44

Compatibility

You go ahead and follow your customs,
and I'll follow mine.
– C. Napier

- Introduction
- C++11 Extensions
  Language Features; Standard-Library Components; Deprecated Features; Coping with Older C++ Implementations
- C/C++ Compatibility
  C and C++ Are Siblings; "Silent" Differences; C Code That Is Not C++; C++ Code That Is Not C
- Advice

44.1 Introduction

This chapter discusses the differences between Standard C++ (as defined by ISO/IEC 14882-2011) and earlier versions (such as ISO/IEC 14882-1998), Standard C (as defined by ISO/IEC 9899-2011) and earlier versions (such as Classic C). The purposes are

- To concisely list what is new in C++11
- To document differences that can cause problems for a programmer
- To point to ways of dealing with problems

Most compatibility problems surface when people try to upgrade a C program to a C++ program, try to port a C++ program from an older version of C++ to a newer one (e.g., C++98 to C++11), or try to compile C++ using modern features with an older compiler. The aim here is not to enumerate every possible compatibility problem but rather to list the most frequently occurring problems and present their standard solutions.

When you look at compatibility issues, a key question to consider is the range of implementations under which a program needs to work. For learning C++, it makes sense to use the most

complete and helpful implementation. For delivering a product, a more conservative strategy might be in order to maximize the number of systems on which the product can run. In the past, this has been a reason (and more often just an excuse) to avoid C++ features deemed novel. However, implementations are converging, so the need for portability across platforms is less cause for extreme caution than it once was.

44.2 C++11 Extensions

First, I list the language features and standard-library components that have been added to C++ for the C++11 standard. Next, I discuss ways of coping with older versions (notably C++98).

44.2.1 Language Features

Looking at a list of language features can be quite bewildering. Remember that a language feature is not meant to be used in isolation. In particular, most features that are new in C++11 make no sense in isolation from the framework provided by older features. The order is roughly that of first occurrence in this book:

[1] Uniform and general initialization using **{}**-lists (§2.2.2, §6.3.5)
[2] Type deduction from initializer: **auto** (§2.2.2, §6.3.6.1)
[3] Prevention of narrowing (§2.2.2, §6.3.5)
[4] Generalized and guaranteed constant expressions: **constexpr** (§2.2.3, §10.4, §12.1.6)
[5] Range-**for**-statement (§2.2.5, §9.5.1)
[6] Null pointer keyword: **nullptr** (§2.2.5, §7.2.2)
[7] Scoped and strongly typed **enum**s: **enum class** (§2.3.3, §8.4.1)
[8] Compile-time assertions: **static_assert** (§2.4.3.3, §24.4)
[9] Language mapping of **{}**-list to **std::initializer_list** (§3.2.1.3, §17.3.4)
[10] Rvalue references (enabling move semantics; §3.3.2, §7.7.2)
[11] Nested template arguments ending with **>>** (no space between the **>**s; §3.4.1)
[12] Lambdas (§3.4.3, §11.4)
[13] Variadic templates (§3.4.4, §28.6)
[14] Type and template aliases (§3.4.5, §6.5, §23.6)
[15] Unicode characters (§6.2.3.2, §7.3.2.2)
[16] **long long** integer type (§6.2.4)
[17] Alignment controls: **alignas** and **alignof** (§6.2.9)
[18] The ability to use the type of an expression as a type in a declaration: **decltype** (§6.3.6.1)
[19] Raw string literals (§7.3.2.1)
[20] Generalized POD (§8.2.6)
[21] Generalized **union**s (§8.3.1)
[22] Local classes as template arguments (§11.4.2, §25.2.1)
[23] Suffix return type syntax (§12.1.4)
[24] A syntax for attributes and two standard attributes: **[[carries_dependency]]** (§41.3) and **[[noreturn]]** (§12.1.7)
[25] Preventing exception propagation: the **noexcept** specifier (§13.5.1.1)

