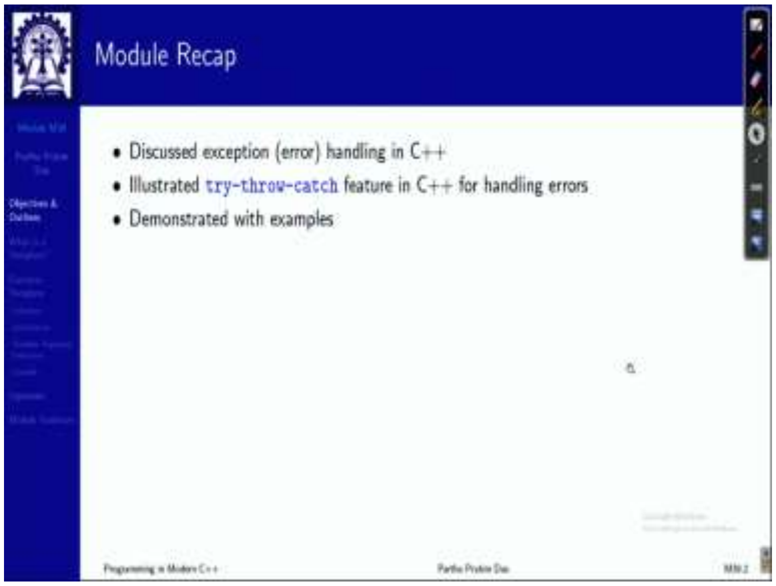
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 38
Template (Function Template): Part 1

Welcome to Programming in Modern C++. We are in Week 8 and we are now going to discuss Module 38.

(Refer Slide Time: 00:36)

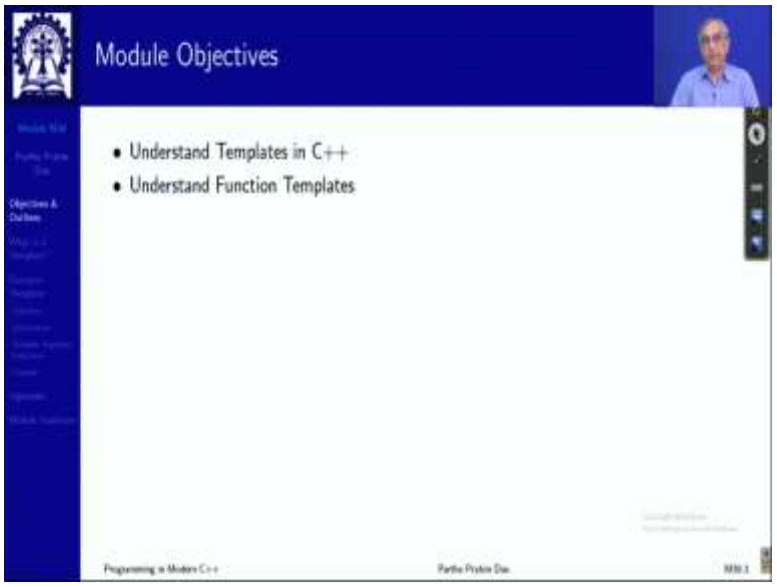


Module Recap

- Discussed exception (error) handling in C++
- Illustrated `try-throw-catch` feature in C++ for handling errors
- Demonstrated with examples

In the last module we concluded discussions on exception handling in C++, the use of try-throw-catch, which is a very, very important feature and you must practice it a lot.

(Refer Slide Time: 00:47)



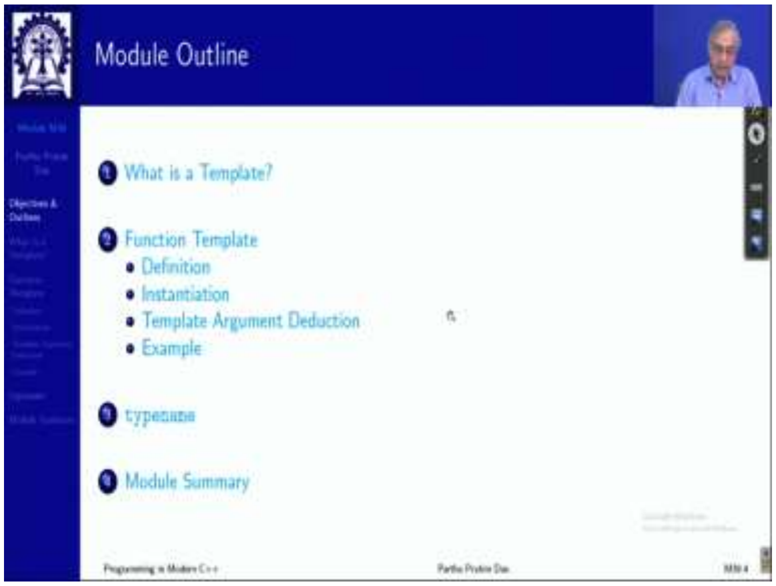
Module Objectives

- Understand Templates in C++
- Understand Function Templates

Programming in Modern C++ Partha Pratim Das

In this module, I start a two-part discussion on generic programming or meta programming feature of C++. We have so far been dwelling with procedural features and with object oriented features. This is a new aspect of C++ generate, generic programming or meta programming that is where you can write programs in C++ which generates programs. So, that feature in C++ is called template. This discussion will be in two parts in the present module. We will talk about function template, and in the next module, we will talk about class template.

(Refer Slide Time: 01:29)



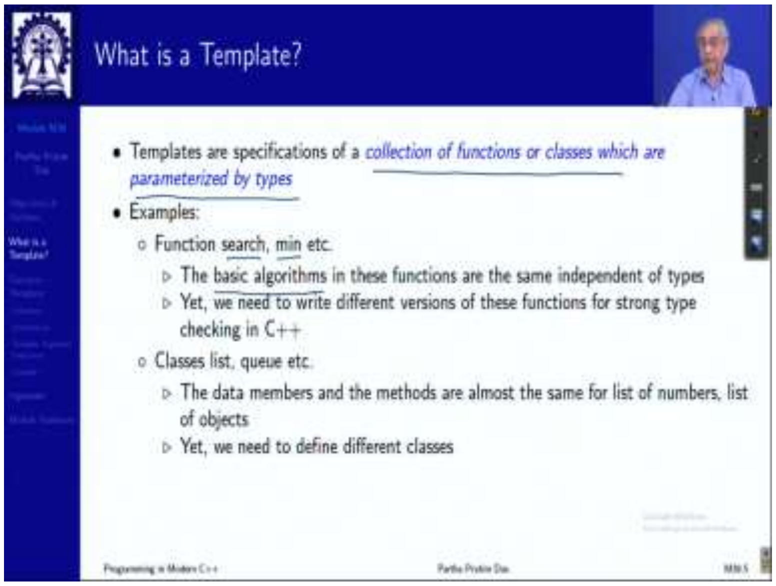
Module Outline

1. What is a Template?
2. Function Template
   - Definition
   - Instantiation
   - Template Argument Deduction
   - Example
3. typename
4. Module Summary

Programming in Modern C++ Partha Pratim Das M38.4

So, this is a outline available on the left panel as usual.

(Refer Slide Time: 01:33)



What is a Template?

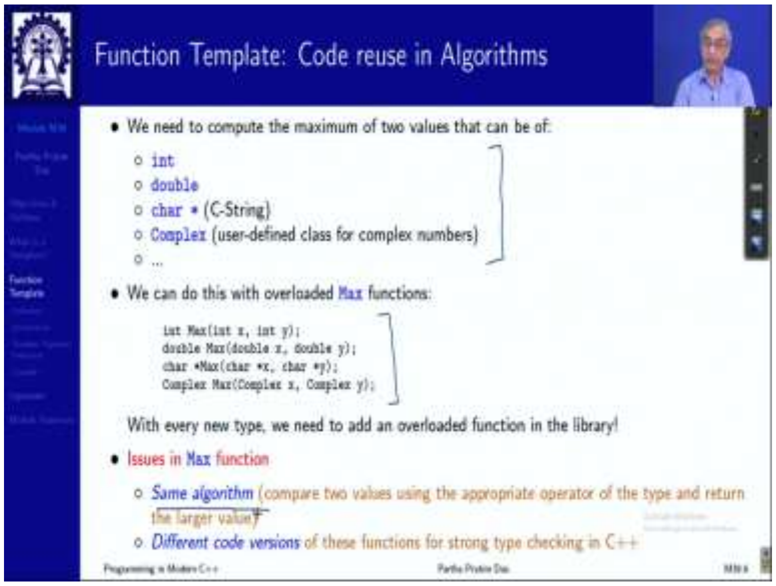
- Templates are specifications of a *collection of functions or classes which are parameterized by types*
- Examples:
  - Function search, min etc.
    - ▷ The basic algorithms in these functions are the same independent of types
    - ▷ Yet, we need to write different versions of these functions for 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 M38.5

So, what is that template? Let us first just talk about that. Templates are a specification of collection of functions which are parameterized by types. So, far functions were parameterized with the, for their arguments, but not by types. Why do we need that? Because there are several functions, several logic which are the, where the algorithm is generic, like search, like finding minimum, like, these are generic in nature of their code. But we need to write different versions of the code, because there is a strong type checking in C++.

