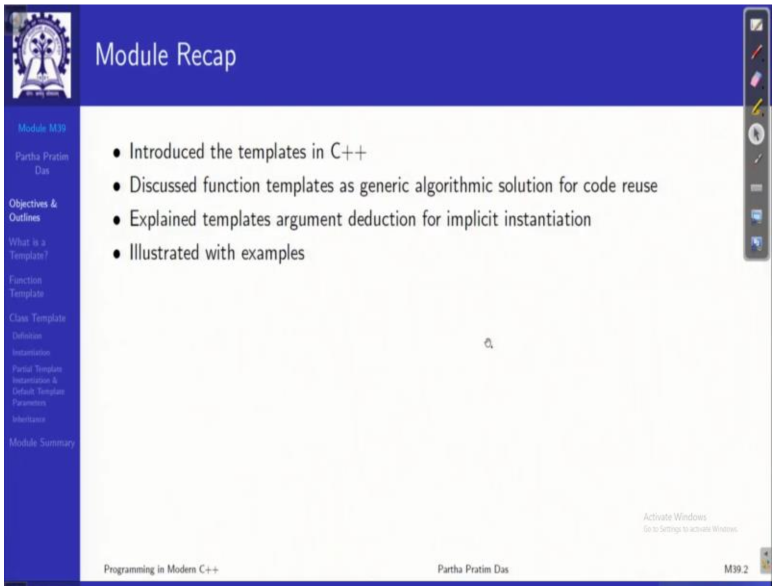
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 39
Lecture 39: Template (Class Template): Part 2

We are discussing Programming in Modern C++, welcome to that. We are in Week 8 and going to cover Module 39.

(Refer Slide Time: 00:37)

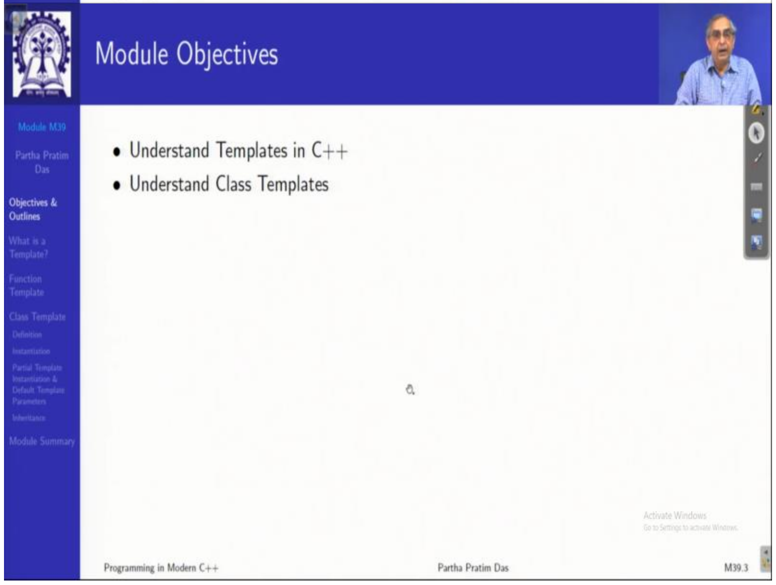


Module Recap

- Introduced the templates in C++
- Discussed function templates as generic algorithmic solution for code reuse
- Explained templates argument deduction for implicit instantiation
- Illustrated with examples

Module M39 | Partha Pratim Das | Objectives & Outlines | What is a Template? | Function Template | Class Template | Definition | Instantiation | Partial Template Instantiation & Default Template Parameters | Inheritance | Module Summary

Programming in Modern C++ | Partha Pratim Das | M39.2



Module Objectives

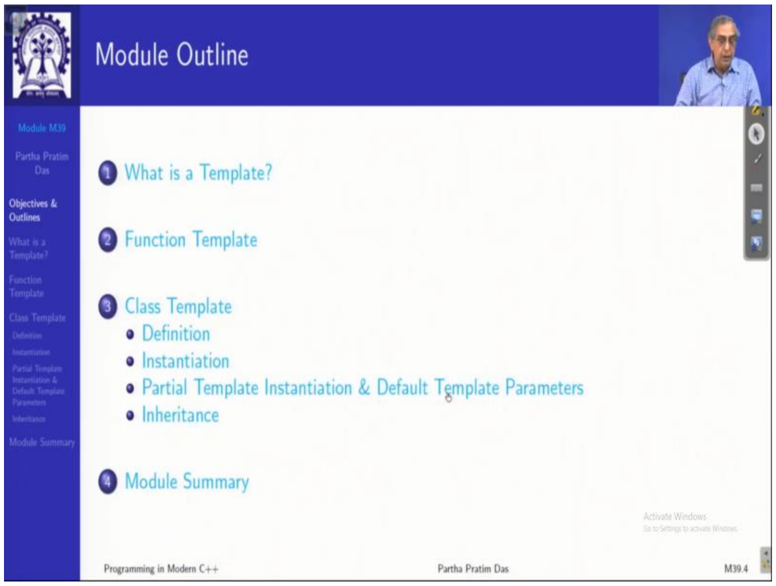
- Understand Templates in C++
- Understand Class Templates

Module M39 | Partha Pratim Das | Objectives & Outlines | What is a Template? | Function Template | Class Template | Definition | Instantiation | Partial Template Instantiation & Default Template Parameters | Inheritance | Module Summary

Programming in Modern C++ | Partha Pratim Das | M39.3

In the last module, we have introduced templates in C++, that is the backbone of generic programming, meta programming or generative programming, and discussed function templates as general algorithmic solutions for code reuse, explained templates with the deduction for implicit instantiation, illustrated with examples, we will continue with that and understand what is class template.

(Refer Slide Time: 01:16)

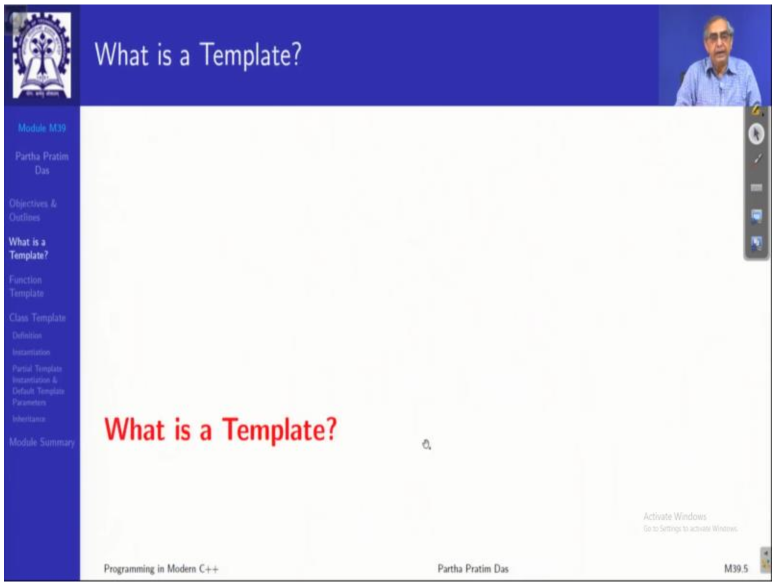


Module Outline

1. What is a Template?
2. Function Template
3. Class Template
   - Definition
   - Instantiation
   - Partial Template Instantiation & Default Template Parameters
   - Inheritance
4. Module Summary

This is the outline which will be available on your left as always.

(Refer Slide Time: 01:22)

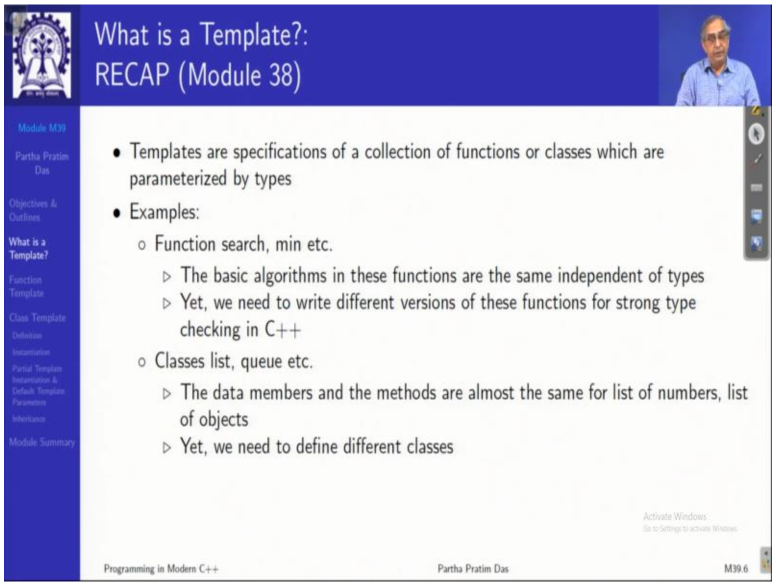


What is a Template?

What is a Template?

Now, before we get into the class template, let us quickly revisit what we have already done.

