Appendix B

Compatibility

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

C/C++ compatibility — silent differences between C and C++ — C code that is not C++ — deprecated features — C++ code that is not C — coping with older C++ implementations — headers — the standard library — namespaces — allocation errors — templates — *for-statement* initializers.

B.1 Introduction

This appendix discusses the incompatibilities between C and C++ and between Standard C++ and earlier versions of C++. The purpose is to document differences that can cause problems for the programmer and point to ways of dealing with such problems. Most compatibility problems surface when people try to upgrade a C program to a C++ program, to try port a C++ program from one pre-standard version of C++ to another, or try to compile C++ using modern features with an older compiler. The aim here is not to drown you in the details of every compatibility problem that ever surfaced in an implementation, 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 sometimes 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 was a couple of years ago.

B.2 C/C++ Compatibility

With minor exceptions, C++ is a superset of C. Most differences stem from C++'s greater emphasis on type checking. Well-written C programs tend to be C++ programs as well. All differences between C++ and C can be diagnosed by a compiler.

B.2.1 ''Silent'' Differences

With a few exceptions, programs that are both C++ and C have the same meaning in both languages. Fortunately, these ''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)`, and a C++ implementation is allowed to choose whatever size is most appropriate for an enumeration (§4.8).

C++ provides the `//` comments; C does not (although many C implementations provide it as an extension). This difference can be used to construct programs that behave differently in the two languages. For example:

```
int f(int a, int b)
{
    return a //* pretty unlikely */ b
        ;    /* unrealistic: semicolon on separate line to avoid syntax error */
}
```

ISO C is being revised to allow `//` as in C++ .

A structure name declared in an inner scope can hide the name of an object, function, enumerator, or type 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++ */
}
```

B.2.2 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.

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

```
main()    /* poor style C. Not 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. Such use is deemed obsolescent in Standard C, but it is not uncommon:

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

void g()
{
    f(2);       /* poor style C. Not 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 and in pre-standard versions of C++, the type specifier defaults to ***int***. For example:

```
const a = 7;    /* In C, type int assumed. Not C++ */
```

ISO C is being revised to disallow ''implicit ***int***,'' just as in C++.

C allows the definition of ***struct***s 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 };
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 a C++ program:

C++ Keywords That Are Not C Keywords					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

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

C++ Keywords That Are C Macros					
and	*and_eq*	*bitand*	*bitor*	*compl*	*not*
not_eq	*or*	*or_eq*	*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 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;      /* defines or declares a single integer 'i'; not C++ */
```

In C++, an entity must be defined exactly once; §9.2.3.

In C++, a class may not have the same name as a ***typedef*** declared to refer to a different type in the same scope; §5.7.

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 (§5.6). For example:

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

C allows jumps to bypass an initialization; C++ does not.

In C, a global ***const*** by default has external linkage; in C++ it does not and must be initialized, unless explicitly declared ***extern*** (§5.4).

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 {
            int a;
      };
      int b;
};

struct T x;      /* ok in C meaning 'S::T x;'. 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++ */
```

B.2.3 Deprecated Features

By deprecating a feature, the standards committee expresses the wish that the feature would go away. However, the committee does not have a mandate to remove a heavily used feature – however redundant or dangerous it may be. Thus, a deprecation is a strong hint to the users to avoid the feature.

The keyword ***static***, which usually means ''statically allocated,'' can be used to indicate that a function or an object is local to a translation unit. For example:

```
// file1:
    static int glob;

// file2:
    static int glob;
```

This program genuinely has two integers called ***glob***. Each ***glob*** is used exclusively by functions defined in its translation unit.

The use of ***static*** to indicate ''local to translation unit'' is deprecated in C++. Use unnamed namespaces instead (§8.2.5.1).

C-style casts should have been deprecated when the new-style casts 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 new-style casts should be preferred because they are more explicit and more visible (§6.2.7).

B.2.4 C++ Code That Is Not C

This section lists facilities offered by C++ but not by C. 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 (§2.3); being added to C
  - [2] Support for restricted character sets (§C.3.1)
  - [3] Support for extended character sets (§C.3.3); being added to C
  - [4] Non-constant initializers for objects in ***static*** storage (§9.4.1)
  - [5] ***const*** in constant expressions (§5.4, §C.5)
  - [6] Declarations as statements (§6.3.1)
  - [7] Declarations in *for-statement* initializers and conditions (§6.3.3, §6.3.2.1)
  - [8] Structure names need not be prefixed by ***struct*** (§5.7)
- Features primarily for strengthening the type system:
  - [1] Function argument type checking (§7.1); later added to C (§B.2.2)
  - [2] Type-safe linkage (§9.2, §9.2.3)
  - [3] Free store management using ***new*** and ***delete*** (§6.2.6, §10.4.5, §15.6)
  - [4] ***const*** (§5.4, §5.4.1); later added to C
  - [5] The Boolean type ***bool*** (§4.2)
  - [6] New cast syntax (§6.2.7)
- Facilities for user-defined types:
  - [1] Classes (Chapter 10)
  - [2] Member functions (§10.2.1) and member classes (§11.12)
  - [3] Constructors and destructors (§10.2.3, §10.4.1)
  - [4] Derived classes (Chapter 12, Chapter 15)
  - [5] ***virtual*** functions and abstract classes (§12.2.6, §12.3)

