C Versus Fortran-77 for Scientific Programming

TOM MACDONALD

Cray Research, Inc., Eagan, MN 55121

ABSTRACT

The predominant programming language for numeric and scientific applications is Fortran-77 and supercomputers are primarily used to run large-scale numeric and scientific applications. Standard C* is not widely used for numerical and scientific programming, yet Standard C provides many desirable linguistic features not present in Fortran-77. Furthermore, the existence of a standard library and preprocessor eliminates the worst portability problems. A comparison of Standard C and Fortran-77 shows several key deficiencies in C that reduce its ability to adequately solve some numerical problems. Some of these problems have already been addressed by the C standard but others remain. Standard C with a few extensions and modifications could be suitable for all numerical applications and could become more popular in supercomputing environments. © 1993 John Wiley & Sons, Inc.

1 INTRODUCTION

Standard C and Fortran-77 are the two most prevalent languages used on supercomputers. A comparison of Standard C and Fortran-77 shows that C contains a wider variety of data types, eloquent sequence control, a standard preprocessor, wider variety of memory allocation options, communication with the program's environment, and additional operators not present in Fortran-77. Many desirable linguistic features provided by Standard C are not present in Fortran-77. Fortran's major strength is its optimization potential. High performance capability is as important as any language feature in the numerical and scientific arena, and this is especially true for supercomputing environments. Additional advantages of Fortran-77 are support of a complex data type, adjustable arrays, assumed size arrays, and intrinsic functions for the standard math functions. The following is an analysis of the strengths and weaknesses of Standard C and Fortran-77 from a numerical programming perspective. The intent is to provide useful information to someone trying to decide which programming language to use for numerical and scientific programming. There is a Fortran-90 standard that is not considered in this article because, unlike Fortran-77, it does not yet have a successful history against which to compare. There is also a discussion of enhancements being considered by committee X3J11.1 the Numerical C Extensions Group (NCEG). Their goal is to produce a quality technical report providing

Received February 1992

Revised November 1992

* The term *Standard C* refers to the standards ANSI X3.159-1989, American National Standard for Information Systems–Programming Language–C, approved December 14, 1989 [1], and the technically identical ISO/IEC 9899:1990, Programming Languages–C, approved approximately 1 year later.

© 1993 by John Wiley & Sons, Inc.
Scientific Programming, Vol. 1, pp. 99–114 (1992)
CCC 1058-9244/93/020099-16

implementers with a formal definition of several new features. These features will enhance C's support for numerical and scientific programming, and will be upward compatible with the C standard.

2 THE ADVANTAGES OF C

2.1 Data Types

Standard C defines a richer set of data types than found in the Fortran-77 standard. Scalar types not present in Fortran-77 include unsigned integers, pointers, and enumerated constants. The following is a list of integer types in Standard C (though there are other ways to declare these types):

`signed char`	`unsigned char`
`signed short`	`unsigned short`
`signed int`	`unsigned int`
`signed long`	`unsigned long`
enumeration type	

An enumeration type provides a mechanism for specifying named integer constants. The following example shows an enumeration type with four members:

```
enum color { red, green, blue, none = -1 };
```

The enumeration type also provides an automatic numbering facility for each uninitialized named member. The value of member `red` is 0, member `green` is 1, and member `blue` is 2. Furthermore, the automatic numbering can be restarted by assigning a specific value to a member. Thus, member `none` starts the sequence over with the value −1. Variables can be declared to have an enumeration type and assigned the values of the enumeration members.

```
enum color pixel = none;
```

Fortran-77 provides only a single signed integer type. Named constants can be specified in Fortran-77 with the PARAMETER statement, but the useful automatic numbering feature is not accommodated.

```
PARAMETER (NAME = 7)
```

No pointer type is found in Fortran-77, which limits the expressiveness of the language but also turns out to be an aid to optimizations. The semantics of Standard C pointers introduces more aliasing, that is hidden from the compiler. Many optimizations depend on the compiler's ability to identify aliasing. Aliasing issues are explored in detail in section "4.4 C Aliasing." Standard C pointers provide easy access to dynamically allocated memory and are discussed in more detail in section "2.6 Memory Allocation." Standard C pointers turn out to be a blessing and a curse. They provide expressive power but at considerable cost in lost efficiency.

Standard C provides multiple aggregate types also. There are structures, unions, and arrays. A structure describes sequentially allocated members with various types.

Example 1

```
struct s_tag {
   double member1;        /* floating-point type */
   enum color member2;    /* integer type        */
   int *member3;          /* pointer type        */
};
```

A union describes an overlapping set of members. Unions appear to be very similar to structures, but permit the same memory locations to be viewed with the different types of its members. Fortran-77 defines only one aggregate type, the array. The Fortran EQUIVALENCE statement is less general than a C union, but does allow overlapping to be defined.

Standard C's typing mechanism can be used to derive other more complicated types such as arrays of structures, structures containing array members, pointers to structures, pointers to arrays, and pointers to functions. The following example uses a pointer to a function.

Example 2

```
/* pointer to function with one double
   parameter that returns a dobule    */

double (*ptr_to_func)(double) = sqrt;
double x;

x = ptr_to_func(2.3);  /* indirect call through a pointer */
```

Standard C's method of constructing derived types provides a powerful capability for data representations that is not permitted with Fortran-77.

The physical layout of memory can be specified to a fine granularity with Standard C's structures. Integral members can be declared to occupy an arbitrary number of bits within a memory word (called a bitfield) allowing table layouts to be mapped precisely and portably. This control makes it much easier to conserve memory in a standard portable manner.

