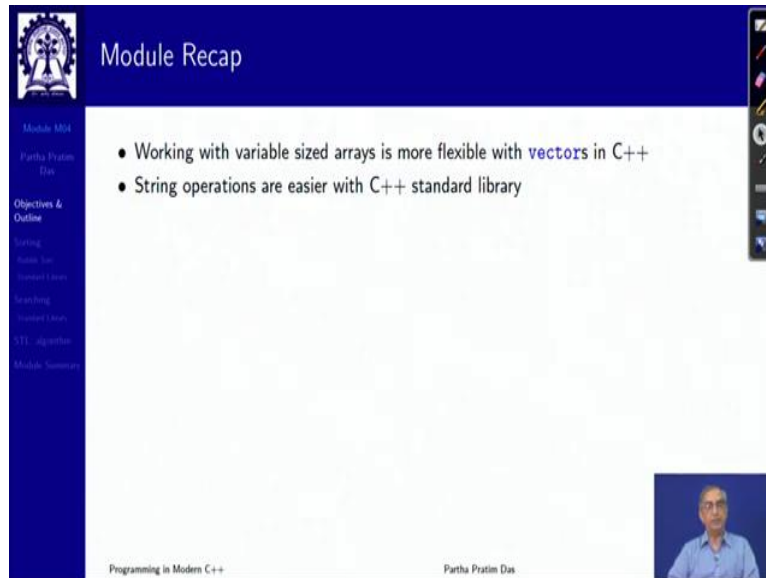


Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture No. 07
Sorting and Searching

Welcome to Programming in Modern C++. We are in week 1 and going to discuss module 4.
(Refer Slide Time: 00:36)



The screenshot shows a presentation slide titled "Module Recap" with a dark blue header. On the left, there is a sidebar with a logo at the top and a list of navigation items: "Module 004", "Partha Pratim Das", "Objectives & Outline", "Sorting", "Stack, List", "Containers Library", "Searching", "String Library", "STL algorithms", and "Module Summary". The main content area is white and contains two bullet points: "• Working with variable sized arrays is more flexible with `vectors` in C++" and "• String operations are easier with C++ standard library". A small video inset of the professor is visible in the bottom right corner. The footer of the slide includes "Programming in Modern C++" and "Partha Pratim Das".

In the previous module, we have talked about various types of arrays, fixed size, variable size, and how we can make the use of vectors in C++ standard library to make the arrays really efficient and easy to use, whether they are fixed size or they are of a dynamically known and dynamically created runtime size, they can actually be resized during the execution of the program as well.

And we have also taken a look in depth about the string operations which are easier with the string type given in the C++ standard library. And along with that, the C standard library of string functions also remain to be available.

(Refer Slide Time: 01:34)

The slide is titled "Module Objectives" and features a dark blue header with a logo on the left. A vertical sidebar on the left contains the text "Module M04", "Partha Pratim Das", "Objectives & Outline", "Sorting", "Bubble Sort", "Standard Library", "Searching", "Standard Library", "STL - algorithm", and "Module Summary". The main content area is white and contains a single bullet point: "• Implementation of Sorting and Searching in C and C++". A small video inset of the presenter is in the bottom right corner. The footer includes "Programming in Modern C++" and "Partha Pratim Das".

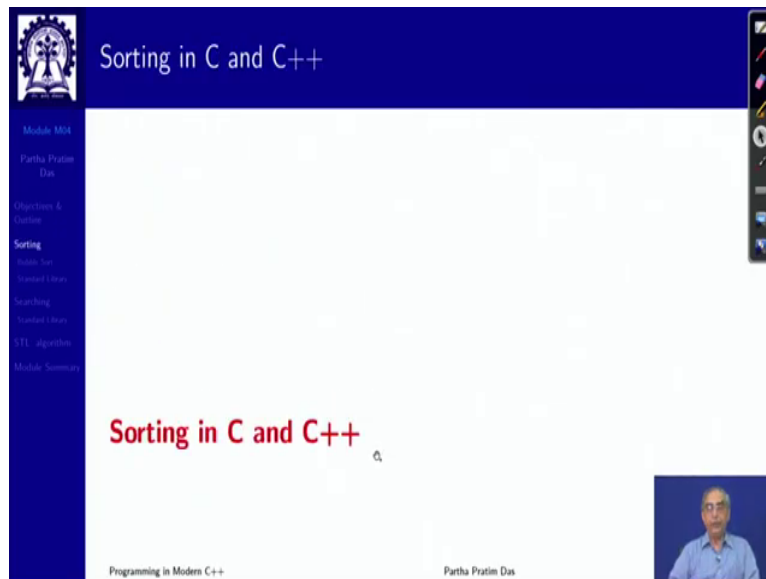
In the discussion in the current module, we will talk about Sorting and Searching in C and C++, which is the most common algorithms everybody learns when they start programming.