[26] Testing for the possibility of a **throw** in an expression: the **noexcept** operator (§13.5.1.1)
[27] C99 features: extended integral types (i.e., rules for optional longer integer types; §6.2.4); concatenation of narrow/wide strings; **__func__** and **__STDC_HOSTED__** (§12.6.2); **_Pragma(X)** (§12.6.3); vararg macros and empty macro arguments (§12.6)
[28] **inline** namespaces (§14.4.6)
[29] Delegating constructors (§17.4.3)
[30] In-class member initializers (§17.4.4)
[31] Control of defaults: **default** (§17.6) and **delete** (§17.6.4)
[32] Explicit conversion operators (§18.4.2)
[33] User-defined literals (§19.2.6)
[34] More explicit control of **template** instantiation: **extern template**s (§26.2.2)
[35] Default template arguments for function templates (§25.2.5.1)
[36] Inheriting constructors (§20.3.5.1)
[37] Override controls: **override** and **final** (§20.3.4)
[38] Simpler and more general SFINAE rule (§23.5.3.2)
[39] Memory model (§41.2)
[40] Thread-local storage: **thread_local** (§42.2.8)

I have not tried to list every minute change to C++98 in C++11. A historical perspective on these features can be found in §1.4.

44.2.2 Standard_Library Components

The C++11 additions to the standard library come in two forms: new components (such as the regular expression matching library) and improvements to C++98 components (such as move constructors for containers).

[1] **initializer_list** constructors for containers (§3.2.1.3, §17.3.4, §31.3.2)
[2] Move semantics for containers (§3.3.1, §17.5.2, §31.3.2)
[3] A singly-linked list: **forward_list** (§4.4.5, §31.4.2)
[4] Hash containers: **unordered_map**, **unordered_multimap**, **unordered_set**, and **unordered_multiset** (§4.4.5, §31.4.3)
[5] Resource management pointers: **unique_ptr**, **shared_ptr**, and **weak_ptr** (§5.2.1, §34.3)
[6] Concurrency support: **thread** (§5.3.1, §42.2), mutexes (§5.3.4, §42.3.1), locks (§5.3.4, §42.3.2), and condition variables (§5.3.4.1, §42.3.4)
[7] Higher-level concurrency support: **packaged_thread**, **future**, **promise**, and **async()** (§5.3.5, §42.4)
[8] **tuple**s (§5.4.3, §28.5, §34.2.4.2)
[9] Regular expressions: **regex** (§5.5, Chapter 37)
[10] Random numbers: **uniform_int_distribution**, **normal_distribution**, **random_engine**, etc. (§5.6.3, §40.7)
[11] Integer type names, such as **int16_t**, **uint32_t**, and **int_fast64_t** (§6.2.8, §43.7)
[12] A fixed-sized contiguous sequence container: **array** (§8.2.4, §34.2.1)
[13] Copying and rethrowing exceptions (§30.4.1.2)
[14] Error reporting using error codes: **system_error** (§30.4.3)

[15] emplace() operations for containers (§31.3.6)
[16] Wide use of constexpr functions
[17] Systematic use of noexcept functions
[18] Improved function adaptors: function and bind() (§33.5)
[19] string to numeric value conversions (§36.3.5)
[20] Scoped allocators (§34.4.4)
[21] Type traits, such as is_integral and is_base_of (§35.4)
[22] Time utilities: duration and time_point (§35.2)
[23] Compile-time rational arithmetic: ratio (§35.3)
[24] Abandoning a process: quick_exit (§15.4.3)
[25] More algorithms, such as move(), copy_if(), and is_sorted() (Chapter 32)
[26] Garbage collection ABI (§34.5)
[27] Low-level concurrency support: atomics (§41.3)

More information about the standard library can be found in

- Chapter 4, Chapter 5, and Part IV
- Implementation technique examples: vector (§13.6), string (§19.3), and tuple (§28.5)
- The emerging specialized C++11 standard-library literature, such as [Williams,2012]
- A brief historical perspective can be found in §1.4.