2.2 Type Synonyms

Standard C supports the declaration of a synonym for all declarable types with the`typedef`keyword. This allows a complicated type to be represented with a synonym.

Example 3

```
typedef struct {
   double real, i, j, k;
} QUATERNIONS[100]; '

QUATERNIONS a, *b;
```

Example 3 defines an identifier QUATERNIONS that is a synonym for the type, array of 100 structures containing four members, each with type double. The variable a has the type *QUATERNIONS*, while variable b has the type pointer to *QUATERNIONS*. This ability to encapsulate a complicated concept with a single meaningful name is very powerful, and an aid to both portability and maintainability. A single `typedef` declaration can be changed (from say, an array of 100 elements to an array of 1,000 elements) and every variable declared with that `typedef` name is automatically changed. This expressive typing mechanism is a definite advantage that Standard C has over Fortran-77.

2.3 Flow of Control

Standard C contains eloquent sequence control statements along with nested scopes that facilitate the use of structured programming techniques. Standard C contains `if`, `if-else`, `for`, `while`, `do-while`, and `switch` statements. This cadre of sequence control statements is complemented with the following statements that nicely alter the flow of control through a program:

1. `continue`—branch to the top of the inner loop
2. `break`—exit a loop or switch statement
3. `return`—exit from a function.

Finally, Standard C supports recursive function calls.

Fortran-77's primary flow of control statements are limited to IF, IF-ELSE, GOTO, DO, and RETURN. The proliferation of statement labels in Fortran programs leads to highly unstructured algorithms. Control flow constructs such as the *Arithmetic IF Statement* and *Assigned GOTO* are inherently unstructured, and often produce difficult to understand algorithms.

One advantage that Fortran-77 does have is a DO statement that can be statically analyzed at compile time. This allows the computation of the loop's trip count (the number of times the loop iterates) to be done exactly once in a straightforward way.

The for loop in Standard C does not have the same guarantees because:

1. Modification of the loop control variable is allowed inside the loop body
2. Loop limit expressions are not necessarily static
3. Trip count can be data dependent

The following show examples of loops whose trip count cannot be computed prior to executing the loop.

Example 4

```
for (i=0; i<n ; i++)) {
   a[i] = b[i] + c[i];
   i =ix[i];    /* modify loop control variable */
   n =iy[i];    /* modify loop limit expression */
   *p = iz[i]; /* 'p' might alias 'i' or 'n'   */
}
/* trip count is data dependent */
for (p = head; p != NULL; p = p->next) {
   p->data++;
}
```

Standard C semantics requires analysis of the entire containing function to compute a loop's trip count. Although trying to determine if the trip count can be computed prior to entering the loop requires considerable analysis, it is not an insurmountable problem. It is not too onerous to adopt a coding style that allows the compiler to precompute the trip count. In the final analysis, the rich flow of control mechanisms available with Standard C is another one of its advantages.

2.4 C Preprocessor

Standard C has a preprocessor that permits source code inclusion, conditional source exclusion, and macro definitions. Source inclusion is accomplished through the preprocessing directive #include, which causes the source line containing the directive to be replaced with the contents of the specified file. Source inclusion permits a standard set of declarations to be included in all compilation units (i.e., source files) allowing an interface to a set of library functions to be implementation dependent yet hidden from the programmer.

```
#include "common.h"  /* common library interface */
```

Conditional source exclusion allows source code to be tailored to specific environments. The #if, #elif, #else, and #endif directives provide this conditional source exclusion capability.

The following example shows how to guarantee that an integral type has more than 16 bits of precision.

Example 5

```
#include <limits.h> see section 2.5 for limits.h discussion

#if INT_MAX == 32767  /* 16 bit int  */
   typedef long data_type;
```

```
#else                   /* > 16 bits  */
   typedef int data_type;
#endif
```

Macro definitions allow a complicated operation or set of statements to be defined with a function-like description that enhances portability and maintainability. The following macro definition:

```
#define ROOT(a, b, c) ( (b + sqrt(b*b - 4*a*c)) / 2*a )
```

is an example of how a cryptic expression can be given a meaningful name through a macro definition. Similarly, access to critical data structures can be encapsulated with macro definitions that improve the flexibility and comprehensibility of the program. The following example:

```
#define denom(X, I)  (X->table[I].denom)
```

allows a complicated data reference to be given a meaningful name that avoids using many long cryptic expressions. In all fairness to Fortran-77, it does permit the declaration of statement functions that provide a subset of the C macro definition capability. The presence of C's built-in preprocessor is a tremendous asset to writing both portable and maintainable programs.

The C preprocessor is often viewed as an advantage by Fortran programmers to the extent that C preprocessing directives and macros are sometimes inserted into Fortran programs. However, the C preprocessor is not a part of the Fortran-77 standard. The C preprocessor can easily cause problems for Fortran programs by expanding macros such that a line is longer than 72 characters, not recognizing Fortran comments, eliminating something that looks like a C comment, and affecting the compiler's ability to print the correct line number when issuing a diagnostic. The C preprocessor is well defined for C but not necessarily for other languages.

