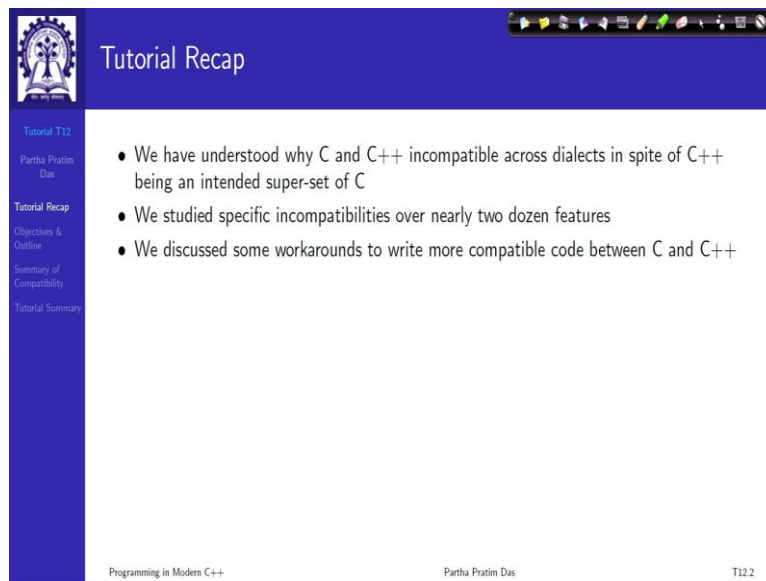


Programming in Modern C++
Professor Parthe Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Tutorial 12
Compatibility of C and C++: Part2: Summary

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)

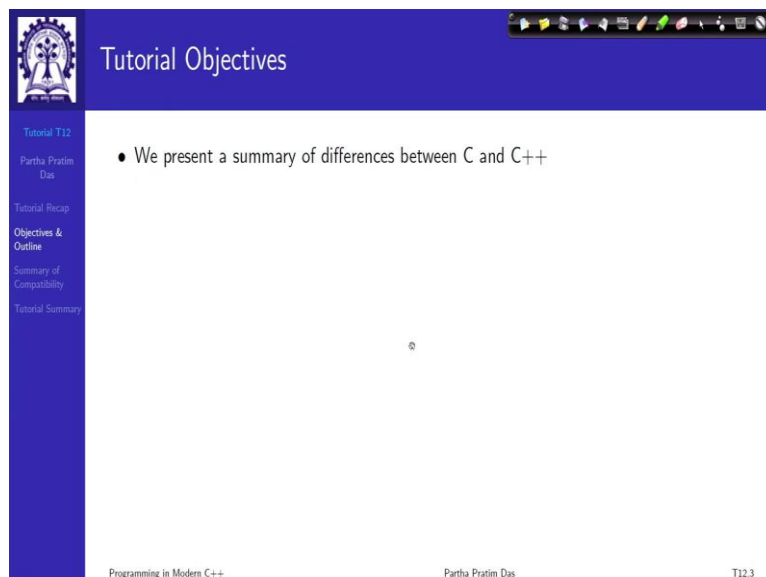


The slide is titled "Tutorial Recap" and features a blue header with the IIT Kharagpur logo on the left. A sidebar on the left contains a table of contents with "Tutorial Recap" highlighted. The main content area lists three bullet points: "We have understood why C and C++ incompatible across dialects in spite of C++ being an intended super-set of C", "We studied specific incompatibilities over nearly two dozen features", and "We discussed some workarounds to write more compatible code between C and C++". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "T12.2".

Tutorial Recap

- We have understood why C and C++ incompatible across dialects in spite of C++ being an intended super-set of C
- We studied specific incompatibilities over nearly two dozen features
- We discussed some workarounds to write more compatible code between C and C++

Programming in Modern C++ Partha Pratim Das T12.2



The slide is titled "Tutorial Objectives" and features a blue header with the IIT Kharagpur logo on the left. A sidebar on the left contains a table of contents with "Objectives & Outline" highlighted. The main content area lists one bullet point: "We present a summary of differences between C and C++". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "T12.3".

