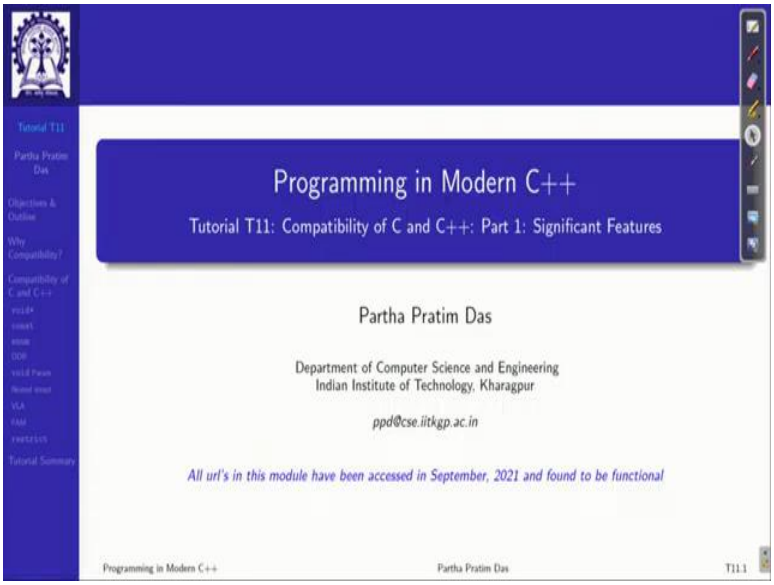
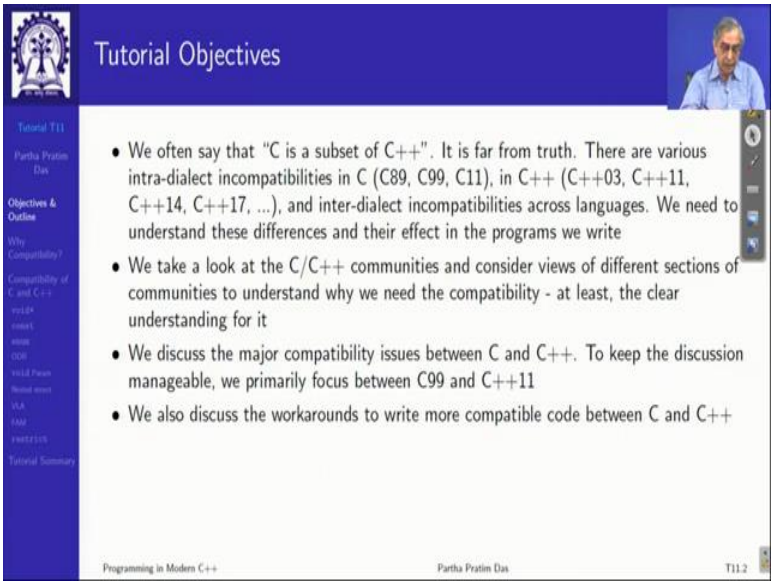
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Tutorial 11
Compatibility of C and C++: Part 1: Significant Features

(Refer Slide Time: 00:43)



Welcome to Programming in Modern C++. I am going to discuss a new aspect in terms of C and C++, the sister languages, in a two part tutorial. This is Tutorial 11, which will be the first part of it. So, we will discuss particularly as to what is, how far C and C++ are actually compatible and we will show you some of the significant features which actually differ between them.

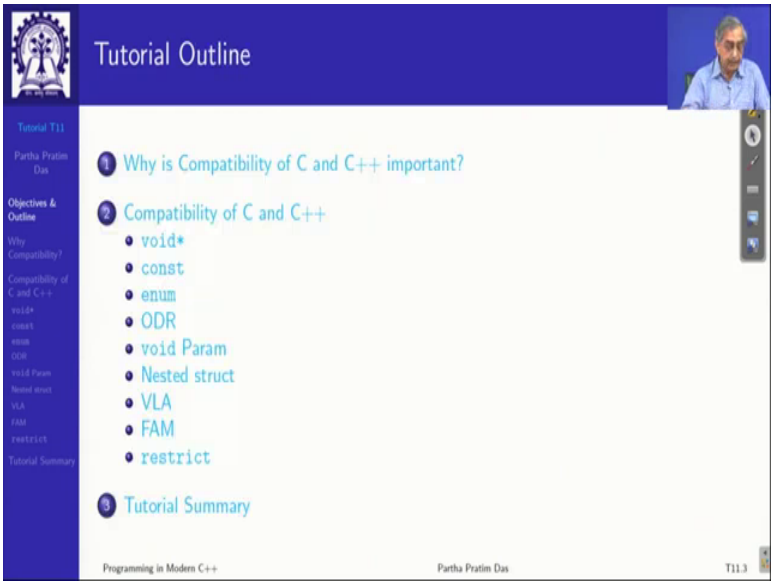
(Refer Slide Time: 00:54)



So, we often say that C is a subset of C++. It is a loose statement and it is far from truth. There are various intra-dialect incompatibilities that is different versions of C or between versions of C++ have incompatibility. It is not only that a version may have a new feature, even the earlier feature may behave differently and certainly there are inter dialect incompatibilities across the language. So, you have some version of C and some version of C++. You are trying to mix them, build them together and so on you may be in for certain surprise.

