ISO/IEC JTC1/SC22/WG21/N0007
X3J16/91-0089

C++: as close as possible to C—but no closer

89-05-11

Andrew Koenig
Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

ANSI C and the C subset of C++ serve subtly different purposes.

The purpose of X3J11 is to provide a standard: to codify existing practice and resolve inconsistencies among existing implementations. The purpose of C++ is to provide C programmers with a tool they can use to shape their thinking in fundamentally different ways. Both aimed at compatibility with "Classic C" and came close to hitting their mark.

The two goals have necessarily resulted in some fundamental differences of approach between the two languages. In a few cases C++ departed slightly from "Classic C" – always with knowledge of the cost of doing so and always with the aim to gain something well worth that cost. X3J11 did the same according to its aims and constraints. Wherever possible, C++ has adopted the X3J11 modifications and resolutions.

The purpose of this note is to summarize the remaining differences between the draft ANSI C standard and C++, explain their motivation, and point out cases where these differences are less important than they might appear at first.

Below, C refers to C as defined by the draft ANSI C standard and C++ refers to C++ as defined in the 1989 draft C++ reference manual. We use the word "difference" to refer to something that is C but not C++. Things that can be done in C++ but not C are not interesting in this context unless they also somehow restrict C++ from expressing something that is C.

Note that in this context pure extensions of C provided by C++ are *not* incompatibilities.

2. Namespaces

C puts variables and structure tags in separate name spaces; C++ uses a single name space. The reason for this, of course, is that abstract data types – classes – are a crucial part of the foundation of C++ and it is important to be able to use them as naturally as if they were built-in types. Essentially every C++ program depends on this.

The place where it matters most – at least the place where people have complained the most – is when a library function deals with a structure with the same name. For instance:

```
struct stat {
        /* member declarations */
};
int stat(const char *, struct stat *);
```

The C++ language definition therefore has a compatibility wart to allow precisely this kind of thing. We believe this will smoothly accommodate most existing C usages while still allowing the economical expression C++ programmers have come to appreciate and depend on.

The only remaining difference between C and C++ in the name space area is that C++ does not allow a name to be declared as both a structure tag and a (different) `typedef` name in the same scope. For example:

```
struct stat {
        /* a bunch of stuff */
};
typedef int stat;
```

Allowing this construct in C++ would create serious problems with composition of header files

describing libraries since declarations of functions, variables, and classes then could undetectably change meaning as the result of header file inclusions. Note that C++ specifically does allow the following common C usage:

```
typedef struct A {
        /* member definitions */
} A;
```

3. Linkage

The major difference here is that C allows

```
extern void f();

main()
{
        f(3,4);     /* legal C, illegal C++ */
        g(5,6);     /* legal C, illegal C++ */
}
```

and C++ does not.

In C++, a function prototype with no arguments means that the function has no arguments, and it is an error to call it with arguments. The draft ANSI C standard lists this (mis)use of function declarations as obsolescent (3.9.4). We think C++ is an excellent place to institute this disappearance.

To declare a function `f` with no arguments in a way that unambiguously means the same thing in C and C++, say something like this:

```
int getcount(void);
```

Furthermore, in C++, a prototype is required for *any* call; an undeclared function cannot be called. This is absolutely fundamental to the type safety of C++; C requires a prototype only for varadic functions such as `printf()`.

4. Linkage Consistency

C++ requires the types of all objects and functions with external linkage to be consistent across separate compilations – and enforces this requirement. This implies that all prototypes for a function must agree (exactly) and that any structure tag used in the type of any object or function with external linkage must refer to the same structure (with the same name) in all files.

We are not convinced that this requirement is actually different from C but we are convinced that actual C practice is such that enforcing the requirement would be unacceptable because it would break too much code.

5. Linkage of const

Global variables declared `const` have external linkage by default in C; in C++ such `const`s have internal linkage by default. The reason for this is to avoid having to allocate memory for things that, in our experience, are usually intended as the equivalent of preprocessor macros and to allow systematic use of integer `const`s in constant expressions. For example:

```
const SIZE = 100;
int table[SIZE];          /* legal C++, illegal C */
```

Again this is not as much of a problem as one might expect. It makes no difference, of course, if a constant is only defined and used in a single file or if the `const` is local. If the same constant is defined in several files, the programmer will have to declare it `static` anyway to

avoid multiple definition errors from the linker or declare it `extern` in all files but one. All uses of `const` where explicit `static` or `extern` is used are compatible as are all uses of local `const`s.

The only case to look out for is:

```
file1:
      const a = 1;

file2:
      extern const a;
```

which will not link in C++ because no definition will be found for the a referenced in file2. The following modification makes the C program acceptable to a C++ compiler:

```
file1:
      extern const a = 1;

file2:
      extern const a;
```

6. Keywords

C++ has a few extra keywords: `asm`, `catch`, `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, and `virtual`. This can't be helped; one cannot add fundamental concepts to a language in a reasonable way without introducing words to refer to those concepts. To avoid chaos, such words must be reserved in languages such as C and C++.

7. Miscellaneous

The differences mentioned above are the most important because they affect the interface between C and C++ programs and limit – if only insignificantly – what can be expressed in header files shared by the the two languages. The relative insignificance of these limitations can be seen from the fact that all the ANSI C standard library header files are also legal C++ header files.

Other differences are limited to individual source files and are less important since no significant C++ program can pass a C compiler anyway.

7.1 Assignment of `void*`

C not only allows any object pointer to be assigned to a `void*` but also a `void*` to be assigned to any object pointer. This opens a blatant hole in the type system that was not present in classic C.