2.5 Standard Library

Standard C defines a standard set of library functions that must be supported by all conforming implementations. The presence of this extensive library is an aid to developing portable programs. The library consists of a set of functions, typedef names, and macro definitions that are grouped according to functionality and declared in standard header files. The standard header files are made available by using the **#include** preprocessing directive. The standard headers are:

`assert.h`	generating program diagnostics
`ctype.h`	testing and mapping characters
`errno.h`	recording error conditions
`float.h`	floating-point characteristics
`limits.h`	sizes of integral types
`locale.h`	supporting international locales
`math.h`	transcendental mathematical functions
`setjmp.h`	nonlocal jumps
`signal.h`	exception handling
`stdarg.h`	processing functions with a variable number of arguments
`stddef.h`	common definitions
`stdio.h`	input and output
`stdlib.h`	general utilities
`string.h`	manipulating character arrays
`time.h`	manipulating time

This rich set of functions provides an excellent interface to the underlying C implementation and is another advantage of Standard C. The implementation is not required to actually provide external source files for the standard headers. To improve compile time they can be special binary encoded files or even built into the internals of the compiler. Fortran-77 provides a limited set of standard intrinsic functions that are primarily mathematical functions. Fortran-77 does provide two very useful intrinsic functions that Standard C does not: MAX and MIN.

2.6 Memory Allocation

Some of the most useful library functions are the dynamic memory support functions declared in the `stdlib.h` header. The standard functions `malloc`, `calloc`, `realloc`, and `free` define a portable way to dynamically manage space associated with a system heap. Heap space can be allocated with `malloc` and `calloc`, the size of allocated space can be increased or decreased with `realloc`, and this space can be given back to the heap manager with `free`. Standard C pointers are used to reference dynamically allocated memory. In addition to dynamically allocated heap space, Standard C permits variables to be declared with global static, local static, or local stack space. Global variables can be either declared to be externally visible to other compilation units, or strictly local to the containing compilation unit.

Fortran-77 only supports global COMMON blocks and local variables. The Fortran-77 standard does not specify if local variables are static or stack allocations. However, the SAVE statement allows variables to preserve their value across calls to the same function. The absence of a portable dynamic storage management system is a serious deficiency in Fortran-77. Often, vendors who provide scientific and numerical library packages have to provide awkward interfaces that require the programmer to specify additional arguments for temporary storage.

The wider variety of storage allocation methods available in Standard C is a definitive advantage. This makes Standard C more portable and convenient to use for many types of applications.

2.7 Additional Considerations

Another advantage that Standard C has over Fortran-77 is the ability to directly communicate with the program's environment through the `argv` and `argc` parameters of the `main` entry function.

Example 6

```
main (int argc, char *argv[ ]) {

   /* argc: number of arguments */
   /* argv: array of argument pointers */

}
```

The ability to pass arguments into a program at start-up time allows communication with the environment in a portable way. Fortran-77 does not provide any similar mechanism. Either nonportable features or indirect methods involving source files must be used for this kind of communication.

Standard C has explicit short circuit operators. The && and || operators control the evaluation of the second operand.

```
/* evaluate 'right' only if 'left' is true */

   left && right

/* evaluate 'right' only if 'left' is false */

   left || right
```

Because Fortran-77 does not mandate short circuit evaluation, a portable program cannot rely on it.

Standard C has bitwise operators that manipulate the bits of integral values.

• one's complement	~
• binary and	&
• binary or	|
• exclusive or	^
• shift right	>>
• shift left	<<

Fortran-77 does not have any bitwise operators. In many ways Standard C has clear advantages over Fortran-77 that make problem solving easier.

3 STANDARD C ENHANCEMENTS

It should be noted that Traditional C (i.e., C as it existed prior to the ANSI and ISO standards) was used primarily for systems applications, and the language's development reflected the needs of systems programmers. Because C has grown in popularity, the strengths of the language make it appealing for different types of applications. The C standard has been an aid to portability. There are already many production quality Standard conforming C compilers available for a variety of systems. There are several enhancements in the C standard that make numerical programming easier. These enhancements are honoring parentheses, defining additional floating-point arithmetic.

defining a standard mathematical library, and defining a set of floating-point characteristics.

3.1 Honoring Parentheses

One of the more surprising features of Traditional C is that parentheses are not honored for certain operators. Traditional C compilers may evaluate the following expression:

```
(a + b) + c
```

as any one of the following:

```
a + (b + c)
(a + c) + b
(a + b) + c
```

The Traditional C compiler is free to reorder the evaluation of operators that are both commutative and associative even in the presence of parentheses. Although the intent was to permit optimizations. it produces problems when trying to control the amount of relative error accumulating in certain floating-point operations. Unlike the mathematical real numbers, floating-point addition is not associative because the infinite amount of information contained in a real number can only be approximated in a finite floating-point format. This is demonstrated by the following contrived example. compiled with a Standard C Compiler (SCC), and run on a Cray Y-MP computer.

Example 7

```
#include <stdio.h>

double a, b;
double c = 524288.0;

main() {

   a = c * (1 << 48);     /* 524288 * 2**48 */
   b = -a;

printf("(a + b) + c = %f \n", (a+b)+c);
printf("a + (b + c) = %f \n", a+(b+c));

}
```

The output from this program is:

```
(a + b) + c = 524288.000000
 a + (b + c) = 0.000000
```

The numerically accurate answer, 524288, is obtained when the operation a+b is performed first. Thus there is a demonstrable need to control the order of evaluation with floating-point arithmetic. The only solution available in Traditional C is to store intermediate results in explicit temporaries (i.e., tmp=a+b). Thus, Traditional C imposes the burden of writing a simple statement such as:

```
x = (a + b) + (c + d);
```

in an unnatural and more complicated way with the following explicit temporaries:

```
tmp1 = a + b;
tmp2 = c + d;
x = tmp1 + tmp2;
```

A numerical programmer wants to be able to express mathematical concepts and needs to have parentheses honored. The C standard now requires an implementation to honor parentheses. This change was made solely for the purpose of making C more suitable for numerical applications. Fortran has always required parentheses to be honored, and the X3J11 committee that defined the C standard greatly enhanced C's usefulness for numerical programming by placing the same requirement on C implementations.

