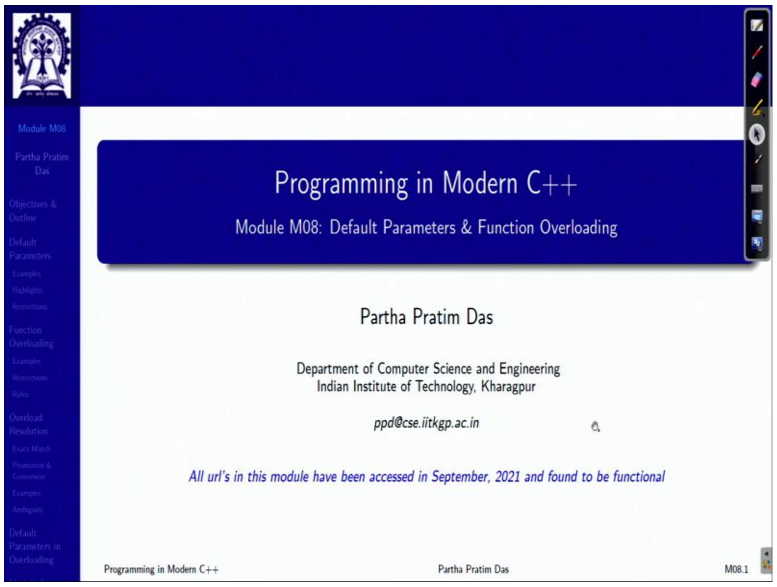
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 8
Default Parameters & Function Overloading

Welcome to programming in modern C++. We are in week 2, and we are going to discuss module 8.

(Refer Slide Time: 0:34)



Programming in Modern C++

Module M08: Default Parameters & Function Overloading

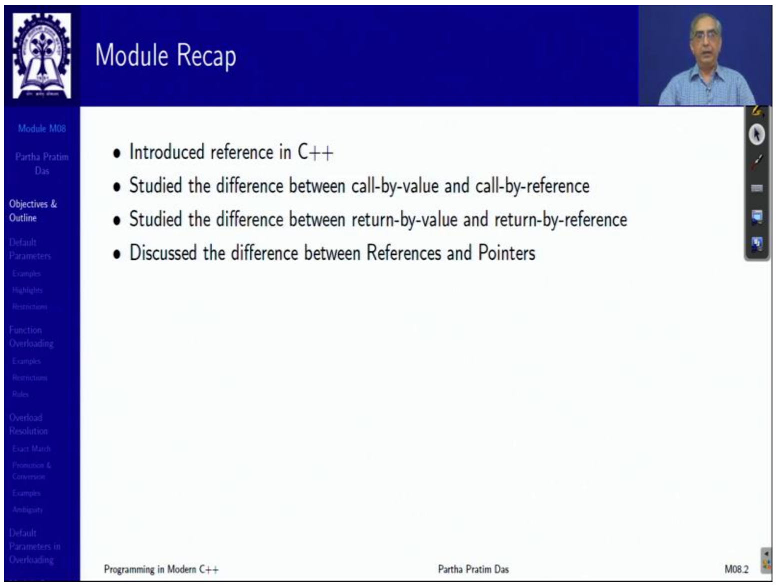
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

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

Programming in Modern C++ Partha Pratim Das M08.1



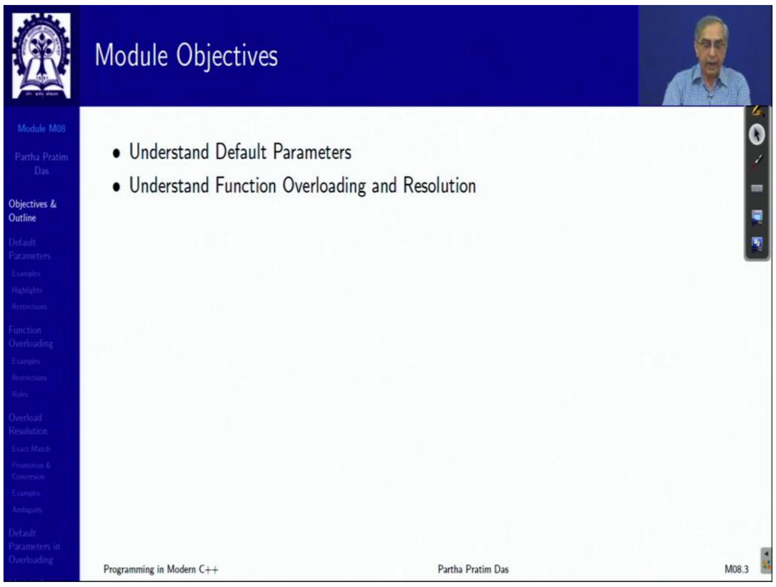
Module Recap

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers

Programming in Modern C++ Partha Pratim Das M08.2

In the last module, we have introduced the concept of reference with discussion on the parameter passing mechanisms of functions in C++, call-by-value, and call-by-reference. And also, the value return mechanism that is returned-by-value and return-by-reference studied them and saw the advantages of call-by-reference in certain contexts. And finally, we have made a comparative study of pointers as well as references.

(Refer Slide Time: 1:15)



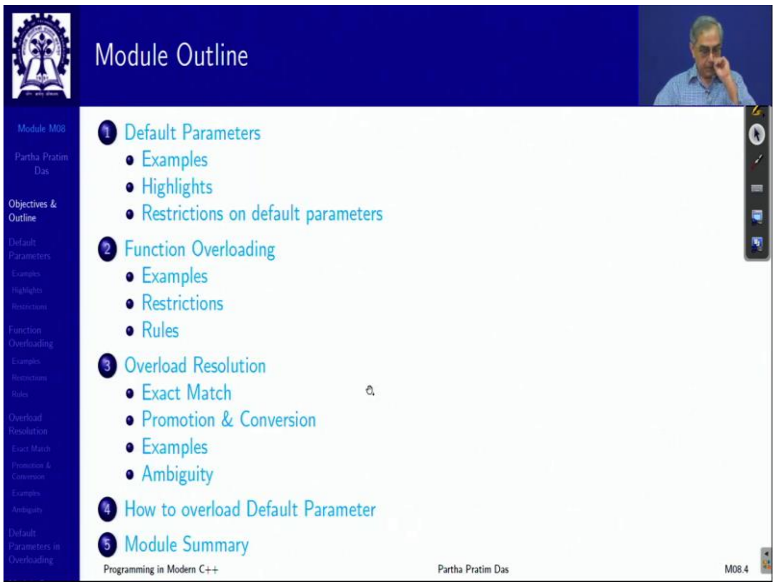
Module Objectives

- Understand Default Parameters
- Understand Function Overloading and Resolution

Programming in Modern C++ Partha Pratim Das M08.3

In this module, we will talk about default parameters, what they are, and what is function overloading, and how that overloading is resolved. It is a new concept, again, being introduced.

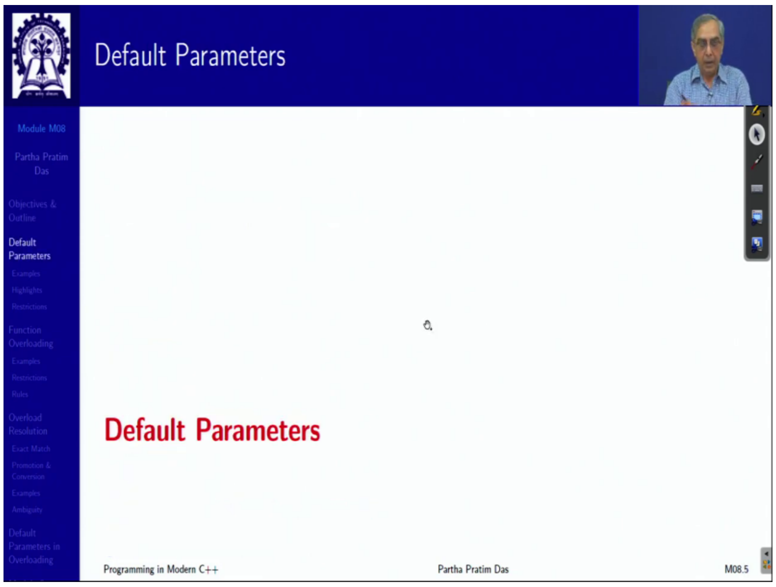
(Refer Slide Time: 1:30)



Module Outline

1. Default Parameters
   - Examples
   - Highlights
   - Restrictions on default parameters
2. Function Overloading
   - Examples
   - Restrictions
   - Rules
3. Overload Resolution
   - Exact Match
   - Promotion & Conversion
   - Examples
   - Ambiguity
4. How to overload Default Parameter
5. Module Summary

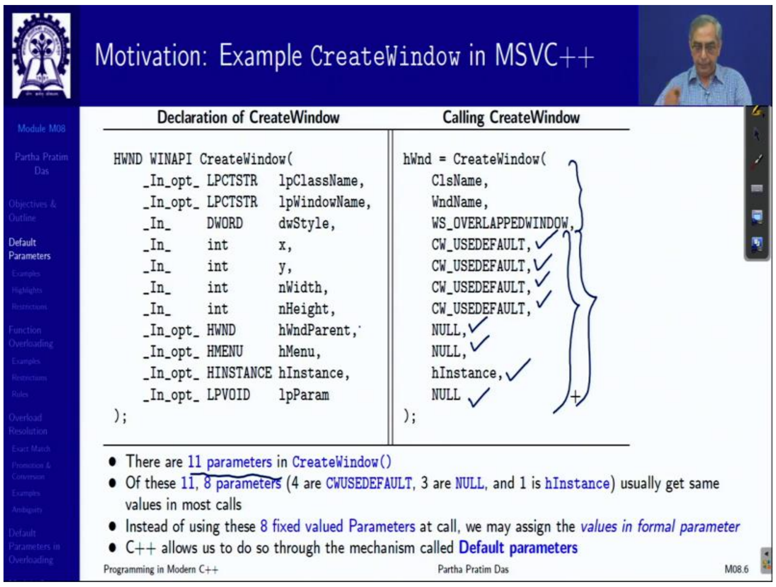
Programming in Modern C++ Partha Pratim Das M08.4


Module M08
Partha Pratim Das
Objectives & Outline
Default Parameters
Examples
Highlights
Restrictions
Function Overloading
Examples
Restrictions
Rules
Overload Resolution
Exact Match
Promotion & Conversion
Examples
Ambiguity
Default Parameters in Overloading

Default Parameters

Default Parameters

Programming in Modern C++ Partha Pratim Das M08.5


Module M08
Partha Pratim Das
Objectives & Outline
Default Parameters
Examples
Highlights
Restrictions
Function Overloading
Examples
Restrictions
Rules
Overload Resolution
Exact Match
Promotion & Conversion
Examples
Ambiguity
Default Parameters in Overloading

Motivation: Example CreateWindow in MSVC++

Declaration of CreateWindow	Calling CreateWindow

`HWND WINAPI CreateWindow(`
`_In_opt_ LPCTSTR lpClassName,`
`_In_opt_ LPCTSTR lpWindowName,`
`_In_ DWORD dwStyle,`
`_In_ int x,`
`_In_ int y,`
`_In_ int nWidth,`
`_In_ int nHeight,`
`_In_opt_ HWND hWndParent,`
`_In_opt_ HMENU hMenu,`
`_In_opt_ HINSTANCE hInstance,`
`_In_opt_ LPVOID lpParam`
`);`
	`hWnd = CreateWindow(`
	`ClsName,`
	`WndName,`
	`WS_OVERLAPPEDWINDOW,`
	`CW_USEDEFAULT,` ✓
	`CW_USEDEFAULT,` ✓
	`CW_USEDEFAULT,` ✓
	`CW_USEDEFAULT,` ✓
	`NULL,` ✓
	`NULL,` ✓
	`hInstance,` ✓
	`NULL` ✓
	`);`

