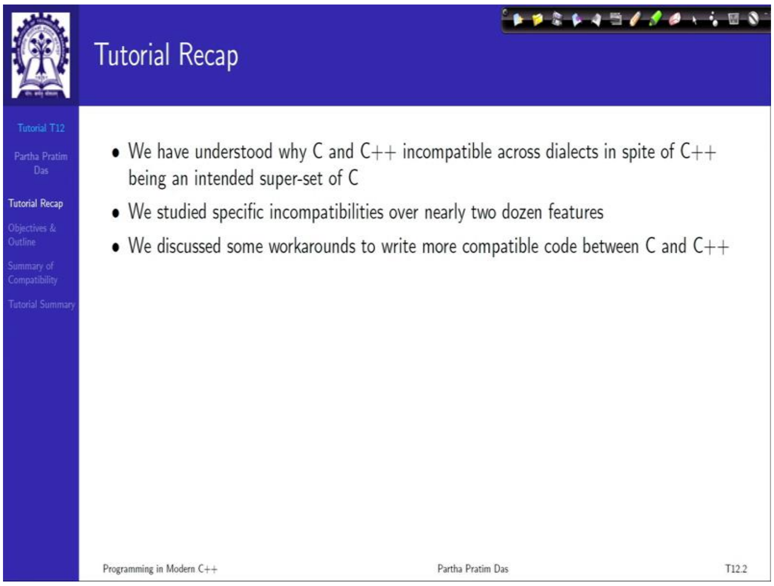
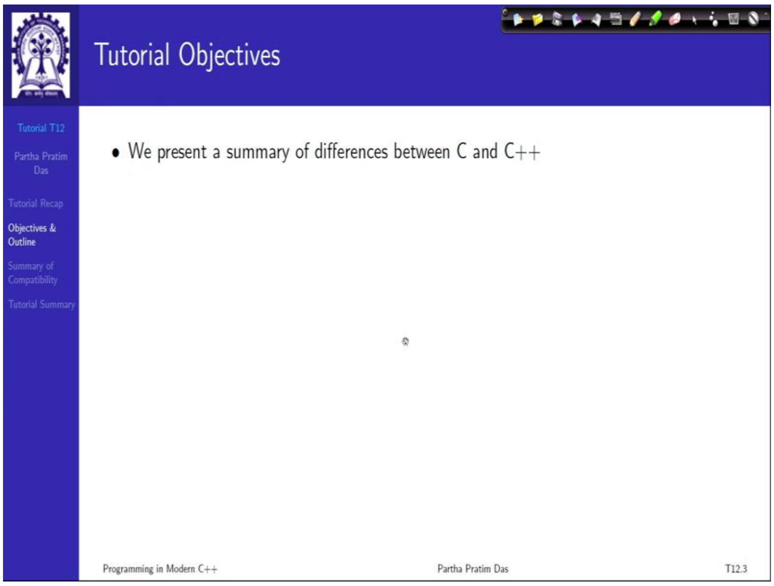
Programming in Modern C++
Professor Parthe Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Tutorial 12
Compatibility of C and C++: Part2: Summary

Welcome to Programming in Modern C++, we have been discussing about compatibility of C and C++ languages as a part of this tutorial.

(Refer Slide Time: 0:47)