(Refer Slide Time: 01:51)

The slide is titled "Module Outline" and features a dark blue header with a logo on the left. A vertical sidebar on the left contains the text "Module M04", "Partha Pratim Das", "Objectives & Outline", "Sorting", "Bubble Sort", "Standard Library", "Searching", "Standard Library", "STL - algorithm", and "Module Summary". The main content area is white and contains a numbered list of four items: "1 Sorting in C and C++" (with sub-bullets "• Bubble Sort" and "• Using Standard Library"), "2 Searching in C and C++" (with sub-bullet "• Using Standard Library"), "3 STL: algorithm - The algorithm Library", and "4 Module Summary". A small video inset of the presenter is in the bottom right corner. The footer includes "Programming in Modern C++" and "Partha Pratim Das".

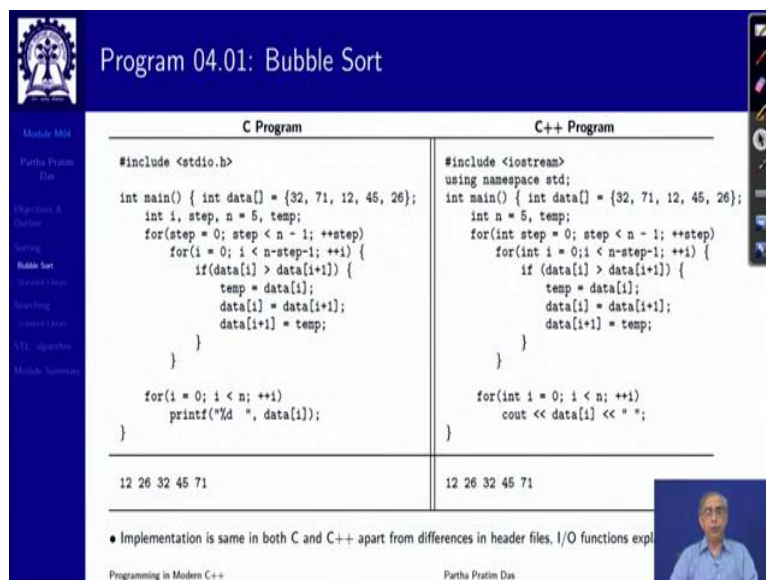
So, this is going to be the outline as will be available on the left-hand side.

(Refer Slide Time: 01:57)



So, first we talk about sorting.

(Refer Slide Time: 02:00)



Now, possibly the most widely known sorting algorithm, the oldest is the Bubble Sort, which I presume that all of you know how to do Bubble Sort, scan the array from one end to the other, look at conjugative elements if they are not in order swap them if they are in order, just move over to the next two elements.

And once one scan is over go over and scan it again keep on doing this till no change is done in an entire scan once that has been achieved, then the array has been sorted. Now, we have studied about the complexity of this efficiency in C or rather relative inefficiency of this sorting, we are not concerned with all of that, we are concerned with the how to really write the sorting.

So, here is a Bubble Sort code. And if you compare the codes between C and C++, you will find that they can be exactly identical except for the header for the io, which we discussed earlier in module two actually, as to what the purpose of these headers are. And accordingly, the print statements are also different.