- There are 11 parameters in CreateWindow()
- Of these 11, 8 parameters (4 are CWUSEDEFAULT, 3 are NULL, and 1 is hInstance) usually get same values in most calls
- Instead of using these 8 fixed valued Parameters at call, we may assign the *values in formal parameter*
- C++ allows us to do so through the mechanism called **Default parameters**

Programming in Modern C++ Partha Pratim Das M08.6

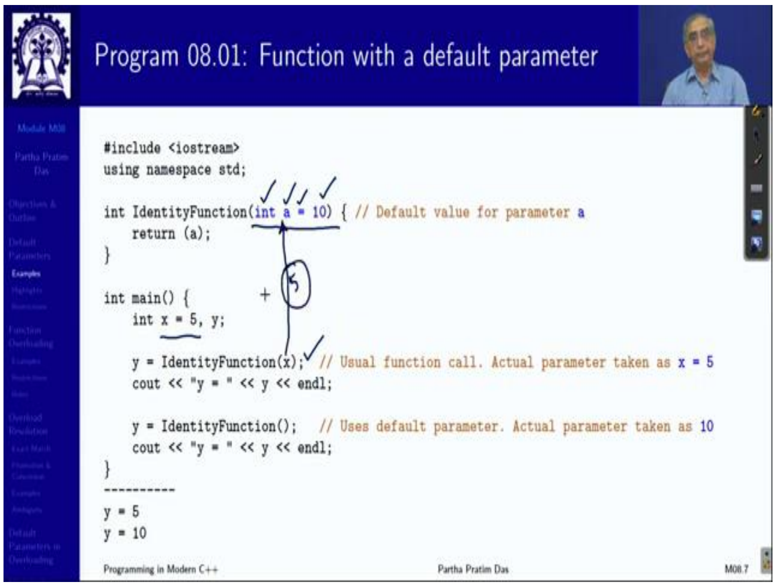
So, this is the outline, which you will find on the left. So, let us talk about default parameters. Now to, for you to realize what is default parameter and why is it important? Let us look at this function header. This is from Microsoft Visual Studio Visual C++, Windows programming. So, this is on left is the declaration of the CreateWindow function. And on the right is a typical call that will be made to the CreateWindow function.

Now, if you look at this, you will find, if you observe carefully, you will find that there are 11 parameters in total. In the CreateWindow, it is a large function, lot of things to be specified. But out of these, if you see 1, 2, 3, 4 are given default values, most cases it will be given default value, as to where you want to create the window, what should be the width, what should be the hight, there is a preferred default, most of the time most programmers would be using. Many a times, these three are set to null.

That is whether it has a parent, whether it is hanging a menu, or whether you are passing some parameters to the window. And the hInstance, the current instance, of the windows on which you are doing this is typically hInstance, called everywhere. So given this, as you can see, out of the 11 parameters, 8 parameters, most of the time, we will take the same value for most of the calls. Only primarily, these three are often what the user would like to set.

So, the question is, do I really need to make the user copy-paste all these 11 parameters, 8 of them is a big list, which hardly differs from one call of the CreateWindow to the other. Or, can I let these default values somehow be represented in the system, so that if the user does not have to change that default value, or does not need to call the function with the value other than the defaulted value, the user does not need to provide these parameters in the call. That is the whole idea.

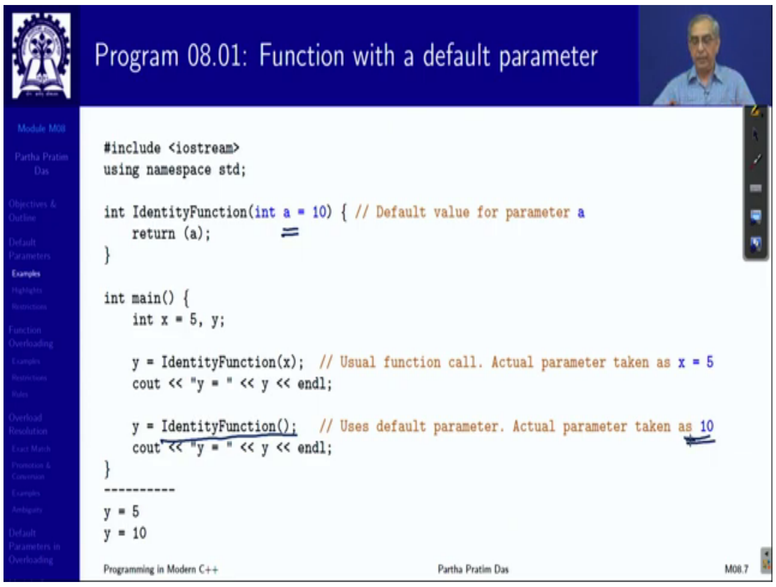
(Refer Slide Time: 4:15)



Program 08.01: Function with a default parameter

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

int IdentityFunction(int a = 10) { // Default value for parameter a
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x); // Usual function call. Actual parameter taken as x = 5
    cout << "y = " << y << endl;

    y = IdentityFunction();  // Uses default parameter. Actual parameter taken as 10
    cout << "y = " << y << endl;
}
----------
y = 5
y = 10
```

Programming in Modern C++ Partha Pratim Das M08.7



Program 08.01: Function with a default parameter

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

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

int IdentityFunction(int a = 10) { // Default value for parameter a
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x);  // Usual function call. Actual parameter taken as x = 5
    cout << "y = " << y << endl;

    y = IdentityFunction();   // Uses default parameter. Actual parameter taken as 10
    cout << "y = " << y << endl;
}
----------
y = 5
y = 10
```

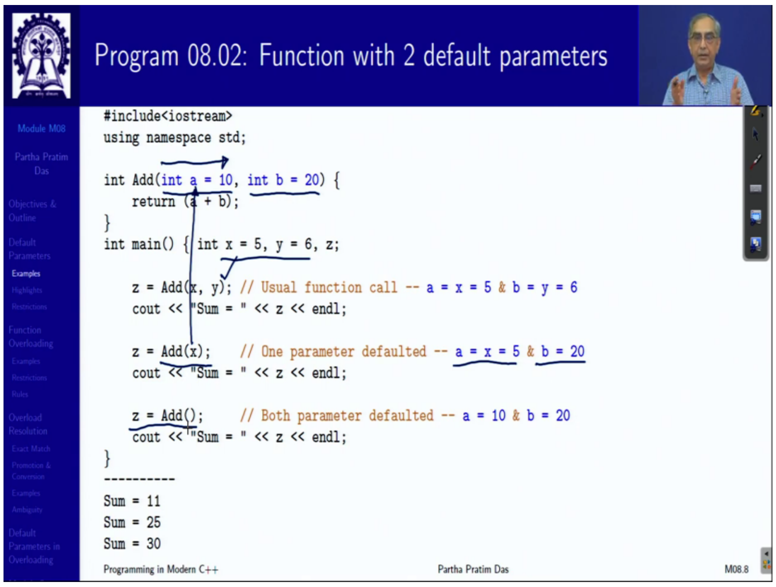
Programming in Modern C++ Partha Pratim Das M08.7

So, let us see a simple example with one parameter which is default. So, this is how you write it. In the header of the function, after you have written the type, and name of the parameter with the initialisation symbol, you write a value.

The idea for this is clear when you call the function. Now, if you call the function with x, which is initialised to 5, if you call it with x, it is a is taken to be x, which is 5 normal call, but there is something different that you can do, you can call this function without actually passing any parameter, but it needs a parameter.

So, what it does when you do not pass a parameter, actual parameter, the function takes the default value 10 as given to be the value of a and goes ahead and does the computation. So, it is taken as 10, that is a basic concept of default parameter, very simple concept and makes life easier in different places.

(Refer Slide Time: 5:38)



Program 08.02: Function with 2 default parameters

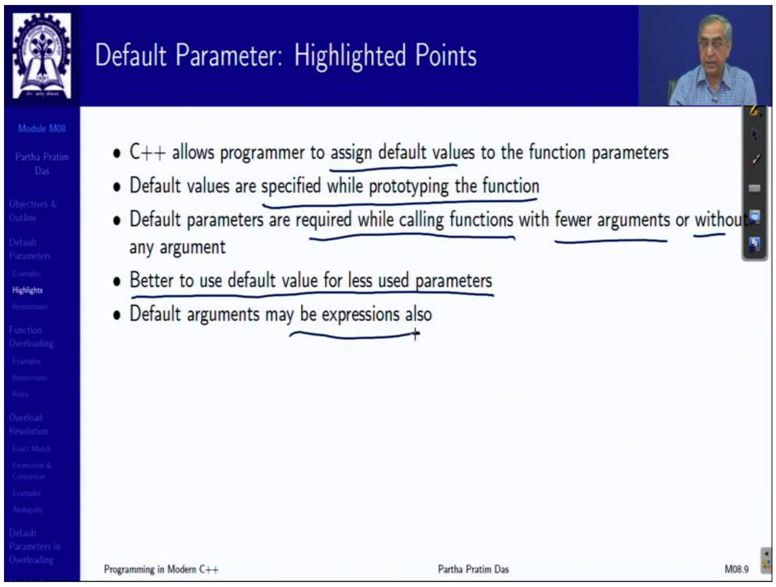
```
#include<iostream>
using namespace std;

int Add(int a = 10, int b = 20) {
    return (a + b);
}
int main() { int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x);    // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add();     // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}
----------
Sum = 11
Sum = 25
Sum = 30
```

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

Now, let us little bit go forward, let us take function with two default parameters. a defaulted for 10, b defaulted for 20, and I can call it with x y, 5 and 6. It will take as x 5 a 5, b 6. I can call it with just x, one parameter If I call it with one parameter whether it is a or it is b. That is decided from the left-hand side, if I pass one parameter, it is the when I pass parameters, it is always taken from the left.

So, x is copied to a. So, that is that becomes 5, but b is not given. So, which default value is taken which is 20. And, in the third case, if I do not pass any parameter, both A and B takes their default values. So, it always goes from the left, and as many parameters as provided will take the values of the actual parameters remaining must be default, and they will get their default values. That is the whole idea of the default parameter.

(Refer Slide Time: 7:00)



Default Parameter: Highlighted Points

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or withou
any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also

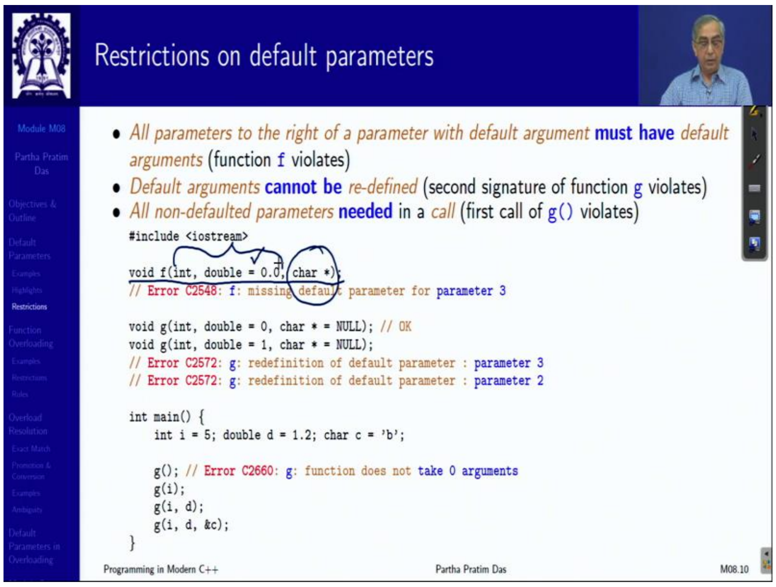
Programming in Modern C++ Partha Pratim Das M08.9

So, it allows programmers to assign default values to the function parameters, that is the basic idea. It is recommended that you provide the default value while you prototype the function, normally what is we write a prototype, and later on we put the function body so, we are actually writing the function header twice, once during prototype, and once during definition.

