Type-safe Linkage for C++

Bjarne Stroustrup AT&T Bell Laboratories

ABSTRACT: This paper describes the problems involved in generating names for overloaded functions in C++ and in linking to C programs. It also discusses how these problems relate to library building. It presents a solution that provides a degree of type-safe linkage. This eliminates several classes of errors from C++ and allows libraries to be composed more freely than has hitherto been possible. Finally the current encoding scheme for C++ names is presented.

1. Introduction

This paper describes the type-safe linkage scheme used by the 2.0 release of C++ and the mechanism provided to allow traditional (unsafe) linkage to non-C++ functions. It describes the problems with the scheme used by previous releases, the alternative solutions considered, and the practicalities involved in converting from the old linkage scheme to the new.

A previous version of this paper appeared in the *Proceedings of the USENIX 1988 C++ Conference* (Denver).

The new scheme makes the `overload` keyword redundant, simplifies the construction of tools operating on C++ object code, makes the composition of C++ libraries simpler and safer, and enables reliable detection of subtle program inconsistencies. The scheme does not involve any run-time costs and does not appear to add measurably to compile or link time.

The scheme is compatible with older C++ implementations for pure C++ programs but requires explicit specification of linkage requirements for linkage to non-C++ functions.

2. *The Original Problem*

C++ allows overloading of function names; that is, two functions may have the same name provided their argument types differ sufficiently for the compiler to tell them apart. For example,

```
double sqrt(double);
complex sqrt(complex);
```

Naturally, these functions must have different names in the object code produced from a C++ program. This is achieved by suffixing the name the user chose with an encoding of the argument types (the *signature* of the function). Thus the names of the two `sqrt()` functions become:

```
sqrt__Fd         // sqrt that takes a double argument
sqrt__F7complex  // sqrt that takes a complex argument
```

Some details of the encoding scheme are described in Appendix A.

When experiments along this line began five years ago it was immediately noticed that for many sets of overloaded functions there was exactly one function of that name in the standard C library. Since C does not provide function name overloading there could not be two. It was deemed essential for C++ to be able to use the C libraries without modification, recompilation, or indirection. Thus the problem became the design of an overloading facility for C++ that allowed calls to C library functions such as `sqrt()` even when the name `sqrt` was overloaded in the C++ program.

3. The Original Solution

The solution, as used in all non-experimental C++ implementations up to now, was to let the name generated for a C++ function be the same as would be generated for a C function of the same name whenever possible. Thus `open()` gets the name `open` on systems where C doesn't modify its names on output, the name `_open` on systems where C prepends an underscore, etc.

This simple scheme clearly isn't sufficient to cope with overloaded functions. The keyword `overload` was introduced to distinguish the hard case from the easy one and also because function name overloading was considered a potentially dangerous feature that should not be accidentally or implicitly applied. In retrospect this was a mistake.

To allow linkage to C functions the rule was introduced that only the second and subsequent versions of an overloaded function had their names encoded. Thus the programmer would write

```
overload sqrt;
double sqrt(double);      // sqrt
complex sqrt(complex);    // sqrt__F7complex
```

and the effect would be that the C++ compiler generated code referring to `sqrt` and `sqrt__F7complex`. This enabled a C++ programmer to use the C libraries. This trick solves the problems of name encoding, linkage to C, and protection against accidental overloading, but it is clearly a hack. Fortunately, it was documented only in the *BUGS* section of the C++ manual page.

4. Problems with the Original Solution

There are at least three problems with this scheme:

- How to name overloaded functions so that one may be a C function.
- How to detect errors caused by inconsistent function declarations.

- How to specify libraries so that several libraries can be easily used together.

4.1 The overload Linkage Problem

Consider a program that uses an overloaded function `print()` to output `globs` and `widgets`. Naturally `globs` are defined in `glob.h` and `widgets` in `widget.h`. A user writes

```
// file1.c:
#include <glob.h>
#include <widget.h>
```

but this elicits an error message from the C++ compiler since `print()` is declared twice with different argument types. The user then modifies the program to read

```
// file1.c:
overload print;
#include <glob.h>
#include <widget.h>
```

and all is well until someone in some other part of the program writes

```
// file2.c:
overload print;
#include <widget.h>
#include <glob.h>
```

This fails to link since the object code file produced from `file1.c` refers to `print` (meaning `print(glob)`) and `print__F6widget`, whereas the output from `file2.c` refers to `print` (meaning `print(widget)`) and `print__F4glob`.

This is of course a nuisance, but at least the program fails to link and the programmer can – after some detective work based on relatively uninformative linker error messages – fix the problem. The nastier variation of this will happen to the conscientious programmer who knows that `print()` is overloaded and inserts the appropriate `overload` declarations, but happens to use only one variation of `print()` in each of two source files:

```
// file1.c:
overload print;
#include <glob.h>

// file2.c:
overload print;
#include <widget.h>
```

The output from `file1.c` and `file2.c` now both refer to `print`. Unfortunately, in the output from file1.c `print` means `print(glob)` whereas `print` refers to `print(widget)` in the output from file2.c. One might expect linkage to fail because `print()` has been defined twice. However, on most systems this is not what happens in the important case where the definitions of `print(glob)` and `print(widget)` are placed in libraries. Then, the linker simply picks the first definition of `print()` it encounters and ignores the second. The net effect is that calls (silently) go to the wrong version of `print()`. If we are lucky, the program will fail miserably (core dump); if not, we will simply get wrong results.