(Refer Slide Time: 01:32)



What is a Template?: RECAP (Module 38)

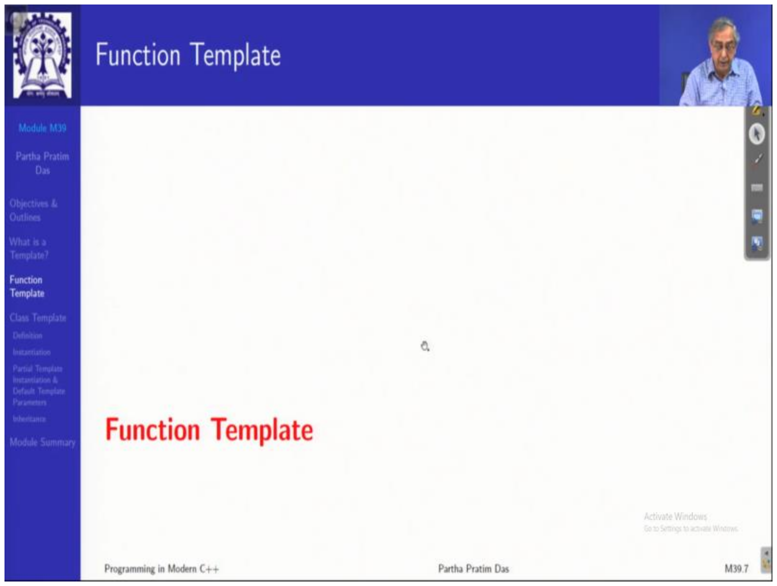
- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
  - Function search, min etc.
    - ▷ The basic algorithms in these functions are the same independent of types
    - ▷ Yet, we need to write different versions of these functions for strong type checking in C++
  - Classes list, queue etc.
    - ▷ The data members and the methods are almost the same for list of numbers, list of objects
    - ▷ Yet, we need to define different classes

Programming in Modern C++ Partha Pratim Das M39.6

What is a template? Template is a specification of collection of functions or classes. Functions is, are what you have already seen, which are parameterized by type, not like normal functions, which are parameterized by value. So, this is useful for functions which are generically having a common algorithm, like search, finding minimum, finding maximum, sorting and so on, but we need to have different versions of those functions implemented for various element types as maybe used in those functions, because of the strong type checking in C++.

Similarly, we can have classes primarily used for supporting data structures, containers most often, like lists, queues, stacks, etc, where data methods, methods and data members are almost the same for all say, list members, all kind of stacks, but their types, the element types do differ. So, we need to define different classes.

(Refer Slide Time: 02:52)



Function Template

Module M39

Partha Pratim Das

Objectives & Outlines

What is a Template?

Function Template

Class Template

Definition

Instantiation

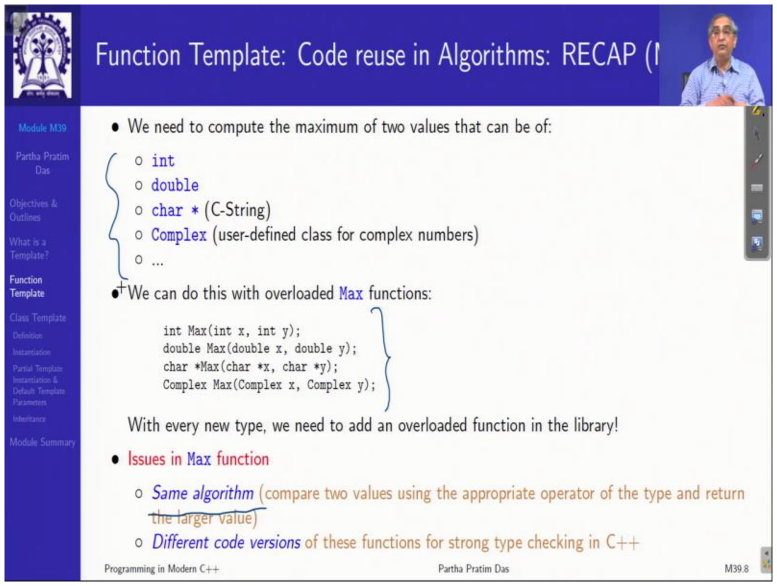
Partial Template Instantiation & Default Template Parameters

Inheritance

Module Summary

Function Template

Programming in Modern C++ Partha Pratim Das M39.7



Function Template: Code reuse in Algorithms: RECAP (M

- We need to compute the maximum of two values that can be of:
  - int
  - double
  - char * (C-String)
  - Complex (user-defined class for complex numbers)
  - ...
- We can do this with overloaded Max functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

With every new type, we need to add an overloaded function in the library!

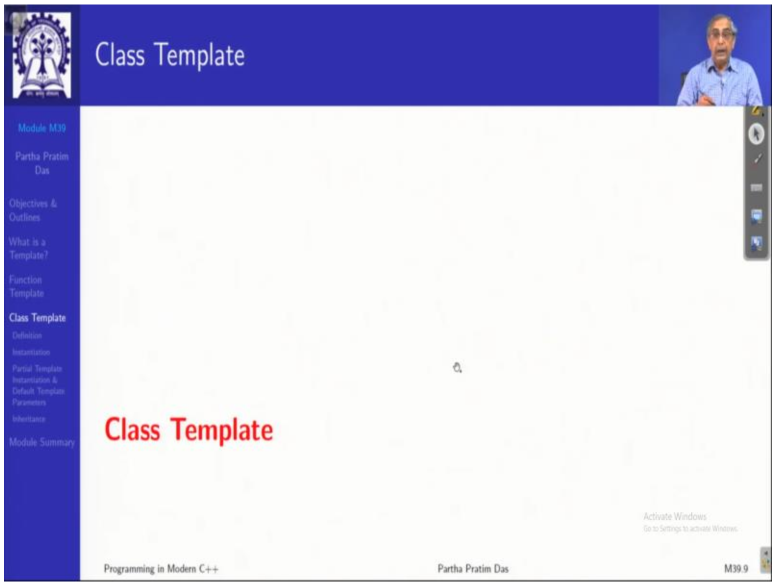
- Issues in Max function
  - *Same algorithm* (compare two values using the appropriate operator of the type and return the larger value)
  - *Different code versions* of these functions for strong type checking in C++

Programming in Modern C++ Partha Pratim Das M39.8

So, to work around that, we have seen in the last module that we can that we can have say, function template. So, to compute the value, which is greater of two given values, that is a maximum, we need to deal with integer type values, double type values, string type values, which in C is represented as char* or maybe user-defined type of values like complex. And we have seen that we can do that by using a number of overloaded Max functions. And, but for each of these types of Max function, we can, we need to define their body, need to define their implementation.

So, these are not only overloaded, but they need separate implementations, and function template is a mechanism by which we can avoid writing their body for each and every type, because all of them use the same algorithm. Finding the maximum of two numbers have the same algorithm. But the code versions are different, because that underlying types could be all of these different ones. And naturally, since this could also be user-defined types, it is not possible to have such overloaded functions in the library and user will have to, for each and every type user has to deal with, user has to write own Max function.

(Refer Slide Time: 04:26)



Class Template

Module M39

Partha Pratim Das

Objectives & Outlines

What is a Template?

Function Template

Class Template

Definition

Instantiation

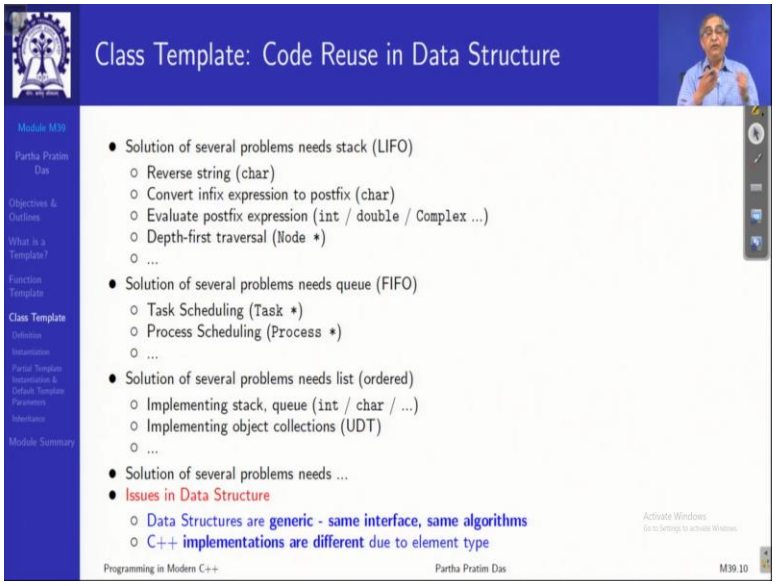
Partial Template Instantiation & Default Template Parameters