We surmise that the reason this is considered acceptable is that C already has a worse hole in its type system in the form of unchecked function arguments and because it provides the C programmer the convenience of not having to cast the results of calling `malloc()`, `calloc()`, etc. Opening this hole is not acceptable in C++ where reliance on the type system is greater. In C++, only the harmless assignment of any object pointer to a `void*` (and not the opposite assignment) is accepted. Furthermore, since C++ programmers use operator `new` in preference to using `malloc()`, etc., directly, the hole in the type system would provide no extra convenience in C++.

In a similar vein, C++ does not allow a pointer to a `const` object to be assigned to `void*`: the program must use a `const void*` instead. This is to catch things like:

```
extern "C" {
        int read(int, void*, int);
        int write(int, const void*, int);
}

const char filename[] = "/etc/passwd";

void f()
{
        read(0, filename, sizeof(filename));        // read into a const?
}
```

If an arbitrary const pointer could be freely assigned to `void*`, there would be no way for the compiler to detect that this program fragement tries to read into a constant array.

Of course a C++ program can use a cast to convert a `void*` to or from any other kind of pointer.

7.2 Type of 'a'

The type of character constants in C++ is `char` instead of `int`. However, since the rules for determining the integer value of a character constant are identical in C and C++ the only way to detect this in a C program is with an expression like `sizeof('a')`. In C++, however, it is essential for the overloaded function resolution mechanism to resolve `'a'` as a `char` so that

```
cout << 'a';
```

can print a instead of `97`.

For the same reason, the type of an enumerator is the type of its enumeration. This *may* lead to an incompatibility since, given

```
enum e { A, B };
```

`sizeof(A)==sizeof(enum e)` and `sizeof(enum e)` are not guaranteed to be equal to `sizeof(int)` in either C or C++.

7.3 Repeated definition

In C++, a "plain" global object declaration (without `extern` or an initializer) is a definition. Two of those in a file give a double definition error. For example:

```
int i;
int i;
```

In C, this is accepted. The reason for the difference is again the uniform treatment of built-in and user-defined types. Suppose `Int` is a class for which a constructor taking no arguments has been declared:

```
Int i;
Int i;
```

Each declaration requires a call of the constructor and each places that call in the sequence of such calls to be executed for the file. Deciding that only one of the declarations is "real" and ignoring the other not only adds complexity to C++ compilers and/or linkers but can also introduce dependency errors into the dynamic initialization.

7.4 Char Array Initialization

For some reason C has come to allow the previously illegal initialization

```
char v[3] = "asd";
```

Allowing this violates the rule that strings are terminated by `'\0'` so C++ still rejects it. The

way to initialize a bounded character array that is not intended to be used as a string is to initialize the individual elements:

```
char v[3] = { 'a', 's', 'd' };
```

7.5 goto Skipping Initialization

The following is legal C, but not C++:

```
void f()
{
        /* ... */
        goto ll;
        {
                int a = 7;
                String b = "asdf";
        ll:
                /* ... */
        }
        /* ... */
}
```

In C, this is merely dangerous and bad style. In C++, it would with great regularity cause core dumps; that innocuous looking `String` might be a class for which a destructor will be called on exit from `f()`.

7.6 enum Assignment

In C, an `int` may be assigned to a variable of an enumeration type. For example:

```
enum e { A, B };

enum e obj1 = 7;

enum e obj2 = 257; /* what if an 'e' is represented by a char? */
```

C leaves the meaning of many such assignments implementation dependent and the ANSI C manual recommends a warning against all such assignments. C++ prohibits them. As usual, casting can defeat type checking:

```
enum e obj3 = (enum e) 7;       /* caveat emptor */
```

7.7 Local enums

In C, an `enum` declared within a `struct` is essentially useless and the enumerators are given the same scope as the `struct` enclosing them. For example:

```
struct s {
        enum e { a, b } ee;
};

int a;              /* C error: 'a' in scope */
int aa = b; /* C++ error: 'b' not in scope */
```

In C++, with its notion that a class establishes a scope, they are very useful, kept in the scope of their class, and accessible elsewhere – subject to access control – by explicit qualification with their class name:

```
class X {
public:
        enum state { good, bad, fail };
        e readstate();
        // ...
private:
        some_operation() { state s = good; /* ... */ }
        // ...
};

void f(X& x)
{
        while (x.readstate() == X::good) // ...
}
```

Note that although the *enumerators* are kept in the scope of a class, the *name* of the enumeration itself is exported into the surrounding scope. This permits a class to define and export an enumerated type whose enumerators are kept secret from the users of the class.

7.8 Comments

It was believed that the introduction of // comments in C++ did not lead to any incompatibilities. Here is a counter example:

```
main()
{
        int a = 4;
        int b = 8//* divide by a*/a;
        +a;
}
```

Note that the use of a prefix operator starting the line is essential, as is the absence of whitespace. We do not consider this incompatibility serious enough to abandon either style of comments.

We note in passing that it is important to cater to // comments in the preprocessor too, lest the following evoke surprising preprocessor complaints:

```
#include <stdio.h>

int flag;    // remember if we have called getc()
```

8. Conclusion

We have tried to summarize the differences between ANSI C and C++. In the design of C++, we have been trying to keep the differences as minor as possible. We believe that we have succeeded in this beyond reasonable expectations and that differences that remain are unimportant to C programmers, essential to C++ programmers and stem from the somewhat different purposes behind C and C++.

9. Acknowledgements

We would like to thank the many C and C++ users who have commented on the relationship between C and C++ and made suggestions about both the ideals and the thorny details of this sensitive topic. In particular we would like to thank Doug McIlroy, David Prosser, and Margaret Quinn.