So, what is recommended is put the default values while you prototype, not during the definition, because the prototype is likely to be included in many other files as well. And therefore, it will be clear as to what the default values are. Default parameters are required while calling function with fewer arguments, or without any argument at all, you have already seen that.

As a design, you should arrange your parameters in a way so that the default values are used for less used parameter. If you use a default value for a frequently used parameter whose value can be different than every time, we will have to provide that anyway. And default arguments can be expressions as well. So, these are some of the basic stuff.

(Refer Slide Time: 8:22)



Restrictions on default parameters

- *All parameters to the right of a parameter with default argument* **must have** *default arguments* (function f violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function g violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of g() violates)

```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: g: function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
}
```

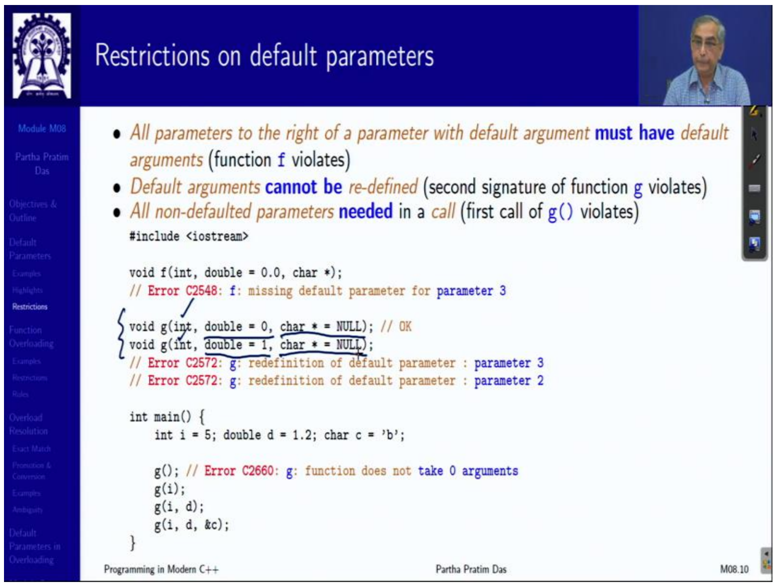
Programming in Modern C++ Partha Pratim Das M08.10

Now, there are some restrictions on the default. The restriction is since we are considering in the call, we are considering the parameters from left. As soon as you default one parameter, all parameters to it right must always be defaulted. So, you have non default parameters, then a parameter which is default and onwards everything has to be default parameters.

So, the entire sequence of parameters from left to right will be sequence of non-default parameters, followed by sequence of default parameters. That will always have to be the pattern, that is because otherwise, there is no way to resolve to know which parameter is what, because in C++, we always associate parameters by their position not by their name.

So, if we look at that, you can see here, I have defaulted this, but I have not defaulted this, this is not allowed. Because if I call it with 2 parameters, because I at-least need 2 parameters int and the char, char* to call this because only one is defaulted. But if I give 2 parameters, the compiler will also, always take them to be these 2 parameters because it is going from left. So that is a basic protocol we have to follow.

(Refer Slide Time: 9:52)



Restrictions on default parameters

- *All parameters to the right of a parameter with default argument* **must have** *default arguments* (function f violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function g violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of g() violates)

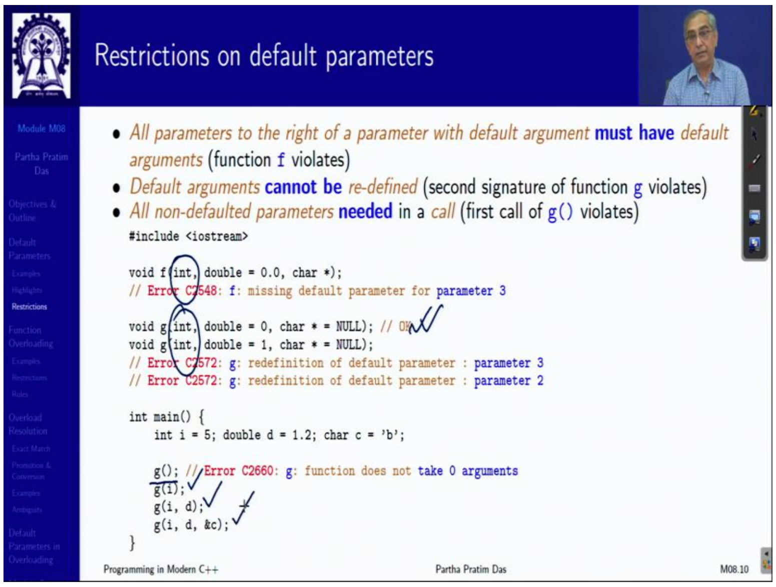
```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: g: function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
}
```

Programming in Modern C++ Partha Pratim Das M08.10

The second is, when you provide default values, you cannot multiply provide default values in two different signatures, two different places, or in two different signatures, even if the values are same. So, you can see here that I have two functions, header of g, int double char start, int is non-default, that is fine. In this I have made double defaulted as zero in this, I have double defaulted as 1, this is not allowed, in fact, this also is not allowed.

It will say that you have already provided a default value, you cannot redefine that. So, that will not be accepted in the compilation, because you do not know which of these headers will be used in compiling which invocation of the function in what file. And therefore, there may be confusion.

(Refer Slide Time: 10:55)



Restrictions on default parameters

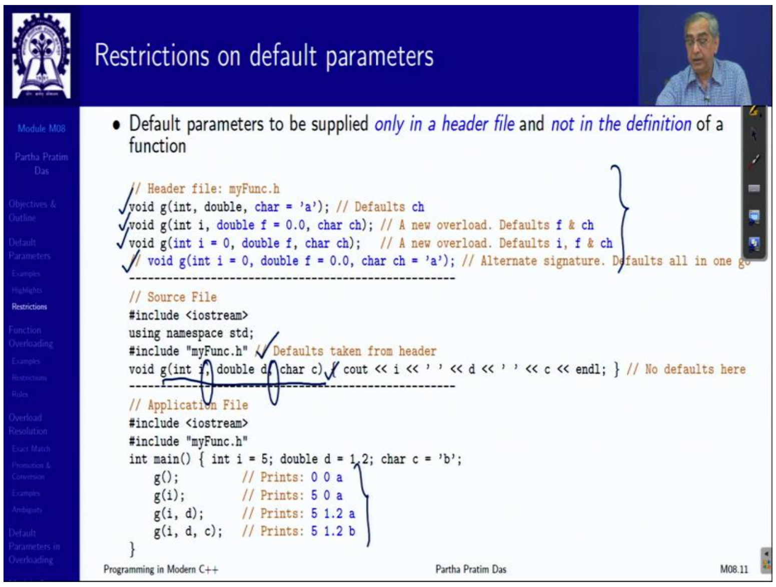
- *All parameters to the right of a parameter with default argument* **must have** *default arguments* (function f violates)
- *Default arguments* **cannot be** *re-defined* (second signature of function g violates)
- *All non-defaulted parameters* **needed** in a *call* (first call of g() violates)

```
#include <iostream>

void f(int, double = 0.0, char *);
// Error C2548: f: missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK
void g(int, double = 1, char * = NULL);
// Error C2572: g: redefinition of default parameter : parameter 3
// Error C2572: g: redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: g: function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
}
```

Programming in Modern C++ Partha Pratim Das M08.10

And naturally, while you call all non-default parameters must be provided. And only the default parameters can be skipped. So, in this context, if I call function g without a parameter, it is an error, because g always has error first parameter. I mean, I am talking in reference to this particular signature of g which is a valid one. But I can call it with one parameter, which will be int, two parameters int and double, three parameters int, double, and char start. So, this is these are basic rule of default parameters.

(Refer Slide Time: 11:36)



Restrictions on default parameters

- Default parameters to be supplied *only in a header file* and *not in the definition* of a function

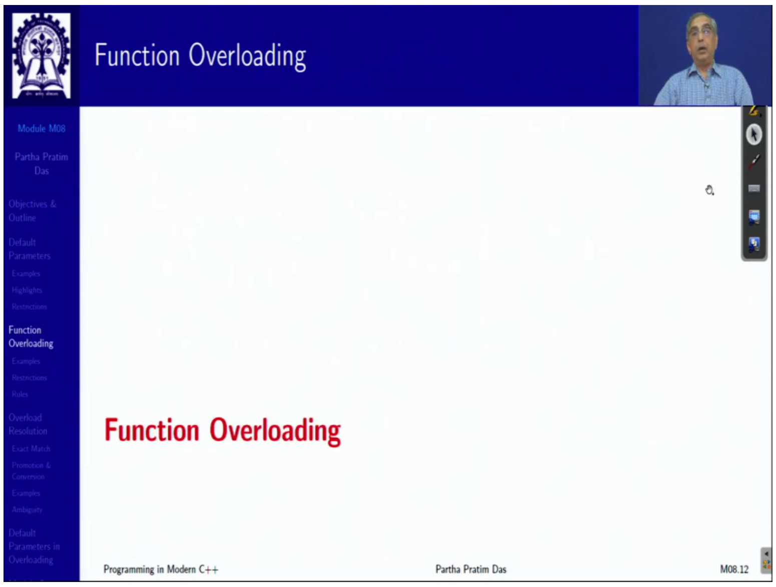
```
// Header file: myFunc.h
void g(int, double, char = 'a'); // Defaults ch
void g(int i, double f = 0.0, char ch); // A new overload. Defaults f & ch
void g(int i = 0, double f, char ch);   // A new overload. Defaults i, f & ch
// void g(int i = 0, double f = 0.0, char ch = 'a'); // Alternate signature. Defaults all in one g
----------------------------------------------------
// Source File
#include <iostream>
using namespace std;
#include "myFunc.h" // Defaults taken from header
void g(int i, double d, char c) { cout << i << ' ' << d << ' ' << c << endl; } // No defaults here
----------------------------------------------------
// Application File
#include <iostream>
#include "myFunc.h"
int main() { int i = 5; double d = 1.2; char c = 'b';
    g();          // Prints: 0 0 a
    g(i);         // Prints: 5 0 a
    g(i, d);      // Prints: 5 1.2 a
    g(i, d, c);   // Prints: 5 1.2 b
}
```

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

Another, which is kind of, I should say a programming practice is, this is not a language specification strictly but the programming practices, you always provide the default parameters in a header file. So, here I have three overloaded functions, we will see what overloads mean. Or I can have this signature of g with all three parameters defaulted, have them in the header file, and then I include this header file to define the function.