For every day type of data for which I want to do a search, I need to have a separate function. Similarly, lot of data structures like stack, list, queue, all of these, they have the same data members the same methods, because that is the behavior of the data structure. But they need to be written with every possible type of data element. So, stack of integers, stack of double, stack of strings, stack of complex, stack of MyClass, all of these will have to be written even though the basic logic of push, pop, top empty and all that will be the same. So, the question is, can we can we get rid of this redundancy and increase the reuse by writing the logic in a type independent manner.

(Refer Slide Time: 03:17)



Function Template: Code reuse in Algorithms

- 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:

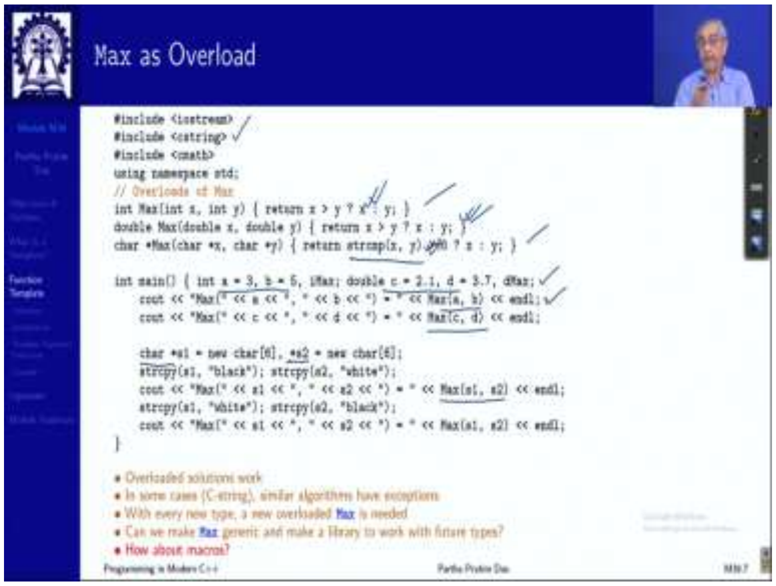
```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

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++

So, to illustrate, let us say we have these different types, and we want to find the maximum of two numbers. So, what will be the immediate C++ solution, we can write all these functions. These can, I mean unlike C, this can reside simultaneously because of the overloading feature. So, I can have a Max for int, Max for double, Max for cast raw string, Max for Complex and so on. But each Max function will have the same algorithm, but different code versions. So, that is a, that is a lot of overhead and lot of redundancy.

(Refer Slide Time: 04:00)



Max as Overload

```
#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;
// Overloads of Max
int Max(int x, int y) { return x > y ? x : y; }
double Max(double x, double y) { return x > y ? x : y; }
char *Max(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { int a = 3, b = 5, iMax; double c = 2.1, d = 3.7, dMax;
    cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl;
    cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl;

    char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl;
}
```

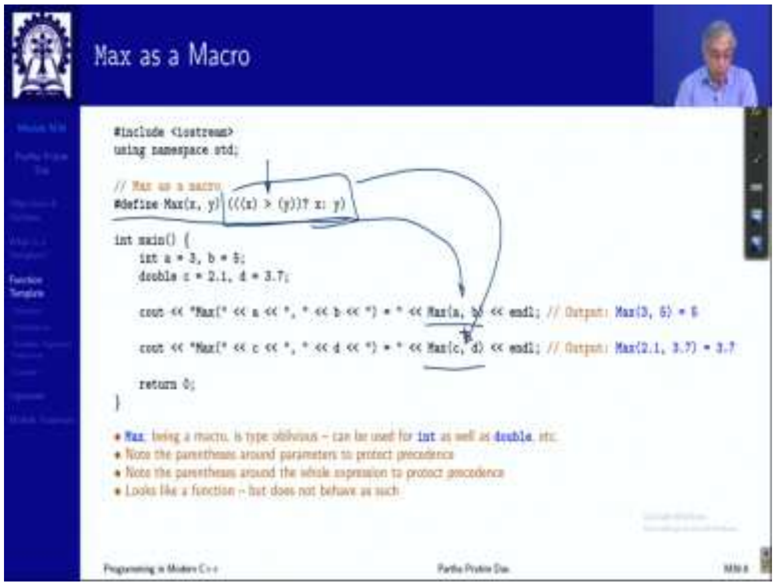
- Overloaded solutions work
- In some cases (C-string), similar algorithms have exceptions
- With every new type, a new overloaded Max is needed
- Can we make Max generic and make a library to work with future types?
- How about macros?

Programming in Modern C++ Partha Pratim Das M36.7

So, these are the three Max functions for integer, double and string. You can see that for string, you need to, since you get two pointers, it is not enough to compare the pointers, you have to do a strcmp, the string comparison from the cstring library and compared which string is less or smaller or is greater. Then we can with the set of values, we can call the Max with a, b depending on their integer type. So, this function will get called, I call Max c, d. They are of double type, so this function will get called, I can call s1, s2 char* types. So, this function will get called, everything will work fine.

But the problem is, all of these have to be provided separately. And if there is a new type for which I need to do a comparison of two values, I need to write another function, even though the logic would be the same. So, how can we write it in a generic type oblivious form? Naturally, one feature that comes to mind immediately is macros. We have seen macros, that was a purpose that C had macros.

(Refer Slide Time: 05:23)



Max as a Macro

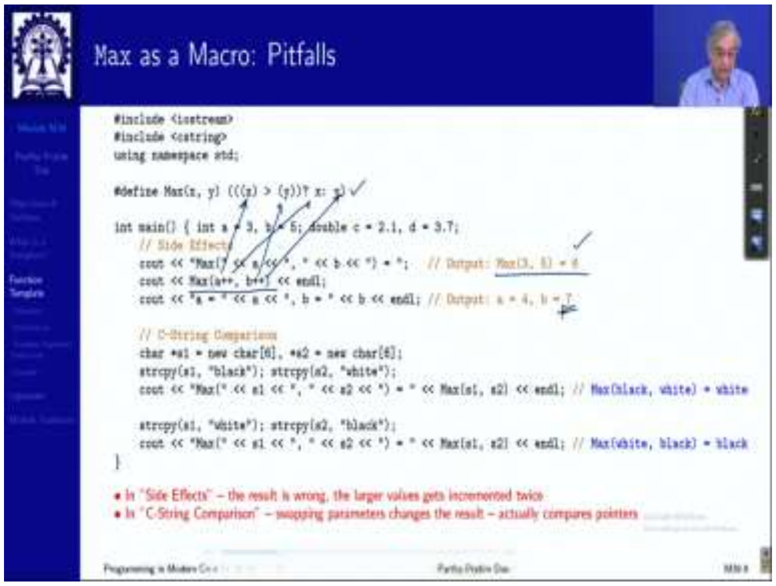
```
#include <iostream>
using namespace std;

// Max as a macro
#define Max(x, y) (((x) > (y))? x: y)

int main() {
    int a = 3, b = 5;
    double c = 2.1, d = 3.7;

    cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl; // Output: Max(3, 5) = 5

    cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl; // Output: Max(2.1, 3.7) = 3.7

    return 0;
}
```

- Max: being a macro, is type oblivious – can be used for int as well as double, etc.
- Note the parentheses around parameters to protect precedence
- Note the parentheses around the whole expression to protect precedence
- Looks like a function – but does not behave as such

Programming in Modern C++ Partha Pratim Das M06.6

So, what, if we can just quickly recap as to what will happen if I want to use macro in this context. So, I will write a macro like this, where this is a comparison. Now, and all these, a lot of parentheses are put to make sure that your precedences do not get mixed up. We have we have discussed this in the early modules when he discussed about shortcomings of C and how C++ is better C. But with this macro, I can do Max(a, b), I can do Max(c, d), different types, I do not need to specify anything, but just textually replaces this code, this entire, this code is textually replaced here, and things work. So, it looks like a function, but as such, it does not behave like a function, you know that.

(Refer Slide Time: 06:21)

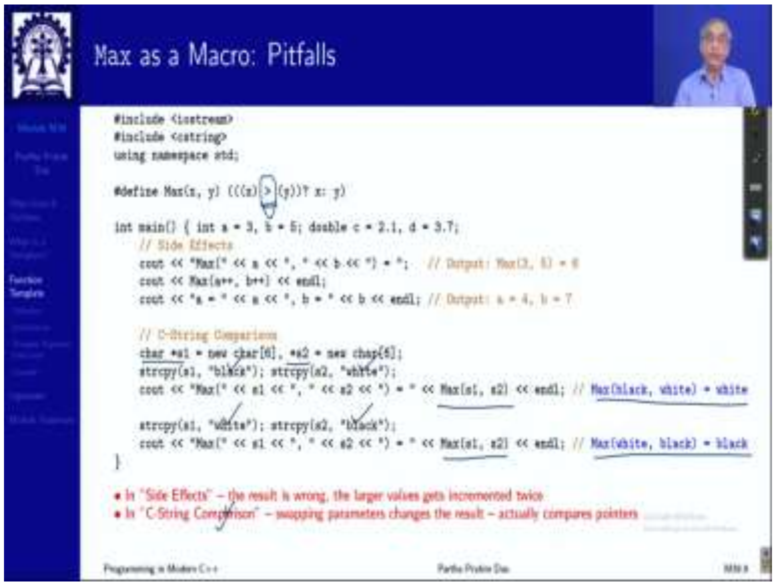


Max as a Macro: Pitfalls

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

#define Max(x, y) (((x) > (y))? x: y)

int main() { int a = 3, b = 5; double c = 2.1, d = 3.7;
    // Side Effects
    cout << "Max(" << a << ", " << b << ") = ";   // Output: Max(3, 5) = 6
    cout << Max(a++, b++) << endl;
    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 7

    // C-String Comparison
    char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(white, black) = black
}
```