(Refer Slide Time: 03:19)

Program 04.02: Using sort from standard library

C Program (Desc order)	C++ Program (Desc order)
<pre>#include <stdio.h> #include <stdlib.h> // qsort function // compare Function Pointer int compare(const void *a, const void *b) { // Type unsafe return (*(int*)a < *(int*)b); // Cast needed } int main () { int data[] = {32, 71, 12, 45, 26}; // Start ptr., # elements, size, func. ptr. qsort(data, 5, sizeof(int), compare); for(int i = 0; i < 5; i++) printf ("%d ", data[i]); }</pre>	<pre>#include <iostream> #include <algorithm> // sort function using namespace std; // compare Function Pointer bool compare(int i, int j) { // Type safe return (i > j); // No cast needed } int main() { int data[] = {32, 71, 12, 45, 26}; // Start ptr., end ptr., func. ptr. sort(data, data+5, compare); for (int i = 0; i < 5; i++) cout << data[i] << " "; }</pre>
71 45 32 26 12	71 45 32 26 12
<ul style="list-style-type: none">• sizeof(int) and compare function passed to qsort• compare function is type unsafe & needs complicated cast	<ul style="list-style-type: none">• Only compare passed to sort. No size is needed• Only Size is inferred from the type int of data• compare function is type safe & simple with no cast

Handwritten notes on the C code: "best: a, b" with arrows pointing to the compare function parameters, and "Type unsafe" and "Cast needed" with arrows pointing to the casted pointers in the compare function.

Now, let us see if we are doing sorting, can we make use of the standard library. In fact, I would presume that many of you who have been programming in C for a while may not be aware or may not have used the sorting function given in the C standard library and which is one of the best sorting algorithms known that is quicksort. The name of the function is qsort as you can see here. And it is available as a part of the standard library, stdlib. So, we include that header.

Now, this is how you invoke the quicksort function. So, what we are intending to do, we have a data array wherein we have initialized with 5 elements and we want to sort that array. So, what are the things that we need to know? Obviously, the quicksort needs to be told as to what is the array to sort. So, we need to tell what is array to sort and the way we tell that is we actually give the address of the zeroth location which is data, the base pointer, which the quicksort would be able to traverse an array starting from that.

Next thing it needs to know naturally is the size. How many elements are there? So, we say there are 5 elements. Now, how do you, from the initialization how do you find out there are 5 elements is a different question, which is a very standard code if you do not know we will discuss at some point of time, but here you can clearly see that there are 5 elements.

So, quicksort now knows the array and the number of elements, but just think of it that when someone wrote this quicksort code that programmer did not know whether you are sorting an array of integers int or an array of double or an array of float or an array of characters or for that matter an array of some structures, the person had no idea and still the person had to write the code for quicksort.

So, how could he have written a code which is general enough, so that it can solve the purpose. What he does is something very simple, assumes that the array is of bytes and it can traverse the array in certain stride of bytes either as a single byte each or two bytes each or four bytes each. So, that if the array is of type of array element type int and if int has a size 4, then what it does it takes 4 consecutive bytes together and takes that as a data value, then it takes the next 4 bytes and takes that as a data value which is this next integer and so on.

So, in brief what it needs to know is what is the size of every element that this array represents. If it knows that, then from the byte array it can easily interpret it as an array of the

specific type of element that you have provided. So, the third thing that quicksort needs to know is what is the size of every element. So, here I have five elements if size of int is 4, then actually quicksort got an array of 20 bytes, 4 times 5.

And now, it knows that there are 5 elements each element is of size 4. So, it will keep on doing a stride of 4 elements, 4 bytes consecutively and interpret them as the every individual element data, 5 different element data in the array, that is the whole idea. Having said that, the final question is to be able to sort you should be able to compare two elements. Now, when it is integer, you know how to compare there is comparison operator, when it is int, you know, when it is say double, you know what is the comparison operator.

But in general, you may not know how to compare two values of certain data type. For example, the elements could be strings themselves, how do you compare two strings you cannot do it by the comparison operator, you have to use strcmp or something, if the elements are user defined structure type, say complex number having two components of the structure, how do you compare them, you would not know how to compare, because quicksort was written well before its use when all these were not known to it.

And it is one code which must work for int, for double, for string, for character, for bool, for user defined structures, and so on so forth. So, the trick that is done is you have to tell quicksort how to compare. So, this is what is done here. You would recall that C has a mechanism called function pointers. So, a function pointer is a function whose address is being passed and the actual function will be given separately.

