Using C and C++ with Fortran

Last update: **Sat Nov 17 16:46:27 2001**

Comments, and reports of errata or bugs, are welcome via e-mail to the author, *Nelson H. F. Beebe* *<beebe@math.utah.edu>.* In your report, please supply the full document URL, and the *title* and *Last update* time stamp recorded near the top of the document.

Table of contents

Background

Because of the large existing body of software, particularly numerical software, written in Fortran, it is desirable to call Fortran routines from other languages, notably, C and C++.

The ISO Fortran committee has tried to work with the ISO C and C++ committees to standardize the interlanguage calling interface, but the latter committees have been unwilling to do so, on the grounds that it would open the door to demands for interfaces to myriad other languages. Thus, there is currently no international protocol for communication between computer programming languages, and one is unlikely to be developed.

In practice, this means that interlanguage communication is only possible if supported by operating systems and compilers.

The architecture of DEC VAX (Open)VMS, for example, carefully defined a language-neutral calling sequence, allowing any pair of languages to communicate on that system.

On IBM PC DOS, calling conventions are up to each compiler, so in general, code compiled by separate compilers cannot be mixed, even if it was in the same source language.

Because the UNIX operating system and run-time libraries were historically written in C (though some now use C++), there is *de facto* standardization of calling sequences across all languages on a single UNIX platform to that used for C.

Despite this promise, there are many other issues that limit the degree of success of mixed-language programming, as the following sections document.

A compact summary of the Fortran language is available.

Alternatives to mixed-language programming

For maximal portability, a better approach is to stick with a single programming language. It may not be desirable, or feasible, to translate code in another language to the target language. LAPACK, for example, has about 660,000 lines of Fortran code: at a commercial code production rate of 1000 lines per month per programmer, that represents more than fifty person-years of programming time.

Fortunately, as long as Fortran source code is available, the excellent `f2c` translator provides a way to convert it to C or C++.

`f2c` does a good job of translating most of Fortran, but its weakness is the handling of I/O statements: they are translated to calls to a run-time library which is then required each time the program is linked. There is a commercial translator made by Cobalt Blue, Inc. which translates Fortran I/O statements to native C I/O statements, making the code easier to maintain, and more C-like.

Language translation is acceptable when the translated code is stable, as is the case for most major numerical libraries in Fortran.

However, it is definitely *not* desirable for code that is still under development: either you will end up maintaining the same program in two languages, with at least double the work, and unavoidable, if unintentional, differences as the code evolves, or you will have to work with repeatedly translated code that is decidedly less clear than code written by a competent programmer.

Language run-time differences

There are three important issues when languages are mixed:

1. interrupt and trap handling,
2. I/O, and
3. management of dynamically-allocated heap memory.

In general, when you mix languages, the code in one of them needs to be entirely free of interrupt handling, I/O, and dynamic memory allocation.

Fortunately, most well-designed numerical Fortran libraries, including EISPACK, ESSL, IMSL, LAPACK, LINPACK, MINPACK, NAG, PLOT79, and PORT, satisfy this requirement, easing their use from other languages.

Language data type differences

Fortran data types

Fortran (1954--) was the first practical high-level language. It was designed primarily for numerical programming, and several years before a good understanding was developed in computer science of how to define programming languages in terms of rigorous language grammars. Thus, Fortran has numerous syntactical quirks, and a limited selection of data types.

Originally, Fortran had only the data types `INTEGER`, `LOGICAL`, `REAL`, and `COMPLEX`, plus untyped word-aligned

character strings known as Hollerith data (e.g., 11HHELLO,WORLD). The latter is named after Herman Hollerith (1860--1929), the inventor of the punched card machines used for processing the 1890 U.S. Census data, and one of the founders of the company which eventually became IBM [actually, the Jacquard loom, invented by Joseph-Marie Jacquard (1752--1834), was driven by punched cards too, but the cards were large and wooden, instead of made of thin cardboard]. For an interesting biography, see Geoffrey D. Austrian, ``Herman Hollerith --- Forgotten Giant of Information Processing, Columbia University Press, 1982, ISBN 0-231-05146-8.

Shortly thereafter, Fortran got the `DOUBLE PRECISION` type, since that was widely implemented in hardware by the early 1960s.

When IBM System/360 was introduced in April 1964, it was one of the first byte-addressable computers, and IBM Fortran compilers were extended to recognize byte-length modifiers: `COMPLEX*8`, `COMPLEX*16`, `INTEGER*2`, `INTEGER*4`, `LOGICAL*1`, `LOGICAL*2`, `LOGICAL*4`, `REAL*4`, and `REAL*8`.

In 1967, when the first IBM System/360 model with quadruple-precision hardware was introduced, IBM Fortran was extended again to handle `REAL*16` and `COMPLEX*32` data types.

Later, other vendors added `BYTE` and `POINTER` types, and even oddities like Harris Computers' `INTEGER*3`, `INTEGER*6`, and `REAL*6`.

The byte-length modifiers reflect particular underlying machine architectures, notably, 32-bit words with 8-bit bytes. IBM's market share was large, and competitors soon added support in their Fortran compilers for those modifiers, even if it did not match their architectures, which might not even by byte-addressable, or have word sizes that are multiples of 8 bits. They simply mapped the byte-length data types to the closest size available on their machines.

However, although the ANSI/ISO Fortran 77 Standard added a `CHARACTER*n` data type to the language (a considerable improvement over typeless Hollerith constants), it did not recognize any of these extended types. ANSI/ISO Fortran 90 and Fortran 95 define `POINTER`, but continue to ignore byte-length modifiers, instead introducing a new syntax to accomplish much the same thing.

Variables of these types can be scalars, or arrays of up-to-three dimensions (up-to-seven in Fortran 66 and later), but there are no record structure data types, although Fortran `COMMON` blocks are often used to group related data to simulate record structures.