- In "Side Effects" – the result is wrong, the larger values gets incremented twice
- In "C-String Comparison" – swapping parameters changes the result – actually compares pointers

Programming in Modern C++ Partha Pratim Das M08.8



Max as a Macro: Pitfalls

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

#define Max(x, y) (((x) > (y))? x: y)

int main() { int a = 3, b = 5; double c = 2.1, d = 3.7;
    // Side Effects
    cout << "Max(" << a << ", " << b << ") = ";   // Output: Max(3, 5) = 6
    cout << Max(a++, b++) << endl;
    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 7

    // C-String Comparison
    char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max(s1, s2) << endl; // Max(white, black) = black
}
```

- In "Side Effects" – the result is wrong, the larger values gets incremented twice
- In "C-String Comparison" – swapping parameters changes the result – actually compares pointers

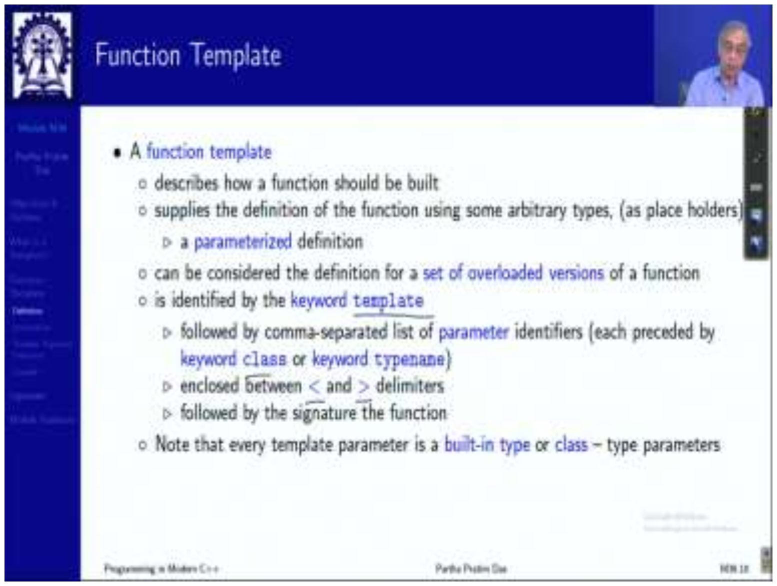
Programming in Modern C++ Partha Pratim Das M08.8

So, for the same, if I, same macro, if I do this, Max(a++, b++), looking at this, we will think that it will take the original value of a, original value of b and compute the maximum and give me the incremented values. But as it turns out, it actually gives, if I do Max of 3 and 5, it actually gives 6. Why is that happening? Because it is textually replaced, we talked about this, it is textually replaced. So, whichever is a larger value for that, the increment has already happened.

And finally, for the larger value, two increments happen, because in the next time, when you have actually chosen the value that the, at that point also. So, you had two a++ happening if a is larger, or 2 happening if b is larger. So, this is this is a major pitfall, this is a major side effect. The other side effect is what will happen if I try to do it for a type for which this comparison operator is not appropriate. So, I have char*, s has two strings, and I do, I have put black in one and white in the other, and I do the Max, I have put white in there, black in there, I have just swapped the order in which I have put.

You can see, if you just do the first one, you might think that it is working correctly, because we get white. But if you do the other, you will see that it is not working, it is giving you black, why? Because this comparison is just comparing the pointer values, not the strcmp function, because you have not no way of saying that. So, you have, if your comparison operator is something different from the basic built-in type operator, then also this solution is not going to work. So, it will not work for string, it will not work for any class that I defined for myself. Because the comparison for that class will not be known to the macro processor.

(Refer Slide Time: 08:52)



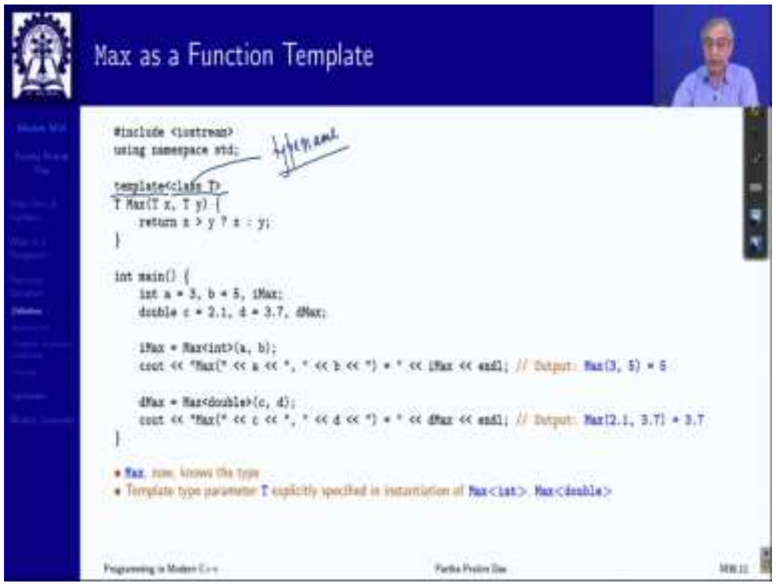
Function Template

- A function template
  - describes how a function should be built
  - supplies the definition of the function using some arbitrary types, (as place holders)
    - a parameterized definition
  - can be considered the definition for a set of overloaded versions of a function
  - 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 signature the function
  - Note that every template parameter is a built-in type or class – type parameters

Programming in Modern C++ Partha Pratim Das

So, this is where function template comes in. It is a parameterized definition, parameterized by the type, and it is prefixed by the keyword template. And within after that, after the function name within the pair of angular brackets, you put the different types of type parameters that you want to put in, you can put in a type parameter by the keyword class or by the keyword typename. It is best to, instead of doing a theoretical discussion, it is best to look at an example.

(Refer Slide Time: 09:32)



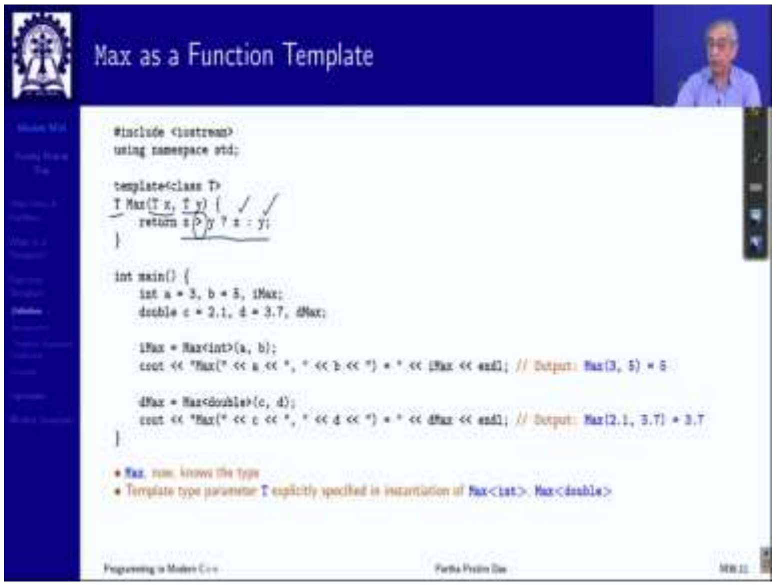
Max as a Function Template

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7
}
```

typename

- Max, now, knows the type
- Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>

