The Journal of

C Language Translation

Volume 3, Number 4

March, 1992

Publisher and Editor ..... Rex Jaeschke
Technical Editor ..... P.J. Plauger
Standards Editor ..... Jim Brodie
Numerical Editor ..... Tom MacDonald
Technical Consultant ..... David Prosser
Subscriptions ..... Jenny Jaeschke

The Journal of C Language Translation (ISSN 1042-5721) is a quarterly publication aimed specifically at implementers of C language translators such as compilers, interpreters, preprocessors, *language*-to-C and C-to-*language* translators, static analysis tools, cross-reference tools, parser generators, lexical analyzers, syntax-directed editors, validation suites, and the like. It should also be of interest to vendors of third-party libraries since they must interface with, and support, vendors of such translation tools. Companies committed to C as a strategic applications language may also be interested in subscribing to *The Journal* to monitor and impact the evolution of the language and its support environment.

Editorial: Address all correspondence to 2051 Swans Neck Way, Reston, Virginia 22091 USA. Telephone (703) 860-0091. Electronic mail address via the Internet is *jct@aussie.com*.

Subscriptions: The cost for one year (four issues) is $235. For three or more subscriptions billed to the same address and person, the discounted price is $200. Add $15 per subscription for destinations outside USA and Canada. All payments must be made in U.S. dollars and checks must be drawn on a U.S. bank.

Submissions: You are invited to submit abstracts or topic ideas, however, *The Journal* will not be responsible for returning unsolicited manuscripts. Please submit all manuscripts electronically or on suitable magnetic media. Final copy is typeset using TEX with the LATEX macro package. Author guidelines are available on request.

Contents

27. A C++ Library

P.J. Plauger

Abstract

The standard library that accompanies C++ has mostly been inherited unchanged from C. X3J16/WG21 are now debating what changes and additions to make for the ANSI/ISO standard. Proposals have ranged from minimalist to significant additions.

This paper traces the historical developments that have brought the C++ library to its present form. It discusses several philosophies that can be followed in "completing" the library. And it ends by outlining an implementation that I am currently undertaking. My goal is to shed some light on the issues raised by these differing philosophies.

Early History

I have recited the history of the C library on several occasions. (See, for example, *The Standard C Library*, Prentice Hall, 1992. Chapter 12: `<stdio.h>` in particular discusses the evolution of the C I/O model.) It's probably worth while, however, to outline that history here once again, if only for completeness. The current form of the C++ library is more an historical accretion than the result of any concerted planning.

The C library had its roots in UNIX, of course. That's where the C language was born and spent its formative years. At the heart of that library were forty-odd functions. Each of these performed a different UNIX system call from a C program. In those early times, you could write quite a few of the simpler text-processing filters (such as `cat` and `cmp`) with little additional code.

The UNIX pioneers were tool builders par excellence. Almost reflexively, they identified and wrote functions that were highly reusable. Before very long, the first C library took form. It was simply an accretion of these functions. A few of the functions have changed over the years, but many have been remarkably stable.

C's cultural predecessor is Fortran. You can see that in the formatted I/O conventions of the early C library. (Even the name `printf` is revealing.) The math library is a very direct imitation. Only the string functions represent a significant new departure. C has obvious advantages over Fortran in manipulating pointers and individual characters.

C is arguably inferior to Fortran in several fundamental ways. The PDP-11, where C began, discouraged floating-point arithmetic in mixed precisions. That encouraged Dennis Ritchie to favor type `double` in both expression arithmetic and arguments to functions. The C Standard softened this bias in expression arithmetic but not in the library. C provides only a `double` library, not the two (or more) precisions that Fortran programmers love to use.

Part of the problem is a lack of generics in C. Fortran lets you write `SIN(X)`, for example, to call either the `REAL` or `DOUBLE PRECISION` versions of the sine function. The argument type determines which version gets called. C insists that you give a different name for each flavor of a function.

C also omits complex arithmetic. You can put it back easily enough in the form of a library of functions, to be sure. Still, the Fortran programmer often misses the ability to write arithmetic expressions that include complex operands. Without user-defined operator overloading, C cannot compete with the notational convenience of Fortran.

C Spreads Out

The C language did emulate, and surpass, Fortran in another important arena. It became a portable language of extraordinary power. Moving C away from UNIX challenged the library severely. The gloriously simple UNIX I/O model could no longer be relied upon.

A layer of insulation went into the library. It stood between the functions that read and write characters and those that do battle with the operating system. To offset potential inefficiencies, it buffered I/O by larger blocks of characters. Thus was born the stream I/O functions, and the header `<stdio.h>`.

Other changes took place as well. Those forty-odd UNIX system calls got worked over. Many could be emulated atop other operating systems, sometimes with small compromises in semantics. Others fared less well. In the end, both C and the world's operating systems changed to meet each other. The spreading influence of UNIX has made C ever easier to port.

With spreading popularity came increasing numbers of potential contributors to the library. (It's easier to toss in another library function, by far, than to add a keyword or semantic construct to the language.) Even before C spread widely outside Bell Laboratories, it had accreted a considerable number of additional functions.

That accretion was far less planned than the construction of the initial C library. A broader community of programmers got involved, with a broader spectrum of styles. The header `<string.h>` is a living monument to this eclectic approach to library building.

Not all of the additions to C served the lofty goal of improved portability. Microcomputers and exotic operating systems each made their special demands. And for every change in the language proper, you can be sure there are ten special library functions as well.

C Gets Standardized

Efforts to standardize the C library actually began several years before the language itself was addressed. In 1980, the precursor of what is now the POSIX operating system standard got under way. Its initial goal was to describe a portable C programming environment for people writing UNIX-based applications. Some of us were determined to broaden this standard to include "UNIX-like" systems. Hence, the shared goal was to minimize C library dependencies on both individual machines and individual operating system vendors.

The POSIX standard effort was well under way when X3J11 began standardizing C. We (X3J11) commandeered much of the work of the POSIX committee as the initial draft of the library portion of the C standard. They had already eliminated the PDP-11isms for us. Our remaining task was to eliminate the UNIXisms. (POSIX got left with what is now called the C binding to the operating system services.)

It was tempting to eliminate the irregularities in the C library as well. X3J11 included considerable expertise on what was both good and bad about programming in C. We probably could have redesigned the C library and got it mostly right. But more conservative sentiments prevailed. Already, sufficient C code existed to make changes in the library expensive for a large constituency.

Adding to the library was a different matter, however. Often, the easiest compromise was to toss in yet another library function to satisfy the perceived need of a significant minority. The cost to individual programmers was primarily name space pollution. Each addition was one more name with external linkage that programmers must be aware of.

The cost, in the end, to implementors has been considerable. My experience was that the C standard at least doubled the size of the C library. If you include proper support for locales and large character sets, it more than trebles in size. (In 1978, the C library I wrote took 2,500 lines of C code. In 1986, the ANSI C version exceeded 5,000 lines. My latest implementation, with locales and all, exceeds 8,000 lines. Other implementors report similar ratios.)

Enter C++

C++ essentially started with an "almost ANSI" C library. It reflected AT&T's serious attempt to track the evolving ANSI standard. It also reflected the inevitable lags. AT&T has an enormous amount of old code to keep alive. The company must also choose its new directions with care. How it supports large character sets in C, for example, has consequences across multiple product lines.

To this good-but-not-great beginning was added the stream classes. Of all the notational advantages possible with C++, the stream I/O probably offers the sexiest. It is easy to sympathize with those who put their energies into streams even as they ignored the rest of the library. After a couple of major revisions, the stream classes appear to be settling down.

Complex arithmetic is another popular theme. Many people want it, it's not in C, and it's largely self-contained. It is also the best candidate for a full bore language extension. Many of the arithmetic operators in C have obvious meaning for complex operands. You can overload lots of operators without getting precious.

Other class libraries tend to be more ambitious. They don't just pave over the irregularities inherent in the C library. Instead, they aim more for the object-oriented ideal. Hide all details of data representation. Provide cradle-to-grave control of stored values. Create temporaries on the fly as needed. Define a broad class of overloaded operators and implicit conversions.

Those are good design principles to follow for most classes. But they often impose overheads that are alien to your typical C programmer. Make them a necessary part of the C++ library and you raise the intellectual cost of migrating from C.

Standardizing C++

These are some of the issues now facing X3J16/WG21, the joint ANSI/ISO committee standardizing C++. They have lots of issues to deal with in the language proper. They must also decide the scope and extent of a standard library for C++.

Part of the library specification is easy in a way, if unappealing. There are strong pressures to keep C++ as compatible as possible with ISO C. That means that the Standard C library will probably be included, warts and all. To omit any significant part will seriously raise the cost of migrating code (and programmers) from C.

Another part is harder. How much stuff beyond the Standard C library should be thrown in the pot? Stream classes are obvious candidates, practically shoo-ins. Complex arithmetic is also a strong contender. But what about all the other classes people have been trying out for the past several years? The committee will have to choose any additions wisely.

That's not the end of the story. C itself is growing. WG14 is actively considering a significant addition to the Standard C library. The Multibyte Support Extension proposed by the Japanese adds numerous analogs to existing functions for manipulating large character sets. And X3J11.1, the Numerical C Extensions Group, is proposing still more (albeit with less clout).

Add to that the functions already set aside for future C standards. Most important of these are probably the `float` and `long double` versions of the math library. Pressure is growing on many vendors to broaden C's numerical support. Type `double` is no longer enough for all needs.

So X3J16/WG21 face another class of decisions. Which of the standards-in-waiting are important enough to include in C++ from the outset? And if they're included, what form should they take?

Philosophies

There is only one way to make a whole set of decisions. That is to settle on a philosophy. The philosophy provides an underlying consistency that weaves the decisions into a uniform fabric. A problem the C++ committee now faces is warring philosophies. Some members view C++ as an incremental addition to C. Others see the new language as a chance to bury the horrid older one once and for all. Many other camps dot the broad middle ground.

One extreme approach to choosing a C++ library is minimalist. Adopt the Standard C library because you probably have to anyway. Throw in stream classes because they're widely used, tried, and proven. Stop there, or as soon as possible thereafter. The other extreme is purist. Use the C++ standard as an opportunity to introduce a truly uniform library. Emulate SmallTalk. Derive all library classes from a protean class. Park all the library functions as methods within these classes. If you have to keep the Standard C library, park it in a corner and discourage its use.

The middle ground has all sorts of stopping-off points between these extremes. You can tidy up the existing functionality a little or a lot. You can add a little or a lot. You can emphasize efficiency or safety. All you have to do, in any case, is convince a significant majority of the C++ community that your choice is the best one.

A Middle Ground

I believe that the community probably wants more than the minimum. I know that I lack the experience to identify the best balance point. I also believe that others can profit from additional data. For all those reasons, I have undertaken to extend the Standard C library I wrote last year. My goal then was to make an exemplary library for C that obeyed both the spirit and the letter of the ANSI/ISO C Standard. The extensions are to make the library compatible with C++ and provide the missing bits. Once again, I hope to produce a library that is exemplary in all senses of the word.

This time, however, I don't have a standard to match. The C++ standard is still very much in a state of flux. To some extent, I can guess where it's going. I can certainly track where it is on a meeting-by-meeting basis. But I can't say where it's going to end up. (A not-so-secret hope, of course, is that I can influence its eventual form in some ways.)

Some of my goals are ambitious. I am implementing `float` and `long double` versions of all the math functions. I am also implementing the full Multibyte Support Extension proposed by the Japanese. And, of course, I am providing stream and complex classes.

The project will be uninteresting if all I add are C functions. That has led me to review a number of existing class libraries and to tinker with a few of my own. So far, the classes I am tinkering with tend mostly to smooth over

irregularities in the functionality of the C library (existing and enhanced). They don't add much in the way of performance overheads.

I believe that many C programmers are attracted to C++ for its improved notation. Overloading function names, for example, avoids the proliferation of almost-same names that C suffers from. The string functions are an obvious example. A handful of similar operations are performed on

- null-terminated strings (`str`∗)
- possibly null-terminated strings of some maximum length (`strn`∗)
- arbitrary strings of a given length (`mem`∗)

Look at all the names consumed for just this simple functionality. Then look at all the additional names you have to add to C to supply `float` and `long double` math functions. Toss in upwards of three flavors of complex operands and it gets even worse. Overloading function names may be just a notational convenience, in one sense, but it is an important one for code hygiene.

It is also important for mental hygiene. At the moment, the set of string functions is incomplete. Their functionality is somewhat irregular. (`strncpy` is a notorious example). So too are their names. Here is a place where a little cleanup can go a long way. You can argue that the string functions are too small a problem to be worth solving. That may be true—until you add the Multibyte Support Extension. Now there are two types of characters and at least six types of strings. (I'm not even counting multibyte strings, which may or may not warrant many additional functions.)

Forward to the Past

My emphasis is on notational convenience with minimum cost in performance. That's a philosophy I think will be appealing to many potential C converts. It will also be found wanting by many devotees of true object-oriented design and programming. As a consequence, I may well change my tune by the time this project is complete. (My goal is to have the code, and another book, complete by the end of the year.)

I can summarize my philosophy in one sentence, though it will probably annoy many people. I believe that an important goal for a C++ library is to make C look more like Fortran.

P.J. Plauger serves as secretary of X3J11, convenor of the ISO C working group WG14, and Technical Editor of The Journal of C Language Translation. His latest book, The Standard C Library, *is available from Prentice-Hall. He can be reached at* uunet!plauger!pjp.

∞

28. A Strong Typing Facility for C

Jim Gimpel
Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426

Abstract

Strong typing as exemplified by the Pascal Language has the advantage of early detection of programming errors. It has the disadvantage of programming inflexibility. We describe in this paper how a lint-like utility can superimpose strong typing wholly or partially on a C program through the use of the `typedef` facility and in the judicious selection of appropriate options.

Additional flexibility is obtained by means of a type hierarchy. In a type hierarchy generic uses of a type are distinguished from, but related to, more specific uses of a type.

Introduction

The notion of strong typing is not usually carefully defined. It generally means the kind of type checking that Pascal has that C does not. This includes the following:

1. User-defined types match only through the nominal type, not through the underlying type as is done with C.
2. A special Boolean type is supported which must be used where Booleans are expected. In C, any scalar can be used as a Boolean and any Boolean is typed `int`.
3. The Pascal-equivalent of `char` and `enum` objects are not automatically converted to and from `int` as is done in C. Explicit conversion is required.
4. Every array has an expected index type and every subscript must match this type. In C, any integral-typed expression can be used as a subscript for any array.
5. Pascal has a set facility typically implemented as a finite number of bits that are either on or off. In C, one uses bit-wise operations on integral quantities to achieve the same effect. C's approach is more flexible but Pascal sets and their members cannot be improperly mixed.

In addition to these static checks, Pascal systems have run-time checks including subscript bounds and null pointer checks. We do not include these under the notion of strong type checking.

In the pages that follow, each of the static type checks enumerated above will be seen to be represented as options to a lint-like processor. They are presented as pages of the FlexeLint user manual. More information about FlexeLint can be obtained from the author.

Strong type checking is a mixed blessing. Rigid adherence to a strong type scheme is sometimes more trouble than it is worth. It is difficult, for example, to write generic functions that operate on nominally different types when types must match exactly. To provide necessary flexibility a type hierarchy scheme has been introduced. This also is described below.