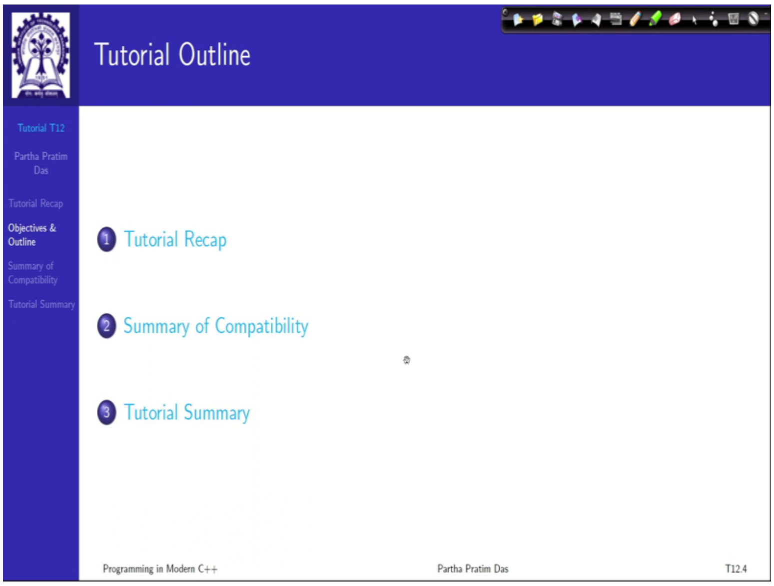


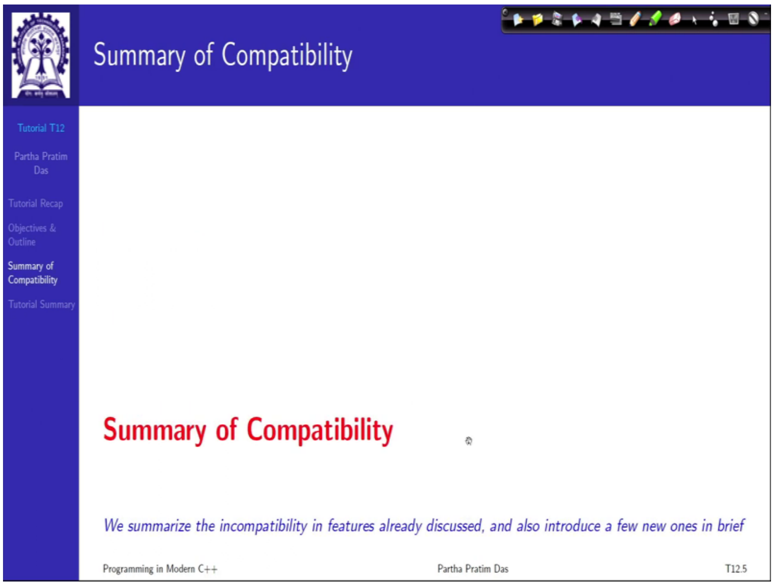


Now, we are in tutorial 12 which is the second part of the summarization part for tutorial 11. In the tutorial 11, we have understood why C and C++ compatibility across different dialects are important and why at all incompatibilities exist in spite of we loosely saying that C is a subset of C++ and so on.

So, we studied some specific incompatibilities for about 2 dozen features and discussed workarounds for some of them, we will summarize the entire collection here in this summary of incompatibilities between C and C++.

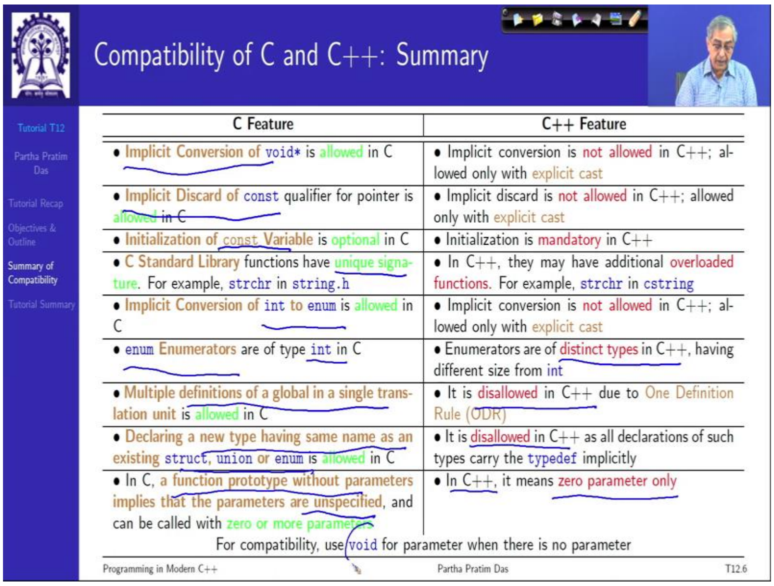
(Refer Slide Time: 01:36)





So, this is the summary is the only thing to discuss for this.

(Refer Slide Time: 01:47)