44.2.3 Deprecated Features

By deprecating a feature, the standards committee expresses the wish that the feature will go away (§iso.D). However, the committee does not have a mandate to immediately remove a heavily used feature – however redundant or dangerous it may be. Thus, a deprecation is a strong hint to avoid the feature. It may disappear in the future. Compilers are likely to issue warnings for uses of deprecated features.

- Generation of the copy constructor and the copy assignment is deprecated for a class with a destructor.
- It is no longer allowed to assign a string literal to a char* (§7.3.2).
- C++98 exception specifications are deprecated:

```
void f() throw(X,Y);  // C++98; now deprecated
```

  The support facilities for exception specifications, unexcepted_handler, set_unexpected(), get_unexpected(), and unexpected(), are similarly deprecated. Instead, use noexcept (§13.5.1.1).
- Some C++ standard-library function objects and associated functions are deprecated: unary_function, binary_function, pointer_to_unary_function, pointer_to_binary_function, ptr_fun(), mem_fun_t, mem_fun1_t, mem_fun_ref_t, mem_fun_ref1_t, mem_fun(), const_mem_fun_t, const_mem_fun1_t, const_mem_fun_ref_t, const_mem_fun_ref1_t, binder1st, bind1st(), binder2nd, bind2nd(). Instead, use function and bind() (§33.5).
- The auto_ptr is deprecated. Instead, use unique_ptr (§5.2.1, §34.3.1).

In addition, the committee did remove the essentially unused export feature, because it was complex and not shipped by the major vendors.

C-style casts should have been deprecated when the named casts (§11.5.2) were introduced. Programmers should seriously consider banning C-style casts from their own programs. Where explicit type conversion is necessary, **static_cast**, **reinterpret_cast**, **const_cast**, or a combination of these can do what a C-style cast can. The named casts should be preferred because they are more explicit and more visible.

44.2.4 Coping with Older C++ Implementations

C++ has been in constant use since 1983 (§1.4). Since then, several versions have been defined, and many separately developed implementations have emerged. The fundamental aim of the standards effort was to ensure that implementers and users would have a single definition of C++ to work from. From 1998, programmers could rely on the ISO C++98 standard, and now we have the ISO C++11 standard.

Unfortunately, it is not uncommon for people to take their first serious look at C++ using a five-year-old implementation. The typical reason is that such implementations are widely available and free. Given a choice, no self-respecting professional would touch such an antique. Also, many modern quality implementations are available for free. For a novice, older implementations come with serious hidden costs. The lack of language features and library support means that the novice must struggle with problems that have been eliminated in newer implementations. Using a feature-poor older implementation, especially if guided by an antique tutorial, warps the novice's programming style and gives a biased view of what C++ is. The best subset of C++ to initially learn is *not* the set of low-level facilities (and not the common C and C++ subset; see §1.3). In particular, to ease learning and to get a good initial impression of what C++ programming can be, I recommend relying on the standard library, and to heavily use classes, templates, and exceptions.

There are still places, where for political reasons or lack of suitable tool chains, C is preferred over C++. If you must use C, write in the common subset of C and C++. That way, you gain some type safety, increase portability, and will be ready when C++ features become available to you. See also §1.3.3.

Use an implementation that conforms to the standard wherever possible, and minimize the reliance on implementation-defined and undefined aspects of the language. Design as if the full language were available, and only use workarounds when necessary. This leads to better organized and more maintainable programs than designing for a lowest-common-denominator subset of C++. Also, use implementation-specific language extensions only when necessary. See also §1.3.2.

