Compatibility of C and C++

The C and C++ programming languages are closely related but have many significant differences. C++ began as a fork of an early, pre-standardized C, and was designed to be mostly source-and-link compatible with C compilers of the time.[1][2] Due to this, development tools for the two languages (such as IDEs and compilers) are often integrated into a single product, with the programmer able to specify C or C++ as their source language.

However, C is **not** a subset of C++,[3] and nontrivial C programs will not compile as C++ code without modification. 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. This article, however, focuses 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++.

Bjarne Stroustrup, the creator of C++, has suggested[4] that the incompatibilities between C and C++ should be reduced as much as possible in order to maximize interoperability between the two languages. Others have argued that since C and C++ are two different languages, compatibility between them is useful but not vital; according to this camp, efforts to reduce incompatibility should not hinder attempts to improve each language in isolation. The official rationale for the 1999 C standard (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."[5]

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.[6]

Constructs valid in C but not in C++

C++ enforces stricter typing rules (no implicit violations of the static type system[1]), and initialization requirements (compile-time enforcement that in-scope variables do not have initialization subverted)[7] than C, and so some valid C code is disallowed in C++. A rationale for these is provided in Annex C.1 of the ISO C++ standard.[8]

- 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,[9] 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[10] [11]):

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

- C++ is also more strict than C about pointer assignments that discard a `const` qualifier (e.g. assigning a `const int*` value to an `int*` variable): in C++ this is invalid and generates a compiler error (unless an

explicit typecast is used),[12] whereas in C this is allowed (although many compilers emit a warning).

- C++ changes some C standard library functions to add additional overloaded functions with `const` type qualifiers, e.g. `strchr` returns `char*` in C, while C++ acts as if there were two overloaded functions `const char *strchr(const char *)` and a `char *strchr(char *)`.
- C++ is also more strict in conversions to enums: ints cannot be implicitly converted to enums as in C. 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`.
- In C++ a `const` variable must be initialized; in C this is not necessary.
- C++ compilers prohibit goto or switch from crossing an initialization, as in the following C99 code:

```
void fn(void)
{
    goto flack;
    int i = 1;
flack:
    ;
}
```

- While syntactically valid, a `longjmp()` results in undefined behaviour in C++ if the jumped-over stack frames include objects with nontrivial destructors.[13] The C++ implementation is free to define the behaviour such that destructors would be called. However, this would preclude some uses of longjmp() which would otherwise be valid, such as implementation of threads or coroutines by longjmping between separate call stacks - when jumping from the lower to the upper call stack in global address space, destructors would be called for every object in the lower call stack. No such issue exists in C.
- C allows for multiple tentative definitions of a single global variable in a single translation unit, which is disallowed as an ODR violation in C++.

```
int N;
int N = 10;
```

- 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`, and `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 int BOOL;
```

- Non-prototype ("K&R"-style) function declarations are not allowed in C++; they are still allowed in C,[14] although they have been deemed obsolescent since C's original standardization in 1990. (The term "obsolescent" is a defined term in the ISO C standard, meaning a feature that "may be considered for withdrawal in future revisions" of the standard.) Similarly, implicit function declarations (using functions that have not been declared) are not allowed in C++, and have been disallowed in C since 1999.
- In C, a function prototype without parameters, e.g. `int foo();`, implies that the parameters are unspecified. Therefore, it is legal to call such a function with one or more arguments, e.g. `foo(42, "hello world")`. 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, the correct way to declare a function that takes no arguments is 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).
- 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`, whereas in C the inner struct is also defined outside the outer struct.
- C allows `struct`, `union`, and `enum` types to be declared in function prototypes, whereas C++ does not.

C99 and C11 added several additional features to C that have not been incorporated into standard C++, such as complex numbers, variable length arrays (note that complex numbers and variable length arrays are designated as optional extensions in C11), flexible array members, the restrict keyword, array parameter qualifiers, compound literals, and designated initializers.

- 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, however.)[15]
- Variable length arrays. This feature leads to possibly non-compile time `sizeof` operator.[16]

```
void foo(size_t x, int a[*]);  // VLA declaration
void foo(size_t x, int a[x])
{
    printf("%zu\n", sizeof a); // same as sizeof(int*)
    char s[x * 2];
    printf("%zu\n", sizeof s); // will print x*2
}
```

- The last member of a C99 structure type with more than one member may be a flexible array member, which takes the syntactic form of an array with unspecified length. This serves a purpose similar to variable-length arrays, but VLAs cannot appear in type definitions, and unlike VLAs, flexible array members have no defined size. ISO C++ has no such feature. Example:

```
struct X
{
    int n, m;
    char bytes[];
}
```

- The `restrict` type qualifier defined in C99 was not included in the C++03 standard, but most mainstream compilers such as the GNU Compiler Collection,[17] Microsoft Visual C++, and Intel C++ Compiler provide similar functionality as an extension.
- Array parameter qualifiers in functions are supported in C but not C++.