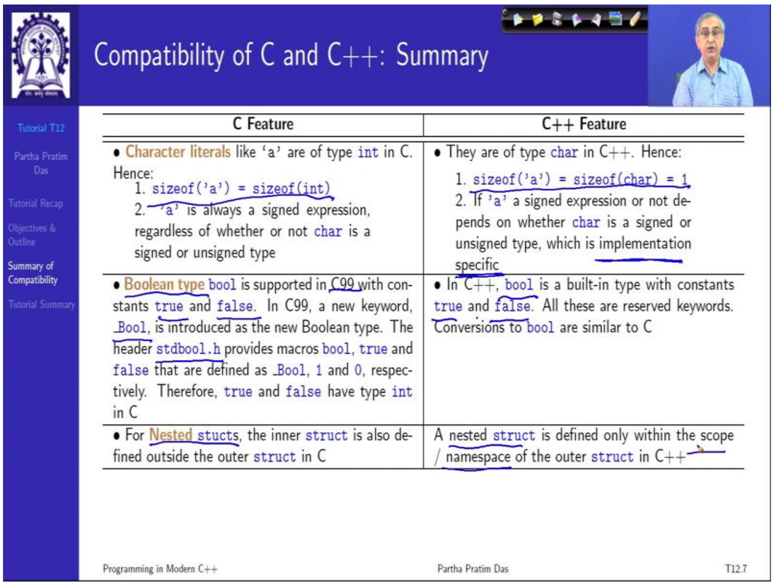
So, we start by making reference to what we have already discussed in tutorial 11. That is implicit conversion of void* is allowed in C and it is not allowed in C++. It is allowed only with explicit casting. Similarly, implicit discard of const also is allowed in C. But in C++, this is not allowed this will be permitted only if explicit casting is use. Initialization of const variable is optional in C, but it is mandatory for C++. C standard library have unique signatures for every function because every function in C is global.

Whereas in C++ there may be additional overloaded functions for the same C standard library component that C++ wraps, for example, we have seen strchr in C string has more than one signature. Implicit conversion from int to enum type is allowed in C and it is not allowed in C++ again, you need an explicit cast, enum Enumerators the different enumerated values are of type int in C, whereas they are have distinct types in C++ having a size which is different from int.

So, we will have to be careful in all these cases, particularly when we are trying to build a C program as if it is a C++ program by the C++ compiler and so on. Then multiple definitions of global in a single translation file is allowed in C, it is disallowed in C++ due to the one definition rule, that we have learned. Declaring new type having the same name as the existing struct, union, or enum is allowed in C, but it is disallowed in C++ as all declarations of the type carry an implicit typedef in C++.

In C function prototype without parameter implies that function prototype without parameters imply that the function parameters are unspecified. So, I can call it without a parameter or I can call it with a number of parameters also. In C++, this means zero parameters only, district means only zero parameters. So, for compatibility, when we want to say that I strictly want a function not to have a parameter, then it is best to use void as a parameter which work uniformly in C as well as in C++. So, these are the first set of differences.

(Refer Slide Time: 05:00)



Let us move on, we have also seen the difference between the character type for the character type, because in C the character literals are of type int. Whereas in C++ they are of type char. So, in C++, if you do sizeof of a character literal, its value would be same as the size of the char, which is 1 in C++. Whereas in C, it will mean that the size of the literal a whose be size of an int.

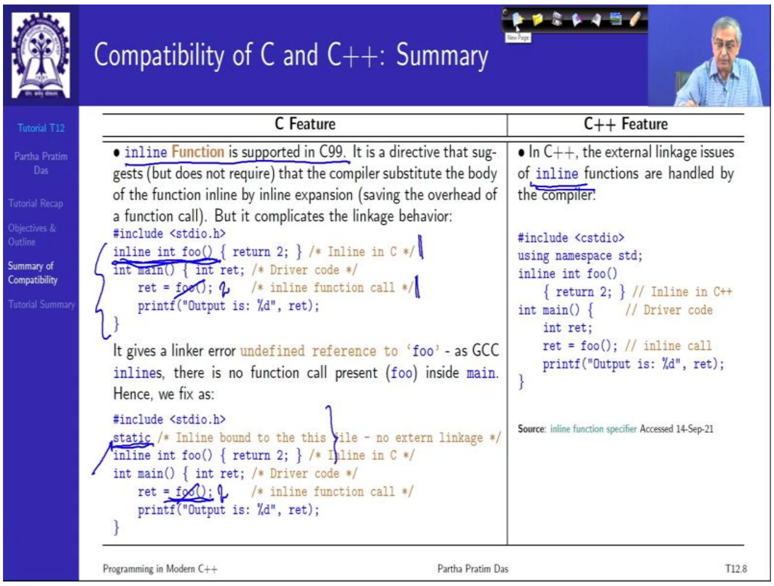
Further, the literal in C++ in C is always a signed expression. Whether or not the char is signed or unsigned, but, in C++ it depends it whether it is a signed expression or not depends on whether char is signed or unsigned type, which is implementation specific that is, if you are trying to rely on that fact in your code, then you must check the compiler specification of the system on which you are going to build.

But remember that if you do that, then it will not, the code will not remain portable to other compilers, because other implementations might have the other interpretation in terms of the signed or unsigned of the char. Boolean type, we have seen earlier also is interesting bool is supported in C99 with the constants true and false and it actually has introduced an _Bool. As a new Boolean type the header stdbool.h provides these macros which define bool, true and false respectively.