Programming in Modern C++ Partha Pratim Das M08.11



Max as a Function Template

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7
}
```

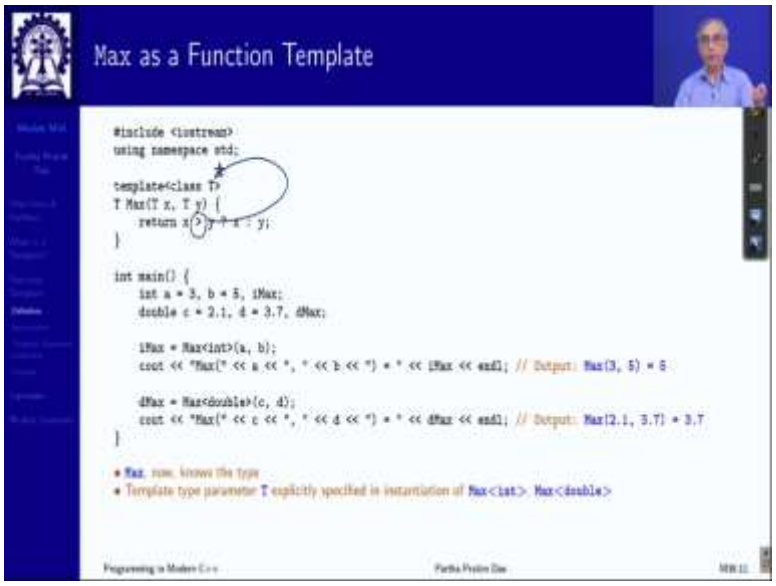
- Max, now, knows the type
- Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>

Programming in Modern C++ Partha Pratim Das M08.11

So, I am writing the Max again. This is mandatory to tell the compiler that you are going to do a generative program. Then you put the type parameter with angular bracket. So, you are saying that T is a type. You say that by saying class, there is an alternate you can say this by typename also. They are exactly the same, anyone you can choose.

Now, having put that, what T becomes, T is neither integer nor double, nor my class, nor complex, T is a type variable. So, here, you are saying, so far, we have dealt with value variables, where the type is known, the value is changing. Here the type itself is not known, I can give it a type. And then I say that my logic of Max is, Max takes two variables of the same type. So, let that type be T. It will return the maximum which is of the same type T, and this is my logic, x greater than y than x otherwise y. And what is this what is this greater than?

(Refer Slide Time: 11:01)

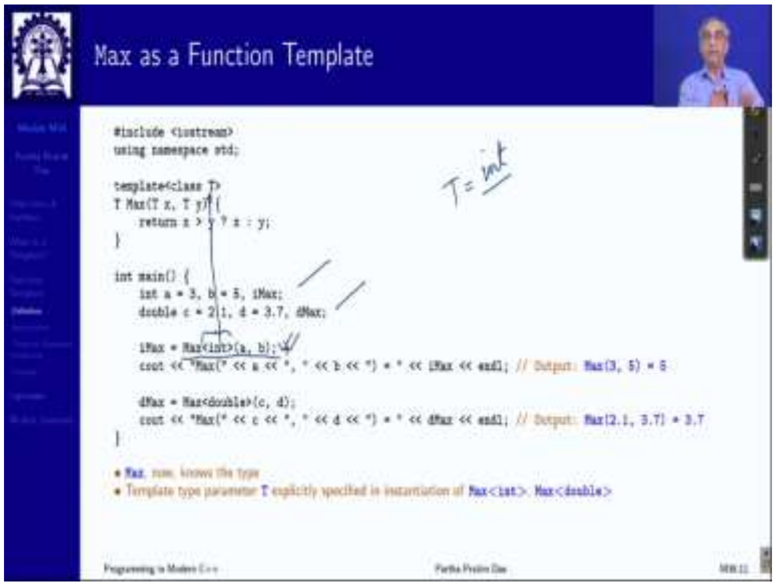


Max as a Function Template

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7
}
```

- Max, now, knows the type
- Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>

Programming in Modern C++ Partha Pratim Das M08.11



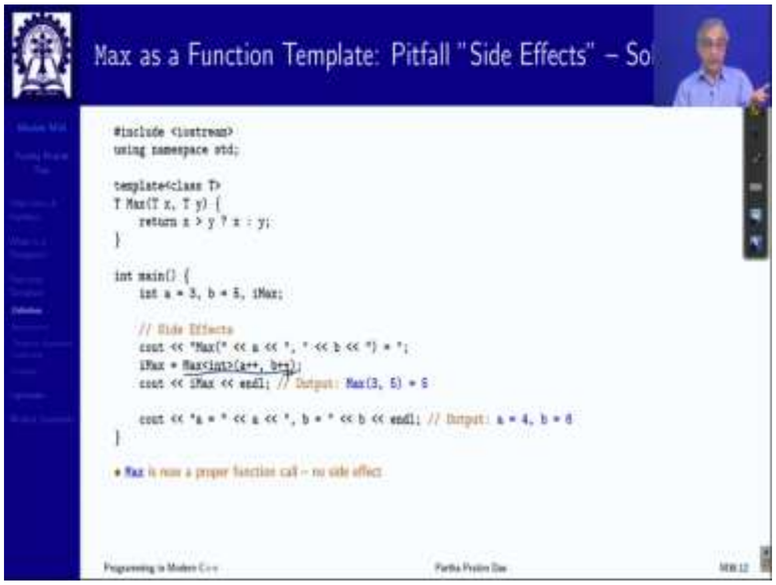
Max as a Function Template

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl; // Output: Max(3, 5) = 5

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl; // Output: Max(2.1, 3.7) = 3.7
}
```

T = int

- Max, now, knows the type
- Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>

Programming in Modern C++ Partha Pratim Das M08.11


Max as a Function Template: Pitfall "Side Effects" – So

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

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;

    // Side Effects
    cout << "Max(" << a << ", " << b << ") = ";
    iMax = Max<int>(a++, b++);
    cout << iMax << endl; // Output: Max(3, 5) = 5

    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 6
}
```

- Max is now a proper function call – no side effect

Programming in Modern C++ Partha Pratim Das M08.12

This has to be the greater than of the type T. So, it is, here, it is not fixing the greater than comparison operator globally, it is just making it specific to the type T. We call it a trait of the type T, a trait of this type is, it has a Boolean comparison operator greater than, which takes two values of this type and gives a Bool result.

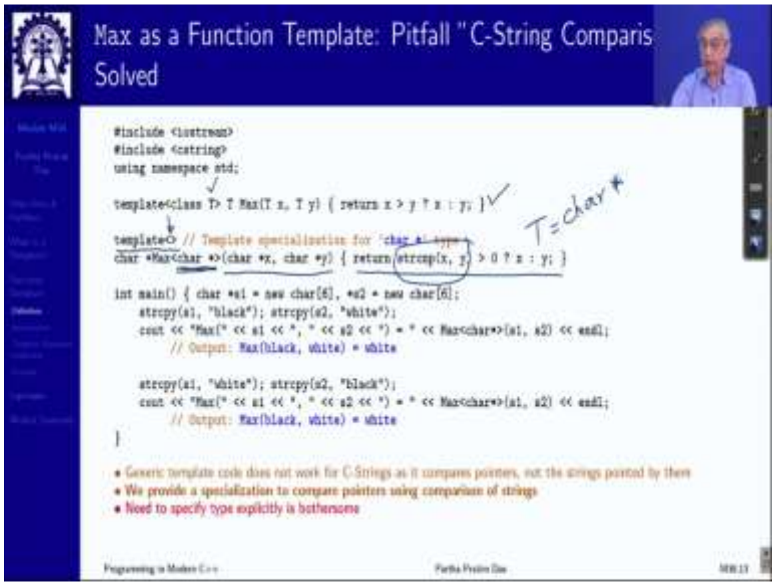
Let us look at the use. So, integer variables, double variables, this is how you call the function. So, instead of calling it Max(a, b), after Max, within corner brackets, you provide the type. So, this becomes the value of T, T becomes equal to int. So, what the compiler does when the compiler comes across this, compiler takes T as int and generates a code for Max.

Similarly, when you do for double, compiler takes T as double, and generates a code for Max, where it is double. So, in this, you can see that the compiler is automatically generating two overloaded versions of Max, because I have asked to have a Max for a type T which I have instantiated with type int in one context and type double in the other context. this is the notion of the generative programming.

So, this is not the code that we will run, but from this template of the code, a code for int will be generated for this case. From this template of the code a, code for double type will be generated in the second case. And they are overloaded and the appropriate one will be called, and now, the compiler knows the type. So, the any kind of side effect that we thought of that we could see like in here in a++ we saw earlier in case of macros will not be there, because it is a proper function.

So, it will only take a and b as x and y having the type int. So, one part of the problem of macro gets solved.