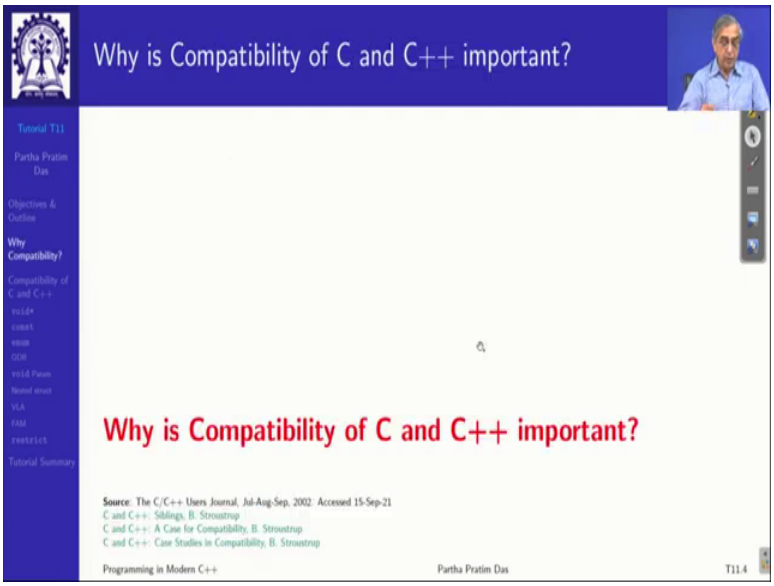
So, we take a look at C/C++ communities, because the communities are not necessarily independent. There is a common community and there is a need to understand how much they are compatible, which part is compatible and which part is not. So that you cannot, you do not get into a lot of surprise. Since there could be several variations of compatibility discussions here I will focus on the two most popular dialects that is C99 for C and C++11 for C++ and for, in cases of, certain cases of incompatibility, we will also discuss about the workarounds. So, that is what we want to achieve in this tutorial and in the next.

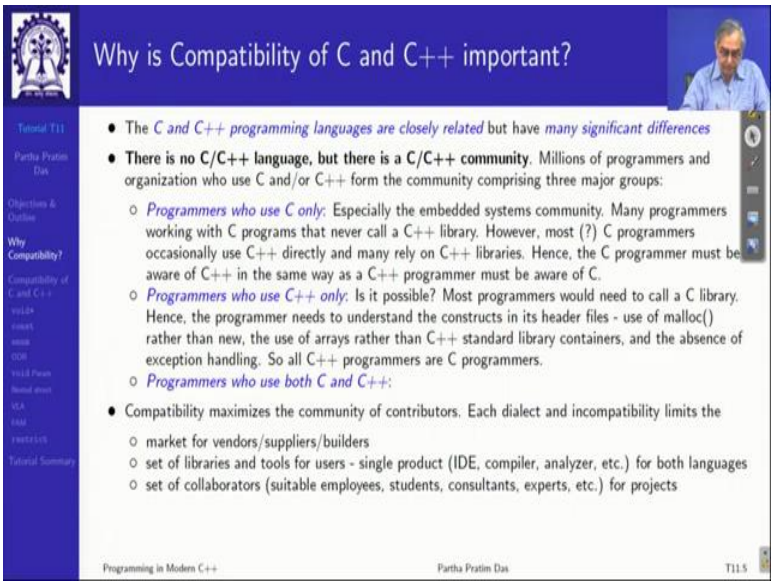
(Refer Slide Time: 02:29)



So, this is the key paths.

(Refer Slide Time: 02:37)





So, first let us ask us to why is compatibility important. Now, C and C++ as languages are closely related. I mean, to the extent that we often call them a C/C++. There is no language as C/C++. It is either C or it is C++. But there are many significant differences. So, though there is no language as C/C++, but there is C/C++ community, the community of developers, thousands of them, probably lakhs of them, who program in C as well as C++ in different extents.

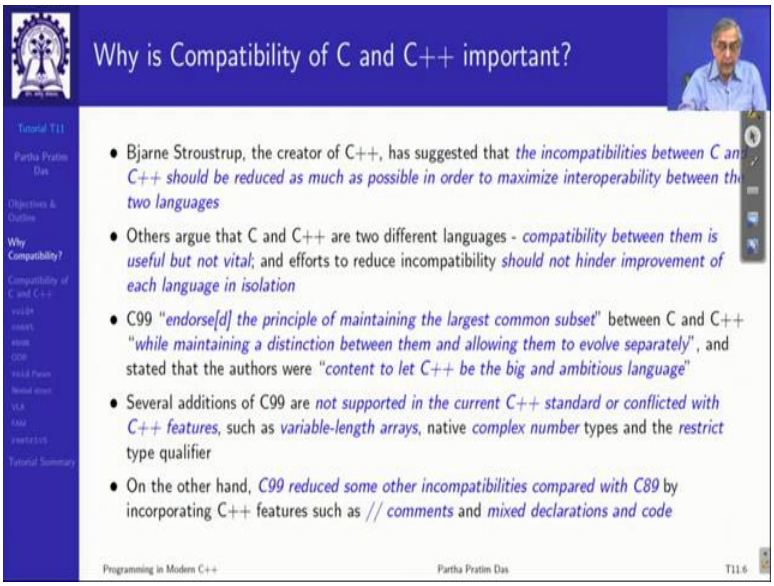
So, we can broadly categorize them into three groups. So, as you mature you will also fall into one of these groups. One is programmers who use C only. There is a large group of programmers who use only C, primarily the embedded systems community and they would probably never call a C++ library. But it is difficult to survive that way, because as C++ has a lot of wealth of code, there is certain wealth of code in C, I mean, C has a lot of wealth of code. Similarly, C++ has also built a lot of valuable code which may benefit a C programmer. But they are they are predominantly C minded programmers and want to focus on that.

The programmers who use C++ only, several of them different systems programmer or complex system developers and so on, naturally they cannot think that they will not know or master C, because of all the overlap and common C features being available in C++ and so on so they have to be very careful as to the part of C in C++, does it behave the same way or behaves the C++ way and so on.

And certainly there is a big third community which who use C and C++ both seamlessly. So, compatibility will maximize the community of builders. If we can more and more we can make

codes compatible, then we will have more users for that, we will have more market for the libraries, we will have better set of tools, more set of collaborators and so on. So, for these reasons, it is very important to have compatible.