And, in this definition, I am not providing the default values. Having it in the header helps because any function, any file which includes the header for calling the function will get to know what the default values are. Because if we were given it here, then this source file may not be available to a user who is wanting to use the function, because this is a different file. Now, since there are three parameters, all of which are defaulted, there are four ways to call this function, which is you must have understood by now very easy, clearly.

(Refer Slide Time: 13:02)



Function Overloading

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

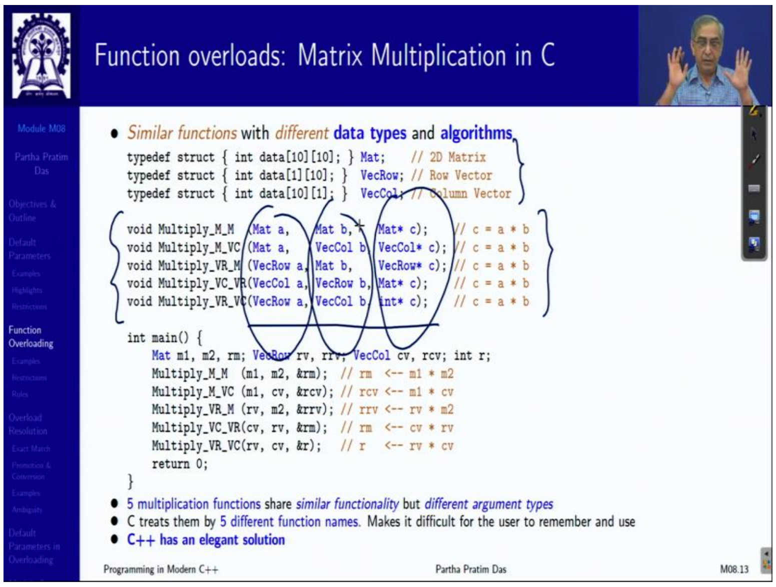
Examples

Ambiguity

Default Parameters in Overloading

Function Overloading

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



Function overloads: Matrix Multiplication in C

- *Similar functions* with *different* **data types** and **algorithms**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply_M_M  (Mat a,    Mat b,    Mat* c);    // c = a * b
void Multiply_M_VC (Mat a,    VecCol b, VecCol* c); // c = a * b
void Multiply_VR_M (VecRow a, Mat b,    VecRow* c); // c = a * b
void Multiply_VC_VR(VecCol a, VecRow b, Mat* c);    // c = a * b
void Multiply_VR_VC(VecRow a, VecCol b, int* c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_M  (m1, m2, &rm);  // rm  <-- m1 * m2
    Multiply_M_VC (m1, cv, &rcv); // rcv <-- m1 * cv
    Multiply_VR_M (rv, m2, &rrv); // rrv <-- rv * m2
    Multiply_VC_VR(cv, rv, &rm);  // rm  <-- cv * rv
    Multiply_VR_VC(rv, cv, &r);   // r   <-- rv * cv
    return 0;
}
```

- 5 multiplication functions share *similar functionality* but *different argument types*
- C treats them by 5 different function names. Makes it difficult for the user to remember and use
- **C++ has an elegant solution**

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

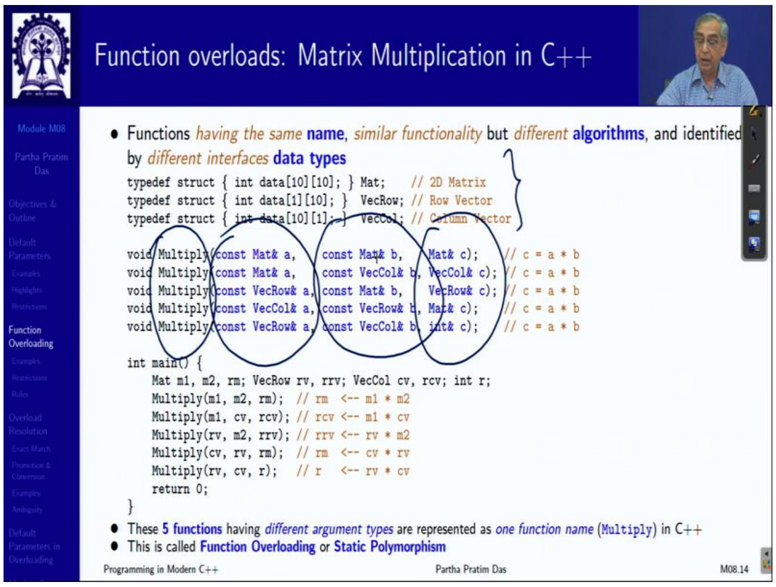
I loosely talked about, you know, I said these functions are overloaded, Let us understand what the function overloading is. Again, the context, the context is suppose there is different situations where, conceptually I am trying to do the same thing or very similar thing. But my data types are different, my algorithms are different, and therefore they have to be different functions. If I want to do this in C, it says function will have to get a unique name.

So let us say, I am talking about matrix multiplication, and I have three data types. One is a square matrix, one is a row vector, and one is a column vector. So, if I have these three data types, then there are 5 possible ways I might multiply, I can multiply a matrix with a matrix, matrix with a column vector, a row vector with a matrix, a column vector with a row vector, which will give me a matrix, or a row vector with the column vector, which is basically dot product, will give me a value.

So, these are you can check the types of parameters being passed, everything is here, these are passed by value, this is by address, because you want the output to come in the typical C style of stuff. But all of them are multiplying kind of using the same matrix multiplication rule, but because their data types are different, you have to give them different names.

So, as a programmer who is trying to use this, the programmer will have to remember, okay, multiplying matrix by a vector column is multiply underscore M underscore VC, very cumbersome. And we will see similar, lot of other problems also arise because of this.

(Refer Slide Time: 15:05)



Function overloads: Matrix Multiplication in C++

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

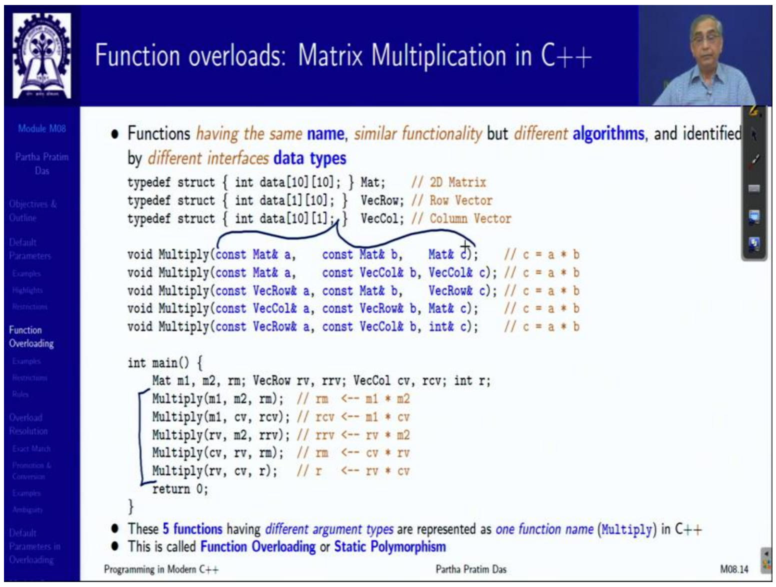
- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

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

So, what I would prefer is I have the same situation, the different data types, different algorithms, but I would like to call all of them by the same name, this is overloading. One name multiple meaning. So, you are overloading, if you consider the meaning to be a load on the name then you are overloading it giving multiple of them, which C does not allow.

C++ allows that and since we are in C++ now, you are, your inputs are all in terms of const reference. Whereas, your output is an in terms of non-const reference, because suddenly you do not want to copy matrices, and do not want the input matrices to change because of that.

(Refer Slide Time: 16:05)



Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

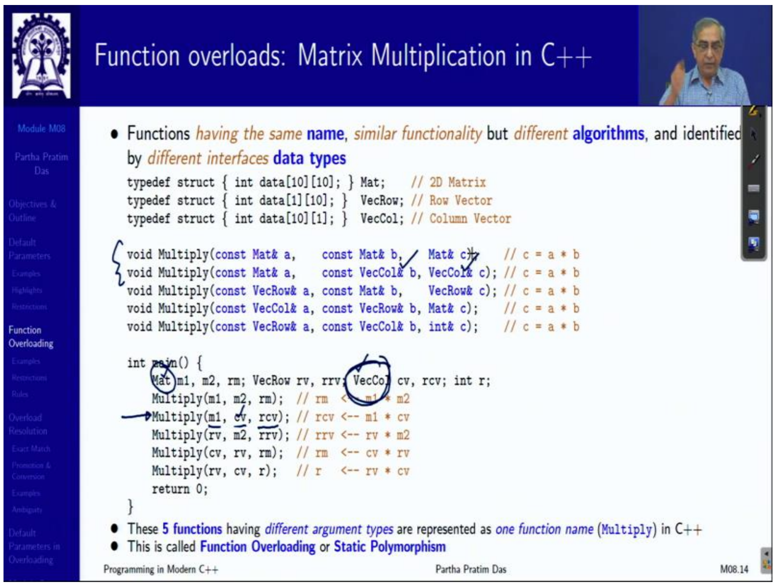
```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm); // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm); // rm  <-- cv * rv
    Multiply(rv, cv, r);  // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

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

And then now, the question that you will have is, naturally 5 functions will have 5 different codes. How will from in C we always knew that the function name is unique globally. So, from the name the system knows which function, what is the body that should be used. Here, how do you know that all of them the same types, all of them have the same name. So, that is where the parameter types become important.

(Refer Slide Time: 16:36)



Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

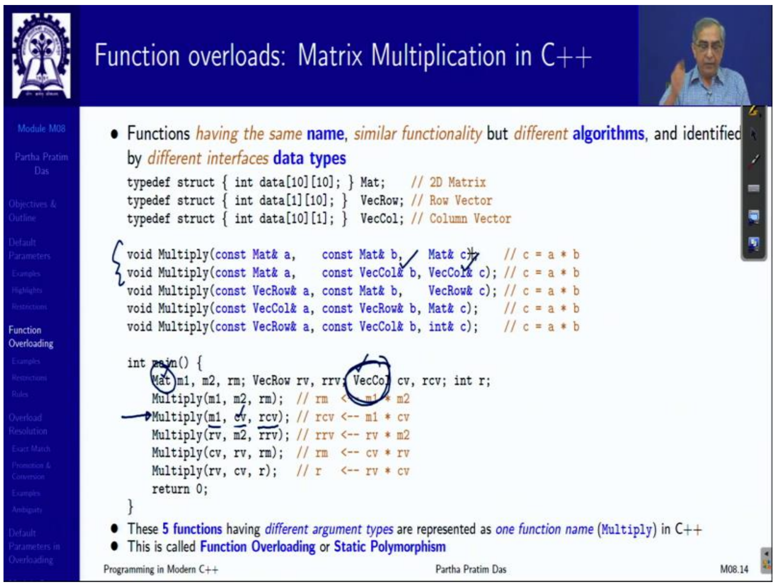
- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

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

So, if you let us say, if you look at say this second function multiply m1, cv, rcv, m1 is of type Mat, cv is of type column vector VecCol. rcv is also of type column vector. So, I will look for out of the 5 I look for that particular overloaded function, which matches this type. The first one is Mat, which means it could be any of these two.

Second one is VecCol, which means it has to be this, I already have got a unique function. And the third one has to match this. Otherwise, it is it is an error. And you can see that between any two functions. If you consider the types of parameters from left to right, they are distinct, there is something which is distinct. For example, just for better understanding.