The requirement that the `overload` keyword must be used and the non-uniform treatment of overloaded functions ("the first overloaded function has C linkage") is a cause of complexity in C++ compilers and in other tools that deal with C++ program text or with object code generated by a C++ compiler.

4.2 The General Linkage Problem

This problem of inconsistent linkage is a variation of the general problem that C provides only the most rudimentary facilities for ensuring consistent linkage. For example, even in ANSI C and in C++ (until now) the following example will compile and link without warning:

```
#include <stdio.h>
extern int sqrt(int);

main()
{
    printf("sqrt(%d) == %d\n",2,sqrt(2));
}
```

and produce output like this

```
sqrt(2) == 0
```

because even though the user clearly specified that an integer `sqrt()` was to be used, the C compiler/linker uses the double precision floating point `sqrt()` from the standard library. This problem can be handled by consistent and comprehensive use of correct and complete header files. However, that is not an easy thing to achieve reliably and is not standard practice. The traditional C and C++ compiler/linker systems do not provide the programmer with any help in detecting errors, oversights, or dangerous practices.

These linkage problems are especially nasty because they increase disproportionally with the size of programs and with the amount of library use.

4.3 Combining Libraries

The standard header `complex.h` overloads `sqrt()`:

```
// complex.h:
overload sqrt;
#include <math.h>
complex sqrt(complex);
```

Some other header, `3d.h`, declares `sqrt()` without overloading it:

```
// 3d.h:
#include <math.h>
```

Now a user wants both the 3d and the complex number packages in a program:

```
#include <3d.h>
#include <complex.h>
```

Unfortunately this does not compile because of this sequence of operations:

```
double sqrt(double); // from <math.h> via <3d.h>
overload sqrt;       // from <complex.h>
```

A function that is to be overloaded must be explicitly declared overloaded before its first declaration is processed. So the programmer, who really did not want to know about the internals of those headers, must reorder the `#include` directives to get the program to compile:

```
#include <complex.h>
#include <3d.h>
```

This will work unless `3d.h` overloads some function, say `atan()`, that `complex.h` does not. Even in that case the programmer can cope with the problem by adding sufficient `overload` declarations where `3d.h` and `complex.h` are included:

```
overload sqrt;
overload atan;
#include <3d.h>
#include <complex.h>
```

This reordering and/or adding of `overload` declarations is irrelevant to the job the programmer is trying to do. Worse, if the extra `overload` declarations were placed in a header file the programmer has now set the scene for the users of the new package to have exactly the same problems when they try combining this new library with other libraries. It becomes tempting to overload all functions or at least to provide header files that overload all interesting functions. This again defeats any real or imagined benefits of requiring explicit `overload` declarations.

5. *A General Solution*

The overloading scheme used for C++ (until now) interacts with traditional C linkage scheme in ways that bring out the worst in both. Overloading of function names, which was introduced to provide notational convenience for programmers, is becoming a noticeable source of extra work and complexity for builders and users of libraries. Either the idea of overloading is bad or else its implementation in C++ is deficient. The insecure C linkage scheme is a source of subtle and not-so-subtle errors. In summary:

1. Lack of type checking in the linker causes problems.
2. Use of the `overload` keyword causes problems.

3. We must be able to link C++ and C program fragments.

A solution to 1 is to augment the name of *every* function with an encoding of its signature. A solution to 2 is to cease to require the use of `overload` (and eventually abolish it completely). A solution to 3 is to require a C++ programmer to state explicitly when a function is supposed to have C-style linkage.

The question is whether a solution based on these three premises can be implemented without noticeable overhead and with only minimal inconvenience to C++ programmers. The ideal solution would

- require no C++ language changes;
- provide type-safe linkage;
- allow for simple and convenient linkage to C;
- break no existing C++ code;
- allow use of (ANSI style) C headers;
- provide good error detection and error reporting;
- be a good tool for library building;
- impose no run-time overhead;
- impose no compile time overhead;
- impose no link time overhead.

We have not been able to devise a scheme that fulfills all of these criteria strictly, but the adopted scheme is a good approximation.

5.1 Type-safe C++ Linkage

First of all, every C++ function name is encoded by appending its signature. This ensures that a program will load only provided every function called has a definition and that the argument types specified in declarations used to compile calls are the same as the types specified in the function definition. For example, given:

```
f(int i) { ... }          // f__Fi
f(int i, char* j) { ... } // f__FiPc
```

These examples will cause correct linkage:

```
extern f(int);        // f__Fi   - links to f(int)
f(1);

extern f(int,char*); // f__FiPc - links to f(int,char*)
f(1,"asdf");
```

These examples will cause linkage errors independent of where in the program they occur because no `f()` with a suitable signature has been defined:

```
// no declaration of f() in this file
// (this is legal only in C programs)
f(1);                // f      - links to ???

extern f(char*);     // f__FPc - links to ???
f("asdf");

extern f(int ...);   // f__Fie - links to ???
f(1,"asdf");
```

One might consider extending this encoding scheme to include global variables, etc., but this does not appear to be a good idea since that would introduce at least as many problems as it would solve. For example:

```
// file1.c:
int aa = 1;
extern int bb;

//file2.c:
char* aa = "asdf"; // error: aa declared int in file1.c
extern char* bb;   // error: bb declared int in file1.c
```

Under the current C scheme, the double definition of `aa` will be caught and the inconsistent declarations of `bb` will not. Using an encoding scheme, the double definition of `aa` would not be caught since the difference in encoding would cause *two* differently named objects to be created – contrary to the rules of C and C++. The fact that the inconsistent declarations of `bb` would be caught by some linkers (not all) does not compensate for the incorrect linkage of `aa`. Consequently only functions are encoded using their signatures.