So, besides, while calling quicksort besides the actual call, it is the responsibility of the user programmer to also define this compare function. So, what is compare function let us consider, a compare function will take two elements, compare them and say if it is less or it is more and so on. So, I can think of compare function taking two elements say a and b and giving me a bool, Boolean result, saying whether it is a is less than b, then say it is true otherwise, it is false.

If I have such a function then qsort can use that function as a function pointer without actually knowing what the function is. So, the user will write that function. Now, there is a big question again, another stumbling block, what is that? If I want to say I am comparing a with b, I should be able to say what is the type of a and what is the type of b, are they ints, are the doubles, are the strings, are they user defined structure?

The comparison logic could be different and the comparison that you need to do that data type would be different for this compare function and you do not know that. So, what do you do you do another trick, I would, more than a trick I should call it a hack, which is very common hack in C, which makes it very risky actually, is that you, let me erase whatever is.

(Refer Slide Time: 11:37)

Program 04.02: Using sort from standard library

C Program (Desc order)	C++ Program (Desc order)
<pre>#include <stdio.h> #include <stdlib.h> // qsort function // compare Function Pointer int compare(const void *a, const void *b) { // Type unsafe return *(int*)a < *(int*)b; // Cast needed } int main () { int data[] = {32, 71, 12, 45, 26}; // Start ptr., # elements, size, func. ptr. qsort(data, 5, sizeof(int), compare); for(int i = 0; i < 5; i++) printf ("%d ", data[i]); }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • sizeof(int) and compare function passed to qsort • compare function is type unsafe & needs complicated cast 	<pre>#include <iostream> #include <algorithm> // sort function using namespace std; // compare Function Pointer bool compare(int i, int j) { // Type safe return (i > j); // No cast needed } int main() { int data[] = {32, 71, 12, 45, 26}; // Start ptr., end ptr., func. ptr. sort(data, data+5, compare); for (int i = 0; i < 5; i++) cout << data[i] << " "; }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • Only compare passed to sort. No size is needed • Only Size is inferred from the type int of data • compare function is type safe & simple with no cast

Programming in Modern C++ Partha Pratim Das M04.7

Program 04.02: Using sort from standard library

C Program (Desc order)	C++ Program (Desc order)
<pre>#include <stdio.h> #include <stdlib.h> // qsort function // compare Function Pointer int compare(const void *a, const void *b) { // Type unsafe return *(int*)a < *(int*)b; // Cast needed } int main () { int data[] = {32, 71, 12, 45, 26}; // Start ptr., # elements, size, func. ptr. qsort(data, 5, sizeof(int), compare); for(int i = 0; i < 5; i++) printf ("%d ", data[i]); }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • sizeof(int) and compare function passed to qsort • compare function is type unsafe & needs complicated cast 	<pre>#include <iostream> #include <algorithm> // sort function using namespace std; // compare Function Pointer bool compare(int i, int j) { // Type safe return (i > j); // No cast needed } int main() { int data[] = {32, 71, 12, 45, 26}; // Start ptr., end ptr., func. ptr. sort(data, data+5, compare); for (int i = 0; i < 5; i++) cout << data[i] << " "; }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • Only compare passed to sort. No size is needed • Only Size is inferred from the type int of data • compare function is type safe & simple with no cast

Programming in Modern C++ Partha Pratim Das M04.7

Program 04.02: Using sort from standard library

C Program (Desc order)	C++ Program (Desc order)
<pre>#include <stdio.h> #include <stdlib.h> // qsort function // compare Function Pointer int compare(const void *a, const void *b) { // Type unsafe return *(int*)a < *(int*)b; // Cast needed } int main () { int data[] = {32, 71, 12, 45, 26}; // Start ptr., # elements, size, func. ptr. qsort(data, 5, sizeof(int), compare); for(int i = 0; i < 5; i++) printf ("%d ", data[i]); }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • sizeof(int) and compare function passed to qsort • compare function is type unsafe & needs complicated cast 	<pre>#include <iostream> #include <algorithm> // sort function using namespace std; // compare Function Pointer bool compare(int i, int j) { // Type safe return (i > j); // No cast needed } int main() { int data[] = {32, 71, 12, 45, 26}; // Start ptr., end ptr., func. ptr. sort(data, data+5, compare); for (int i = 0; i < 5; i++) cout << data[i] << " "; }</pre> <p style="text-align: center;">71 45 32 26 12</p> <ul style="list-style-type: none"> • Only compare passed to sort. No size is needed • Only Size is inferred from the type int of data • compare function is type safe & simple with no cast