[6] Public/protected/private access control (§10.2.2, §15.3, §C.11)
[7] ***friend***s (§11.5)
[8] Pointers to members (§15.5, §C.12)
[9] ***static*** members (§10.2.4)
[10] ***mutable*** members (§10.2.7.2)
[11] Operator overloading (Chapter 11)
[12] References (§5.5)

– Features primarily for program organization (in addition to classes):
[1] Templates (Chapter 13, §C.13)
[2] Inline functions (§7.1.1)
[3] Default arguments (§7.5)
[4] Function overloading (§7.4)
[5] Namespaces (§8.2)
[6] Explicit scope qualification (operator `::`; §4.9.4)
[7] Exception handling (§8.3, Chapter 14)
[8] Run-time Type Identification (§15.4)

The keywords added by C++ (§B.2.2) 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. In addition to the language features listed here, the C++ library (§16.1.2) is mostly C++ specific.

The ***__cplusplus*** macro can be used to determine whether a program is being processed by a C or a C++ compiler (§9.2.4).

B.3 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. Until that definition becomes pervasive in the C++ community, however, we have to deal with the fact that not every implementation provides every feature described in this book.

It is unfortunately 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. 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 also 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; §1.2). In particular, I recommend relying on the standard library and on templates to ease learning and to get a good initial impression of what C++ programming can be.

The first commercial release of C++ was in late 1985. The language was defined by the first edition of this book. At that point, C++ did not offer multiple inheritance, templates, run-time type information, exceptions, or namespaces. Today, I see no reason to use an implementation that

doesn't provide at least some of these features. I added multiple inheritance, templates, and exceptions to the definition of C++ in 1989. However, early support for templates and exceptions was uneven and often poor. If you find problems with templates or exceptions in an older implementation, consider an immediate upgrade.

In general, it is wise to use an implementation that conforms to the standard wherever possible and to minimize the reliance on implementation-defined and undefined aspects of the language. Design as if the full language were available and then use whatever workarounds are needed. This leads to better organized and more maintainable programs than designing for the lowest-common-denominator subset of C++. Also, be careful to use implementation-specific language extensions only when absolutely necessary.

B.3.1 Headers

Traditionally, every header file had a *.h* suffix. Thus, C++ implementations provided headers such as *<map.h>* and *<iostream.h>*. For compatibility, most still do.

When the standards committee needed headers for redefined versions of standard libraries and for newly added library facilities, naming those headers became a problem. Using the old *.h* names would have caused compatibility problems. The solution was to drop the *.h* suffix in standard header names. The suffix is redundant anyway because the < > notation indicates that a standard header is being named.

Thus, the standard library provides non-suffixed headers, such as *<iostream>* and *<map>*. The declarations in those files are placed in namespace *std*. Older headers place their declarations in the global namespace and use a *.h* suffix. Consider:

```
#include<iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

If this fails to compile on an implementation, try the more traditional version:

```
#include<iostream.h>

int main()
{
    cout << "Hello, world!\n";
}
```

Some of the most serious portability problems occur because of incompatible headers. The standard headers are only a minor contributor to this. Often, a program depends on a large number of headers that are not present on all systems, on a large number of declarations that don't appear in the same headers on all systems, and on declarations that appear to be standard (because they are found in headers with standard names) but are not part of any standard.

There are no fully-satisfactory approaches to dealing with portability in the face of inconsistent headers. A general idea is to avoid direct dependencies on inconsistent headers and localize the remaining dependencies. That is, we try to achieve portability through indirection and localization.

For example, if declarations that we need are provided in different headers in different systems, we may choose to *#include* an application specific header that in turn *#include*s the appropriate header(s) for each system. Similarly, if some functionality is provided in slightly different forms on different systems, we may choose to access that functionality through application-specific interface classes and functions.

B.3.2 The Standard Library

Naturally, pre-standard-C++ implementations may lack parts of the standard library. Most will have iostreams, non-templated *complex*, a different *string* class, and the C standard library. However, some may lack *map*, *list*, *valarray*, etc. In such cases, use the – typically proprietary – libraries available in a way that will allow conversion when your implementation gets upgraded to the standard. It is usually better to use a non-standard *string*, *list*, and *map* than to revert to C-style programming in the absence of these standard library classes. Also, good implementations of the STL part of the standard library (Chapter 16, Chapter 17, Chapter 18, Chapter 19) are available free for downloading.

Early implementations of the standard library were incomplete. For example, some had containers that didn't support allocators and others required allocators to be explicitly specified for each class. Similar problems occurred for other ''policy arguments,'' such as comparison criteria. For example:

```
list<int> li;                          // ok, but some implementations require an allocator
list<int, allocator<int> > li2;        // ok, but some implementations don't implement allocators

map<string, Record> m1;                // ok, but some implementations require a less-operation
map<string, Record, less<string> > m2;
```

Use whichever version an implementation accepts. Eventually, the implementations will accept all.

Early C++ implementations provided *istrstream* and *ostrstream* defined in *<strstream.h>* instead of *istringstream* and *ostringstream* defined in *<sstream>*. The *strstream*s operated directly on a *char*[] (see §21.10[26]).

The streams in pre-standard-C++ implementations were not parameterized. In particular, the templates with the *basic_* prefix are new in the standard, and the *basic_ios* class used to be called *ios*. Curiously enough, *iostate* used to be called *io_state*.

B.3.3 Namespaces

If your implementation does not support namespaces, use source files to express the logical structure of the program (Chapter 9). Similarly, use header files to express interfaces that you provide for implementations or that are shared with C.

In the absence of namespaces, use *static* to compensate for the lack of unnamed namespaces. Also use an identifying prefix to global names to distinguish your names from those of other parts of the code. For example:

```
// for use on pre-namespace implementations:

class bs_string { /* ... */ };     // Bjarne's string
typedef int bs_bool;               // Bjarne's Boolean type
```

```
class joe_string;                       // Joe's string
enum joe_bool { joe_false, joe_true };  // Joe's bool
```

Be careful when choosing a prefix. Existing C and C++ libraries are littered with such prefixes.

B.3.4 Allocation Errors

In pre-exception-handling-C++, operator ***new*** returned ***0*** to indicate allocation failure. Standard C++'s ***new*** throws ***bad_alloc*** by default.

In general, it is best to convert to the standard. In this case, this means modify the code to catch ***bad_alloc*** rather than test for ***0***. In either case, coping with memory exhaustion beyond giving an error message is hard on many systems.

However, when converting from testing ***0*** to catching ***bad_alloc*** is impractical, you can sometimes modify the program to revert to the pre-exception-handling behavior. If no ***_new_handler*** is installed, using the ***nothrow*** allocator will cause a ***0*** to be returned in case of allocation failure:

```
X* p1 = new X;              // throws bad_alloc if no memory
X* p2 = new(nothrow) X;     // returns 0 if no memory
```

B.3.5 Templates

The standard introduced new template features and clarified the rules for several existing ones.

If your implementation doesn't support partial specialization, use a separate name for the template that would otherwise have been a specialization. For example:

```
template<class T> class plist : private list<void*> {  // should have been list<T*>
    // ...
};
```

If your implementation doesn't support member templates, some techniques become infeasible. In particular, member templates allow the programmer to specify construction and conversion with a flexibility that cannot be matched without them (§13.6.2). Sometimes, providing a nonmember function that constructs an object is an alternative. Consider:

```
template<class T> class X {
    // ...
    template<class A> X(const A& a);
};
```

In the absence of member templates, we must restrict ourselves to specific types:

```
template<class T> class X {
    // ...
    X(const A1& a);
    X(const A2& a);
    // ...
};
```

Most early implementations generated definitions for all member functions defined within a template class when that template class was instantiated. This could lead to errors in unused member

functions (§C.13.9.1). The solution is to place the definition of the member functions after the class declaration. For example, rather than

```
template<class T> class Container {
    // ...
public:
    void sort() { /* use < */ }      // in-class definition
};

class Glob { /* no < for Glob */ };

Container<Glob> cg; // some pre-standard implementations try to define Container<Glob>::sort()
```

use

```
template<class T> class Container {
    // ...
public:
    void sort();
};

template<class T> void Container<T>::sort() { /* use < */ }

class Glob { /* no < for Glob */ };   // out-of-class definition

Container<Glob> cg; // no problem as long as cg.sort() isn't called
```

Early implementations of C++ did not handle the use of members defined later in a class. For example:

```
template<class T> class Vector {
public:
    T& operator[](size_t i) { return v[i]; }  // v declared below
    // ...
private:
    T* v;                                     // oops: not found!
    size_t sz;
};
```

In such cases, either sort the member declarations to avoid the problem or place the definition of the member function after the class declaration.

Some pre-standard-C++ implementations do not accept default arguments for templates (§13.4.1). In that case, every template parameter must be given an explicit argument. For example:

```
template<class T, class LT = less<T> > class map {
    // ...
};

map<int> m;                 // Oops: default template arguments not implemented
map< int, less<int> > m2;   // workaround: be explicit
```

B.3.6 For-Statement Initializers

Consider:

```
void f(vector<char>& v, int m)
{
    for (int i= 0; i<v.size() && i<=m; ++i) cout << v[i];

    if (i == m) {    // error: i referred to after end of for-statement
        // ...
    }
}
```

Such code used to work because in the original definition of C++, the scope of the controlled variable extended to the end of the scope in which the *for-statement* appears. If you find such code, simply declare the controlled variable before the *for-statement*:

```
void f2(vector<char>& v, int m)
{
    int i= 0;   // i needed after the loop
    for (; i<v.size() && i<=m; ++i) cout << v[i];

    if (i == m) {
        // ...
    }
}
```

.