Inheritance

Module Summary

Class Template

Programming in Modern C++ Partha Pratim Das M39.9



Class Template: Code Reuse in Data Structure

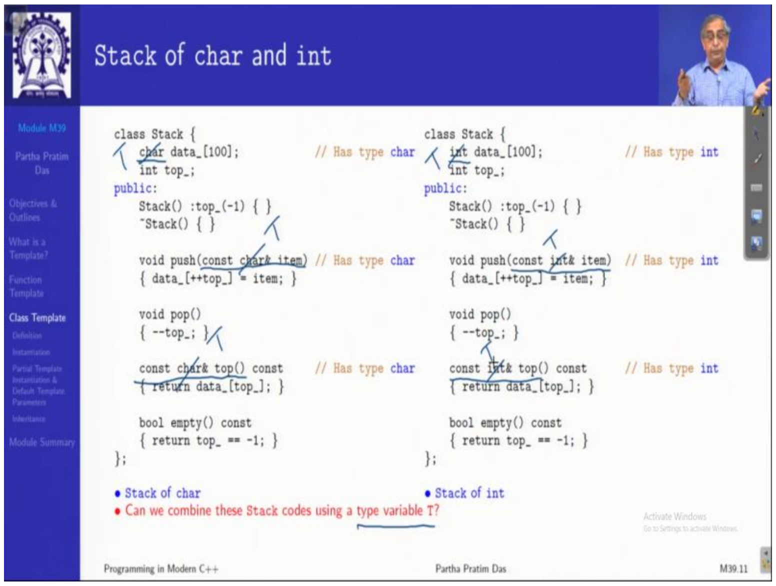
- Solution of several problems needs stack (LIFO)
  - Reverse string (`char`)
  - Convert infix expression to postfix (`char`)
  - Evaluate postfix expression (`int` / `double` / `Complex` ...)
  - Depth-first traversal (`Node *`)
  - ...
- Solution of several problems needs queue (FIFO)
  - Task Scheduling (`Task *`)
  - Process Scheduling (`Process *`)
  - ...
- Solution of several problems needs list (ordered)
  - Implementing stack, queue (`int` / `char` / ...)
  - Implementing object collections (UDT)
  - ...
- Solution of several problems needs ...
- Issues in Data Structure
  - Data Structures are **generic - same interface, same algorithms**
  - C++ **implementations are different** due to element type

Programming in Modern C++ Partha Pratim Das M39.10

Class template, like the function template we are solving, class template is designed to solve similar problems in similar situations. For example, solution of several problems needs stack, last in, first out container structure for reversing string or converting infix expression to postfix, to evaluate a post fix expression, which could again be done for integer double complex and so on for depth-first traversal, we need FIFO or queue for several different task scheduling problems, process scheduling problems and so on.

We need list for implementation of several data structures and even for several normal data maintenance activities and so on. So, variety of data structures have common functionality in them, but need to be implemented over and over again, because their underlying data type element type and the associated types differ. So, data structures in general are thought to be generic. They have the same interface, the same algorithm, plus C++ implementations are different. This is the problem which the class template actually addresses.

(Refer Slide Time: 05:44)



Stack of char and int

```
class Stack {
    char data_[100];          // Has type char
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const char& item) // Has type char
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const char& top() const    // Has type char
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

```
class Stack {
    int data_[100];           // Has type int
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const int& item) // Has type int
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const int& top() const     // Has type int
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

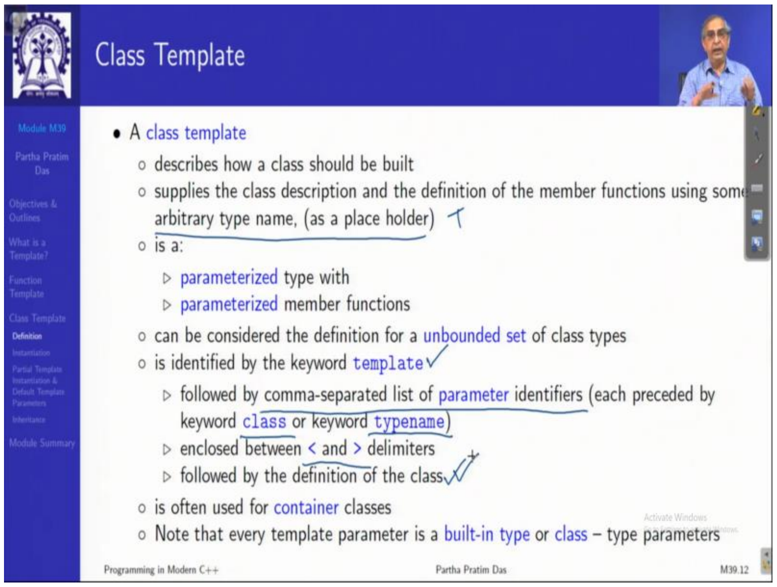
- Stack of char
- Stack of int
- Can we combine these Stack codes using a type variable T?

Programming in Modern C++ Partha Pratim Das M39.11

So, just to give you a more concrete example here, we have side by side the stack implementation the, on left, you have a stack of character, and on right, you have a stack of integer. So, you can see that these are the two points where the code differs. Similarly, when I want to push, I need to push a character to the stack here, whereas I need to push an integer to the stack here. Pop, or for that matter, constructor, destructor has nothing different, pop also has nothing different. But when I do top, I get a character reference, constant character reference here, whereas for a stack of integer, I get a constant integer reference. Again, empty does not differ.

So, these places, these three places where the code differ, because the underlying data type in that case, left case, is char and on the right case is int. So, the question is, can I not replace this char and int by type variable T? Could I just say that this is T, this is T, this is T. If I write that, this is T, this is T, this is T. If I write that, then the code on the left and the right become identical, and that is the class template. So, all that I do is I make the type a variable and then make an option that when I need a stack, I will tell whether I need a stack of char or of int or of anything else. That is a basic idea of class template.

(Refer Slide Time: 07:34)



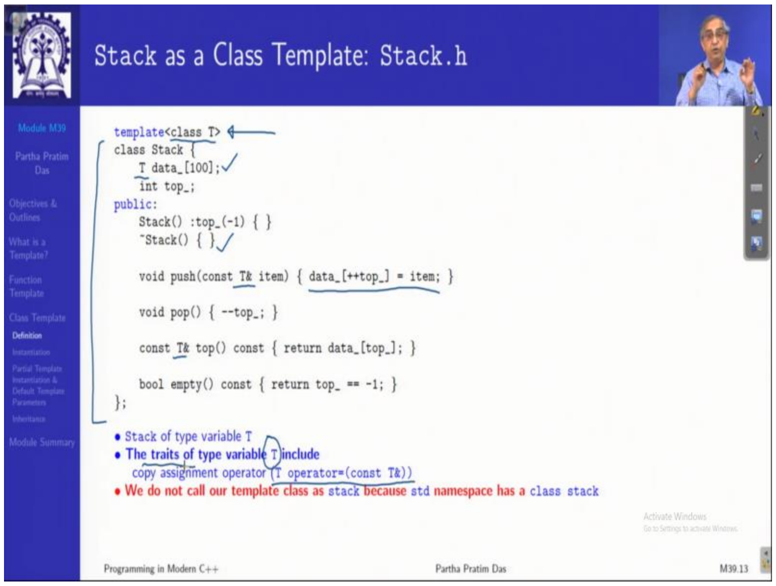
Class Template

- A class template
  - describes how a class should be built
  - supplies the class description and the definition of the member functions using some arbitrary type name, (as a place holder) T
  - is a:
    - parameterized type with
    - parameterized member functions
  - can be considered the definition for a unbounded set of class types
  - is identified by the keyword template
    - followed by comma-separated list of parameter identifiers (each preceded by keyword class or keyword typename)
    - enclosed between < and > delimiters
    - followed by the definition of the class
  - is often used for container classes
  - Note that every template parameter is a built-in type or class – type parameters

So, a class template describes how a class should be built, it is data members interfaces, supplies the description, definition of the data members using some arbitrary type name, using some arbitrary type name, like the T we just saw. So, it is parameterized with type, parameterized member functions, and it can be considered, for example, a, an unbounded set of class types, because you can have unbounded instantiations for the type T, the type variable, and for every such type, you will have a class being generated from the same template.

In order to express that, we use the keyword template, like we did in case of function as well. And we have comma separated parameter identifiers, which are preceded by the keyword class or typename, we have seen this before, which are enclosed between angle brackets and then follows the definition of the class. And as I said, most often, it can be used in several contexts, but most often, these are, this find use in the container classes, particularly the ones that we find in the library.