Therefore, true and false are necessarily of int type in C, whereas in C++, this is not coming from a standard library, bool is a built in type now, with 2 predefined constants true and false 2 literals, true and false. Therefore, these are all reserved word in C++ and conversion to bool is similar to C though.

Nested structures C++ has a specific namespace or scoping rule. So, if I nest one structure within another, then in C both of their names are globally visible, whereas in C++ the name of the outer structure is globally visible, the name of the inner structure is qualified by the scope name of the outer structure, we have seen all of these examples. So, please try to remember these things.

(Refer Slide Time: 08:20)



Inline function this you may not be much aware about this, we talked about inline function in contrast to macro with parameters and in C++ and justified as to why inline function is important for the C++ specification and how it makes life easier. Inline is actually supported in C, but maybe not widely used. It like C++ it is a directive. So, if I declare this function foo as inline, then at this call, the compiler might decide to directly replace the code and optimize.

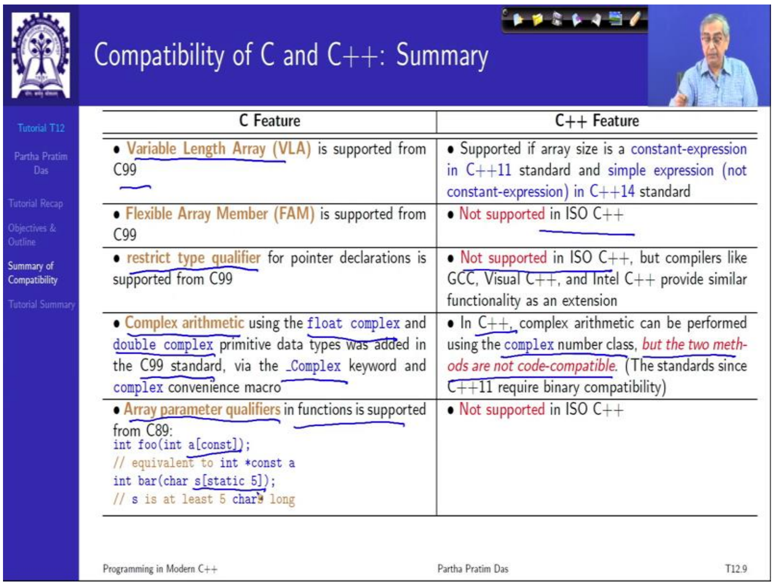
So, basically, this call to foo will be replaced by just the value 2 that it returns because it is again compile time computable. But please remember that in C you may get into variety of linkage errors because of inlining. Just one case, I am mentioning, let us say that, instead of just calling for a function here, let us say that, we say to the static function, you would recall that static in front of a global function means that this name is available only within the file within which this function is actually occurring.

Now, if we do that, then the compiler will inline and replace this call by the constant 2, which means that there will be no trace of this function in the compilent body. So, the linker who is supposed to link and look for, all these things will not be able to do that. So, here we will face that kind of a situational problem.

When we make it static, then the problem gets solved, because, now, if it is static, it is limited only within that file scope. So, the linker is not expecting to link it across different files. And therefore, it will not give a linkage error, but, actually we are not wanting this to be possibly static in that file scope, which means that if this function is to be used in any other file, where I wanted as an inline, I will have to provide a separate header for it, which also will be static.

So, and you know, if we have 2 copies of the inline function implementation, then we are in potential trouble which we have earlier discussed. So, there is, that is some of the reasons that inline is not very popular in C, in C++, we do not have to worry about all these the compiler collects the information and handshakes with the linker to take care of this very nicely.

(Refer Slide Time: 11:49)



Moving on, we talked about variable length array which is available in C99 and C++ needs the array size to be a constant expression, simple expressions are allowed in C++14, but, in general C++ does not support variable length array. C supports flexible array member where a the last as the last member of a struct an array may not have a defined dimension it is possible in C99. It is not supported in any of the C++ dialects.