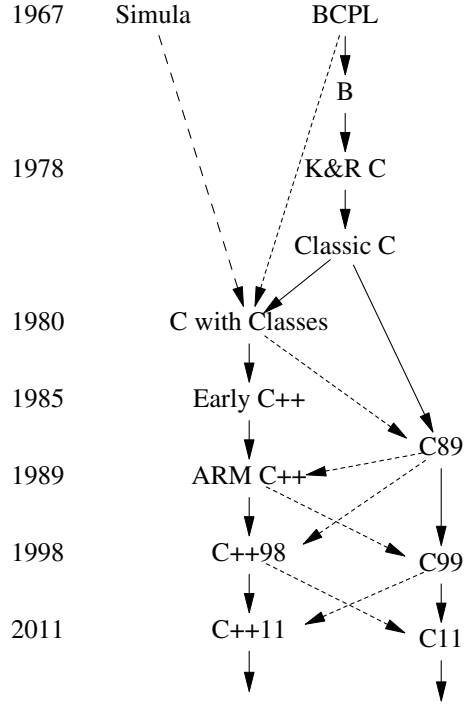
44.3 C/C++ Compatibility

With minor exceptions, C++ is a superset of C (meaning C11, defined by ISO/IEC 9899:2011(E)). Most differences stem from C++'s greater emphasis on type checking. Well-written C programs tend to be C++ programs as well. A compiler can diagnose every difference between C++ and C. The C99/C++11 incompatibilities are listed in §iso.C. At the time of writing, C11 is still very new and most C code is Classic C or C99.

44.3.1 C and C++ Are Siblings

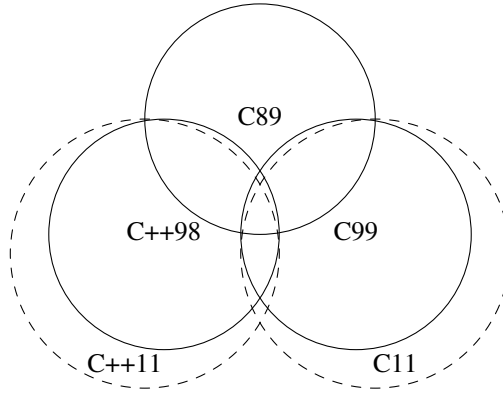
Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of libraries and tools for C and C++.

How can I call C and C++ siblings? Clearly, C++ is a descendant of C. However, look at a simplified family tree:



A solid line means a massive inheritance of features, a dashed line a borrowing of major features, and a dotted line a borrowing of minor features. From this, ISO C and ISO C++ emerge as the two major descendants of K&R C, and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. I picked the term "Classic C" from a sticker that used to be affixed to Dennis Ritchie's terminal. It is K&R C plus enumerations and **struct** assignment.

Incompatibilities are nasty for programmers in part because they create a combinatorial explosion of alternatives. Consider a simple Venn diagram:



The areas are not to scale. Both C++11 and C11 have most of K&R C as a subset. C++11 has most of C11 as a subset. There are features belonging to most of the distinct areas. For example:

C89 only	Call of undeclared function
C99 only	Variable-length arrays (VLAs)
C++ only	Templates
C89 and C99	Algol-style function definitions
C89 and C++	Use of the C99 keyword **restrict** as an identifier
C++ and C99	**//** comments
C89, C++, and C99	**struct**s
C++11 only	Move semantics (using rvalue references; **&&**)
C11 only	Type-generic expressions using the **_Generic** keyword
C++11 and C11	Atomics

Note that differences between C and C++ are not necessarily the result of changes to C made in C++. In several cases, the incompatibilities arise from features adopted incompatibly into C long after they were common in C++. Examples are the ability to assign a **T*** to a **void*** and the linkage of global **const**s [Stroustrup,2002]. Sometimes, a feature was even incompatibly adopted into C after it was part of the ISO C++ standard, such as details of the meaning of **inline**.

