

Programming in Modern C++
Professor Pratha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture: 43
C++ Standard Library: Part 1 (Generic Programming)

(Refer Slide Time: 00:32)

Programming in Modern C++
Module M43: C++ Standard Library: Part 1 (Generic Programming)

Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Programming in Modern C++ Partha Pratim Das M43.1

Welcome to programming in modern C++. We are in week 9 and we are going to discuss module 43.

(Refer Slide Time: 00:37)

Module Recap

- Understood object-oriented I/O of C++
- Learnt the major standard library components

Programming in Modern C++ Partha Pratim Das M43.2

In the last module, we understood the object-oriented I/O of C++, the `iostream`, `fstream` and other headers and learned some of the major standard library components. We will actually

go further in that to get an overview of the overall standard library components of C++ and understand the generic programming what is generic programming for STL.

(Refer Slide Time: 01:13)

The slide is titled "Module Objectives" and features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white and contains two bullet points. A vertical navigation menu is on the left side of the slide.

- To get an overview of Standard Library components of C++
- To understand generic programming for STL

Navigation menu items: Module M43, Partha Pratim Das, Objectives & Outlines, Standard Library, C++ Lib, C++ STL Lib, STL, Header Conventions, Generic Programming, Common Tasks, Lifting Example, Model, Examples, Module Summary.

Page footer: Programming in Modern C++, Partha Pratim Das, M43.3

The slide is titled "Module Outline" and features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white and contains a numbered list of three items. A vertical navigation menu is on the left side of the slide.

- 1 Standard Library
 - C Standard Library
 - C++ Standard Library
 - std
 - Header Conventions
- 2 Generic Programming
 - Common Tasks
 - Lifting Example
 - Algorithms-Iterators-Containers Model
 - Examples
- 3 Module Summary

Navigation menu items: Module M43, Partha Pratim Das, Objectives & Outlines, Standard Library, C++ Lib, C++ STL Lib, STL, Header Conventions, Generic Programming, Common Tasks, Lifting Example, Model, Examples, Module Summary.

Page footer: Programming in Modern C++, Partha Pratim Das, M43.4

This is the outline of points on standard library and generic programming which will be available on the left panel.

(Refer Slide Time: 01:26)

Standard Library

Module: M43
Partha Pratim Das

Objectives & Outlines
Standard Library
C 98 C++
C++98 C++11
STL
Header Conventions
Generics
Programming
Common Traits
Library Example
Model
Example
Module Summary

Standard Library

Programming in Modern C++ Partha Pratim Das M43.5

What is Standard Library?

Module: M43
Partha Pratim Das

Objectives & Outlines
Standard Library
C 98 C++
C++98 C++11
STL
Header Conventions
Generics
Programming
Common Traits
Library Example
Model
Example
Module Summary

- A *standard library in programming* is the library made available across implementations of a language
- These libraries are usually described in *language specifications (C/C++)*; however, they may also be determined (in part or whole) by *informal practices of a language's community (Python)*
- A language's standard library is often treated as part of the language by its users, although the designers may have treated it as a separate entity
- Many language specifications define a *core set that must be made available in all implementations*, in addition to other portions which may be optionally implemented
- The line between a *language and its libraries* therefore differs from language to language
- Bjarne Stroustrup, designer of C++, writes:
What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, "What ought to be in some library?" but "What ought to be in the standard library?" The answer "Everything!" is a reasonable first approximation to an answer to the former question but not the latter. A standard library is something every implementer must supply so that every programmer can rely on it.
- This suggests a *relatively small standard library*, containing only the constructs that "every programmer" might reasonably require when building a large collection of software
- **This is the philosophy that is used in the C and C++ standard libraries**

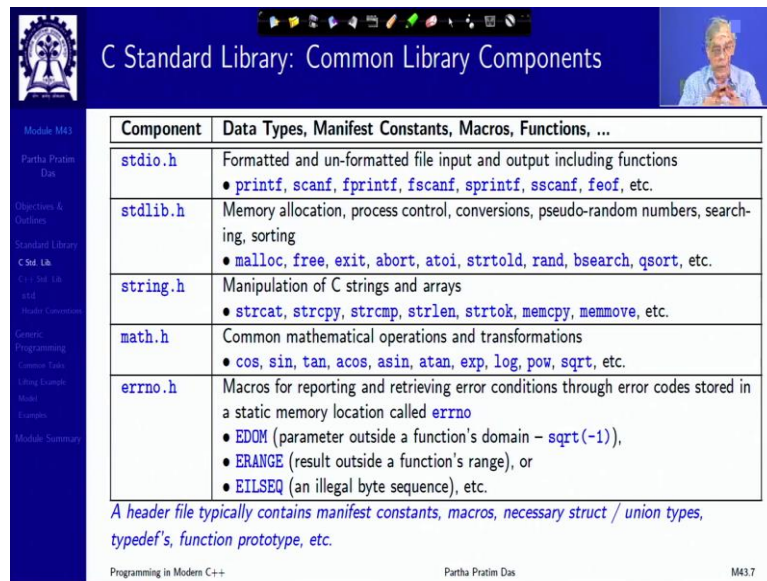
Source: Standard library, Wiki Accessed 13-Sep-21

Programming in Modern C++ Partha Pratim Das M43.6

So, just to quickly recap, what is the standard library? A standard library is a collection of functions or classes typically templated which are made available in addition to the core language. So, the core language has a lot of features as we have already studied at least, all major features of C++98, C++03 we have already studied. Will do some more for C++11.

But in order to facilitate the programming the development work for the software developers in an easy manner languages do provide a standard library which is also, specified in the language standard. So, C had a standard library much of which we have seen through practice and C++ has a significant standard library which is designed keeping, relatively small size in mind though that small size itself is not small enough. And it is useful for every programmer who needs to do programming in C++.

(Refer Slide Time: 02:37)