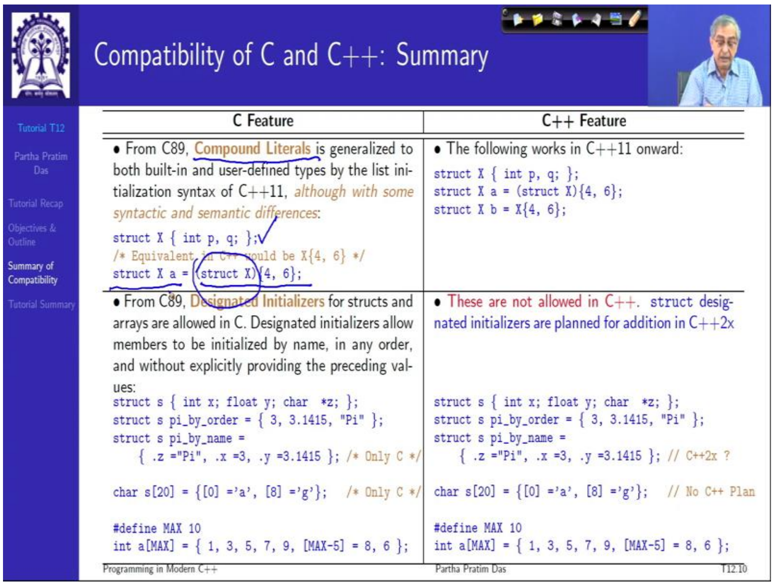
Restrict qualifier, to say that a particular pointer is holding an object to which no other pointer is being held, is not supported in C++ standards. Though some specific compilers provide a similar functionality which is non-portable and therefore, not very advisable to be used.

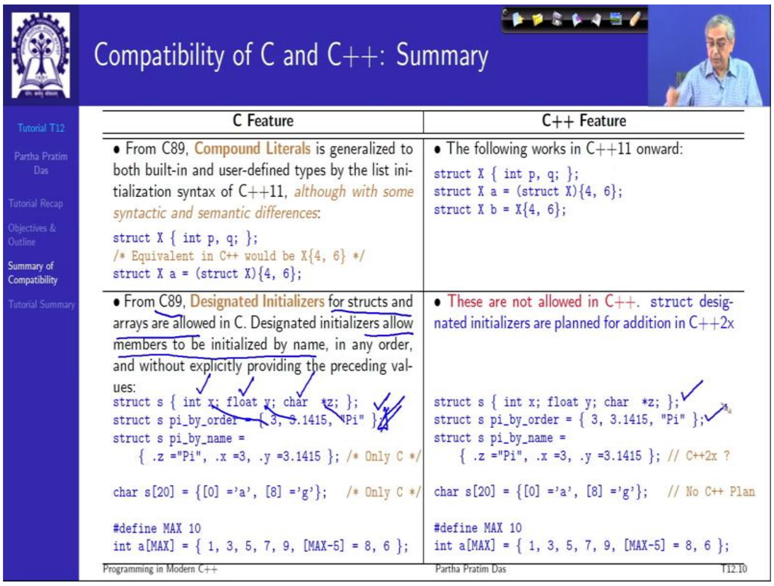
Complex arithmetic is directly supported in C99 with the float complex and double complex primitive data type which C99 introduces using the _Complex keyword. So, you have a complex as a part of the language type. Whereas, in C++ now, the like bool moved from being a standard library to being a built in type in C++, here it is the other way it is a part of the, in C complex is a part of the language now, from C99 But in C++ the complex cannot be performed, it is using the complex number class you can do it.

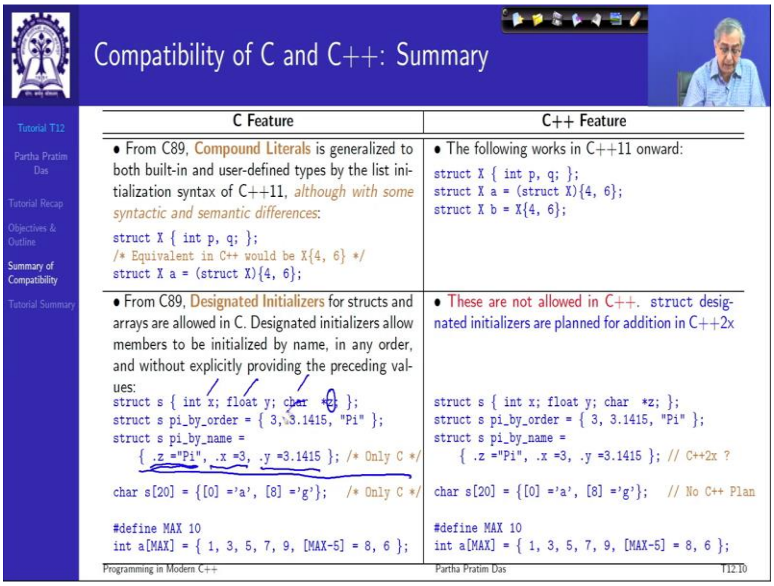
Which is provided as in STL, but, both these methods of how it works in C and how it works in C++ are not compatible. So, you have to be very careful to keep them separate or if you are you know building a C code with C++ compiler, you will have to make sure that this references to complex in C is appropriately ported to, that is the code we will have to be edited, changed, according to what the complex library of STL required since C++.