For a similar reason function argument types are *not* encoded (except for pointer to argument types):

```
// hypothetical encoding using return types:

// file1.c:
int f() { ... };            // f__Fv_i

//file2.c:
char* f();                  // f__Fv_Pc
```

Here a linker would report `f()` undefined because of the name mismatch. This could be quite confusing.

The adopted linkage scheme is much safer than what is currently used for C, but it cannot detect all linkage problems. For example, if two libraries each provides a function `f(int)` as part of their public interface there is no mechanism that allows the compiler to detect that there are supposed to be two different `f(int)`s. If the .o files are loaded together the linker will detect the error, but when a library search mechanism is employed the error may go undetected.

Note that this linking scheme simply enforces the C++ rules that every function must be declared before it is called and that every declaration of an external name in C++ must have exactly the same type.

In essence, we use the name encoding scheme to "trick" the linker into doing type checking of the separately compiled files. More comprehensive solutions can be achieved by modifying the linker to understand C++ types. For example, a linker could check the types of global data objects and the return types of functions. It might also provide features for ensuring the consistency of global constants and classes. However, getting an improved linker into use is typically a hard and slow process. The scheme presented here is portable across a great range of systems and can be used immediately.

5.2 Implicit Overloading

If a function is declared twice with different argument types it is overloaded. For example:

```
double sqrt(double);
complex sqrt(complex);
```

is accepted without any explicit `overload` declaration. Naturally, `overload` declarations will be accepted in the foreseeable future; they are simply not necessary any more.

Does this relaxation of the C++ rules cause new problems? It does not appear to. For example, originally I imagined that obvious mistakes such as

```
double sqrt(double);      // sqrt__Fd
double d = sqrt(2.3);

double sqrt(int d) { ... }// sqrt__Fi
```

would cause hard-to-find errors. It certainly would with the traditional C linkage rules, but with type-safe linkage the program simply will not link because there is no function called `sqrt__Fd` defined anywhere. Even the standard library function will not be found because its name is as always "plain" `sqrt`.

Another imagined problem was that a call

```
f(x);
```

would suddenly change its meaning when a function became overloaded by the inclusion of a new header file containing the declaration of another function `f()`. The only case where `f(x)` can have its meaning changed by the introduction of a new declaration `f(T)` is where `T` is the type of `x`. In this case the meaning of `f(x)` ought to change. In all other cases, the C++ ambiguity rules ensure that the introduction of a new `f()` will either leave the meaning of `f(x)` unchanged (when the new `f()` is unrelated to the type of `x`) or will cause a compile time error (when an ambiguity is introduced).

5.3 C Linkage

This leaves the problem of how to call a C function or a C++ function "masquerading" as a C function. To do this a programmer must state that a function has C linkage. Otherwise, a function is assumed to be a C++ function and its name is encoded. To express this an extension of the "extern" declaration is introduced into C++:

```
    extern "C" {
        double sqrt(double); // sqrt(double) has C linkage
    }
```

This linkage specification does not affect the semantics of the program using `sqrt()` but simply tells the compiler to use the C naming conventions for the name used for `sqrt()` in the object code. This means that the name of *this* `sqrt()` is `sqrt` or `_sqrt` or whatever is required by the C linkage conventions on a given system. One could even imagine a system where the C linkage rules were the type-safe C++ linkage rules as described above so that the name of `sqrt()` was `sqrt__Fd`.

Linkage specifications nest, so that if we had other linkage conventions, such as Pascal linkage, we could write:

```
                    // default: C++ linkage here
    extern "C" {
                    // C linkage here
        extern "Pascal" {
                    // Pascal linkage here
            extern "C++" {
                    // C++ linkage here
            }
                    // Pascal linkage here
        }
                    // C linkage here
    }
                    // C++ linkage here
```

Such nestings will typically occur as the result of nested `#includes`.

The `{}` in a linkage specification does *not* introduce a new scope; the braces are simply used for grouping. This use of `{}` strongly resembles their use in enumerations.

The keyword `extern` was chosen because it is already used to specify linkage in C and C++. Strings (for example, `"C"` and `"C++"`) were chosen as linkage specifiers because identifiers (e.g. `C` and `Cplusplus`) would de facto introduce new keywords into the language and because a larger alphabet can be used in strings.

Naturally, only one of a set of overloaded functions can have C linkage, so the following causes a compile time error:

```
extern "C" {
   double sqrt(double);
   complex sqrt(complex);
}
```

Note that C linkage can be used for C++ functions intended to be called from C programs as well as for C functions. In particular, it is necessary to use C linkage for C++ functions written to implement standard C library functions for use by C programs. However, using the encoded C++ name from C preserves type-safety at link time. This technique can be valuable in other languages too. I have already seen an example of the C++ scheme applied to assembly code to prevent nasty link errors for low level routines. One might consider using this C++ linkage scheme for C also, but I suspect that the sloppy use of type information in many C programs would make that too painful.