(Refer Slide Time: 08:59)



Stack as a Class Template: Stack.h

```
template<class T>
class Stack {
    T data_[100];
    int top_;
public:
    Stack() :top_(-1) { }
    ~Stack() { }

    void push(const T& item) { data_[++top_] = item; }

    void pop() { --top_; }

    const T& top() const { return data_[top_]; }

    bool empty() const { return top_ == -1; }
};
```

- Stack of type variable T
- The traits of type variable T include
  copy assignment operator (T operator=(const T&))
- We do not call our template class as stack because std namespace has a class stack

Programming in Modern C++ Partha Pratim Das M39.13

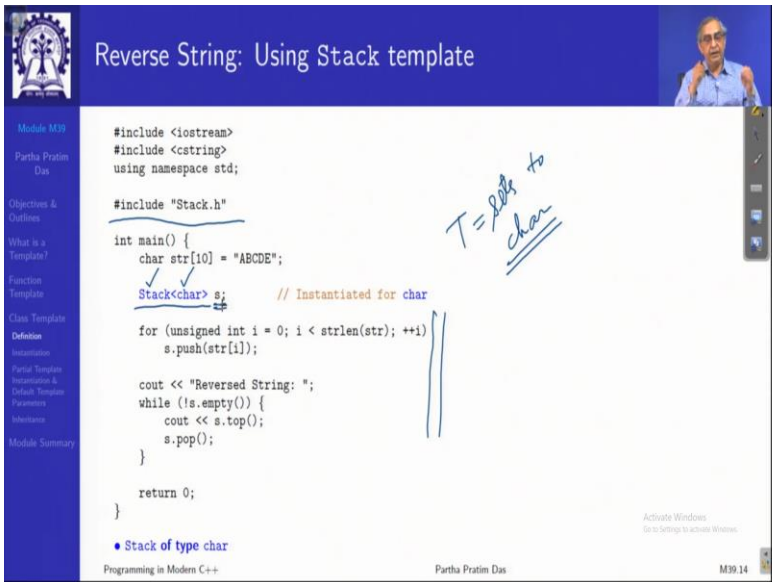
So, let us write stack as a template. So, there is no difference. All that is done is, as I said, we have changed these three positions to stack, to type T, the type variable T. Everything else in this class remains same, have added a first line, which say template within angular bracket class T. This tells that in the following definition of the class, T is a type variable. And whenever I want, I can replace a type for T and get the stack of a specific type.

Now, naturally, a question is if T is a type, then what are the properties that T should have? These are loosely called traits, traits are a big matter of study, but we will just take it in the way that, well, certain properties, this type must satisfy to be able to use it as an instantiation type for the class template. For example, we need to be able to do assignment. I should be able to assign that, I am doing it here, in push. I am taking the item and putting it in the container where the container here is an array, the actual physical container on the line.

So, unless T has a copy assignment operator, I certainly cannot do this. Now, naturally I have not elaborated, but it is quite obvious that T must have a constructor and a destructor to be able to, for me to be able to construct it if needed, and obviously, destruct it when the stack is being destroyed all over automatically. So, the array elements will have to be destroyed the structure is required, but I will point to the special functions or operators that are needed for T.

For example, just as, or rather as a counter example, I would say that, well, I do need to know if T can be compared or not. Whether given two elements of T, I can say whether they are equal or unequal or less or greater, if I cannot say that, that is if T does not have any overloaded comparison operator, I do not have a problem of using it as a stack. In some other context I may have. So, the properties that I need of this type variability defines the traits of it and we must be aware of that trait to be able to actually instantiate a class template.

(Refer Slide Time: 11:46)



Reverse String: Using Stack template

Module M39

Partha Pratim Das

Objectives & Outlines

What is a Template?

Function Template

Class Template

Definition

Instantiation

Partial Template Instantiation & Default Template Parameters

Inheritance

Module Summary

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

#include "Stack.h"

int main() {
    char str[10] = "ABCDE";

    Stack<char> s;        // Instantiated for char

    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);

    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }

    return 0;
}
```

T = sets to char

- Stack of type char

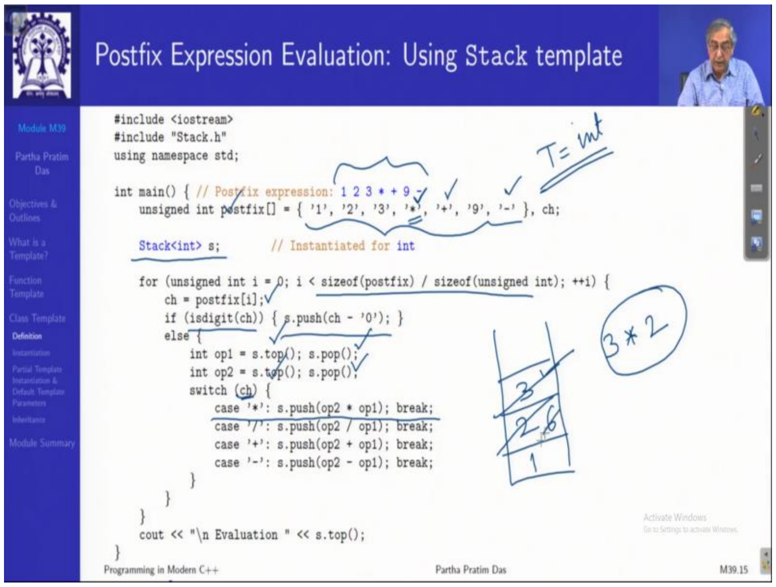
Programming in Modern C++ Partha Pratim Das M39.14

So, how do we use it? Once we have got that template defined, as we have done in this header file, we use it in a manner which is very similar to the way we did in case of case of function, we had function name followed by the type being instantiated with in corner brackets. Here, we have the class name followed by the type. So, this defines it stack of character. So, what the compiler does is it, it sets in that header file code, it sets to char. Generates a code, which replaces T by char, and then compiles it.

So, since it generates a code, it just had that template, and it is generating the code, this is called generative programming. Since it, I am, I am programming at a different level than the normal programming, where I will know all the types, this is called programming about programming. So, this is called meta programming. So, that is the, that is a basic reason for the names.

And then I can use that Stack as defined as s here. And there is nothing specifically different here. The same code that would have happened if I had explicitly created a, written a stack of character, I am writing the same code as an application. So, this makes it really, really convenient to make use of, reuse of the stack template code.

(Refer Slide Time: 13:20)



Postfix Expression Evaluation: Using Stack template

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() { // Postfix expression: 1 2 3 * + 9 -
    unsigned int postfix[] = { '1', '2', '3', '*', '+', '9', '-' }, ch;

    Stack<int> s;       // Instantiated for int

    for (unsigned int i = 0; i < sizeof(postfix) / sizeof(unsigned int); ++i) {
        ch = postfix[i];
        if (isdigit(ch)) { s.push(ch - '0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\n Evaluation " << s.top();
}
```

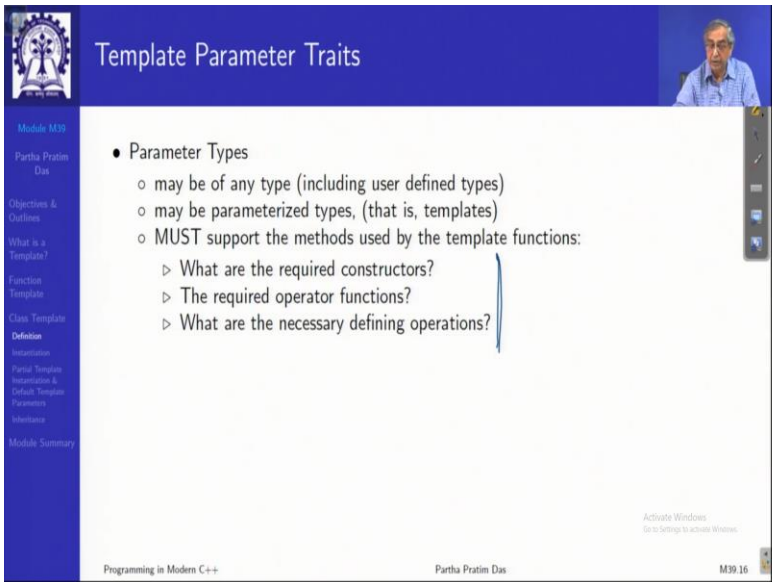
In a different context, I am using instantiating the stack, the same stack template code using int. So, T now is replaced by int, in the whole of the Stack definition. And what you have is a stack of int where every element is expected to be an integer. Then given a postfix expression here, which here I assume that is taken as an array of characters for different constants and different operators. I can go ahead evaluate it using the stack.

All that I need to do is to check that go over this entire array, this entire area to see that I still have elements, and if I get at the element as I get here, then I check whether it is a digit. If it is a digit, I find out its numerical value ch - the ASCII code of 0 which, so I am assuming that the constants here are single digit constants, just for simplicity, and push that in the stack.

And when it is not a digit, it must be an operator like this or like this. So, as I go ahead, I will have things like, I will have 1 going in first, 2 going in next, 3 going in next, and then I will have this operator. And when I get that, I will take the top of the stack, which is 3, and take it out. Because I know that this multiplication has to happen with the two top most literals that I have in the stack. Then I again look at the top, I get, I have got 3. I take it out, I have got 2.

And then I check what operation I had, I do that in this switch, and I push the result of this multiplication, because it is a multiplication symbol. So, now, my stack becomes 1, 6. And again, I have a +. So, as I have a +, I will go ahead and do that. I will take this out, take this out, add them as 6 + 1, using this option and so on. So, I can very conveniently evaluate a postfix expression using the stack where I did not have to write the stack code, I could just reuse the templatized stack through instantiation.