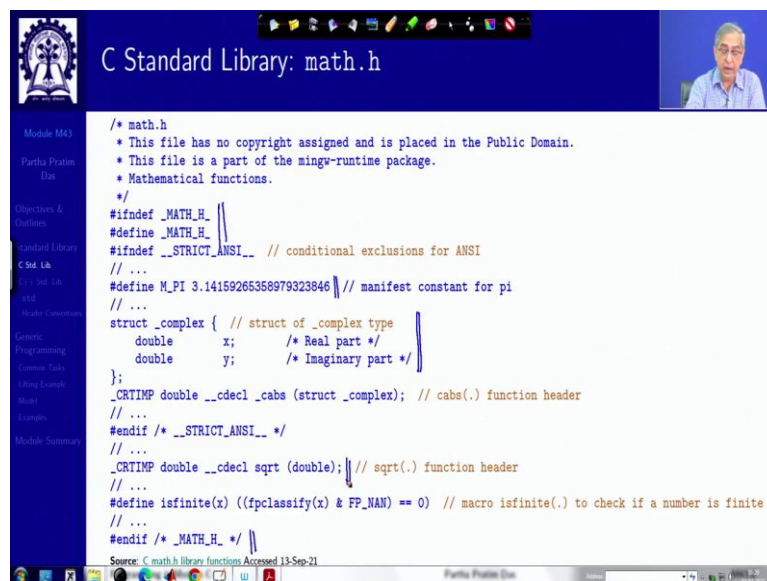
Component	Data Types, Manifest Constants, Macros, Functions, ...
<code>stdio.h</code>	Formatted and un-formatted file input and output including functions • <code>printf</code> , <code>scanf</code> , <code>fprintf</code> , <code>fscanf</code> , <code>sprintf</code> , <code>sscanf</code> , <code>feof</code> , etc.
<code>stdlib.h</code>	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting • <code>malloc</code> , <code>free</code> , <code>exit</code> , <code>abort</code> , <code>atoi</code> , <code>strtold</code> , <code>rand</code> , <code>bsearch</code> , <code>qsort</code> , etc.
<code>string.h</code>	Manipulation of C strings and arrays • <code>strcat</code> , <code>strcpy</code> , <code>strcmp</code> , <code>strlen</code> , <code>strtok</code> , <code>memcpy</code> , <code>memmove</code> , etc.
<code>math.h</code>	Common mathematical operations and transformations • <code>cos</code> , <code>sin</code> , <code>tan</code> , <code>acos</code> , <code>asin</code> , <code>atan</code> , <code>exp</code> , <code>log</code> , <code>pow</code> , <code>sqrt</code> , etc.
<code>errno.h</code>	Macros for reporting and retrieving error conditions through error codes stored in a static memory location called <code>errno</code> • <code>EDOM</code> (parameter outside a function's domain - <code>sqrt(-1)</code>), • <code>ERANGE</code> (result outside a function's range), or • <code>EILSEQ</code> (an illegal byte sequence), etc.

A header file typically contains manifest constants, macros, necessary struct / union types, typedef's, function prototype, etc.

Programming in Modern C++ Partha Pratim Das M43.7

So, quick look at some of the major common library components of C which we are regularly using `stdio.h`, `stdlib.h`, `string.h`, `math.h`, are 4 that almost every time we need.

(Refer Slide Time: 02:54)



```
/* math.h
 * This file has no copyright assigned and is placed in the Public Domain.
 * This file is a part of the mingw-runtime package.
 * Mathematical functions.
 */
#ifndef _MATH_H
#define _MATH_H
#ifdef __STRICT_ANSI__ // conditional exclusions for ANSI
// ...
#define M_PI 3.14159265358979323846 // manifest constant for pi
// ...
struct _complex { // struct of _complex type
    double x; /* Real part */
    double y; /* Imaginary part */
};
_CRTIMP double __cdecl _cabs(struct _complex); // cabs(.) function header
// ...
#endif /* __STRICT_ANSI__ */
// ...
_CRTIMP double __cdecl sqrt(double); // sqrt(.) function header
// ...
#define isfinite(x) ((fpclassify(x) & FP_NAN) == 0) // macro isfinite(.) to check if a number is finite
// ...
#endif /* _MATH_H */
```

Source: C math.h library functions Accessed 13-Sep-21

This is how a typical header would look like if you just go into your system and dig out a header I have dugout `math.h`. So, this is the compilation multiple inclusion guard present here which we learned about here are some constants defined as manifest constants because it is C. So, if you need the value of PI in a language, the developer can use `M_PI`, here is a complex structure defined with `_complex` name. This is the addition in C++ at a later point.

Here is a signature for the function square root that we regularly use and So, on. Of course, a lot of the code here I have just hidden just to give you a glimpse of the kinds of things that the standard library headers will have. Headers in C++ will, in addition, have classes and templated definitions.

(Refer Slide Time: 03:57)

Component	Data Types, Manifest Constants, Macros, Functions, Classes, ...
<code>iostream</code>	Stream input and output for standard I/O • <code>cout</code> , <code>cin</code> , <code>endl</code> , ..., etc.
<code>fstream</code>	(Module 42)
<code>string</code>	Manipulation of string objects • Relational operators, IO operators, iterators, etc.
<code>memory</code>	High-level memory management • Pointers: <code>unique_ptr</code> , <code>shared_ptr</code> , <code>weak_ptr</code> & <code>allocator</code> etc.
<code>exception</code>	Generic Error Handling • <code>exception</code> , <code>bad_exception</code> , <code>unexpected_handler</code> , & <code>terminate_handler</code>
<code>stdexcept</code>	Standard Error Handling • <code>logic_error</code> , <code>invalid_argument</code> , <code>domain_error</code> , <code>length_error</code> , <code>out_of_range</code> , <code>runtime_error</code> , <code>range_error</code> , <code>overflow_error</code> , <code>underflow_error</code> , etc.
STL	
Containers	
<code>vector</code>	<code>deque</code> , <code>list</code> , <code>stack</code> , <code>queue</code> , <code>priority_queue</code> , <code>set</code> , <code>multiset</code> , <code>map</code> , <code>multimap</code>
<code>C++11</code>	<code>array</code> , <code>forward_list</code> , <code>unordered_set</code> / <code>multiset</code> / <code>map</code> / <code>multimap</code>
Iterators	
<code>begin</code> & <code>end</code> , <code>rbegin</code> & <code>rend</code> , <code>C++11</code> : <code>cbegin</code> & <code>cend</code> , <code>crbegin</code> & <code>crend</code> .	
Algorithm	
<code>algorithm</code>	Non-Numerical : <code>for_each</code> , <code>find</code> , <code>find_if</code> , <code>count</code> , <code>search</code> , <code>copy</code> , <code>move</code> , <code>swap</code> , <code>replace</code> , <code>fill</code> , <code>generate</code> , <code>remove</code> , <code>reverse</code> , <code>rotate</code> , <code>sort</code> , <code>binary_search</code> , <code>merge</code> , <code>min</code> , <code>max</code> , ...
Numeric	
<code>numeric</code>	Numerical : <code>accumulate</code> , <code>adjacent_difference</code> , <code>inner_product</code> , and <code>partial_sum</code> <code>C++11</code> : <code>iota</code> , <code>equal_to</code> , <code>not_equal_to</code> , <code>greater</code> , <code>greater_equal</code> , <code>less</code> , <code>less_equal</code> ; <code>plus</code> , <code>minus</code> , <code>multiplies</code> , <code>divides</code> , <code>modulus</code> ; <code>logical_and</code> , <code>logical_not</code> , <code>logical_or</code> . <code>C++11</code> : <code>bit_and</code> , <code>bit_or</code> , <code>bit_xor</code>
Functions	
Imported from C Standard Library	
cmath	
Common mathematical operations and transformations	
• <code>cos</code> , <code>sin</code> , <code>tan</code> , <code>acos</code> , <code>asin</code> , <code>atan</code> , <code>exp</code> , <code>log</code> , <code>pow</code> , <code>sqrt</code> , etc.	
cstdlib	
Memory alloc., process control, conversions, pseudo-rand nos., searching, sorting	
• <code>malloc</code> , <code>free</code> , <code>exit</code> , <code>abort</code> , <code>atoi</code> , <code>strtol</code> , <code>rand</code> , <code>bsearch</code> , <code>qsort</code> , etc.	