(Refer Slide Time: 17:49)



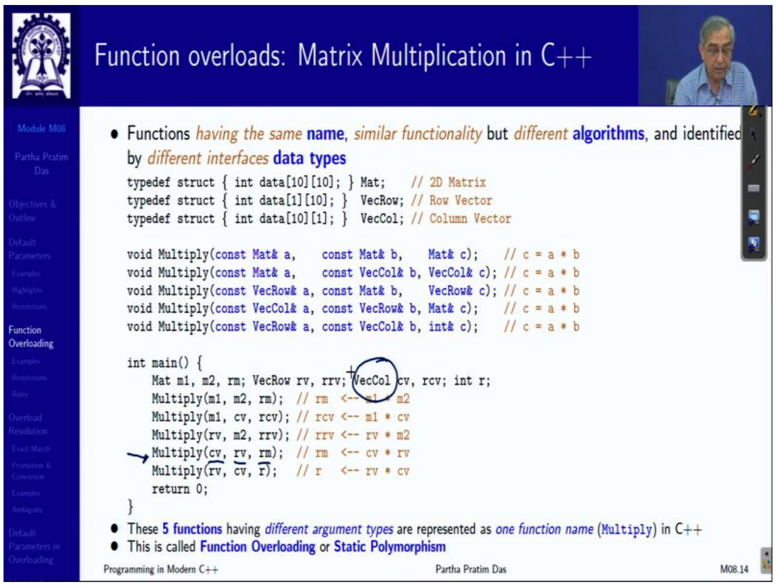
Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

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



Function overloads: Matrix Multiplication in C++

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

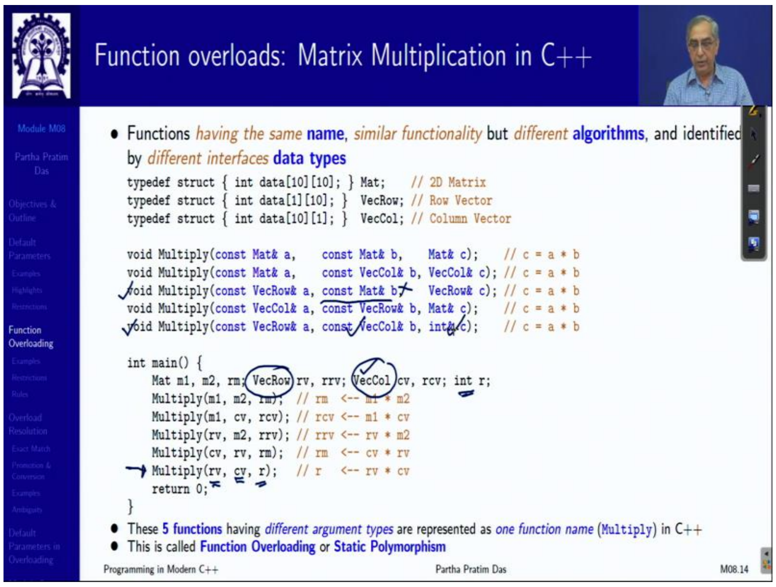
Ambiguity

Default Parameters in Overloading

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**



Function overloads: Matrix Multiplication in C++

- Functions *having the same* **name**, *similar functionality* but *different* **algorithms**, and identified by *different interfaces* **data types**

```
typedef struct { int data[10][10]; } Mat;    // 2D Matrix
typedef struct { int data[1][10]; }  VecRow; // Row Vector
typedef struct { int data[10][1]; }  VecCol; // Column Vector

void Multiply(const Mat& a,    const Mat& b,    Mat& c);    // c = a * b
void Multiply(const Mat& a,    const VecCol& b, VecCol& c); // c = a * b
void Multiply(const VecRow& a, const Mat& b,    VecRow& c); // c = a * b
void Multiply(const VecCol& a, const VecRow& b, Mat& c);    // c = a * b
void Multiply(const VecRow& a, const VecCol& b, int& c);    // c = a * b

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm);  // rm  <-- m1 * m2
    Multiply(m1, cv, rcv); // rcv <-- m1 * cv
    Multiply(rv, m2, rrv); // rrv <-- rv * m2
    Multiply(cv, rv, rm);  // rm  <-- cv * rv
    Multiply(rv, cv, r);   // r   <-- rv * cv
    return 0;
}
```

- These **5 functions** having *different argument types* are represented as *one function name* (`Multiply`) in C++
- This is called **Function Overloading** or **Static Polymorphism**

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

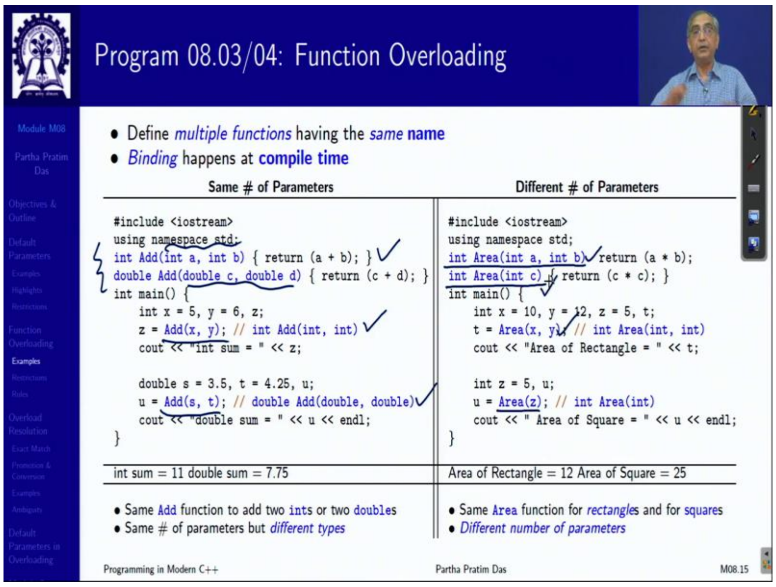
Let us say, let us take so the fourth one, cv, rv, rm, cv is VecCol. So, this does not help, because this immediately gets you here, there is only one function which has first parameters as VecCol. So let us look at something which can be more confusing. So let us look at five. The first is rv, rv is VecRow, if it is VecRow, then it can be either this function, or this function whose first parameter is VecRow.

Then the second parameter is cv, which is VecCol. So, in this the second parameter is Mat, which cannot be, so it has to be this, and the third parameter is r, which is int, and it matches, so is the fifth function. So, what you are doing now is in C, globally by their name you were deciding the function, we said this is binding with the name and the actual body of the function we said this is a binding.

The binding was happening just by the name, now the binding is happening by the name as well as the types of parameters, this is known as static polymorphism, or function overloading. Why static polymorphism, polymorphism means poly means many, morph means change, forms. So, polymorphism refers to the fact that when you have something which is in multiple forms, but common, so this is these are polymorphic.

And why is it static? Because the compiler itself can find out what is the right function to bind to. So that is, that is the basic point you make here.

(Refer Slide Time: 19:51)



Program 08.03/04: Function Overloading

Module M08

Partha Pratim Das

Objectives & Outline
Default Parameters
Examples
Highlights
Restrictions
Function Overloading
Examples
Restrictions
Rules
Overload Resolution
Exact Match
Promotion & Conversion
Examples
Ambiguity
Default Parameters in Overloading

- Define *multiple functions* having the *same* **name**
- *Binding* happens at **compile time**

Same # of Parameters

```
#include <iostream>
using namespace std;
int Add(int a, int b) { return (a + b); }
double Add(double c, double d) { return (c + d); }
int main() {
    int x = 5, y = 6, z;
    z = Add(x, y); // int Add(int, int)
    cout << "int sum = " << z;

    double s = 3.5, t = 4.25, u;
    u = Add(s, t); // double Add(double, double)
    cout << "double sum = " << u << endl;
}
```

```
int sum = 11 double sum = 7.75
```

- Same Add function to add two ints or two doubles
- Same # of parameters but *different types*

Different # of Parameters

```
#include <iostream>
using namespace std;
int Area(int a, int b) { return (a * b);
int Area(int c) { return (c * c); }
int main() {
    int x = 10, y = 12, z = 5, t;
    t = Area(x, y); // int Area(int, int)
    cout << "Area of Rectangle = " << t;

    int z = 5, u;
    u = Area(z); // int Area(int)
    cout << " Area of Square = " << u << endl;
}
```

```
Area of Rectangle = 12 Area of Square = 25
```

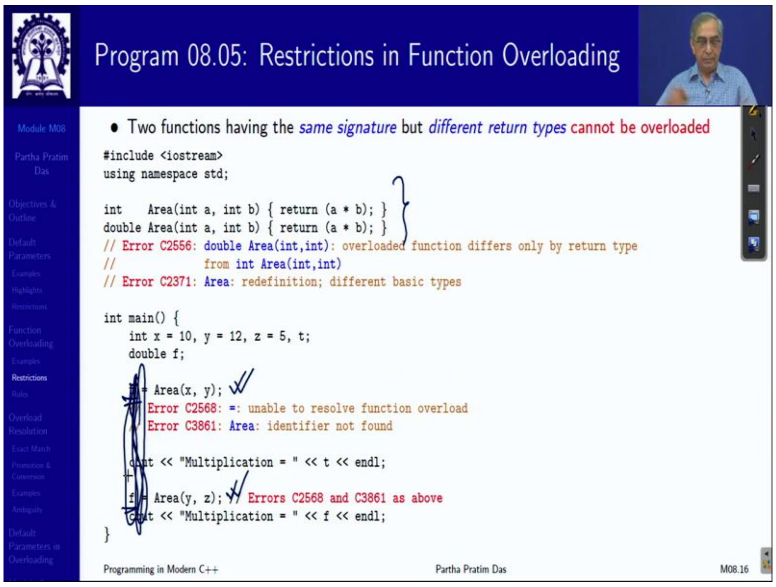
- Same Area function for *rectangles* and for squares
- *Different number of parameters*

Programming in Modern C++ Partha Pratim Das M08.15

So, here quickly we will have examples. So, I have two functions both of which are double both of which has 2 parameters, but one has two ints, and one has two doubles. So, when I call it with add x y, which are both int, this function will be called the first one, when I do both with double, the second function will be called by the type it will be able to dissolve.

Here I have an example of Area function where one has 2 parameters, and one has 1 parameter. So, naturally if I one is basically computing for rectangle other is for square. So, once I do x y, it will call the first function, if I do this, it will bind to the second function, that is the whole idea of the polymorphic static polymorphism or function overloading.