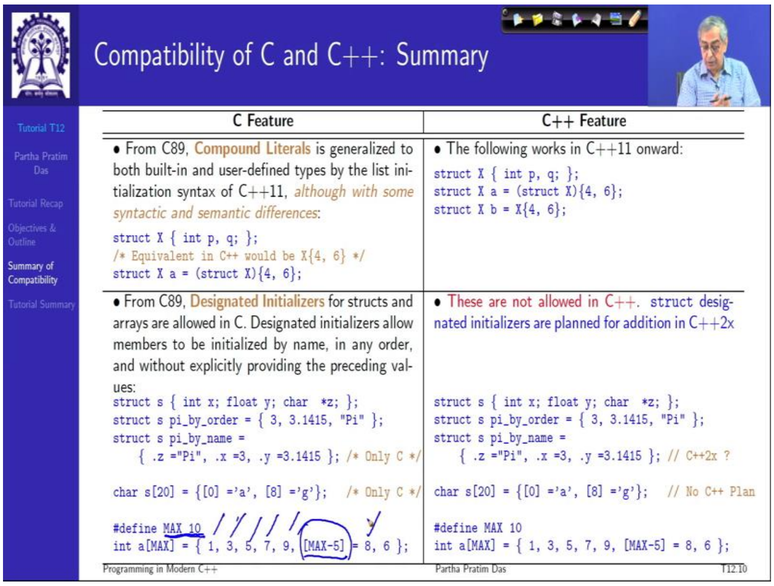
Array parameters qualifier in function is supported in C89, like this. Which is equivalent to saying something like this is not supported in C++. Not a very common feature though in C. So, it is in a way.

(Refer Slide Time: 14:48)









Here there is a lot of changes have gone in from C89. The Compound literal, compound literal means, literals like 5, character constant A, or true, these are simple literal. Compound literal is when we have more than 1 component. So, typically they occur in terms of the structure. So, if I have a structure like this and I say that the structure, I want to initialize it as 4 comma 6, because the 2, so, 4, 6 this pair is also a literal.

It is a compound literal and I specify I cast that to struct x, to say that, now it becomes a structure. So, from this compound become a compound literal, I make it a structure and initialize some variable a of that structure type, which otherwise would have required either component wiser initialization or would have required something like writing something like this, which says that, I am actually constructing x with 2 components 4 and 6 respectively.

Now, from C++11 onwards, there are similar support which is given, but earlier C++03 does not support similar kind of compound literals. C++11 does so, in C++11 you can write this you can write this all will work.

The next is designated initializer in C89, there are designated initializer for struct and arrays designated initializers allow members to be initialized by name. For example, if I am suppose I have this structure with 3 components int, float and char.

So, I know that I can initialize it with simply putting that the literals initializing literals in that sequence. So, this is initialization by position the first one will be taken to be x, second to be y, third to be z, the corresponding type of the component and the type of the literal must be matching. This is same in C++ you can do this in C++.

Now, suppose in C you can also do this that is instead of just relying on the order, you can say that .z that is this component, .z, that is a notation there is no name of the object yet. So, normally we will write that say a .z, or a .y that is not there.

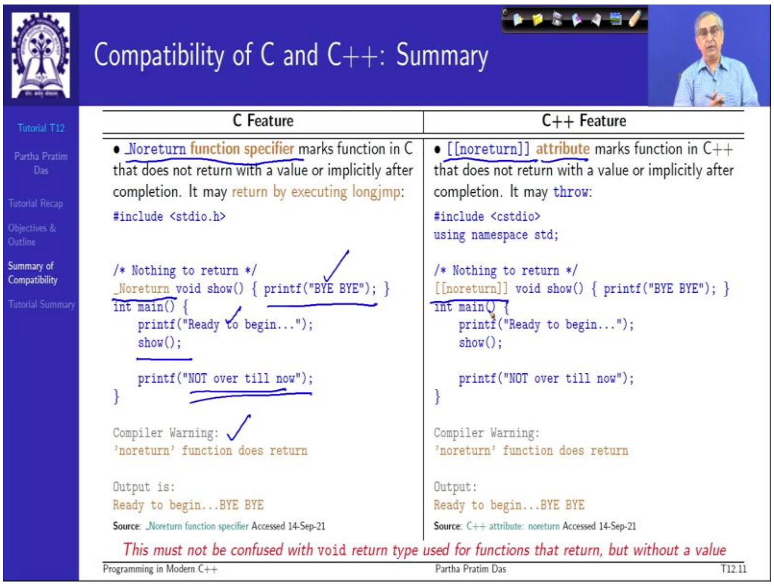
So, you just say .z, initialized with this value, .x initialized with this value, .y initialized with this value. This is allowed in C and they do not need to be in the same order as of the order of these members, because I am. So, this is by name. And the advantage also is that I can skip initializing some of the components which for example, here I cannot skip initializing y, because I have to specify z after that.