Now, in C++ standard library if we want to take a look then some of the components which we have already done well is `iostream`, `fstream` all those I/O libraries which we have done in the last module itself, we have been using the `string` component very heavily we will make use of `memory` component `STL` we have not discussed that yet we have made use of the `exception` components and so on. So, there are a number of components which are very useful for us. What is special in C++ standard library is the support for `STL`. `STL` stands for Standard Template Library. `STL` is neither the full of the library.

It is kind of a subset by of the standard library, but it does provide support for quite a very few important and interesting features of this standard C++ library. The most useful of them include containers, like in C, you did not have a support for data structure only array is available as a language feature. Using pointer, we had to build up linked lists of every kind.

But anything beyond that, whether it is stack or it is a queue or a priority queue or if I want to do a `HashMap` anything, the programmer had to create his or her own library for that. So, that has been strongly facilitated in C++ by providing a number of containers, almost a complete set of what you will need, I mean, this kind of is resonates with what you have in Python, where you have five basic data structure given as a part of the Python language itself.

So, in terms of the containers, we have vectors, which we have already been using quite extensively in place of array, it is an array of flexible size, but the same efficiency, you have doubly linked list, you have stack queue, priority queue or heap data structure, you also have Set, Map, which is basically kind of hashing and So, on.

And C++11 has added a lot more components in that. So, each one of these container components are a header by itself and has a complete functionality for that data structure. So, using C++ the need for doing any basic data structure is almost not there, the standard library will help that. To support that containers we have a number of iterators which can go over the container and check at different elements, will understand what iterators are in more depth. We have components for common algorithms like find, copy for doing something for each element of a data structure and so on.

We have some numeric very useful flexible numeric algorithms, given numeric components given these are not the numerical computation that we know, but these are like simple numeric operations like summing the elements of a vector and those kind of. And we have something which is very, very special in terms of the different functions that are provided.

So, there are, we have studied about functors in module 40. So, using functors using function objects, a number of very useful functions are provided, which are in turn used in the algorithms component and the container components. So, this is the main chunk of C++ standard library to specially learn and that is what we will be focusing in this module and in the next two.

(Refer Slide Time: 08:08)

namespace std for C++ Standard Library

C Standard Library	C++ Standard Library
<ul style="list-style-type: none">• All names are global• <code>stdout, stdin, printf, scanf</code>	<ul style="list-style-type: none">• All names are within <code>std</code> namespace• <code>std::cout, std::cin</code>• Use <code>using namespace std;</code> <p>to get rid of writing <code>std::</code> for every standard library name</p>
W/o using	W/ using
<pre>#include <iostream> int main() { std::cout << "Hello World in C++" << std::endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int main() { cout << "Hello World in C++" << endl; return 0; }</pre>

Programming in Modern C++ Partha Pratim Das M43.10

Now, coming to besides that, obviously, we know that C standard library headers can also be used in C++ by prefixing the word the letters `c` with it and you have the entire C standard library functionality available though in C++, they may differ a little bit differently at times. Just a quick recap that in C every name is in a namespace is which is global, because C does not have the concept of separate namespaces.

So, all functions standard library functions are in the global namespace. So, you cannot have your own functions by the same name. Whereas C++ puts all standard library components under a namespace `std`, which you have to prefix before the standard library symbols or you can use the mean have the flexibility of doing the using command as we have seen earlier.

(Refer Slide Time: 08:50)

Standard Library: C/C++ Header Conventions

	C Header	C++ Header
C Program	Use <code>.h</code> . Example: <code>#include <stdio.h></code> <i>Names in global namespace</i>	Not applicable
C++ Program	Prefix <code>c</code> , no <code>.h</code> . Example: <code>#include <cstdio></code> <i>Names in std namespace</i>	No <code>.h</code> . Example: <code>#include <iostream></code>

- A C std. library header is used in C++ with prefix 'c' and without the `.h`. These are in `std` namespace:

```
#include <cmath> // In C it is <math.h>
...
std::sqrt(5.0); // Use with std::
```

It is possible that a C++ program include a C header as in C. Like:

```
#include <math.h> // Not in std namespace
...
sqrt(5.0); // Use without std::
```

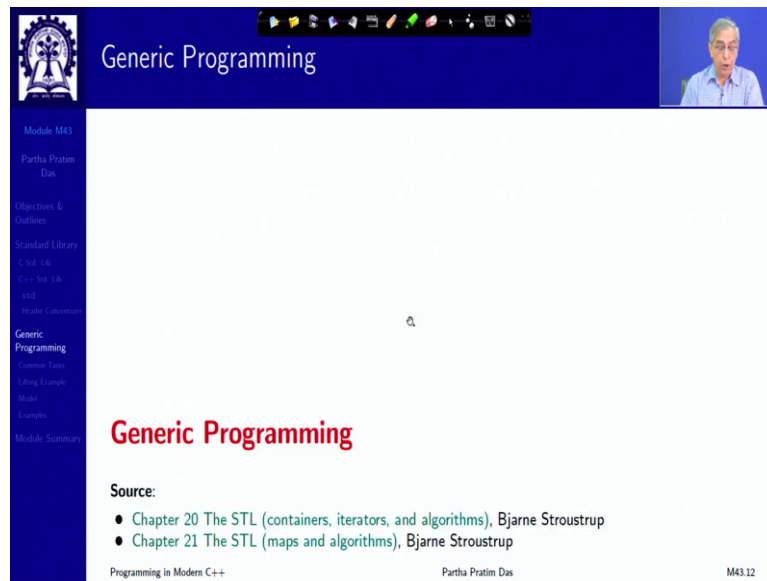
This, however, is not preferred
- **Using `.h` with C++ header files, like `iostream.h`, is disastrous. These are deprecated. It is dangerous, yet true, that some compilers do not error out on such use. Exercise caution.**

Programming in Modern C++ Partha Pratim Das M43.11

In terms of header names, all headers in C has an extension `.h` in the standard library, `stdio.h` whereas in C++, the standard library headers do not have this `.h` extension. If their standard library headers borrowed from the C standard library, then there named in the C standard library it is prefixed with `c` and the `.h` is dropped. So, `stdio.h` to be used in a C++ program must be included as `cstdio`.