C/C++ data types

C (1969--) and C++ (1982/1986--) have a rich array of data types, from integer bit fields of arbitrary size (up to the number of bits in a memory word), to `enum` integer types, to integers of implementation-dependent-size (`char`, `short`, `int`, `long`, and, optionally, `long long`), optionally qualified by `signed` or `unsigned` modifiers, to floating-point (`float`, `double`, and optionally, `long double`), to pointers to functions and data.

Variables of these types can be scalars, or arrays of any number of dimensions. The C/C++ `struct` and `union` types, and the C++ `class` type, can be used to group data of different types for use as a single variable, which in turn can be either a scalar or an array.

Boolean and complex types are notably absent from C, although C adopts the convention for Boolean values that numeric zero means false, and numeric nonzero means true.

The more recent ISO language standards that define C++98 and C99 have `bool`, `true`, and `false`.

Support for complex numeric types has been introduced to C++98 via classes, operator overloading, and templates.

C99 has both `complex` and `double complex`, and optionally, `long double complex`, `float _imaginary`, `double _imaginary`, and `long double _imaginary`.

Fortran and C/C++ common data types

For mixed-language programming in C/C++ and Fortran, only the minimal intersection of their many data types can be relied on:

- `int` == `INTEGER`,
- `float` == `REAL`,
- `double` == `DOUBLE PRECISION`.

No other data types can be expected to be exchanged without serious compromise of portability.

Unfortunately, the Fortran LAPACK library uses `CHARACTER*n` data types in argument lists; we shall see below that this poses a huge problem for portable C and C++ code.

Array storage-order differences

Fortran stores arrays in row order, with the first subscript increasing most rapidly. C and C++ store arrays in column order with the last subscript increasing most rapidly. Despite arguments to the contrary, there is really no reason to favor either storage convention: rows and columns are equally important concepts.

Since two-dimensional arrays (matrices) are prevalent in most Fortran numerical libraries, the C/C++ user must be keenly aware of this difference. One solution is to wrap array-transposition code (an O(N^2) process) around calls to Fortran routines. Another is to program the C/C++ matrix code with reversed index order. A third, and often better, approach is to access the C/C++ matrix data through a C preprocessor macro, or C++ class access functions, to hide the index differences, and avoid unnecessary data movement. Thus, you could write

```
#define A(i,j) a[j][i]
```

and then use A(i,j) throughout your C code.

The differing array storage order has, however, a serious performance impact, because almost all modern machines have fast cache memory. Access to a memory location results in the hardware loading multiple consecutive memory locations (typically, 8 to 256 bytes) into cache, so that the next access to that data, which is likely to be the next array element, can be resolved from the cache, which can be many times faster. For a detailed discussion of this, see the document *High-Performance Matrix Multiplication.*

The best advice is thus to choose the array storage order that reflects the most common use of the data, and to order the array indexing loops accordingly.

Array indexing differences

Fortran array indexing starts at one, while C/C++ indexing starts at zero. An N-by-N matrix is declared in Fortran as `typename A(N,N)`, and indexed from 1 to N, while a corresponding C/C++ array is declared as `typename a[N][N]`, but indexed from 0 to N-1. This feature of C/C++ is a frequent source of off-by-one errors, and when you mix C/C++ code with Fortran, you are even more likely to be confused.

If most of the matrix data access is in the Fortran code, then it may be best to write the C/C++ code as if the arrays were Fortran arrays:

```
#define B(i,j) b[j-1][i-1]

void foo(void)
{
    ...
    for (j = 1; j <= N; ++j)
    {
        for (i = 1; i <= N; ++i)
        {
            ... B(i,j) ...
        }
    }
}
```

Even though C/C++ programmers frown on this practice, it may be the best way to avoid indexing errors in mixed-language programming. The access macro, `B`, is after all just a kind of data abstraction, so it is certainly in the spirit of C++.

Function return types

Fortran has `SUBROUTINE`, which does not return a value, and must be invoked by a `CALL` statement, and `FUNCTION`, which returns a scalar value, and must be invoked by function-like notation in an expression. While some

Fortran implementations permit a `FUNCTION` to be referenced by a `CALL` statement, it is decidedly nonportable to do so, and is a violation of all national and international standards for Fortran.

Fortran functions can return only scalar values, not arrays.

C and C++ have only functions, but those `functions' may or may not return a value. Good programming practice declares non-value-returning functions to be of type `void`, and value-returning ones to be of any non-`void` scalar type.

C/C++ functions can return only scalar values, not arrays, but they can return `struct` and (for C++) `class` values.

Unfortunately, returning composite objects that occupy more than a single register, or an adjacent register pair, is fraught with peril. Older C and C++ compilers did not support this at all, and newer ones may do it differently than Fortran compilers do: thus, you should not expect to use Fortran functions that return types such as `COMPLEX` or `COMPLEX*16`. Write a `SUBROUTINE` interface to your Fortran function instead, and then invoke it as a `void` function from C or C++.

File types

The Fortran view of files

Fortran files are of two fundamental types: `FORMATTED` (text) and `UNFORMATTED` (binary).

Binary files are compact, fast to read and write, and mostly incomprehensible to humans. They also avoid data conversion and accuracy loss, since data is stored in such files with exactly the same bit patterns as in memory.

Text file properties are the opposite of these, but text files have the advantage of being highly portable, and readable (and editable) by humans.

Data in Fortran files is divided into *records*, which are recognizable objects in the language and run-time libraries, and are logically the smallest objects in files that the language recognizes.

For text files, line boundaries mark records, and such files are generally trivial to process with any programming language or text software tool.

For Fortran binary files, special markers (usually 4 to 12 bytes in length) must be recorded in the files to mark the start and end of records, where a `record' corresponds to the data that is in the I/O list of a Fortran `READ` or `WRITE` statement.

For sequential files, such records may have widely-different lengths, and zero-length records are legal.

A Fortran binary `READ (unit)` statement with no I/O list can be used to skip forward over records, and a `BACKSPACE` statement may be used to skip backward over records.