(Refer Slide Time: 20:46)



Program 08.05: Restrictions in Function Overloading

- Two functions having the *same signature* but *different return types* cannot be overloaded

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

int    Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2556: double Area(int,int): overloaded function differs only by return type
//              from int Area(int,int)
// Error C2371: Area: redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 5, t;
    double f;

    t = Area(x, y);
    // Error C2568: =: unable to resolve function overload
    // Error C3861: Area: identifier not found

    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2568 and C3861 as above
    cout << "Multiplication = " << f << endl;
}
```

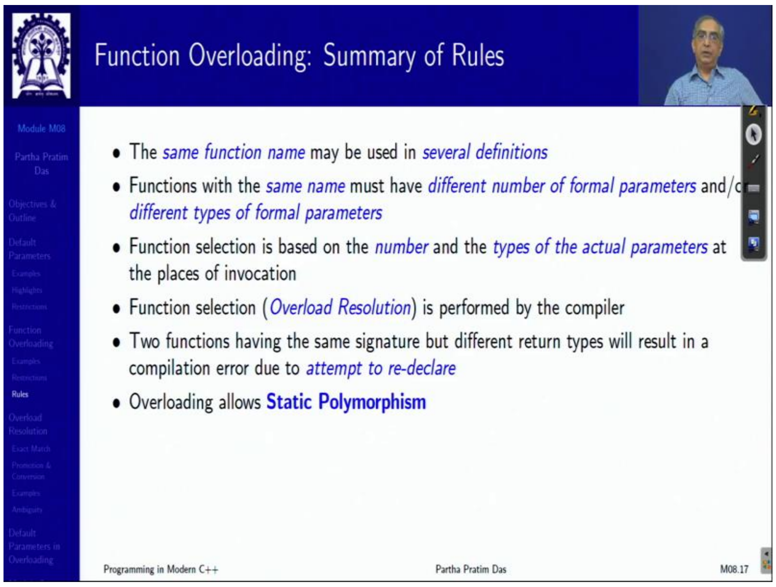
Programming in Modern C++ Partha Pratim Das M08.16

But, however, it the resolution must be possible by the parameters only, you cannot resolve based on the return type. Two functions which are indistinguishable, two functions having the same name of course, which are indistinguishable by their parameter types cannot be decided by their return type.

Very simply, because when you call the function, you only pass the parameters, there is no way to tell what kind of return value you are expecting, it is whatever you get. So, that is not possible. So, here in this example, we have just illustrated that, that between these two Area functions, the resolution is not possible.

So, if I call it like this, or I call it like this, whether I am taking the value in int or I am taking the value in double is not something. So, this part is not something which the system knows. Because, I may not just copy the value, I can just call the function also that is valid. So, but there the parameter types are identical. So, this is not permitted.

(Refer Slide Time: 22:01)



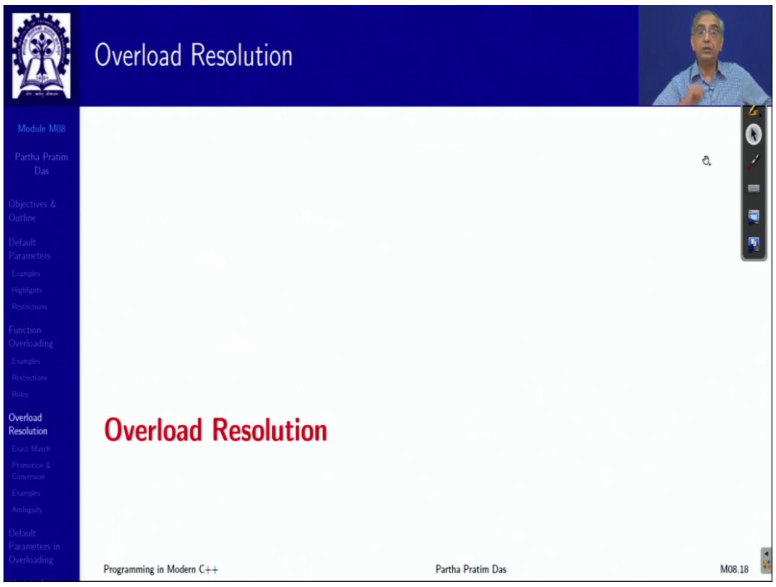
Function Overloading: Summary of Rules

- The *same function name* may be used in *several definitions*
- Functions with the *same name* must have *different number of formal parameters* and/o
  different types of formal parameters
- Function selection is based on the *number* and the *types of the actual parameters* at the places of invocation
- Function selection (*Overload Resolution*) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to *attempt to re-declare*
- Overloading allows **Static Polymorphism**

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

So, the summary of the rules same function name, several definitions, different number of formal parameters are the same number of formal parameters with different types, and the selection is based on the number of the actual parameters, and if there are multiple such, like I just showed in case of matrix multiplication, there are 5, and how we are able to choose the right one for binding from the 5. So, that process is the process of resolving, how you decide on the resolution of the actual function that needs to be bound.

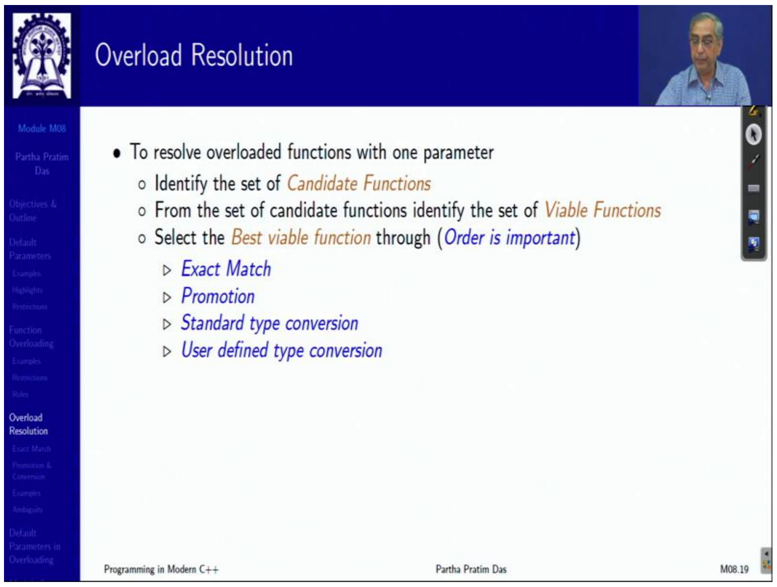
(Refer Slide Time: 22:39)



Overload Resolution

Overload Resolution

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



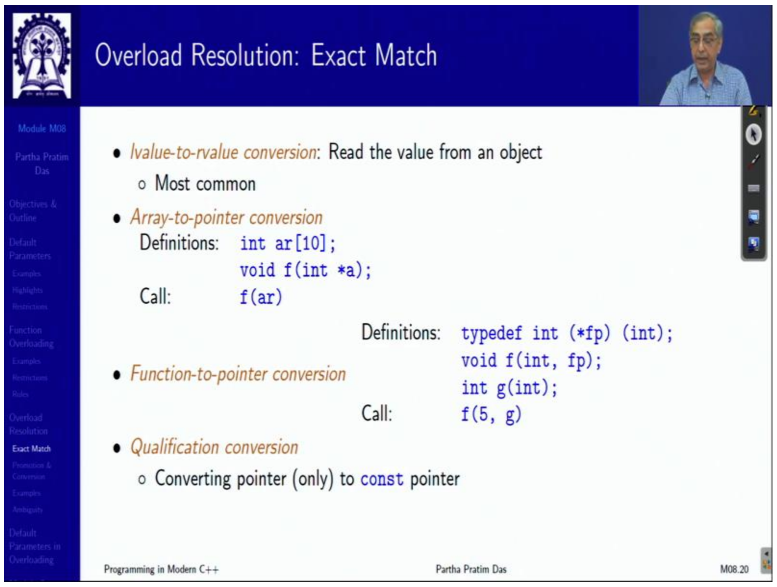
Overload Resolution

- To resolve overloaded functions with one parameter
  - Identify the set of *Candidate Functions*
  - From the set of candidate functions identify the set of *Viable Functions*
  - Select the *Best viable function* through (*Order is important*)
    - *Exact Match*
    - *Promotion*
    - *Standard type conversion*
    - *User defined type conversion*

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

So, this is known as overload resolution. Now, overload resolution goes through certain phases, first is you identify a set of Candidate Functions, then you identify the set of Viable Functions from the Candidate Functions, and finally, you choose the Best viable function, according to this set of steps, what is best by exact match? If not by promotion, if not by standard type conversion, if not by user defined type conversion. Let us go through that.

(Refer Slide Time: 23:15)



Overload Resolution: Exact Match

- *lvalue-to-rvalue conversion*: Read the value from an object
  - Most common
- *Array-to-pointer conversion*

```
Definitions: int ar[10];
             void f(int *a);
Call:        f(ar)
```

- *Function-to-pointer conversion*