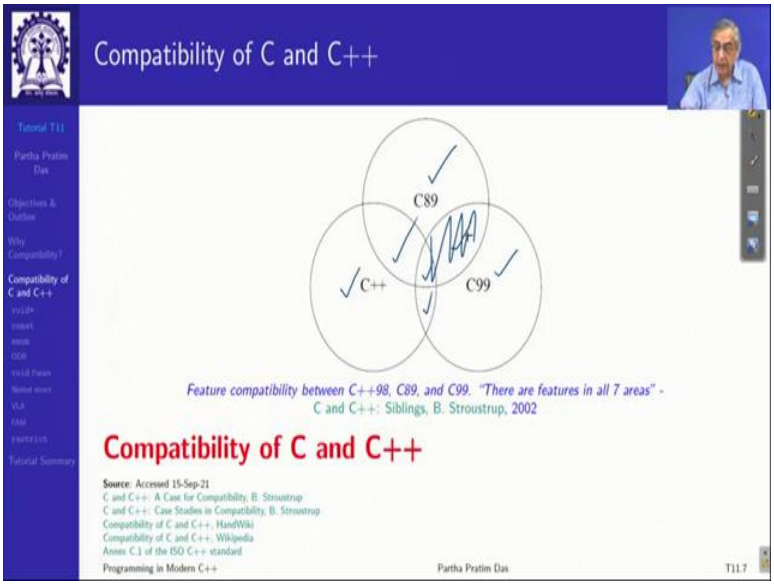
(Refer Slide Time: 05:14)



So, it has always been our vision that C++ should be as compatible to C as possible. And Professor Bjarne Stroustrup the creator suggests that the incompatibilities between C and C++ should be reduced as much as possible in order to maximize interoperability between the two languages, but still there is, there are differences.

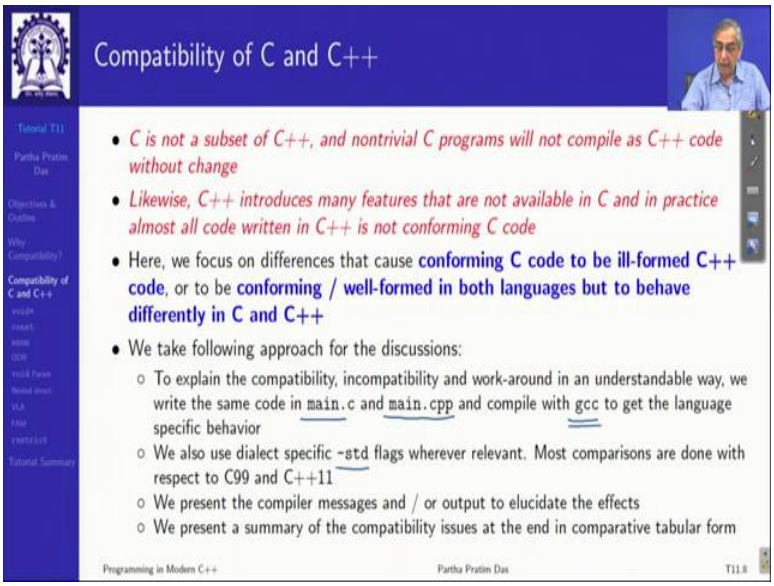
Of course, there is a counter view to this who think that there are different languages and compatibility between them is useful, but not vital, because each of these language has its own philosophy and the more you make them compatible forcefully, then their philosophy gets compromised and so on. So, there have been different such views and also the support of different dialects of C in the corresponding dialect of C++ has not been uniform. So, it is important to understand what these are.

(Refer Slide Time: 06:22)



So, just to give you an idea, there is a Venn diagram of features if. So, C90, C89 that is our oldest standard as you have known, C99 is the most commonly used standard and C++ let us take it as C++98, C++03. If you take these three dialects and draw this Venn diagram, you get 7, 1, 2, 3, 4, 5, 6, 7, 7 different, and interestingly there is some feature which fall in this category that is there are features which are between say these and have compatibility issues.

(Refer Slide Time: 07:21)

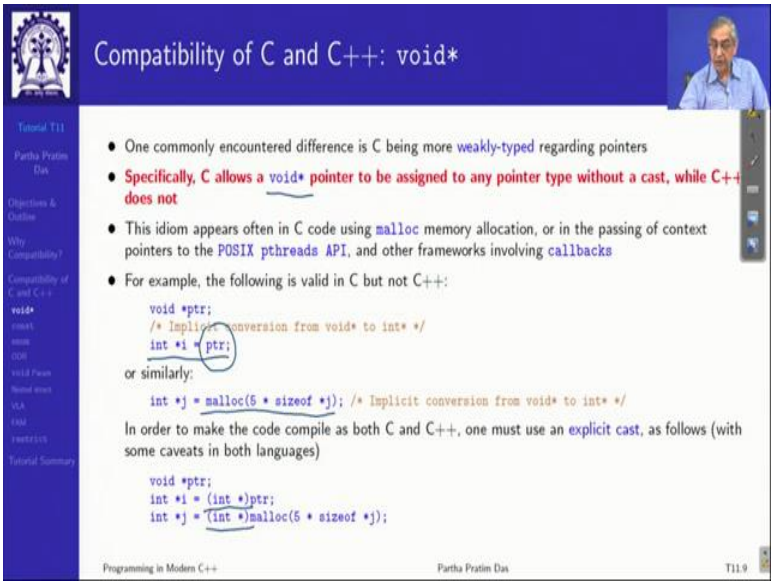


So, let us look at these issues, some of these issues one by one, because C++ is not a subset of, C is not a subset of C++. So, the way we will illustrate this is we will normally try to write the

same code in C file as well as C++ file. So, main.c and main.cpp and see what their behavior is we will use the same gcc compiler and we will use the -std flag if we want a particular dialect and see the difference in the compilation time behavior, in the runtime behavior and so on to understand the, obviously, you can you can read up, the standard the book and all that and know the differences and so on, but I prefer to do it in this hands on way because then we do not need to remember.

You just know that this is, this way I can write the code, compile and get to see what how it is behaving. This finally is engineering. So, the more you can it can do a learn by actually building and executing code is what makes things easier, because you do not have to remember horde of rules written in the books and the blogs and so on.