Strong Types

Have you ever gone to the trouble of `typedef`ing types and then wondered whether it was worth the effort? It didn't seem like the compiler was checking these types for strict compliance.

Consider the following typical example:

```
typedef int Count;
typedef int Bool;
Count n;
Bool stop;

n = stop; /* mistake but no warning */
```

This "mismatch" typically goes undetected by the compiler because the compiler is required by the C standard to check only underlying types which, in this case, are both the same (`int`).

The `-strong` option and its supplementary option `-index` exist to support full or partial `typedef`-based type-checking. We refer to this as *strong* type-checking. In addition to checking, these options have an effect on generated prototypes. See *Strong Types and Prototypes*.

The `-strong` Option

Description of `-strong`

The FlexeLint syntax for this option is

`-strong(` [*flags*] [`,` *name*] ... `)`

This identifies each *name* as a strong type with properties specified by *flags*. Presumably there is a later `typedef` defining any such *name* to be a type. This

option has no effect on `typedef`s defined earlier. If *name* is omitted, then *flags* specifies properties for all `typedef`ed types that are not identified by some other `-strong` option.

The possible values for *flags* are:

`A` Issue a warning upon some kind of Assignment to the strong type (e.g., assignment operator, return value, argument passing, initialization). `A` may be followed by one or more of the following letters which soften its meaning.

- `i` ignore Initialization.
- `r` ignore Return statements.
- `p` ignore argument Passing.
- `a` ignore the Assignment operator.
- `c` ignore assignment of Constants.

As an example, `-strong(Ai,BITS)` will issue a warning whenever a value whose type is not `BITS` is assigned to a variable whose type is `BITS` except when the variable is being initialized. (If the strong type is a pointer then `&x`, where `x` is a `static` or automatic variable, is considered a constant.)

`X` Check for strong typing when a value is eXtracted. This causes a warning to be issued when a strongly typed value is assigned to a variable of some other type (in one of the four ways described above). But note, the softeners (`i`, `r`, `p`, `a`, `c`) cannot be used with `X`.

`J` Check for strong typing when a value is Joined (i.e., combined) with another type across a binary operator. This can be softened with one or more of the following letters:

- `e` ignore Equality and conditional operators.
- `r` ignore the Relational operators.
- `o` ignore the Other binary operators.
- `c` ignore combining with Constants.

By 'ignoring' we mean that no message is produced. If, for example, `Meters` is a strong type then it might be appropriate to check only Equality and Relational operators and leave others alone. In this case `Jo` would be appropriate.

`B` The type is Boolean. Normally only one *name* would be provided and normally this would be used in conjunction with other flags. (If through the fortunes of using a third party library, multiple Boolean's are thrust upon you, make sure these are related through a type hierarchy. See *Type Hierarchies*.) This flag has two effects:

1. Every Boolean operator will be assumed, for the purpose of strong type-checking, to return this type. The Boolean operators are those that indicate true or false and include the relational and equality operators, `!`, `&&`, and `||`.
2. Every context expecting a Boolean, such as an `if` clause, `while` clause, second expression of a `for` statement, and operands of `!`, `||`, and `&&`, will expect to see this strong type or a warning will be issued.

`b` This is like flag `B` except that it has only effect number 1 above. It does not have effect 2. Boolean contexts do not require the type.

Flag `B` is quite restrictive insisting as it does that all Boolean contexts require the indicated Boolean type. By contrast, flag `b` is quite permissive. It insists on nothing by itself and serves to identify certain operators as returning a designated Boolean type rather than an `int`. See also the `l` flag below.

`l` is the Library flag. This designates that the objects of the type may be assigned values from, or combined with, library functions (or objects) or may be passed as arguments to library functions. The usual scenario is that a library function is prototyped without strong types and the user is passing in strongly typed arguments. Presumably the user has no control over the declarations within a library. Also, this flag is necessary to get built-in predicates such as `isupper` to be accepted with flag `B`. See the example below.

`f` goes with `B` or `b` and means that bit-fields of length one should not be Boolean (otherwise they are). See Bit-field example below.

These flags may appear in any order except that softeners for `A` and `J` must immediately follow the letter. There is at most one `B` or `b`. If there is an `f` there should also be a `B` or `b`. In general, lowercase letters reduce or soften the strictness of the type checking whereas uppercase letters add to it. The only exceptions are possibly `b` and `f` where it is not clear whether they add or subtract strictness.

If no flags are provided, the type becomes a 'strong type' but engenders no specific checking other than for declarations.

Examples of `-strong`

The option

```
-strong(A)
```

indicates that, by default, all `typedef` types are checked on Assignment (`A`) to see that the value assigned has the same `typedef` type.

The options:

```
-strong(A)  -strong(Ac,Count)
```

specify that all `typedef` types will be checked on Assignment and constants will be allowed to be assigned to variables of type `Count`.

As another example,

```
-strong(A)  -strong(,Count)
```

removes strong checking for `Count` but leaves Assignment checking in for everything else. The order of the options may be inverted. Thus

```
-strong(,Count)  -strong(A)
```

is the same as above.

Consider the following:

```
/*lint -strong(Ab,Bool) */
typedef int Bool;

Bool gt(a,b)
    int a, b;
    {
    if(a) return a > b;    /* OK */
    else return 0;         /* Warning */
    }
```

This identifies `Bool` as a strong type. If the flag `b` were not provided in the `-strong` option, the result of the comparison operator in the first `return` statement would not have been regarded as matching up with the type of the function. The second `return` results in a warning because `0` is not a `Bool` type. An option of `-strong(Acb,Bool)` (i.e., adding the `c` flag) would suppress this warning.

We do not recommend the option `c` with a Boolean type. It is better to instead use

```
#define False (bool) 0

return False;
```

Had we used an upper-case `B` rather than lower-case `b` as in:

```
-strong(AB,Bool)
```

this would have resulted in a warning that the `if` clause is not Boolean (variable

`a` is `int`). Presumably we should write:

```
if( a != 0 ) ...
```

As another example:

```
/*lint -strong(AJXl,STRING)  */
typedef char *STRING;
STRING s;

    s = malloc(20);
    strcpy( s, "abc" );
```

Since `malloc` and `strcpy` are library routines, we would ordinarily obtain strong type violations when assigning the value returned by `malloc` to a strongly typed variable `s` or when passing the strongly typed `s` into `strcpy`. However, the `l` flag suppresses these strong type clashes.

Strong types can be used with bit-fields. Bit-fields of length one are assumed to be, for the purpose of strong type checking, the prevailing Boolean type if any. If there is no prevailing Boolean type, or if the length is other than one, then, for the purpose of strong type checking, the type is the bulk type from which the fields are carved. Thus:

```
/*lint -strong(AJXb,Bool)     */
/*lint -strong(AJX,BitField)  */

typedef int Bool;
typedef unsigned BitField;

struct foo
    {
    unsigned a:1, b:2;
    BitField c:1, d:2, e:3;
    } x;

void f()
    {
    x.a = (Bool) 1;     /* OK */
    x.b = (Bool) 0;     /* strong type violation */
    x.a = 0;            /* strong type violation */
    x.b = 2;            /* OK */
    x.c = x.a;          /* OK */
    x.e = 1;            /* strong type violation */
    x.e = x.d;          /* OK */
    }
```

In the above, members `a` and `c` are strongly typed `Bool`, members `d` and `e` are typed `BitField` and member `b` is not strongly typed.

To suppress the Boolean assumption for one-bit bit-fields use the flag `f` in the `-strong` option for the Boolean. In the example above, this would be `-strong(AJXbf,Bool)`.

The `-index` Option

Description of `-index`

The FlexeLint syntax for this option is

`-index(` [*flags*]`,` *ixtype*`,` *sitype* [`,` *sitype*] ... `)`

This option is supplementary to, and can be used in conjunction with, the `-strong` option. It specifies that *ixtype* is the exclusive index type to be used with arrays of (or pointers to) the Strongly Indexed type *sitype* (or *sitype*s if more than one is provided). Both the *ixtype* and the *sitype* are assumed to be names of types subsequently defined by a `typedef` declaration. *flags* can be

`c` allow Constants as well as *ixtype*, to be used as indices.

`d` allow array Dimensions to be specified without using an *ixtype*.

Examples of `-index`

For example:

```
/*lint -index(d,Count,Temperature)
        Only Count can index a Temperature */

typedef float Temperature;
typedef int Count;
Temperature t[100];  /* OK because of d flag */
Temperature *pt = t; /* pointers are also checked */

    /* ... within a function  */
    Count i;

    t[0] = t[1];       /* Warnings, no c flag */
    for( i = 0; i < 100; i++ )
        t[i] = 0.0;   /* OK, i is a Count */
    pt[1] = 2.0;      /* Warning */
    i = pt - t;       /* OK, pt-t is a Count */
```

In the above, `Temperature` is said to be *strongly indexed* and `Count` is said to be a *strong index*.

If the `d` flag were not provided, then the array dimension should be cast to the proper type as in the following:

```
Temperature t[ (Count) 100 ];
```

However, this is a little cumbersome. It is better to define the array dimension in terms of a manifest constant, as in:

```
#define MAX_T  (Count) 100
Temperature t[MAX_T];
```

This has the advantage that the same `MAX_T` can be used in the `for` statement to govern the range of the `for`.

Note that pointers to the Strongly Indexed type (such as `pt` above) are also checked when used in array notation. Indeed, whenever a value is added to a pointer that is pointing to a strongly indexed type, the value added is checked to make sure that it has the proper strong index.

Moreover, when strongly indexed pointers are subtracted, the resulting type is considered to be the common Strong Index. Thus, in the example,

```
i = pt - t;
```

no warning resulted.

It is common to have parallel arrays—arrays with identical dimensions but different types—processed with similar indices. The `-index` option is set up to support this conveniently. For example, if `Pressure` and `Voltage` were types of arrays similar to the array `t` of `Temperature` one might write:

```
/*lint -index(,Count,Temperature,Pressure,Voltage)*/

Temperature t[MAX_T];
Pressure p[MAX_T];
Voltage v[MAX_T];
```

Multidimensional Arrays

The indices into multidimensional arrays can also be checked. Just make sure the intermediate type is an explicit `typedef` type. An example is `Row` in the code below:

```
/*  Types to define and access a 25x80 Screen.
    a Screen is 25 Row's
    a Row is 80 Att_Char's */

/*lint -index(d,Row_Ix,Row)
       -index(d,Col_Ix,Att_Char) */

typedef unsigned short Att_Char;
typedef Att_Char Row[80];
typedef Row Screen[25];
typedef int Row_Ix;     /* Row Index */
typedef int Col_Ix;     /* Column Index */

#define BLANK (Att_Char) (0x700 + ' ')

Screen scr;
Row_Ix row;
Col_Ix col;

void main()
    {
    int i = 0;

    scr[row][col] = BLANK;      /* OK */
    scr[ i ][col] = BLANK;      /* Warning */
    scr[col][row] = BLANK;      /* Two Warnings */
    }
```

In the above, we have defined a `Screen` to be an array of `Rows`. Using an intermediate type does *not* change the configuration of the array in memory. Other than for type checking, it is the same as if we had written:

```
typedef Att_Char Screen[25][80];
```

Type Hierarchies

A discovery that was made only after the first version of strong typing was implemented was that some sort of type hierarchy was needed.

The Need for a Type Hierarchy

Consider a *Flags* type which supports the setting and testing of individual bits within a word. An application might need several different such types. For example, one might write:

```
typedef unsigned Flags1;
typedef unsigned Flags2;
typedef unsigned Flags3;

#define A_FLAG (Flags1) 1
#define B_FLAG (Flags2) 1
#define C_FLAG (Flags3) 1
```

Then, with strong typing, an `A_FLAG` can be used with only a `Flags1` type, a `B_FLAG` can be used with only a `Flags2` type, and a `C_FLAG` can be used with only a `Flags3` type. This, of course, is just an example. Normally there would be many more constants of each *Flags* type.

What frequently happens, however, is that some generic routines exist to deal with *Flags* in general. For example, you may have a stack facility that will contain routines to push and pop *Flags*. Or you might have a routine to print *Flags* (given some table that is provided as an argument to give string descriptions of individual bits).

Although you could cast the *Flags* types to and from another more generic type, the practice is not to be recommended, except as a last resort. Not only is a cast unsightly, it is hazardous since it suspends type-checking completely.

The Natural Type Hierarchy

The solution is to use a type hierarchy. Define a generic type called `Flags` and define all the other *Flags* in terms of it:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef Flags Flags3;
```

In this case `Flags1` can be combined freely with `Flags`, but not with `Flags2` or with `Flags3`.

This depends, however, on the state of the `fhs` (Hierarchy of Strong types) flag which is normally ON. If you turn it off with the `-fhs` option, the natural hierarchy is not formed.

We say that `Flags` is a *parent* type to each of `Flags1`, `Flags2` and `Flags3` which are its *children*. Being a parent to a child type is similar to being a base type to a derived type in an object-oriented system with one very important difference. A parent is interchangeable with each of its children; a parent can be assigned to a child and a child can be assigned to a parent. But a base type is a subset of a derived type and assignment can go only one way.

A generic *Flags* type can be useful for all sorts of things, such as a generic zero value, as the following example shows:

```
/*lint -strong(AJX) */
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
#define FZERO (Flags) 0
#define F_ONE (Flags) 1

void m()
    {
    Flags1 f1 = FZERO;     /*  OK  */
    Flags2 f2;

    f2 = f1;               /* Warn */
    if(f1 & f2)            /* Warn */
        f2 = f2 | F_ONE;   /*  OK  */
    f2 = F_ONE | f2;       /*  OK  */
    f2 = F_ONE | f1;       /* Warn */
    }
```

Note that the type of a binary operator is the type of the most restrictive type of the type hierarchy (i.e., the child rather than the parent). Thus, in the last example above, when a `Flags` ORs with a `Flags1` the result is a `Flags1` which clashes with the `Flags2`.

Type hierarchies can be an arbitrary number of levels deep.

There is evidence that type hierarchies are being built by programmers even in the absence of strong type-checking. For example, the header for Microsoft's Windows development kit, `windows.h`, contains the following extract:

```
typedef unsigned int        WORD;
typedef WORD                ATOM;
typedef WORD                HANDLE;
typedef HANDLE              HWND;
typedef HANDLE              GLOBALHANDLE;
typedef HANDLE              LOCALHANDLE;
typedef HANDLE              HSTR;
typedef HANDLE              HICON;
typedef HANDLE              HDC;
typedef HANDLE              HMENU;
typedef HANDLE              HPEN;
typedef HANDLE              HFONT;
typedef HANDLE              HBRUSH;
typedef HANDLE              HBITMAP;
typedef HANDLE              HCURSOR;
typedef HANDLE              HRGN;
typedef HANDLE              HPALETTE;
```

Adding to the Natural Hierarchy

The strong type hierarchy tree that is naturally constructed via `typedef`s has a limitation. All the types in a single tree must be the same underlying type. The `-parent` option can be used to supplement (or completely replace) the strong type hierarchy established via `typedef`s.

An option of the form:

`-parent(` *Parent*`,` *Child* [`,` *Child*] ... `)`

where *Parent* and *Child* are type names defined via `typedef`, will create a link in the strong type hierarchy between the *Parent* and each of the *Child* types. The *Parent* is considered to be equivalent to each *Child* for the purpose of Strong type matching. The types need not be the same underlying type and normal checking between the types is unchanged. A link that would form a loop in the tree is not be permitted.

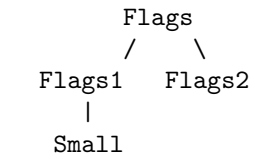
Given the options:

```
-parent(Flags1,Small)
-strong(AJX)
```

and the following code:

```
typedef unsigned Flags;
typedef Flags Flags1;
typedef Flags Flags2;
typedef unsigned char Small;
```

then the following type hierarchy is established:



If an object of type `Small` is assigned to a variable of type `Flags1` or `Flags`, no strong type violation will be reported. Conversely, if an object of type `Flags` or `Flags1` is assigned to type `Small`, no strong type violation will be reported but a loss of precision message will still be issued (unless otherwise inhibited) because normal type checking is not suspended.

A visual picture of the hierarchy tree can be obtained using the letter `h` in connection with the `-v` option.

If the `-fhs` option is set (turning off the hierarchy of strong types flag) `typedef`s will not add hierarchical links. The only links that will be formed will be via the `-parent` option.

Hints on Strong Typing

1. Beware of excessive casting. If, in order to pull off a system of strong typing you need to cast just about every access, you are missing the point. The casts will inhibit even ordinary checking, which has considerable value. Remember, strong type-checking is gold, normal type-checking is silver, and casting is brass.

2. Rather than cast, use type hierarchies. For example:

```
/*lint -strong(AXJ,Tight) -strong(,Loose) */
typedef int Tight;
typedef Tight Loose;
```

   `Tight` has a maximal amount of Strong Type checking; `Loose` has none. Since `Loose` is defined in terms of `Tight` the two types are interchangeable from the standpoint of Strong Type checking. Presumably you work with `Tight int`s most of the time. `Loose` is used when absolutely necessary to achieve some effect.

3. A time when it is really good to cast is to endow some otherwise neutral constant with a special type. `FZERO` of the previous section is an example.

4. For large, mature projects, add strong typing slowly, working on one family of strong types at a time. A family of strong types is one hierarchy structure.

5. Don't bother making pointers to functions strong types. For example:

```
typedef  int (*Func_Ptr)(void);
```

   If you make `Func_Ptr` strong, you're not likely to get much more checking than if you didn't make it strong. The problem is that you would then have to cast any existing function name when assigning to such a pointer. This represents a net loss of type-checking (remember: gold, silver, brass).

6. Rather than making a pointer a strong type, make the base type a strong type. For example:

```
typedef char TEXT;
typedef TEXT *STRING;

TEXT buffer[100];
STRING s;
```

   It may seem wise to strong type both `STRING` and `TEXT`. This would be a mistake since whenever you assign `buffer` to `s`, for example, you would

have to cast. But note that `-strong(Ac, STRING)` would allow the assignment. It is usually better to strong type just `TEXT`. Then when `buffer` is assigned to `s` the indirect object `TEXT` is strongly checked and no cast is needed.

This holds for structures as well as for scalars. For example, in MS Windows programming there are a number of `typedef`ed types that are pointers. Examples include: `LPRECT`, `LPLOGFONT`, `LPMSG`, `LPSTR`, `LPWNDCLASS`, etc. If you make these `-strong(A)` you will have problems passing to Windows functions addresses of Window's `struct`s. At most make them `-strong(AcX)`.

7. Care is needed in declaring strong self-referential `struct`s. The usual method, i.e.,

```
typedef struct list { struct list * next ; ... }
    LIST;
```

is incompatible with making `LIST` a strong type because its member `next` will not be a pointer to a strong type. It is better to use

```
typedef struct list LIST;
struct list { LIST * next; ... };
```

This is explicitly sanctioned in ANSI C (§3.5.2.3) and will make `next` compatible with other pointers to `LIST`.

Reference Information

Strong Expressions

An expression is strongly typed if it is

1. a strongly typed variable, function, array, or member of `union` or `struct` or an indirectly referenced pointer to a strong type.
2. a cast to some strong type.
3. one of the *type-propagating* unary operators (such as `+`, `-`, `++`, `--`, and `~`) applied to a strongly typed expression.
4. formed by one of the *balance and propagate* binary operators applied to two strongly typed expressions (having the same strong type). The balance and propagate operators consist of `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`), and `?:`.
5. a shift operator whose left side is a strong type.

6. a comma operator whose right side is a strong type.
7. an assignment operator whose left side is a strong type.
8. a *Boolean operator* and some type has been designated as Boolean (with a `b` or `B` flag in the `-strong` option). The Boolean operators consist of the relational, equality, and logical operators.

General Information

When the option

`-strong(` [*flags*] [`,` *name*] ... `)`

is processed, *name* and *flags* are entered into a so-called Strong Table created for this purpose.

If there is no *name*, then a variable, Default Flags, is set to the *flags* provided. When a subsequent `typedef` is encountered within the code, the Strong Table is consulted first. If the `typedef` name is not found, the Default Flags are used. These flags become the identifying flags for strong typing purposes for the type.

The option

`-index(` [*flags*]`,` *ixtype*`,` *sitype* [`,` ...] `)`

is treated similarly. Each *sitype* is entered into the Strong Table (if not already there) and its index flags ORed with other strong flags in the table. A pointer is established from *sitype* to *ixtype* which is another entry in the Strong Table.

For these reasons it does not, in general, matter in what order the `-strong` options are placed other than that they be placed before the associated `typedef`. There should be at most one option that specifies Default Flags.

Strong Types and Prototypes

If you are producing prototypes with some variation of the `-od` option (Output Declarations), and if you want to see the `typedef` types rather than the raw types, just make sure that the relevant `typedef` types are strong. You can make them all strong with a single option: `-strong()`. Since you have not specified `A`, `J`, or `X` you will not receive messages owing to strong type mismatches for Assigning, Joining or eXtraction. However, you may get them for declarations. Another option

```
-etd(strong)
```

is available to inhibit any such messages.

Final Comments

The hierarchy of strong types compels one to compare this hierarchy with the object-oriented hierarchy of C++ and other languages. If one closely examines the phrase "code reusability" touted as a property of OOP, one finds that it refers to the fact that one can write generic functions (i.e., functions that operate on more than just one `struct` type). The way that one can do this in C is by passing pointers to a function expecting a `void` pointer and hoping that all `struct`s passed this way are compatible. With strict type checking (without hierarchies) you cannot do it at all. So perhaps it is now a little clearer why code reusability is such a puzzle to C programmers. With a loosely typed language such as C, code reusability was something you did not have to work very hard for. In the past, when C code would mix pointers and `int`s freely, code reusability had always been a fact of life. Coming from Pascal or Ada, however, OOP really does provide for the writing of generic functions. One may speculate that C's general success over Pascal may be attributed in part to its greater code reusability. For C, OOP provides not reuse but type checking.

Viewed in this way the strong type hierarchies described in this paper serve the same purpose for scalars as the OOP hierarchies do for `struct`s. Indeed, it is possible to advantageously mimic an OOP structure hierarchy. For each `struct` one could have an associated `void` pointer and arrange these in a hierarchy. For example:

```
typedef void    *vShape;
typedef vShape  vCircle;
typedef vShape  vSquare;
```

With these as strong types a server routine accepting `vShape`s as arguments could accept `vCircle`s and `vShape`s as well. The strong type facility would check types and keep them in line. Many of the benefits of object-orientedness would result with one additional bonus. The server routines alone would know or care about the internal structure of a Shape, Circle or Square. This would also reduce the number of headers needing to be included.

Jim Gimpel is founder and President of Gimpel Software, a firm specializing in tools for C programmers. He has been the leading architect in the development of C-terp, BASTOC, PC-lint, FlexeLint, and the C Shroud. He can be reached at (215) 584-4261.

∞

29. Electronic Survey Number 11

Compiled by **Rex Jaeschke**

Introduction

Occasionally, I conduct polls via electronic mail and publish the results. (Those polled also receive an E-mail report on the results.)

The following questions were posed to about 80 people, with 17 of them responding. Since some vendors support more than one implementation, the totals in some categories may exceed the number of respondents. Also, some respondents did not answer all questions, or deemed them 'not applicable.' I have attempted to eliminate redundancy in the answers by grouping like responses. Some of the more interesting or different comments have been retained.

Diagnostic/Message Levels

What levels of user diagnostics/messages do you support and what level is on by default? Can the programmer select levels?

- 4 – Compiler fatal
- 16 – User error
- 11 – User warning
- 8 – Comments/information
- 5 – Extension
- Comments:

    1. The user can select the warning message level using the command-line option `-Wnum`.
        - If `num` is 0, no warning messages are issued.
        - If `num` is 1, most warning messages are issued (this is the default value).
        - If `num` is 2, more warning messages are issued, including use of functions with no declared return type, failure to put return statements in functions with non-void return types, and data conversions that would cause loss of data or precision.

– If `num` is 3, all of the above warning messages are issued, along with warnings about any non-ANSI features and extended keywords and function calls before the appearance of function prototypes in the program.

This option does not affect compiler error message output.

The option `-WX` causes all warnings to be fatal. It can be used in combination with `-W0`, `-W1`, `-W2`, or `-W3`.

2. We have four levels of user-selectable message generation:

   0 – undefined behavior, warning (default)
   1 – syntax or constraint error, recoverable
   2 – syntax or constraint error, fatal
   3 – implementation limit, fatal

   The user can independently choose the warning level and the "fatal" level (so they could abort on a level 0 warning, if they want).

   I should point out that our compiler does not generate comments and cautions for code such as `if (a = b)` or the statement `a == b + c;`. Our compilers do provide command line switches to relax the ANSI rules for various types of declarations, and for K&R compatibility. These options tend to change the warnings issued.

3. Just one level. This is an interactive environment in which the first error ends compilation.

4. There are three levels of warning which may be selected with a command line option:

   0 – errors only
   1 – add regular warnings (default)
   2 – add warnings about old-fashioned (pre-ANSI) stuff
   3 – add portability warnings (extensions)

5. We have had requests for selectively deleting particular warnings.

6. We have three major levels: error, warning, compiler error. Within warnings we distinguish between "usage" and "conformance." The former category is used to indicate legal constructs that are in some way dubious. The later are used to indicate illegal constructs that we accept. By default all messages are issued. The user can selectively disable all warnings or only one of the subcategories. We also have several categories we use internally that get mapped to one of the above (e.g., a "not implemented" becomes an error) with a slightly cryptic indication in the message. We have several different compilation modes (K&R, Strict ANSI, Extended ANSI, Strict C++, Cfront compatible C++) and many messages are mapped differently depending on the mode. For example, many conformance warnings are only issued in the Strict modes.

7. There are ten levels. There can be multiple levels associated with each error. The default is to select the lowest level. This behavior can be changed to select the highest level, or even cut off levels below a certain point.
8. I separately allow for selective enable/disable on each distinct message the compiler issues.
9. Diagnostics come in three flavors: internal, error, and warning. Messages of the first type are not supposed to occur. The second eventually causes the compile to fail. The third type does not cause a compile failure, no matter how many are issued. Certain warnings are not always enabled. Extension versus other is covered by the text of the message. In my opinion, a compiler's job is to do its best to compile the program, not to nag about code that's borderline.

Array Bounds Checking

Do you support any form of execution-time bounds checking for array references? Pointer dereferences?

- 1 – Yes
- 15 – No
- Comments:

1. Not currently but there are hooks for the capability, however, at least for storage whose array bounds are known.
2. Not currently. We had plans for such a feature but haven't implemented anything yet.
3. Yes for both arrays and pointers. It is very hard to do the job properly.
4. Comprehensive bounds checking presents an interface compatibility problem for pre-compiled libraries. Carrying dynamic bounds information with each stored pointer means the libraries have to know about the bounds data.
5. No. Too costly. Moreover, what could be done when a free-standing program violates a bound?

sig_atomic_t

Is the following code fragment standard-conforming?

```
#include <signal.h>

volatile sig_atomic_t var;
```

Specifically, can an implementor include the qualifier `volatile` *in the typedef for* `sig_atomic_t`*? Must an implementor include the qualifier* `volatile` *in the typedef for* `sig_atomic_t`*? If the typedef includes it the programmer can't also use it since that would result in a constraint violation. If it is not included, the programmer needs to use it.*

§4.7 (pp 121 line 6) of the ANSI C Standard says that `sig_atomic_t` *"is the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts." The Rationale document §2.2.3 (pp 17–18) seems to imply that* `sig_atomic_t` *is* `volatile`*-qualified. However, the standard in §4.7.1.1 (pp 122 line 15–16) talks about a variable of type* `volatile sig_atomic_t`*.*

- 13 – The Standard is quite clear—The user must supply the `volatile` and the implementor should not.
- 2 – Confused.
- Comments:

  1. The standard says that `sig_atomic_t` is a type, not a qualified type. Similarly, a misguided implementer can include `volatile` in the `size_t` typedef! Nothing in the C Standard prohibits misusing `volatile` in this way.
  2. I agree this definition is weak. What comes to mind is processors which have special instructions to lock the bus for access to *any* size type, and a compiler which uses locking instructions for access to any `volatile` object. In this environment, `sig_atomic_t` really doesn't have any meaning. However I don't think that this is the intent of either `volatile` or `sig_atomic_t`. I read `sig_atomic_t` as the type which the processor can access with a single bus access (excluding multi-bus-access types).
  3. In my opinion, §4.7.1.1 is definitive. §4.7 could have been worded more clearly, but does not necessarily contradict it. The Rationale *is not* part of the Standard.
  4. My reading is that the standard doesn't say one way or another whether `sig_atomic_t` is `volatile`. So the above is conforming, but not strictly conforming.

5. Regarding "Can an implementor include the qualifier `volatile` in the typedef for `sig_atomic_t`," the standard gives requirements that must be met. It does *not* define implementation.
6. Unless the Standard specifies that a given type is qualified, it is not. Qualifying a typedef is a poor practice anyway; it just restricts the ways you can declare objects. I think the text in §4.7.1.1 makes it very clear that `sig_atomic_t` is not `volatile`-qualified. Otherwise it would be suggesting a constraint-violation as the only way to accomplish something.
7. The Rationale seems to be misleading. When the Standard is read alone, there is no doubt. The type `sig_atomic_t` is an atomically accessed integer type. Whether to qualify this type by `volatile` must be the choice of the programmer. The implementation should not prevent it. (Although an implementation might add extra semantics behind the application's back for `sig_atomic_t` objects.) The qualification by `volatile` of a `sig_atomic_t` object is only appropriate if it is going to be modified outside of the program's normal execution path.

Translation Limits

How do you handle the translation limits of §2.2.4.1? Do you special-case each one, use a table lookup for checking and diagnostics, or what?

- 9 – Special case.
- Comments:

  1. Everything is allocated dynamically.
  2. We special-case each one, issue a warning when the limit specified in the standard is reached (if in standard conformance mode), and then avoid any kind of limit, or increase the limit significantly over the standard limit.

Type-Safe Linkage Support

Do you support type-safe linking (ala C++ or similar) across translation units?

- 1 – Yes
- 14 – No
- Comments:

  1. No, but something along this line is in our future.

2. Our programming environment provides messages about conflicts, but the compiler proper doesn't detect this.
3. Yes. We also do "quick" type checking via a 32-bit checksum.

Bit-Field Types

What types can be used in bit-field declarations beyond `int`*,* `signed int`*, and* `unsigned int`*?*

- 4 – Just the standard ones
- 1 – Standard ones plus `short`
- 3 – Standard ones plus `char`, `short`, and `int`
- 6 – Any integer type (including `enum`s)
- 1 – Other
- Comments:

1. Any non-floating scalar type (including pointers), and structure types whose fields are *all* bit-fields themselves and whose size is less than `sizeof(int)`.
2. The real question is how the different base types affect the way that bit-fields are allocated (and whether they are even accepted). Our implementation disallows bit-field widths that are longer than their base size. If there are enough bits left in an appropriately aligned object of the base size to permit the bit-field to fit, the space is allocated. Otherwise, the entire bit-field is allocated from the next appropriately aligned object of the base type. Whether the allocation is low-to-high or high-to-low depends on the target machine.

∞

30. Massively Parallel C: Data-Parallel Execution

Tom MacDonald
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

Abstract

Data-parallel execution permits operators to manipulate data-parallel objects. Data-parallel execution is a natural programming style on many massively parallel platforms. In many languages these parallel data objects are arrays. The fact that Standard C does not support arrays as first-class objects has motivated some vendors to invent new object types that are similar to arrays but are first-class objects. First-class arrays introduce many other features into the language.

Introduction

Numerous proposals have been submitted to the Numerical C Extensions Group (NCEG) that introduce data-parallel execution into Standard C. The following is a partial list of features present in various proposals:

- Fortran-90 style of array syntax, `a[1:n:2] = b[:]  + c[:];`
- functional notation, `VecAsgn(a,1,n,2,VecAdd(b,m,n,1,c));`
- vector statements, `vector (n/2) a[2*^] = b[^] + c[^];`
- selector index, `at[i:;][j:;] = a[j][i]; /* transpose */`
- circular pointers, `char * circ cp;`
- masked element selection, `object[$circle] = blue;`
- parallel objects, `shape [2][32768] twoD; int:twoD parobj;`
- a bit type

Although the number of widely-varying proposals has made it difficult to converge on a single proposal, it does show that there is considerable interest in such a feature. Each champion of a particular proposal has his own prime motivation, but there is a common set of goals. One such goal is to provide

additional expressiveness. Many numerical applications would benefit from the ability to express vector algorithms in a style that is closer to the mathematical notation. Another common goal is to provide constructs that map more easily onto state-of-the-art parallel hardware. Vector-oriented features appear to offer a realistic solution for both the programmer and the implementor. Since there are many platforms that do not support parallel execution, it is important to ensure that any new data-parallel features also map onto existing serial environments.

The following is a look at the issues involved with adding data-parallel semantics to C. Two proposals are contrasted: C*[1] and what is referred to as the `[:]` (pronounced "square bracket colon") approach. Comparing these two proposals will expose the issues that the NCEG subcommittee struggle with at each meeting. An examination of the issues helps with the process of deciding which features are most useful, which features are flawed, and which features are too complicated.

The Trouble with C Arrays

The need for some new syntax becomes apparent when the behavior of C arrays is compared to Fortran-90 arrays. Consider the following Fortran-90 array syntax example:

```
SUBROUTINE ADD_ARRAYS()
COMMON /ARRAYS/ A(1000), B(1000), C(1000)

A = B + C
RETURN
END
```

and compare it to the following, seemingly identical, C example (assume some analogous array syntax exists in C):

```
float a[1000], b[1000], c[1000];
void add_arrays() {
   a = b + c;
}
```

The problems associated with extending Standard C arrays such that they behave like Fortran-90 arrays become apparent. The majority of the problems are there because C arrays are not first-class objects. §3.2.2.1 of the ANSI C Standard states:

> "Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of

[1] C* is a registered trademark of Thinking Machines Corporation.

> the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)."

However, for arrays the Standard states:

> "Except when it is the operand of the `sizeof` operator or the unary `&` operator, ... an lvalue that has type 'array of *type*' is converted to an expression that has type 'pointer to *type*' that points to the initial element of the array object and is not an lvalue."

New syntax is needed because `a` can never be assigned to, and arrays are converted into pointer expressions that point to the first array element. There is no palatable upward-compatible way to turn C arrays into first class objects without adding new syntax.

The [:] Approach

One proposal introduces syntax that overloads the subscript operator. The idea is that the token sequences `[:]` and `[::]` are used to designate an *array section* that describes a mapping of a previously allocated set of array elements. The array syntax example becomes:

```
float a[1000], b[1000], c[1000];
void add_arrays() {
   a[:] = b[:] + c[:];
}
```

and the `[:]` syntax is used to describe a mapping that maps corresponding elements of entire arrays onto data-parallel operations.

Another array syntax variation uses `[;]` in place of `[:]` with slightly different semantics, but essentially equivalent functionality. (This approach is one that I pursued in one of my proposals.) The primary difference between the two approaches is the array subsection semantics. Both approaches permit array subsections, but the `[:]` approach uses *initial index* and *final index* specifiers, while the `[;]` approach uses *initial index* and *length* specifiers. There are other proposals that recommend using both. With the `[;]` approach the following mappings occur:

```
double a[12];
a[0 ;12]        /* specifies the entire array     */
a[0 ; 6 ; 2]    /* the even elements of the array */
a[11; 4 ;-3]    /* every 3rd element starting with
                   last element going backwards   */
```

and with the `[:]` approach the following mappings occur:

```
double a[12];
a[0 : 11]       /* specifies the entire array     */
a[0 : 11 : 2]   /* the even elements of the array */
a[11:  0 :-3]   /* every 3rd element starting with
                   last element going backwards   */
```

There are advantages and disadvantages to both approaches. The `[:]` syntax is used throughout the rest of this document. The following example shows how sub-sectioning can be used to compute the sum of an array of integers in O(log n) time.

```
#include <stdio.h>

#define M 8
#define N (1<<M)  /* 2**M */

extern int a[N];

void sum_array() {
   int i, j, k;

   for (i = 1; i <= M; i++) {
      k = (1 << i);                /* 2**i     */
      j = k - 1;                   /* 2**i - 1 */
      a[j:N-1:k] += a[j-k/2:N-1:k];
   }
   printf("sum = %d\n", a[N-1]);   /* last element */
                                   /* holds the sum */
}
```

For simplicity, this algorithm assumes that the number of elements in array `a` is an integral power of two.

The `[:]` approach permits a wide variety of data-parallel expressions, including vector valued expressions such as `a[ix[:]]`. However, it does not introduce arrays as first-class objects but, rather, introduces a way of mapping into arrays. That is, we still must use something artificial like

`a[] = b[] + c[];` instead of the more intuitive `a = b + c;`

when whole array references are desired. There is no way to declare a function that accepts whole array arguments or return whole array values. Many important functions—such as math functions like `sin`, `cos`, `sqrt`, and `pow`—cannot be used with this array syntax proposal. Finally, many useful vector operations were never defined. These include sum reduction, transpose, matrix multiply, max, min, and shifting operations. A complete array syntax feature would make C a significantly larger and more complicated language.

C* and Parallel Variables

C* introduces the concept of parallel objects into C (along with many other features). Parallel objects share many of the same characteristics with C arrays, but there is one major difference—they are never implicitly converted to a pointer to the first element. Therefore, operations such as storing the address of the first element in a pointer, and incrementing that pointer a given number of times, do not produce predictable results. There is no way to use a pointer to walk through the elements of a parallel object in a predictable fashion. This is because the parallel object can be distributed across a network.

C* contains the keyword `shape` that specifies the shape of a parallel variable. A shape must be declared before a parallel variable of that shape can be declared.

```
shape [256][256]A_shape;
int:A_shape p1, p2, p3;
```

This example declares three parallel variables `p1`, `p2`, and `p3`, each of which has two dimensions (rank 2), with 256 elements in each dimension (their extents). Before parallel operations can be applied to parallel objects, the correct shape must be selected by using the `with` statement. In general, parallel variables referenced inside a `with` block must have the same shape as the *shape-specifier* on the `with` statement. The following example demonstrates the usage of the `with` statement.

```
sum = 0;

with (A_shape) {
/*1*/    p1 = 1;
/*2*/    p2 = 2;
/*3*/    p3 = p1 + p2;
/*4*/    printf("[1][2]p3 = %d\n", [1][2]p3);
/*5*/    sum += p3;
}
```

Statement 1 assigns the value `1` to each element of parallel variable `p1`. Statement 2 assigns the value `2` to each element of parallel variable `p2`. Statement 3 adds elements of `p1` to corresponding elements of `p2` and stores the sums in corresponding elements of `p3`. Statement 4 uses the left-indexing feature to reference individual elements of parallel variables. An individual element such as `[1][2]p3` can be referenced by placing the subscript operators on left side of the parallel variable. From the programmer's point of view, this is conceptually identical to referencing an array element such as: `a[1][2]`. Statement 5 uses the `+=` operator as an *assignment reduction* operator. That is, all the elements of parallel variable `p3` are added together and that result is added to the value of the scalar variable `sum`, and the result is stored in `sum`.

C* supports declarations of arrays of parallel objects also. In the following example:

```
int:A_shape apv[8];
```

`apv` is an array of parallel objects in that each element of `apv` is a parallel object. Parallel elements can be referenced in expressions such as:

```
apv[1] = apv[2] * apv[3];     /* parallel operations */
```

and scalar elements can be referenced in expressions such as:

```
[i][j]apv[1] = [i][j]apv[2];  /* scalar operations */
```

Parallel objects can be thought of as having one or more parallel dimensions and arrays of parallel objects have both serial and parallel dimensions. The programmer can tell which dimensions are intended to execute in parallel and which must execute serially by looking at the declaration.

Active and Inactive Elements

The keyword `where` is used to control which elements in a parallel variable are operated on (active), and which ones are left alone (inactive).

```
with (A_shape) {
   where (p3 != 0)
      p1 = p2 / p3;
   else
      p1 = 0;
}
```

There is also an `everywhere` statement that overrides the current active set of a `where` statement and makes all elements active. It is possible to loop until no more elements are active by using the unary `|=` reduction operator as follows:

```
with (A_shape) {
   while (|= (p1 > 0))
      where (p1 > 0)
         --p1;
}
```

Another way of controlling which elements are active is by using the `||`, `&&`, and `?:` operators. The following statement:

```
p3 = (p1 > 5) && (p2++)
```

is equivalent to:

```
where (p1 > 5)
   p2++
```

When these operators have parallel variables as operands they are not true short circuit operators. Instead one operand controls access to corresponding elements.

Interprocessor Communication

Since C* was developed to run on a distributed memory machine, serial and parallel objects also describe data layout. Parallel objects are distributed and serial objects are not. Since references to parallel variables must appear inside a `with` statement, interprocessor communication can be kept to a minimum. However, vector-valued subscript expressions do cause interprocessor communication to occur. A "get" operation with parallel variables is accomplished by a statement like the following:

```
target = [index]source;
```

and a send operation by:

```
[index]target = source;
```

These are called *parallel left index* operations. Another use of the parallel left index operations is to reshape a parallel variable. That is, if parallel variable `index` is two dimensional, then the result of left indexing a one-dimensional parallel variable with `index` is a two-dimensional parallel value.

```
shape [64][64]A_shape;
int:A_shape p4, p5;
shape [16384]B_shape;
int:B_shape p6;

with (A_shape)
   p5 = [p4]p6;
```

The expression `[p4]p6` conforms to the shape `A_shape` that is the *shape-specifier* in the controlling `with` statement. The parallel variable `p4` reshapes parallel variable `p6` into the shape of `p4`.

Functions

C* allows parallel variables to be passed as arguments and returned from functions as first-class objects. Finally, C* permits user-defined function overloading. This allows a function with the same name to operate on scalar and parallel data. The keyword `overload` specifies names of functions to be overloaded.

Evaluation of C*

C* fulfills the goal of adding a first-class data-parallel object to C in a largely upward-compatible way. C* supports first-class data-parallel objects by overloading existing C operators, introducing reduction operators that are spelled the same as the unary assignment operators, and by adding some new operators like max `>?=` and min `<?=`. User-defined overloaded functions allow parallel and serial functions to have the same interface. It is certainly possible to write highly parallel applications with C*. No language is perfect and this is true with C* also. There are a number of issues that need to be raised.

C* does not permit array subsections to be specified. A statement such as:

```
[0:n:2]p1 = [1:n:2]p2;
```

is not accommodated. There are applications that are easier to write if *ghost* borders are declared around the edges of the parallel object. Access to the actual object is easily specified by:

```
x[1:N-2][1:N-2]
```

if array subsections are permitted.

C++ would have a difficult time adding left indexing because there appears to be no straightforward way to add operator overloading. It would be too easily confused with right indexing.

The left-indexing notation allows for easy recognition of parallel dimensions but makes it difficult to interchange parallel dimensions with serial dimensions. Interchanging a parallel dimension and serial dimension could force the programmer to manually change all `[i][j]apv[k]` expressions to another form such as `[i][k]apv[j]`. There is some loss of type richness in that it is possible to declare an array of parallel objects but it is not possible to declare a parallel variable whose elements are arrays.

One of the goals for an array syntax was to increase its expressiveness by allowing vector constructs. C* limits vector expressiveness to parallel dimensions. From a pure expressiveness point of view, it seems desirable to be able to use data-parallel syntax on any dimension—serial or parallel.

The `with` statement is too limiting. Since data-parallel operations can only occur inside a `with` statement, there is no way to call a function with two

arguments of different shapes. It is quite convenient to be able to have array arguments that have different rank and extents.

The sequence needed to loop until every element reaches some condition seems awkward.

```
with (A_shape) {
   while (|= (p1 > 0))
      where (p1 > 0)
         --p1;
}
```

It seems unnecessary to have to specify the condition `(p1 > 0)` twice. The natural way to code this is:

```
while (p1 > 0)
  --p1;
```

which seems more understandable and straightforward.

Finally, the new reduction operators do not behave the same as the traditional C operators with the same spelling. For example:

```
s1 -= p2;
```

is equivalent to:

```
s1 = s1 - (+= p2);
```

instead of the more traditional:

```
s1 = s1 - p2;
```

Finally, C* adds too many new keywords. (Some are: `with`, `where`, `shape`, `shapeof`, `rankof`, `positionsof`, `physical`, `overload`, `everywhere`, `current`, `bool`, and `boolsizeof`.)

Although, there are issues raised with C*, it is a complete language that addresses many of the data-parallel issues that NCEG is trying to address. I doubt that C* could become a future standard in its present form even though it contains many good ideas.

Conclusions

Data-parallel languages like Fortran-90 and C* permit programmers to declare data-parallel objects. In Fortran-90, arrays can be used as data parallel objects while C* adds an additional feature to Standard C called parallel variables. Both Fortran-90 and C* are huge languages when compared to C. When I

look at the number of features that need to be included to fully support array syntax, I question the wisdom of putting such features in C. The biggest issue is user-defined function overloading. Is the language still C if it contains this feature or has it moved too far into the domain of C++?

Another issue is passing array arguments by value to functions and returning array values from functions. Such a feature encourages additional features like the *reference* parameter feature of C++.

Should some of the standard library functions like `sin` and `cos` be turned into generic intrinsic functions that accept arbitrary array arguments and return arbitrary array results? This is far more palatable than user-defined overloaded functions but needs close examination.

It is difficult to turn away from array syntax because so many have expressed a desire for such a feature. Data-parallel execution on massively parallel machines offers a natural programming style. The reason the NCEG array syntax subcommittee is struggling with this feature is because their goal of providing a first-class array object in C carries a great deal of baggage along with it. I doubt a feature that extensive can be successfully integrated into C. Perhaps a new language is needed or we need to compromise on the extent of the feature.

Tom MacDonald is the Numerical Editor of The Journal of C Language Translation. He is the Cray Research Inc representative to X3J11 and a major contributor to the floating-point enhancements made by the ANSI C standard. He specializes in the areas of floating-point, vector, array, and parallel processing with C language and can be reached at (612) 683-5818, tam@cray.com, *or* uunet!cray!tam.

∞

31. Pragmania

Rex Jaeschke

You can include the following pragmas within a source file. They may not be used within a function, however. A pragma has effect from the point at which it is specified to the end of the translation unit (or until another pragma changes its status).

Optimization Pragmas

This section lists pragmas that affect how optimization is done.

