C xor C++ Programming

WG21 Document Number: P2735R0
WG14 Document Number: N3065
Date: 2022-12-05
Author, Affiliation: Aaron Ballman <aaron@aaronballman.com>, Intel
Audience: SG22 C and C++ Compatibility Study Group

Summary of Changes

R0/N3065

- Original proposal

Introduction and Rationale

It is not uncommon to hear about C/C++ programming as a shorthand for "C and C++" programming. This implies that C and C++ are similar, but distinct, programming languages with the obvious interpretation being that C++ is a proper superset of C. However, this does not accurately describe the situation. The C++ programming language is inspired by the C programming language and supports much of the syntax and semantics of C, but is not a superset that is built on top of C. Despite sharing a historical relationship to one another, the languages have evolved independently and are specified in separate language standards. Due to this separation of the two specifications, incompatibilities have crept into the shared space of code that can be compiled by either a C compiler or a C++ compiler.

This document enumerates instances where the same source code has different meaning when compiled with C and C++ implementations. Such source code is often a pain point for users and implementers because it represents a "sharp edge" in both languages, especially if the code appears in a header file that may be compiled in separate C and C++ translation units. These sharp edges are areas where either committee may be interested in exploring unification efforts.

This document explicitly does not cover situations where one language has a feature that the other language does not have, as the two languages are intentionally distinct and so these kinds of incompatibilities are to be expected. It also does not cover incompatibilities between the standard library functionality.

Each incompatibility has a stable name (a name in square brackets) to make it easier to refer to a particular instance of incompatibility, contrived code examples in each language demonstrating the incompatibility, a link to an online compiler demonstrating the behaviors shown, and an explanation of the incompatibility with citations from the standards.

List of Incompatibilities

[type.bitfield]	
C	C++

struct S {
  int i : 1;
} s;
 	
sizeof(1, s.i); // OK, sizeof(int)
	struct S {
	  int i : 1;
	} s;
	
	sizeof(1, s.i); // Error

https://godbolt.org/z/nsjjjK	
In C++, the result of a comma operator is (effectively) the right-hand operand. If the right-hand operand is a bit-field, then the result is a bit-field. In C, the result is the value and type of the right-hand operand, but the operand undergoes lvalue conversion and the resulting type is the type of the bit-field (int). Note: 6.7.2.1p12 says "A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits." which suggests that the resulting type after lvalue conversion perhaps should retain that it's a bit-field.	

[comma.value-category]	
C	C++
int i = 0; (1, i) = 12; // Error	int i = 0; (1, i) = 12; // OK, assigns to i
https://godbolt.org/z/WoTK5x	
In C++, the result of a comma operator is (effectively) the right-hand operand. In C, the result is the value and type of the right-hand operand but does not have the same value category.	

	[assign.value-category]
C	C++

`int i = 0;`
`(i = 12) = 11; // Error`
	`int i = 0;`
	`(i = 12) = 11; // OK, assigns to i`

https://godbolt.org/z/4ErTTr	
The result of an assignment expression in C is not an lvalue (6.5.16p3), but it is an lvalue in C++ ([expr.ass]p1).	

	[incdec.value-category]
C	C++

`int i = 0;`
`++i = 12; // Error`
	`int i = 0;`
	`++i = 12; // OK, assigns to i`

https://godbolt.org/z/Kr5s3bYaj	
The result of an increment or decrement expression in C is not an lvalue (6.5.3.1p2, 6.5.16p3), but it is an lvalue in C++ ([expr.ass]p1).	