But here you can skip some initialization also, similar thing can be done for arrays as well, I can initialize some elements of the array directly, which is not possible in terms of doing a initialization by position, because I have to say it from 0 onwards to the rest of the thing.

So, this is initialization, designated initializer for struct is not supported in C++11, C++14, C++17 and so on. There is some plan to add this in C++ 2x, in the future releases. But we still do not know when it might be coming. Whereas designated initializer for array C++ has no plan to add it. So, if you have such code in C, you will have to do it in a significant rewrite to make it compatible to C++.

So, for example, you can see here I have said that MAX is 10 and I say MAX some initial elements are given in the order. And then I say, MAX minus 5, that is 10 minus 5, 5. Fifth element is 8, I can mix this also in designated array initialization. So, this is the difference in terms of designated initializer for struct and array.

(Refer Slide Time: 20:31)



Now, this is another interesting thing, it is no free return function specifier, if you can write a function in C, with this particular keyword in front of it _Noreturn, N capital in that. Do not confuse it with a function which has void as the return type.

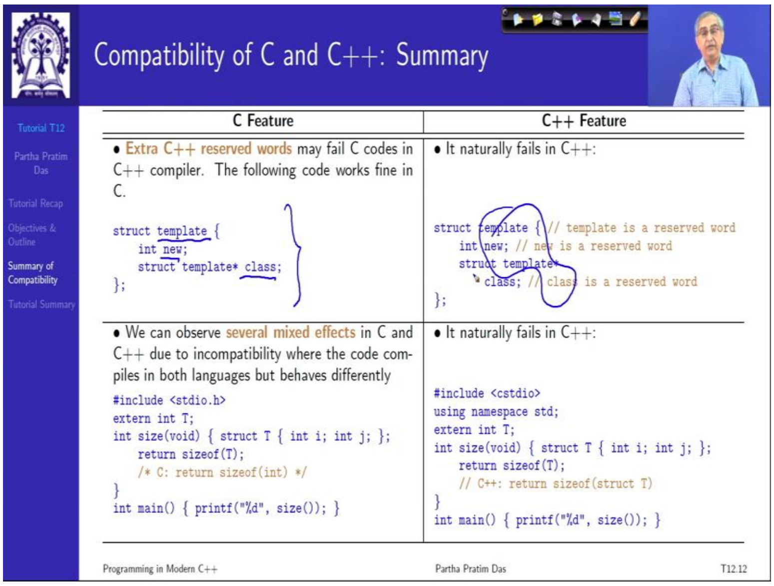
If a function has void as the return type, it means that the function returns but returns no value. If you say _Noreturn, it means that the function does not return, it is kind of a sink, you go under your control as it sinks there. So, you might do wonder why at all we will want that because, for example, if you with this show, if you do the print, that will happen. But this will never get printed. Because the function does not return, it is kind of get stuck there.

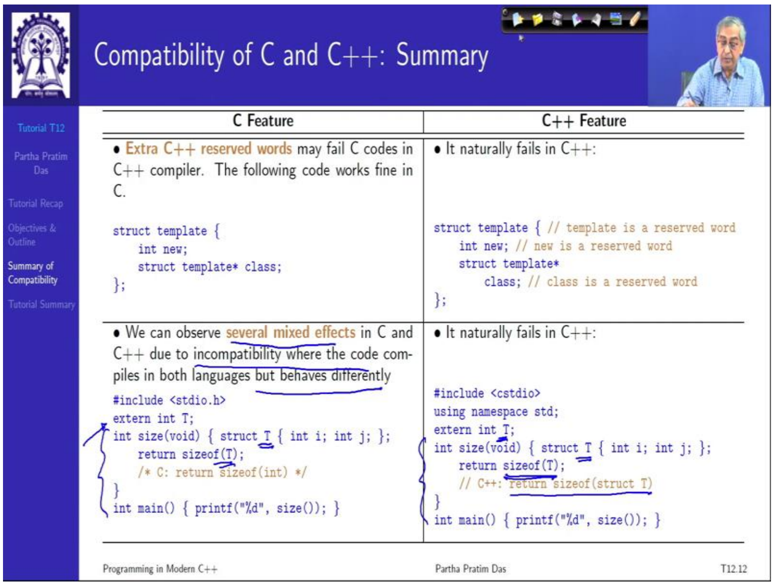
So, if you write _Noreturn, then the compiler will give you a warning because compiler wants to make sure that you really want this behavior and it will only have ready to begin and by by not the next one that is the behavior it is a little a typical behavior. But there is reason why exist in C++ there is an attribute which is written like this, within a pair of square brackets, you write noreturn and it behaves in the same way. So, this is an the keyword is is a different attribute here.