(Refer Slide Time: 16:05)



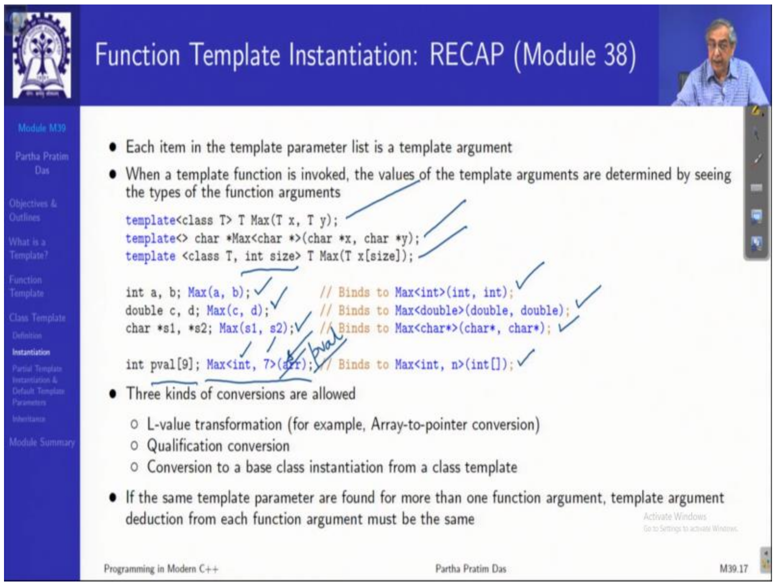
Template Parameter Traits

- Parameter Types
  - may be of any type (including user defined types)
  - may be parameterized types, (that is, templates)
  - MUST support the methods used by the template functions:
    - What are the required constructors?
    - The required operator functions?
    - What are the necessary defining operations?

So, I already talked about the parameter trace, traits, parameter types, maybe of any type including user-defined types, it could be a complex type of stack or a person type of stack and so on. It, they themselves maybe parameterized types that is there themselves, T itself could be a template that it could also be a parameterized type that is possible. And it must support the methods used by the template functions.

For example, does it require a constructor? Does, what are the operator functions required? Like stack requires a copy assignment. So, that must be supported by the type that I am trying to instantiate with. So, these are necessary traits that the, that my instantiating type must support, and those traits are important to always keep in mind.

(Refer Slide Time: 17:03)



Function Template Instantiation: RECAP (Module 38)

Module M39

Partha Pratim Das

Objectives & Outlines

What is a Template?

Function Template

Class Template

Definition

Instantiation

Partial Template Instantiation & Default Template Parameters

Inheritance

Module Summary

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<> char *Max<char *>(char *x, char *y);
template <class T, int size> T Max(T x[size]);

int a, b; Max(a, b);          // Binds to Max<int>(int, int);
double c, d; Max(c, d);       // Binds to Max<double>(double, double);
char *s1, *s2; Max(s1, s2);   // Binds to Max<char*>(char*, char*);

int pval[9]; Max<int, 7>(arr); // Binds to Max<int, n>(int[]);
```

pval

- Three kinds of conversions are allowed
  - L-value transformation (for example, Array-to-pointer conversion)
  - Qualification conversion
  - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

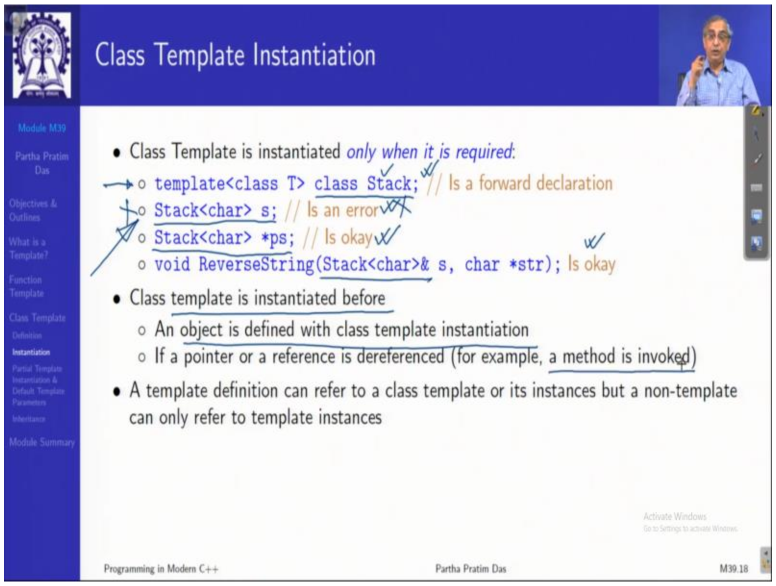
Programming in Modern C++ Partha Pratim Das M39.17

So, going, looking back again, how did we instantiate in function, in function template? We used, this is the generic template, can compute max of any two values of type key. Then we have specialized it in the instantiation, because for char*, which has taken to be a cstring, the comparison has to be done differently.

So, we saw that we can specialize it and have a different definition. If the function is passing is be, trying to call by two char* parameter, a different definition will be used. And then we have one which takes a takes an array and the size of the array, which is a special type of template parameter, which is not a type, but an integral value, which you can also specify for a template. You have seen all this.

So, with that Max(a, b) for integer, we will generate this function, Max(c, d) for double will generate an overloaded max function. Max(s1, s2) for char* will use the specialized, the template function that I have to defined. And if I call it with the type int and value 7 on an array arr, or maybe we can use pval here, if we, if we are referring to this, then it will bind to this special function, which also is overloading Max. So, this is the basic idea that we have seen. And in terms of class template, this idea will only get extended.

(Refer Slide Time: 18:57)


Class Template Instantiation

- Class Template is instantiated *only when it is required*:
  - `template<class T> class Stack;` // Is a forward declaration
  - `Stack<char> s;` // Is an error
  - `Stack<char> *ps;` // Is okay
  - `void ReverseString(Stack<char>& s, char *str);` Is okay
- Class template is instantiated before
  - An object is defined with class template instantiation
  - If a pointer or a reference is dereferenced (for example, a method is invoked)
- A template definition can refer to a class template or its instances but a non-template can only refer to template instances

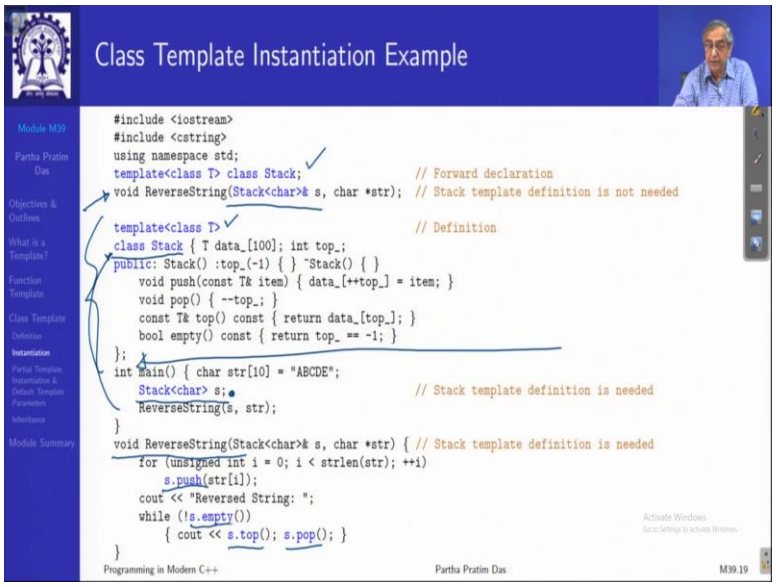
So, what I can do is template instantiation is done only when it is required. But I can proceed with just the knowledge that this is templatized class. But without actually requiring its definition. Because as long as I do not instantiate it. So, here is one way to write the template as a forward declaration, I say, template class T. So, T is the type variable, and class Stack, so it says that class Stack is a class named Stack, which has T as a type variable. Now, with that, if I try to write this, I will have an error. Why would I have an error? Because I am trying to instantiate, but I have not yet told what the class is. I have not yet defined the class.

So, this will be an error. This is not permitted, but if I want to define a pointer to it, it is quite okay, because to set a pointer, I do not need to actually know the internals of the class, all that I am saying that the pointer will point to a stack of type character. So, that is the, that is the difference? So, with the forward declaration, I can, it is okay to write that with the forward declaration, I can create a reference, it is okay to do that.

But with the forward declaration, I cannot create an instantiation. And these forward declarations, such declarations are known as incomplete types, because it is telling you, part of what the type is, like it is a Stack, and has a, has a type parameter T, but it does not tell you the details in terms of the data members and methods.

So, the class template must be instantiated before, and object is defined with the class template instantiation, which is what was getting violated at this point. If a pointer or reference is dereferenced, then also I will need that. Just to define the pointer, I do not need to know the details, but if I want to dereference, then I will suddenly need to know the objects. So, I need to know the definition of that particular template, right? So, this is the basic idea of class template instantiation and we will use that in the example now.

(Refer Slide Time: 21:19)



Class Template Instantiation Example