The presence of record markers in Fortran binary files makes it impossible to use standard Fortran to write binary files that can be processed by other software, *even by other Fortran implementations on the same system.* Such files must be viewed as distinctly unportable, and many Fortran implementations do not even document precisely how records are identified in binary files, making it very difficult to process them with other languages.

Both text and binary file types may be accessed sequentially or randomly, although with random access, the records must be of uniform length, and in older Fortran implementations, and even on some current operating systems, the number of records must be known and declared at file-open time.

Standard Fortran does not define any way to read or write data at the byte level.

The C/C++ view of files

In C and C++, on the other hand, a file is viewed as an *unstructured stream of zero or more bytes*, and the C `fgetc()` and `fputc()` functions can access a byte at a time. Any additional required file structure must be supplied by the user's program itself.

This low-level view of files as unstructured byte streams makes it simple in C and C++ to write files that have any desired structure, since the user has complete control over exactly what bytes are read or written. Nothing is added or subtracted by the run-time library, with the sole exception of text files on systems that do not use a simple ASCII `LF` character to delimit lines. On such systems, the C `\n` (newline) character may be mapped to something else on output (`CR` on Apple Macintosh, `CR LF` on IBM PC DOS, Microsoft Windows, and several historic now-retired operating systems, or possibly a length field recorded at the start of the record (common practice in VAX VMS and old PDP-11 text files)). However, even on those systems, the file can be opened in binary mode, suppressing all such translations, and giving the user complete control over the data stream.

Mixed-language file processing

You should thus expect only to be able to share file data between Fortran and C programs with text files, and even there, you need to avoid the use of the Fortran `D` exponent letter: use the `Ew.d` Fortran `FORMAT` item, *not* `Dw.d`.

Fortran 77 permits the exponent field to contain leading blanks, and some Fortran implementations output numbers in the form 0.12E 1 instead of 0.12E+01. Fortran 90 and 95 outlawed leading blanks in the exponent field, and none of the many Fortran compilers on UNIX systems that I tested output blanks there.

The default field width allocated for the exponent field is only two digits, but in IEEE 754 floating-point arithmetic, used by virtually all computers on the market by 2000, a double-precision exponent can require three digits, and a quadruple-precision one, four digits. When three digits are required, the exponent letter is dropped, producing values like 0.12+306.

Embedded blanks and dropped exponent letters make such numbers illegal input for virtually all other programming languages. Fortunately, there is a solution: use the `Ew.dEd` Fortran `FORMAT` item, with the exponent width set to 3 for double-precision data, and 4 for quadruple-precision data. You should also use the `1P` scale factor, to avoid a leading zero: `1PE12.5E3` produces numbers like 1.2345E+123. Just remember that `1P` also affects subsequently-processed `Fw.d` items, so they need to be prefixed by `0P` to turn off the scaling.

The exponent-width `FORMAT` item modifier was introduced in Fortran 77, which was widely supported by the mid 1980s. However, older Fortran code will not have used it, so you should update all floating-point `FORMAT` items to ensure that the program's output will be readable from other programming languages.

In the other direction, you must avoid using any nondecimal numeric data, or length suffixes (`l`, `L`, `f`, `F`, ...), or unsigned numeric data types, in files expected to be read by Fortran programs.

The Fortran legacy of 80-column punched cards and 132-column band printers means that Fortran text files tend to have values that occupy fixed numbers of columns, possibly without intervening space (e.g., `123` can be read as 123, or 12 and 3, or 1 and 23, or 1 and 2 and 3, depending on the input `FORMAT` items).

More modern languages tend to view text files as sequences of objects of possibly varying width with some suitable separator (often space or comma or newline) between items. It is possible, though painful, in C or C++ to read and write text files with fixed-length fields, but you can avoid the problem entirely by ensuring that your Fortran output leaves at least one space between fields.

There is no C or C++ equivalent of Fortran list-directed I/O (`READ (unit,*) io-list`, `WRITE (unit,*) io-list`), or of Fortran `NAMELIST` I/O (`READ (unit,namelist-name)`, `WRITE (unit,namelist-name)`). Avoid those Fortran features if you expect to process such files with programs written in other programming languages.

Fortran data that is too wide for the available `FORMAT` width is output as asterisks. In C and C++, the field width is merely silently expanded to hold the data. Both of these practices pose problems if the output is to be read by another computer program. You should therefore choose output formats carefully, to ensure adequate field widths for all possible data values.

Language memory management

Most programs in C and C++ make heavy use of dynamically allocated memory, in C through the `malloc()` / `free()` family, and in C++ through the `new` operator (the C memory allocators are available in C++, but strongly deprecated). However, neither language garbage collects memory that is allocated, but no longer in use. Such *memory leaks* plague most long-running programs written in these languages.

As noted above , the Fortran `POINTER` data type is uncommon, and even where available, is a poor substitute for C's much more powerful dynamic memory management. The absence of dynamic memory support prior to the

Fortran 90 standard means that most older Fortran routines are burdened with additional array arguments that provide scratch space.

Most large Fortran programs start with a `main` program that reads in one or more variables defining problem sizes, then computes offsets into one or more large working arrays, cross sections of which are then passed to functions and subroutines as scratch space.

Clearly, C and C++ callers can do this too, or they can use their own dynamic memory allocation support to allocate the scratch arrays immediately before the call to the Fortran routine, and then free them immediately on return from the Fortran routine.

Language calling sequence differences

Argument addressing

In Fortran, all routine arguments, without exception, are passed *by address*. This means that within a `FUNCTION` or `SUBROUTINE`, assignment to an argument variable will change its value in the calling program. The exact behavior depends on whether arguments are handled by direct reference, or by copying into local variables on routine entry, and copying back on exit. Code like this

```
      CALL FOO(3)
      III = 3
      PRINT *,III
      ...
      SUBROUTINE FOO(L)
      L = 50
      END
```

will print 3 on some systems, and 50 on others, with surprising effects in the rest of the program from instances of 3 in the source code having been silently changed to 50.