(Refer Slide Time: 13:40)



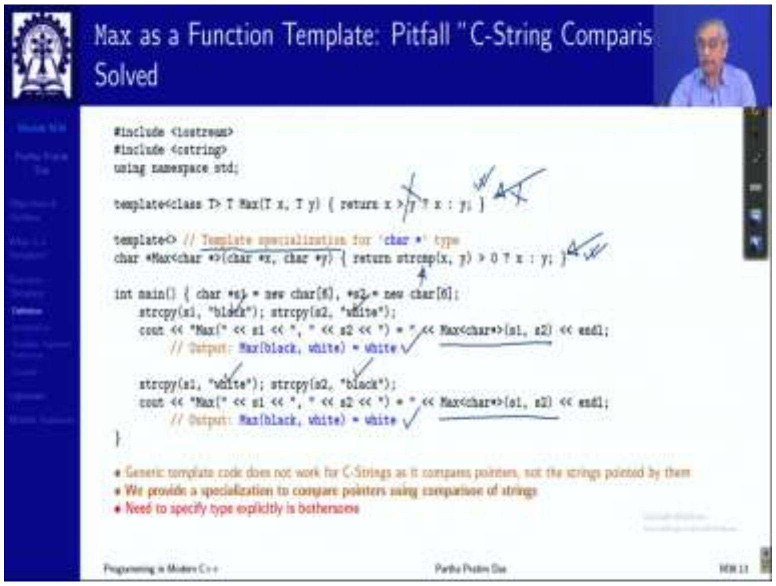
Max as a Function Template: Pitfall "C-String Comparis Solved

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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<> // Template specialization for 'char *' type
char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white
}
```

T=char*

- Generic template code does not work for C-Strings as it compares pointers, not the strings pointed by them
- We provide a specialization to compare pointers using comparison of strings
- Need to specify type explicitly is bothersome

Programming in Modern C++ Partha Pratim Das M08.13



Max as a Function Template: Pitfall "C-String Comparis Solved

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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<> // Template specialization for 'char *' type
char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { char *s1 = new char[6], *s2 = new char[6];
    strcpy(s1, "black"); strcpy(s2, "white");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white

    strcpy(s1, "white"); strcpy(s2, "black");
    cout << "Max(" << s1 << ", " << s2 << ") = " << Max<char*>(s1, s2) << endl;
        // Output: Max(black, white) = white
}
```

- Generic template code does not work for C-Strings as it compares pointers, not the strings pointed by them
- We provide a specialization to compare pointers using comparison of strings
- Need to specify type explicitly is bothersome

Programming in Modern C++ Partha Pratim Das M08.13

The second was with the string. So, what I can do is this is something very, very interesting again. We have provided this template. So, we are saying that for T, this is the template. But if the type is char*, if the type is char*, then I have a different template. If the type is char*, then I have a different template.

Now, you will say that how can I have a template? Because there is only one parameter T. So, if I say that parameter is char*, then everything in the code must be known. Yes it is. So, it is not necessary that a template must have a type parameter. I can have it template without parameter also. So, what I have done in this case, I have specified explicitly that T is char star. I do that by writing the name of Max in this way, as the way I was invoking.

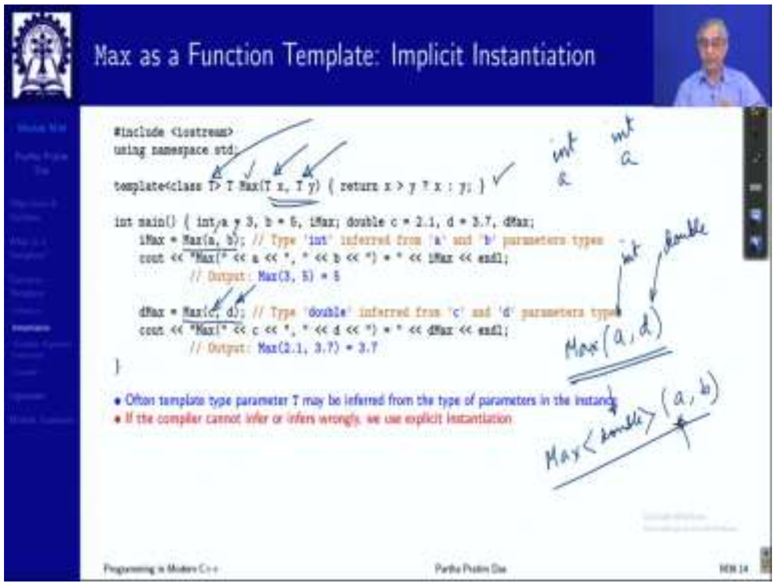
Therefore, the temp, this template has no separate type parameter. So, this type parameter is no more a parameter. So, therefore, it has a null type parameter list. And then using T as char*, I write the signature, and then I write the specific code where I can use this specialized compile as an operator.

So, this is called, this, so what I have done is I have, now there is, this template itself can be, can generate multiple overloads int, double and so on. And I have explicitly given an, a overload for it. So, if it is used with char*, this template will not be used for char*, because I have given particular specialization. So, this feature is known as template specialization.

So, if I give a generic template for a large number of types and specific templates for specific types, which has a different handling, then I make use of the specialization. Here since we, I had only one template parameter, the specialization is full. If I had two or more parameters, it is possible that I specialize on one parameter and retain the other parameters. So, those are more complex examples.

Once we have done this, then if we look at doing this with black and white, and with white and black, if I do this Max, now we will find that in both cases, I get white, because actually, this code is not being used, but this specialized code will be used, which rightly does, the strcmp based on the pointer types, and does a string comparison. So, with this template specialization feature, we also circumvent the pitfall of C string comparison, which macros were giving us.

(Refer Slide Time: 17:21)



Max as a Function Template: Implicit Instantiation

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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

int main() { int a = 3, b = 5, iMax; double c = 2.1, d = 3.7, dMax;
    iMax = Max(a, b); // Type 'int' inferred from 'a' and 'b' parameters types
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
          // Output: Max(3, 5) = 5

    dMax = Max(c, d); // Type 'double' inferred from 'c' and 'd' parameters types
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
          // Output: Max(2.1, 3.7) = 3.7
}
```

- Often template type parameter T may be inferred from the type of parameters in the instance
- If the compiler cannot infer or infers wrongly, we use explicit instantiation

Programming in Modern C++ Partha Pratim Das M08.14

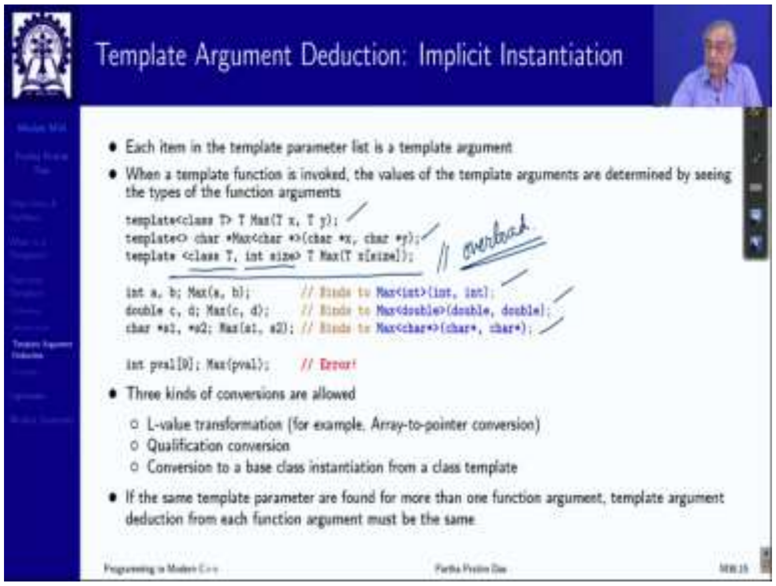
So, we have, in a way, we have a lot of things solved. In terms of the template, I have the generator, like the macro does in a weaker way. It is strongly typed, it behaves exactly like a function, and it can be specialized for certain types. The only, you know, somewhat pain remains in the syntax font, because I have to specify the type for which I am calling that function. Now, to a large extent, that pain can also be removed. That is, I can just, given this template, I can just call it as Max. How does it work? Very simply the parameters of types. So, the type of a is int, type of b is int, I know.

So, and I am trying to call Max. So, in terms of calling Max, I need to know the type T. And I know, an int is expected here and int is expected here. So, I can easily resolve that T has to be int for this whole call to succeed. Similarly, in this case, c and d are double. So, T has to be doubled for this to succeed. But if I call Max(a, d), this call will not compile. Why? This is int? This is double, it needs both these types to be same, this particular template, they are different. So, what is the value of T? Is it int or is it double? No conversions allowed? Nothing.

It has to be exact. Otherwise, there will be a lot of confusion. So, the compiler will say that I am confused, I cannot decide. You have to tell me what to do? But okay. I say, no, comparing with integer double, my understanding is the comparison has to happen as a double. So, I say, so I get back to the explicit instantiation syntax, and specify that I am looking for a double function with double. So, T is taken as double. So, when a as an int is passed on, a as a parameter will be implicitly converted to double and the function. But without explicit mention of that type, this is not logically resolvable. But so, but at the same time, it is understandable that most often you would not need to do that.

So, you can just use the implicit instantiation that is available from the compiler, and thereby, you get the benefit all benefits together. It looks just like a function and you did not have to write it for every type, but it works for every type, you know, reasonably and it can specialize for specific types and so on so forth. It is a win all situation.

(Refer Slide Time: 20:43)



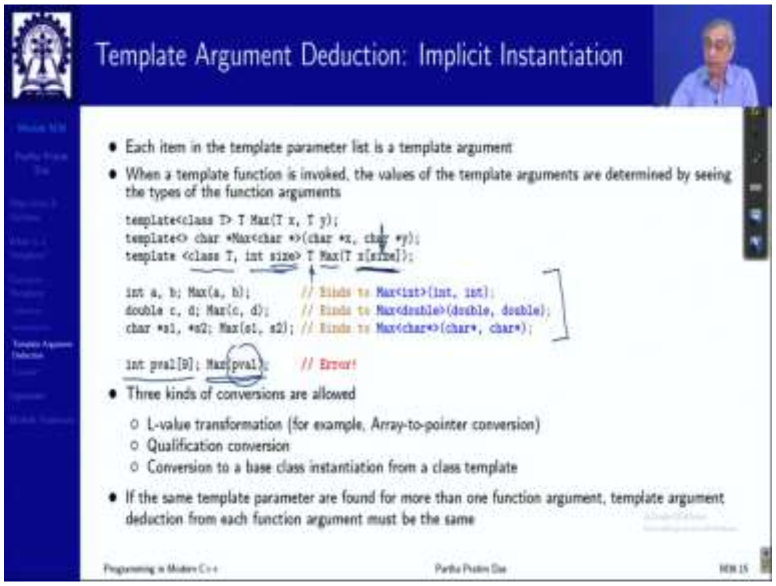
Template Argument Deduction: Implicit Instantiation

- 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]);          // overload

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(pval);       // Error!
```