```
#include <iostream>
#include <cstring>
using namespace std;
template<class T> class Stack;                  // Forward declaration
void ReverseString(Stack<char>& s, char *str);  // Stack template definition is not needed

template<class T>                               // Definition
class Stack { T data_[100]; int top_;
public: Stack() :top_(-1) { } ~Stack() { }
    void push(const T& item) { data_[++top_] = item; }
    void pop() { --top_; }
    const T& top() const { return data_[top_]; }
    bool empty() const { return top_ == -1; }
};
int main() { char str[10] = "ABCDE";
    Stack<char> s;                              // Stack template definition is needed
    ReverseString(s, str);
}
void ReverseString(Stack<char>& s, char *str) { // Stack template definition is needed
    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);
    cout << "Reversed String: ";
    while (!s.empty())
        { cout << s.top(); s.pop(); }
}
```

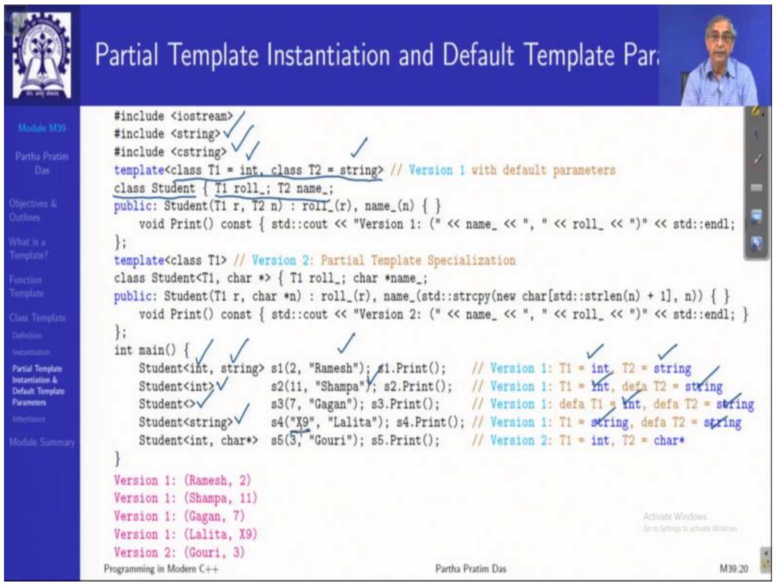
Programming in Modern C++ Partha Pratim Das M39.19

So, there is a forward declaration. And with that, I am trying to use a stack of characters to reverse a string. So, I want to make a forward declaration, I want to write the signature of the reverse string function and to be able to do that, I need this Stack type to be known. And I have passing the stack by reference, note that I have yet defined the stack class and its perfectly, because I am just defining a reference. Then again, I say, class T, template class T, class Stack, and I provide the definition.

Now, once I have provided the definition from this point onwards, from this point onward, I can refer to it for defining pointer and reference, but additionally, I can actually create objects for it. I can create the instantiation, so I can create the instantiation here now and it is going to require this definition, which I have already provided. Had this definition not been given, I would not have been able to do this.

Similarly, I can implement ReverseString function, which is invoking various stack methods. And I am able to do this, because I have already defined it. With just a declaration, I would not be able to do that. So, that is the, that is the basic process of class template instantiation. And you have seen a simple, but complete example for that.

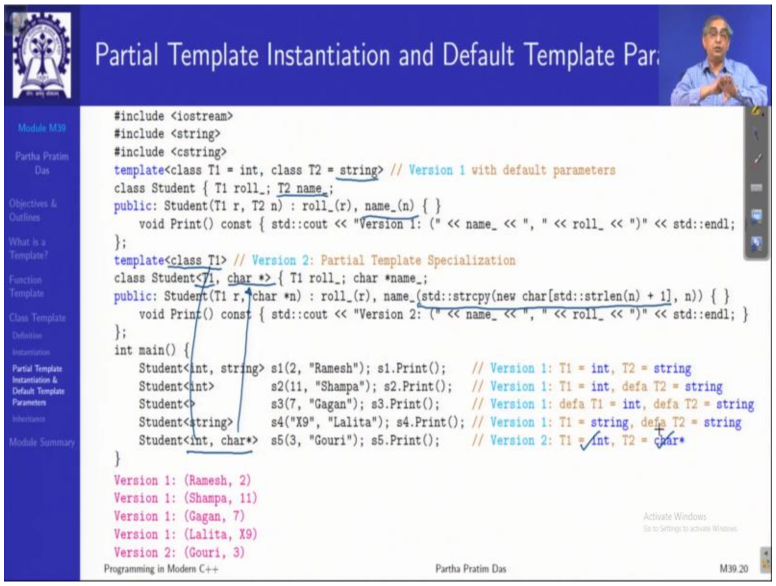
(Refer Slide Time: 22:55)



Partial Template Instantiation and Default Template Par

```
#include <iostream>
#include <string>
#include <cstring>
template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;
public: Student(T1 r, T2 n) : roll_(r), name_(n) { }
    void Print() const { std::cout << "Version 1: (" << name_ << ", " << roll_ << ")" << std::endl;
};
template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char *> { T1 roll_; char *name_;
public: Student(T1 r, char *n) : roll_(r), name_(std::strcpy(new char[std::strlen(n) + 1], n)) { }
    void Print() const { std::cout << "Version 2: (" << name_ << ", " << roll_ << ")" << std::endl; }
};
int main() {
    Student<int, string> s1(2, "Ramesh"); s1.Print();    // Version 1: T1 = int, T2 = string
    Student<int>         s2(11, "Shampa"); s2.Print();   // Version 1: T1 = int, defa T2 = string
    Student<>            s3(7, "Gagan"); s3.Print();     // Version 1: defa T1 = int, defa T2 = string
    Student<string>      s4("X9", "Lalita"); s4.Print(); // Version 1: T1 = string, defa T2 = string
    Student<int, char*>  s5(3, "Gouri"); s5.Print();     // Version 2: T1 = int, T2 = char*
}
Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```

Programming in Modern C++ Partha Pratim Das M39.20



Partial Template Instantiation and Default Template Par

```
#include <iostream>
#include <string>
#include <cstring>
template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;
public: Student(T1 r, T2 n) : roll_(r), name_(n) { }
    void Print() const { std::cout << "Version 1: (" << name_ << ", " << roll_ << ")" << std::endl;
};
template<class T1> // Version 2: Partial Template Specialization
class Student<T1, char *> { T1 roll_; char *name_;
public: Student(T1 r, char *n) : roll_(r), name_(std::strcpy(new char[std::strlen(n) + 1], n)) { }
    void Print() const { std::cout << "Version 2: (" << name_ << ", " << roll_ << ")" << std::endl; }
};
int main() {
    Student<int, string> s1(2, "Ramesh"); s1.Print();    // Version 1: T1 = int, T2 = string
    Student<int>         s2(11, "Shampa"); s2.Print();   // Version 1: T1 = int, defa T2 = string
    Student<>            s3(7, "Gagan"); s3.Print();     // Version 1: defa T1 = int, defa T2 = string
    Student<string>      s4("X9", "Lalita"); s4.Print(); // Version 1: T1 = string, defa T2 = string
    Student<int, char*>  s5(3, "Gouri"); s5.Print();     // Version 2: T1 = int, T2 = char*
}
Version 1: (Ramesh, 2)
Version 1: (Shampa, 11)
Version 1: (Gagan, 7)
Version 1: (Lalita, X9)
Version 2: (Gouri, 3)
```

Programming in Modern C++ Partha Pratim Das M39.20

Now, you can, like, you can have default parameters for functions where you can give default values, you can have default parameters for templates also and those known as default types. So, I am trying to define here a class Student and the student is, just for simplicity, is expected to have two data members. One is the role number, which is of type T1 and the other is a name which is a of type T2. What I am going to illustrate is certainly a role number can be considered to be an integer or a string, name could be considered to be a char*, that is is C style string or the string type that is given by the C++ standard library.

So, I have included string as well as C string here. Now, I say that instead of just saying Class T1, class T2, I say class T1 defaulted with int, class T1 defaulted with string. So, if I do not specify anything, or if I specify just int, the other parameter, the parameters not specified will be taken by default. So, if I specify it by int and string, this is int, this is string, and I can write this. I can instantiate this object.

