Programming in Modern C++

Tutorial T12: Compatibility of C and C++: Part 2: Summary

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

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++

Tutorial Objectives

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

Tutorial Outline

1. Tutorial Recap

2. Summary of Compatibility

3. Tutorial Summary

Summary of Compatibility

Summary of Compatibility

We summarize the incompatibility in features already discussed, and also introduce a few new ones in brief

Compatibility of C and C++: Summary

C Feature	C++ Feature
• **Implicit Conversion of void* ** is allowed in C	• Implicit conversion is not allowed in C++; allowed only with explicit cast
• **Implicit Discard of const** qualifier for pointer is allowed in C	• Implicit discard is not allowed in C++; allowed only with explicit cast
• **Initialization of const Variable** is optional in C	• Initialization is mandatory in C++
• **C Standard Library** functions have unique signature. For example, strchr in string.h	• In C++, they may have additional overloaded functions. For example, strchr in cstring
• **Implicit Conversion of int to enum** is allowed in C	• Implicit conversion is not allowed in C++; allowed only with explicit cast
• enum **Enumerators** are of type int in C	• Enumerators are of distinct types in C++, having different size from int
• **Multiple definitions of a global in a single translation unit** is allowed in C	• It is disallowed in C++ due to One Definition Rule (ODR)
• **Declaring a new type having same name as an existing struct, union or enum** is allowed in C	• It is disallowed in C++ as all declarations of such types carry the typedef implicitly
• In C, **a function prototype without parameters implies that the parameters are unspecified**, and can be called with zero or more parameters	• In C++, it means zero parameter only

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

Compatibility of C and C++: Summary

C Feature	C++ Feature
• Character literals like ‘a’ are of type int in C. Hence: 1. sizeof(’a’) = sizeof(int) 2. ’a’ is always a signed expression, regardless of whether or not char is a signed or unsigned type	• They are of type char in C++. Hence: 1. sizeof(’a’) = sizeof(char) = 1 2. If ’a’ a signed expression or not depends on whether char is a signed or unsigned type, which is implementation specific
• Boolean type bool is supported in C99 with constants true and false. In C99, a new keyword, _Bool, is introduced as the new Boolean type. The header stdbool.h provides macros bool, true and false that are defined as _Bool, 1 and 0, respectively. Therefore, true and false have type int in C	• In C++, bool is a built-in type with constants true and false. All these are reserved keywords. Conversions to bool are similar to C
• For Nested stucts, the inner struct is also defined outside the outer struct in C	A nested struct is defined only within the scope / namespace of the outer struct in C++

Tutorial T12

Partha Pratim Das

Tutorial Recap

Objectives & Outline

Summary of Compatibility

Tutorial Summary

Compatibility of C and C++: Summary

C Feature

- inline **Function** 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:

```
#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);
}
```

It gives a linker error `undefined reference to 'foo'` - as GCC `inline`s, there is no function call present (`foo`) inside `main`. Hence, we fix as:

```
#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);
}
```

C++ Feature

- In C++, the external linkage issues of `inline` functions are handled by the compiler:

```
#include <cstdio>
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);
}
```

Source: inline function specifier Accessed 14-Sep-21

Compatibility of C and C++: Summary

C Feature	C++ Feature
• **Variable Length Array (VLA)** is supported from **C99**	• Supported if array size is a constant-expression in **C++11** standard and simple expression (not constant-expression) in **C++14** standard
• **Flexible Array Member (FAM)** is supported from C99	• Not supported in ISO C++
• **restrict type qualifier** for pointer declarations is supported from **C99**	• Not supported in ISO C++, but compilers like GCC, Visual C++, and Intel C++ provide similar functionality as an extension
• **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	• 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)

• **Array parameter qualifiers** in functions is supported from **C89**:
`int foo(int a[const]);`
`// equivalent to int *const a`
`int bar(char s[static 5]);`
`// s is at least 5 chars long`
	• Not supported in ISO C++

Compatibility of C and C++: Summary

C Feature	C++ Feature

• From **C89**, **Compound Literals** is generalized to both built-in and user-defined types by the list initialization syntax of **C++11**, *although with some syntactic and semantic differences*:
`struct X { int p, q; };`
`/* Equivalent in C++ would be X{4, 6} */`
`struct X a = (struct X){4, 6};`
	• The following works in **C++11** onward:
	`struct X { int p, q; };`
	`struct X a = (struct X){4, 6};`
	`struct X b = X{4, 6};`

• 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:
`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 };`
	• These are not allowed in C++. struct designated initializers are planned for addition in C++2x
	
	`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 };`

Compatibility of C and C++: Summary

C Feature

- _Noreturn function specifier marks function in C that does not return with a value or implicitly after completion. It may return by executing longjmp:

```
#include <stdio.h>


/* Nothing to return */
_Noreturn void show() { printf("BYE BYE"); }
int main() {
    printf("Ready to begin...");
    show();

    printf("NOT over till now");
}

Compiler Warning:
'noreturn' function does return

Output is:
Ready to begin...BYE BYE
```

Source: _Noreturn function specifier Accessed 14-Sep-21

C++ Feature

- [[noreturn]] attribute marks function in C++ that does not return with a value or implicitly after completion. It may throw:

```
#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");
}

Compiler Warning:
'noreturn' function does return

Output:
Ready to begin...BYE BYE
```

Source: C++ attribute: noreturn Accessed 14-Sep-21

This must not be confused with `void` *return type used for functions that return, but without a value*

Compatibility of C and C++: Summary

C Feature	C++ Feature

• **Extra C++ reserved words** may fail C codes in C++ compiler. The following code works fine in C.
`struct template {`
`    int new;`
`    struct template* class;`
`};`
	• It naturally fails in C++:
	`struct template { // template is a reserved word`
	`    int new; // new is a reserved word`
	`    struct template*`
	`        class; // class is a reserved word`
	`};`

• We can observe **several mixed effects** in C and C++ due to incompatibility where the code compiles in both languages but behaves differently
`#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()); }`
	• It naturally fails in C++:
	`#include <cstdio>`
	`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()); }`

Tutorial Summary

- We presented a summary of differences table between **C99** and **C++11**