```
#pragma OPTIMIZE ON | OFF
```

This pragma turns optimization ON or OFF depending on which option you use.

```
#pragma OPT_LEVEL 1 | 2
```

An argument of `1` sets the optimization level to local. An argument of `2` sets the optimization level to global and local.

Shared Library Pragma

This section describes the pragma for assigning a version number to a shared library module.

```
#pragma HP_SHLIB_VERSION "mm/[yy]yy"
```

This assigns a version number to a shared library module. This enables you to store multiple versions of a subroutine in a shared library.

The version number is specified by *mm*/[*yy*]*yy*. *mm* represents the month, and must be in the range 1–12. [*yy*]*yy* represents the year, either in 2 or 4 digits. If the 2 digit form is used it must be in the range 90–99 and will be interpreted as 1990–1999. The 4 digit form must be in the range 1990–7450.

This pragma provides a way to guard against unexpected side-effects when a shared library is updated. You can put multiple versions of a routine in the library and ensure that programs use the correct version. The date in the `SHLIB_VERSION` pragma provides the version number. Programs call the version in the shared library with a date less than or equal to the date the program was linked.

The version number should only be incremented when changes made to a subroutine make the new version of the subroutine incompatible with previous versions.

Data-Alignment Pragma

This section discusses the data-alignment pragma `HP_ALIGN` and its various arguments available on the HP 9000 Series 600, 700, and 800, to control alignment across platforms.

```
#pragma HP_ALIGN alignment
```

alignment can be one of a number of target machine names or `POP`. `POP` removes any `HP_ALIGN` settings and reverts back to the native alignment for the host machine. This pragma takes effect from its placement point until a subsequent encounter of the `HP_ALIGN` pragma.

Listing Pragmas

The following listing pragmas are available:

```
#pragma LINES linenum
```

This sets the number of lines per page to *linenum*. The default is 63. The minimum number of lines per page is 20.

```
#pragma WIDTH pagewidth
```

This sets the width of the page to *pagewidth*. The default is 80 columns. The minimum number of columns per page is 50. Place the `WIDTH` pragma before any `TITLE` or `SUBTITLE` pragmas. The width of the title and subtitle fields varies with the page width.

```
#pragma TITLE "string"
```

This makes *string* the title of the listing. *string* can have a length of up to 44 characters less than the page width (additional characters are truncated with no warning). The default is no title.

```
#pragma SUBTITLE "string"
```

This directive makes *string* the subtitle of the listing. *string* can have a length of up to 44 characters less than the page width (additional characters are truncated with no warning). The default is no subtitle.

```
#pragma PAGE
```

This causes a page break and begins a new page.

```
#pragma LIST ON | OFF
```

This turns listing functionality `ON` or `OFF` when used with the `-Wc,-L` command line option. The default is `ON`. Use this pragma to exclude source lines you do not need to list such as headers.

```
#pragma AUTOPAGE ON | OFF
```

This directive is used to produce a page break after each function definition. The default is `OFF`.

Miscellaneous Pragmas

`#pragma ALLOCS_NEW_MEMORY` $fname_1$, ..., $fname_n$

This directive states that the function *fname* returns a pointer to new memory that it allocates, or a routine that it calls allocates. This pragma provides additional information to the optimizer which results in more efficient code.

```
#pragma COPYRIGHT "string"
```

This directive places a copyright notice in the object file, using the *string* argument and the date specified using `COPYRIGHT_DATE`. If no date has been specified using `#pragma COPYRIGHT_DATE`, the current year is used. For example, assuming the year is 1990, the directive `#pragma COPYRIGHT "Acme Software"` places the following string in the object code:

```
(C) Copyright Acme Software, 1990.  All rights
reserved.  No part of this program may be photocopied,
reproduced, or transmitted without prior written
consent of Acme Software.
```

```
#pragma COPYRIGHT_DATE "string"
```

This specifies a date string to be used in a copyright notice appearing in an object module.

`#pragma FLOAT_TRAPS_ON` *fname*$_1$`,` ... `,` *fname*$_n$

This informs the compiler that function *fname* may enable floating-point trap handling. When the compiler is so informed, it will not perform loop-invariant code motion on floating-point operations in the functions named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled and the code is optimized.

`#pragma INTRINSIC` *intrinsic*$_1$ [*user-name*], ...

This pragma declares an external *user-name* as an intrinsic.

`#pragma INTRINSIC_FILE "`*path*`"`

This directive specifies the path of a file in which the compiler can locate information about intrinsic functions.

`#pragma LOCALITY "`*string*`"`

This specifies a name to be associated with the code written to a relocatable object module. All code following the `LOCALITY` pragma is associated with the name specified in *string*. The smallest scope of a unique `LOCALITY` pragma is a function. For example, `#pragma LOCALITY "MINE"` will build the name `"$CODE$MINE$"`.

Code that is not headed by a `LOCALITY` pragma is associated with the name `$CODE$`. An empty *string* causes the code name to revert to the default name of `$CODE$`.

`#pragma NO_SIDE_EFFECTS` *fname*$_1$`,` ... `,` *fname*$_n$

This directive states that *fname*, and all the functions that *fname* calls, will not modify any of a program's local or global variables. This pragma provides additional information to the optimizer which results in more efficient code.

`#pragma VERSIONID "`*string*`"`

This specifies a version string to be associated with a particular piece of code. The *string* is placed into the object file produced when the code is compiled.

∞

32. A Proposed Specification of BLAS Routines in C

S.P. Datardina, J.J. Du Croz,
S.J. Hammarling, and M.W. Pont
NAG Limited
Wilkinson House
Jordan Hill Road
OXFORD
United Kingdom, OX2 8R

Abstract

We propose a specification for a C implementation of Fortran's Basic Linear Algebra Subprograms (BLAS).

Introduction

In 1973, Hanson, Krogh, and Lawson described the advantages of adopting a set of basic routines for problems in linear algebra [6]. The original basic linear algebra subprograms, commonly referred to as the BLAS or, in view of subsequent developments, the Level 1 BLAS, have been very successful and have been used in a wide range of software including LINPACK [3]. An excellent discussion of the *raison d'être* of the BLAS is given in Dodson and Lewis, [2].

Subsequently, a set of Level 2 BLAS for matrix-vector operations, motivated by the development of vector-processing machines, was specified by Dongarra *et al.*, [4]. More recently a set of Level 3 BLAS for matrix-matrix operations, motivated by the development of hierarchical memory and parallel machines, has been specified by Dongarra *et al.*, [5].

Optimized implementations of all three levels of BLAS are now provided by many manufacturers of high performance computers. Thus the BLAS provide an infrastructure for the construction of portable and efficient software for linear algebra and related areas of numerical computation. In particular the BLAS are used as building blocks for LAPACK [1]. All three levels of BLAS are now included in the NAG Fortran Library [9], and are used extensively in many chapters of the Library.

NAG has recently launched a C Library [8]. In the continuing development of that library the question of C equivalents of the Fortran BLAS naturally arises. This report proposes a specification for a set of basic linear algebra functions in C. We welcome comments on the proposal.

BLAS in C

One immediate question arises as to the acronym for the basic linear algebra functions in C. The logical choice would perhaps be the BLAFs, but since BLAS has come to be thought of as a generic name, we propose that they be referred to as the BLAS in C, or the C BLAS.

It may be argued that there is no real need to have separate C versions of the BLAS. On many systems it is possible to call the Fortran BLAS from a C program simply by linking with the appropriate library routines, and possibly adjusting the names slightly, for example by adding an underscore to the end, and passing all scalar arguments by reference. We believe that there is a genuine need for the C BLAS. Firstly, even if all C programmers had access to the Fortran BLAS, they might not know how to write their names so that the C compiler recognises them. In any case, it is surely preferable to use a natural C interface. Secondly, and most importantly, the Level 2 and 3 BLAS functions operate on two-dimensional arrays to perform matrix-vector and matrix-matrix operations. Anyone calling the Fortran BLAS from C would need to be aware that such arrays are stored in row-major order in C and in column-major order in Fortran, meaning that matrices would need to be transposed before calling the BLAS. To avoid the unacceptable overhead of physically transposing matrices before and after BLAS calls, the entire C program would have to be written to store matrices in an unnatural way. This might make it almost impossible to insert BLAS calls into already existing C code.

The proposed argument lists closely follow the equivalent Fortran ones, the main changes being that enumerated types are used instead of character types for option specification, and leading dimension parameters (i.e., `LDA`) are replaced by trailing dimension parameters (i.e., `tda`). Inside the BLAS functions, two-dimensional arrays would typically be declared to have just one dimension, for example `double a[]`. The only array size information known to the function is therefore the trailing dimension parameter `tda` associated with the array `a` which is passed to the function by the user. This can be used to perform address arithmetic to access `a` as though it had two dimensions. Users are, of course, at liberty to declare their arrays to have either one or two dimensions, so long as they access the array elements correctly. This approach is preferable to the alternative approach of using vectors of pointers to vectors, which would force the user program to perform rather a large amount of work when initializing them, and because the BLAS are low-level building blocks.

It has been suggested that the names of the C versions of the BLAS should be different from the Fortran versions, because sometimes both Fortran and C versions will be available, and some systems do not distinguish between Fortran and C names. The distinction could be made, for example, simply by prefixing each name with `c_`. This would lead to names of up to eight characters, though only the first six are guaranteed unique by the C standard.

We would prefer to keep the C BLAS names identical to the Fortran BLAS. If this is not possible on any particular system, it should be relatively easy to

remove ambiguity by use of the `#define` directive. In this document, the usual Fortran names have been used.

The argument lists use the following data types:

`Integer`:	an integer data type of at least 32 bits.
`float`:	the regular single precision floating-point type.
`double`:	the regular double precision floating-point type.
`Scomplex`:	a single precision complex type.
`Complex`:	a double precision complex type.

plus the enumeration types given by

```
typedef enum {NoTranspose, Transpose, ConjugateTranspose}
        MatrixTranspose;
typedef enum {UpperTriangle, LowerTriangle}
        MatrixTriangle;
typedef enum {UnitTriangular, NotUnitTriangular}
        MatrixUnitTriangular;
typedef enum {LeftSide, RightSide}
        OperationSide;
```

The format of the names of the enumerated types, using phrases with capitalized initial letters and no underscores, is intended to follow one common naming convention.

It is expected that the complex data types would be stored in Cartesian form, i.e., as real and imaginary parts. For example:

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

The operations performed by the C BLAS are intended to be identical to those performed by the corresponding Fortran BLAS, as specified in [2], [4], and [5].

We have not proposed C equivalents of two of the Level 1 BLAS—`DQDOTI` and `DQDOTA`—which require quadruple precision arithmetic.

Since the publication of [2], [4], and [5], there have been some suggestions for adding to the Fortran BLAS—for example, Level 1 routines for complex-plane rotations, or Level 2 routines for complex-symmetric matrices. We have not proposed adding such routines to the C BLAS at this stage, because we hope that a 'standard' set of additions to the Fortran BLAS may be specified as a by-product of the LAPACK project [1]. It would then be straightforward to propose corresponding C BLAS, along the same lines as the routines proposed here.

The following code fragment, which overwrites the upper triangle of a symmetric positive-definite matrix by its Cholesky factorization, demonstrates how calls to the BLAS functions `dgemv` and `dscal` might be used. We assume that declarations and initializations such as the following have been made.

```
double a[10][10];
Integer i, j, k, n, tda;
n = 5;
tda = 10;
```

Without BLAS:

```
for (i = 0; i < n; ++i)
{
    for (k = 0; k < i; ++k)
        for (j = i; j < n; ++j)
            a[i][j] -= a[k][i] * a[k][j];
        a[i][i] = sqrt(a[i][i]);
        for (j = i + 1; j < n; ++j)
            a[i][j] = a[i][j] / a[i][i];
    }
```

With BLAS:

```
for (i = 0; i < n; ++i)
{
    dgemv(Transpose, i, n-i, -1.0, &a[0][i], tda, &a[0][i],
        tda, 1.0, &a[i][i], (Integer)1);
    a[i][i] = sqrt(a[i][i]);
    dscal(n - i - 1, 1.0/a[i][i], &a[i][i+1], (Integer)1);
}
```

Since the first draft of this paper, we have seen details of a rather different approach to defining C BLAS functions. Leyk [7] proposes a set of functions and macros that allow a user to maintain vectors and matrices with associated dimension information in structures. All vector and matrix storage must be allocated using the provided functions. The report restricts itself to consideration of Level 1 BLAS functions (apart from a matrix-vector multiply function), operating on real and integer types.

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1990). *LAPACK: A Portable Linear Algebra Library for High-Performance Computers* LAPACK Working Note No. 20, University of Tennessee, Techni-

cal Report CS-90-105, Department of Computer Science, 107 Ayres Hall, Knoxville, TN 37996-1301.

[2] Dodson, D. and Lewis, J. (1985). *Issues Relating to Extension of the Basic Linear Algebra Subprograms* ACM Signum Newsletter, **20**, pp. 2–18.

[3] Dongarra, J.J., Bunch, J.R., Moler, C.B., and Stewart, G.W. (1978). LINPACK Users' Guide. SIAM, Philadelphia.

[4] Dongarra, J.J., Du Croz, J., Hammarling, S., and Hanson, R.J. (1988). *An Extended Set of FORTRAN Basic Linear Algebra Subprograms* ACM Trans. Math. Softw., **14**, pp. 1–32.

[5] Dongarra, J.J., Du Croz, J., Duff, I.S., and Hammarling, S. (1990). *A Set of Level 3 Basic Linear Algebra Subprograms* ACM Trans. Math. Softw., **16**, pp. 1–28.

[6] Hanson, R., Krogh, F., and Lawson, C. (1973). *A Proposal for Standard Linear Algebra Subprograms* ACM Signum Newsletter, **8**, p. 16.

[7] Leyk, Z. (1991). *Basic Linear Algebra Functions for C Language Usage (C-BLAF)* Cornell University, Technical Report '91 – 24, Mathematical Sciences Institute.

[8] The NAG C Library, Mark 1. (1990). The Numerical Algorithms Group Ltd, Oxford.

[9] The NAG Fortran Library, Mark 15. (1991). The Numerical Algorithms Group Ltd, Oxford.

Level 1 BLAS in C

Generate a plane rotation

