Number Crunching in C

Jonathan Thornburg
Dept. of Geophysics and Astronomy
The University of British Columbia
Vancouver B.C. V6T 1W5 Canada

Introduction

Despite its many competitors, FORTRAN is still the dominant language used for numerical software ("number crunching"). This note will outline some of the lessons learned by the author in using C for this purpose. In particular, five potential problem areas (two of them minor) will be pointed out, and two changes to C which could help will be discussed.

Advantages

Most of these are obvious—nice data structures, a rich set of control flow constructs, the preprocessor, nice I/O facilities, ...

Potential Problems

The first and probably the most important requirement of a language to be used for number crunching, is that FORTRAN subprograms be callable from it. It's also highly desirable for the converse to hold. The reason for these requirements is simply to be able to make use of existing numerical software, almost all of which is written in FORTRAN. (For example, the IMSL, NAG, LINPACK, EISPACK, ITPACK, SPARSPAK, ... libraries.) In addition, much utility software (notably graphics packages) is "FORTRAN compatible". Most C implementations meet this FORTRAN compatibility requirement quite well, although any bugs due to differences in argument passing conventions[1] can be *very* hard to find.

The second (this time actual, not potential) problem area is C's prejudice against single precision floating point numbers. All single precision floating point function parameters are converted to double precision, and all floating point expressions are evaluated in double precision. Thus, if x, y, and z are all single precision variables, the expression

```
x = y + z;
```

will cause the values y and z to be converted to double precision, the resulting temporaries to be added in double precision, and the result to be converted back to single precision before being stored in x. This not only increases object code size, but negates any run-time speed benefits of single precision arithmetic.

The most obvious possible solution to this problem would be to modify C's "usual arithmetic conversions" (Kernighan and Ritchie (1978), p. 184) to allow arithmetic operations (including the implied assignment of function argument passing) both of whose operands are single precision floating point expressions, to be done in single precision themselves. This would yield a type system (for arithmetic) similar to FORTRAN's. However, this would result in the expression

```
dx = (3.0/7.0) * dy;
```
(where dx and dy are double precision variables)

using a *single precision* value of $\frac{3}{7}$ (most FORTRANs also do this).

---

[1] For example, the DEC VMS FORTRAN compiler assumes that a called procedure will not modify its argument list, while the VMS C compiler generates code that does exactly this whenever the C function modifies its arguments. This leads to effects similar to the classic FORTRAN bug of passing a constant to a subroutine which modifies the corresponding argument.

A better solution would be to use Kahan's "widest need" evaluation strategy (Corbett (1982)). This evaluates each expression using the highest precision needed to evaluate *any* part of that expression. This would result in a double precision value of $\frac{3}{7}$ being used in the above example. The analogous change for integer values would probably *not* be a good one, since it would require partial-word arithmetic to manipulate character variables. Corbett (1982) reports a trial implementation of this evaluation strategy with no major problems.

The third problem area concerns passing multidimentional arrays to functions. Unless the array bounds[2] are compile-time constants, C is unwilling to do the subscript computation. The preprocessor can be used to cover up this problem, but a nicer (though harder to implement) solution would be to change the language definition so that array declarations which do not allocate storage (*i.e.* those which merely describe variables which already exist, such as function arguments) could have run-time expression bounds.

This would require that a choice be made of when the subscript polynomial is to be evaluated (in case the bounds expression changes value during the function's execution). Re-evaluation each time the array is subscripted is most consistent with C's `for`-loop construct, and also allows the bounds expression to reference a quantity computed by the function (*i.e.* an array might be of size $n^2 \times n^2$, with n passed as an argument to the function). There is the potential for some inefficiency in such a case (assuming n to be constant within the function). However, this can be handled by the usual mechanisms for optimizing common subexpressions, either by the programmer (introduction of an explicit temporary variable) or the compiler.

Note that this change would still leave a function's stack frame size calculable at compile time, so there should be no extra storage management or variable addressing overhead.

The fourth potential problem area is the lack of a "complex" data type in C. This can be simulated by declaring a structure containing real and imaginary parts and defining suitable arithmetic functions (the recent addition of structure assignments and function values to C helps here), but the syntax isn't as nice as for the built-in data types. The severity of this problem depends on whether or not the application uses complex numbers—most don't, so this doesn't justify a language change.

The final potential problem is similar in scope to the previous one—C lacks an infix exponentiation operator. This is only a minor inconvenience.

Conclusions

C has proven to be an excellent language for writing numerical software. Two language changes have been suggested which would make it even better. Neither of these should add any overhead to functions which don't use them, and both have already been implemented in other languages and compilers.

References

Corbett, R. P. (1982): "Enhanced Arithmetic for Fortran", SIGPLAN Notices **17**(12), p. 41–48.

Kernighan, B. W., and Ritchie, D. M. (1978): "The C Programming Language", Prentice–Hall, Englewood Cliffs, New Jersey.

---

[2] Strictly speaking, all but the first array bound, which can be omitted, since it's not used in the subscript computation.