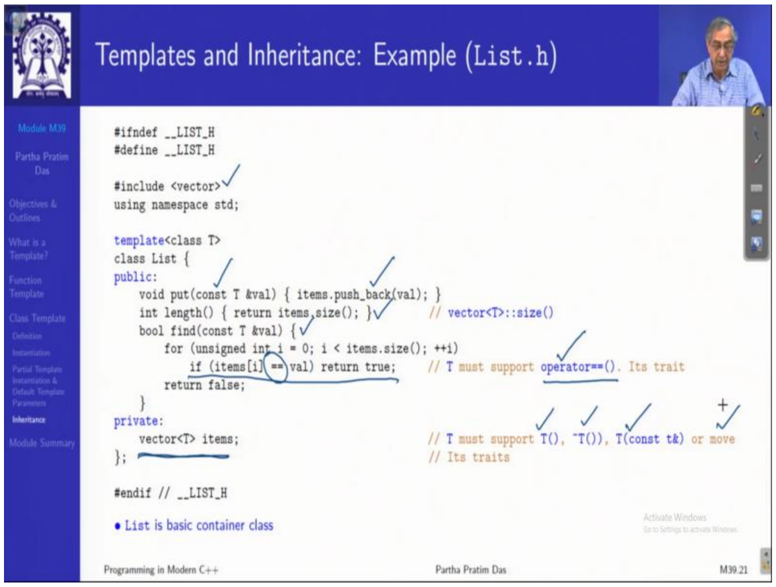
If I just say int, then the first parameter is specified as int. The second parameter is taken as default, which is string. So, this is specified as int, this is taken as string, though. I have not written it as string, and I can construct s2. If I do not specify anything, then both parameters are taken by default. So, I have default of T1 as int and T2 as string. If I instantiate here a string, then T1 is taken as string, T2 is default string. So, in this case, I am assuming that the role number is string.

So, this is all that I can easily do. Now, with this, I can also do a partial template specialization. I did template specialization for function when I introduced Max for char*. So, similar thing we are going to do here. So, what I am saying is, instead of two, I will give you one parameter, which is the first one, whereas the second parameter has already been specialized or already been specified to be char*. Because as you can understand that when I have the parameter here at string, then my, at my construction time, I need to do a copy construction of string from the given string to the data member name.

Whereas if it is a char*, then the, if I, if I follow the first definition, I will have an error, because what it will copy is not the string, but it will copy the pointer. As we have seen this. So, it will be a shallow copy. It will not be able to do the deep copy. So, I have to have a separate specialization for this template to deal with the second type parameter being char*. And if that is done, then I need to do all this allocation for that string, and then in that allocated area, I need to copy the value that is what I am doing here is necessarily a deep copy, which is specifically required for char* only.

So, if I instantiate now with Student<int, char*>, then int is taken to T1 and char* let me use the second specialized definition of this template I get. And therefore, if I had not provided this, then doing Student<int, char*> would have been an error, because you would have, it would have just copied the pointer, not the actual value. And then in during destruction time, it would have been difficult. So, this is, I mean, it can be generalized in any ways further, but this is what is known as partial template specialization.

(Refer Slide Time: 27:35)



Templates and Inheritance: Example (List.h)

```
#ifndef __LIST_H
#define __LIST_H

#include <vector>
using namespace std;

template<class T>
class List {
public:
    void put(const T &val) { items.push_back(val); }
    int length() { return items.size(); }          // vector<T>::size()
    bool find(const T &val) {
        for (unsigned int i = 0; i < items.size(); ++i)
            if (items[i] == val) return true;      // T must support operator==(). Its trait
        return false;
    }
private:
    vector<T> items;                               // T must support T(), ~T()), T(const t&) or move
};                                                 // Its traits

#endif // __LIST_H
```

- List is basic container class

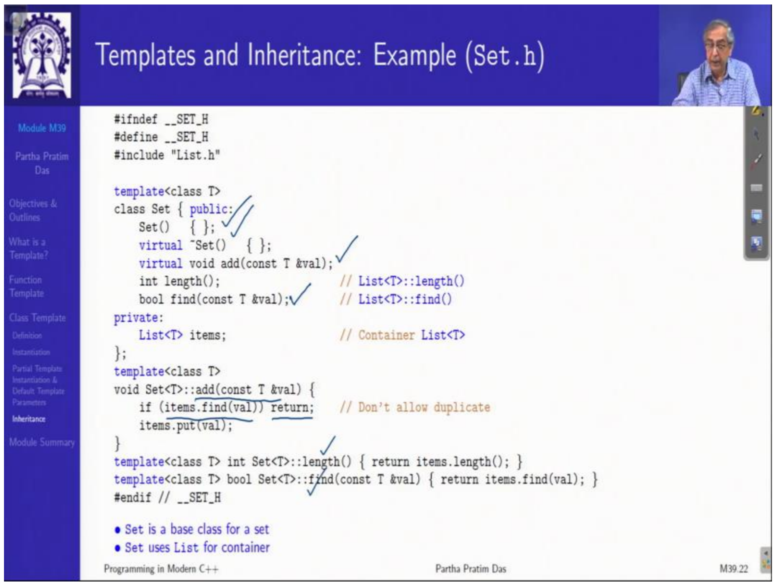
Now, finally, before we close, let me take a look at what happens if you do inheritance. Because since you have class, you can do inheritance with the template classes. So, what I provide, I could have used the library definitions also from the standard library, but I am just using vector as a basic container, but otherwise, I am trying to define my data structure. So, I say, I have a list where I actually keep the elements. My objective is to define classes for maintaining set and specifically bounded set.

So, in the list, what do I need? I need a way to add a mechanism. So, which is put, which in terms of vector, because your container is a vector. So, it is a push_back, so that will add an element. So, as if a list is being implemented in terms of an array, something peculiar, but naturally your vector will allow you to do that as long as you do not want to, you know, manipulate in the middle of the list.

Then size is simply the size of the vector, find we can go along, this is just doing a linear scan and find out if that element exists, if it exists, it will give you true, others, it will give you false. And you can see by the side that when I do this, I require the equality operator to be available, to compare two values of t.

Otherwise, I will not be able to do this comparison. So, the trait must include this. Similarly, for creating a vector and vector can expand, so it might need to copy where it expands. So, I need the constructor, destructor, copy constructor or move, we have not seen move yet. It will come in C++11 for the list. So, this is my basic container. And with that, I will now define what is a set?

(Refer Slide Time: 29:34)



Templates and Inheritance: Example (Set.h)

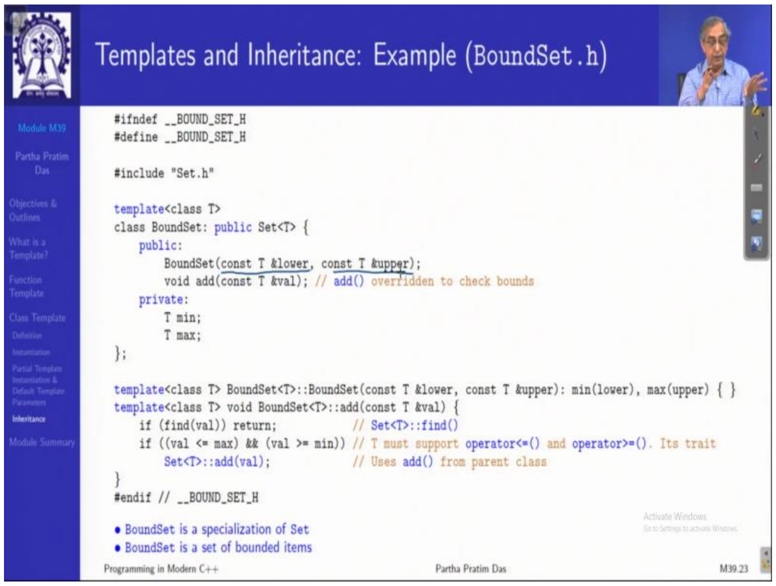
```
#ifndef __SET_H
#define __SET_H
#include "List.h"

template<class T>
class Set { public:
    Set()   { };
    virtual ~Set()   { };
    virtual void add(const T &val);
    int length();                       // List<T>::length()
    bool find(const T &val);            // List<T>::find()
private:
    List<T> items;                      // Container List<T>
};
template<class T>
void Set<T>::add(const T &val) {
    if (items.find(val)) return;        // Don't allow duplicate
    items.put(val);
}
template<class T> int Set<T>::length() { return items.length(); }
template<class T> bool Set<T>::find(const T &val) { return items.find(val); }
#endif // __SET_H
```

- Set is a base class for a set
- Set uses List for container

So, this is my Set, which has the virtual construct, the virtual destructive, because I want to make it a base class. I have an add method to add an element to the set. And what is the peculiarity of the set? The peculiarity of the set is it cannot have duplicate. So, I need to use find function to check if the element being added is already there.

So, when I try to do add, I first do a find. If it is there, I do not add it, because the duplicates are not allowed. Otherwise, I add it by doing the put of the list. This is a simple thing that I do. I can find out the length using the length of the list, which is the length of the vector. And certainly, I can do the find using the find in the list. So, this will give me a class to maintain any set using list as a container.