(Refer Slide Time: 08:40)

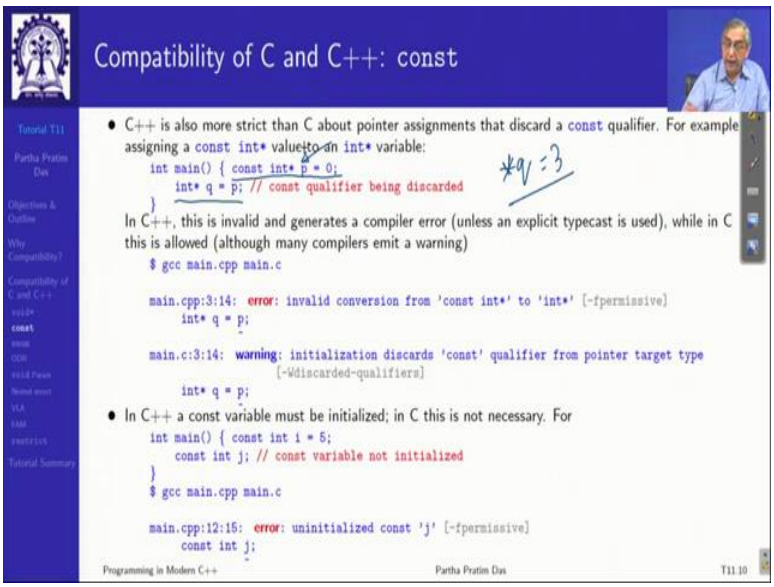


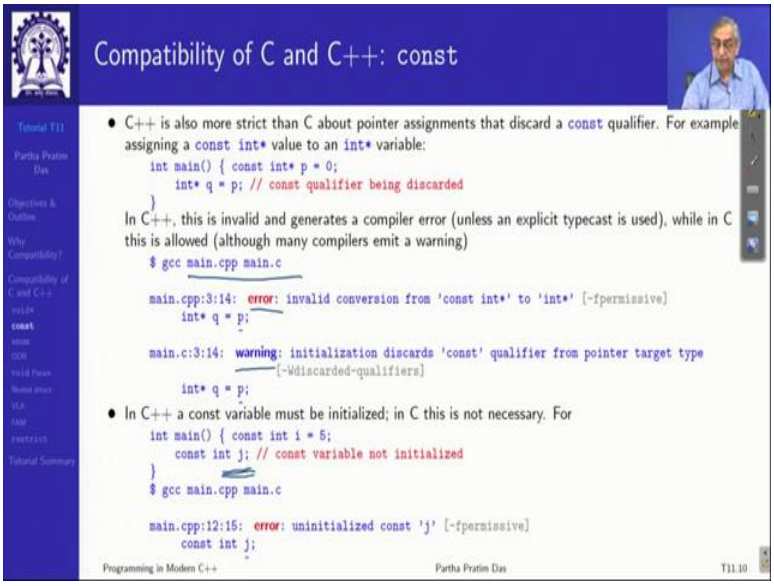
So, the basic issue between C and C++ is C is weakly typed. C is not very strong. C has types, but it is not very strong about that. And one major thing happens with void star. For example, in C any pointer which is void* can be seamlessly cast, implicitly cast to any other point. This is valid in C. So, this was a void* pointer and you have used that to initialize a int* pointer. So, after you do this, it becomes an int star pointer, but this is just allowed.

Similarly, for malloc you can do this, but these things are not allowed in C++. Unless you do an explicit cast, void* will not be allowed to be implicitly converted to any other pointer type in C++ you need to explicitly write this. So, this is a first and very commonly used feature

everywhere that we use in different in malloc in the context of say POSIX thread library, in different frameworks of callback and so on, like. So, we will, this is what shown.

(Refer Slide Time: 10:08)