Do not put that `.h` and the the pure C++ standard library headers do not have a `.h`, So, it is `iostream` simply not `iostream.h`, be very careful about this because in the older versions of C, the `.h` extension was there in the standard library. So, if you are using a little bit old system then it might have somewhere in the corner, some `iostream.h` header available. So, if you use that `.h` that wrong old file will be used. So, remember, in C standard library will always have `.h` in C++ standard library headers will never have `.h`.

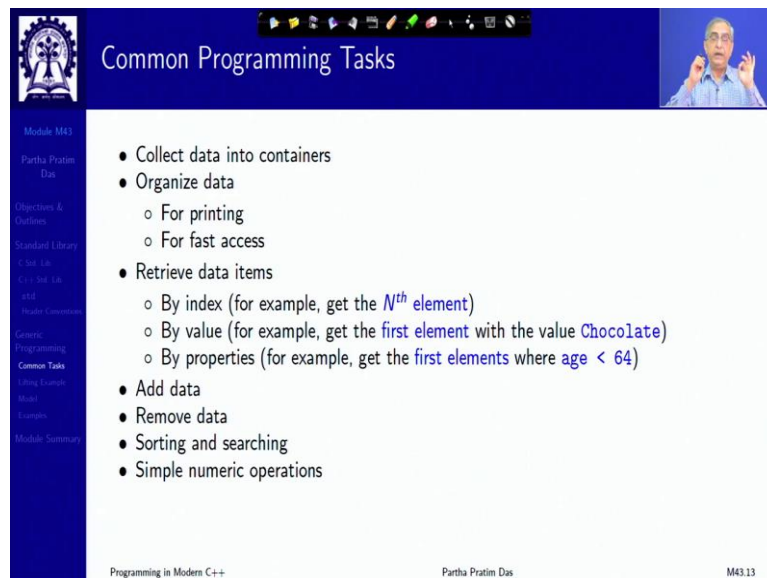
(Refer Slide Time: 10:02)



The slide is titled "Generic Programming" and features a navigation menu on the left side. The menu includes items such as "Module M43", "Partha Pratim Das", "Objectives & Outlines", "Standard Library", "C++ Lib", "C++ Std Lib", "STL", "Module Contents", "Generic Programming", "Common Tasks", "Library Examples", "Model", "Exercises", and "Module Summary". The main content area displays the title "Generic Programming" in red, followed by the text "Source:" and a list of two bullet points: "Chapter 20 The STL (containers, iterators, and algorithms), Bjarne Stroustrup" and "Chapter 21 The STL (maps and algorithms), Bjarne Stroustrup". At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "M43.12".

With these few words. Let me go over to discussing what is generic programming. Before we can get into the containers, iterators, algorithms, functions of the C++ standard library, we need to understand this concept of generic programming.

(Refer Slide Time: 10:25)



The slide is titled "Common Programming Tasks" and features a navigation menu on the left side. The menu includes items such as "Module M43", "Partha Pratim Das", "Objectives & Outlines", "Standard Library", "C++ Lib", "C++ Std Lib", "STL", "Module Contents", "Generic Programming", "Common Tasks", "Library Examples", "Model", "Exercises", and "Module Summary". The main content area displays a list of tasks: "Collect data into containers", "Organize data" (with sub-points "For printing" and "For fast access"), "Retrieve data items" (with sub-points "By index (for example, get the Nth element)", "By value (for example, get the first element with the value Chocolate)", and "By properties (for example, get the first elements where age < 64)"), "Add data", "Remove data", "Sorting and searching", and "Simple numeric operations". At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "M43.13".

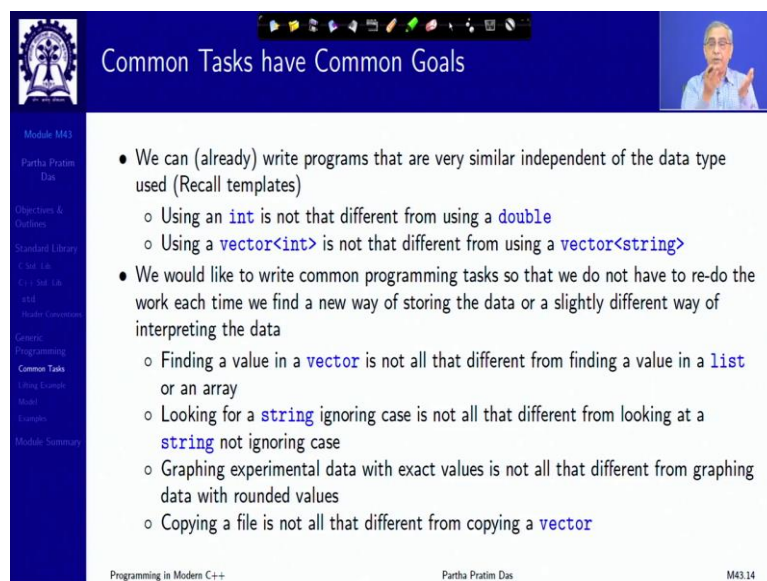
We mentioned about this earlier too. And in the context of C++, this is often also referred to as template metaprogramming. Because that is the mechanism through which generic programming is realized in C++. So, what is the purpose of generic programming? Consider the common programming tasks that we often have to do. What do we do, we think in a very

abstract way, we collect the data into containers, containers are nothing but data structures, which contain data.

And we organize them for different purposes like I might want to print the data or I might want to access the data at a very high speed, I may want to retrieve data items by different criterion, for example, I may want to retrieve a data item by an index or the position have the data in the data structure, or I might want to access that data by value that I want the value first element in my list, which has a value chocolate.

Or I might want to access a data by a property, you can say that well, from the collection of stood say person records, get me that record the first record where the age is less than 64. So, there could be several such but generically, we want to retrieve data items based on certain criteria, we want to add data, remove data, sort data, search data and so on, and do some simple arithmetic, numeric computation. So, these are things which, irrespective of which software project you are doing, you would be doing these or requiring to do this quite often.

(Refer Slide Time: 12:05)



Common Tasks have Common Goals

- We can (already) write programs that are very similar independent of the data type used (Recall templates)
 - Using an `int` is not that different from using a `double`
 - Using a `vector<int>` is not that different from using a `vector<string>`
- We would like to write common programming tasks so that we do not have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data
 - Finding a value in a `vector` is not all that different from finding a value in a `list` or an array
 - Looking for a `string` ignoring case is not all that different from looking at a `string` not ignoring case
 - Graphing experimental data with exact values is not all that different from graphing data with rounded values
 - Copying a file is not all that different from copying a `vector`

Programming in Modern C++

So, what we saw earlier is to facilitate these kinds of repeated tasks, we saw the use of templates, we have understood templates. So, we know that using an `int` is not very different from using a `double` or using a `vector` of `int`, is not very different from using a `vector` of `string` and so on. And so, we have seen that if we needed a stack of `int` or of `char` or of `string` or of a user defined data type, unlike C, I do not need to write the stack code, every time in C++.