Fortran offers no way to avoid such problems: arguments passed to other routines are always subject to modification in the called routines!

In C and C++, scalar objects are passed *by value*, and array objects *by address* of the first (index zero) element. In C, C++, and Fortran, arguments that are themselves routine names are passed *by address*. If we rewrite the above Fortran sample in C

```
   foo(3);
   iii = 3;
   printf("%d\n", iii);
...
void foo(int l)
{
    l = 50;
}
```

the program will always print 3, since function `foo(l)` has no access to the original location of its argument. The change to the argument is entirely local to the function `foo(l)`.

In summary then, a Fortran argument list of the form `(A,B,C)` must always be declared in C and C++ in the form `(typename *a, typename *b, typename *c)`, and used in the form `(&a, &b, &c)`.

Routine naming

In C, a global function name is represented by an identical external name used by the linker:

```
% cat foo.c
void foo() {}

% cc -c foo.c && nm foo.o | grep ' T '
00000000 T foo
```

[The `nm` utility dumps the external symbol table of an object file; `grep` selects the output lines that contain the string ' T ', marking a function definition.]

In C++, a global function name is *mangled* in a compiler-dependent way to include representations of the function value and argument types. Together with mandatory function prototypes, it is this name mangling

that ensures that, except for varyadic functions like `printf()`, it is impossible to call a C++ function with incorrect argument types or argument count, something that is very easy to do in C, with dire consequences at run time. For example, on GNU/Linux on Intel x86, the GNU C++ compiler produces this:

```
% cat foo.cc
void   foo   (void)             {}
void   dbl   (double d)         {}
double fdbl  (double d)         {return 0;}
float  fflt  (float f)          {return 0;}
int    intg  (int a, int b)     {return 0;}
bool   fbool (bool a)           {return (true); }

% g++ -c foo.cc && nm foo.o | grep ' T '
0000000c T dbl__Fd
00000048 T fbool__Fb
00000018 T fdbl__Fd
00000028 T fflt__Ff
00000000 T foo__Fv
00000038 T intg__Fii
```

The Portland Group C++ compiler produces different external symbols in some cases, making it difficult to mix object code from these two compilers:

```
% pgCC -c foo.cc && nm foo.o | grep ' T '
00000020 T dbl__Fd
00000060 T fbool__Fb
00000030 T fdbl__Fd
00000040 T fflt__Ff
00000010 T foo__Fv
00000050 T intg__FiT1
```

On Sun Solaris, the GNU C++ compiler produces the same symbol names as shown above, but the native C++ compiler produces very different ones, making it completely impossible to mix object code from these two compilers:

```
%% CC -c foo.cc && nm foo.o | grep ' T '
0000000000000038 T __1cDdbl6Fd_v_
0000000000000010 T __1cDfoo6F_v_
0000000000000070 T __1cEfdbl6Fd_d_
00000000000000c8 T __1cEfflt6Ff_f_
0000000000000108 T __1cEintg6Fii_i_
0000000000000140 T __1cFfbool6Fb_b_
```

In order for a C function to be called from C++, it must be declared with new syntax known to C++, but not to C. System header files therefore tend to have code like this extract from Sun Solaris `<stdio.h>`:

```
#ifdef  __cplusplus
extern "C" {
#endif
...
extern int      getchar(void);
...
#ifdef  __cplusplus
}
#endif
```

The preprocessor symbol `__cplusplus` is defined by C++ compilers, but not by C compilers. Thus, the C++ compiler sees the bracketing `extern "C" {...}`, but the C compiler does not. Both languages can then use the C library functions.

All C and C++ implementations produce external symbols as shown above. However, Fortran implementations on UNIX systems have exhibited at least three different external symbol conventions:

1. The world's first Fortran 77 compiler, produced in 1976 by Stu Feldman at Bell Laboratories in Murray Hill, NJ, where C and UNIX were both developed, appended an underscore on Fortran external names. Thus, Fortran `SUBROUTINE foo` is known as `foo_()` in C (and in C++ as well, provided that it is declared as `extern "C" void foo_(void)`). `f77` compilers in Berkeley UNIX, FreeBSD, NetBSD, and OpenBSD all follow this practice, as do f2c, GNU `g77`, and Fortran 77, 90, and 95 compilers from Compaq/DEC, the Portland Group, SGI, and Sun. The trailing underscore distinguishes Fortran code, which is important because of its call-by-address calling convention. This made it possible to provide Fortran equivalents of much of the C run-time library: programs in both languages could use `putchar(c)`, but different functions would actually be called in the two languages.
2. UNIX Fortran compilers from Hewlett-Packard and IBM, alas, chose to make Fortran external names the same: `SUBROUTINE foo` is `foo()` in C. This has two problems: C library functions are no longer available to

Fortran code under the same familiar names, and the naming is different from that used by most other UNIX systems.
3. The `f77` compilers from the now-defunct Ardent, Stellar, and their merged company, Stardent, converted Fortran external names to uppercase, so Fortran `SUBROUTINE foo` was known as `FOO()` in C.

These variations are a headache for interlanguage calling. The PLOT79 `<p79.h>` header file contains code like this:

```
/* CONS(a,b) should return ab, the concatenation
   of its arguments */
#if  __STDC__ || __APOGEE__
#define CONS(a,b) a##b
#else
#define CONS(a,b) a/**/b
#endif

#ifdef ardent
#define FORTRAN(lcname,ucname)  ucname
#endif

#ifdef _IBMR2
#define FORTRAN(lcname,ucname)  lcname
#endif

#ifdef __hp9000s800
#define FORTRAN(lcname,ucname)  lcname
#endif

#ifndef FORTRAN
#define FORTRAN(lcname,ucname)  CONS(lcname,_)
#endif
```

The CONS() function is needed to support old-style Kernighan & Ritchie C, as well as 1989 Standard C.