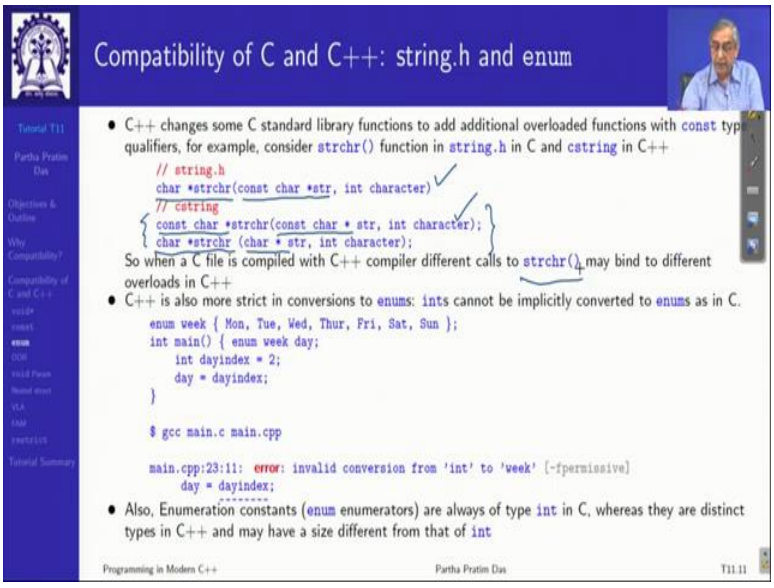


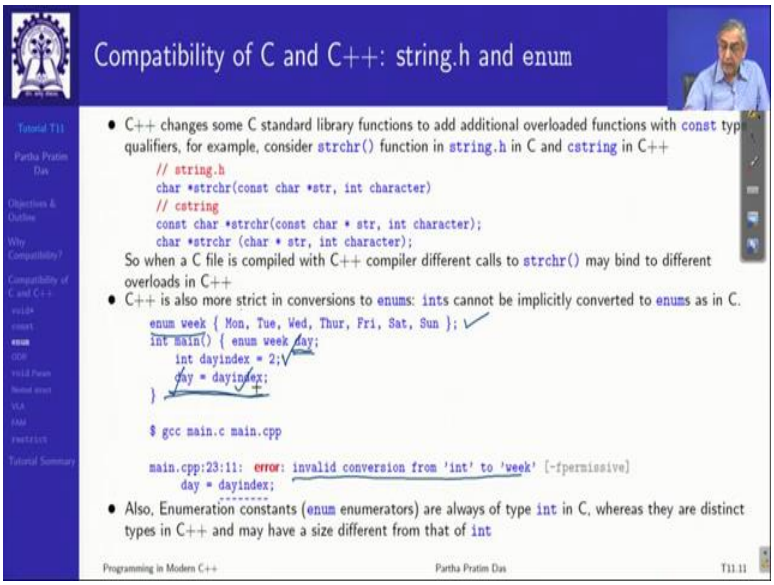
Second is the use of const. C++ is lot more strict in its use of const. So, suppose you have a pointer to a const data, and you use that pointer to initialize a non-const, pointer to a non-const data. Now, this, as we have learned is dangerous, because with this I can actually violate the original rule because I can do *q and change that data which will actually change the value that p is pointing to. But C does allow you to silently discard the const qualifier, which C++ will not allow you to do.

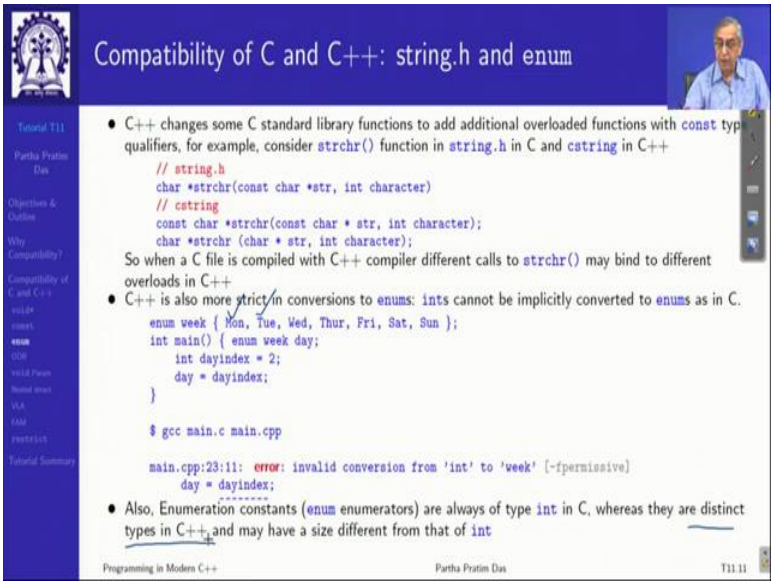
So, if you write this code and you compile this with C and C++ both, you will see that C part will just go with, giving me a warning that you are discarding the const, but it will compile. C++ will give you an error, because this is not valid in C++. C++ is more strict in terms of the type. Another in terms of const, we learned that const variable will always have to be initialized. Rule number one we learned from the very beginning, but not in C. In C, you can write a const like this, where it does not have any initialization. Obviously, if you compile it in C++, it is an error, because uninitialized constants are not allowed.

So, you can see that the first thing that we have to keep in mind is C++ is quite strict about types and every time very few things will be done implicitly. And that too is done implicitly because it has to support certain big chunks of code from C without much changes. But C will, C++ will always tell you that if you had changing a type, please be explicit, use a cast and do that, do not do it silently.

(Refer Slide Time: 12:27)







Now, let us say that compatibility of string.h and enum, string.h is header you know. So, this is the string.h in C standard library. If you open the corresponding, we said that the corresponding standard library component for string.h is cstring.h, put a c in the beginning cstring. But if you actually open the cstring header file, you will find that it has an extra function for, I mean, almost all string function it is there too. So, here if you see the difference is in C, this returns a pointer to character and takes a pointer to constant character.

In C++, there are two, one, it takes a pointer to constant character and returns the same type. Here it takes a pointer to character and returns the same type. So, when you actually call it from the C perspective, this function will get called, because whether the pointer is const or non-const,

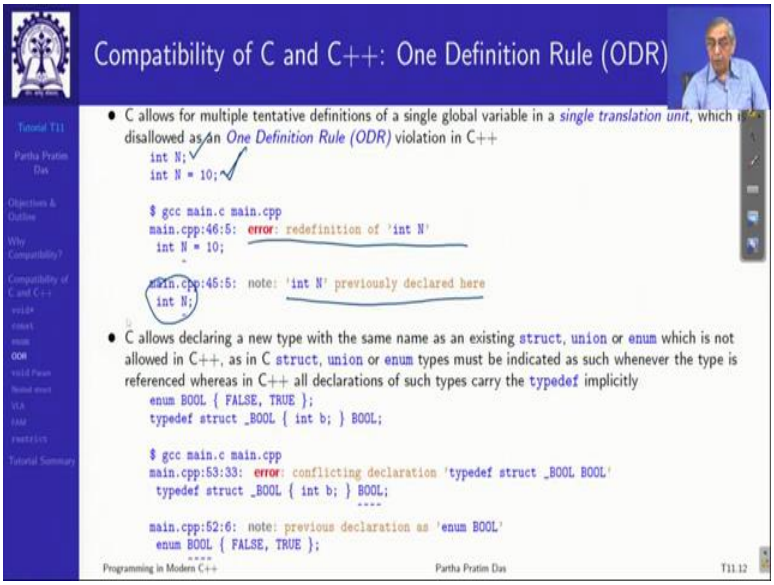
it can always be treated as a pointer to a constant data. But the fact that you have this tells you that C++ does provide an overloading in terms of even basic standard library functions that exist in the string.h and C++ does that to make sure that its whole logical paradigm of consciousness and development and protection of data can be propagated all across.