	[conditional.value-category]
C	C++

`int i = 0;`
`0 ? 1 : i = 12;   // Error`
`(0 ? 1 : i) = 12; // Error`
	`int i = 0;`
	`0 ? 1 : i = 12;   // OK, assigns to i`
	`(0 ? 1 : i) = 12; // Error`

https://godbolt.org/z/vbG1dn	

[decl.tag]

C	C++
`struct S { int i; };`	`struct S { int i; };`
`union U { int i; float f; };`	`union U { int i; float f; };`
`enum E { One };`	`enum E { One };`
`S s; // Error`	`S s; // OK`
`U u; // Error`	`U u; // OK`
`E e; // Error`	`E e; // OK`

https://godbolt.org/z/oKqaoe

C has the notion of tag name spaces for structures, unions, and enumerations (aka, tag types). Because tag types introduce names into separate name spaces, you must specify the tag type to look up the name. C++ does not have separate tag name spaces, but does allow you to write an elaborated type specifier that includes the tag name for disambiguation purposes from other identifiers that are in scope.

[decl.bitfield-width]

C	C++
`struct S {`	`struct S {`
`  int i : 67; // Error`	`  int i : 67; // OK, extra bits are padding`
`};`	`};`

https://godbolt.org/z/Th76sr

C does not allow a bit-field to specify a width that exceeds the width of the underlying type of the declaration (6.7.2.1p4) while C++ allows it and defines the extra bits as padding ([class.bit]p1).

[decl.anon-param-type]

C

```
void func(
  struct S { int x; } s // OK
);
```

C++

```
void func(
  struct S { int x; } s // Error
);
```

https://godbolt.org/z/W6Wefj

As a natural consequence of the grammar for the language, C allows a type definition to appear anywhere a type can be specified while C++ does not.

[decl.qualified-return-type]

C

```
struct S { int i; };
const struct S func(void);
void foo(void) {
  typeof(func()) s = {12};
  s.i = 100; // OK
}
```

C++

```
struct S { int i; };
const struct S func(void);
void foo(void) {
  decltype(func()) s = {12};
  s.i = 100; // Error
}
```

https://godbolt.org/z/jT614EoYv

After the resolution of DR 423, in C the return type of a function declarator is the unqualified version of the specified return type (6.7.6.3p4). However, in C++, the return type of the function is not similarly adjusted ([dcl.fct]p1). This can be observed during redeclaration merging or in situations where the languages allow some amount of type inspection, such as _Generic in C or templates in C++.

[decl.enum-in-struct]

C

```
struct S { enum E { One } e; };
int I = One; // OK
```

C++

```
struct S { enum E { One } e; };
int I = One; // Error
```

https://godbolt.org/z/rP1W7TfsK

In C, an enumeration constant is declared in its surrounding scope (6.7.7.2p4), and a structure does not form a new scope (6.2.1p4). In C++, a structure does form a scope ([basic.scope.class]p1. Thus, in C the enumeration constant One is visible in the global scope and in C++ it is not visible and can only be accessed through S::One.

[expr.qualified-cast]

C

```
struct S { int i; };
void foo(void) {
  struct S orig;
  typeof((const struct S)orig)
    s = { 12 };
  s.i = 100; // OK
}
```

C++

```
struct S { int i; };
void foo(void) {
  S orig;
  decltype((const S)orig)
    s = { 12 };
  s.i = 100; // Error
}
```

https://godbolt.org/z/dh7xvK3hc

After the resolution of DR 423, explicit cast operations in C ignore the qualifiers specified in the cast (6.5.4p5). However, in C++, the type of the cast operation is not similarly adjusted ([expr.cast]p1). This can be observed in situations where the languages allow some amount of type inspection, such as _Generic in C or templates in C++.

	[expr.implicit-cast-from-void-ptr]
C	C++

`#include <stdlib.h>`
`int *ptr =`
`  malloc(sizeof(int)); // OK`
	`#include <stdlib.h>`
	`int *ptr =`
	`  malloc(sizeof(int)); // Error`

https://godbolt.org/z/WbE99G	
C allows a pointer to void to implicitly cast to any other pointer type (6.3.2.3p1, 6.5.16.1p1) while C++ does not.	

	[stmt.return-void]
C	C++

`void bar(void);`
`void foo(void) {`
`  return bar(); // Error`
`}`
	`void bar(void);`
	`void foo(void) {`
	`  return bar(); // OK`
	`}`

https://godbolt.org/z/4ss611	
In C, a return statement with an expression is a constraint violation if the function returns void, even if the expression used in the return statement is a void expression (6.8.6.4p1). C++ has no such restriction ([stmt.return]p2). N.B. implementations frequently allow this construct in C as a conforming extension.	

