Programming in Modern C++

Tutorial T11: Compatibility of C and C++: Part 1: Significant Features

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 Objectives

- We often say that "C is a subset of C++". It is far from truth. There are various intra-dialect incompatibilities in C (**C89**, **C99**, **C11**), in C++ (**C++03**, **C++11**, **C++14**, **C++17**, ...), and inter-dialect incompatibilities across languages. We need to understand these differences and their effect in the programs we write
- We take a look at the C/C++ communities and consider views of different sections of communities to understand why we need the compatibility - at least, the clear understanding for it
- We discuss the major compatibility issues between C and C++. To keep the discussion manageable, we primarily focus between **C99** and **C++11**
- We also discuss the workarounds to write more compatible code between C and C++

Tutorial Outline

1. Why is Compatibility of C and C++ important?
2. Compatibility of C and C++
   - `void*`
   - `const`
   - `enum`
   - ODR
   - `void` Param
   - Nested struct
   - VLA
   - FAM
   - `restrict`
3. Tutorial Summary

Why is Compatibility of C and C++ important?

Why is Compatibility of C and C++ important?

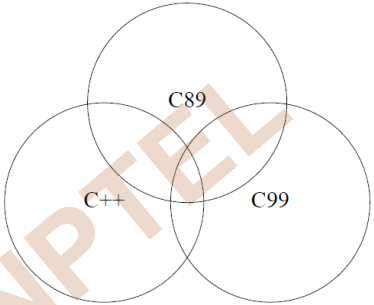
Source: The C/C++ Users Journal, Jul-Aug-Sep, 2002. Accessed 15-Sep-21
C and C++: Siblings, B. Stroustrup
C and C++: A Case for Compatibility, B. Stroustrup
C and C++: Case Studies in Compatibility, B. Stroustrup

Why is Compatibility of C and C++ important?

- The *C and C++ programming languages are closely related* but have *many significant differences*
- **There is no C/C++ language, but there is a C/C++ community**. Millions of programmers and organization who use C and/or C++ form the community comprising three major groups:
  - *Programmers who use C only*: Especially the embedded systems community. Many programmers working with C programs that never call a C++ library. However, most (?) C programmers occasionally use C++ directly and many rely on C++ libraries. Hence, the C programmer must be aware of C++ in the same way as a C++ programmer must be aware of C.
  - *Programmers who use C++ only*: Is it possible? Most programmers would need to call a C library. Hence, the programmer needs to understand the constructs in its header files - use of malloc() rather than new, the use of arrays rather than C++ standard library containers, and the absence of exception handling. So all C++ programmers are C programmers.
  - *Programmers who use both C and C++*:
- Compatibility maximizes the community of contributors. Each dialect and incompatibility limits the
  - market for vendors/suppliers/builders
  - set of libraries and tools for users - single product (IDE, compiler, analyzer, etc.) for both languages
  - set of collaborators (suitable employees, students, consultants, experts, etc.) for projects

Why is Compatibility of C and C++ important?

- Bjarne Stroustrup, the creator of C++, has suggested that *the incompatibilities between C and C++ should be reduced as much as possible in order to maximize interoperability between the two languages*
- Others argue that C and C++ are two different languages - *compatibility between them is useful but not vital*; and efforts to reduce incompatibility *should not hinder improvement of each language in isolation*
- **C99** "*endorse[d] the principle of maintaining the largest common subset*" between C and C++ "*while maintaining a distinction between them and allowing them to evolve separately*", and stated that the authors were "*content to let C++ be the big and ambitious language*"
- Several additions of **C99** are *not supported in the current C++ standard or conflicted with C++ features*, such as *variable-length arrays*, native *complex number* types and the *restrict* type qualifier
- On the other hand, **C99** *reduced some other incompatibilities compared with* **C89** by incorporating C++ features such as *// comments* and *mixed declarations and code*



Feature compatibility between C++98, C89, and C99. "There are features in all 7 areas" - C and C++: Siblings, B. Stroustrup, 2002

Compatibility of C and C++

Source: Accessed 15-Sep-21
C and C++: A Case for Compatibility, B. Stroustrup
C and C++: Case Studies in Compatibility, B. Stroustrup
Compatibility of C and C++, HandWiki
Compatibility of C and C++, Wikipedia
Annex C.1 of the ISO C++ standard

Compatibility of C and C++

- *C is not a subset of C++, and nontrivial C programs will not compile as C++ code without change*
- *Likewise, C++ introduces many features that are not available in C and in practice almost all code written in C++ is not conforming C code*
- Here, we focus on differences that cause **conforming C code to be ill-formed C++ code**, or to be **conforming / well-formed in both languages but to behave differently in C and C++**
- We take following approach for the discussions:
  - To explain the compatibility, incompatibility and work-around in an understandable way, we write the same code in `main.c` and `main.cpp` and compile with `gcc` to get the language specific behavior
  - We also use dialect specific `-std` flags wherever relevant. Most comparisons are done with respect to **C99** and **C++11**
  - We present the compiler messages and / or output to elucidate the effects
  - We present a summary of the compatibility issues at the end in comparative tabular form