```
void srotg( float *a,  float *b,  float *c,  float *s);
void drotg(double *a, double *b, double *c, double *s);
```

Generate a modified plane rotation

```
void srotmg( float *d1,  float *d2,  float *a,  float b,  float *param);
void drotmg(double *d1, double *d2, double *a, double b, double *param);
```

Apply a plane rotation

```
void srot(Integer n, float *x, Integer incx, float *y, Integer incy,
    float c, float s);

void drot(Integer n, double *x, Integer incx, double *y, Integer incy,
    double c, double s);
```

Apply a modified plane rotation

```
void srotm(Integer n, float *x, Integer incx, float *y, Integer incy,
    float *param);

void drotm(Integer n, double *x, Integer incx, double *y, Integer incy,
    double *param);
```

$x \leftrightarrow y$

```
void sswap(Integer n,    float *x, Integer incx,    float *y, Integer incy);
void dswap(Integer n,   double *x, Integer incx,   double *y, Integer incy);
void cswap(Integer n, Scomplex *x, Integer incx, Scomplex *y, Integer incy);
void zswap(Integer n,  Complex *x, Integer incx,  Complex *y, Integer incy);
```

$x \leftarrow \alpha x$

```
void  sscal(Integer n,    float alpha,    float *x, Integer incx);
void  dscal(Integer n,   double alpha,   double *x, Integer incx);
void  cscal(Integer n, Scomplex alpha, Scomplex *x, Integer incx);
void  zscal(Integer n,  Complex alpha,  Complex *x, Integer incx);
void csscal(Integer n,    float alpha, Scomplex *x, Integer incx);
void zdscal(Integer n,   double alpha,  Complex *x, Integer incx);
```

$y \leftarrow x$

```
void scopy(Integer n,    float *x, Integer incx,    float *y, Integer incy);
void dcopy(Integer n,   double *x, Integer incx,   double *y, Integer incy);
void ccopy(Integer n, Scomplex *x, Integer incx, Scomplex *y, Integer incy);
void zcopy(Integer n,  Complex *x, Integer incx,  Complex *y, Integer incy);
```

$y \leftarrow \alpha x + y$

```
void saxpy(Integer n, float alpha, float *x, Integer incx, float *y,
    Integer incy);

void daxpy(Integer n, double alpha, double *x, Integer incx, double *y,
    Integer incy);

void caxpy(Integer n, Scomplex alpha, Scomplex *x, Integer incx,
    Scomplex *y, Integer incy);

void zaxpy(Integer n, Complex alpha, Complex *x, Integer incx,
    Complex *y, Integer incy);
```

$dot \leftarrow x^T y$

```
float   sdot(Integer n,  float *x, Integer incx,  float *y, Integer incy);
double dsdot(Integer n,  float *x, Integer incx,  float *y, Integer incy);
double  ddot(Integer n, double *x, Integer incx, double *y, Integer incy);
```

$dot \leftarrow \alpha + x^T y$

```
float sdsdot(Integer n, float alpha, float *x, Integer incx,
    float *y, Integer incy);
```

$dotu \leftarrow x^T y$

```
Scomplex cdotu(Integer n, Scomplex *x, Integer incx, Scomplex *y,
    Integer incy);

Complex  zdotu(Integer n,  Complex *x, Integer incx,  Complex *y,
    Integer incy);
```

$dotc \leftarrow x^H y$

```
Scomplex cdotc(Integer n, Scomplex *x, Integer incx, Scomplex *y,
    Integer incy);

Complex  zdotc(Integer n,  Complex *x, Integer incx,  Complex *y,
    Integer incy);
```

$nrm2 \leftarrow \|x\|_2$

```
float   snrm2(Integer n,    float *x, Integer incx);
double  dnrm2(Integer n,   double *x, Integer incx);
float  scnrm2(Integer n, Scomplex *x, Integer incx);
double dznrm2(Integer n,  Complex *x, Integer incx);
```

$asum \leftarrow \|x\|_1$

```
float  sasum(Integer n,  float *x, Integer incx);
double dasum(Integer n, double *x, Integer incx);
```

$asum \leftarrow \|\mathrm{Re}\,(x)\|_1 + \|\mathrm{Im}\,(x)\|_1$

```
float  scasum(Integer n, Scomplex *x, Integer incx);
double dzasum(Integer n,  Complex *x, Integer incx);
```

$amax \leftarrow$ first k such that $|x_k| = \max|x_i|$

```
Integer isamax(Integer n,  float *x, Integer incx);
Integer idamax(Integer n, double *x, Integer incx);
```

$amax \leftarrow$ first k such that $|\mathrm{Re}(x_k)| + |\mathrm{Im}(x_k)| = \max(|\mathrm{Re}(x_i)| + |\mathrm{Im}(x_i)|)$

```
Integer icamax(Integer n, Scomplex *x, Integer incx);
Integer izamax(Integer n,  Complex *x, Integer incx);
```

Level 2 BLAS in C

$y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, $y \leftarrow \alpha A^H x + \beta y$, A is m by n

```
void sgemv(MatrixTranspose trans, Integer m, Integer n, float alpha,
    float *a, Integer tda, float *x, Integer incx, float beta,
    float *y, Integer incy);

void dgemv(MatrixTranspose trans, Integer m, Integer n, double alpha,
    double *a, Integer tda, double *x, Integer incx, double beta,
    double *y, Integer incy);
```

```
void cgemv(MatrixTranspose trans, Integer m, Integer n, Scomplex alpha,
    Scomplex *a, Integer tda, Scomplex *x, Integer incx,
    Scomplex beta, Scomplex *y, Integer incy);

void zgemv(MatrixTranspose trans, Integer m, Integer n, Complex alpha,
    Complex *a, Integer tda, Complex *x, Integer incx, Complex beta,
    Complex *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, $y \leftarrow \alpha A^T x + \beta y$, $y \leftarrow \alpha A^H x + \beta y$, A is m by n band

```
void sgbmv(MatrixTranspose trans, Integer m, Integer n, Integer kl,
    Integer ku, float alpha, float *a, Integer tda, float x,
    Integer incx, float beta, float *y, Integer incy);

void dgbmv(MatrixTranspose trans, Integer m, Integer n, Integer kl,
    Integer ku, double alpha, double *a, Integer tda, double x,
    Integer incx, double beta, double *y, Integer incy);

void cgbmv(MatrixTranspose trans, Integer m, Integer n, Integer kl,
    Integer ku, Scomplex alpha, Scomplex *a, Integer tda, Scomplex x,
    Integer incx, Scomplex beta, Scomplex *y, Integer incy);

void zgbmv(MatrixTranspose trans, Integer m, Integer n, Integer kl,
    Integer ku, Complex alpha, Complex *a, Integer tda, Complex x,
    Integer incx, Complex beta, Complex *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n Hermitian

```
void chemv(MatrixTriangle uplo, Integer n, Scomplex alpha,
    Scomplex *a, Integer tda, Scomplex *x, Integer incx,
    Scomplex beta, Scomplex *y, Integer incy);

void zhemv(MatrixTriangle uplo, Integer n, Complex alpha,
    Complex *a, Integer tda, Complex *x, Integer incx,
    Complex beta, Complex *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n Hermitian band

```
void chbmv(MatrixTriangle uplo, Integer n, Integer k, Scomplex alpha,
    Scomplex *a, Integer tda, Scomplex *x, Integer incx,
    Scomplex beta, Scomplex *y, Integer incy);

void zhbmv(MatrixTriangle uplo, Integer n, Integer k, Complex alpha,
    Complex *a, Integer tda, Complex *x, Integer incx,
    Complex beta, Complex *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n Hermitian in packed form

```
void chpmv(MatrixTriangle uplo, Integer n, Scomplex alpha, Scomplex *ap,
    Scomplex *x, Integer incx, Scomplex beta, Scomplex *y, Integer incy);

void zhpmv(MatrixTriangle uplo, Integer n, Complex alpha, Complex *ap,
    Complex *x, Integer incx, Complex beta, Complex *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n symmetric

```
void ssymv(MatrixTriangle uplo, Integer n, float alpha, float *a,
    Integer tda, float *x, Integer incx, float beta, float *y, Integer incy);
```

```
void dsymv(MatrixTriangle uplo, Integer n, double alpha, double *a, Integer
    tda, double *x, Integer incx, double beta, double *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n symmetric band

```
void ssbmv(MatrixTriangle uplo, Integer n, Integer k, float alpha,
    float *a, Integer tda, float *x, Integer incx, float beta,
    float *y, Integer incy);

void dsbmv(MatrixTriangle uplo, Integer n, Integer k, double alpha,
    double *a, Integer tda, double *x, Integer incx, double beta,
    double *y, Integer incy);
```

$y \leftarrow \alpha Ax + \beta y$, A is n by n symmetric in packed form

```
void sspmv(MatrixTriangle uplo, Integer n, float alpha, float *ap,
    float *x, Integer incx, float beta, float *y, Integer incy);

void dspmv(MatrixTriangle uplo, Integer n, double alpha, double *ap,
    double *x, Integer incx, double beta, double *y, Integer incy);
```

$x \leftarrow Ax$, $x \leftarrow A^T x$, $x \leftarrow A^H x$, A is n by n triangular

```
void strmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, float *a, Integer tda,
    float *x, Integer incx);

void dtrmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, double *a, Integer tda,
    double *x, Integer incx);

void ctrmv(MatrixTriangle uplo, MatrixTranspose trans,
MatrixUnitTriangular diag, Integer n, Scomplex *a, Integer tda,
    Scomplex *x, Integer incx);

void ztrmv(MatrixTriangle uplo, MatrixTranspose trans,
MatrixUnitTriangular diag, Integer n, Complex *a, Integer tda,
    Complex *x, Integer incx);
```

$x \leftarrow Ax$, $x \leftarrow A^T x$, $x \leftarrow A^H x$, A is n by n triangular band

```
void stbmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, Integer k, float *a,
    Integer tda, float *x, Integer incx);

void dtbmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, Integer k, double *a,
    Integer tda, double *x, Integer incx);

void ctbmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, Integer k, Scomplex *a,
    Integer tda, Scomplex *x, Integer incx);

void ztbmv(MatrixTriangle uplo, MatrixTranspose trans,
    MatrixUnitTriangular diag, Integer n, Integer k, Complex *a,
    Integer tda, Complex *x, Integer incx);
```

$x \leftarrow Ax$, $x \leftarrow A^T x$, $x \leftarrow A^H x$, A is n by n triangular in packed form

```
void stpmv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, float *ap, float *x, Integer incx);

void dtpmv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, double *ap, double *x, Integer incx);

void ctpmv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Scomplex *ap, Scomplex *x, Integer incx);

void ztpmv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Complex *ap, Complex *x, Integer incx);
```

$x \leftarrow A^{-1}x$, $x \leftarrow A^{-T}x$, $x \leftarrow A^{-H}x$, A is n by n triangular

```
void strsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, float *a, Integer tda, float *x, Integer incx);

void dtrsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, double *a, Integer tda, double *x, Integer incx);

void ctrsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Scomplex *a, Integer tda, Scomplex *x, Integer incx);

void ztrsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Complex *a, Integer tda, Complex *x, Integer incx);
```

$x \leftarrow A^{-1}x$, $x \leftarrow A^{-T}x$, $x \leftarrow A^{-H}x$, A is n by n triangular band

```
void stbsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Integer k, float *a, Integer tda, float *x,
    Integer incx);

void dtbsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Integer k, double *a, Integer tda, double *x,
    Integer incx);

void ctbsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Integer k, Scomplex *a, Integer tda, Scomplex *x,
    Integer incx);

void ztbsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Integer k, Complex *a, Integer tda, Complex *x,
    Integer incx);
```

$x \leftarrow A^{-1}x$, $x \leftarrow A^{-T}x$, $x \leftarrow A^{-H}x$, A is n by n triangular in packed form

```
void stpsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, float *ap, float *x, Integer incx);

void dtpsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, double *ap, double *x, Integer incx);

void ctpsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Scomplex *ap, Scomplex *x, Integer incx);
```

```
void ztpsv(MatrixTriangle uplo, MatrixTranspose trans, MatrixUnitTriangular
    diag, Integer n, Complex *ap, Complex *x, Integer incx);
```

$A \leftarrow \alpha x y^T + A$, A is m by n

```
void sger(Integer m, Integer n, float alpha, float *x, Integer incx,
    float *y, Integer incy, float *a, Integer tda);

void dger(Integer m, Integer n, double alpha, double *x, Integer incx,
    double *y, Integer incy, double *a, Integer tda);
```

$A \leftarrow \alpha x y^T + A$, A is m by n

```
void cgeru(Integer m, Integer n, Scomplex alpha, Scomplex *x,
    Integer incx, Scomplex *y, Integer incy, Scomplex *a, Integer tda);

void zgeru(Integer m, Integer n, Complex alpha, Complex *x,
    Integer incx, Complex *y, Integer incy, Complex *a, Integer tda);
```

$A \leftarrow \alpha x y^H + A$, A is m by n

```
void cgerc(Integer m, Integer n, Scomplex alpha, Scomplex *x,
    Integer incx, Scomplex *y, Integer incy, Scomplex *a, Integer tda);

void zgerc(Integer m, Integer n, Complex alpha, Complex *x,
    Integer incx, Complex *y, Integer incy, Complex *a, Integer tda);
```

$A \leftarrow \alpha x x^H + A$, A is n by n Hermitian

```
void cher(MatrixTriangle uplo, Integer n, Scomplex alpha,
    Scomplex *x, Integer incx, Scomplex *a, Integer tda);

void zher(MatrixTriangle uplo, Integer n, Complex alpha,
    Complex *x, Integer incx, Complex *a, Integer tda);
```

$A \leftarrow \alpha x x^H + A$, A is n by n Hermitian in packed form

```
void chpr(MatrixTriangle uplo, Integer n, Scomplex alpha,
    Scomplex *x, Integer incx, Scomplex *ap);

void zhpr(MatrixTriangle uplo, Integer n, Complex alpha,
    Complex *x, Integer incx, Complex *ap);
```

$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$, A is n by n Hermitian

```
void cher2(MatrixTriangle uplo, Integer n, Scomplex alpha, Scomplex *x,
    Integer incx, Scomplex *y, Integer incy, Scomplex *a, Integer tda);

void zher2(MatrixTriangle uplo, Integer n, Complex alpha, Complex *x,
    Integer incx, Complex *y, Integer incy, Complex *a, Integer tda);
```

$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$, A is n by n Hermitian in packed form

```
void chpr2(MatrixTriangle uplo, Integer n, Scomplex alpha, Scomplex *x,
    Integer incx, Scomplex *y, Integer incy, Scomplex *ap);
```

```
void zhpr2(MatrixTriangle uplo, Integer n, Complex alpha, Complex *x,
    Integer incx, Complex *y, Integer incy, Complex *ap);
```

$A \leftarrow \alpha x x^T + A$, A is n by n symmetric

```
void ssyr(MatrixTriangle uplo, Integer n,    float alpha,
    float *x, Integer incx,    float *a, Integer tda);

void dsyr(MatrixTriangle uplo, Integer n, double alpha,
    double *x, Integer incx, double *a, Integer tda);
```

$A \leftarrow \alpha x x^T + A$, A is n by n symmetric in packed form