44.3.2 "Silent" Differences

With a few exceptions, programs that are both C++ and C have the same meaning in both languages. Fortunately, these exceptions (often referred to as *silent differences*) are rather obscure:

- In C, the size of a character constant and of an enumeration equals **sizeof(int)**. In C++, **sizeof('a')** equals **sizeof(char)**.
- In C, an enumerator is an **int**, whereas a C++ implementation is allowed to choose whatever size is most appropriate for an enumeration (§8.4.2).
- In C++, the name of a **struct** is entered into the scope in which it is declared; in C, it is not. Thus, the name of a C++ **struct** declared in an inner scope can hide the name in an outer scope. For example:

```
int x[99];
void f()
{
        struct x { int a; };
        sizeof(x);              /* size of the array in C, size of the struct in C++ */
        sizeof(struct x);       /* size of the struct */
}
```

44.3.3 C Code That Is Not C++

The C/C++ incompatibilities that cause most real problems are not subtle. Most are easily caught by compilers. This section gives examples of C code that is not C++. Most are deemed poor style or even obsolete in modern C. A comprehensive list of incompatibilities can be found in §iso.C.

- In C, most functions can be called without a previous declaration. For example:

```
int main()      // not C++; poor style in C
{
        double sq2 = sqrt(2);                           /* call undeclared function */
        printf("the square root of 2 is %g\n",sq2);     /* call undeclared function */
}
```

  Complete and consistent use of function declarations (function prototypes) is generally recommended for C. Where that sensible advice is followed, and especially where C compilers provide options to enforce it, C code conforms to the C++ rule. Where undeclared functions are called, you have to know the functions and the rules for C pretty well to know whether you have made a mistake or introduced a portability problem. For example, the previous `main()` contains at least two errors as a C program.

- In C, a function declared without specifying any argument types can take any number of arguments of any type at all.

```
void f();   /* argument types not mentioned */

void g()
{
        f(2);       /* poor style in C; not C++ */
}
```

  Such use is deemed obsolete in ISO C.

- In C, functions can be defined using a syntax that optionally specifies argument types after the list of arguments:

```
void f(a,p,c) char *p; char c; { /* ... */ }          /* C; not C++ */
```

  Such definitions must be rewritten:

```
void f(int a, char* p, char c) { /* ... */ }
```

- In C, `struct`s can be defined in return type and argument type declarations. For example:

```
struct S { int x,y; } f();          /* C; not C++ */
void g(struct S { int x,y; } y);    /* C; not C++ */
```

The C++ rules for defining types make such declarations useless, and they are not allowed.

- In C, integers can be assigned to variables of enumeration type:

```
enum Direction { up, down };
enum Direction d = 1;               /* error: int assigned to Direction; OK in C */
```

- C++ provides many more keywords than C does. If one of these appears as an identifier in a C program, that program must be modified to make it a C++ program:

C++ Keywords That Are Not C Keywords					
alignas	alignof	and	and_eq	asm	bitand
bitor	bool	catch	char16_t	char32_t	class
compl	const_cast	constexpr	decltype	delete	dynamic_cast
explicit	false	friend	inline	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or_eq	private	protected	public	reinterpret_cast	static_assert
static_cast	template	this	thread_local	throw	true
try	typeid	typename	using	virtual	wchar_t
xor	xor_eq				

In addition, the word **export** is reserved for future use. C99 adopted **inline**.

- In C, some of the C++ keywords are macros defined in standard headers:

C++ Keywords That Are C Macros								
and	and_eq	bitand	bitor	bool	compl	false	not	not_eq
or	or_eq	true	wchar_t	xor	xor_eq			

This implies that in C they can be tested using **#ifdef**, redefined, etc.

- In C, a global data object may be declared several times in a single translation unit without using the **extern** specifier. As long as at most one such declaration provides an initializer, the object is considered defined only once. For example:

```
int i;
int i; /* just another declaration of a single integer "i"; not C++ */
```

In C++, an entity must be defined exactly once; §15.2.3.