In an "all C++" environment no linkage specifications would be needed. The linkage mechanism is intended to ease integration of C++ code into a multi-lingual system.

5.4 Caveat

One could extend this linkage specification mechanism to other languages such as Fortran, Lisp, Pascal, PL/1, etc. The way such an extension is done should be considered very carefully because one "obvious" way of doing it would be to build into a C++ compiler the full knowledge of the type structure and calling conventions of such "foreign" languages. For example, a C++ compiler *might* handle conversion of zero-terminated C++ strings into Pascal strings with a length prefix at the call point of a function with Pascal linkage and *might* use Fortran call by reference rules when calling a function with Fortran linkage, etc.

There are serious problems with this approach:

- The complexity and speed of a C++ compiler could be seriously affected by such extensions.
- Unless an extension is widely available and accepted programs using it will not be portable.
- Two implementations might "extend" C++ with a linkage specification to the same "foreign" language, say Fortran, in

different ways so as to make identical C++ programs have subtly different effects on different implementations.

Naturally, these problems are not unique to linkage issues or to this approach to linkage specification.

I conjecture that in most cases linkage from C++ to another language is best done simply by using a common and fairly simple convention such as "C linkage" plus some standard library routines and/or rules for argument passing, format conversion, etc., to avoid building knowledge of non-standard calling conventions into C++ compilers. This ought to be simpler from C++ than from most other languages. For example, reference type arguments can be used to handle Fortran argument passing conventions in many cases and a Pascal string type with a constructor taking a C style string can trivially be written. Where extension are unavoidable, however, C++ now provides a standard syntax for expressing them.

6. *Experience*

The natural first reaction to this scheme is to look for a way of handling linkage and overloading without requiring explicit linkage specifications. We have not been able to come up with a system that enabled C linkage to be implicit without serious side effects. I will summarize the advantages of the adopted scheme here and discuss several possible objections to it. Section 7 below describes alternative schemes that were considered and rejected.

6.1 *Making Linkage Specifications Invisible*

One obvious advantage of this scheme is that it allows a programmer to give a set of functions C linkage with a single linkage specification without modifying the individual function declarations. This is particularly useful when standard C headers are used. Given a C header (that is, an ANSI C header with function prototypes, etc.),

```
// C header:
// C declarations
```

one can trivially modify the header for use from C++:

```
// C++ header:

extern "C" {
      // C header:
      // C declarations
}
```

This creates a C++ header that cannot be shared with C.

Sharing with C can be achieved using `#ifdef`:

```
// C and C++  header:

#ifdef __cplusplus
extern "C" {
#endif
      // C header:
      // C declarations
#ifdef __cplusplus
}
#endif
```

where `__cplusplus` is defined by every C++ compiler.

In cases where one for some reason cannot or should not modify the header itself one can use an indirection:

```
// C++ header:

extern "C" {
#include "C_header"
}
```

Fortunately, such transformations can be done by trivial programs so that most of the effort in converting C headers need not be done by hand.

It was soon discovered that even though programmers tend to scatter function declarations throughout the C++ program text, most C functions actually come from well-defined C libraries for which there are – or ought to be – standard header files.

Placing all of the necessary linkage specifications in standard header files means that they are not seen by most users most of the time. Except for programmers studying the details of C library interfaces, programmers installing headers for new C libraries for C++ users, and programmers providing C++ implementations for C interfaces, the linkage specifications are invisible.

6.2 Error Handling

The linker detects errors, but reports them using the names found in the object code. This can be compensated for by adding knowledge about the C++ naming conventions to the linker or (simpler) by providing a filter for processing linker error messages. This output was produced by such a filter:

```
C++ symbol mapping:

PathListHead::~PathListHead() __dt__12PathListHeadFv
Path_list::sepWork()          sepWork__9Path_listFv
Path::pathnorm()              pathnorm__4PathFv
Path::operator&(Path&)        __ad__4PathFR4Path
Path::first()                 first__4PathFv
Path::last()                  last__4PathFv
Path::rmfirst()               rmfirst__4PathFv
Path::rmlast()                rmlast__4PathFv
Path::rmdots()                rmdots__4PathFv
Path::findpath(String&)       findpath__4PathFR6String
Path::fullpath()              fullpath__4PathFv
```

Introducing this filter had the curious effect of replacing the usual complaint about "ugly C++ names" with complaints that the linker didn't provide enough information about C functions and global data objects.

The reason for presenting the encoded and unencoded names of undefined functions side by side is to help users who use tools, such as debuggers, that haven't yet been converted to understand C++ names.

A plain C debugger such as `sdb`, `dbx`, or `codeview` can be used for C++ and will correctly refer to the C++ source, but it will use the encoded names found in the object code. This can be avoided by employing a routine that "reverses" the encoding, that is, reads an encoded name and extracts information from it.[1] The

1. Naturally, this would be the same function that was used to write the linker output filter. The examples here are based on the name decoding routine written by Steve Brandt and used to modify the UNIX System V C debugger `sdb` into `sdb++`.

encoding scheme is described in Appendix A. A C++ name decoder should be generally available for use by debugger writers and others who deal directly with object code. Until such decoders are in widespread use the programmer must have at least a minimal understanding of the encoding scheme.