	[type.char-literal]
C	C++
`sizeof('a'); // sizeof(int)`	`sizeof('a'); // sizeof(char)`
https://godbolt.org/z/rhs7K5	
Character literals in C are of type int (6.4.4.4p11) while character literals in C++ are of type char ([lex.ccon]p1).	

[decl.non-narrow-type]

C	C++
`wchar_t a = L'a';     // Error`	`wchar_t a = L'a';     // OK`
`char16_t c16 = u'a'; // Error`	`char16_t c16 = u'a'; // OK`
`char32_t c32 = U'a'; // Error`	`char32_t c32 = U'a'; // OK`

https://godbolt.org/z/8aaavfdej

wchar_t, char16_t, and char32_t are builtin datatypes in C++ but requires including a header file in C. Also note that this means that wchar_t, char16_t, and char32_t are typedefs to integer types in C while the same is not true in C++.

[decl.empty-tag]

C	C++
`struct S {}; // Error`	`struct S {}; // OK`
`enum E {};   // Error`	`enum E {};   // OK`

https://godbolt.org/z/sjvbhe

The grammar for tag declarations (struct, enum, or union) in C does not allow a tag declaration with no members (6.7.2.1p1, 6.7.2.2p1), which is allowed by C++.

N.B. implementations frequently allow these constructs in C as a conforming extension.

[decl.anonymous-struct]

C	C++

`struct S {`
`  struct { // OK`
`    int i;`
`  };`
`  union { // OK`
`    float f;`
`    char c;`
`  };`
`  int j;`
`};`
	`struct S {`
	`  struct { // Error`
	`    int i;`
	`  };`
	`  union { // OK`
	`    float f;`
	`    char c;`
	`  };`
	`  int j;`
	`};`

https://godbolt.org/z/nrEfsT

C allows for the declaration of an anonymous structure or anonymous union type (6.7.2.1p15) while C++ only allows for the declaration of an anonymous union type ([class.union.anon]).

N.B. implementations frequently allow the declaration of an anonymous struct in C++ as a conforming extension.

[decl.str-init-without-null-term]

C	C++

`char c[4] =`
`  "asdf"; // OK, but not null terminated`
	`char c[4] =`
	`  "asdf"; // Error`

https://godbolt.org/z/ffe7xf

C++ requires there to be sufficient room for all of the initializers including the terminating null character ([dcl.init.string]p2), while C has no such requirement (6.7.9p14).

[type.non-const-str-literal]

C

```
"foo"[0] = 'b'; // Compiles, but with
                // undefined behavior
```

C++

```
"foo"[0] = 'b'; // Error
```

https://godbolt.org/z/Gz5b1P

String literals in C are of type char[] (6.4.5p6) and are of type const char[] ([lex.string]p5) in C++. This means that the assignment in C is valid (it meets all of the requirements for simple assignment) but the attempted modification of the string literal is still undefined behavior (6.4.5p7).

[expr.call-main]

C

```
int main(void) {
  ...
}


static void foo() {
  main(); // OK
}
```

C++

```
int main(void) {
  ...
}


static void foo() {
  main(); // Error
}
```

https://godbolt.org/z/PaGoh8

C++ prohibits calling the main() function ([basic.start.main]p3 while C has no such restriction.

N.B. implementations frequently allow calling main() in C++ as a conforming extension.

	[decl.main-signature]
C	C++

`// Impl-defined if signature is OK`
`float main(void) {`
`  return 0.0f; // Unspecified result`
`               // returned to the host`
`               // environment`
`}`
	`// Error: incorrect signature`
	`float main(void) {`
	`  return 0.0f;`
	`}`

https://godbolt.org/z/934jns	
C++ restricts the signature of main() in several ways, one of which is that the return type must be int ([basic.start.main]p2). C has far less constraints and allows for a fully implementation-defined signature of main(), including the return type (5.1.2.2.1p1). In C, if the return type of main() is not compatible with int, the actual value returned to the host environment is unspecified (5.1.2.2.3p1).	