Programming in Modern C++ Partha Pratim Das M04.7

Now, look at the compare function, the compare function and see what is the first argument and what is the second argument, the numbers to compare. What we say is the first argument is `const void*`, forget about the `const` part, we have not done `const`, all that it says is you cannot change this formal parameter from within this function, forget about that we will study that later, but the basic point is the value is passed not as a value but as a pointer to it.

But if the value were, if `a` is `int`, if `a` were `int`, then the value that the pointer that we will need to pass `int*`, if it is `double`, you need to pass a `double*` you do not know that. So, what do you do? You pass `void*` which says that this is a pointer to a type that I do not know. So, the quicksort actually is calling a function to compare which takes two `void*` pointers.

The two elements it wants to compare, so that whatever is the element type it can pass the addresses of the corresponding locations and then the user function will know what is the type, so the user function has to cast the pointer to the appropriate type compare the values and return the Boolean. So, this is the basic idea.

So, if you now see these are the two parameters with `void*` pointer, `const void*` pointer, you cast first you cast that as `int*` because you know that you are doing this to sort an array having elements of the `int` type. So, you cast it to `int*` it becomes `int*` pointer, the `a`'s address is now `int*` pointer and then you dereference `a*`. So, it now gives you the value of `a`.

Similarly, here you get the value of `b`. So, this is equivalent to `a < b`. But if you had done an array of `double`, you would have done this casting not to `int*` you would have done it to `doubl*`, whatever is your element type you will cast to that. You do the comparison and return the Boolean result and as you know in C in general you may not have Boolean, particularly we have seen how to do `stdbool` and all that.

So, you quicksort takes that what you return, actually quicksort was given in the library in the K and RC time. So, when the `bool` was not there, so the only way to realize Boolean was through `int`. So, the prototype of compare that function pointer for compare is `int`, return type of that is `int`, so it returns an integer value for example, if `a < b`, it will return 1, if `a > b` it will return 0.

So, using that, now we pass that function pointer. And as you know, for function pointer, it is not necessary to take the address so I can just write the function name. Alternately, I could have written, `&compare` also, both of them would be correct. So now, quicksort knows the array, the number of elements, the size of every element to stride, and the comparison function and with that quicksort will be able to sort this.

And as a part of C programming, I do not know, if you have used this, if you have not, you can use it, it is a very efficient and safe way of using. Though, there is a lot of questions regarding this compare function, you have to write it very, very carefully. Let us go over to C++. C++ standard library also provides a sorting function, it is called `sort` and it is provided in the standard library component called `algorithm`. `Algorithm` is a component of C++ standard library, which gives you a number of algorithms.

So, when you call `sort` again, we have the same data array for the elements which we want to sort so you call `sort`, first you say what is the starting address, starting pointer then what you say you do not really say the number of elements, but you give a pointer to the last element. `data + 5`, what will be `data + 5`? This is `data`, this is `data + 1`, `data + 2`, `data + 3`, `data + 4`. So, `data + 5` is here. So, earlier I discussed about defining a range starting from an element and ending but not including the last element.

So, the starting pointer is data, the ending which is not included is data + 5 which is data[5] in the in the array notation. So, it will not get included, so that is of 5 elements. It also has to give a compare function to tell how to compare. Now, the point to note here is it does not need to provide the number of elements, I am sorry, the size of the element. Why? Because in C++, what it does, it knows the type of the array data.

And it uses some trick in writing the sort function, a trick of templates as we will see later, so that the compiler can generate an appropriate sort function for you from the templated code given in the algorithm, when you use it for int type, which is known from the type of the data array that you have declared.