I can use the stack component from the C++ standard library, set the data element type to be of the appropriate type, whether it is int or char, and the template automatically sets the type for the container underlying container element of the stack, it sets the type for the different push pop operations and so on.

So, using template, we have been able to generalize the different data structure different classes based on certain templated type parameters. So, this is one level of generalization that we have been able to do. But still there is more that is possible. For example, suppose I want to find a value in a data structure. Now, I that data structure could be a vector, the data structure could be a list, naturally, the code typically would be different.

The same code cannot do this, of the kind of programming we know So, far. But conceptually, if you think is it significantly different as to whether I am finding the value in a vector or in a list, it is a collection. So, I want to find the value in that collection. So, the use of template first gave us the liberty I mean liberation from the underlying element, data type and so on.

But I want more liberty that conceptually, if an operation is applicable for multiple different containers, multiple different data structures, I should not be able to write a common code to do that operation. I want to I am looking for strings in a collection. Now, whether I look for strings with case sensitivity or in a case insensitive manner, will mean certain difference in the code. But conceptually, it is not very different. Conceptually, I am looking for strings, that is all.

(Refer Slide Time: 14:58)

Ideals for Commonly used Common Codes

- Code that is
 - Easy to read
 - Easy to modify
 - Regular
 - Short
 - Fast
- Uniform access to data
 - Independently of how it is stored
 - Independently of its type
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
 - Retrieval of data
 - Addition of data
 - Deletion of data
- Standard versions of the most common algorithms
 - Copy, find, search, sort, sum, ...

Programming in Modern C++ | Partha Pratim Das | M43.15

So, generic programming or commonly tries to find the common code with certain characteristics. So, what it should be? The characteristics that are desirable is ideals we say is the code should be easy to read, easy to modify, it should be more or less regular, short, fast and so on. The access to the data should be uniform independent of how it is stored, I am working with a stack I should not be bothered about whether the elements are stored in an array or in a linked list.

It should be independent of that, I am doing a search on a data structure, I should not be bothered about whether the data is in a vector or is in a list, whether the list is a single linked list or is a doubly linked list and so on. And of course, it should be independent of its type. C++ is strongly typed. So, access must be typesafe it should be easy to traverse the data the storage must be compact retrieval, addition, deletion, everything should be fast.

And there should be standard versions of common tasks like copying, search, finding, sorting and so on so forth. Can we write algorithms or codes in that level of generality? If we can and that is what is called generic programming.

(Refer Slide Time: 16:26)

Examples

- Sort a vector of strings
- Find a number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than "Peterson"?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pairwise product of the elements of two sequences
- What are the highest temperatures for each day in a month?
- What are the top 10 best-sellers?
- What is the entry for "C++" (say, in Google)?
- What is the sum of the elements?

Module M43
Partha Pratim Das
Objectives & Outlines
Standard Library
C++ Lib
C++ STL Lib
std
Header Summaries
Generic Programming
Common Tasks
Using Lambda
Map
Example
Module Summary

Programming in Modern C++ Partha Pratim Das M43.16

So, here, just a jumble list of some common tasks like sorting a vector of strings, finding the first value larger than “Peterson” and what is the entity for C++ say, when you search in Google and so on so forth, which are generic in nature, but comes from a wide variety of domains, for a wide variety of types, for a wide variety of possible values and So, on.

(Refer Slide Time: 17:04)

Generic Programming

- Generalize algorithms
 - Sometimes called *lifting an algorithm*
- The aim (for the end user) is
 - Increased correctness
 - ▷ Through better specification
 - Greater range of uses
 - ▷ Possibilities for re-use
 - Better performance
 - ▷ Through wider use of tuned libraries
 - ▷ Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
 - The other way most often leads to bloat

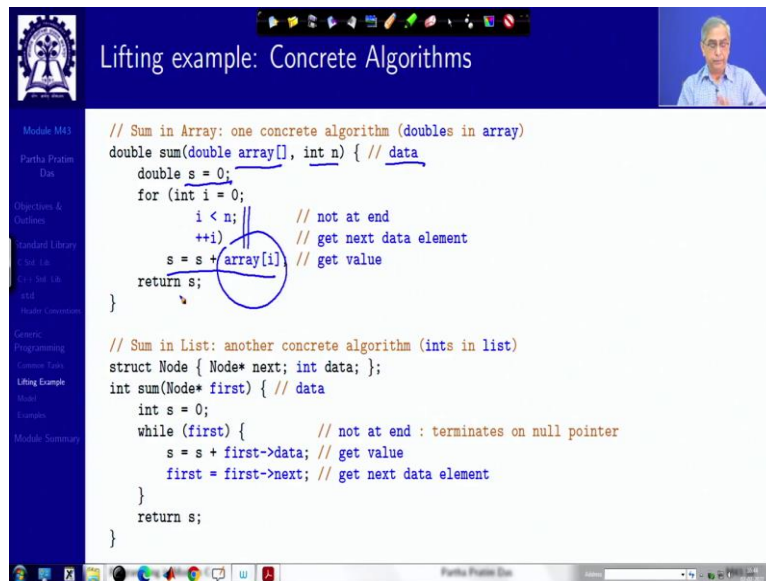
Programming in Modern C++ Partha Pratim Das M43.17

So, instead of having to write separate specific code for these tasks, can we generalize? That is the basic question that we are trying to answer in the generic programming. So, generic programming tries to generalize the algorithm it is also, called lifting an algorithm. So, that its correctness can be increased, because you are generalizing you are specifying it in a very compact form, and possibly once to be used a million times afterwards.

So, a better specification must be possible, there should be greater range of use, that is the basic purpose and the performance must be wisely usable for tuned libraries. So, for doing this, we try to go from concrete algorithms that we write down in terms of C or C++ code, to more abstract forms of algorithms.

So, as you come to concrete, things start bloating up, the code for stack, for input, for stack, for a double, bloated up, we made an abstraction to made it a, made the element type to be a type parameter in the template, it became compact. Now, it still can bloat if it is a stack of vector. Stack, realized by a vector or a stack realized by a doubly linked list. Can I abstract it by just saying that it is a container, where I can put an element take out an element, check if the container is empty, and so on so forth.