Function definitions like this then provide Fortran-callable primitive functions implemented in C:

```
int
FORTRAN(p79col,P79COL)()
{
...
}
```

`CHARACTER*n` arguments

We have left one of the nastiest interlanguage calling issues to last, but it is now time to reveal the horrid story of the Fortran 77 `CHARACTER*n` data type. The details are complex and lengthy, but their understanding is necessary if we are to be able to pass this data type between code written in Fortran and in other programming languages.

The Hollerith constants described above provided limited character string support in Fortran from 1954 until Fortran 77 appeared (late, on April 3, 1978).

The need to count characters, and the lack of a standard data type to represent routine arguments of Hollerith type, were substantial inconveniences. However, the following sample of legacy Fortran code is still usable, almost 50 years after Fortran was invented!

```
% cat hhello.f
      CALL S(12HHello, world, 12)
      END
      SUBROUTINE S(MSG,N)
      INTEGER K, N, M
      INTEGER MSG(1)
      M = (N + 3) / 4
      WRITE (6,'(20A4)') (MSG(K), K = 1,M)
      END

% f95 hhello.f && ./a.out
Hello, world
```

This is a remarkable testment to software portability and longevity, and is also a record unmatched by *any* other programming language:

A good programmable editor, like `emacs`, can eliminate the drudgery of character counting during Hollerith string input. Careful hiding of the machine-dependent constants needed to map character (byte) counts to

word counts, and the programming discipline to represent such data as Fortran `INTEGER` arrays, makes substantial character processing in Fortran feasible. The entire PLOT79 graphics system, consisting of more than 493,000 lines of code, uses Hollerith strings, with a convenient set of character primitives to make Hollerith string processing simple, and highly portable.

Nevertheless, by the mid 1960s, Fortran compilers were being extended to allow Hollerith strings to be represented as quoted strings, with the convention of doubling embedded quotes: thus, `7HO'Neill` could be written as `'O''Neill'` on many systems. Lowercase letters became available once the uppercase-only keypunch was retired, in the 1970s.

When Hollerith strings were passed as routine arguments, they still had to be received in the guise of one of the standard Fortran data types. The only portable type turned out to be `INTEGER`. A `BYTE` type would have been most convenient, but only a few Fortran compilers implemented it, and none of the Fortran compilers on word-addressable architectures had it. `LOGICAL` was unsuitable because numeric operations are not permitted on such data, and `LOGICAL*1` was not universally treated as a single-byte type. Floating-point types risked bit scrambling and concomitant data destruction, since some architectures renormalized floating-point data in load and store operations.

Until the C programming language became widely available in the mid to late 1980s, there was only one choice of programming language for writing portable software, and that language was Fortran. No contending language even came remotely close in popularity, or amount of code written. [Certainly a lot of Cobol [1960--] code exists, but Cobol was less portable, incredibly verbose, and completely devoid of support for floating-point arithmetic, so it was entirely ignored outside the business community.]

Thus, already by the mid 1960s, there was interest in supporting a quoted character string data type in Fortran.

Although Fortran was the first programming language to be standardized, in 1966, the American Standards Association (ASA) rules were that ASA Standards should encode common existing practice, not create new ones. It was not until the second standardization effort, with Fortran 77, that the now American National Standards Institute (ANSI) Fortran committee ventured to create new language features: the block `IF (expr) THEN ... ELSE IF (expr) THEN ... ELSE ... END IF` statement, and the `CHARACTER*n` data type.

Unfortunately, committee work usually involves many compromises, and the result was less desirable than might have been produced under a single visionary architect, such as happened with John McCarthy's LISP, Dennis Ritchie's C, Dennis Ritchie's and Ken Thompson's UNIX, and Niklaus Wirth's Pascal, Modula, Modula-2, and Oberon.

The block `IF` statement was a big improvement over Fortran's early arithmetic and logical `IF` statements, but was not accompanied by its obvious companions of `CASE`, `WHILE`, and `UNTIL` statements, or by internal procedures. All of these existed as clean prior art in the SFTRAN3 preprocessor developed at the Jet Propulsion Laboratory in Pasadena, CA, and used for writing interplanetary spacecraft flight control software in the 1970s, as well as for writing most of the PLOT79 system.

The Fortran 77 (and Fortran 90 and Fortran 95) `CHARACTER*n` data type is almost a total design botch, with more than a dozen deadly sins:

1. Only fixed-length strings are provided, even though character processing deals with words of varying length.
2. There is no empty string: that is like having integers without a zero! You cannot start with an empty string and gradually append things to it, unless you are willing to permit trailing blanks to be insignificant.
3. Assignment of a longer string to a shorter string silently, and undetectably, truncates the target string, even though the data loss may be fatal for subsequent processing.
4. Assignment of a shorter string to a longer string silently pads the target on the right with blanks.
5. Although a substring facility is provided, it foolishly cannot be applied to string constants!
6. There is no `FORMAT` item support for centered or left-adjusted output in character fields: they are always output flush-right according to `FORMAT` item `Aw`.
7. `CHARACTER*n` functions can only return strings of a length fixed at compile time.
8. An inadequate set of string intrinsic functions is provided:
    - `CHAR(n)` (to convert an ordinal character number to a character);
    - `ICHAR(n)` (to convert a character to its ordinal character number);
    - `INDEX()` (to search for the first occurrence of a substring in a string);
    - `LENGTH()` (to return the count of characters in the argument);
    - lexical comparison functions `LGE()`, `LGT()`, `LLE()`, and `LLT()` (to compare strings in the ASCII collating sequence).

There are no primitives for reverse searching, for letter-case conversion, for comparisons ignoring letter case in the native and ASCII character sets, for translation, for subset and regular-expression matching, or for tokenizing. While these can all be written portably in Fortran 77, in practice, their omission from the language definition means that they are rarely used, or that users keep reinventing them with different names and implementations.