	[decl.tentative]
C	C++

`// At file scope`
`int i;`
`int i; // OK`
	`// At file scope`
	`int i;`
	`int i; // Error, redefinition`

https://godbolt.org/z/8cq91K	
C has the notion of a tentative definition for a variable at file scope (6.9.2p2) and allows for multiple tentative definitions of the same variable with the result behaving as though there was only a single definition of the object. C++ does not have tentative definitions.	

	[decl.linkage]
C	C++

`// At file scope`
`const int i = 12; // external linkage`
`int j = 10; // external linkage`
	`// At file scope`
	`const int i = 12; // internal linkage`
	`int j = 10; // external linkage`

https://godbolt.org/z/e5rz6v	
A declaration of an object at file scope in C always has external linkage unless the declaration explicitly gives a different linkage (6.2.2p5). C++ has the same rule except that it carves out an exception for the declaration of const objects at file scope, which are given internal linkage ([basic.link]p3).	

	[expr.inc-dec-bool]
C	C++

`#include <stdbool.h>`
`bool b = true;`
`++b; // OK, still true`
	`bool b = true;`
	`++b; // Error`

https://godbolt.org/z/Mv55ce	
C++ does not allow either prefix or postfix ++ or -- to be applied to an object of type bool ([expr.pre.incr]p1-2, [expr.post.incr]p1-2). C allows the expression, but gives a perhaps surprising result that ++ does not always invert the Boolean value. N.B. while C++ disallows prefix ++ and -- on an object of type bool, it still allows some_bool += 1 (and some_bool -= 1) despite defining prefix ++ and -- as being equivalent to += 1 and -= 1 ([expr.pre.incr]p1-2).	

[decl.missed-init]

C

```
void func(void) {
  goto bad; // OK
  int i = 12;
bad:
  ; // i is uninitialized here.
}
```

C++

```
void func(void) {
  goto bad; // Error
  int i = 12;
bad:
  ;
}
```

https://godbolt.org/z/oscWd8

C does not place restrictions on jumping over a declaration with an initializer provided, while C++ explicitly disallows it ([stmt.dcl]p3) in order to ensure that a variable declared with an initializer is always initialized by the time you can access the in-scope object.

[decl.useless-storage-spec]

C

```
// Ok, but useless declaration
static struct S {
  int i;
};
```

C++

```
static struct S { // Error
  int i;
};
```

https://godbolt.org/z/jxxb4W

The grammar productions for both C and C++ allow specifying a storage class specifier along with a type specifier. This allows you to declare both the type and a variable (with the given storage class specifier) in the same declaration, as in static struct S { int i; } s; In C++, if there is no declaration for the storage class specifier to apply to, the code is ill-formed ([dcl.stc]p1) while in C, the useless storage class specifier is ignored.

	[decl.unitialized-const]
C	C++
`const int i; // Okay`	`const int i; // Error`
https://godbolt.org/z/P7G3bs	
In C++, default initialization of a const object requires that the object be const default constructible so that the object is initialized appropriately ([dcl.init]p7). C does not have such a requirement. N.B. that the C++ default initialization rules still apply even if the object would be initialized through other means, such as through zero initialization.	

	[decl.dup-quals]
C	C++
`// Okay, duplicate const ignored` `const const int i = 12;`	`const const int i = 12; // Error`
https://godbolt.org/z/nnoGxd	
The grammar productions for type qualifiers allows type qualifiers to be duplicated. C explicitly ignores duplicate qualifiers as though only a single qualifier was specified (6.7.3p6) while C++ explicitly disallows qualifiers from being duplicated ([dcl.type.general]p2).	

	[decl.auto-storage]
C	C++
`auto int i; // OK`	`auto int i; // Error, two type specifiers`
https://godbolt.org/z/eeETd4	
auto is a storage class specifier in C and a type specifier in C++. So the declaration in C is a valid declaration anywhere an automatic declaration is allowed, but is an error in C++ because the declaration specifies multiple type specifiers. N.B. C2x supports type inference through auto, but auto is still a storage class specifier rather than a type specifier, which is different than in C++ where auto is a type specifier.	