6.3 Upgrading Existing C++ Programs

Decorating the standard header files with the appropriate linkage specifications had two effects. The first phenomenon observed was that most of the declarations scattered in the program text that were referring to C functions were either redundant (because the function had already been declared in a header) or at least potentially incorrect (because they differed from the declaration of that header file on some commonly used system). The second phenomenon observed was that every non-trivial program converted to the new linkage system contained inconsistent function declarations. A noticeable number of declarations found in the program text were plain wrong, that is, different from the ones used in the function definition. This was caused in part by sloppiness, for example, where a programmer had declared a function

```
char* f(int ...);
```

to suppress compiler warnings instead of looking up the type of the second argument. A more common problem was that the "standard" header files had changed since the function declaration was placed in the text so that the "local" declaration didn't match any more; this often happens when a file is transferred from one system to another, say from a BSD to a System V.

In summary, introducing the new linkage system involved adding linkage specifications. Typically, these linkage specifications were only needed in standard header files. The process of introducing linkage specifications invariably revealed errors in the programs – even in programs that had been considered correct for years. The process strongly resembles trying `lint` on an old C program.

As was expected, some programmers first tried to get around the requirements for explicit C linkage by enclosing their entire program in a linkage directive. This might have been considered a fine way of converting old C++ programs with minimum effort had it not had the effect of ensuring that every program that uses facilities provided by such a program would also have to use the unsafe C linkage. To achieve the benefits from the new linkage scheme most C++ programs must use it. The requirement that at most one of a set of overloaded functions can have C linkage defeats this way of converting programs. The slightly slower and more involved method of using standard header files (already containing the necessary linkage specifications) and adding a few extra linkage specifications in local headers where needed must be used. This also has the benefit of unearthing unexpected errors.

7. *Details*

The scope of C function declarations has always been a subject for debate. In the context of C++ with linkage specifications and overloaded functions it seems prudent to answer some variations of the standard questions.

7.1 Default Linkage

Consider:

```
extern "C" {
    int f(int);
}

int f(int);         // default (C++ linkage) overruled:
                    // f() has C linkage
```

Is it the same `f()` that was defined with C linkage above and does it have C or C++ linkage? It is the same `f()` and it does (still) have C linkage. The first linkage specification "wins" provided the second declaration has "only" default (that is, C++) linkage.

Where linkage is explicitly specified for a function, that specification must agree with any previous linkage. For example:

```
extern "C" {
    int f(int);  // f() has C linkage
}
```

```
int g();          // default: g() has C++ linkage
int f(int);       // fine: default overruled,
                  //       f() has C linkage

extern "C" {
    int f(int);   // fine
    int g();      // error: inconsistent
                  //        linkage specification
}
```

The reason to require agreement of explicit linkage specifications is to avoid unnecessary order dependencies. The reason to allow a second declaration with implicit C++ linkage to take on the linkage from a previous explicit linkage specification is to cope with the common case where a declaration occurs both in a `.c` file and in a standard header file.

7.2 Declarations in Different Scopes

Consider:

```
extern "C" {
    int f(int);
}

void g1()
{
    int f(int);
    f(1);
}
```

Is the `f()` declared local to `g1` the same as the global `f()` and does the function called in `g1()` have C linkage? It is the same `f()` and it does have C linkage.

Consider:

```
extern "C" {
    int f(int);
}

void g2()
{
    int f(char*);
    f(1);
    f("asdf");
}
```

Does the local declaration of `f()` overload the global `f()` or does it hide it? In other words, is the call `f(1)` legal? That call is an error because the local declaration introduces a new `f()` that hides the global `f()`. In the tradition of C, the declaration of `f(char*)` also draws an warning.

Consider:

```
void g3()
{
    int ff(int);
};

void g4()
{
    int ff(char*);
    ff("asdf");
    ff(1);
};
```

Does the second declaration of `ff()` overload the first? In other words, is the call `ff(1)` legal? The call is an error and a warning is issued about the two declarations of `ff()` because (as in the example above) overloading in different scopes is considered a likely mistake.

7.3 Local Linkage Specification

Linkage specifications are *not* allowed inside function definitions. For example:

```
void g5()
{
    extern "C" { // error: linkage specification
        int h(); //        in function
    }
}
```

The reason for this restriction is to discourage the use of local declarations of C functions and to simplify the language rules.

8. *Alternative Solutions*

So, the linkage specification scheme works, but isn't there a better way of achieving the benefits of that scheme? Several schemes were considered. This section presents the first two or three alternatives people usually come up with and explains why we rejected them. Naturally, we also considered more and weirder solutions, but all the plausible ones were variations of the ones presented here.

8.1 The Scope Trick

The first attempt to provide type-safe linkage involved the use of `overload` and the C++ scope rules. All overloaded function names were encoded, but non-overloaded function names were not. This scheme had the benefit that the linkage rules for most functions were the C linkage rules – and had the problem that those rules are unsafe. The most obvious problem was that at first glance there is no way of linking an overloaded function to a standard C library function. This problem was handled using a "scope trick":

```
overload sqrt;
complex sqrt(complex);
inline double sqrt(double d)
{
   extern double sqrt(double); // A new sqrt()
                               // not overloaded

   return sqrt(d);             // not a recursive call
                               // but a call of the C
                               // function sqrt
}
```

In effect, we provided a C++ calling stub for the C function `sqrt()`. The snag is that having thus *defined* `sqrt(double)` in a standard header a user cannot provide an alternative to the standard version. The problems with library combination in the presence of `overload` are not addressed in this scheme, and are actually made worse by the proliferation of definitions of overloaded functions in header files. In particular, if two "standard" libraries each overload a function then these two libraries cannot be used together since that function will be defined twice: once in each of the two standard headers.