Tutorial Objectives

- We present a summary of differences between C and C++

Programming in Modern C++ Partha Pratim Das T12.3

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)

The slide is titled "Tutorial Outline" and features a blue header with a logo on the left. A vertical sidebar on the left contains a list of navigation items: "Tutorial T12", "Partha Pratim Das", "Tutorial Recap", "Objectives & Outline", "Summary of Compatibility", and "Tutorial Summary". The main content area lists three items: "1 Tutorial Recap", "2 Summary of Compatibility", and "3 Tutorial Summary". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "T12.4".

The slide is titled "Summary of Compatibility" and features a blue header with a logo on the left. A vertical sidebar on the left contains a list of navigation items: "Tutorial T12", "Partha Pratim Das", "Tutorial Recap", "Objectives & Outline", "Summary of Compatibility", and "Tutorial Summary". The main content area displays the title "Summary of Compatibility" in red, followed by the text: "We summarize the incompatibility in features already discussed, and also introduce a few new ones in brief". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "T12.5".

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

(Refer Slide Time: 01:47)

C Feature	C++ Feature
<ul style="list-style-type: none">• Implicit Conversion of <code>void*</code> is allowed in C	<ul style="list-style-type: none">• Implicit conversion is not allowed in C++; allowed only with explicit cast
<ul style="list-style-type: none">• Implicit Discard of <code>const</code> qualifier for pointer is allowed in C	<ul style="list-style-type: none">• Implicit discard is not allowed in C++; allowed only with explicit cast
<ul style="list-style-type: none">• Initialization of <code>const</code> Variable is optional in C	<ul style="list-style-type: none">• Initialization is mandatory in C++
<ul style="list-style-type: none">• C Standard Library functions have unique signature. For example, <code>strchr</code> in <code>string.h</code>	<ul style="list-style-type: none">• In C++, they may have additional overloaded functions. For example, <code>strchr</code> in <code>cstring</code>
<ul style="list-style-type: none">• Implicit Conversion of <code>int</code> to <code>enum</code> is allowed in C	<ul style="list-style-type: none">• Implicit conversion is not allowed in C++; allowed only with explicit cast
<ul style="list-style-type: none">• <code>enum</code> Enumerators are of type <code>int</code> in C	<ul style="list-style-type: none">• Enumerators are of distinct types in C++, having different size from <code>int</code>
<ul style="list-style-type: none">• Multiple definitions of a global in a single translation unit is allowed in C	<ul style="list-style-type: none">• It is disallowed in C++ due to One Definition Rule (ODR)
<ul style="list-style-type: none">• Declaring a new type having same name as an existing <code>struct</code>, <code>union</code> or <code>enum</code> is allowed in C	<ul style="list-style-type: none">• It is disallowed in C++ as all declarations of such types carry the <code>typedef</code> implicitly
<ul style="list-style-type: none">• In C, a function prototype without parameters implies that the parameters are unspecified, and can be called with zero or more parameters	<ul style="list-style-type: none">• In C++, it means zero parameter only

For compatibility, use `void` for parameter when there is no parameter

Programming in Modern C++ Partha Pratim Das T12.6

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, distinct 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)

C Feature	C++ Feature
<ul style="list-style-type: none"> Character literals like 'a' are of type int in C. Hence: <ol style="list-style-type: none"> <code>sizeof('a') = sizeof(int)</code> 'a' is always a signed expression, regardless of whether or not char is a signed or unsigned type 	<ul style="list-style-type: none"> They are of type char in C++. Hence: <ol style="list-style-type: none"> <code>sizeof('a') = sizeof(char) = 1</code> If 'a' a signed expression or not depends on whether char is a signed or unsigned type, which is implementation specific
<ul style="list-style-type: none"> Boolean type bool is supported in C99 with constants true and false. In C99, a new keyword, <code>_Bool</code>, is introduced as the new Boolean type. The header <code>stdbool.h</code> provides macros <code>bool</code>, <code>true</code> and <code>false</code> that are defined as <code>_Bool</code>, 1 and 0, respectively. Therefore, <code>true</code> and <code>false</code> have type int in C 	<ul style="list-style-type: none"> In C++, <code>bool</code> is a built-in type with constants <code>true</code> and <code>false</code>. All these are reserved keywords. Conversions to <code>bool</code> are similar to C
<ul style="list-style-type: none"> For Nested structs, the inner struct is also defined outside the outer struct in C 	<ul style="list-style-type: none"> A nested struct is defined only within the scope / namespace of the outer struct in C++

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)

The slide is titled "Compatibility of C and C++: Summary" and is presented by Partha Pratim Das. It compares the handling of inline functions in C and C++.