[decl.designated-init]

C

```
struct S {
  int a, b, c;
  struct T {
    int x, y;
  } t;
  float f[3];
} s = {
  .a = 12,
  .c = 10,
  .t.x = 1, 2,
  .f[1] = 1.0f
};
```

C++

```
struct S {
  int a, b, c;
  struct T {
    int x, y;
  } t;
  float f[3];
} s = {
  .a = 12,
  .c = 10, // Error, skips init of b
  .t.x = 1, 2, // Error, specifies subobject
               // Error, does not explicitly
               // specify the second element
               // being initialized
  .f[1] = 1.0f // Error, specifies array index
};
```

https://godbolt.org/z/9WE1EW

Designated initialization in C++ is considerably more restricted than designated initialization in C. In C, designated initialization can occur in any order, does not need to initialize every element, can be used on array elements, can initialize subobjects, and can initialize subsequent members by position rather than by name. In C++, designated initialization must occur in declaration order, must explicitly name the member being initialized, and cannot be used on arrays or subobjects ([dcl.init.list], [dcl.init.aggr]).

Further, the evaluations of the initializer subexpressions are unsequenced in C (6.7.9p23), but are sequenced in declaration order in C++ ([dcl.init.aggr]p6).

	[type.enumerator]
C	C++

`enum foo { one, two };`
`enum bar { red, green };`
 	
`enum foo f = red; // OK`
`enum bar b = 1;    // OK`
	`enum foo { one, two };`
	`enum bar { red, green };`
	
	`enum foo f = red; // Error`
	`enum bar b = 1;    // Error`

https://godbolt.org/z/s9seqv	
In C, the enumerators defined within an enumeration have type int (6.7.2.2p4) and the integer conversion rules allow for an implicit conversion between the enumeration type and its compatible integer type, so the assignments are allowed. In C++, the enumerators have the type of the enumeration ([dcl.enum]p5) and require an explicit cast to avoid the type mismatch on assignment. N.B. the fact that the enumerators have different types in C and C++ can come up in other contexts. For instance, assert(sizeof(enum foo) == sizeof(one)); is guaranteed in C++ but not in C.	

	[decl.empty-tu]
C	C++
`/* empty translation unit: Error */`	`/* empty translation unit: OK */`
https://godbolt.org/z/4vd735	
By virtue of the grammar used in C, a translation unit must declare at least one declaration (6.9p1), while in C++, a translation unit need not declare anything ([basic.link]p1).	

	[decl.empty-decl]
C	C++

`/* At top-level of the`
`   translation unit */`
`; // Error`
	`/* At top-level of the`
	`   translation unit */`
	`; // OK`

https://godbolt.org/z/fGr4K9	
The C++ grammar has an empty-declaration ([dcl.pre]p1) production that the C grammar does not have (6.7p1).	

	[type.scope]
C	C++

`struct X {`
`  int x;`
`  struct Y { int a; } y;`
`};`
 	
`struct Y y; // OK`
	`struct X {`
	`  int x;`
	`  struct Y { int a; } y;`
	`};`
	
	`struct Y y; // Error`

https://godbolt.org/z/nc77qr	
In C++, the scope of a nested class declaration within a class type is the outer class type itself ([class.nest]p1, [class.nested.type]p1), while in C, the nested declaration is scoped to the translation unit (6.7.2.1p10).	

	[stmt.for-decl-storage-class]
C	C++

`for (static int i = 0; // Error`
`  i < 10; ++i)`
`    ;`
	`for (static int i = 0; // OK`
	`  i < 10; ++i)`
	`    ;`

https://godbolt.org/z/76hnnE	
C explicitly prohibits the declaration of a variable with a storage class other than auto or register in a for loop (6.8.5p3), while C++ allows other storage classes to be used.	