There is also a compile time overhead involved. In retrospect, I consider this scheme somewhat worse than the original "the first overloaded has C linkage" scheme.

8.2 C "storage class"

It is clear that the definitions providing a calling stub are redundant. We could simply provide a way of stating that a member of a set of overloaded functions should be a C function. For example:

```
complex sqrt(complex);
cdecl double sqrt(double); //sqrt(double) has C linkage
```

This is equivalent to

```
complex sqrt(complex);
extern "C" {
   double sqrt(double);
}
```

but less ugly. However, it involves complicating the C++ language with yet another keyword. Functions from other languages will have to be called too and they each have separate requirements for linkage so the logical development of this idea would eventually make `ada`, `fortran`, `lisp`, `pascal`, etc., keywords. Using a keyword also requires modification of the declarations of the C functions and those are exactly the declarations we would want *not* to touch since they will typically live in header files shared with an ANSI C compiler. In some cases we would even like not to touch a file in which such declarations reside.

8.3 Overload "storage class"

The use of a keyword to indicate that a function is a C function is logically very similar to the linkage specification solution, though inferior in detail. An alternative is to have a keyword indicate that a function should have its signature added. The keyword `overload` might be used. For example:

```
overload complex sqrt(complex); // use C++ linkage
double sqrt(double);            // C linkage by default
```

This has the disadvantage that the programmer has to add information to gain type safety rather than having it as default and would de facto ensure that the C++ type-safe linkage rules would be used only for overloaded functions. Furthermore, this would mean that libraries could be combined only if the designers of these libraries had decorated all the relevant functions with `overload`. This scheme also invalidates all old C++ programs without providing significant benefits.

8.4 Calling Stubs

One way of dealing with C linkage would be *not* to provide any facilities for it in the C++ language, but to require every function called to be a C++ function. To achieve this one would simply re-compile all libraries and have one version for C and another for C++. This is a lot of work, a lot of waste, and not feasible in general. In the cases where recompilation of a C program as a C++ program is not a reasonable proposition (because you don't have the source, because you cannot get the program to compile, because you don't have the time, because you don't have the file space to hold the result, etc.) you can provide a small dummy C++ function to call the C function. Such a function would be written in C (for portability) or in assembler (for efficiency). For example:

```
double sqrt__Fd(d) double d;
{               /* C calling stub for sqrt(double): */
    extern double sqrt();
    return sqrt(d);
}
```

A program can be provided to read the linker output and produce the required stubs.

This scheme has the advantage that the user works in what appears to be an "all C++" environment (but so does the adopted scheme once a few C libraries have been recompiled with C++ and/or a few header files have been decorated with linkage specifications). It does, however, also suffer from a few severe disadvantages. A "C calling stub maker" program cannot be written portably. Therefore, it would become a bottleneck for porting

C++ implementations and C++ programs and thus a bottleneck for the use of C++. It is also not clear that this approach can be implemented everywhere without loss of efficiency since it requires large numbers of functions to have two names (a C name and a C++ name). This takes up code space and introduces large numbers of extra names that would slow down programs reading object files such as linkers, loaders, debuggers, etc. The C calling interfaces would also be ubiquitous and available for anyone to use by mistake, thus re-introducing the C linkage problems in a new guise.

8.5 Encode only C++ Functions

The fundamental problem with all but the last scheme outlined above is that they require the programmer to decorate the source code with directives to help the compiler determine which functions are C functions. Ideally, the compiler would simply look at the program and determine the linkage necessary for each individual function based on its type. Could the compiler be that smart? Unfortunately, no. There is no way for the compiler to know whether

```
extern double sqrt(double);
```

is written in C or C++. However, one might handle most cases by the heuristic that if a function is clearly a C++ function it gets C++ linkage and if it isn't it gets C linkage. For example:

```
complex sqrt(complex); // clearly C++: sqrt__F7complex
double sqrt(double);   // could be C:  sqrt
```

Since `complex` is a class, `sqrt(complex)` is clearly a C++ function and it is encoded. The other `sqrt()` might be C so it isn't.

Applying this heuristic would mean that most functions would not have type-safe linkage – but we are used to that. It would also mean that overloading a function based on two C types would be impossible or require special syntax:

```
int max(int,int);
double max(double,double);
```

Such overloading *must* be possible because there are many such examples and several of those are important, especially when support for both single and double precision floating point arithmetic becomes widespread:

```
float sqrt(float);
double sqrt(double);
```

This implies that either `overload` or linkage specifications must be introduced to handle such cases. The heuristic nature of the specification of where these directives are needed will lead to confusion, overuse, and errors.

If `overload` is re-introduced, the cautious programmer will use it systematically wherever a relatively simple class is used (in case a revision of the system should turn it into a plain C struct), wherever an argument is typedef'd (because that `typedef` might some day refer to a plain C type), and wherever there is any doubt. This will lead to the now well known problems of combining libraries. Similarly, if linkage specifications are required anywhere, they will proliferate because of doubts about where they are needed.

It does not seem wise to refrain from checking linkage in a large number of cases and to introduce a rather arbitrary heuristic into the linking rules for C++ without being able to reduce the complexity of the language or to reduce the burden on the programmer somewhere.