```
Definitions: typedef int (*fp) (int);
             void f(int, fp);
             int g(int);
Call:        f(5, g)
```

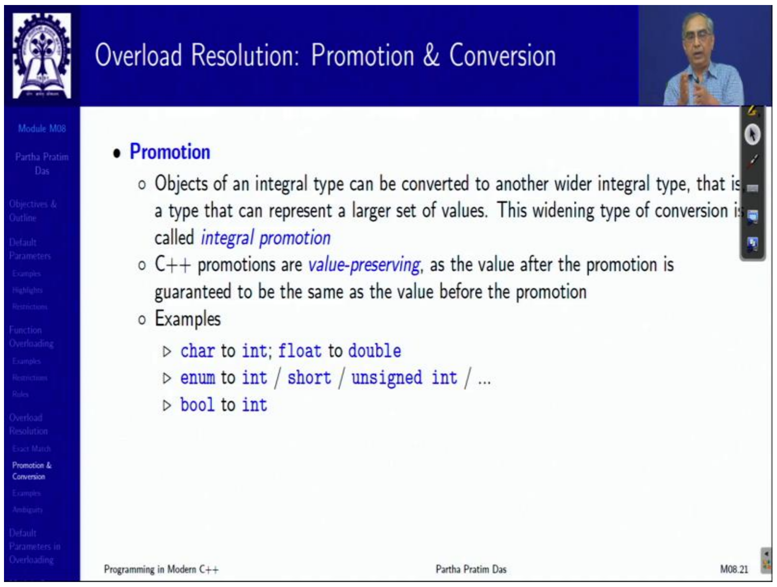
- *Qualification conversion*
  - Converting pointer (only) to `const` pointer

Programming in Modern C++ Partha Pratim Das M08.20

So, what is exact match, when 2 parameters exactly match. Now, they will exactly match if they are exactly identical type of course, but certain things are considered loosely as exact match. Like, lvalue-to- rvalue conversion is the exact match, if you take say variable a, then in the context of the left-hand side, it is an lvalue address, and the context of the right hand side is an rvalue, value. So, any variable can be used by taking its value, that is a match.

Arrays and pointers conversion are an exact match. So, I can define an array, I can have a function which takes a pointer of type integer and pass an integer array to that, I can have a function pointer and pass a function to that without making it a pointer, I can convert pointers only to constant pointer, this is these are all exact match conversions that you can be done.

(Refer Slide Time: 24:23)



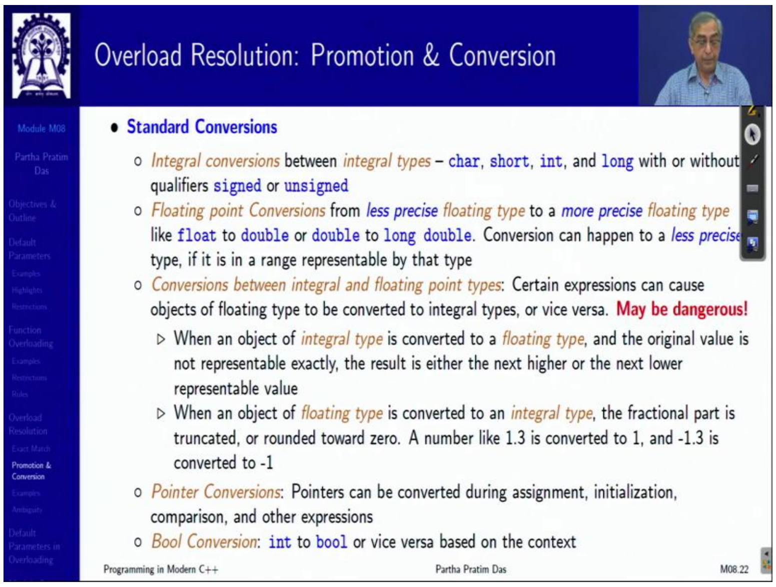
Overload Resolution: Promotion & Conversion

- **Promotion**
  - Objects of an integral type can be converted to another wider integral type, that is a type that can represent a larger set of values. This widening type of conversion is called *integral promotion*
  - C++ promotions are *value-preserving*, as the value after the promotion is guaranteed to be the same as the value before the promotion
  - Examples
    - `char` to `int`; `float` to `double`
    - `enum` to `int` / `short` / `unsigned int` / ...
    - `bool` to `int`

Programming in Modern C++ Partha Pratim Das M08.21

Now, the second that I can do is promotion, Promotion is when I have a value of a smaller type smaller in in terms of size, and it is typically apply, promotion typically applies to integral types only, then I can always represent it with a bigger type. So, I have a char, I can represent it by short int, or I have a int, I can represent it by long, I have a float, I can represent it by double, and so on. Which is quite obvious.

(Refer Slide Time: 24:58)



Overload Resolution: Promotion & Conversion

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

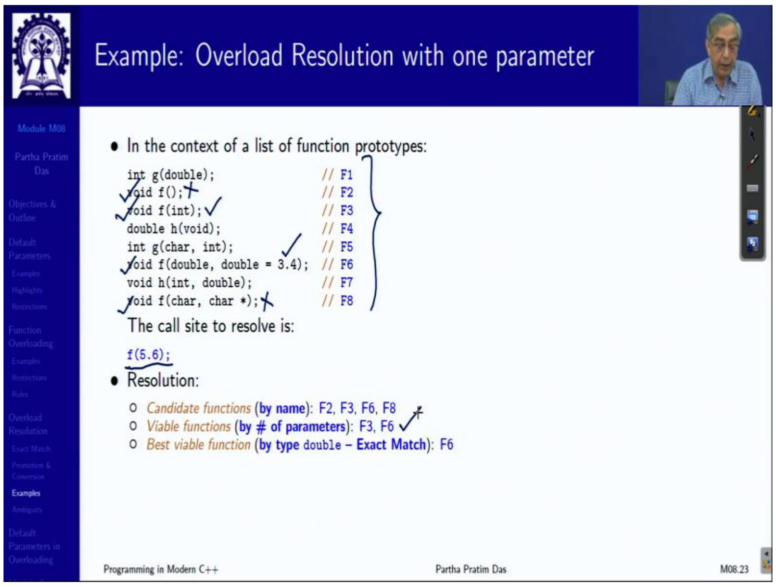
- **Standard Conversions**
  - *Integral conversions* between *integral types* – `char`, `short`, `int`, and `long` with or without qualifiers `signed` or `unsigned`
  - *Floating point Conversions* from *less precise floating type* to a *more precise floating type* like `float` to `double` or `double` to `long double`. Conversion can happen to a *less precise* type, if it is in a range representable by that type
  - *Conversions between integral and floating point types*: Certain expressions can cause objects of floating type to be converted to integral types, or vice versa. **May be dangerous!**
    - When an object of *integral type* is converted to a *floating type*, and the original value is not representable exactly, the result is either the next higher or the next lower representable value
    - When an object of *floating type* is converted to an *integral type*, the fractional part is truncated, or rounded toward zero. A number like 1.3 is converted to 1, and -1.3 is converted to -1
  - *Pointer Conversions*: Pointers can be converted during assignment, initialization, comparison, and other expressions
  - *Bool Conversion*: `int` to `bool` or vice versa based on the context

Programming in Modern C++ Partha Pratim Das M08.22

Then there are certain conversions which are possible, for example, I can convert between integral types with or without the qualifier of signed, I can convert between floating point types, float being converted to double, double being converted to long double, and so on.

So, when I do that, I can be from less precise like float to more precise like double, that will have no issue. But, if I want to do the reverse, then not always it may be possible, because the more precise will have a bigger chunk of possible numbers, I can convert pointers, I can do conversion of bool integer.

(Refer Slide Time: 25:45)



Example: Overload Resolution with one parameter

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

- In the context of a list of function prototypes:

```
int g(double);                    // F1
void f(); ✗                       // F2
void f(int); ✓                    // F3
double h(void);                   // F4
int g(char, int);         ✓       // F5
void f(double, double = 3.4);     // F6
void h(int, double);              // F7
void f(char, char *); ✗           // F8
```

The call site to resolve is:

`f(5.6);`

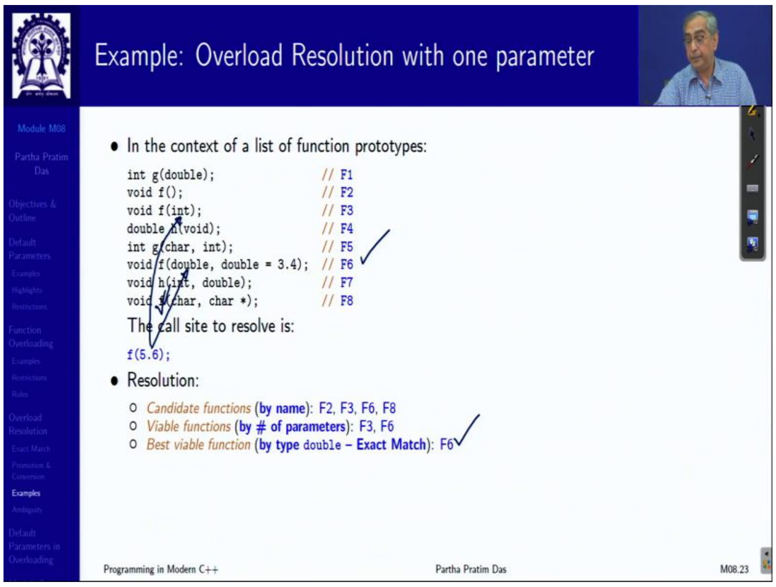
- Resolution:
  - *Candidate functions* (**by name**): F2, F3, F6, F8
  - *Viable functions* (**by # of parameters**): F3, F6 ✓
  - *Best viable function* (**by type `double` – Exact Match**): F6

Programming in Modern C++ Partha Pratim Das M08.23

So, an example to illustrate how the resolution will happen. Let us say I have a function called f(5.6). So, it takes one parameter, 5.6 is a literal of double type, so it expects a double type. And these are the functions available to you. So, first what you do you create the set of candidate functions, which are functions having the same name. So, f, f, f, f, these four are candidate functions f 2, f 3. f 6, f 8.

Out of these candidate functions, you decide on the set of Viable functions, Viable functions are those which match the number of parameters, that has to be possible. So, here there is one parameter. So, this goes out, this takes 0 parameters, this also goes out, because it needs 2 parameters, this will be valid, it has one parameter, this will also be valid because though it takes 2 parameters, the second parameter can be defaulted. So, I can call it with one parameter. So, what remains from this is f 3, and f 6.

(Refer Slide Time: 26:59)



Example: Overload Resolution with one parameter

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

- In the context of a list of function prototypes:

```
int g(double);                  // F1
void f();                       // F2
void f(int);                    // F3
double h(void);                 // F4
int g(char, int);               // F5
void f(double, double = 3.4);   // F6
void h(int, double);            // F7
void f(char, char *);           // F8
```

The call site to resolve is:

```
f(5.6);
```

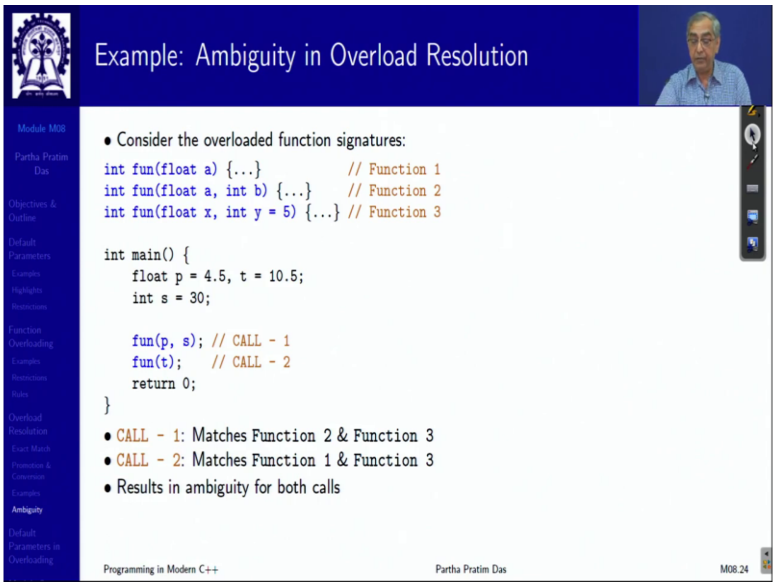
- Resolution:
  - *Candidate functions* (**by name**): F2, F3, F6, F8
  - *Viable functions* (**by # of parameters**): F3, F6
  - *Best viable function* (**by type** `double` **– Exact Match**): F6

Programming in Modern C++ Partha Pratim Das M08.23

So, if I want to call, then actually 5.6 the actual parameter value, will have to go to here for f 6 and have to go to here for f 3. The first one is exact match, double to double, and the second one is floating point to integer conversion. So, what did they say, exact match promotion, conversion, user defined conversion.

So, the top most, the first is exact match. So, I have, we have got an exact match here. Therefore, f 6 is my resolution, you can try it out by trying to run this program, we will see that it binds with this function, whatever is there in that function, is what will be executed.

(Refer Slide Time: 27:59)



Example: Ambiguity in Overload Resolution

- Consider the overloaded function signatures:

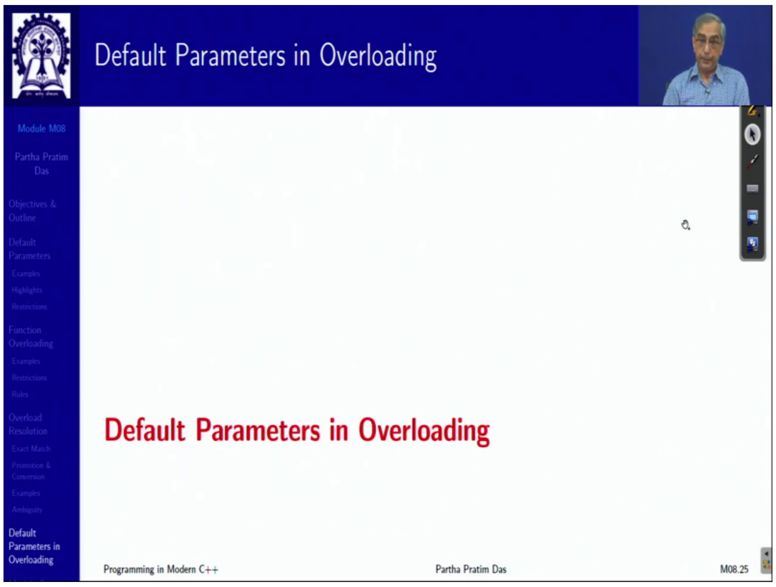
```
int fun(float a) {...}            // Function 1
int fun(float a, int b) {...}     // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity for both calls

Programming in Modern C++ Partha Pratim Das M08.24

Now, there could be ambiguity also, for example, here, as you can see that three candidate functions, and I am calling it with 2 parameters in CALL – 1. So, there are two viable functions, and there is no way to resolve them because they have the same types. Similarly, if I call it in CALL – 2, with one parameter, there also the variable there are two viable functions, Function 1, and Function 3. So, I cannot resolve them. So, in such cases, the compiler will say that I am confused, and I cannot resolve this.