Interestingly, if this were in array of double the compiler will generate a code for double, if it were for a user defined structure, the compiler will generate a code for that user defined structure wherein, we will buy the time we will know what is the size, so this no not necessary to pass this third parameter which you require for the qsort. But you still need to tell the compare function because of two reasons, one is what do you assume to be comparing is it less or it is more, there are two choices.

So, you cannot make both the choices in the library, so you have to decide on which choice you are making and accordingly you will have to put that function. So, you can do that if you pass it as a function pointer. Second, in general, what you want to do for comparison is your choice, you may want to do a lexicographic sorting, you may want to do a topological sorting, it depends on what kind of sort you want to do.

So, the library allows you to give pass a function pointer here. And interestingly, since the type is now known to be int, you do not need to do all this roundabout of const void* and all that, you can directly have the element type in the parameters of the compare function. So, a different compare function for double when you are sorting double will have double as the two parameters of the compare function.

And certainly, in C you have bool so you will use bool directly as a return type. And with this, like in the case of C you will be able to sort that, very simple. So, one major problem that we always have is to have a sorting is directly given, just it is just a one line of code and a compare function that you provide.

(Refer Slide Time: 20:11)

Program 04.03: Using default sort of algorithm

```
C++ Program (Asc Order)
// sort.cpp
#include <iostream>
#include <algorithm> // sort function
using namespace std;

int main () {
    int data[] = {32, 71, 12, 45, 26}

    sort(data, data+5);

    for (int i = 0; i < 5; i++)
        cout << data[i] << " ";

    return 0;
}
```

12 26 32 45 71

- Sort using the default sort function of algorithm library which does the sorting in ascending order only
- No compare function is needed

Programming in Modern C++ Partha Pratim Das M04.8

So, let us move on. Now, this, the example that I showed is sorts in descending order, if you want to sort in ascending order, it becomes even simpler, because that is considered to be the default for the algorithm sort given. So, you do not even need the third parameter, you just say what is the range, data, data + 5 and asked to sort.

So, for predefined type, built-in types, and for ascending order, you do not even need to specify the compare function to be passed as the function pointer, if you want to use it for user defined types and or for ascending order or for some other order, you will have to write the compare function for yourself.

So, this is a story of sorting between C and C++ and you must have noticed that how easy it is to write this sorting function from the, use the sorting function from the standard library in C++ compared to the cumbersome your different casting that you need to do wherein you can always make mistakes and so on. And it is just straightforward compare function and sort call that will solve the problem in C, C++.

(Refer Slide Time: 21:39)

Searching in C and C++

Module 04
Partha Pratim Das
Objectives & Outline
Sorting
Built-in
Standard Library
Searching
Searching
STL algorithms
Module Summary

Searching in C and C++

Programming in Modern C++ Partha Pratim Das M04.9

Program 04.04: Binary Search

C Program	C++ Program
<pre>#include <stdio.h> #include <stdlib.h> // bsearch function // compare Function Pointer int compare(const void * a, const void * b) { // Type unsafe if (*(int*)a < *(int*)b) return -1; // Cast needed if (*(int*)a == *(int*)b) return 0; // Cast needed if (*(int*)a > *(int*)b) return 1; // Cast needed } int main () { int data[] = {1,2,3,4,5}, k = 3; if (bsearch(&k, data, 5, sizeof(int), compare)) printf("found!\n"); else printf("not found\n"); }</pre>	<pre>#include <iostream> #include <algorithm> // binary_search function using namespace std; int main() { int data[] = {1,2,3,4,5}, k = 3; if (binary_search(data, data+5, k)) cout << "found!\n"; else cout << "not found\n"; }</pre>
found!	found!
<ul style="list-style-type: none"> compare function is type unsafe & needs complicated cast 	<ul style="list-style-type: none"> No compare function needed

Programming in Modern C++ Partha Pratim Das M04.10

Now, let us take a look at searching wherein we have a very similar solution. Again, I am not sure if you have used it in your programming, but C standard library provides a binary search function as we all know, for searching binary search is a good way to search in an array, just keep that sorted and then we are binary search. So, for how do you invoke binary search? What you will have to provide?