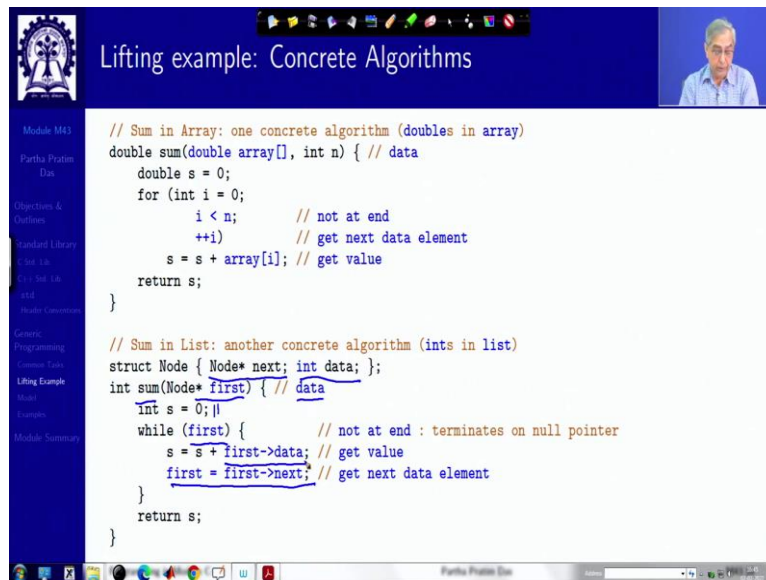
(Refer Slide Time: 18:35)



Lifting example: Concrete Algorithms

```
// Sum in Array: one concrete algorithm (doubles in array)
double sum(double array[], int n) { // data
    double s = 0;
    for (int i = 0;
        i < n; // not at end
        ++i) // get next data element
        s = s + array[i]; // get value
    return s;
}

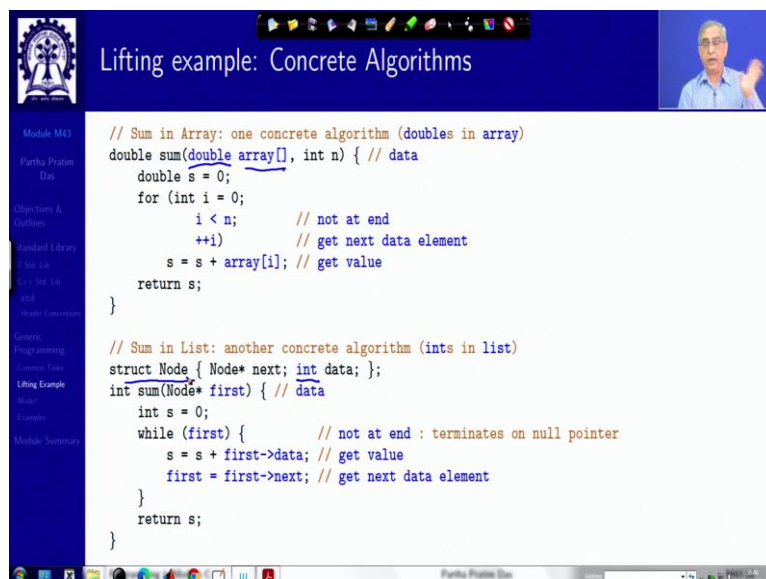
// Sum in List: another concrete algorithm (ints in list)
struct Node { Node* next; int data; };
int sum(Node* first) { // data
    int s = 0;
    while (first) { // not at end : terminates on null pointer
        s = s + first->data; // get value
        first = first->next; // get next data element
    }
    return s;
}
```



Lifting example: Concrete Algorithms

```
// Sum in Array: one concrete algorithm (doubles in array)
double sum(double array[], int n) { // data
    double s = 0;
    for (int i = 0;
        i < n; // not at end
        ++i) // get next data element
        s = s + array[i]; // get value
    return s;
}

// Sum in List: another concrete algorithm (ints in list)
struct Node { Node* next; int data; };
int sum(Node* first) { // data
    int s = 0;
    while (first) { // not at end : terminates on null pointer
        s = s + first->data; // get value
        first = first->next; // get next data element
    }
    return s;
}
```



Lifting example: Concrete Algorithms

```
// Sum in Array: one concrete algorithm (doubles in array)
double sum(double array[], int n) { // data
    double s = 0;
    for (int i = 0;
        i < n; // not at end
        ++i) // get next data element
        s = s + array[i]; // get value
    return s;
}

// Sum in List: another concrete algorithm (ints in list)
struct Node { Node* next; int data; };
int sum(Node* first) { // data
    int s = 0;
    while (first) { // not at end : terminates on null pointer
        s = s + first->data; // get value
        first = first->next; // get next data element
    }
    return s;
}
```


So, that is the basic, lifting the algorithm is a basic idea, and I give you an example. Suppose we are trying to do, we are trying to add the numbers in a data structure, let the data structure be an array. So, that is the data given to us. And, well, since it is an array, I need to also tell what is the size of the array, I tell that and then how do I add the elements, I will have an accumulating variable s , says where I keep the sum, which I initialized to 0, and then I keep on traversing on the array.

And there are two steps one is $i < n$, which checks if I have reached the end of the array, because there are n elements, I can go up to $n - 1$. And there is $++i$, which takes me from the current index to the next index. That is it gets the next element. And as I have got the next element, I add it to the accumulated variable s , the sample code, C code. All of us have seen this a number of times.

Think about an equivalent code, equivalent way of doing this for a list. Let us say it is a Singly Linked List, which has a node which points to the next node contains the data and you write a similar function some passing the header of the node. So, which is the marker of the data. Now you have a similar accumulator for, for s which will have the sum. Now, you start with the first header and you keep on traversing the linked list we know.

So, what we have the header, we go to the next pointer, get the next value, go to the next pointer, get the next value till we are at the end of the list where typically customarily we will have a null value assigned to mean that the list has ended. So, we can go to the next element by doing this operation, changing first from first to first point and next.

And any when at a certain point, then the value that I need to access is first pointer data. Now, if you compare these two, then you see there are differences in terms of the first underlying type of data, double and int. There is differences in terms of the data structure. One is a list and other is an array.

But if you look at the loop in general, you can see that it is almost similar requiring three basic operation that to be able to check that you are not at the end, to be able to start at the beginning and then check that you are not at the end. And this has to be a way to go to the next element. And there has to be a way to access that element and get that value to do that accumulation.

(Refer Slide Time: 21:40)

Lifting example: Abstract the data structure

```
// pseudo-code for a more general version of both algorithms

int sum(data) {           // somehow parameterize with the data structure
    int s = 0;             // initialize
    while (not at end) {  // loop through all elements
        s = s + get value; // compute sum
        get next data element;
    }
    return s;             // return result
}
```

- We need three operations (on the data structure):
 - not at end
 - get value
 - get next data element

Lifting example: Abstract the data structure

```
// pseudo-code for a more general version of both algorithms

int sum(data) {           // somehow parameterize with the data structure
    int s = 0;             // initialize
    while (not at end) {  // loop through all elements
        s = s + get value; // compute sum
        get next data element;
    }
    return s;             // return result
}
```

- We need three operations (on the data structure):
 - not at end
 - get value
 - get next data element

So, we can lift these two concrete algorithms into an abstract algorithm to write it in a in terms of a pseudo code, we say okay, given the data, which could be array or list whatever, it is a while loop, which continues till the end is reached every time I check the current value-added to s and then go to the next element. If we can do that, if we can do these three steps, then we will be able to easily, generalize both these concrete algorithms into an abstraction and that is what we are trying to do here a specific element type has been used, which, can be very easily liberated by using a templated type.