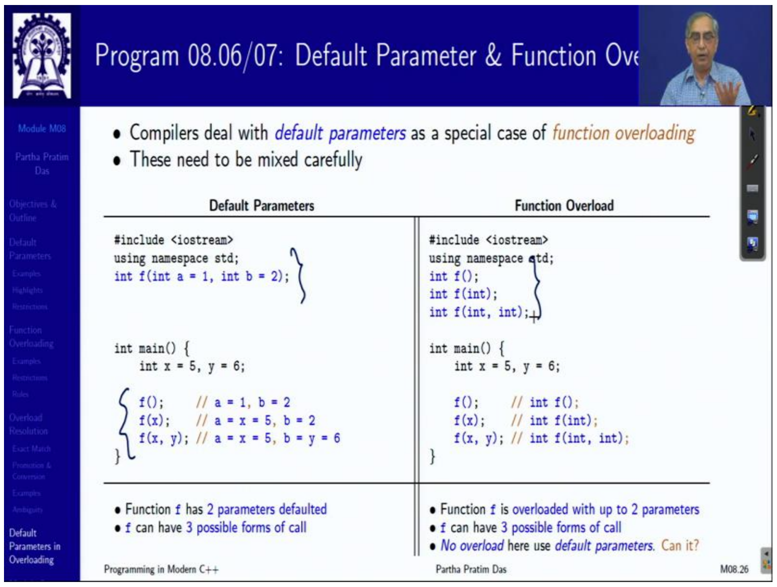
(Refer Slide Time: 28:35)



Default Parameters in Overloading

Default Parameters in Overloading

Programming in Modern C++ Partha Pratim Das M08.25



Program 08.06/07: Default Parameter & Function Ove

- Compilers deal with *default parameters* as a special case of *function overloading*
- These need to be mixed carefully

Default Parameters	Function Overload

`#include <iostream>`
`using namespace std;`
`int f(int a = 1, int b = 2);`
 	
`int main() {`
`    int x = 5, y = 6;`
 	
`    f();     // a = 1, b = 2`
`    f(x);    // a = x = 5, b = 2`
`    f(x, y); // a = x = 5, b = y = 6`
`}`
	`#include <iostream>`
	`using namespace std;`
	`int f();`
	`int f(int);`
	`int f(int, int);`
	
	`int main() {`
	`    int x = 5, y = 6;`
	
	`    f();     // int f();`
	`    f(x);    // int f(int);`
	`    f(x, y); // int f(int, int);`
	`}`

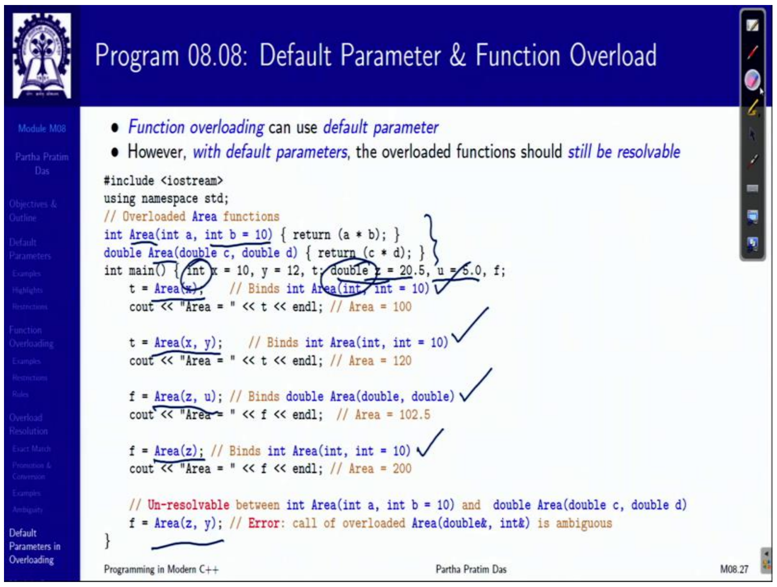
• Function `f` has 2 parameters defaulted • `f` can have 3 possible forms of call	• Function `f` is overloaded with up to 2 parameters • `f` can have 3 possible forms of call • *No overload* here use *default parameters*. Can it?

Programming in Modern C++ Partha Pratim Das M08.26

Now, naturally, you can have default parameters. Also, with function overloading, or rather, the default parameter mechanism itself is a function overloading. Because if you look at this, then there are three possible ways to call it, which means that if you had three different overloaded functions, you would have had the same behaviour.

So, default parameter is a semantically different entity, but it actually tells us, or tells the compiler that you are doing an overloading in terms of the number of parameters and their types. So, it is the default parameters are handled in the same way as the function overloading is handled.

(Refer Slide Time: 29:27)



Program 08.08: Default Parameter & Function Overload

- *Function overloading* can use *default parameter*
- However, *with default parameters*, the overloaded functions should *still be resolvable*

```
#include <iostream>
using namespace std;
// Overloaded Area functions
int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }
int main() { int x = 10, y = 12, t; double z = 20.5, u = 5.0, f;
    t = Area(x);      // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 100

    t = Area(x, y);     // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // Area = 120

    f = Area(z, u); // Binds double Area(double, double)
    cout << "Area = " << f << endl;  // Area = 102.5

    f = Area(z); // Binds int Area(int, int = 10)
    cout << "Area = " << f << endl; // Area = 200

    // Un-resolvable between int Area(int a, int b = 10) and  double Area(double c, double d)
    f = Area(z, y); // Error: call of overloaded Area(double&, int&) is ambiguous
}
```

Programming in Modern C++ Partha Pratim Das M08.27

Now, you can have a mix of them as well. For example, you can have function overloading with default parameters like in what we have here. Is overloaded function Area, this has one default parameter, the other has none. So, there are multiple ways. So, what is Area(x)? x is int, it is one parameter. So, it cathed to be the first function and it matches.

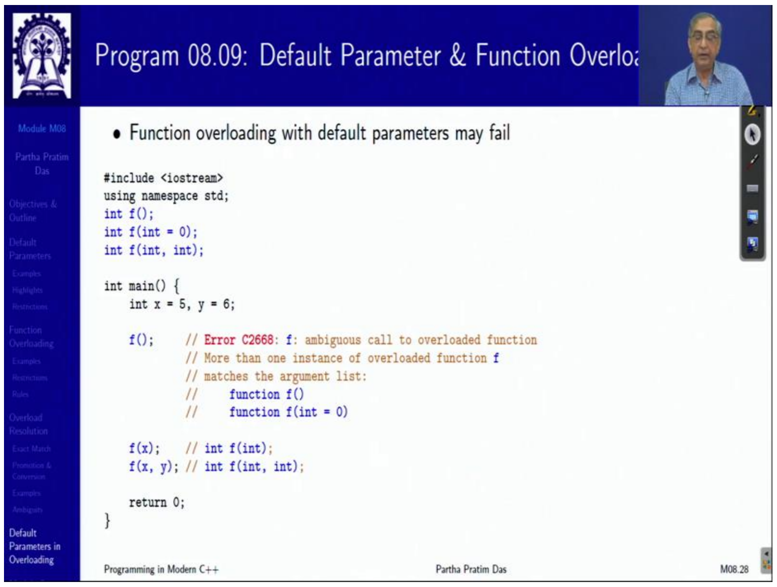
So, this is an easy case, bind says. What is Area(x, y)? 2 parameters so, both functions are actually viable, but the first one has the exact match. So, this is what you get. If we do Area(z, w), z is double, w is double. I am sorry z u, z is double w, u also is double. So, the second function out of the two viable function, the second function has a perfect match, as an exact match.

But suppose you call it with Area(z), where z is double, then naturally, there is only one viable function, which is the first one, and therefore, it will still be able to bind, but it will go through conversion from double to int. Now, suppose if you call it with Area(z, y), where z is double, and y is int, double an int. 2 parameters, so both of them are our viable.

And you see, if you want to find the exact match in the first, then the first parameter needs a conversion, second parameter is an exact match. If you take the second, the first parameter is a match, exact match, second parameter needs a conversion. So, the weightage of these two are same to the compiler.

So, the compiler will again get confused and say that I cannot resolve, and we cannot have this Area z y, this invocation is un-resolvable as a overloaded function.

(Refer Slide Time: 31:42)



Program 08.09: Default Parameter & Function Overloa

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

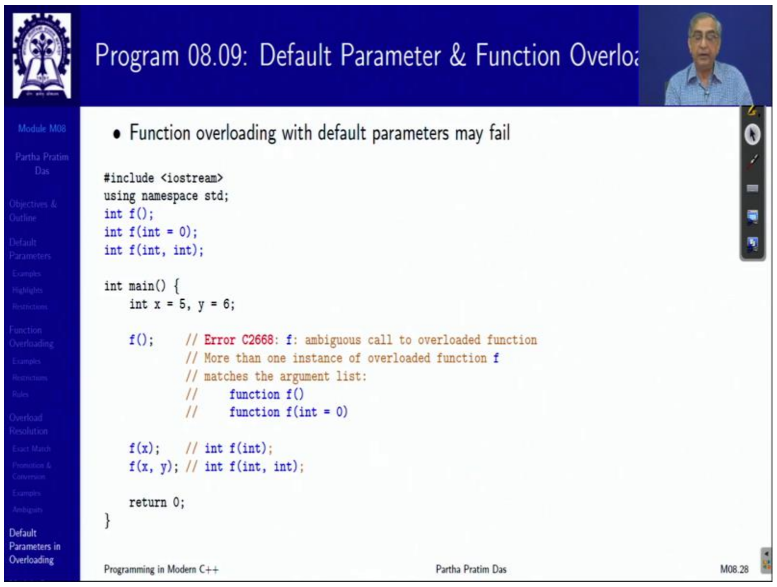
- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();     // Error C2668: f: ambiguous call to overloaded function
             // More than one instance of overloaded function f
             // matches the argument list:
             //     function f()
             //     function f(int = 0)

    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

Programming in Modern C++ Partha Pratim Das M08.28

Similarly, you will have to make sure that when you have default parameters, and overloading, default parameters and overloading together must be resolvable. For example, here a function call without parameter is not resolvable, because between the first and the second, it can be either of them. Whereas with one parameter or 2 parameters, this is resolved.

(Refer Slide Time: 32:06)



Program 08.09: Default Parameter & Function Overloa

Module M08

Partha Pratim Das

Objectives & Outline

Default Parameters

Examples

Highlights

Restrictions

Function Overloading

Examples

Restrictions

Rules

Overload Resolution

Exact Match

Promotion & Conversion

Examples

Ambiguity

Default Parameters in Overloading

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();     // Error C2668: f: ambiguous call to overloaded function
             // More than one instance of overloaded function f
             // matches the argument list:
             //     function f()
             //     function f(int = 0)

    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

Programming in Modern C++ Partha Pratim Das M08.28

So, to summarize, we have introduced the notion of default parameters, and discuss several examples, and particularly the necessity of function overloading and how does it bring in the first major type of polymorphism known as a static polymorphism, which will play a very critical role in different aspects of modern C++ as we go forward. Thank you very much for your attention, and we will meet in the next module.