```
int foo(int a[const]);     // equivalent to int *const a
```

```
int bar(char s[static 5]); // annotates that s is at least 5 chars long
```

- The functionality of compound literals in C 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 a = (struct X){4, 6};  // The equivalent in C++ would be X{4, 6}. The
```

- Designated initializers for structs and arrays are valid only in C, although struct designated initializers are planned for addition in C++2x:

```
struct X a = {.n = 4, .m = 6};   // to be allowed in C++2x (requires order of i
char s[20] = {[0] = 'a', [8] = 'g'};  // allowed in C, not allowed in C++ (nor C
```

- Functions that do not return can be annotated using a noreturn attribute in C++ whereas C uses a distinct keyword.

C++ adds numerous additional keywords to support its new features. This renders C code using those keywords for identifiers invalid in C++. For example:

```
struct template
{
    int new;
    struct template* class;
};
```

is valid C code, but is rejected by a C++ compiler, since the keywords "template", "new" and "class" are reserved.

Constructs that behave differently in C and C++

There are a few syntactical constructs that are valid in both C and C++ but produce different results in the two languages.

- Character literals such as `'a'` are of type `int` in C and of type `char` in C++, which means that `sizeof 'a'` will generally give different results in the two languages: in C++, it will be `1`, while in C it will be

`sizeof(int)`. As another consequence of this type difference, in C, `'a'` will always be a signed expression, regardless of whether or not `char` is a signed or unsigned type, whereas for C++ this is compiler implementation specific.

- C++ assigns internal linkage to namespace-scoped `const` variables unless they are explicitly declared `extern`, unlike C in which `extern` is the default for all file-scoped entities. Note that in practice this does not lead to silent semantic changes between identical C and C++ code but instead will lead to a compile-time or linkage error.
- In C use of inline functions requires manually adding a prototype declaration of the function using the extern keyword in exactly one translation unit to ensure a non-inlined version is linked in, whereas C++ handles this automatically. In more detail, C distinguishes two kinds of definitions of `inline` functions: ordinary external definitions (where **extern** is explicitly used) and inline definitions. C++, on the other hand, provides only inline definitions for inline functions. In C, an inline definition is similar to an internal (i.e. static) one, in that it can coexist in the same program with one external definition and any number of internal and inline definitions of the same function in other translation units, all of which can differ. This is a separate consideration from the *linkage* of the function, but not an independent one. C compilers are afforded the discretion to choose between using inline and external definitions of the same function when both are visible. C++, however, requires that if a function with external linkage is declared **inline** in any translation unit then it must be so declared (and therefore also defined) in every translation unit where it is used, and that all the definitions of that function be identical, following the ODR. Note that static inline functions behave identically in C and C++.
- Both C99 and C++ have a boolean type `bool` with constants `true` and `false`, but they are defined differently. In C++, `bool` is a built-in type and a reserved keyword. 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.

Several of the other differences from the previous section can also be exploited to create code that compiles in both languages but behaves

differently. For example, the following function will return different values in C and C++:

```
extern int T;

int size(void)
{
    struct T {  int i;  int j;  };

    return sizeof(T);
    /* C:   return sizeof(int)
     * C++: return sizeof(struct T)
     */
}
```

This is due to C requiring `struct` in front of structure tags (and so `sizeof(T)` refers to the variable), but C++ allowing it to be omitted (and so `sizeof(T)` refers to the implicit `typedef`). Beware that the outcome is different when the `extern` declaration is placed inside the function: then the presence of an identifier with same name in the function scope inhibits the implicit `typedef` to take effect for C++, and the outcome for C and C++ would be the same. Observe also that the ambiguity in the example above is due to the use of the parenthesis with the `sizeof` operator. Using `sizeof T` would expect `T` to be an expression and not a type, and thus the example would not compile with C++.

Linking C and C++ code

While C and C++ maintain a large degree of source compatibility, the object files their respective compilers produce can have important differences that manifest themselves when intermixing C and C++ code. Notably:

- C compilers do not name mangle symbols in the way that C++ compilers do.[18]
- Depending on the compiler and architecture, it also may be the case that calling conventions differ between the two languages.

For these reasons, for C++ code to call a C function `foo()`, the C++ code must prototype `foo()` with `extern "C"`. Likewise, for C code to call a C++ function `bar()`, the C++ code for `bar()` must be declared with `extern "C"`.