(Refer Slide Time: 22:26)

```
// Concrete STL-style code for a more general version of both algorithms

template<class Iter, class T> // Iter should be an Input_iterator
                             // T should be something we can + and =
T sum(Iter first, Iter last, T s) { // T is the accumulator type
    while (first != last) { // not at end
        s = s + *first; // get value
        ++first; // get next data element
    }
    return s;
}

• Let the user initialize the accumulator

float a[] = { 1,2,3,4,5,6,7,8 }; // a[0], a[0], ..., a[7]
double d = 0;
d = sum(a, a+sizeof(a)/sizeof(*a), d); // [&a[0],a[8]) = {a[0], a[0], ..., a[7]}
```

```
// Concrete STL-style code for a more general version of both algorithms

template<class Iter, class T> // Iter should be an Input_iterator
                             // T should be something we can + and =
T sum(Iter first, Iter last, T s) { // T is the accumulator type
    while (first != last) { // not at end
        s = s + *first; // get value
        ++first; // get next data element
    }
    return s;
}

• Let the user initialize the accumulator

float a[] = { 1,2,3,4,5,6,7,8 }; // a[0], a[0], ..., a[7]
double d = 0;
d = sum(a, a+sizeof(a)/sizeof(*a), d); // [&a[0],a[8]) = {a[0], a[0], ..., a[7]}
```

So, if I try to write this in the form of a template, I can say that, well, I will have something like, like a pointer to the data, which I called Iter, or iterator. So, I will have a class iterator which can point to any data element. Now, mind you, this pointing mechanism would be maybe different between an array and a linked list. Because the way I go to the next element in an array, and the way I go to the next element in a linked list are different.

But I can generically think that there is a pointer like thing, which say, if I have these data elements, So far I am talking about a linear data structure. So, I can there is a sequencing order in this, I am not saying whether they are indexed like this or whether they are linked like this either that is possible. And I say that I have a iterator, which points to the first. So, it is for the array it is a[0] for the list, it is a header.

And then I say that I have something like a node which is beyond my last element one beyond my last element. So, for an array of size n it is the location $a[n]$. For a linked list, it is a value null, which is where the next element should have been. So, I call this as my first and I call this as my last. This is a basic process of iteration.

And then I have a template parameter T which tells me the element type of which the sum type will have to be the same. So, what I do is I would say that I have an operator++, if you think about pointer, you will understand is very immediately. I have an operator++ which takes my pointer, the iterator from the current position to the next, then again current that position to the next So, that takes me to the to get the next data element.

I have a $*$ operator thinking in terms of as if it is a pointer, it is again similar it is basically dereferencing whatever that iterator is pointing to I can get that element. So, that is get value. And how long do I continue? I will continue till from the first is going till it becomes equal to last, when this becomes equal to last the when first becomes equal to last I know that have gone past the last element because it is one beyond the last element.

So, I will continue till first is not equal to last. So, I should be there should be a way to compare these two iterators leads to values of the iterators and this like comparing pointers as you can think. So, with that a this becomes a generic code which can work in different contexts provided I can support this, ++, $*$ and $!=$ operators in a proper manner I show it use in in case of a of a concrete array where there is an array of 8 elements.

So, this is my begin Iter which actually a is nothing but address of a 0. This is the size the number of elements which is I know there are 8 elements. So, it is 1 beyond $a[8]$, is the address of $a[8]$, sorry there should be address of, is the address of $a[8]$, but it just does not go there goes one before that, and I will go up to that this is my end last Iter.

(Refer Slide Time: 26:42)

Lifting example

- Almost the standard library accumulate
 - A bit for terseness is simplified
- Works for
 - arrays
 - **vectors**
 - **lists**
 - **istreams**
 - ...
- Runs as fast as *hand-crafted* code
 - Given decent inlining
- The code's requirements on its data has become explicit
 - We understand the code better

Programming in Modern C++ Partha Pratim Das M43.21

So, with this generalization, now, I can easily do things uniformly for arrays, for vectors, for lists, and so on.

(Refer Slide Time: 26:53)

Basic Model: Algorithms ==> Iterators <== Containers

Algorithms (sort, find, search, copy, ...)

Iterators

Containers (vector, list, map, deque, ...)

- **Separation of Concerns**
 - *Algorithms* manipulate data, but do not know about *Containers*
 - *Containers* store data, but do not know about *Algorithms*
 - *Algorithms* and *Containers* interact through *Iterators*
 - Each *Container* has its own *iterator types*

Partha Pratim Das

And that will be as efficient as doing it in the handcrafted code. So, for this C++ standard library provides a generic model, which say that, well, what are the things that I need to do? I will have to do different I will have different containers as in here, these are the different data structures, I can have a vector of a list, as a map and so on. I have different algorithms to perform, I want to do a search, I want to do a find, I want to do a copy, I want to do a sort and so on. So, these are different algorithms.

(Refer Slide Time: 29:04)

Basic Model: Algorithms + Iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is *one past the last element*
 - not *the last element*
 - That is necessary to elegantly represent an empty sequence
 - One-past-the-last-element is not an element
 - ▷ You can compare an iterator pointing to it
 - ▷ You cannot dereference it (read its value)
- Returning the end of the sequence is the standard idiom for *not found* or *unsuccessful*

some iterator: [] the end: []

An empty sequence: begin: [] end: []

Programming in Modern C++ Partha Pratim Das M43.24

So, algorithms can now be written just thinking of the iterators. So, whatever you need to do, you just do not have to think whether you are going over an array or a list or a set or anything, you just think that the iterator, call the iterator operation, and that will give you the next element to work with. You can work simply with that.

(Refer Slide Time: 29:28)

Basic Model: Containers + Iterators

vector []

list doubly linked []

set kind of tree []

Partha Pratim Das

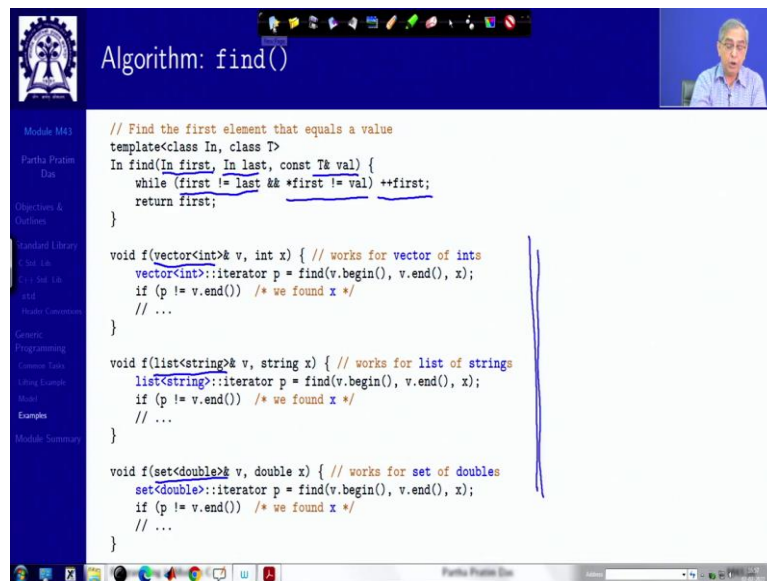
Similarly, containers do not need to think about the algorithms all that they need to support the iterator and here we show that the support can be provided not only for the sequence iterators sequence containers like vector, which is kind of an array or a list, which is linked

singly or doubly. But it can be provided for a nonlinear data structure like set. Set is a collection of unique elements. So, it is typically kept in the form of a binary search tree.