3.2 Floating-point Arithmetic

Another deficiency of Traditional C is the existence of only one type of floating-point arithmetic. Traditional C required all floating-point operations to be performed in type `double`. The type `float` existed only to conserve memory. All operands of type `float` were converted to type `double` before any arithmetic was performed. Clearly, the bias in Traditional C was toward integer arithmetic where arithmetic was defined for the four integral types `int`, `unsigned int`, `long`, and `unsigned long`.

The presence of four types of integer arithmetic is now complemented by three types of floating-point arithmetic. The C standard defines three floating-point types: `float`, `double`, and `long double`. The new `long double` type must be at least as precise and contain at least as much range in the exponent as type `double`. The restriction that all operands of type `float` be converted to type `double` before any operation is performed has been removed. The implementation may convert floating-point operands to a wider precision if that is desirable, but this conversion is not required. This important change provides a variety of arithmetic types from which to choose.

However, these are not the same rules that are present in Fortran-77. The C standard permits an implementation to map all three floating-point types on the same underlying precision and range. Fortran-77 requires that variables with type DOUBLE PRECISION occupy twice as much storage as variables with type REAL. The implication is that DOUBLE PRECISION provides greater precision than REAL although the Fortran-77 standard does not explicitly require this behavior. Regardless of what the Fortran-77 standard requires the marketplace demands that all Fortran vendors provide greater precision for DOUBLE PRECISION. It is not clear that all C vendors offer the same support, which again reflects the traditional use of C for systems programming. Finally, Fortran-77 does not allow floating-point operations to be performed in a wider precision.

3.3 Standard Mathematical Library

One optimization not permitted in Traditional C is the ability to recognize standard transcendental functions and perform them inline or through a fast interface. Traditional C allows a programmer to redefine any function in the library. This means a programmer is at liberty to define a function named tan and expect all calls to tan to use that function in place of the standard library function. Actually, a function named tan need not compute a tangent at all, but could, for instance, return a value that represents the color tan. Therefore, there is no way in Traditional C to tell at compile time that a particular function is a standard mathematical function. This can significantly affect performance of the standard mathematical functions (e.g., pow, exp, log, sqrt, etc.).

The C standard actually reserves the names of all standard functions. This means that if the function tan is called in the presence of the <math.h> header, the compiler can assume that the standard library function is being called. This allows the implementation to replace calls to standard library functions with inline code or special intrinsic versions with fast entry and exit sequences. Furthermore, the C standard permits functions to be called with "assignment compatible" arguments, which is not allowed in Traditional C. This means pow can, for instance, be called with either of its actual arguments being an integral type. The following statement is required to behave as if the 2 were converted to 2.0 (in the presence of <math.h>) before the function is called.

```
y = pow(x, 2);
```

This permits pow(x, 2) to be evaluated inline and to be treated as x**2 when appropriate.

However, the Standard C rules are not quite as convenient as the Fortran-77 rules from the numerical programming point of view. None of the standard transcendental library functions are defined to return either long double or float values. Additional names, such as tanl and tanf that accept and return values with types long double or float, respectively, are reserved by the standard for future use. However, this means any usage of these functions in programs is not currently portable. Even if they become portable in a future standard the number of names that must be remembered is inconvenient at best.

Fortran-77 defines a generic intrinsic function TAN that accepts arguments for the types REAL, DOUBLE PRECISION, and COMPLEX. Fortran's rules are more convenient because the generic intrinsic functions are overloaded to accept arguments with different types. Finally Fortran-77 provides an exponentiation operator, while C only provides a library function. There is no analogy to the expression i**n because Standard C forces the result to have a floating-point result. This can result in a serious performance penalty for any application that performs a reasonable amount of integer exponentiation.

3.4 Floating-Point Characteristics

Another floating-point enhancement provided by Standard C is the <float.h> header. This header contains a set of macro names that provide useful information about the floating-point characteristics of the implementation. The following is the list of names provided for type float and a brief description of their characteristics:

FLT_ROUNDS	rounding mode for floating-point addition
FLT_RADIX	base of the exponent
FLT_MANT_DIG	number of base digits in mantissa
FLT_DIG	number of decimal digits in mantissa
FLT_MAX	maximum positive representable number

FLT_MIN	minimum positive representable number
FLT_MIN_EXP	exponent of smallest positive number
FLT_MIN_10_EXP	smallest negative X such that $10^X \geq$ FLT_MIN
FLT_MAX_EXP	exponent of largest positive number
FLT_MAX_10_EXP	largest positive X such that $10^X \leq$ FLT_MAX
FLT_EPSILON	smallest positive X such that X + 1.0 ≠ 1.0

There are identical sets of names for type `double` that begin with DBL instead of FLT and for `long double` that begin with LDBL. These macro names can be used to interrogate the system at run time about useful floating-point characteristics. These floating-point characteristics are defined in terms of a floating-point model and the model is defined in terms of the following parameters:

s	sign (± 1)
b	base or radix of exponent representation
e	exponent (integer between $e_{\min}$ and $e_{\max}$)
p	precision (number of base-b digits in significand)
f_k	nonnegative integer $< b$

A normalized floating-point number $x (f_1 > 0 \text{ if } x \neq 0)$ is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \; e_{\min} \leq e \leq e_{\max}$$

The Fortran-77 standard does not define any way to portably interrogate for floating-point characteristics. However, the definition of the Standard C floating-point model was taken from the Fortran-90 standard to maintain some commonality across language standards. The addition of the floating-point model to the C standard is a valuable aid to writing portable numerical and scientific applications.