C Feature	C++ Feature
<ul style="list-style-type: none">• <u>inline function</u> is supported in C99. It is a directive that suggests (but does not require) that the compiler substitute the body of the function inline by inline expansion (saving the overhead of a function call). But it complicates the linkage behavior: <pre>#include <stdio.h> inline int foo() { return 2; } /* Inline in C */ int main() { int ret; /* Driver code */ ret = foo(); /* inline function call */ printf("Output is: %d", ret); }</pre> <p>It gives a linker error <code>undefined reference to 'foo'</code> - as GCC inlines, there is no function call present (<code>foo</code>) inside <code>main</code>. Hence, we fix as:</p> <pre>#include <stdio.h> static /* Inline bound to the this file - no extern linkage */ inline int foo() { return 2; } /* inline in C */ int main() { int ret; /* Driver code */ ret = foo(); /* inline function call */ printf("Output is: %d", ret); }</pre>	<ul style="list-style-type: none">• In C++, the external linkage issues of <u>inline functions</u> are handled by the compiler. <pre>#include <stdio> using namespace std; inline int foo() { return 2; } // Inline in C++ int main() { // Driver code int ret; ret = foo(); // inline call printf("Output is: %d", ret); }</pre> <p>Source: inline function specifier Accessed 14-Sep-21</p>

Programming in Modern C++ | Partha Pratim Das | T12.8

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 compiled 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)

C Feature	C++ Feature
<ul style="list-style-type: none"> Variable Length Array (VLA) is supported from C99 	<ul style="list-style-type: none"> Supported if array size is a constant-expression in C++11 standard and simple expression (not constant-expression) in C++14 standard
<ul style="list-style-type: none"> Flexible Array Member (FAM) is supported from C99 	<ul style="list-style-type: none"> Not supported in ISO C++
<ul style="list-style-type: none"> restrict type qualifier for pointer declarations is supported from C99 	<ul style="list-style-type: none"> Not supported in ISO C++, but compilers like GCC, Visual C++, and Intel C++ provide similar functionality as an extension
<ul style="list-style-type: none"> Complex arithmetic using the float complex and double complex primitive data types was added in the C99 standard, via the _Complex keyword and complex convenience macro 	<ul style="list-style-type: none"> In C++, complex arithmetic can be performed using the complex number class, but the two methods are not code-compatible. (The standards since C++11 require binary compatibility)
<ul style="list-style-type: none"> Array parameter qualifiers in functions is supported from C89: <pre>int foo(int a[const]); // equivalent to int *const a int bar(char s[static 5]); // s is at least 5 char* long</pre> 	<ul style="list-style-type: none"> Not supported in ISO C++

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)




Compatibility of C and C++: Summary




C Feature	C++ Feature
<ul style="list-style-type: none"> From C89, Compound Literals is generalized to both built-in and user-defined types by the list initialization syntax of C++11, <i>although with some syntactic and semantic differences</i>: <pre>struct X { int p, q; }; /* Equivalent in C++ would be X{4, 6} */ struct X a = (struct X){4, 6};</pre> From C89, Designated Initializers for structs and arrays are allowed in C. Designated initializers allow members to be initialized by name, in any order, and without explicitly providing the preceding values: <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; /* Only C */ char s[20] = {[0] = 'a', [8] = 'g'}; /* Only C */ #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> 	<ul style="list-style-type: none"> The following works in C++11 onward: <pre>struct X { int p, q; }; struct X a = (struct X){4, 6}; struct X b = X{4, 6};</pre> These are not allowed in C++. struct designated initializers are planned for addition in C++2x <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; // C++2x ? char s[20] = {[0] = 'a', [8] = 'g'}; // No C++ Plan #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre>