So, in that binary search tree, if we do a in order traversal, then naturally the nodes are traversed in a certain way left, then parent then right, if we do that. So, the traversal order here would be I will start from here, I will call left left left until I get to a left node, this is the first iteration value this is the first element I will get, these are first of the iterator.

Then I will have this element on ++, this element on ++, next ++ is here, next ++ is this this this this call, because it keeps on going down, then it is this element then it is this element, then it is this element and then it does not have anything where it should. So, that is my end of the iteration. So, you can see that even when a data structure is nonlinear, it can be linearized through this process of iteration very simply.

(Refer Slide Time: 30:55)



```
Algorithm: find()

// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val) {
    while (first != last && *first != val) ++first;
    return first;
}

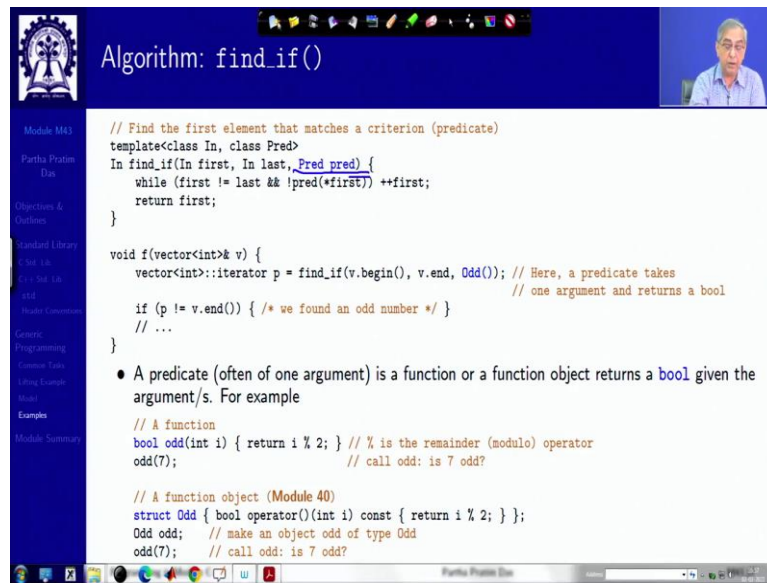
void f(vector<int>& v, int x) { // works for vector of ints
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end()) /* we found x */
        // ...
}

void f(list<string>& v, string x) { // works for list of strings
    list<string>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end()) /* we found x */
        // ...
}

void f(set<double>& v, double x) { // works for set of doubles
    set<double>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end()) /* we found x */
        // ...
}
```

So, using that we can have very nice algorithms written for example, if you compare these codes to find it is one is for vector of int, one is for list of strings, and one is for set of doubles, you can see that the code is practically identical, because all that it needs to know is the first and last of the iterator and the type of the value it has to look for, which is all templated and all that you are doing is you are checking that you are covering the entire data structure. You are checking if the current iteration value, which is start first equals the value given. If it does not, then you go to the next one. And you keep on doing that you can see that all this code are perfectly identical.

(Refer Slide Time: 31:46)



The slide is titled "Algorithm: find_if()". It contains the following C++ code:

```
// Find the first element that matches a criterion (predicate)
template<class In, class Pred>
In find_if(In first, In last, Pred pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}

void f(vector<int>& v) {
    vector<int>::iterator p = find_if(v.begin(), v.end(), Odd()); // Here, a predicate takes
                                                                // one argument and returns a bool
    if (p != v.end()) { /* we found an odd number */ }
    // ...
}

// A function
bool odd(int i) { return i % 2; } // % is the remainder (modulo) operator
odd(7); // call odd: is 7 odd?

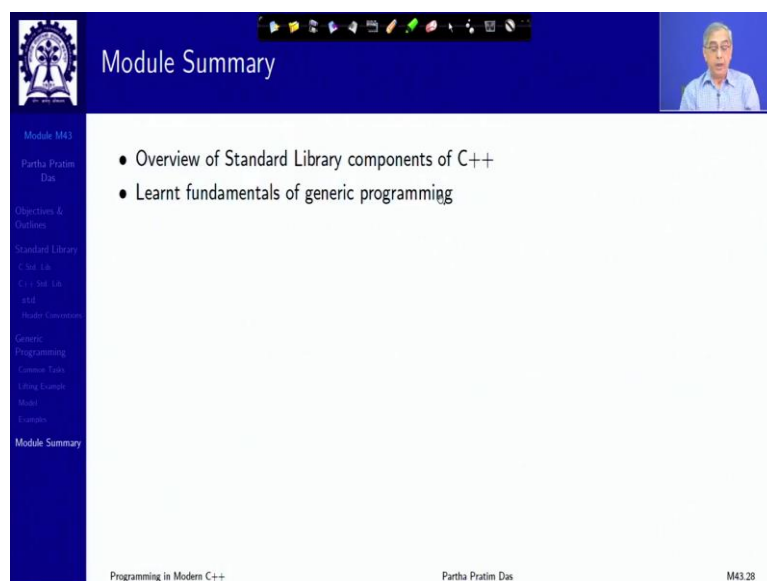
// A function object (Module 40)
struct Odd { bool operator()(int i) const { return i % 2; } };
Odd odd; // make an object odd of type Odd
odd(7); // call odd: is 7 odd?
```

A bullet point defines a predicate: "A predicate (often of one argument) is a function or a function object returns a bool given the argument/s. For example".

So, that is a big advantage that we get by doing this kind of generic programming this is another where if you are doing find and you can even generalize it further saying that find finds the element which is equal to the given element, but I can say that I do not just want to find “PPD”, I want to find the person who is teaching programming in modern C++.

So, I can provide a condition statement predicate for that, which can be provided as a as a functor. As you can see here, the function object as you can see here, we will talk more about this as we go forward.

(Refer Slide Time: 32:30)



The slide is titled "Module Summary". It contains the following bullet points:

- Overview of Standard Library components of C++
- Learnt fundamentals of generic programming

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M43.28".

So, this was a basic introduction to standard library component, particularly the generic programming side of it. So, that we can subsequently discuss the actual STL which will be the topic for the next module. Thank you very much for your attention and see you in the next module.