	[stmt.for-scope]
C	C++

`for (int i = 0; i < 10; ++i) {`
`  int i = 12; // OK`
`}`
	`for (int i = 0; i < 10; ++i) {`
	`  int i = 12; // Error`
	`}`

https://godbolt.org/z/vYzxPb7zd	
In C++, the scope of the variable declared in the init-statement of the for loop is the same as the scope of the variable declared in the for loop substatement ([stmt.for]p1), while in C the scopes are different (6.8.5.3p1). The result is a redeclaration error in C++ and shadowing in C.	

	[decl.thread-local-storage-class]
C	C++

`#include <threads.h>`
 	
`void func(void) {`
`  thread_local int i; // Error`
`}`
	`void func(void) {`
	`  thread_local int i; // OK`
	`}`

https://godbolt.org/z/b1bK5r	
Thread local variables must have static storage duration in both C and C++. In C++, if the static keyword is absent in the declaration specifiers for the type, static is assumed implicitly ([dcl.stc]p3). C requires the storage duration to be specified (6.7.1p2).	

[expr.left-shift]

C

```
_Static_assert(sizeof(1) == 4,
  "specific to 32-bit ints");
void func(void)
  int i = 1 << 31; // UB
}
```

C++

```
static_assert(sizeof(1) == 4,
  "specific to 32-bit ints");
void func(void) {
  int i = 1 << 31; // OK
}
```

https://godbolt.org/z/oEe6ex

C (6.5.7p4) requires the result of the left-shift expression to be representable in the result type, otherwise the expression has undefined behavior. C++ ([expr.shift]p1-2) only makes the behavior undefined if the shift operand has a value that is negative or is the same (or larger) than the width of the promoted left operand.

[expr.unsequenced-modification]

C

```
void foo(int i) {
  i = i++ + 1; // UB
}
```

C++

```
void foo(int i) {
  i = i++ + 1; // OK, same as i = i + 1;
}
```

https://godbolt.org/z/6she6eahj

In C++, the assignment is sequenced after the value computation of the right-hand side ([expr.ass]p1), so the operation is not unsequenced. In C, the evaluation of both operands are unsequenced (6.5.16p3).

[dcl.array-vla]

C

```
const int i = 4;
int foo[i]; // Error


void func(void) {
  char bar[i] =
    {'a', 'b', 'c'}; // Error
}
```

C++

```
const int i = 4;
int foo[i]; // OK


void func(void) {
  char bar[i] =
    {'a', 'b', 'c'}; // OK
}
```

https://godbolt.org/z/se7cj64ed

In C, a variable declared as const is not an integer constant expression (6.6p6), and so its use to declare the bounds in an array declaration causes the array to be a variable-length array. Variable-length arrays are not allowed to appear at file scope (6.7.6.2p2) and are not allowed to have an initializer (6.7.9p3). In C++, which does not have variable-length arrays, the const variable is an integer constant expression ([expr.const]p8) and produces a valid array declaration.

[lex.ucn]

C	C++
`char c = '\u0025'; // Error`	`char c = '\u0025'; // OK`

https://godbolt.org/z/a6fx5qohe

C places different constraints on universal character names (UCN) than C++. For example, code points less than 0x00A0 are restricted in C (6.4.3p2) and restricted differently in C++ ([lex.charset]p2).

[type.punning]

C

```
int func(float f) {
  union {
    int i;
    float f;
  } u;
  u.f = f;
  return u.i; // OK
}
```

C++

```
int func(float f) {
  union {
    int i;
    float f;
  } u;
  u.f = f;
  return u.i; // UB
}
```

https://godbolt.org/z/Wchs3GTa5

In C++, only one member of the union may be active at a time, and only the active member of the union is within its lifetime ([class.union.general]p1). In the example above, the assignment to u.f causes f to be the active member of the union and starts the lifetime of f. Accessing an object outside of its lifetime is undefined behavior ([basic.life]). This is why accessing u.i is undefined behavior in C++; the lifetime of i has not started.. C does not have the notion of an active member of a union; all union members have the same lifetime as that of the union itself. When nominating a union member, the named member is accessed for its value (6.5.2.3p3), and the result is valid so long as the union is within its lifetime. Thus, it is valid to use a union to type pun in C but not C++.