```
void sspr(MatrixTriangle uplo, Integer n,    float alpha,
    float *x, Integer incx,    float *ap);

void dspr(MatrixTriangle uplo, Integer n, double alpha,
    double *x, Integer incx, double *ap);
```

$A \leftarrow \alpha x y^T + \alpha y x^T + A$, A is n by n symmetric

```
void ssyr2(MatrixTriangle uplo, Integer n, float alpha, float *x,
    Integer incx, float *y, Integer incy, float *a, Integer tda);

void dsyr2(MatrixTriangle uplo, Integer n, double alpha, double *x,
    Integer incx, double *y, Integer incy, double *a, Integer tda);
```

$A \leftarrow \alpha x y^T + \alpha y x^T + A$, A is n by n symmetric in packed form

```
void sspr2(MatrixTriangle uplo, Integer n, float alpha, float *x,
    Integer incx, float *y, Integer incy, float *ap);

void dspr2(MatrixTriangle uplo, Integer n, double alpha, double *x,
    Integer incx, double *y, Integer incy, double *ap);
```

Level 3 BLAS in C

$C \leftarrow \alpha op(A) op(B) + \beta C$, $op(X) = X, X^T, X^H$, C is m by n

```
void sgemm(MatrixTranspose transa, MatrixTranspose transb, Integer m,
    Integer n, Integer k, float alpha, float *a, Integer tda,
    float *b, Integer tdb, float beta, float *c, Integer tdc);

void dgemm(MatrixTranspose transa, MatrixTranspose transb, Integer m,
    Integer n, Integer k, double alpha, double *a, Integer tda,
    double *b, Integer tdb, double beta, double *c, Integer tdc);

void cgemm(MatrixTranspose transa, MatrixTranspose transb, Integer m,
    Integer n, Integer k, Scomplex alpha, Scomplex *a, Integer tda,
    Scomplex *b, Integer tdb, Scomplex beta, Scomplex *c, Integer tdc);

void zgemm(MatrixTranspose transa, MatrixTranspose transb, Integer m,
    Integer n, Integer k, Complex alpha, Complex *a, Integer tda,
    Complex *b, Integer tdb, Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha BA + \beta C$, C is m by n, A is m by m symmetric

```
void ssymm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    float alpha, float *a, Integer tda, float *b, Integer tdb,
    float beta, float *c, Integer tdc);

void dsymm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    double alpha, double *a, Integer tda, double *b, Integer tdb,
    double beta, double *c, Integer tdc);

void csymm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b, Integer tdb,
    Scomplex beta, Scomplex *c, Integer tdc);

void zsymm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    Complex alpha, Complex *a, Integer tda, Complex *b, Integer tdb,
    Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha BA + \beta C$, C is m by n, A is m by m Hermitian

```
void chemm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b,
    Integer tdb, Scomplex beta, Scomplex *c, Integer tdc);

void zhemm(OperationSide side, MatrixTriangle uplo, Integer m, Integer n,
    Complex alpha, Complex *a, Integer tda, Complex *b, Integer tdb,
    Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AA^T + \beta C$, $C \leftarrow \alpha A^T A + \beta C$, C is n by n

```
void ssyrk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k,    float alpha,    float *a, Integer tda,
    float beta,    float *c, Integer tdc);

void dsyrk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, double alpha, double *a, Integer tda,
    double beta, double *c, Integer tdc);

void csyrk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Scomplex alpha, Scomplex *a, Integer tda,
    Scomplex beta, Scomplex *c, Integer tdc);

void zsyrk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Complex alpha, Complex *a, Integer tda,
    Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AA^H + \beta C$, $C \leftarrow \alpha A^H A + \beta C$, C is n by n

```
void cherk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Scomplex alpha, Scomplex *a, Integer tda,
    Scomplex beta, Scomplex *c, Integer tdc);

void zherk(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Complex alpha, Complex *a, Integer tda,
    Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C$, $C \leftarrow \alpha A^H B + \alpha B^H A + \beta C$, C is n by n

```
void cher2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b,
    Integer tdb, Scomplex beta, Scomplex *c, Integer tdc);

void zher2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Complex alpha, Complex *a, Integer tda, Complex *b,
    Integer tdb, Complex beta, Complex *c, Integer tdc);
```

$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$, $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, C is n by n

```
void ssyr2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, float alpha, float *a, Integer tda, float *b,
    Integer tdb, float beta, float *c, Integer tdc);

void dsyr2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, double alpha, double *a, Integer tda, double *b,
    Integer tdb, double beta, double *c, Integer tdc);

void csyr2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b,
    Integer tdb, Scomplex beta, Scomplex *c, Integer tdc);

void zsyr2k(MatrixTriangle uplo, MatrixTranspose trans, Integer n,
    Integer k, Complex alpha, Complex *a, Integer tda, Complex *b,
    Integer tdb, Complex beta, Complex *c, Integer tdc);
```

$B \leftarrow \alpha op(A)B$, $B \leftarrow \alpha Bop(A)$, $op(A) = A, A^T, A^H$, B is m by n

```
void strmm(MatrixTriangle side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    float alpha, float *a, Integer tda, float *b, Integer tdb);

void dtrmm(MatrixTriangle side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    double alpha, double *a, Integer tda, double *b, Integer tdb);

void ctrmm(MatrixTriangle side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b, Integer tdb);

void ztrmm(MatrixTriangle side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    Complex alpha, Complex *a, Integer tda, Complex *b, Integer tdb);
```

$B \leftarrow \alpha op(A^{-1})B$, $B \leftarrow \alpha Bop(A^{-1})$, $op(A) = A, A^T, A^H$, B is m by n

```
void strsm(OperationSide side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    float alpha, float *a, Integer tda, float *b, Integer tdb);

void dtrsm(OperationSide side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    double alpha, double *a, Integer tda, double *b, Integer tdb);
```

```
void ctrsm(OperationSide side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    Scomplex alpha, Scomplex *a, Integer tda, Scomplex *b, Integer tdb);

void ztrsm(OperationSide side, MatrixTriangle uplo, MatrixTranspose
    transa, MatrixUnitTriangular diag, Integer m, Integer n,
    Complex alpha, Complex *a, Integer tda, Complex *b, Integer tdb);
```

The Authors are all involved with the development of numerical software for the NAG Libraries. Jeremy Du Croz and Sven Hammarling where also involved in the development of the Level 2 and Level 3 BLAS in Fortran 77. They can be reached at sven@nag.co.uk.

∞

33. Trigraphs: The Search for Alternatives

Jim Brodie
Motorola, Inc.
Tempe, Arizona

Abstract

Trigraphs, introduced by X3J11 to address differences in the character sets used around the world, have proven to be one of the more controversial additions to the C language. A number of people, most notably the ISO C committee representative from Denmark, have opposed trigraphs as unreadable and have asked for alternative solutions. This paper will look at some of the history behind this controversy. We will look at several of the proposals that have been presented, the issues raised, and the resulting compromise positions that were recently approved within WG14, the ISO C language standard committee, and X3J16/WG21, the U.S. and ISO C++ standards committees.

Background

The C language, as originally defined, requires a large set of characters, including a number of special characters (such as `#`, `|`, `[`, {, and `~`). These characters have proven to be a problem because they are missing from many of the national character sets in use around the world. To understand this situation better we need to look, at least briefly, at the international standards for character sets.

In many European (and other) countries the national character set standard is a version of the International Organization for Standardization (ISO) 646:1983 international standard for 7-bit character sets. ISO-646 splits the character set into two parts. It defines a core set of characters that all standard-compliant character sets must provide. This is known as the ISO-646 Invariant Subset. Included in this subset are the standard English upper- and lowercase letters, the digits 0–9, and the printing characters `!`, `"`, `%`, `&`, `'`, `(`, `)`, `*`, `+`, space, `,`, `-`, `.`, `/`, `:`, `;`, `<`, `=`, `>`, and `?`.

ISO-646 also sets aside a part of the character set so that each national standard group can define special characters that are needed in their environment. In the U.S., the ASCII character set was used to fill in the extra slots. These nation-specific character-set positions are used to define characters such as `#`, `[`, and {. In other parts of the world these character-set positions have been assigned to other characters such as £, ä, or Æ.

If we are going to define the C language so that it can be effectively written using any of these standard national characters sets, we must allow alternatives for the original C characters that fall into the nation-specific portion of the character sets.

The solution of using whatever national characters happen to occupy the same character set positions has been used but results in programs that are very difficult to read, write, and understand. A better solution is to allow programs to be written using only the ISO-646 Invariant Subset.

The C characters that fall into the nation-specific portion of the ISO-646 character set are: `#`, `[`, `]`, {, }, \, `|`, `~`, and `^`.

The Search for Solutions

Early in the X3J11 deliberations (in 1984) proposals were made to address this issue. These proposals focused on providing alternative spellings or representations for the special characters. The proposal that gained wide-spread support was prepared by two European members of X3J11, Mike Banahan and Keith Winter. Their proposal was the foundation for the current trigraph solution.

After examining the ISO-646 Invariant Subset, it became clear to Banahan and Winter that simple digraphs (two character escape sequences) would not be sufficient to allow alternate spellings for the special characters. Every one of the characters in the ISO-646 Invariant Subset is used by C. This means that there is no single, always available, character that can be used to signal the start of an alternate spelling. This led to the conclusion that a two-character escape sequence was needed to indicate an alternate spelling for one of the special characters. (It was unfortunate that the escape character already used in C, the backslash \, was one of the problem children. Otherwise, the backslash's escape function could have been extended to cover these additional characters.)

This led to the proposal of trigraphs—alternate spellings comprised of three characters. Trigraphs use a two-character escape-sequence prefix, followed by a single character to represent each special characters. The Winter/Banahan paper proposed the use of a `??` escape sequence prefix followed by a single character that was similar to the character that the trigraph sequence was replacing.

The following is the set of the special characters along with the trigraphs that were recommended and adopted:

Standard C Trigraphs

Sequence	*Meaning*
??!	|
??'	^
??(	[
??)	]
??-	~
??/	\
??<	{
??=	#
??>	}

Winter and Banahan explained their selection of the ?? escape sequence in the following way:

> "The reason for choosing the ?? as the introductory escape sequence is that it is not used anywhere else in C and suggests that "something funny" is going on. From the point of view of readability, we have chosen to use a set of trigraphs that have some graphical similarity with the characters that they are supposed to stand for. Furthermore most of them can be surrounded by white space when they are used, allowing them to stand out.
>
> The sequence ??? can only be generated when a ? is followed by a character requiring trigraph representation and must be treated as such. Whenever that sequence is seen on the input stream, it must be interpreted as a single query mark followed by what may or may not turn out to be the start of a trigraph."

Perhaps we should have seen the controversy that was coming when there was significant debate on the form of the trigraphs. The minutes of the X3J11 meeting where the trigraphs were adopted noted:

> "There followed a rambling tour of 'Punctuation Land' as people suggested various escape characters for writing trigraphs."

The impact of using trigraphs can be seen in the following example. The first version of the program uses the full familiar character set:

```
#include <stdio.h>

main(int argc, char *argv[])
{
        if (argc < 1 || *argv[0] == '\0') return;
        printf("Hello, %s\n", argv[0]);
}
```

Using trigraphs this program becomes:

```
??=include <stdio.h>

main(int argc, char *argv??(??))
??<
        if (argc < 1 ??!??! *argv??(0??) == '??/0') return;
        printf("Hello, %s??/n", argv??(0??));
??>
```

The Search for More Readable Alternatives

Even when trigraphs were proposed, people realized that they were not the perfect solution. Even Banahan and Winter noted in their proposal "[Trigraphs] are neither elegant nor pleasing ..."

As people began to work with trigraphs, more and more complaints were heard that the resulting programs were unreadable. As a result, various proposals were made to solve this readability problem.

The latest round of proposals for an alternate to trigraphs can be traced to a 1988 paper by Keld Simonsen and Bjarne Stroustrup. The proposal split the problem into two pieces. Within character and string literals, trigraphs were accepted as the only viable alternative. However, for other contexts, alternatives to the trigraphs were provided.

The proposal recommended a collection of keywords and new lexical symbols as follows:

Keyword Alternative

Keyword	*Meaning*
or	|
cor	||
and	&
cand	&&
xor	^
compl	~

This still left the square brackets and braces, as well as the backslash. *[Ed: It also introduced a "solution" for the* & *character which was not part of the problem.]* Rather than introduce new keywords for these, the proposal presented a set of digraphs. In the limited context of C operators and punctuators, digraphs can be used to supply alternative spellings for the special characters. The proposed alternative ways of representing these remaining characters included:

Digraph Alternative #1

Digraph	*Meaning*
(:	{
:)	}
!(	[
)	]

The Simonsen and Stroustrup paper made the following comments on their proposed choices:

> "We decided to make the compound statement brackets digraphs, finding `begin` and `end` too long to write and too likely to be found 'not in the spirit of C' by large numbers of programmers."

The proposal would allow the above program to be written

```
??=include <stdio.h>

main(int argc, char *argv!())
(:
        if (argc < 1 cor *argv!(0) == '??/0') return;
        printf("Hello, %s??/n", argv!(0));
:)
```

The paper further proposed that an extension to this would be to allow the `!` to be viewed as an infix subscripting operator. The authors noted:

> "The grammar of C precludes using `(` and `)` for subscripting (as is done for many other languages). Parentheses are typically unnecessary for subscripting and `!` should be considered an infix subscript operator (as in BCPL). The binding strength of the binary `!` operator (subscripting) should be just above the unary operators and it should be left-associative. For example, `a!b.c!2*d` means `((a!(b.c))!2)*d`."

This addition, allows the sample program to be written as follows:

```
??=include <stdio.h>

main(int argc, char *argv!)
(:
        if (argc < 1 cor *argv!0 == '??/0') return;
        printf("Hello, %s??/n", argv!0);
:)
```

Essentially this same proposal was later proposed to WG14, the ISO committee producing an international standard for C.

This proposal led to significant debate within X3J11. Overall, the committee did not support the proposal.

There were objections to the introduction of additional keywords, with the side-effect of breaking existing code. Although the position was not unanimous, there was a preference for a collection of macros rather than new keywords. Tom Plum, vice-chair of X3J11, noted in one of his papers responding to the proposal:

> "It has been suggested several times that simple macro names provide all that is required for greater readability.
>
> It would be a pure extension to define a new header—call it `<iso646.h>` say—in which a specified set of new names are provided. The S&S [Simonsen and Stroustrup] proposal goes so far as to suggest that 'the new keywords could be conditionally in effect for new programs.' If they are conditional, then why not confine them to a header-file definition?
>
> The only point in the S&S paper that favors new keywords over a header-file is that (by re-programming every parser) the new keywords can be combined with `=` to make assignment operators. But on the other hand, the header-file can provide `or_equals`, `and_equals`, and `xor_equals`."

Another major objection appeared in the area of the new use of `!` in array declarations and subscripting operations. This part of the proposal goes beyond simple spelling replacements. Beyond the concern for introducing a new operator, it was pointed out that `!` becomes a postfix operator in the situation where it replaces empty brackets (e.g., `int a[]` becomes `int a!`). There was a concern that all of the ramifications of this addition might not be well understood for a long time. (X3J11 has experienced a number of "innocuous looking" changes where the full ramifications were not fully appreciated for several years.)