Compatibility of C and C++: `void*`

- One commonly encountered difference is C being more weakly-typed regarding pointers
- **Specifically, C allows a `void*` pointer to be assigned to any pointer type without a cast, while C++ does not**
- This idiom appears often in C code using `malloc` memory allocation, or in the passing of context pointers to the `POSIX pthreads API`, and other frameworks involving `callbacks`
- For example, the following is valid in C but not C++:

```
void *ptr;
/* Implicit conversion from void* to int* */
int *i = ptr;
```

or similarly:

```
int *j = malloc(5 * sizeof *j); /* Implicit conversion from void* to int* */
```

In order to make the code compile as both C and C++, one must use an explicit cast, as follows (with some caveats in both languages)

```
void *ptr;
int *i = (int *)ptr;
int *j = (int *)malloc(5 * sizeof *j);
```

Compatibility of C and C++: const

- C++ is also more strict than C about pointer assignments that discard a `const` qualifier. For example, assigning a `const int*` value to an `int*` variable:

```
int main() { const int* p = 0;
    int* q = p; // const qualifier being discarded
}
```

In C++, this is invalid and generates a compiler error (unless an explicit typecast is used), while in C this is allowed (although many compilers emit a warning)

```
$ gcc main.cpp main.c

main.cpp:3:14:  error: invalid conversion from 'const int*' to 'int*' [-fpermissive]
     int* q = p;
              ^

main.c:3:14:  warning: initialization discards 'const' qualifier from pointer target type
                       [-Wdiscarded-qualifiers]
     int* q = p;
              ^
```

- In C++ a const variable must be initialized; in C this is not necessary. For

```
int main() { const int i = 5;
    const int j; // const variable not initialized
}
$ gcc main.cpp main.c

main.cpp:12:15:  error: uninitialized const 'j' [-fpermissive]
     const int j;
               ^
```

Compatibility of C and C++: string.h and enum

- C++ changes some C standard library functions to add additional overloaded functions with `const` type qualifiers, for example, consider `strchr()` function in `string.h` in C and `cstring` in C++

```
// string.h
char *strchr(const char *str, int character)
// cstring
const char *strchr(const char * str, int character);
char *strchr (char * str, int character);
```

  So when a C file is compiled with C++ compiler different calls to `strchr()` may bind to different overloads in C++

- C++ is also more strict in conversions to `enum`s: `int`s cannot be implicitly converted to `enum`s as in C.

```
enum week { Mon, Tue, Wed, Thur, Fri, Sat, Sun };
int main() { enum week day;
    int dayindex = 2;
    day = dayindex;
}

$ gcc main.c main.cpp

main.cpp:23:11:  error: invalid conversion from 'int' to 'week' [-fpermissive]
     day = dayindex;
           ^~~~~~~~
```

- Also, Enumeration constants (`enum` enumerators) are always of type `int` in C, whereas they are distinct types in C++ and may have a size different from that of `int`

Compatibility of C and C++: One Definition Rule (ODR)

- C allows for multiple tentative definitions of a single global variable in a *single translation unit*, which is disallowed as an *One Definition Rule (ODR)* violation in C++

```
int N;
int N = 10;

$ gcc main.c main.cpp
main.cpp:46:5:  error: redefinition of 'int N'
 int N = 10;
     ^
main.cpp:45:5:  note: 'int N' previously declared here
 int N;
     ^
```

- C allows declaring a new type with the same name as an existing `struct`, `union` or `enum` which is not allowed in C++, as in C `struct`, `union` or `enum` types must be indicated as such whenever the type is referenced whereas in C++ all declarations of such types carry the `typedef` implicitly

```
enum BOOL { FALSE, TRUE };
typedef struct _BOOL { int b; } BOOL;

$ gcc main.c main.cpp
main.cpp:53:33:  error: conflicting declaration 'typedef struct _BOOL BOOL'
 typedef struct _BOOL { int b; } BOOL;
                                 ^~~~
main.cpp:52:6:  note: previous declaration as 'enum BOOL'
 enum BOOL { FALSE, TRUE };
      ^~~~
```