- 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 M08.15



Template Argument Deduction: Implicit Instantiation

- 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(pval);       // Error!
```

- 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 M08.15

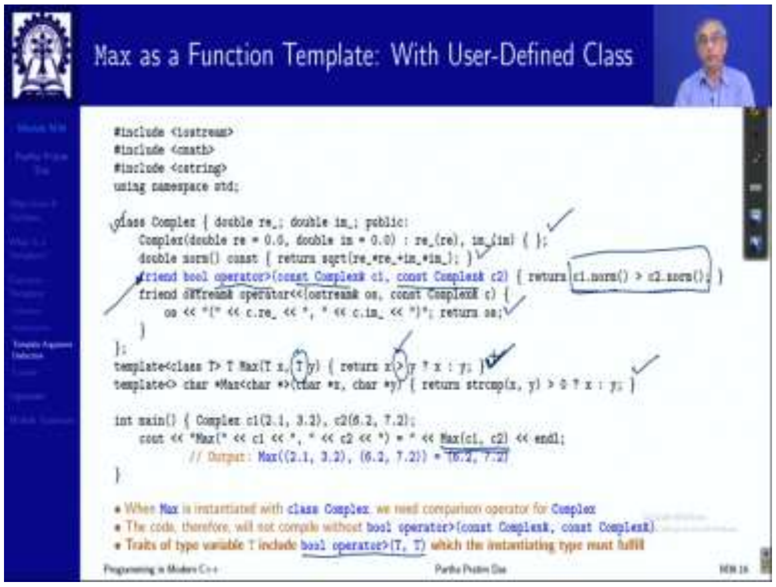
So, this is some bit more of this. These are the different instantiations that you have. This is in the general, this is in the specialized. Now, what we provide by the name Max, we provide yet another template, but with a different parameter signature. What does that call, what is that called? Overload. So, we are trying to overload the template. What do I just read into this template? It is saying like, before it is saying class T, then it says int size. This is a specific feature of template that a template parameter could also be an integer, nothing else. You cannot have a char there or anything, but it can be an integer wherein you can pass on a integral number.

Why do I need that? Look at the intention carefully. So, Class T size, and I am saying, I want to do a Max for x size. So, what is x? x must be an array. So, x is an array of elements of type T and the array, if it is an array, I need to know what is the size it has got. And then it returns the, so this also does a Max, but in a different context. Earlier, Max was between two values, now it is between a array of numbers. But I want the same the same or very closely same signature to be there, I want the same name to be used, I do not want to call this array Max or something different.

So, these will easily bind. But if I try to do this Max pval, then I will get an error. This will not compile. The reason is simple, is, as you can understand that it needs to know the array as well as the size. In C++, the compile time defined size cannot be automatically passed as, passed along with the array. Array is just the pointer, it does not say how many other, it is programmer's responsibility to remember that.

So, it cannot resolve the size part, and it will give it will an error. We will soon see how to get, get around with this. So, in terms of doing this kind of overloading, three kinds of conversions will be allowed. One is L-value transformation that is array two pointer kind of conversion, qualification conversion and conversion to the base class instantiation from class template. So, using these, you can actually create overloads of template functions and automatically, rather implicitly deduce the arguments, and we saw the pitfall where we were not able to deduce it implicitly.

(Refer Slide Time: 24:13)



Max as a Function Template: With User-Defined Class

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

class Complex { double re_; double im_; public:
    Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) { };
    double norm() const { return sqrt(re_*re_+im_*im_); }
    friend bool operator>(const Complex& c1, const Complex& c2) { return c1.norm() > c2.norm(); }
    friend ostream& operator<<(ostream& os, const Complex& c) {
        os << "(" << c.re_ << ", " << c.im_ << ")"; return os;
    }
};
template<class T> T Max(T x, T y) { return x > y ? x : y; }
template<> char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() { Complex c1(2.1, 3.2), c2(6.2, 7.2);
    cout << "Max(" << c1 << ", " << c2 << ") = " << Max(c1, c2) << endl;
          // Output: Max((2.1, 3.2), (6.2, 7.2)) = (6.2, 7.2)
}
```

- When Max is instantiated with class Complex, we need comparison operator for Complex
- The code, therefore, will not compile without bool operator>(const Complex&, const Complex&)
- Traits of type variable T include bool operator>(T, T) which the instantiating type must fulfill

Programming in Modern C++ Partha Pratim Das M08.16

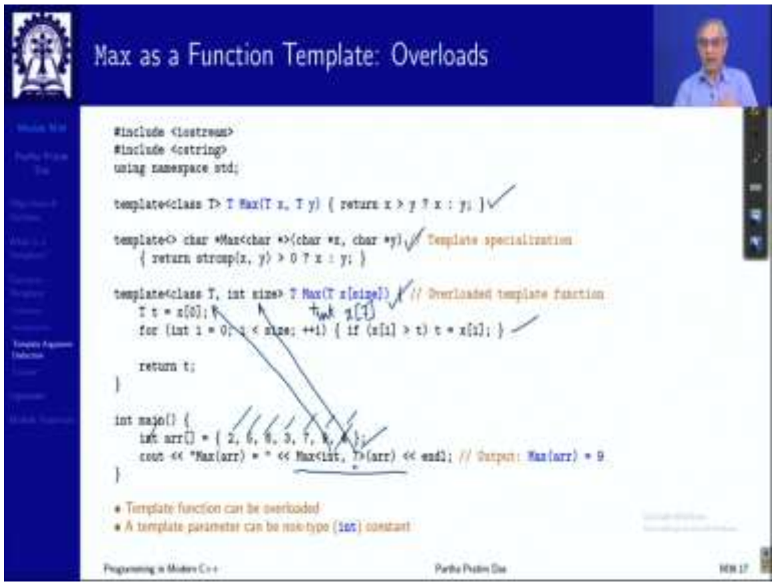
So, let us now try to do this Max for a user-defined class. Let me define the class complex with its constructor, its non-function, output streamer and so on. I have a specialization here. I have a general generic definition of the Max here and a specialization for char star. I want to use Max for c1 and c2, two complex numbers. So, I have two ways. One is, I can again specialize for complex and provide a different specialization. But I would like to actually do it smarter, I would like to use a general definition itself. So, but if I try, it will not compile, because it does not know how to compare these two.

But for that, we have already discussed operator overloading. This operator overloading will solve this problem. So, what I have to do is, if I want complex numbers to be compared, naturally I have to provide the logic for that comparison. So, my logic here is their norms to be compared, whatever, you can give some other logic. But the basic point is, I must provide a operator greater than, a overload for operator greater than, which takes 2 complex and returns a bool. So, remember I had mentioned that this type T has a trait. Its trait is, it will have a operator greater than available which takes two values of this type T and returns a Boolean.

So, if I want, this Max template to work for my defined class, I have to provide an overload for the comparison function. This in any case is logical, because I want to find a Max, so I must know how to compare them, even if I had to write a separate function for it, I would have required to kind of provide this algorithm for doing it.

So, but by providing this, by making the class more self-sufficient and complete, I let the template code generator to generate code for my complex class. And so for any user-defined class, that would want to use Max. All that is required is satisfy the trait, provide the comparison operator which you will need to obviously say for your class, you have to tell the algorithm, and with that, you do not need to write any other Max function, this template will do the entire job for you. So, you can see that how it has, it makes it easy to reuse code at a kind of highest level and makes things really easy for us.