Programming in Modern C++
Partha Pratim Das
T12.T0



Compatibility of C and C++: Summary



C Feature	C++ Feature
<ul style="list-style-type: none"> From C89, Compound Literals is generalized to both built-in and user-defined types by the list initialization syntax of C++11, <i>although with some syntactic and semantic differences</i>: <pre>struct X { int p, q; }; /* Equivalent in C++ would be X{4, 6} */ struct X a = (struct X){4, 6};</pre> From C89, Designated Initializers for structs and arrays are allowed in C. Designated initializers allow members to be initialized by name, in any order, and without explicitly providing the preceding values: <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; /* Only C */ char s[20] = {[0] = 'a', [8] = 'g'}; /* Only C */ #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> 	<ul style="list-style-type: none"> The following works in C++11 onward: <pre>struct X { int p, q; }; struct X a = (struct X){4, 6}; struct X b = X{4, 6};</pre> These are not allowed in C++. struct designated initializers are planned for addition in C++2x <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; // C++2x ? char s[20] = {[0] = 'a', [8] = 'g'}; // No C++ Plan #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre>

Programming in Modern C++
Partha Pratim Das
T12.T0

Compatibility of C and C++: Summary

C Feature	C++ Feature
<ul style="list-style-type: none"> From C89, Compound Literals is generalized to both built-in and user-defined types by the list initialization syntax of C++11, <i>although with some syntactic and semantic differences</i>: <pre>struct X { int p, q; }; /* Equivalent in C++ would be X{4, 6} */ struct X a = (struct X){4, 6};</pre> From C89, Designated Initializers for structs and arrays are allowed in C. Designated initializers allow members to be initialized by name, in any order, and without explicitly providing the preceding values: <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; /* Only C */ char s[20] = {[0] = 'a', [8] = 'g'}; /* Only C */ #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> 	<ul style="list-style-type: none"> The following works in C++11 onward: <pre>struct X { int p, q; }; struct X a = (struct X){4, 6}; struct X b = X{4, 6};</pre> These are not allowed in C++. struct designated initializers are planned for addition in C++2x <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; // C++2x ? char s[20] = {[0] = 'a', [8] = 'g'}; // No C++ Plan #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre>

Programming in Modern C++ Partha Pratim Das T12.10

Compatibility of C and C++: Summary

C Feature	C++ Feature
<ul style="list-style-type: none"> From C89, Compound Literals is generalized to both built-in and user-defined types by the list initialization syntax of C++11, <i>although with some syntactic and semantic differences</i>: <pre>struct X { int p, q; }; /* Equivalent in C++ would be X{4, 6} */ struct X a = (struct X){4, 6};</pre> From C89, Designated Initializers for structs and arrays are allowed in C. Designated initializers allow members to be initialized by name, in any order, and without explicitly providing the preceding values: <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; /* Only C */ char s[20] = {[0] = 'a', [8] = 'g'}; /* Only C */ #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre> 	<ul style="list-style-type: none"> The following works in C++11 onward: <pre>struct X { int p, q; }; struct X a = (struct X){4, 6}; struct X b = X{4, 6};</pre> These are not allowed in C++. struct designated initializers are planned for addition in C++2x <pre>struct s { int x; float y; char *z; }; struct s pi_by_order = { 3, 3.1415, "Pi" }; struct s pi_by_name = { .z = "Pi", .x = 3, .y = 3.1415 }; // C++2x ? char s[20] = {[0] = 'a', [8] = 'g'}; // No C++ Plan #define MAX 10 int a[MAX] = { 1, 3, 5, 7, 9, [MAX-5] = 8, 6 };</pre>

Programming in Modern C++ Partha Pratim Das T12.10

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 wise 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)

The slide is titled "Compatibility of C and C++: Summary" and is presented by Partha Pratim Das. It compares the handling of non-returning functions in C and C++.

C Feature	C++ Feature
<ul style="list-style-type: none">• _Noreturn function specifier marks function in C that does not return with a value or implicitly after completion. It may return by executing <code>longjmp</code>. <pre>#include <stdio.h> /* Nothing to return */ _Noreturn void show() { printf("BYE BYE"); } int main() { printf("Ready to begin..."); show(); printf("NOT over till now"); }</pre> <p>Compiler Warning: 'noreturn' function does return</p> <p>Output is: Ready to begin...BYE BYE</p> <p>Source: _Noreturn function specifier Accessed 14-Sep-21</p> <p><i>This must not be confused with void return type used for functions that return, but without a value</i></p>	<ul style="list-style-type: none">• [[noreturn]] attribute marks function in C++ that does not return with a value or implicitly after completion. It may throw. <pre>#include <cstdio> using namespace std; /* Nothing to return */ [[noreturn]] void show() { printf("BYE BYE"); } int main() { printf("Ready to begin..."); show(); printf("NOT over till now"); }</pre> <p>Compiler Warning: 'noreturn' function does return</p> <p>Output: Ready to begin...BYE BYE</p> <p>Source: C++ attribute: noreturn Accessed 14-Sep-21</p>