4 REMAINING NUMERICAL C DEFICIENCIES

Although the C standard contains features that make C more desirable for numerical programming than its predecessors, there are still deficiencies that are severe enough to tilt the scales in favor of Fortran-77 for certain types of applications. The severest deficiencies still present in Standard C are the absence of complex and variable length array types, error reporting through a globally modifiable object (`errno`), and performance problems associated with unrestricted aliasing. These important issues are being examined by committee X3J11.1 (NCEG) and must be resolved in order to make Standard C a viable alternative to Fortran-77 as a numerical language. The following is a discussion of proposals to extend Standard C that are being considered by committee X3J11.1.

4.1 Complex Arithmetic

The absence of a predefined complex type in Standard C forces programmers to define their own complex type. The most common way of accomplishing this is through a typedef, similar to the following:

```
typedef struct { double re, im; } complex;
```

Although this allows declarations of objects with a complex type, it inhibits the use of standard infix operators such as: /, *, −, +. Instead functions or macros must be defined to perform these operations. This means that using natural infix expressions such as:

```
a = (b + c) * (d + e);
```

is not accommodated. Instead, programmers are forced to write with a functional notation such as:

```
a = CMUL(CADD(b, c), CADD(d, e));
```

Substituting a functional notation for the elegant infix operators makes the expression harder to decipher. An application requiring extensive use of complex arithmetic is difficult to code in C. Standard C needs a complex type. Fortran has supported a complex type for many years. An application that requires a complex type if probably going to be much easier to develop in Fortran-77 than in Standard C.

Adding a new type to a language is difficult to get "right" because of the complexity associated with closure on the language. Committee X3J11.1 is attempting to define a complex extension to C and has identified a number of issues that must be resolved before this language extension is approved. For instance, an obvious approach to this extension is to add a new keyword, `complex`, to the language, permitting declarations of complex types. However, too many programs already exist that use:

```
typedef struct { double re, im; } complex;
```

and branding these programs as nonconforming by carelessly adding a new keyword, is unacceptable. The solution is to add a new header `complex.h` that introduces the new type. Existing programs will not include this header and therefore will not be affected.

Because there are three floating-point types, there should be three corresponding complex types. These types are declared: `float complex`, `double complex`, and `long double complex`. Complex constants are provided by introducing a new suffix, `i`. A constant with an `i` suffix has a complex type, the real part has the value 0, and imaginary part has the value of the constant.

`3.14fi`	float complex
`3.14i`	double complex
`3.14Li`	long double complex

A complex constant with a non-zero real part is created with an expression like: 2.3 + 4.5i (real part is 2.3 and imaginary part is 4.5).

With the addition of complex types in C, new arithmetic conversion rules are needed to define the result type of expressions that contain both real and complex operands. For instance, if one operand has type `long double` and the other operand has type `float complex`, the rules should be such that the most information is preserved. Therefore, the rules are enhanced to produce a result type of long double `complex`. The proposed rules are described below.

All types have three type attributes called the ***dimension***, the ***format***, and the ***length***. The dimension attribute specifies whether the values of the type can be represented on a one-dimensional line (i.e., real numbers) or on a two-dimensional plane (i.e., complex numbers). The format attribute specifies whether the values of the type are represented with an exponent part (i.e., floating numbers) or without an exponent part (i.e., integral numbers). The length attribute specifies how many bits are used to represent the magnitude and precision of the type. The values of each of these attributes are ranked, from highest to lowest (Table 1). For example, complex ranks higher than real for the dimension attribute.

Many binary operators that have operands of arithmetic types cause implicit conversions of one or both operands. The purpose of the conversions is to yield a common format and length for the two operands and the type of the result. These implicit conversions of the operands are called the usual arithmetic conversions.

The conversions shall preserve the original magnitude and precision of both operands except that precision may be lost when an integral type is converted to a floating type. This will occur if the magnitude of the integer is too greater for the mantissa of the floating type to represent exactly. The rules for the usual arithmetic conversions are:

1. The dimension of the result type is that of the higher ranking dimension of the operands.
2. The format of the result type is that of the higher ranking format of the operands.
3. If the format of the result type is floating, then the length of the result type is that of the higher ranking floating length of the operands. If the format of the result type is integral, then the integral promotions are performed on both operands.

Complex library functions need to be defined for the complex types. However, the lack of intrinsic functions in C is again a serious impediment. Defining complex library functions such as `sin`, `cos`, `exp`, `log`, `pow`, `sqrt`, and `abs` requires

Table 1. Values of Each Attribute

Dimension	Format	Floating Length	Integral Length
Complex numbers	Floating numbers	Long double	Unsigned long Signed long
Real numbers	Integral numbers	Double Float	Unsigned int Signed int

three additional names for each function. For example:

```
csinf complex sine for float complex
csin  complex sine for double complex
csinl complex sine for long double complex
```

This is in addition to `sinf`, `sin`, and `sinl` that are already reserved by the C standard for floating-point numbers. The proliferation of names is extreme and again it is apparent that some form of generic intrinsic name is needed.