(Refer Slide Time: 27:25)



Max as a Function Template: Overloads

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

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<> char *Max<char *>(char *x, char *y) // Template specialization
    { return strcmp(x, y) > 0 ? x : y; }

template<class T, int size> T Max(T x[size]) { // Overloaded template function
    T t = x[0];
    for (int i = 0; i < size; ++i) { if (x[i] > t) t = x[i]; }

    return t;
}

int main() {
    int arr[] = { 2, 6, 8, 3, 7, 9, 4 };
    cout << "Max(arr) = " << Max<int, 7>(arr) << endl; // Output: Max(arr) = 9
}
```

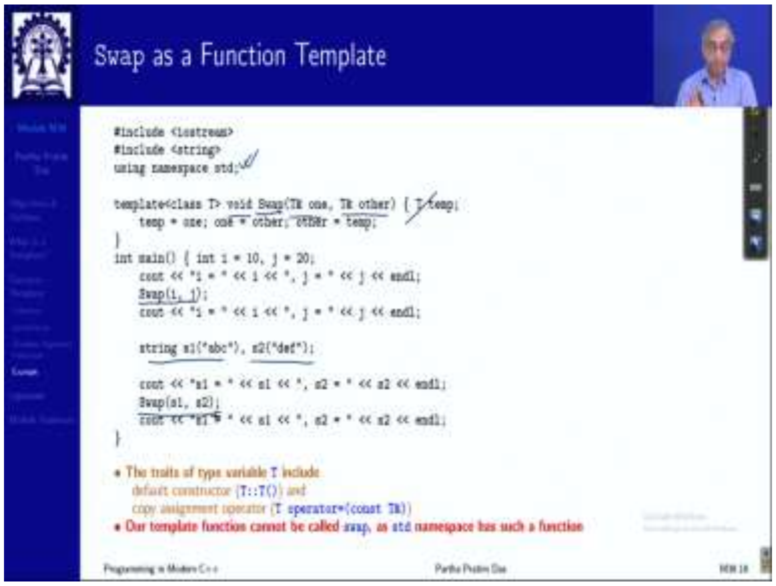
- Template function can be overloaded
- A template parameter can be non-type (int) constant

Programming in Modern C++ Partha Pratim Das M08.17

In terms of the, coming back to the overload question, we did see that this is one definition, this is a specialization and this is a overload which we could not make work. Here I have just also provided the logic for it. So, this needs to be, this cannot be implicitly instantiated, because it needs to get both these as particularly a parameter size, which cannot be implicitly deduced by the compiler. So, for using this, the only additional pain we will have to bear is to do an explicit instantiation that is specify the type, the type of the array, and the size. Here we have 1, 2, 3, 4, 5, 6, 7. 7, so we specify the size 7.

So, this becomes T, size becomes 7. So, here you are actually calling x[7] int. It becomes a valid parameter type and things will work. So, you can see that, you know, you can not only have multiple generative forms specialized templates, you can also have overloads. And overloads again can have specializations. All of these are actually overloads that are automatically coming in as a part of the template mechanism.

(Refer Slide Time: 28:58)

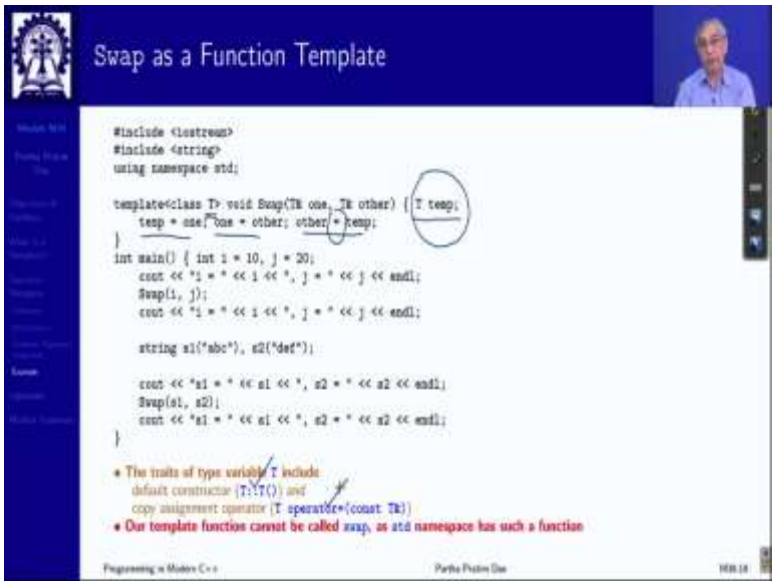


Swap as a Function Template

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

template<class T> void Swap(T& one, T& other) { T temp;
    temp = one; one = other; other = temp;
}
int main() { int i = 10, j = 20;
    cout << "i = " << i << ", j = " << j << endl;
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << endl;

    string s1("abc"), s2("def");

    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
    Swap(s1, s2);
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}
```

- The traits of type variable T include
  default constructor (T::T()) and
  copy assignment operator (T operator=(const T&))
- Our template function cannot be called swap, as std namespace has such a function

Programming in Modern C++ Partha Pratim Das M08.18



Swap as a Function Template

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

template<class T> void Swap(T& one, T& other) { T temp;
    temp = one; one = other; other = temp;
}
int main() { int i = 10, j = 20;
    cout << "i = " << i << ", j = " << j << endl;
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << endl;

    string s1("abc"), s2("def");

    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
    Swap(s1, s2);
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}
```

- The traits of type variable T include
  default constructor (T::T()) and
  copy assignment operator (T operator=(const T&))
- Our template function cannot be called swap, as std namespace has such a function

Programming in Modern C++ Partha Pratim Das M08.18

This is just to show the swapping function, very common swap function as a template. Mind you, I am writing this swap as capital S, because I am in the namespace std, namespace std already has a provided swap function. If I do not use that namespace, then I can use the swap with lowercase. I just chose to use a different name. So, it takes a reference to two objects of the same type. Swap, same, does not return anything, and this is a swapping logic, which is clear. So, I can

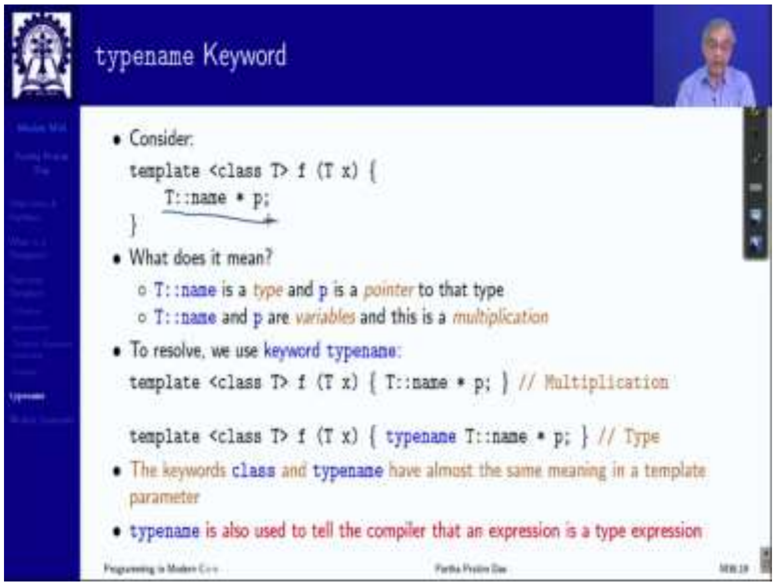
I can use this for two numbers, they will get swapped, I can use it for two strings, they will get swapped. And so on. This is easy to see. Now, the question is, what is the trait? What is the trait of this type T that I am using for swapping. That is what are the properties, this type must satisfy, so that it can be used as a template instantiation type parameter. So, what are the things I am having to do?

I am having to create a T temp. So, it must have a constructor, and that constructor must be default. Because other than that, without a default constructor, I cannot certainly make this code work. I cannot pass any parameter to that. So, it must have a default constructor. And obviously, corresponding destructor.

Then I am making copies here. So, it must define a copy assignment operator, it must have a copy assignment operator. Otherwise, I might get into that deep copy, shallow copy issues. I do not know what other types that you are, it may be an old structure which has a lot of pointers with allocated values. So, you need to, may need to do some deep copy.

But this copy assignment has to be supported. So, default constructor, destructor of course, and copy assignment operator. Without these, the swap cannot be used. So, that is the, that is a take back that to be able to define the template, you must always identify what is a trait of the type that you are issuing, that trait will have to be satisfied, only then you will be able to use a type as a type parameter in that template instantiation.

(Refer Slide Time: 31:53)



typename Keyword

- Consider:

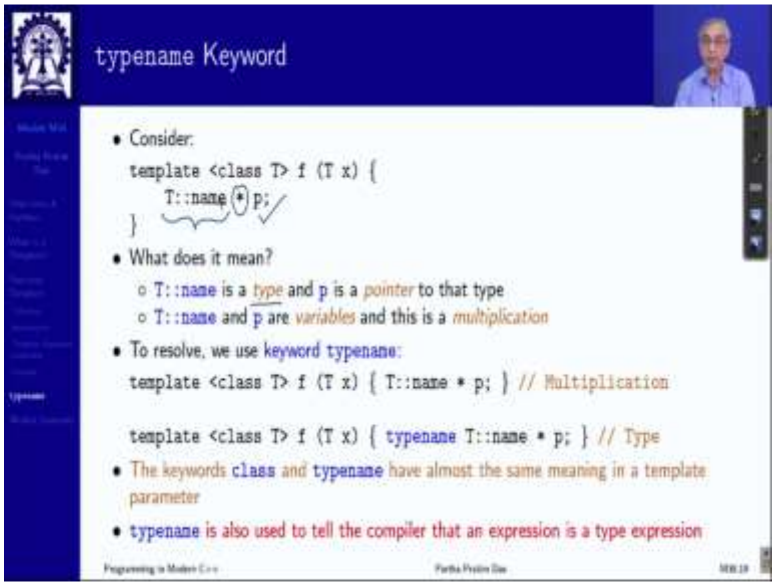
```
template <class T> f (T x) {
    T::name * p;
}
```

- What does it mean?
  - T::name is a *type* and p is a *pointer* to that type
  - T::name and p are *variables* and this is a *multiplication*
- To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

- The keywords class and typename have almost the same meaning in a template parameter
- typename is also used to tell the compiler that an expression is a type expression

Programming in Modern C++ Partha Pratim Das M08.19



typename Keyword

- Consider:

```
template <class T> f (T x) {
    T::name * p;
}
```

- What does it mean?
  - T::name is a *type* and p is a *pointer* to that type
  - T::name and p are *variables* and this is a *multiplication*
- To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

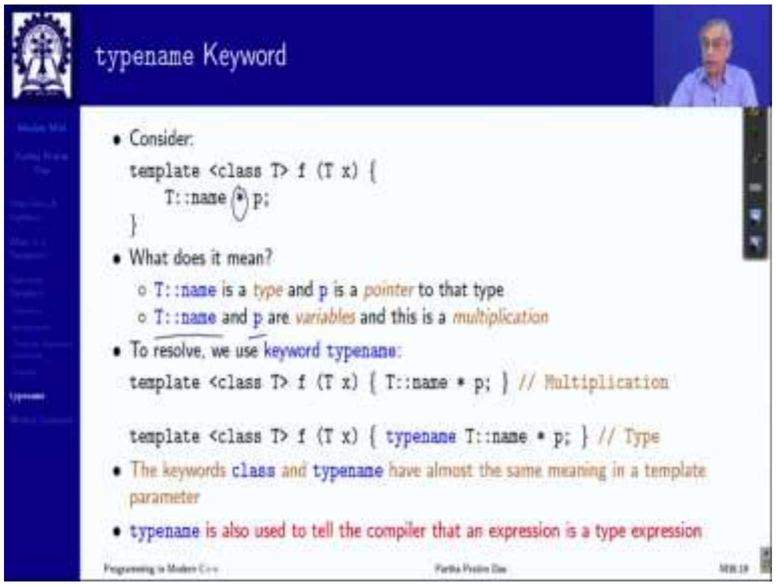
- The keywords class and typename have almost the same meaning in a template parameter
- typename is also used to tell the compiler that an expression is a type expression

Programming in Modern C++ Partha Pratim Das M08.19

The… I said that template type parameter can be prefixed with the class or typename in general, no differences. There is one context for the introduction of this typename keyword. There is a reason why the type name is given. Consider a template here. T is an unknown type ::name *p.

What does this mean? What does this expression mean? We just observed that this expression can be interpreted in two ways logically correctly. One is, I can think that my T:: name is a type. T is a type, within that, I have defined, I have done some type diff, I have another type. So, T::name could be a type. In that case, it is a pointer declaration, that is this is a typical pointer star.

(Refer Slide Time: 33:13)



typename Keyword

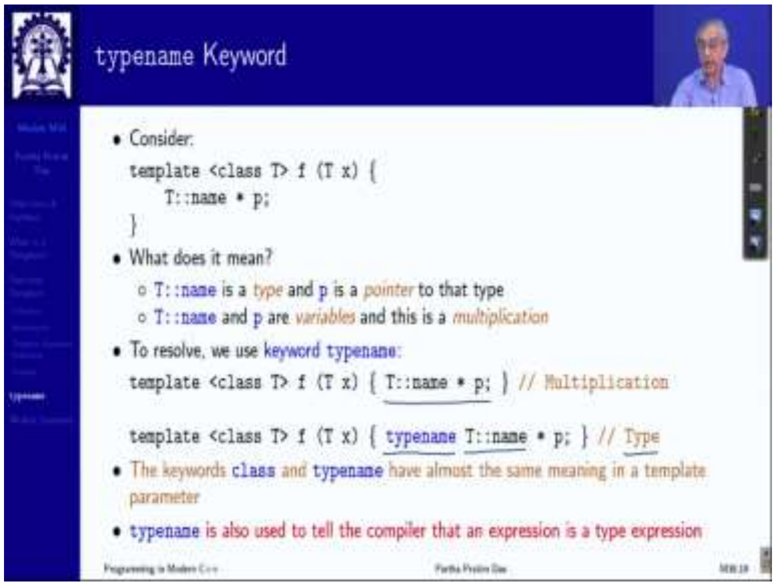
- Consider:

```
template <class T> f (T x) {
    T::name * p;
}
```

- What does it mean?
  - T::name is a *type* and p is a *pointer* to that type
  - T::name and p are *variables* and this is a *multiplication*
- To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

- The keywords class and typename have almost the same meaning in a template parameter
- typename is also used to tell the compiler that an expression is a type expression



typename Keyword

- Consider:

```
template <class T> f (T x) {
    T::name * p;
}
```

- What does it mean?
  - T::name is a *type* and p is a *pointer* to that type
  - T::name and p are *variables* and this is a *multiplication*
- To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication

template <class T> f (T x) { typename T::name * p; } // Type
```

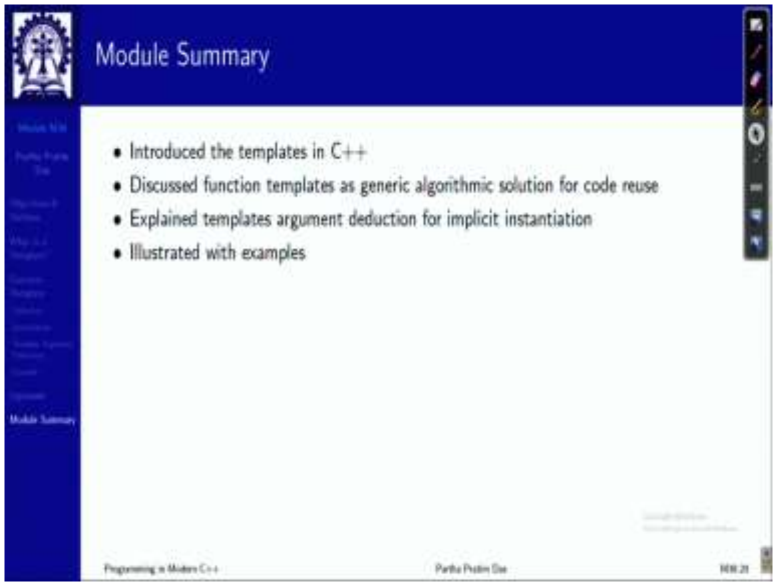
- The keywords class and typename have almost the same meaning in a template parameter
- typename is also used to tell the compiler that an expression is a type expression

In other words, it could also be that T::name, name is a variable, not a type in type T and p is another variable. So, this basically is a multiplication. So, one is the conflict of understanding, semantics is between a pointer declaration and a multiplication. Now, naturally there is not enough information available on this for the compiler to deduce. So, user has to provide, has to tell which one is this?

So, what has been set in C++ is that if you just write this, this is T::name is taken to be variable, and therefore it is a multiplication, right? This is the default. If I want to mean that T:: name is actually a type, you have to put the keyword typename before that. If you put the keyword typename before this T::name, then this will be treated as a type and it will not be treated as a multiplication.

So, that is the specific context in which the typename keyword is required with a different behavior. You could not have used the keyword class, because that has, in this context, that has all different kinds of semantics already loaded. Otherwise, class and typename are exactly interchangeable in terms of the template parameter definitions.

(Refer Slide Time: 34:39)



Module Summary

- 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

So, to summarize, we have introduced templates, the generic programming, meta programming in C++ and we have discussed the function template that is how we can write very core algorithms in short function templates that can help generating a larger number of overloaded function when I instantiate with different kind of types and instantiation could be implicit as well, in some cases, it must be explicit. And then based on that, template arguments can be deduced. So, thank you very much for your attention. We will continue this in the next module discussing about class templates.