[expr.unary.plus]

C

```
void func(int *p) {
  +p; // Error
}
```

C++

```
void func(int *p) {
  +p; // OK
}
```

https://godbolt.org/z/Wq7dK8hY6

The unary + operator in C++ operates on an arithmetic, unscoped enumeration, or pointer type ([expr.unary.op]p7) and yields the value of the operand. However, in C, the unary + operator is constrained to operate only on an arithmetic type (6.5.3.3p1), which includes enumeration types (6.2.5p20) but does not include pointer types.

[expr.alignment]	
C	C++
`(void)_Alignof(int[]); // Error`	`(void)alignof(int[]); // OK`
https://godbolt.org/z/vdGd1cEKW	
In C, the operand to _Alignof must be a complete object type (6.5.3.4p1). If the type is an array type, the alignment returned is that of the element (6.5.3.4p3). In C++, the operand to alignof must be a complete object type or an array thereof ([expr.alignof]p1), and if the type is an array, the alignment returned is that of the element ([expr.alignof]p3). An array of unknown bounds is an incomplete type, which is why this example is invalid in C.	

[type.align]	
C	C++

`#include <stdalign.h>`
 	
`int alignas(int) unsigned i; // OK`
	`int alignas(int) unsigned i; // Error`

https://godbolt.org/z/3cz4vv8j6	
In C, the alignas macro expands to the _Alignas keyword (7.15p2), and the _Alignas keyword is part of the *type-specifier-qualifier* production (6.7.2.1p1), which means it may appear anywhere a type specifier or type qualifier may appear. In C++, the alignas keyword is an *attribute-specifier* ([dcl.attr]p1), which means it may appear anywhere an *attribute-specifier-seq* may appear. An attribute specifier sequence is not allowed in all of the places a type specifier or qualifier is allowed.	

	[conformance.error]
C	C++
`#error "error"`	`#error "error"`
N/A	
In C, the #error directive requires the program to not translate (Clause 4p4), while in C++, the directive causes the program to be ill-formed ([cpp.error]p1), and an ill-formed program can still be translated successfully ([intro.compliance.general]p8. N.B. A core issue is being opened in WG21 to address this and no C++ implementation is known to successfully translate a programming containing #error directive, which is why no link is provided demonstrating the diverging behavior.	

	[expr.deref-void-pointer]
C	C++

`void func(void *vp) {`
`  (void)*vp; // Undefined behavior`
`}`
	`void func(void *vp) {`
	`  (void)*vp; // Error`
	`}`

https://godbolt.org/z/d9K1GWobE	
In C++, the operand to a unary * operator requires the operand to be a pointer to an object type ([expr.uunary.op]p1), and void is not an object type ([basic.types.general]p8). C has the same requirements on the operand to a unary * operator, but treats void as being an object type (6.2.5p21). Despite it not being a constraint violation in C, it is nevertheless undefined behavior because the standard doesn’t define what happens in this case.	

	[expr.access-struct-member]
C	C++

`struct S {`
`  char a[10];`
`  // Error`
`  _Static_assert(sizeof(a) == 10);`
`};`
	`struct S {`
	`  char a[10];`
	`  // OK`
	`  static_assert(sizeof(a) == 10);`
	`};`

https://godbolt.org/z/rsT6sKG6f	
The lookup rules of C++ have the ability to find member declarations when performing a lookup within the context of the structure containing the member by virtue of an implicit this object ([expr.prim.id], [basic.lookup.unqual]). C does not treat members of structures or unions as identifiers as ordinary identifiers but as members in their own name space (6.2.3p1) and primary expressions only look up identifiers, not members (6.5.1p1).	

Acknowledgements

I would like to recognize the following people for their help in this work: Jens Gustedt, Tom Honermann, Vsevolod Livinskii, Jens Maurer, Shafik Yaghmour, and Segher Boessenkool.