9. There is no provision for representing unprintable characters by escape sequences, as is possible in C, C++, and many other modern programming languages. To represent the C string `"Hello\tWorld\n"`, the Fortran 77 programmer has to write `'Hello' // char(9) // 'world' // char(10)`, thereby embedding knowledge of one particular character set (ASCII in this example), and destroying readability and portability. [In practice, one should invent symbolic names for the nonprintable characters, and put those definitions as `PARAMETER` statements in an `INCLUDE` file. Fortran 77 does not have an `INCLUDE` statement, although all UNIX compilers, and most others, provide it.]
10. It is illegal to assign a source string to a target string if they overlap. For routine arguments, this requirement is untestable in Fortran, which lacks pointers and an `ADDRESSOF()` function. Thus, the trivial assignment `S = T` must be written as a loop with a temporary variable:

```
      CHARACTER*(*) S, T
      CHARACTER TMP
      INTEGER I
      ...
      DO 10 I = 1, LENGTH(T)
          TMP = T(I:I)
          S(I:I) = TMP
   10 CONTINUE
```

11. `CHARACTER*n` data cannot reside in a `COMMON` block with data of any other type.
12. `CHARACTER*n` data cannot overlay data of other types, either through the `EQUIVALENCE` statement, or via subroutine or function argument associations.
13. Lastly, Fortran 77 `CHARACTER*n` data are unlike any other Fortran data type: they carry their length around invisibly. For no other Fortran data type is it possible to inquire what its current size is, neither of scalars, nor of arrays: there is no analogue of the C/C++ `sizeof()` operator.

The inadequate Fortran 77 intrinsic function support means that serious string processing still requires a more powerful string library, such as the library developed for the Fortran 77 version of PLOT79.

The last two points in particular have serious ramifications for interlanguage calling. The requirement that a string know its own length means that `CHARACTER*n` variables have to be passed differently from any other data type, since both an address of the first character, and the length, must be passed.

Stu Feldman's original UNIX `f77` compiler handled this problem properly. That compiler supplies one additional argument, the string length, for each `CHARACTER*n` argument, but those extra arguments are all placed at the *end* of the argument list. Thus, these (extended) Fortran 66, Fortran 77, and C89 examples are compatible:

```
      CALL FOO(123, 'Hello', 3.14, 'World')
      ...

      SUBROUTINE FOO(A, B, C, D)
      INTEGER A
      INTEGER B(1)
      REAL C
      INTEGER D(1)
      ...
      END

      SUBROUTINE FOO(A, B, C, D)
      INTEGER A
      CHARACTER*(*) B
      REAL C
      CHARACTER D(*)
      ...
      END

void foo_(int *a, char *b, float *c, char *d,
         int *_len_b, int *_len_d)
{
}
```

Notice how clean Feldman's solution is. It works for both old Fortran with quoted strings, but without a `CHARACTER*n` data type, as well as for the new Fortran 77 data type. It even works for Hollerith strings, provided the programmer either passes the additional length arguments, or does not need the lengths in the called routine. It does not require the obnoxious restrictions on argument association and `COMMON` blocks imposed by the Fortran 77 Standard. This is about as perfect as possible, given the requirements of the Fortran 77

Standard, and continues to support the vast body of ancient, medieval, and modern Fortran code without requiring any changes to that code. The many UNIX compilers noted above that follow Feldman's trailing-underscore-on-Fortran-external-name mapping also use Feldman's design for `CHARACTER*n` arguments.

Sadly, several other conventions have been encountered in other Fortran compilers, notably, those from Hewlett-Packard and IBM. IBM's conventions differ, depending on the operating system:

- Pass the string length arguments as additional arguments *by address*, but immediately following the string argument, instead of at the end of the argument list.
- Pass the string length arguments as additional arguments *by value*, but immediately following the string argument, instead of at the end of the argument list.
- Pass the address of a record structure that includes at least the string address, and the string length.
- Pass the address of a record structure that includes at least the string address, and the string length, and pass the string length as an additional argument as well. For example, for the Hewlett-Packard HP 9000/800 in HP-UX 8.x, PLOT79's `<p79.h>` file has this code fragment:

```
#if __hp9000s500
/* The HP 9000/800 passes the lengths by value
immediately following  each CHARACTER argument. */
typedef int*    FINT;
typedef struct
{
    int         max_length;
    char        *curlenword;
    int         first_byte;
    int         length;
    char        *address;
} *FCHAR;
#define CH(x,n) ((x->address)[n])/* n-th character of x */
#define FLEN(x) (x->length)
     /* declared length of CHARACTER string */
#define FLENARG(x) CONS(x,_len)
     /* declared length of CHARACTER string */
#endif /* __hp9000s500 */
```

and a primitive in C that uses it looks like this:

```
void
#if __hp9000s800 && (HPUX == 8)
FORTRAN(p79chm,P79CHM)(filename, FLENARG(filename), mode)
#else
FORTRAN(p79chm,P79CHM)(filename,mode,FLENARG(filename))
#endif
FCHAR           filename;
FINT            mode;
int             FLENARG(filename);
{
    char        *p;

    p = p79ftc(&CH(filename,0),FLEN(filename));
    if (p)
    {
        (void)chmod(p,INTVAL(mode));
        (void)free(p);
    }
}
```

The test for version 8 of the HP-UX operating system is necessary, because in later versions, the string length was moved to the end of the visible argument list. Because the string might be passed as either a structure, or a pointer to its first character, references to it have to be wrapped in the `CH()` macro, which has a system-dependent definition.

As I promised at the start of this section, the story of the Fortran 77 `CHARACTER*n` data type is indeed horrid, complex, and long. Most C/C++ programmers should simply avoid using Fortran code that requires character string arguments, or should write portable wrappers in Fortran that are themselves devoid of such arguments, or should translate the Fortran code to C, either automatically, or if it is not large, by hand.