4.2 Variable Length Arrays

Another deficiency that might inhibit the use of C is the absence of variable length arrays. Because arrays must be defined with a constant dimension, there is no way in C to declare an array whose size is dynamic. Function arguments that are arrays are implicitly converted to pointers to the first element before the function is called. This pointer can be used to access all of the array elements. It is easy to define a function that operates on single dimensioned arrays of any length. However, a problem still exists with multidimensional arrays. For instance, a two-dimensional array is converted to be a pointer to an array and the array portion of the type must still contain a constant dimension. This prevents a simple definition of a function that performs a matrix operation on arbitrary M × N matrices. Ideally, Standard C would permit a declaration of a function performing a matrix multiply to look similar to the following:

```
/* a = b × c */
void mxm(int n, int m,
double a[n][m], double b[n][m],
                double c[n][m]);
```

The size of each array is dynamically determined each time the function is called. Because an array is always converted to a point to the first element of the array whenever it is passed as a function argument, the problems seems to exist only for multidimensional arrays. However, the inability to declare a one-dimensional stack array that is the same size as a formal parameter suggests that this feature is desirable for all array types that reside on the stack. This would permit:

```
f(int n, double ary[n]) {
   double tmp[n];
   /* ... */
}
```

to declare both the formal parameter `ary` and the stack array `tmp` to be variably dimensioned arrays with the same size. This approach is analogous to the Fortran notion of adjustable and automatic arrays.

Committee X3J11.1 is looking at two proposals that extend C along these lines. The first approach is the one described above. A second approach involves the use of descriptors that are capable of representing the address of the array and the size of each dimension. For example:

```
void func(double (*desc)[?][?])  /* tentative syntax */
```

declares a parameter `desc` that is a descriptor with three pieces of information: address of the base, length of the first dimension, and length of the second dimension. The type of parameter `desc` is pointer to adjustable array of adjustable array of double. This descriptor can be used to reference arrays whose dimension sizes vary at execution time.

Fortran-77 does not support variable length stack arrays. It is a very common extension to most Fortran implementations, however. One additional advantage available with Fortran-77 arrays is the ability to specify both upper and lower bounds for each dimension. C arrays are always zero based and the length of the dimension is specified (not the bounds). Some problems are more naturally viewed as non-zero based.

For completeness sake it should be noted that there is an existing solution to the variable length array problem that is standard conforming. Essentially, this solution involves using the library function `malloc` to dynamically allocate an array of pointers. Each pointer element of the array points to a different row of the matrix. This array of pointers permits access to the entire matrix through multiple indirection. The following example demonstrates this technique by performing a matrix multiply with dynamically allocated arrays of pointers.

Example 8

```
#include <stdlib.h>

#define M 10
#define N 20

double a [N][N], b[N][M], c[M][N];
void mxm(int n, int m, double **a, double **b, double **c);

main(){
   double **pa, **pb, **pc;
   int i;

   /* allocate arrays of pointers */
   pa = malloc(N*sizeof(double *));
   pb = malloc(N*sizeof(double *));
   pc = malloc(M*sizeof(double *));

   /* setup array of pointers */
   for (i=0; i<N; i++) {
      pa[i] = a[i];              /* pa[i] points to i-th row of a */
      pb[i] = b[i];              /* pb[i] points to i-th row of b */
   }

   for (i = 0; i < M; i++)
      pc[i] = c[i];              /* pc[i] points to i-th row of c */

   mxm(N, M, pa, pb, pc);

   free(pa); free(pb); free(pc);  /* free allocated arrays */

} /* main */

/* a = b × c (for arbitrary N×M shapes) */
void mxm(int n, int m, double **a, double **b, double **c) {
   int i, j, k;

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         a[i][j] = 0;
         for (k = 0; k < m; k ++)
            a[i][j] += b[i][k] * c[k][j]; /* double indirection */
      }
} /* mxm */
```

The following method describes the high level algorithm used in Example 8.

1. The pointers pa, pb, and pc each point at an array of pointers dynamically allocated by malloc.
2. Each pointer element pa[i], pb[i], and pc[i] is assigned to point at each row of matrices a, b, and c, respectively.
3. pa, pb, and pc are passed to the function mxm, which performs the matrix multiply using double indirection.

4. The memory for the dynamically allocated arrays is freed.

The problems associated with this approach are complexity and extensibility. The solution is neither intuitive nor easy to understand. Finally, the solution does not extend easily to three or four dimensions. The three-dimensional solution requires an array of pointers each pointing at a two-dimensional array of pointers, each pointing at a row of the three-dimensional array. The complexity grows rapidly with each new dimension.