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)

C Feature	C++ Feature
<ul style="list-style-type: none"> Extra C++ reserved words may fail C codes in C++ compiler. The following code works fine in C. <pre> struct template { int new; struct template* class; }; </pre>	<ul style="list-style-type: none"> It naturally fails in C++: <pre> struct template { // template is a reserved word int new; // new is a reserved word struct template *class; // class is a reserved word }; </pre>
<ul style="list-style-type: none"> We can observe several mixed effects in C and C++ due to incompatibility where the code compiles in both languages but behaves differently <pre> #include <stdio.h> extern int T; int size(void) { struct T { int i; int j; }; return sizeof(T); /* C: return sizeof(int) */ } int main() { printf("%d", size()); } </pre>	<ul style="list-style-type: none"> It naturally fails in C++: <pre> #include <stdio> using namespace std; extern int T; int size(void) { struct T { int i; int j; }; return sizeof(T); // C++: return sizeof(struct T) } int main() { printf("%d", size()); } </pre>

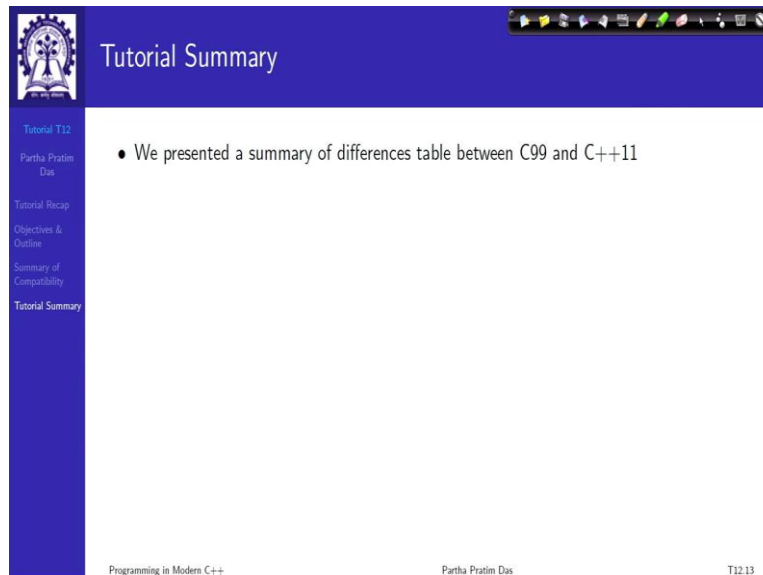
C Feature	C++ Feature
<ul style="list-style-type: none"> Extra C++ reserved words may fail C codes in C++ compiler. The following code works fine in C. <pre> struct template { int new; struct template* class; }; </pre>	<ul style="list-style-type: none"> It naturally fails in C++: <pre> struct template { // template is a reserved word int new; // new is a reserved word struct template *class; // class is a reserved word }; </pre>
<ul style="list-style-type: none"> We can observe several mixed effects in C and C++ due to incompatibility where the code compiles in both languages but behaves differently <pre> #include <stdio.h> extern int T; int size(void) { struct T { int i; int j; }; return sizeof(T); /* C: return sizeof(int) */ } int main() { printf("%d", size()); } </pre>	<ul style="list-style-type: none"> It naturally fails in C++: <pre> #include <stdio> using namespace std; extern int T; int size(void) { struct T { int i; int j; }; return sizeof(T); // C++: return sizeof(struct T) } int main() { printf("%d", size()); } </pre>

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)



The image shows a presentation slide with a blue header and footer. The header contains the text "Tutorial Summary" and a small logo on the left. The main content area is white and contains a single bullet point: "• We presented a summary of differences table between C99 and C++11". The footer contains the text "Programming in Modern C++", "Partha Pratim Das", and "T12.13".

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.