You will have to provide the array naturally, you have to provide the array, so the starting pointer. You have to tell the number of elements in the same way. You have to tell the size of every element because `bsearch` was again wrote long before your code was wrote, so it does not know what is the strike, what is the unit of the elements in terms of the number of bytes that it knows.

You have to give the function to compare so that you can do the comparison. And you have to also tell the element the value that you want to search. So, that is the first parameter. Note that this is passed as an address, because it becomes easier for the `bsearch` to manipulate that because internally it does not again know what is the type of value that is involved. So internally, it cannot have a value `k` because it does not know the type.

So internally what it does, it assumes that it is getting an address, which is again of the type `void*` and makes use of that, using the size, it make figured out as to how much of it is to be read. Now, coming to the compare function. In binary search, in while you are sorting, it is just okay to know if it is less or if it is more, so less and greater than or equal to or greater and less than equal to solves a sorting problem, because you really are not concerned about the ordering of the equal elements, they are obviously they will become conservative in this and they are indistinguishable.

Whereas, in binary search, you have a three-way comparison to be done, you want to hit the middle. See if that value is matching, if it is matching you have already found it, if it is not you will have to go to left assuming that the array is sorted in increasing order as is given in this example. So, you have to go to the left half of the array, if it is more you have to go to the right half of the array. So, you need a three way, you need a three-way comparison result less, equal to, more.

So, you have to have a different kind of comparison function, so that comparison function is what is written here. So again, you take the parameters as `const void*` because you do not know that type, because `bsearch` does not know that type. So, the function signature, function pointer signature does cannot have any other type, then you have to cast them in the same way, first cast it to integer pointer, then dereference to get the value, if it is less, you have to return `-1`.

If it is equal, you have to return `0`. If it is greater, you have to return `+1` that is a protocol, it is pretty much like what the `strcmp` does, `-1` for less, `0` for equal, and `+1` for greater. So, naturally this is not `int` interpreted as `bool` this necessarily is a `bool` because you have three possible outcomes and that is the assumption that `bsearch` has made in taking this function pointers.

So, now, you have the compare function pointer given here, whatever is your type accordingly, he will write this part of the compare code to tell binary search how to compare two elements, how to find if they are equal or if 1 is less than the other greater than the other and so on. And pass that along with the array base address, number of elements, size of every element, and pointed to the particular value that you want to search. So, if you do that, you will find. Naturally we took `3`, so it is found to be available.

So, if you do the similar thing, if you want to do that similar thing in in C++, again, the algorithm component has a `binary_search` function given, algorithm given. And all that you need to give you can understand that you do not need to provide any size parameter, you need to give the range data, `data + 5` like before, you need to provide the value that you want to search for, which is `k`.

Again, since it knows the type, you do not need to pass a pointer because again, it is internally during that template magic so that given `int` it will create a `binary_search` for you for integer values, so it knows the `int`, if it is of your user defined type, it will do that accordingly. And in this case, since you are doing a `binary_search` on a predefined or built-in type, you do not need to provide the compare function. If you want to do it for your own data type, or your defined structure, you will have to provide the compare function as well. So, it is as simple as that.

(Refer Slide Time: 27:39)