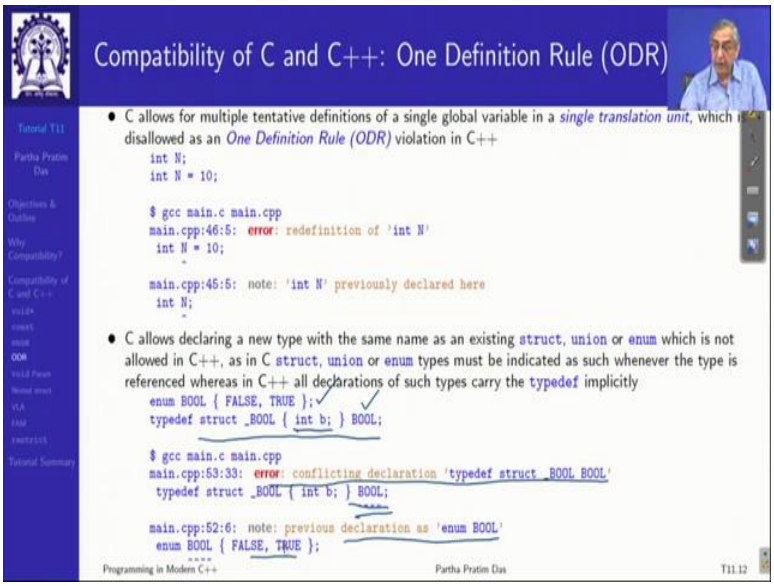
So, in C++ the call to strchr will bind to a different function than what it will bind in C. So there is a significant difference in that and this is just kind of an example. Similar differences exhibit at other places in the standard library as well. Very interesting is a case of enum. Suppose you have an enum and in C we say that enum is nothing but a subtype of int. So, you can treat take an int and implicitly convert it to enum. So, I can say that day is a variable of type enum week, so which means it has these seven possible values that it can take. Then I define an integer which is 2 and I make this assignment. So, this side is an integer and this side is an enum.

Now, this C allows this implicit conversion. C++ gives you an error. The reason C++ gives you an error is enum is not a subtype of int in C++. Enum is a different type in C++. So, it cannot just as a subtype here, you can seamlessly convert implicitly but being a different type it needs explicit conversion. Also the enum constants like all these different seven constants here, these constants are of type int in C, whereas they are distinct types in C++ and may have a different size from int.

For example, if there are seven, it could be they are presented as 3 bits which gives you eight options, not a whole of a 32 bit or 64 bit integer. So, because of this differences, enum will also have to be carefully treated between C and C++. These are very common things.

(Refer Slide Time: 16:39)



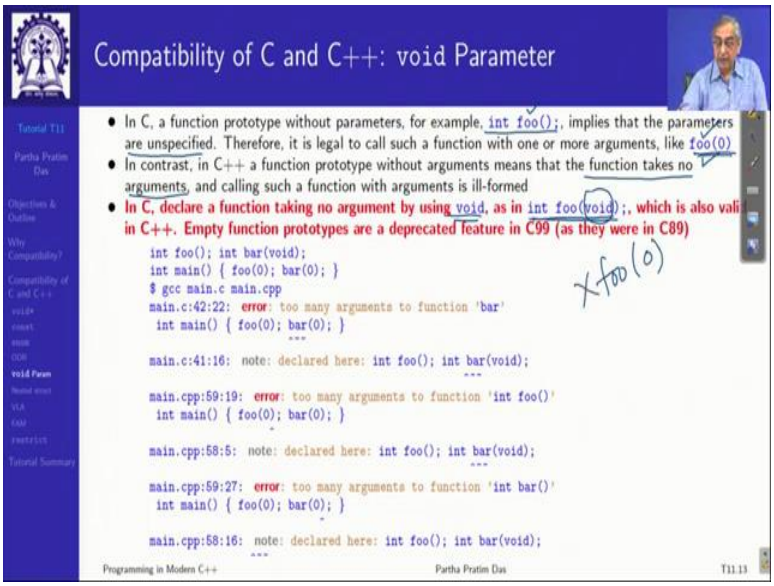


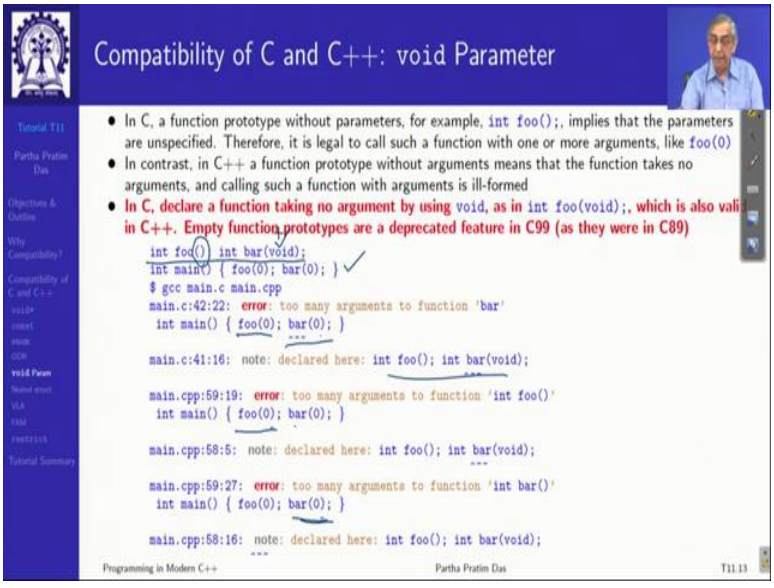
Then C++ has one definition rule that you can have only one definition of a variable, you cannot have multiple. So, in C, you can write say a static variable int n and then later you can write int n initialized 10. In C, it is fine. But in C++, this is an error. When it counts as the second one, it says it is a redefinition. n is previously declared already in in here. One definition rule excludes that. One definition rule excludes the redefinition of the new type by the same name.

For example, in C, you can define a enum bool, say my enum bool is false, true. And you can take make a structure say _bool having a data member b and give it a typedef it to bool. In C, this is permitted. So, but you are actually reusing the name, same name. In C++ this will not be

permitted. In C++ it will say conflicting declaration here. Previous declaration was enum here. So, this just I took just two different types of examples, but it all comes under the ODR or one definition rule of C++ that in C++ you can have only one definition of a variable or a type and trying to redefine is always an error. In C it is not always so.