4.3 The errno Macro

The errno macro expands into an lvalue expression that specifies a storage location that the C environment modifies under exceptional conditions. The worst problem with errno is its interaction with the standard transcendental functions defined in the <math.h> header. Standard C prevents any mathematical library functions from causing a visible exception. That is, the program cannot stop execution or pass control to a signal handler, just because a mathematical function cannot compute its result. There are two reasons why these functions might not be able to compute their results: domain error or range error. A domain error occurs if an input argument is outside the domain over which a mathematical function is defined (e.g., sqrt (−1)). When a domain error is detected the value of a macro named EDOM is stored in errno. A range error occurs when the result of a mathematical function cannot be represented as a double value (e.g., pow(DBL_MAX, DBL_MAX)). When a range error is detected the value of a macro named ERANGE is stored into errno, and if the function result overflows the function returns the value of another macro named HUGE_VAL. This means that in the expression:

```
pow (x1, y1) * pow(x2, y2)
```

neither call to pow is allowed to cause an exception but the multiplication operator is! Typically, the value of HUGE_VAL is large in magnitude, meaning the multiplication can easily overflow and cause an exception. This implies that the preferred technique is to use explicit temporaries as follows:

```
errno =0;
tmp1 = pow(x1, y1);
tmp2 = pow(x2, y2);
if (errno != 0) panic(errno);
```

This technique proliferates the unnatural use of temporaries.

Another problem with errno semantics is that it inhibits automatic vectorization and parallelization of loops containing calls to math functions as the following example demonstrates:

Example 9

```
for (i = 0; i < 100; i ++) {
   errno = 0;
   x[i] = sqrt(y[i]);
   if (errno != 0) panic(errno);
}
```

This for loop cannot be vectorized because if, for example, the seventh iteration contained a range error then the first six values must have been fully computed before the error was detected. A pipelined architecture does not necessarily guarantee that the first six vector elements are fully evaluated if the seventh element produces an exception. Now if errno is omitted from the example, as follows:

```
for (i = 0; i < 100; i++) {
   x[i] = sqrt(y[i]);
}
```

then the error condition goes undetected because the sqrt function cannot fail. This affects both automatic vectorization and parallelization. This is such a limiting condition that vendors who provide high-performance systems are forced to provide two environments: a strictly conforming environment that supports errno but does not vectorize or parallelize loops containing calls to math functions, and an environment that disassociates errno from the math library. Modern supercomputers can often times compute the results faster than the code needed to detect the error and update the memory location designated by errno. Attempting to define error conditions in terms of a globally modifiable object creates many problems including severe performance degradations. Committee X3J11.1 is exploring a new mathematical library definition without the presence of errno.

The Fortran-77 standard wisely chose to say nothing about exceptional conditions. This allows the compiler to optimize statements containing calls to the transcendental functions in a standard conforming way. One of Fortran's greatest assets is its potential for optimization.

4.4 C Aliasing

Many numerical programs are computationally intensive and benefit from optimizations. The optimization capability of a vendor's Fortran compiler is often times crucial to providing access to the full capabilities of the hardware from a high-level language. This is especially true for supercomputer environments where compilers perform automatic vectorization and automatic parallelization. Supercomputer sales are largely based on the performance the system can deliver.

C pointers present problems for optimizing compilers because they introduce hidden aliases. Essentially, whenever an object is modified through a pointer, the compiler must make worst case assumptions if it cannot determine which object is being modified. Parallel processing is one current approach being taken to increase performance of many computationally intensive applications. The hidden aliases introduced by C pointers make automatic compile time detection of parallelism an intractable problem. Aliases must be resolved before a compiler can determine that a loop is safe to parallelize. For instance, it is important to know how the elements of an array are accessed and modified. Consider the following loop:

Example 10

```
int a[5] = {0, 2, 4, 6, 8};

void f( ) {
   int i;

   for (i = 1; i < 5; i++)
      a[i] = a[i-1] + 1;
}  /* f */
```

The semantics of this loop dictate the following execution order that produces scalar results.

```
a[1]=a[0]+1;
a[2]=a[1]+1;
a[3]=a[2]+1;
a[4]=a[3]+1;
```

Vectorization results require the order of accesses and modifications to be rearranged. The following reordering allows several array elements to be accessed and modified simultaneously.

```
V[0]=a[0];
V[1]=a[1];
V[2]=a[2];
V[3]=a[3];
a[1]=V[0]+1;
a[2]=V[1]+1;
a[3]=V[2]+1;
a[4]=V[3]+1;
```

One common parallel optimization technique is to simultaneously execute different iterations of a loop on multiple processors. This means the order of references to objects occurring in different loop iterations is undefined. The following results are obtained for array **a** by each method.

```
scalar results:     0, 1, 2, 3, 4
vector results:     0, 1, 3, 5, 7
parallel results:   indeterminate
```

The scalar results are always correct because that is what is dictated by both the Fortran-77 and C standards. Any parallelization that is performed must preserve the scalar results. In general, if an object referenced in a particular iteration of the loop is also modified in a different iteration of the loop, then automatic vectorization and parallelization must somehow preserve the order of the accesses and modifications of that object. Example 5 contains the easily detectable aliases a[i] and a[i-1]. A compiler can detect this alias at compile time and generate a scalar loop. However, the following example demonstrates that C pointers can introduce hidden aliases that are not detectable at compile time.