- In C, a **void*** may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not (§7.2.1). For example:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int));  /* not C++; in C++, allocate using "new" */
}
```

This is probably the single most difficult incompatibility to deal with. Note that the implicit conversion of a **void*** to a different pointer type is *not* in general harmless:

```
char ch;
void* pv = &ch;
int* pi = pv;          // not C++
*pi = 666;             // overwrite ch and other bytes near ch
```

If you use both languages, cast the result of **malloc()** to the right type. If you use only C++, avoid **malloc()**.

- In C, the type of a string literal is "array of **char**," but in C++ it is "array of **const char**," so:

```
char* p = "a string literal is not mutable";      // error in C++; OK in C
p[7] = 'd';
```

- C allows transfer of control to a labeled statement (a **switch** or a **goto**; §9.6) to bypass an initialization; C++ does not. For example:

```
goto foo;              // OK in C; not C++
// ...
{
    int x = 1;
foo:
    if (x!=1) abort();
    /* ... */
}
```

- In C, a global **const** by default has external linkage; in C++ it does not and must be initialized, unless explicitly declared **extern** (§7.5). For example:

```
const int ci;          // OK in C; const not initialized error in C++
```

- In C, names of nested structures are placed in the same scope as the structure in which they are nested. For example:

```
struct S {
    struct T { /* ... */ } t;
    // ...
};

struct T x;            // OK in C, meaning "S::T x;"; not C++
```

- In C++, the name of a class is entered into the scope in which it is declared; thus it cannot have the same name as another type declared in that scope. For example:

```
struct X { /* ... */ };
typedef int X;         // OK in C; not C++
```

- In C, an array can be initialized by an initializer that has more elements than the array requires. For example:

```
char v[5] = "Oscar";   // OK in C, the terminating 0 is not used; not C++
printf("%s",v);        // likely disaster
```

44.3.3.1 "Classic C" Problems

Should you need to upgrade Classic C programs ("K&R C") or C89 programs, a few more problems will emerge:

- C89 does not have the `//` comments (though most C89 compilers added them):

```
int x;      // not C89
```

- In C89, the type specifier defaults to `int` (known as "implicit `int`"). For example:

```
const a = 7;    /* in C89, type int assumed; not C++ or C99 */

f()    /* f()'s return type is int by default; not C++ or C99 */
{
       /* .. */
}
```

44.3.3.2 C Features Not Adopted by C++

A few additions to C99 (compared with C89) were deliberately not adopted in C++:

[1] Variable-length arrays (VLAs); use `vector` or some form of dynamic array
[2] Designated initializers; use constructors

The C11 features are too new to have been considered for C++, except for features such as the memory model and atomics (§41.3) that came from C++.

44.3.4 C++ Code That Is Not C

This section lists facilities offered by C++ but not by C (or adopted by C years after their introduction in C++, as marked, so that they may be missing in old C compilers). The features are sorted by purpose. However, many classifications are possible, and most features serve multiple purposes, so this classification should not be taken too seriously.

- Features primarily for notational convenience:
  [1] `//` comments (§2.2.1, §9.7); added to C99
  [2] Support for restricted character sets (§iso.2.4); partially added to C99
  [3] Support for extended character sets (§6.2.3); added to C99
  [4] Non-constant initializers for objects in `static` storage (§15.4.1)
  [5] `const` in constant expressions (§2.2.3, §10.4.2)
  [6] Declarations as statements (§9.3); added to C99
  [7] Declarations in `for`-statement initializers (§9.5); added to C99
  [8] Declarations in conditions (§9.4.3)
  [9] Structure names need not be prefixed by `struct` (§8.2.2)
  [10] Anonymous `union`s (§8.3.2); added to C11
- Features primarily for strengthening the type system:
  [1] Function argument type checking (§12.1); partially added to C (§44.3.3)
  [2] Type-safe linkage (§15.2, §15.2.3)
  [3] Free-store management using `new` and `delete` (§11.2)

[4] **const** (§7.5, §7.5); partially added to C
[5] The Boolean type **bool** (§6.2.2); partially added to C99
[6] Named casts (§11.5.2)

- Facilities for user-defined types:
  [1] Classes (Chapter 16)
  [2] Member functions (§16.2.1) and member classes (§16.2.13)
  [3] Constructors and destructors (§16.2.5, Chapter 17)
  [4] Derived classes (Chapter 20, Chapter 21)
  [5] **virtual** functions and abstract classes (§20.3.2, §20.4)
  [6] Public/protected/private access control (§16.2.3, §20.5)
  [7] **friend**s (§19.4)
  [8] Pointers to members (§20.6)
  [9] **static** members (§16.2.12)
  [10] **mutable** members (§16.2.9.3)
  [11] Operator overloading (Chapter 18)
  [12] References (§7.7)
- Features primarily for program organization (in addition to classes):
  [1] Templates (Chapter 23)
  [2] Inline functions (§12.1.3); added to C99
  [3] Default arguments (§12.2.5)
  [4] Function overloading (§12.3)
  [5] Namespaces (§14.3.1)
  [6] Explicit scope qualification (operator **::**; §6.3.4)
  [7] Exceptions (§2.4.3.1, Chapter 13)
  [8] Run-Time Type Identification (Chapter 22)
  [9] Generalized constant expressions (**constexpr**; §2.2.3, §10.4, §12.1.6)

The C++11 features listed in §44.2 are not in C.

The keywords added by C++ (§44.3.3) can be used to spot most C++-specific facilities. However, some facilities, such as function overloading and **const**s in constant expressions, are not identified by a keyword.

C++'s linking for functions is type-safe, whereas C's rules do not require type safety when linking functions. This implies that on some (most?) implementations, a C++ function must be declared **extern "C"** to be compiled as C++ and also conform to C calling conventions (§15.2.5). For example:

```
double sin(double);              // may not link to C code
extern "C" double cos(double);   // will link to C code
```

The **__cplusplus** macro can be used to determine whether a program is being processed by a C or a C++ compiler (§15.2.5).

In addition to the features listed, the C++ library (§30.1.1, §30.2) is mostly C++-specific. The C standard library offers type-generic macros in **<tgmath.h>** and **_Complex** number support in **<complex.h>**, approximating **<complex>**.

C also offers **<stdbool.h>**, offering **_Bool** and the alias **bool** to approximate C++'s **bool**.

44.4 Advice

[1] Before using a new feature in production code, try it out by writing small programs to test the standards conformance and performance of the implementations you plan to use; §44.1.
[2] For learning C++, use the most up-to-date and complete implementation of Standard C++ that you can get access to; §44.2.4.
[3] The common subset of C and C++ is not the best initial subset of C++ to learn; §1.2.3, §44.2.4.
[4] Prefer standard facilities to nonstandard ones; §36.1, §44.2.4.
[5] Avoid deprecated features such as **throw**-specifications; §44.2.3, §13.5.1.3.
[6] Avoid C-style casts; §44.2.3, §11.5.
[7] "Implicit **int**" has been banned, so explicitly specify the type of every function, variable, **const**, etc.; §44.3.3.
[8] When converting a C program to C++, first make sure that function declarations (prototypes) and standard headers are used consistently; §44.3.3.
[9] When converting a C program to C++, rename variables that are C++ keywords; §44.3.3.
[10] For portability and type safety, if you must use C, write in the common subset of C and C++; §44.2.4.
[11] When converting a C program to C++, cast the result of **malloc()** to the proper type or change all uses of **malloc()** to uses of **new**; §44.3.3.
[12] When converting from **malloc()** and **free()** to **new** and **delete**, consider using **vector**, **push_back()**, and **reserve()** instead of **realloc()**; §3.4.2, §43.5.
[13] When converting a C program to C++, remember that there are no implicit conversions from **int**s to enumerations; use explicit type conversion where necessary; §44.3.3, §8.4.
[14] A facility defined in namespace **std** is defined in a header without a suffix (e.g., **std::cout** is declared in **<iostream>**); §30.2.
[15] Use **<string>** to get **std::string** (**<string.h>** holds the C-style string functions); §15.2.4.
[16] For each standard C header **<X.h>** that places names in the global namespace, the header **<cX>** places the names in namespace **std**; §15.2.2.
[17] Use **extern "C"** when declaring C functions; §15.2.5.