(Refer Slide Time: 18:28)





Now, how do you treat void as a parameter. Suppose in C you provide a function prototype as int foo, no parameter. Now, this implies that parameters are unspecified. Whereas when you write it in parameters are unspecified. When you create the same signature in C++, it means that it takes no arguments. So, in C, if you have declared something as int foo without parameter, as in here,

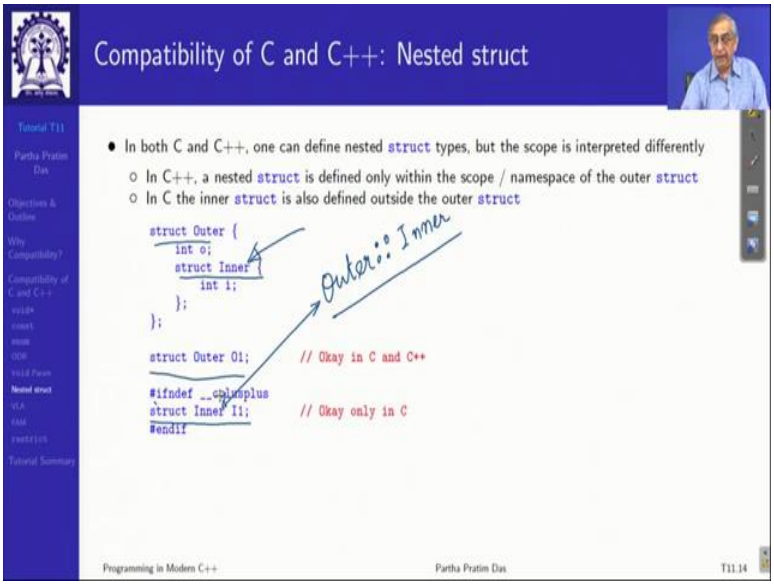
you can call it as foo(0), because you said it is unspecified. So, you can call it with any number of parameters. In C++, what you meant that it takes no parameters. So this calling it as foo(0) is actually a violation of the signature that you have. So, that is a subtle difference.

So, in C if you want to declare a function which is kind of equivalent to C++ no argument, you have to use argument void. So it says the same thing in a little roundabout way. All that it says that it takes a parameter of type void which means that it takes a no parameter, but it is specified. It takes a no parameter. You saying it in this way. This is void type.

Now, you cannot call, now if you call foo(0), this is an error in C as well. So, this is, you can see that when you get into mixing, porting and all that these kinds of things will cause pain of compatibility, because if you have these two declarations and you have a main which calls both of these functions with 0, a parameter 0, then obviously foo(0) is in terms of C it is fine, because it is unspecified. bar(0) is not because it is specified that you will not have a parameter.

And so, this is, in C++, foo(0) is not possible, because you have 0 number of parameters specified. Similarly, bar(0) is also not possible because you have said it is void which also means no parameters. So, this is the, so you can see that the same code you compile with two different compilers you will have different compile time behavior and these are the typical compatibility issues you will face.

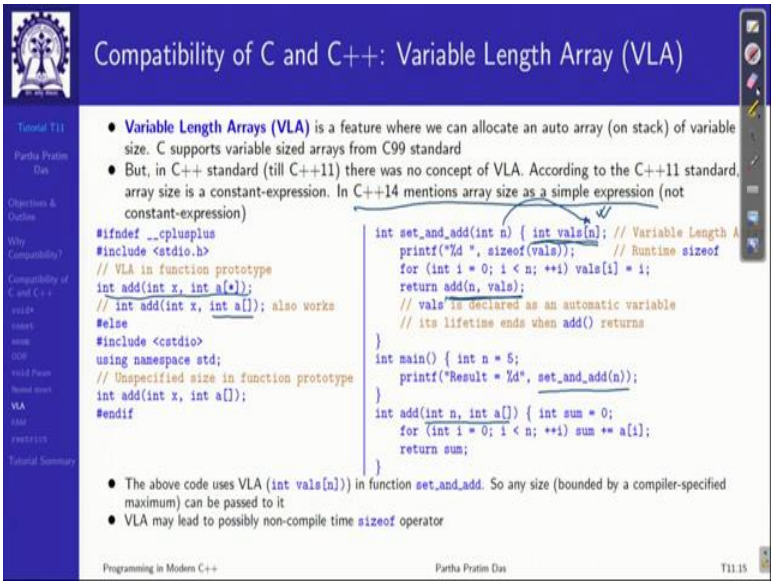
(Refer Slide Time: 21:29)



Nested structure you will understand from your discussions on the namespace that I can have a structure outer and inside that I have a structure inner. In outer I can refer to in C and C++ in the same way. I can in C, I can refer to inner directly, because every name in C is global. So, if not defined C++ that if it is C inner. But in C++ I cannot refer to inner directly, because the every struct is a namespace.

So, the actual name of this inner is not inner. It is outer::inner. So, you can see the difference between C and C++ in case of nested structure. It looks something, I mean, there is no C++ feature apparently here, but the interpretations are very different. And therefore, you will have incompatibility between them.

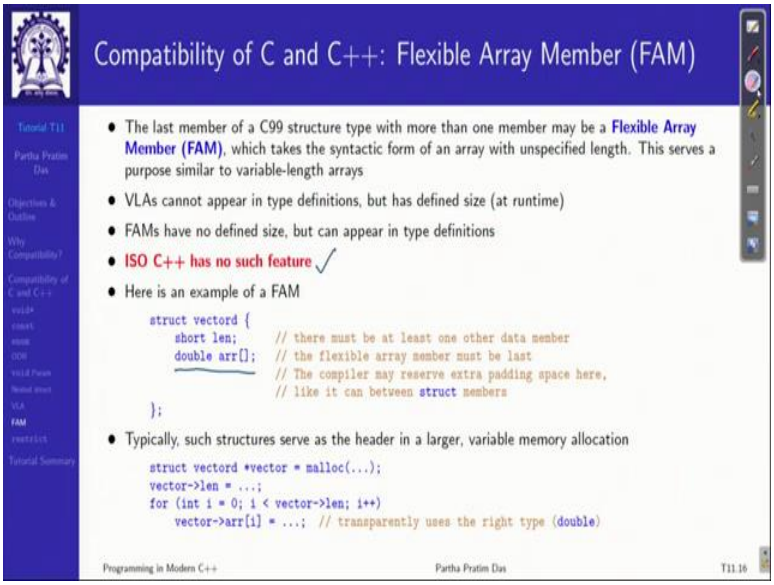
(Refer Slide Time: 22:36)



Variable length array is a well talked of feature in C where you can pass a array without any, without a fixed size and you can deal with that. So, this is the way you can say I have a variable length array you can also say this. Now, here in this function set and add I have declared this where n is a parameter. This is a variable length array which is available in C. And so I can, from the main I can do set and add for n that keeps an appropriate size of the vals array as an automatic variable during the set add function call and from that I use the function add which takes this array and does addition.