Example 11

```
#include <stdio.h>

int a[6] = {0, 1, 2, 3, 4, 5};
int b[6] = {9, 8, 7, 6, 5, 4};
int c[6];
```

```
06
07 void blackbox(int *p1, int *p2, int *p3, int n);
08
09 main( ){
10    int i;
11
12    blackbox(c, b, a, 6);          /* no aliases */
13    for (i=0; i<6; i++)
14       printf(" c[%d] = %d ", i, c[i]);
15    putchar('\n');
16
17    blackbox(&a[1], &a[1], a, 5);        /* aliases */
18    for (i=0; i<6; i++)
19       printf(" a[%d] = %d ",i, a[i]);
20    putchar('\n');
21 }  /* main */
22
23 void blackbox(int *p1, int *p2, int *p3, int n) {
24    int i;
25
26    for (i=0; i<n; ++i)
27       *p1++ = *p2++ + *p3++;
28 }  /* blackbox */
```

The following output is produced when the program is executed in scalar fashion.

```
c[0] = 9 c[1] = 9 c[2] = 9 c[3] = 9 c[4] = 9 c[5] = 9
a[0] = 0 a[1] = 1 a[2] = 3 a[3] = 6 a[4] = 10 a[5] = 15
```

The function **blackbox**, whose definition starts on line 23, appears to add the corresponding elements of two arrays together, storing the results into a third array. This is exactly what happens when **blackbox** is called without any aliases at line number 12. The resulting array c contains the sum of **a** and **b**. This makes the loop inside **blackbox** appear to be a candidate for parallelization. However, when **blackbox** is called with aliases at line number 17 something different happens. Each element of the resulting array **a** contains partial sums of the values in the preceding elements. This time the loop must be executed as a scalar loop to obtain the correct results. Because **blackbox** is not declared static it can be called from a separately compiled module. Therefore, the compiler must make the worst case assumption that this loop might contain aliases. C does not provide any way to restrict the aliasing of formal parameters that are pointers.

Fortran-77, however, does not permit aliasing through formal parameters if the actual object is modified. This means that a formal parameter cannot reference the same object referenced by another formal parameter, nor a global object that is part of a COMMON block. Fortran-77 rules require subroutine arguments to behave as if they are copied in when the subroutine is called and copied out when the subroutine returns control to its caller. Because the order in which arguments are copied in and out is unspecified, aliases produce unpredictable results. Fortran-77 is a proven performer that provides reasonable semantics for exploiting automatic parallelism.

C aliasing is not only the most critical remaining deficiency, it is one of the most difficult to resolve. Many ideas have been proposed that solve part of the problem but none have provided a general solution that encompasses all pointers. One promising proposal that is currently being explored by committee X3J11.1 is a new kind of pointer called a restricted pointer. A restricted pointer gives the compiler the liberty to assume that the pointer behaves like an array for aliasing purposes. That is, because the compiler can assume that two different arrays are not aliases with each other, it can also assume that two different restricted pointers are not aliases.

```
void func(double *restrict p, double *restrict q)
```

This example shows how formal parameters p and q can be declared as restricted pointers by using a new keyword restrict. The compiler can assume that p and q point to different objects.

Another useful application of restricted pointers is to point at space allocated by the dynamic memory allocation functions, calloc, malloc, and realloc. The C standard [1] guarantees that "each such allocation shall yield a pointer to an object disjoint from any other object."

```
double * restrict p = malloc( N * sizeof(double) );
```

The simplicity of this proposal makes it easy to comprehend. This is important because incorrect usage results in undefined behavior. The simplicity of this proposal also makes it easy for an optimizer to exploit, because the same logic currently being applied to arrays can now be applied to restricted pointers. Finally, it is trivial to port an application that uses the new restrict keyword to other environments because the following preprocessing directive:

```
#define restrict
```

harmlessly eliminates all occurrences of the keyword.

The C-aliasing problem was identified as the highest priority issue facing committee X3J11.1 at its initial meeting but is also recognized as one of the most difficult issues on which to reach consensus.

5 CONCLUSIONS

C is an important language that is not used very often as a numerical or scientific programming language. Even if C is the language of choice for some people they might be dissuaded from using it for a particular application because of the deficiencies documented above. Many of C's advantageous features could be put to good use in numerical codes if the current limitations are overcome. Certainly Fortran-77 will be a popular numerical language for a long time. C, on the other hand, must continue to evolve and improve or it will remain a secondary language for numerical and scientific programming. Committee X3J11.1 is a vehicle for exploring the evolution of C into a better numerical language. Whether the resulting language gives programmers a usable language remains to be seen.

REFERENCES

[1] ANSI X3J11 Committee, *American National Standard X3.159-1989, Programming Language C* (approved December 14, 1989). Santa Ana, CA: Global Engineering Documents, Inc.

[2] ANSI X3J3 Committee, *American National Standard X3.9-1978, Programming Language FORTRAN* (approved April 3, 1978). Santa Ana, CA: Global Engineering Documents, Inc.

[3] ISO/IEC JTC1/SC22/WG5 Fortran Working Group, *International Standard 1539:1991 Programming Language FORTRAN*. Santa Ana, CA: Global Engineering Documents, Inc.

[4] T. MacDonald, "C for numerical computing," *J. Supercomputing*, vol. ?, pp. 31–48, 1991.

[5] T. MacDonald, "C language and numerical programming," *J. C Lang. Translation*, pp. 9–16, 1989 (sample issue).

[6] T. MacDonald, "Adding complex arithmetic to C," *J. C Lang. Translation*, vol. 1, pp. 20–31, 1989.

[7] T. MacDonald, "Aliasing issues in C," *J. C Lang. Translation*, vol. 1, pp. 83–95, 1989.

[8] P. J. Plauger and J. Brodie, *Standard C: Programmers Quick Reference Guide*. Redmond, WA: Microsoft Press, 1989.

[9] R. Jaeschke, *Portability and the C Language*. Indianapolis, IN: Hayden, 1989.

[10] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall, 1988.

[11] S. Harbison and G. Steele, *C, A Reference Manual* 3rd ed.). Englewood Cliffs, NJ: Prentice Hall, 1991.