Compatibility of C and C++: void Parameter

- In C, a function prototype without parameters, for example, `int foo();`, implies that the parameters are unspecified. Therefore, it is legal to call such a function with one or more arguments, like `foo(0)`
- In contrast, in C++ a function prototype without arguments means that the function takes no arguments, and calling such a function with arguments is ill-formed
- **In C, declare a function taking no argument by using `void`, as in `int foo(void);`, which is also valid in C++. Empty function prototypes are a deprecated feature in C99 (as they were in C89)**

```
int foo(); int bar(void);
int main() { foo(0); bar(0); }
$ gcc main.c main.cpp
main.c:42:22:  error: too many arguments to function 'bar'
 int main() { foo(0); bar(0); }
                      ~~~
main.c:41:16:  note: declared here: int foo(); int bar(void);
                                          ~~~

main.cpp:59:19:  error: too many arguments to function 'int foo()'
 int main() { foo(0); bar(0); }
                   ^
main.cpp:58:5:  note: declared here: int foo(); int bar(void);
                                        ~~~

main.cpp:59:27:  error: too many arguments to function 'int bar()'
 int main() { foo(0); bar(0); }
                           ^
main.cpp:58:16:  note: declared here: int foo(); int bar(void);
                    ~~~
```

Compatibility of C and C++: Nested struct

- In both C and C++, one can define nested `struct` types, but the scope is interpreted differently
  - In C++, a nested `struct` is defined only within the scope / namespace of the outer `struct`
  - In C the inner `struct` is also defined outside the outer `struct`

```
struct Outer {
    int o;
    struct Inner {
        int i;
    };
};

struct Outer O1;        // Okay in C and C++

#ifndef __cplusplus
struct Inner I1;        // Okay only in C
#endif
```

Compatibility of C and C++: Variable Length Array (VLA)

- **Variable Length Arrays (VLA)** is a feature where we can allocate an auto array (on stack) of variable size. C supports variable sized arrays from C99 standard
- But, in C++ standard (till **C++11**) there was no concept of VLA. According to the **C++11** standard, array size is a constant-expression. In **C++14** mentions array size as a simple expression (not constant-expression)

```
#ifndef __cplusplus
#include <stdio.h>
// VLA in function prototype
int add(int x, int a[*]);
// int add(int x, int a[]); also works
#else
#include <cstdio>
using namespace std;
// Unspecified size in function prototype
int add(int x, int a[]);
#endif
```

```
int set_and_add(int n) { int vals[n]; // Variable Length Array
    printf("%d ", sizeof(vals));      // Runtime sizeof
    for (int i = 0; i < n; ++i) vals[i] = i;
    return add(n, vals);
    // vals is declared as an automatic variable
    // its lifetime ends when add() returns
}
int main() { int n = 5;
    printf("Result = %d", set_and_add(n));
}
int add(int n, int a[]) { int sum = 0;
    for (int i = 0; i < n; ++i) sum += a[i];
    return sum;
}
```

- The above code uses VLA (`int vals[n]`)) in function `set_and_add`. So any size (bounded by a compiler-specified maximum) can be passed to it
- VLA may lead to possibly non-compile time `sizeof` operator

Compatibility of C and C++: Flexible Array Member (FAM)

- The last member of a **C99** structure type with more than one member may be a **Flexible Array Member (FAM)**, which takes the syntactic form of an array with unspecified length. This serves a purpose similar to variable-length arrays
- VLAs cannot appear in type definitions, but has defined size (at runtime)
- FAMs have no defined size, but can appear in type definitions
- **ISO C++ has no such feature**
- Here is an example of a FAM

```
struct vectord {
    short len;      // there must be at least one other data member
    double arr[];   // the flexible array member must be last
                    // The compiler may reserve extra padding space here,
                    // like it can between struct members
};
```

- Typically, such structures serve as the header in a larger, variable memory allocation

```
struct vectord *vector = malloc(...);
vector->len = ...;
for (int i = 0; i < vector->len; i++)
    vector->arr[i] = ...;  // transparently uses the right type (double)
```

Compatibility of C and C++: restrict

- `restrict` keyword is mainly used in pointer declarations as a type qualifier for pointers
- It adds no functionality - only informs the compiler about an optimization
- When we use `restrict` with a pointer `ptr`, it tells the compiler that `ptr` *is the only way to access the object pointed by it*, in other words, *there is no other pointer pointing to the same object*. That is, `restrict` keyword *specifies that a particular pointer argument does not alias any other* and the *compiler does not need to add any additional checks*
- If a programmer uses `restrict` keyword and violate the above condition, the behavior is undefined
- `restrict` is supported from C99. **It not supported by ISO C++**

```
#include <stdio.h>
// The purpose of restrict is to show only syntax. It does not change anything in output (or logic)
// It is just a way for programmer to tell compiler about an optimization
void use(int* a, int* b, int* restrict c) {
    *a += *c;
    // Since c is restrict, compiler will not reload value at address c in its assembly code
    // Therefore generated assembly code is optimized
    *b += *c;
}
int main(void) { int a = 50, b = 60, c = 70;
    use(&a, &b, &c);
    printf("%d %d %d", a, b, c);
}
```

Source: restrict keyword in C and How to Use the restrict Qualifier in C Accessed 15-Sep-21

Tutorial Summary

- 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++