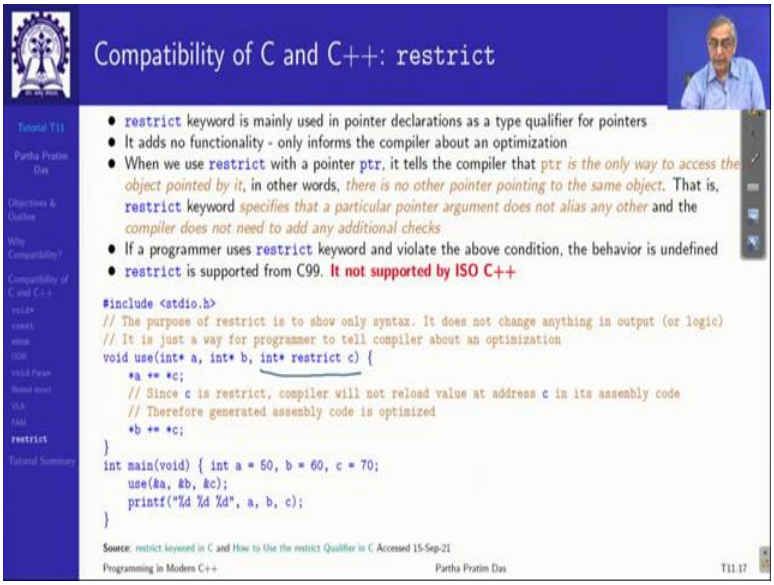
So, this is a feature which is available in C. In C++ standard till C++11 there is no concept of variable length array. In, up to C++11 array size the constant expression. C++14 is introducing this as a simple expression, not a constant expression. We will learn more about that in that part. But what I wanted to highlight is if you are using any kind of, any kind of VLA in C that will not be compatible with the C++.

(Refer Slide Time: 24:28)



Flexible array member is an extension of that where C allows that the last member of a struct could be an array without specified dimension. So, it could be of any size. So, this is called a flexible array member which becomes easy for use because you can get to know the actual data requirement at the runtime and accordingly it will take care of the number of data you want to put in of the appropriate type. But again this is a C only feature. ISO C++ has no such feature as FAM. So, C++ does not even recognize this as a feature. You will have a severe incompatibility for this.

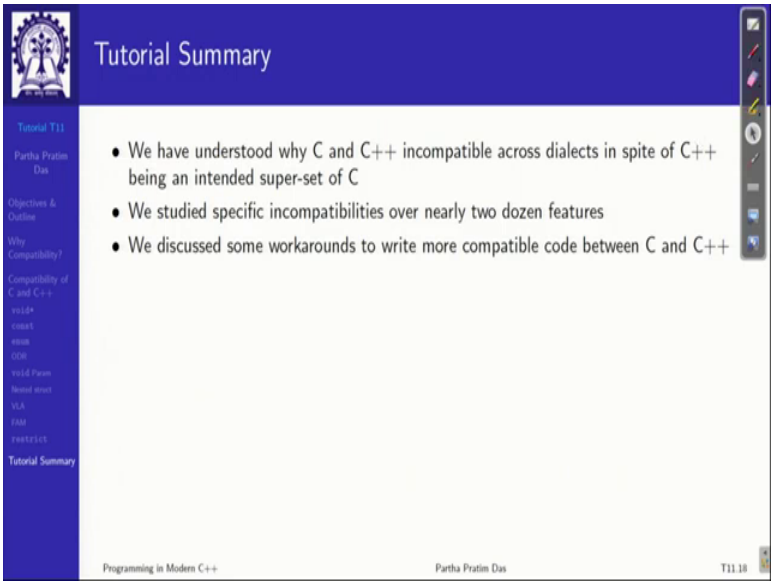
(Refer Slide Time: 25:17)



restrict was provided in C++ to mean uniqueness of pointers. So, you say that a pointer is restrict, which means you are trying to say that it points to an object which is not pointed to by anyone else. If I say int* restrict c, then I am saying that c is a pointer to an object which is not pointed to by anyone else. So, if it is not pointed to by anyone else, naturally, things become much easier for the compiler, because it does not have to look at the possibility of that value getting changed by someone else and so on.

So, restrictor was provided in C99. But somehow this has been a very well debated feature. And so far ISO C++ does not support this restrict feature. Rather you can get the similar effect by unique_ptr and those kinds of stuff that you get through functors, smart pointers and so on. But this language built-in feature of restrict is not available in C++.

(Refer Slide Time: 26:40)



So, to summarize, this is not exhaustive. But in this part of the tutorial, I have tried to take you through the fact that those C and C++ are very closely related. There are some marginal to medium to severe incompatibility across dialects of C and C++ which need to be clearly understood or at least studied when you want to do a mix of programs between C and C++ or you are trying to migrate a simple C program into C++ and so on.

So, the major features we have discussed in the second part of this tutorial in tutorial 12. I will present these and a number of minor features also in terms of a comparison table, which you can be, I mean, it is a couple of pages of slides of comparison table which you can keep handy in

case you are into mixed language project or migration project which is very, very common in the industry to have. Thank you very much. Try this out and see the difficulties for yourself, and see you in the second part of this tutorial.