8.6 Nothing

Naturally, while considering these alternative schemes the easy option of doing nothing was regularly re-considered. However, the original scheme still suffers from the problems described in section 4: insecure linkage, spurious `overload` declarations, overloading rules that complicate the life of library writers and library users, and unnecessary complexity for tools builders.

9. *Syntax Alternatives*

The scheme of giving all C++ functions type-safe linkage and providing a syntax for expressing that a given function is to have C linkage was thus chosen and tried. However, there were still several alternatives for expressing C linkage for this general scheme.

9.1 *Why extern?*

Instead of employing the existing keyword `extern` we might have introduced a new one such as `linkage` or `foreign`. The introduction of a new keyword always breaks some programs (though usually not in any serious way and for a well chosen new keyword not many programs) and `extern` already has the right meaning in C and C++. In almost all cases `extern` is redundant since external linkage is the default for global names and for locally declared functions. When used, `extern` simply emphasizes the fact that a name should have external linkage. The use of `extern` introduced here merely allows the programmer to tag an `extern` declaration with information of *how* that linkage is to be established.

9.2 *Linkage for Individual Functions*

One obvious alternative is to add the linkage specification to each individual function:

```
extern "C" double sqrt(double);
                  // sqrt(double) has C linkage
```

The advantage of this scheme is that the linkage is obvious from looking at an individual function declaration. The problem with this is that it does not serve the need to be able to give a set of C functions C linkage with one declaration and requires the declaration of every C function to be modified. In particular, it does not allow a C header (that is, an ANSI C header) to be used from a C++ program in such a way that all the functions declared in it get C linkage.

This notation for linkage specification of individual functions is not just an alternative to the linkage "block" adopted but also an obvious extension to the adopted syntax. After observing the use of linkage blocks for a while and listening to the comments from users this extension was adopted.

```
extern "C" double sqrt(double);
                // sqrt(double) has C linkage
```

is by definition equivalent to

```
extern "C" { double sqrt(double); }
                // sqrt(double) has C linkage
```

Naturally, a linkage specification applies to all members of a declaration list:

```
extern "C" double sin(double), cos(double);
                // sin and cos have C linkage
```

9.3 Linkage Pragmas

The original implementation of the linkage specifications used a `#pragma` syntax:

```
#pragma linkage C
double sqrt(double);  // sqrt(double) has C linkage
#pragma linkage
```

This was considered too ugly by many but did appear to have significant advantages. For example, it can be argued that linkage to "foreign languages" is not part of the language proper. Such linkage cannot be specified once and for all in a language manual since it involves the *implementations* of *two* languages on a given system. Such implementation specific concepts are exactly what pragmas were introduced into Ada and ANSI C to handle. The `#pragma` syntax was trivial to implement and easy to read. It was also ugly enough to discourage overuse and to encourage hiding of linkage specifications in header files.

There are problems with this view, though. For example, it is most often assumed that any `#pragma` can be ignored without affecting the meaning of a program. This would not be the case with linkage pragmas. Another problem is that for the moment many C implementations do not support a pragma mechanism and it is not certain that those that do can be relied upon to "do the right thing" for linkage pragmas used by a C generating C++ compiler.

Linkage to a particular foreign language does not belong in C++ because such linkage will in principle be local to a given system and non-portable. However, the fact that linkage to other languages occurs is a general concept that can and ought to be supported by a language intended to be used in multi-language environments. In practice, one can assume that at least C and Fortran will be available on most systems where C++ is used and that a large group of users will need to call functions written in these languages. Consequently, one would expect C++ implementations to support C and Fortran linkage.

The fact that C (like most other languages) does not provide a concept of linkage to program fragments written in other languages led to the absence of an explicit linkage mechanism in C++ and to the problems of link safety and overloading.

9.4 Special Linkage Blocks

Another approach would be to introduce a new keyword, say `linkage`, and use it to specify both the start and the end of a linkage block:

```
linkage("C");
double sqrt(double);   // sqrt(double) has C linkage
linkage("");
```

This avoids introducing yet another meaning for `{}`, allows setting and restoring of linkage to be two separate operations, allows all linkage directives to be found by simple pattern matching in a line oriented editor, and allows all linkage directives to be suppressed by a single macro

```
#define linkage(a)
```

The problem with this seems to be that it tempts people to think of as linkage as a compiler "mode" that can be switched on and off at random times and doesn't obey block structure. For example:

```
linkage("C");
double sqrt(double);   // sqrt(double) has C linkage
```

```
f() {
   extern g();          // g() has C linkage
   linkage("");
   extern h();          // h() has C++ linkage
   . . .
}
```

It also becomes hard to convince people that linkage specifications come in pairs and can be nested.

The same approach, with the same educational problems, can be tried without introducing a new keyword:

```
extern "C";
double sqrt(double);  // sqrt(double) has C linkage
extern "";
```

Note that whatever syntax was chosen, linkage specifications were intended to obey block structure to fit cleanly into the language. In particular, if linkage "blocks" and ordinary blocks were not obliged to nest, the job of writers of tools manipulating C++ source text, such as a C++ incremental compilation environment, would be needlessly complicated.

9.5 Special Header File Names

The impact of linkage issues on the C++ language would appear to be minimized if the distinction between functions with C linkage and functions with C++ linkage was done not in the language itself but in "the environment."[2] For example, it was suggested that one might decide that functions declared in header files with the `.h` suffix should be given C linkage and functions declared in header files with the `.H` suffix should be given C++ linkage. Naturally, the reduction in complexity would only be apparent because the same disctinction between C and C++ linkage would still have to be made by the programmer and handled by the compiler.