Now, particularly in C++ in C it has a equivalent reason particularly in C++ it might return by throwing, that is it throws an exception. So, what you expect that the default behavior of the function is to continue working and you do not want the control back until you actually have something like it through and you have an exception where the stack is unwinded, just, you will have to recall the behavior we discussed in terms of the exceptions in the module and with tack the control might again come back same thing might happen in C if you use the setjump, long jump paradigm.

But by normal flow of control all that you want to say that well this function is that it and this function will continue and I do not want the control back. So, little peculiar kind of semantics, but these are there and you can see the differences between them.

(Refer Slide Time: 23:40)

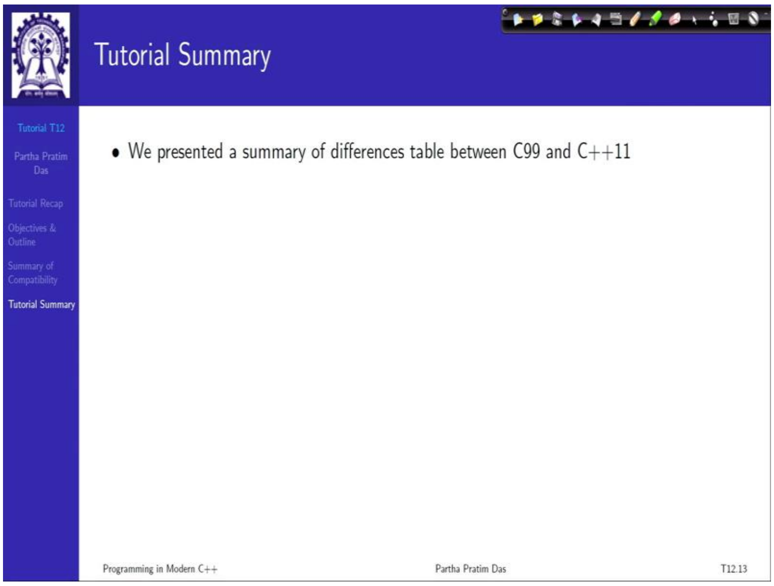




Naturally, this is the most common one that there are extra reserved words in C++. So, a code like this will fail. Which has the name of a struct or tag of a struct as template name of a variable as new, name of a data member of the struct, as class, these all will fail because these are all different keywords in C++, which is easy to understand reduces the portability of C and C++.

And we can observe several mixed effects in C and C++ due to incompatibility of the code though both code compiles in both these languages, this the code that we have given here extern int T and now here, if we look at in C, it what is saying is int T so T is basically a variable of type int. C allows you to write this T. And when you return the sizeof, it returns the sizeof the integer, you write the same thing here write sizeof T, it returns the sizeof struct T. So, you know the language has interpret them differently. That is it a big problem. So, always try to avoid these kinds of situations in your code.

(Refer Slide Time: 25:46)



So, what we have done is, we have summarized in a very brief couple of slides as to all significant differences between C99 and C++11 primarily at times we have digressed to C89 or to C++03 or even C++ 2x. And certainly, many of these are very important to remember, you will know and some of these are not so frequently occurring. So, you will probably not be conscious about them all the time. But keep this difference list handy.

So, that if you are facing difficulties in porting C to C++ or otherwise, you will be able to make those references and see why certain features are not compiling in C++ or are behaving differently across the languages and so on.

So, with this, I conclude the two part tutorial on compatibility of C and C++. I will continue to talk about them in different parts of the modules, particularly in the C++11 or the Modern C++ side because differences are in one way there they are being made more compatible at the same time differences are also increasing because of the newer semantics being added to C++11 onwards. Thank you very much for your attention. Have a good time.