`COMMON` blocks

Fortran provides for shared global memory via `COMMON` blocks. These are named, or anonymous (so-called `blank' `COMMON`), areas of memory in which one or more variables are stored. The only things known to the linker about these memory areas are their names, and their lengths. There is no information whatever about their

contents, either their local variable names, or their data types.

In the past, computer memory was very expensive, and very limited, so it was common for old Fortran programs to economize on memory use by putting lots of variables in COMMON blocks, not for sharing, but simply for reuse of memory, and each routine normally declared a different list of variables for each such block.

Today, memory is much cheaper, and reasonably plentiful for many applications, so modern practice is to make each use of a particular COMMON block identical, typically by placing its definition in a header file that is incorporated in the source code at compile time by a nonstandard (but widely implemented) INCLUDE statement.

This modern use of COMMON provides for maintenance of state across calls, and for information hiding. A few libraries, notably, PORT and PLOT79, make heavy use of COMMON blocks for such reasons: none of these blocks are ever expected to be examined or modified by end-user code.

Variables in COMMON blocks can be initialized in either of two ways: by direct assignment at runtime, usually early on, or by DATA statements in a BLOCK DATA routine. Standard Fortran has these restrictions, although there are compilers that silently relax some or all of them:

- A named COMMON block may be initialized with DATA statements in only a *single* BLOCK DATA routine.
- It is *illegal* to initialize blank COMMON variables with DATA statements.
- It is *illegal* to initialize COMMON variables with DATA statements inside a SUBROUTINE or FUNCTION.

Although a BLOCK DATA routine can be named, there is little significance to the name, since such a routine cannot be called (it has no executable code, not even a *return* instruction). This is an unfortunate design flaw in Fortran, since it means that such routines can never be resolved from a load library; they must always be explicitly loaded at link time. For that reason, libraries that employ COMMON blocks cannot use BLOCK DATA routines; they must use run-time initialization by assignment instead. Unfortunately, that has the defect that unless the initializing routine is explicitly called, execution may proceed with random garbage in the COMMON variables.

In most other programming languages, including C and C++, there is no equivalent of Fortran COMMON blocks, although named global data may be supported.

Let us see how Fortran makes COMMON block names and BLOCK DATA routine names available to the linker (this experiment was run on a Sun Solaris 2.7 system):

```
% cat common.f
      REAL A,B,C
      COMMON / / A,B,C

      INTEGER U,V,W
      COMMON /UVWCB/ U,V,W

      DOUBLE PRECISION X,Y,Z
      COMMON /XYZCB/ X,Y,Z

      A = 1.0
      B = 2.0
      C = 3.0

      WRITE (6,'(3(F6.3, 5X))') A,B,C
      WRITE (6,'(3(I6,   5X))') U,V,W
      WRITE (6,'(1P3E11.3E3)')  X,Y,Z

      END

****************************************

      BLOCK DATA BDXYZ

      DOUBLE PRECISION X,Y,Z
      COMMON /XYZCB/ X,Y,Z

      DATA X,Y,Z / 1.1111111111111111D+100,
     X             2.2222222222222222D+200,
     X             3.3333333333333333D+300 /

      END

****************************************

      BLOCK DATA

      INTEGER U,V,W
      COMMON /UVWCB/ U,V,W
```

```
      DATA U,V,W / 123456, 234567, 345678 /

      END
```

```
% f77 -c common.f &&
    f77 common.o &&
      ./a.out &&
        nm common.o | grep ' [CDT] '
common.f:
 MAIN:
 BLOCK DATA bdxyz:
 BLOCK DATA:
 1.000      2.000      3.000
123456     234567     345678
 1.111E+100 2.222E+200 3.333E+300
0000000000000010 T MAIN_
00000000000001e0 T _BLKDT__
000000000000000c C _BLNK__
00000000000001b8 T bdxyz_
0000000000000208 T main
0000000000000000 D uvwcb_
0000000000000010 D xyzcb_
```

[The `nm` utility dumps the external symbol table of an object file, and `grep` selects the output lines that contain the strings ' C ', ' D ', or ' T ', marking a `COMMON` block, a `BLOCK DATA` routine, or a `SUBROUTINE` or `FUNCTION` definition.]

Evidently, this compiler produces external names for `COMMON` blocks the same way that it does for `SUBROUTINE` and `FUNCTION` units. However, it assigns the name `_BLKDT__` to the unnamed `BLOCK DATA` routine.

Experiments with other compilers showed that the GNU Fortran compiler, `g77`, assigns the name `_BLOCK_DATA__` to the unnamed routine. Other compilers tested used names like `.BLOCKDATA.`, `blk@data_`, `data$common_`, `common$DATA`, or `.blockdata._`. Two compilers assigned no name at all.

All compilers tested named the named `COMMON` block using the same conventions as for other external names. However, the unnamed blank `COMMON` was variously called `_BLNK__` on most systems, and `#BLNK_COM` on IBM AIX.

Most computer architectures require data to be aligned at memory addresses that are multiples of their length. On many, failure to do so causes a fatal run-time errors. On some, a run-time fixup is made, slowing execution. On a few, there is no such alignment requirement, or performance impact. Since a Fortran `COMMON` block specifies the memory layout of the variables in the block, it is important to order the variables by order of decreasing size of their data types:

1. `COMPLEX*32`;
2. `REAL*16` and `COMPLEX*16`;
3. `COMPLEX*8`, `REAL*8`, and `DOUBLE PRECISION`;
4. `REAL`, `INTEGER`, and `LOGICAL`;
5. `INTEGER*2` and `LOGICAL*2`;
6. `BYTE`, `INTEGER*1`, and `LOGICAL*1`.