The fundamental problem with this scheme is that the meaning of a program becomes dependent on the way the program is

2. This section was added since the USENIX C++ proceedings version of this paper. The idea was among the ones originally considered, but when I wrote this paper I did not consider it necessary to explain why it was bad.

stored and cannot be determined simply from the source text of the program itself.

There are, and hopefully there will continue to be, many more environments in which C++ is used than there are dialects of the C++ language. Relying on header files names or other environment conventions will therefore affect the portability of C++ code, the portability of tools operating on C++ programs, and the effort needed to understand C++ programs. In particular, every tool working with C++ source text (including the compiler) would need access to the source file names and have to know about the (probably non-standardized) header file conventions.

Relying on header file names also makes it infeasible to define all the versions of an overloaded function in a single header file: If one of the set of overloaded functions should have C linkage it would have to be defined in a separate header from the rest.

10. Conclusions

The use of function name encodings involving type signatures provides a significant improvement in link safety compared to C and earlier C++ implementations. It enables the (eventual) abolition of the redundant keyword `overload` and allows libraries to be combined more freely than before. The use of linkage specifications enables relatively painless linkage to C and eventually to other language as well. The scheme described here appears to be better than any alternative we have been able to devise.

Acknowledgements

The new linkage and overloading scheme was essentially a joint effort of Andrew Koenig, Doug McIlroy, Jerry Schwarz, Jonathan Shopiro, and me. Brian Kernighan made many useful comments. The name encoding scheme is based on a proposal by Stan Lippman and Steve Dewhurst with input from Andrew Koenig and me. Steve Dewhurst, Margaret Ellis, Georges Gonthier, Bill Hopkins, Jim Howard, Mike Mowbray, Tim O'Konski, and Roger Scott also made valuable comments on earlier versions on this paper.

Appendix A: The Function Name Encoding Scheme

The (revised) C++ function name encoding scheme was originally designed primarily to allow the function and class names to be reliably extracted from encoded class member names. It was then modified for use for *all* C++ functions and to ensure that relatively short encodings (less than 31 characters) could be achieved reliably for systems with limitations on the length of identifiers seen by the linker. The description here is just intended to give an idea of the technique used, not as a guide for implementers.

The basic approach is to append a function's signature to the function name. The separator `__` is used so a decoder could be confused by a name that contained `__` except as an initial sequence, so don't use names such as `a__b__c` in a C++ program if you like your debugger and other tools to be able to decompose the generated names.

The encoding scheme is designed so that it it easy to determine

- if a name is an encoded name;
- what (unencoded) name the user wrote;
- what class (if any) the function is a member of;
- what are the types of the function arguments.

The basic types are encoded as

`void`	`v`
`char`	`c`
`short`	`s`
`int`	`i`
`long`	`l`
`float`	`f`
`double`	`d`
`long double`	`r`
`. . .`	`e`

A global function name is encoded by appending `__F` followed by the signature so that `f(int,char,double)` becomes `f__Ficd`. Since `f()` is equivalent to `f(void)` it becomes `f__Fv`.

Names of classes are encoded as the length of the name followed by the name itself to avoid terminators. For example, `x::f()` becomes `f__1xFv` and `rec::update(int)` becomes `update__3recFi`.

Type modifiers are encoded as

```
unsigned   U
const      C
volatile   V
signed     S
```

so `f(unsigned)` becomes `f__FUi`. If more than one modifier is used they will appear in alphabetical order so `f(const signed char)` becomes `f__FCSc`.

The standard modifiers are encoded as

```
pointer         *       P
reference       &       R
array           [10]    A10_
function        ()      F
ptr to member   S::*    M1S
```

So `f(char*)` becomes `f__FPc` and `printf(const char* ...)` becomes `printf__FPCce`.

Function return types are encoded for arguments of type *pointer to function*. The return type appears after the argument types preceded by a single underscore; for example, `f(int (*)(char*))` becomes `f__FPFPc_i`. The return type is not encoded except for pointer to argument types (see §5).

To shorten encodings repeated types in an argument list are not repeated in full; rather, a reference to the first occurrence of the type in the argument list is used. For example:

```
f(complex,complex);          // f__F7complexT1
   // the second argument is
   // of the same type as argument 1

f(record,record,record,record);    // f__F6recordN31
   // the 3 arguments 2, 3, and 4 are
   // of the same type as argument 1
```

A slightly different encoding is used on systems without case distinction in linker names. On systems where the linker imposes a restriction on the length of identifiers, the last two characters in the longest legal name are replaced with a hash code for the remaining characters. For example, if a 45 character name is generated on a system with a 31 character limit, the last 16 characters are replaced by a 2 character hash code yielding a 31 character name.

Naturally, the encoding of signatures into identifier of limited length cannot be perfect since information is destroyed. However, experience shows that even truncation at 31 characters for the old and less dense encoding was sufficient to generate distinct names in real programs. Furthermore, one can often rely on the linker to detect accidental name clashes caused by the hash coding. The chance of an undetected error is orders of magnitude less than the occurrence of known problems such as C programmers accidentally choosing identical names for different objects in such a way that the problem isn't detected by the compiler or the linker.

[*submitted Nov. 4, 1988; accepted Dec. 12, 1988*]