So, in this way I can print out roll numbers versus names in alphabetic order of the roll number, if you wanted in alphabetical, I am sorry, alphabetic order of the name, if you want to printed according to the roll numbers then of course, you can sort by the roll number and get it printed, get that printed as well. So, this concludes the topic of the standard template library and we will stop this sub lecture here, thank you.