(Refer Slide Time: 30:43)



Templates and Inheritance: Example (BoundSet.h)

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet: public Set<T> {
    public:
        BoundSet(const T &lower, const T &upper);
        void add(const T &val); // add() overridden to check bounds
    private:
        T min;
        T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper): min(lower), max(upper) { }
template<class T> void BoundSet<T>::add(const T &val) {
    if (find(val)) return;          // Set<T>::find()
    if ((val <= max) && (val >= min)) // T must support operator<=() and operator>=(). Its trait
        Set<T>::add(val);           // Uses add() from parent class
}
#endif // __BOUND_SET_H
```

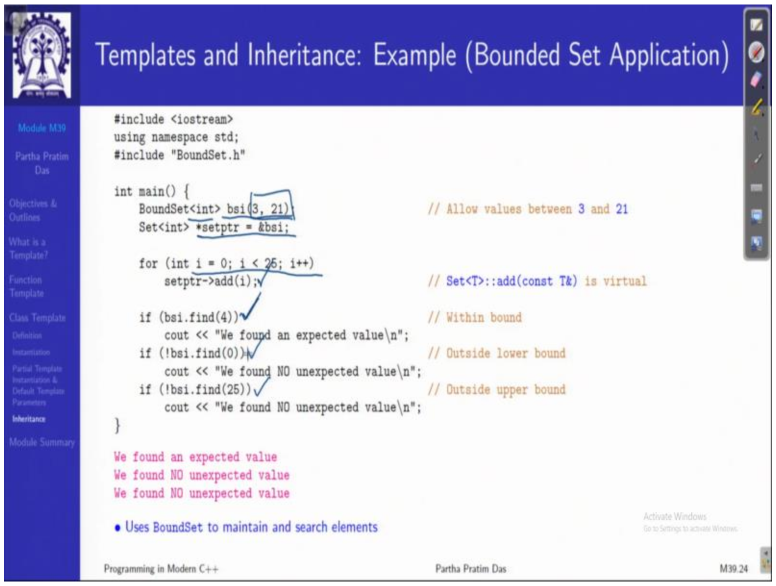
- BoundSet is a specialization of Set
- BoundSet is a set of bounded items

Programming in Modern C++ Partha Pratim Das M39.23

Now, I want to have a specialized set. A set where the elements are bounded within a range. So, I call it the BoundedSet. So, in the BoundedSet, I have two bounds, lower and upper. So, every element has to be in between that. And certainly, I will not allow any element, which is outside of it. So, if I want to add an element, I have to check whether the bounds, the bounds that are given here are satisfied. That is a value must be must be, must be less than equal to the max and greater than equal to the min only than the add is enough. So, I need to override the add for the Set class.

So, this is a polymorphic inheritance, as you have seen, because add was a virtual function in Set. So, I override that, and I check if it is within the bound, if it is within the bound, then I make use of the add of the parent class to actually do the adding, which finds for duplicate, and if the duplicate is not there, we will put it in the list and so on. So, in this way, using set specializing from Set, I can have a BoundedSet, and all of these list Set and BoundedSet are parameterized by a type parameter T.

(Refer Slide Time: 32:02)



Templates and Inheritance: Example (Bounded Set Application)

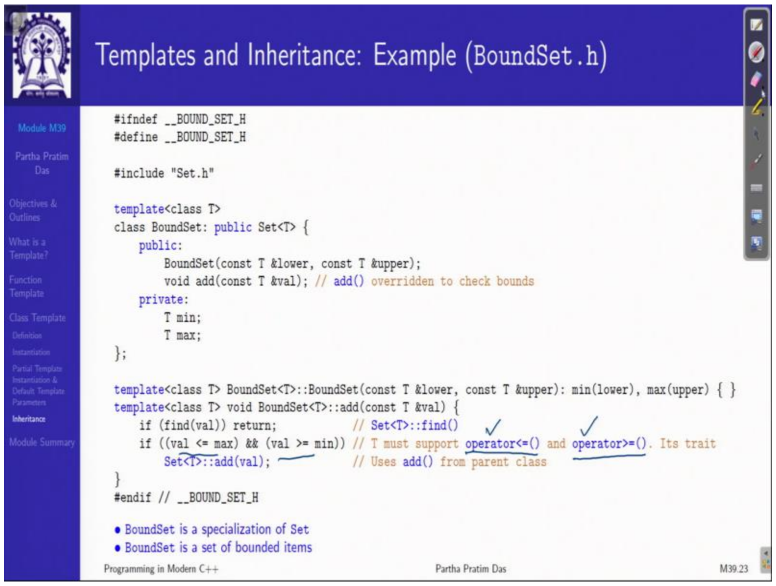
```
#include <iostream>
using namespace std;
#include "BoundSet.h"

int main() {
    BoundSet<int> bsi(3, 21);                        // Allow values between 3 and 21
    Set<int> *setptr = &bsi;

    for (int i = 0; i < 25; i++)
        setptr->add(i);                              // Set<T>::add(const T&) is virtual

    if (bsi.find(4))                                 // Within bound
        cout << "We found an expected value\n";
    if (!bsi.find(0))                                // Outside lower bound
        cout << "We found NO unexpected value\n";
    if (!bsi.find(25))                               // Outside upper bound
        cout << "We found NO unexpected value\n";
}
```

```
We found an expected value
We found NO unexpected value
We found NO unexpected value
```

- Uses BoundSet to maintain and search elements

So, using that, I can have a Set, BoundedSet of integers defined i, create a pointer to set and let it point here. So, it is, it is doing an upcast automatically of the pointer type. Then it adds since the actual elements are the actual BoundedSet is a specialization. So, this ad will call the add of the BoundedSet, and then I can do find directly to see if it is, if an element is within the bound, and then it must be found. If it is added, all values from 0 to 25 have been attempted to be added with the bound of 3 and 21. So, naturally 0, 1, 2 could not have been added, 22, 23, 24, 25 could not have been added. So, this will succeed, this will fail on 0, this will fail on 25, and that is what you get. So, this is a simple illustration for what you can inherit and create a specialized definition from.

(Refer Slide Time: 33:14)



Templates and Inheritance: Example (BoundSet.h)

Module M39

Partha Pratim Das

Objectives & Outlines

What is a Template?

Function Template

Class Template

Definition

Instantiation

Partial Template Instantiation & Default Template Parameters

Inheritance

Module Summary

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet: public Set<T> {
    public:
        BoundSet(const T &lower, const T &upper);
        void add(const T &val); // add() overridden to check bounds
    private:
        T min;
        T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper): min(lower), max(upper) { }
template<class T> void BoundSet<T>::add(const T &val) {
    if (find(val)) return;             // Set<T>::find()
    if ((val <= max) && (val >= min)) // T must support operator<=() and operator>=(). Its trait
        Set<T>::add(val);              // Uses add() from parent class
}
#endif // __BOUND_SET_H
```

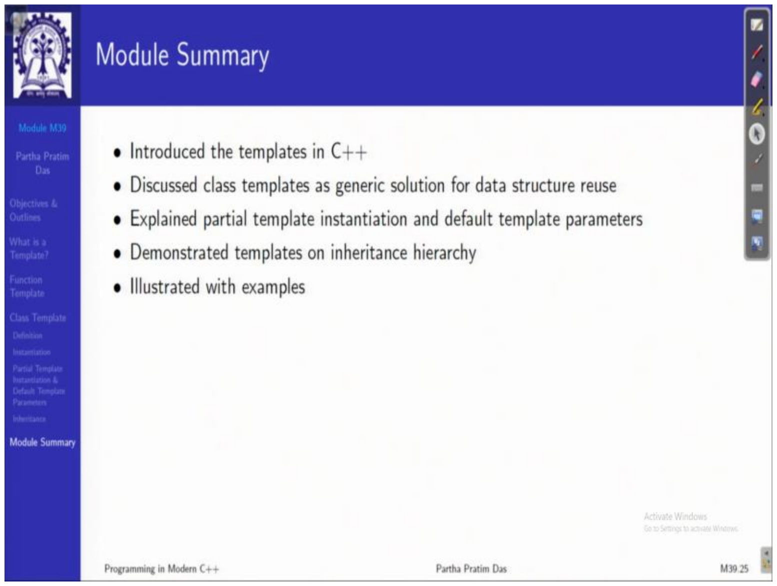
- BoundSet is a specialization of Set
- BoundSet is a set of bounded items

Programming in Modern C++ Partha Pratim Das M39.23

I am just going back one slide to highlight that here, for example, you are the trait of this type now needs the Set type did not need, because Set type is not checking comparison of values, but the moment you make BoundedSet, you are checking comparison of values, less than equal to greater than equal to.

So, these operators these Boolean operators must be supported by that type T. So, I can have it on int, because in has these operators, I can have it on double, because double has these operators. But I may not be able to have it on Student, because I have not defined any such operator, right? So, this is, this is your basic class template, and with a complete example.

(Refer Slide Time: 34:05)



Module Summary

- Introduced the templates in C++
- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

I think to summarize, we have talked about templates, the generic programming in C++, discussed the class template as a generic solution for data structure reuse and demonstrated them also on the hierarchy. We will have a lot more examples in our assignments and some in the tutorial, this is a very, very strong area, and we go into the modern part of C++ discussing, C++11, we will make use of this concept of template very widely and very rampantly, both for the function as well for classes. Thank you for your attention and we will meet in the next module.