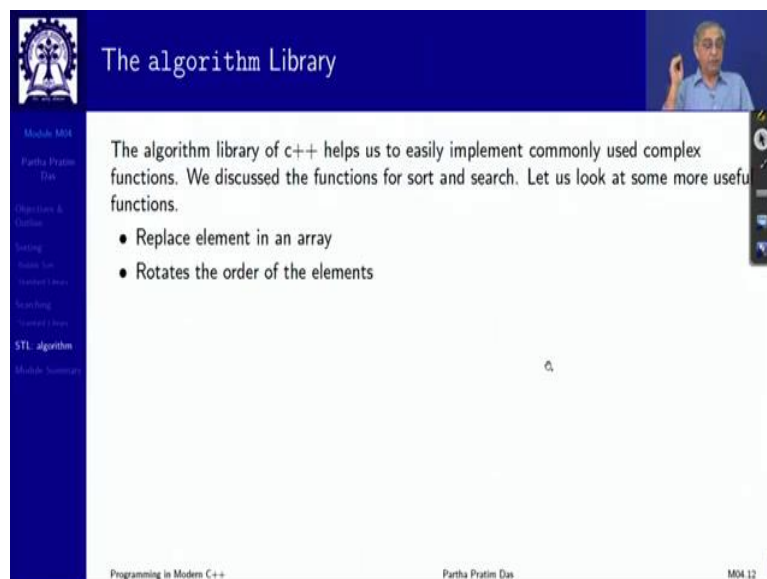


Slide 11: STL: algorithm - The algorithm Library

Module M04
Partha Pratim Das
Objectives & Outline
Sorting
Stack Set
Standard Library
Searching
Standard Library
STL: algorithm
Module Summary

STL: algorithm - The algorithm Library

Programming in Modern C++ Partha Pratim Das M04.11



Slide 12: The algorithm Library

Module M04
Partha Pratim Das
Objectives & Outline
Sorting
Stack Set
Standard Library
Searching
Standard Library
STL: algorithm
Module Summary

The algorithm library of c++ helps us to easily implement commonly used complex functions. We discussed the functions for sort and search. Let us look at some more useful functions.

- Replace element in an array
- Rotates the order of the elements

Programming in Modern C++ Partha Pratim Das M04.12

So, as we can see that sorting and searching are really, really easy using the C++ standard library. Now, this algorithm library has several other functions as well, which are of great

convenience, for example, you can replace an element in an array, you can rotate the elements in the array, which are things you require in various algorithms.

(Refer Slide Time: 28:05)

Program 04.05: replace and rotate functions

Replace	Rotate
<pre>// Replace.cpp #include <iostream> #include <algorithm> // replace function using namespace std; int main() { int data[] = {1, 2, 3, 4, 5}; replace(data, data+5, 3, 2); for(int i = 0; i < 5; ++i) cout << data[i] << " "; return 0; }</pre>	<pre>// Rotate.cpp #include <iostream> #include <algorithm> // rotate function using namespace std; int main() { int data[] = {1, 2, 3, 4, 5}; rotate(data, data+2, data+5); for(int i = 0; i < 5; ++i) cout << data[i] << " "; return 0; }</pre>
1 2 2 4 5	3 4 5 1 2
• 3 rd element replaced with 2	• Array circular shifted around 3 rd element

Programming in Modern C++ Partha Pratim Das MOA.13

So here I have given code snippets, I mean codes to show the use of some of these functions like replace, here will take the range, which now you have understood is the style of doing it, then it takes in the range, the positional value point where you have to want to make the change. So, this is 1, 2, 3 I want to make changes here. So, I write 3 and the value that I change it to, so I give 2 you can see, this third value or data 2 has got changed to the value 2. Simple one call and I can need to do that in generality.

You can also rotate by giving the range the basic convention is little different, this is where you give the range from the first and the third and in the middle, you see the point up to which you rotate. So, this says that you rotate up to 2. So, if you rotate once, then 1 comes here, if you rotate twice, then 2 comes here and therefore it becomes 3 4 5 1 2, 3 4 5 1 2. And data 2 is what becomes the beginning after the rotation. So, these are some of those, some of the very common algorithms which are all available, you can just need to include the algorithm component and keep using it.

(Refer Slide Time: 29:40)

The image shows a presentation slide with a dark blue header and footer. The header contains the text 'Module Summary' and a small logo on the left. The main content area is white and contains three bullet points. The footer contains the text 'Programming in Modern C++', 'Partha Pratim Das', and 'M04.14'.

Module Summary

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the `algorithm` library
- Predefined optimised versions of these sort and search functions can also be used
- There are a number of useful functions like `rotate`, `replace`, `merge`, `swap`, `remove` etc. in `algorithm` library

Programming in Modern C++ Partha Pratim Das M04.14

So, besides rotate, replace, you also have merge, you have swap already given, you have remove an element, and so on so forth. So, it has a lot of good value. So, what you have seen in this module is that compared to C standard library, it has C++ standard library has all the same functionality, but in a more convenient way in the algorithm component.

And in addition, it provides a lot of additional functions which are commonly used in writing different problems. So, that brings us to the end of this module. Thank you very much for your attention, and we will meet in the next module.