The external name variation on `BLOCK DATA` and blank `COMMON` strongly discourages attempts to reference them from C and C++ programs. For named `COMMON`, however, you can create `struct` definitions like these (using the `FORTRAN()` macro defined above to hide the external name mapping) to reference the `COMMON` blocks defined in the sample file above:

```
#include <stdio.h>
#include <stdlib.h>

#include "common.h"    /* for FORTRAN() macro */

#define C_xyzcb FORTRAN(xyzcb,XYZCB)
#define C_uvwcb FORTRAN(uvwcb,UVWCB)

extern struct {
    int u;
    int v;
    int w;
} C_uvwcb;

extern struct {
    double x;
    double y;
    double z;
} C_xyzcb;
```

```
int main()
{
    (void)printf("%6d     %6d     %6d\n",   C_uvwcb.u, C_uvwcb.v, C_uvwcb.w);
    (void)printf("%11.3le%11.3le%11.3le\n", C_xyzcb.x, C_xyzcb.y, C_xyzcb.z);
    return (EXIT_SUCCESS);
}
```

Run-time array dimensions

Fortran has a convenient feature that allows arrays to be passed to routines and used there without having to know at compile time what their dimensions are:

```
      INTEGER NA, NB, NC
      PARAMETER (NA = 3, NB = 4, NC = 17)
      REAL X(NA,NB,NC)
      ...
      CALL SOLVE(X, NA, NB, NC, MA, MB, MC)
      ...
      SUBROUTINE SOLVE(Y, MAXYA, MAXYB, MAXYC, KYA, KYB, KYC)
      INTEGER KYA, KYB, KYC, MAXYA, MAXYB, MAXYC,
      REAL Y(MAXYA, MAXYB, MAXYC)
      INTEGER I, J, K

      ...
      DO 30 K = 1, KYC
          DO 20 J = 1, KYB
              DO 10 I = 1, KYA
              ... Y(I,J,K) ...
   10         CONTINUE
   20     CONTINUE
   30 CONTINUE
      END
```

Notice that two sets of dimensions are typically passed: the current working dimensions, and the maximum dimensions. This language feature is used very heavily in numeric libraries in Fortran.

Because the last dimension is not required for array address computations (`Y(I,J,K)` is found at `addressof(Y(1,1,1)) + (I - 1) + (J - 1)*MAXYA + (K-1)*MAXYA*MAXYB`), old Fortran code often omitted a dimension argument for that dimension, and used the constant 1 in its place in the declaration (`Y(MAXYA, MAXYB, 1)` in the above example).

One problem with this old practice is that if the compiler wants to do array bounds checking, it does not know whether that dimension is really 1, or some larger, but unknown, value. For that reason, Fortran 77 introduced the possibility of replacing the unit dimension by an asterisk, meaning unspecified. Subscript checking can then be done for all but the last dimension.

The older EISPACK and LINPACK libraries use 1 for trailing dimensions, while the newer LAPACK library uses asterisk.

In C and C++, arrays are treated as equivalent to a pointer to the first element, and as in Fortran, they carry no hidden information about their true dimensions. However, unlike Fortran, C and C++ historically did not provide for run-time dimensioning. This is a major inconvenience for numerical work, but its lack was not felt in the operating system and software tool applications that C was originally designed for.

Once again, the C preprocessor can come to the rescue:

```
#define Y(i,j,k) ((i)*maxyb*maxyc + (j)*maxyc + (k))
            /* assuming C/C++ storage order */
void solve(float y[][][],
           int maxya, int maxyb, int maxyc,
           int kya, int kyb, int kyc)
{
    ... Y(i,j,k) ...
}
```

It is very easy to get the addressing wrong, particularly as the number of dimensions increases, so it is highly desirable to encapsulate such definitions in shared header files that can be debugged once and for all.

The GNU C and C++ compilers, `gcc` and `g++`, added support for run-time array dimensions sometime in the 1990s. Based on that experience, the 1999 ISO C Standard finally introduced support similar to that which Fortran has enjoyed since 1954, calling it the new ``variable-length array'' feature.

At the time of writing in late 2001, *none* of the vendor-provided C and C++ compilers available to this author on 16 different UNIX platforms have this feature, unless they are derived from the GNU compilers. However, once compilers for this language level become widely available, it will be possible to rewrite the above example as:

```
void solve2(int maxya, int maxyb, int maxyc,
            int kya, int kyb, int kyc,
            float y[maxya][maxyb][maxyc])
{
    ... y[i][j][k] ...
}
```

Notice the different order of arguments: the C99 variable-length array feature requires that all dimensions be declared in the argument list *before* they are used as dimensions. This restriction has never been part of Fortran, and since the common Fortran practice is to follow array arguments with their dimensions, it remains a barrier for multilanguage code documentation and translation.

The new feature is *not* part of the 1998 ISO C++ Standard, so once it is used in C programs, they will no longer be compilable by C++ compilers.

Recursion

Most programming languages defined since the early 1970s support recursion: the ability of a routine to call itself, either directly or indirectly.

Efficient implementation of recursion generally requires a memory data structure known as a *stack*, and most computer architectures designed since the 1970s provide fast hardware support for stacks. The routine calling convention on the machines of the 1950s for which Fortran was originally developed stored the return address in the called routine, completely preventing recursion.

Despite the existence of hardware support for recursion, its provision by almost all programming languages designed after Fortran and Cobol, and its widespread use in theoretical computer science, and algorithm and data structure design, the ASA, ANSI, and ISO Fortran committees were reluctant to admit to its existence. It was not until the ISO Fortran 90 Standard that Fortran officially got support for recursion, and then only in a rather crippled form, requiring explicit declaration of all routines that will make recursive calls.

None of the major numerical libraries that we have discussed until now use the Fortran 90 `RECURSIVE` attribute on routine declarations, so C and C++ programmers must be very careful to avoid using these libraries in such a way that recursion through a Fortran routine could occur.

In most cases, there is no possibility of such recursion, because the Fortran library routine is called, does its work, and returns. However, some libraries have routines that receive a user-defined function that they in turn call. Numerical quadrature routines commonly do this. If that function ever calls the same Fortran library routine, failure is virtually certain. This can happen, for example, in a multidimensional quadrature.