A common practice for header files to maintain both C and C++ compatibility is to make its declaration be `extern "C"` for the scope of the header:[19]

```
/* Header file foo.h */
#ifdef __cplusplus /* If this is a C++ compiler, use C linkage */
extern "C" {
#endif

/* These functions get C linkage */
void foo();

struct bar { /* ... */ };

#ifdef __cplusplus /* If this is a C++ compiler, end C linkage */
}
#endif
```

Differences between C and C++ linkage and calling conventions can also have subtle implications for code that uses function pointers. Some compilers will produce non-working code if a function pointer declared `extern "C"` points to a C++ function that is not declared `extern "C"`.[20]

For example, the following code:

```
1 void my_function();
2 extern "C" void foo(void (*fn_ptr)(void));
3
4 void bar()
5 {
6     foo(my_function);
7 }
```

Using Sun Microsystems' C++ compiler, this produces the following warning:

```
$ CC -c test.cc
"test.cc", line 6: Warning (Anachronism): Formal argument fn_ptr of type
extern "C" void(*)() in call to foo(extern "C" void(*)()) is being passed
void(*)().
```

This is because `my_function()` is not declared with C linkage and calling conventions, but is being passed to the C function `foo()`.

References

1. ^ [a] [b] Stroustrup, Bjarne. "An Overview of the C++ Programming Language in The Handbook of Object Technology (Editor: Saba Zamir). CRC Press LLC, Boca Raton. 1999. ISBN 0-8493-3135-8" (PDF). p. 4. Archived (PDF) from the original on 16 August 2012. Retrieved 12 August 2009.
2. ^ B.Stroustrup. "C and C++: Siblings. The C/C++ Users Journal. July 2002" (PDF). Retrieved 17 March 2019.
3. ^ "Bjarne Stroustrup's FAQ – Is C a subset of C++?". Retrieved 22 September 2019.
4. ^ B. Stroustrup. "C and C++: A Case for Compatibility. The C/C++ Users Journal. August 2002" (PDF). Archived (PDF) from the original on 22 July 2012. Retrieved 18 August 2013.
5. ^ Rationale for International Standard—Programming Languages—C Archived 6 June 2016 at the Wayback Machine, revision 5.10 (April 2003).
6. ^ "C Dialect Options - Using the GNU Compiler Collection (GCC)". *gnu.org*. Archived from the original on 26 March 2014.
7. ^ "N4659: Working Draft, Standard for Programming Language C++" (PDF). §Annex C.1. Archived (PDF) from the original on 7 December 2017. ("It is invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered). ... With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.")
8. ^ "N4659: Working Draft, Standard for Programming Language C++" (PDF). §Annex C.1. Archived (PDF) from the original on 7 December 2017.
9. ^ "IBM Knowledge Center". *ibm.com*.
10. ^ "FAQ > Casting malloc - Cprogramming.com". *faq.cprogramming.com*. Archived from the original on 5 April 2007.
11. ^ "4.4a — Explicit type conversion (casting)". 16 April 2015. Archived from the original on 25 September 2016.

12. ^ "Const correctness, C++ FAQ". Parashift.com. 4 July 2012. Archived from the original on 5 August 2013. Retrieved 18 August 2013.
13. ^ "longjmp - C++ Reference". *www.cplusplus.com*. Archived from the original on 19 May 2018.
14. ^ "2011 ISO C draft standard" (PDF).
15. ^ "std::complex - cppreference.com". *en.cppreference.com*. Archived from the original on 15 July 2017.
16. ^ "Incompatibilities Between ISO C and ISO C++". Archived from the original on 9 April 2006.
17. ^ Restricted Pointers Archived 6 August 2016 at the Wayback Machine from *Using the GNU Compiler Collection (GCC)*
18. ^ "IBM Knowledge Center". *ibm.com*.
19. ^ "IBM Knowledge Center". *ibm.com*.
20. ^ "Oracle Documentation". Docs.sun.com. Archived from the original on 3 April 2009. Retrieved 18 August 2013.

External links

- Detailed comparison, sentence by sentence, from a C89 Standard perspective.
- Incompatibilities Between ISO C and ISO C++, David R. Tribble (August 2001).
- Oracle (Sun Microsystems) C++ Migration Guide, section 3.11, Oracle/Sun compiler docs on linkage scope.
- Oracle: Mixing C and C++ Code in the Same Program, overview by Steve Clamage (ANSI C++ Committee chair).

By: Wikipedia.org
Edited: 2021-06-18 15:16:58
Source: Wikipedia.org