An additional concern was raised over the use of `(:` as a new alternate spelling for the square brackets. This character sequence can appear in an existing standard-conforming programs. Consider a macro invocation that accepts arguments. It is perfectly valid for the replacement tokens that make up the argument list to include colons. Consider the following simple example:

```
#define str( x )  #x

str(: Out of memory :)
```

In the current standard this is perfectly legitimate, resulting in the string

```
": Out of memory:".
```

Under the proposal the result would be equivalent to

```
str{Out of memory}
```

In fact, with macro invocations, it is possible to have any token sequence between the opening and closing parenthesis of the invocation argument list. This means that any digraph proposal that uses parentheses risks breaking existing code.

X3J11 recognized the need for a solution that would be acceptable to the U.S. while meeting the readability requirements of those proposing alternate solutions to the general use of trigraphs. In an attempt to give some specific guidance to those making alternative proposals, X3J11 passed several motions outlining the boundaries of what they would consider acceptable solutions to the readability problems of trigraphs. These motions were:

> "Solutions to the problem of national variants of ISO-646 should be restricted to alternate spellings for existing tokens and the introduction of new macros in new headers."

The second motion read:

> "Solutions to the problem of national variants of ISO-646 should *not* cause a change in behavior of a strictly conforming program."

These proposals clearly indicate that X3J11 would *not* support the introduction of new keywords or operators.

The Compromise

A compromise position has evolved from these proposals and the concerns expressed. It includes a standard header, `<iso646.h>`, that establishes the following mapping between a set of macro names and the corresponding operators:

Macro Alternative

Macro	*Meaning*
`bitand`	`&`
`and`	`&&`
`bitor`	`|`
`or`	`||`
`xor`	`^`
`compl`	`~`
`and_eq`	`&=`
`or_eq`	`|=`
`xor_eq`	`~=`

In addition, a new set of digraph spellings has been added as an alternative to a number of the other problem characters. The following six constructs are equivalent to the use of the listed characters *outside* character and string literals:

Digraph Alternative #2

Digraph	*Meaning*
<%	{
%>	}
<:	[
:>	]
%%	#
%%%%	##

The digraphs %% and %%%% are used only in preprocessing directives.

This proposal was accepted by WG14 at its last meeting in Milano, Italy, in December 1991. *[Ed: Acceptance was not unanimous. It will be considered by X3J11 at their next meeting, in May, in Salt Lake City.]*

Now our sample program is written as:

```
%%include <stdio.h>
%%include <iso646.h>

main(int argc, char *argv<::>)
<%
        if (argc < 1 or *argv<:0:> == '??/0') return;
        printf("Hello, %s??/n", argv<:0:>);
%>
```

The C++ Solution

X3J16 and WG21, the ANSI and ISO C++ committees, have adopted a slightly different proposal—they actually went further than the WG14 resolution.

X3J16/WG21 adopted the list of alternate names from the proposal accepted by WG14, plus two additional ones for "consistency." The additional names are:

C++ Additions

Name	*Meaning*
not	!
not_eq	!=

WG14 did not address ! since it *is* included in the ISO-646 invariant subset. The major difference between the proposal adopted by WG14 and the one adopted by X3J16/WG21 is not the additional names. **The C++ committees voted to treat all these new names as reserved library names, rather than as macro names.** This means that they will always be available, rather than only when <iso646.h> is explicitly included.

Since C++ will have a header called <iso646.h>, C programs will not be rejected by a C++ translator because of any reference to this header, but in C++ this header may well be empty. One advantage of this approach is that

everyone has access to the names and no one feels like his native environment is "second class" and must specify something special to get the translator to handle his national character set. On the negative side, there is the possibility that existing code will be broken and all translators will have to be modified to accept these new reserved words.

It is interesting to note the different tolerances for changes exhibited by X3J11 and X3J16/WG21. X3J11 has a "released product" and is reluctant to make changes that are not fully upward-compatible. Changes, particularly those which are not 100% upward-compatible with the existing standard, are a big deal. They are only supported if there is *no* other viable technical alternative to a serious problem. X3J16/WG21, on the other hand, is at a stage where it expects some, if not many, changes before they are ready to release their standard. All translators that wish to be C++ standard-conforming will be required to change in the future. Therefore, any change such as adding keywords is not perceived, at least emotionally, as being as significant. This is not to say that X3J11 is more mature or that X3J16/WG21 is more flexible. The committees are simply at different points in their life cycles. Because of this, it is not surprising that they address similar problems with different solutions. Whether a single common solution will be identified and accepted by both the C and C++ communities, before all this runs its course, is open to debate. We won't know for a while. It does, however, appear that a final solution is taking shape.

Postscript

In the long run (ten years, or less, perhaps), the use of trigraphs—and indeed digraphs—will diminish as machines using standard 8-bit character sets replace older equipment. These 8-bit character sets have national characters as well as room for all of the special characters used by C.

The evolution of the solution for the special character problems of C is typical of many of the problems addressed in the standards arena. The exchange of ideas and different proposals and counter proposals has allowed us to move to a compromise position that is acceptable to the majority of the people affected.

Jim Brodie is Chair of the ANSI C standards committee, X3J11, and a Chief Software Engineer for Motorola, Inc. He has coauthored books with P.J. Plauger and Tom Plum and is the Standards Editor for The Journal of C Language Translation. Jim can be reached at (602) 897-4390 or brodie@ssdt-tempe.sps.mot.com.

∞

34. Parasol: A C-Based Systems Programming Language, Part II

Bob Jervis
17645 Via Sereno
Monte Sereno, CA 95030

Abstract

This is the second of a two-part series that discusses a new systems programming language called Parasol, a language derived from C. New features are described including support for incremental compilation, object-oriented programming, and parallel programming. Each feature is compared with C, and a description is given as to why that feature was adopted.

Object-oriented Extensions (continued)

Initializers

Parasol allows structure constants that use the same syntax as aggregate initializers. I decided that I would merge the initializer syntax for static and automatic objects with the concept of structure constants. I changed the initializer syntax for Parasol to use square brackets instead of curly braces. This way, there would be no syntactic ambiguity in expressions. Curly braces starting a structure constant might be hard to distinguish from the beginning of a block at the start of an expression. Structure constants can't appear at the beginning of an expression, but the LALR grammar needed to disambiguate the cases might be a little complicated. Square brackets are completely unambiguous since subscripting can only appear after an operand[2].

For example:

```
Days: const [] int = [ 31, 28, 31, 30, 31, 30,
                       31, 31, 30, 31, 30, 31 ];
```

This statement initializes a constant array of integers with the number of days in each month. Parasol initializers do not allow brackets to be optionally omitted or included. Each aggregate object being initialized must have a

[2]NCEG has proposed an aggregate constant using {}'s, but that requires a leading cast.

bracket-enclosed initializer and each scalar initializer must be a simple expression. This simplification of the initializer syntax was not really necessary, but was done because I felt that the simpler rules were also more sensible. The new rules eliminate a parsing ambiguity that Standard C acknowledges. In older C implementations, partially bracketed initializers got different answers; it depended on the parsing technique used. Parasol simply prohibits such constructs altogether and avoids the problem.

The above statement can also be written as follows in Parasol:

```
Days:     [12] int;

Days = [ 31, 28, 31, 30, 31, 30,
         31, 31, 30, 31, 30, 31 ];
```

This sort of assignment obviously means that `Days` cannot be constant as in the first declaration, but in the second example one might conceivably assign a different initializer depending on whether the table is needed for a leap year.

Structures present some problems. Take the following for example:

```
Figure: {
        private:
        x:        int;
        y:        int;
        visible:
        sizex:    int;
        sizey:    int;
        public:
        shape:    figureShapes;
};

Figure = [ 10, 20, F_CIRCLE ];
```

In this example, I am assigning a structure constant to a structure with private members. Since you cannot access private members, you cannot initialize them. The rule is that private members are initialized to zero and the values in an initializer correspond to the public and visible members. In the above example, the `x` and `y` members are both set to zero, the `sizex` member is set to 10, `sizey` to 20 and `shape` to the constant `F_CIRCLE`.

Initializing Dynamic Function Vectors

An interesting question arises concerning dynamically allocated objects in Parasol. In a factory method, for example, you might call `alloc` to get the memory for an object. When the address is returned by `alloc` the memory is not yet tagged any particular way. By the time the method returns the address as a pointer to the object type, if you have any dynamic methods for the type, you

need to plug in the tag for it.

Static objects are initialized at compile-time, since the address of the dynamic vector table is known. An automatic object gets initialized when its declaration statement is reached.

One way of setting the tag field of an object is to copy a static object into the allocated memory. That notion of a copy into the target object is exactly how dynamic objects are initialized, but you don't have to create a static object just for the purpose. Simply assigning a structure constant through the pointer to the newly allocated object will set the tag field.

Object-Oriented Issues

While many older languages have been cited as possessing object-oriented features, it was SmallTalk that first established the term in the computer science community. SmallTalk is also among the purest expressions of the basic ideas. Philosophically, SmallTalk is miles away from C.

SmallTalk does no compile-time type checking, while C does only compile-time type checking. Like C, Parasol uses extensive static type checking. Computer instruction sets are almost universally designed to fix the data types with the instruction itself. Static compile-time type checking allows a compiler to pick the exact instruction needed for a given arithmetic operator and thus avoid a lot of runtime overhead. When execution time of arithmetic is important, compile-time type binding becomes a necessity. While SmallTalk interpreters can lower that overhead through cacheing, the overhead cannot be completely eliminated.

Since a major component of SmallTalk's flexibility is the ease of writing generic code that can, say, print or draw an object without the exact type of the object being known beforehand, compile-time type checking is not possible. You are trading the knowledge that most objects do have a known type at compile-time for the added complexity of managing references when you don't know the exact type.

SmallTalk uses a completely incremental development environment in which there is no boundary line between the operating system, the runtime library, and user supplied code. C is now defined with a very clear distinction between built-in operators, Standard library functions and user-supplied functions. Almost all C implementations also provide low-level interfaces to the underlying operating-system services as well.

The high integration of SmallTalk environments has not been used in Parasol. Even though the first implementation of Parasol runs on an operating system written in Parasol, it does not completely rely on that special operating system. Parasol should be implementable on many operating systems, although some of the parallel programming constructs require the equivalent of inter-process messages. In any event, a clear line is drawn between operating-system services and user code.

In SmallTalk an object has a type. In C, an object is defined as storage, but

type is imposed by the expression used to manipulate the storage. Thus, while it is in general not portable, C implementations support the ability to twiddle bits in data using several different data types. Low-level software like operating systems or compilers that must efficiently read data from external devices and process it must have this ability.

In Parasol I have adopted the notion of object as defined for C. Systems-programming code needs to dissect structured data in many subtle ways. Rigidly enforcing object identity and type is inconsistent with that need. This is far from a simple decision, however. Just as most objects have a known type at compile-time, most objects are not manipulated using multiple types. It is possible to state that the object has a type and retains it for its lifetime. Complicating matters is the use of `malloc` in C (`alloc` in Parasol) which skates at the thin edge of this distinction. `malloc` manipulates memory that your program may view as having a type other than 'array of `char`.'

SmallTalk assumes the presence of a garbage collector, while C does not. Almost all SmallTalk objects are stored as references while C requires that you explicitly declare and manage references.

Lastly, SmallTalk uses a very simple expression syntax with no operator precedence, and operators can be defined for new data types. In C, expression syntax is complicated, operators have many levels of precedence, and, are strictly limited to the built-in data types.

Given all these differences, it is difficult to find a satisfactory wedding of the concepts embodied in SmallTalk and the concepts embodied in C. What is the best way to merge the two languages? Is it best to think of bringing object-oriented extensions to C, or C extensions to SmallTalk? I have definitely looked at the problem as bringing object-oriented extensions to C. Let us examine each of the above major differences and how they have been dealt with in Parasol. Where relevant, I will also mention how C++ has dealt with the same issue.

Garbage Collection

Parasol does not incorporate a garbage collector. The mutable nature of type makes it next to impossible to collect garbage. Here is where Parasol begins to be different from C++ as well as SmallTalk.

Garbage collection itself is a well-understood process. In order to collect garbage, you must know where all objects are located that are still going to be needed to finish the program. All other objects are garbage and their memory can be reclaimed for reuse.

This is usually accomplished by tracing from an initial set of known objects (like static and auto objects) through all references to the set of allocated objects that can still be legitimately reached. This approach has the advantage that code runs at full speed, changing the values of references with impunity. At some point, usually when memory is getting low, this tracing algorithm is run. When the number of objects is very large this algorithm can take a long time to run.

SmallTalk implementations typically use a reference count approach that avoids the long execution of a tracing algorithm and absorbs overhead each time a reference value is changed. Reference counts alone are not sufficient to reclaim all garbage, however, since if two objects refer to each other and no other references exist, each of the two objects is garbage. Alas, the reference counts will not reveal this fact.

C++ defines destructors to serve as a mechanism to collect garbage in still another way. Instead of retrieving memory automatically, the designer of an object determines the memory allocation needed and the compiler merely arranges for the destructor to be called at the appropriate time. Determining the appropriate times is somewhat complicated, and the object designer must still make a detailed analysis of memory usage to be sure that a large program will function properly.

In C, memory management is completely under the programmer's control. This has the advantage that no overhead is incurred, but forces the programmer to analyze memory usage and devise schemes for reclaiming unneeded memory.

I feel that a systems-programming language is designed to serve specific needs. Operating systems, compilers, and related programs share certain common features. More so than most other forms of programming, systems programming is dominated by data structures. Computations rarely amount to much more than counting, but there is a large amount of testing and moving of data. Memory management is a significant and direct part of the programming problem.

Any given automatic memory management strategy may be appropriate for some problems, but for others the strategy may incur substantial penalties of excess memory usage, slow execution, or inappropriate pauses. Because of the importance of memory management, I believe that the correct approach to the problem is exactly what C has done. Parasol provides no automatic memory management procedures at all.

I believe that destructors are not an effective mechanism for systems programmers either. What they introduce is unpredictable overhead. The object designer may have some idea of what destructors there are, but the user of a class is led to believe that instances will take care of themselves. So, for a programmer to have some understanding of how memory management is being handled, the programmer must discover through trial and error how the destructors are being introduced. Mechanisms like `setjmp` and `longjmp` that violate the strict stack discipline completely invalidate destructors.

What Parasol does is to establish a convention that a public method should be defined for each object, called `dispose`, that is responsible for cleaning up the object. Instead of being inserted automatically, the programmer must simply be aware that the `dispose` method is part of the usage of the object. It is equivalent to the need for calling `close` when you are done with a file.

I believe that the cost of remembering to call dispose before freeing memory is small relative to the benefit of making explicit any such overhead. If Parasol were intended for business applications programming, or for a casual

programming environment, then garbage collection of some form would make much more sense.

Parasol does not solve the problem of releasing allocated memory around `setjmp` and `longjmp`. I believe that at least since cleanup is explicitly done, the programmer has a better idea of what cleanup needs to be handled.

Operator Overloading

I decided not to support operator or function overloading. In SmallTalk, which effectively supports such features, the cost in the compiler is small because there is no compile-time typing, no operator precedence, no need for generating conversions. Conversions, if any, are performed within the code being executed for each operator. In effect, in SmallTalk operators are simply non-alphabetic spellings for certain commonly used methods.

Because the expression syntax of Parasol incorporates most all of the rules of C for precedence and conversions, operator overloading would only be sensible if conversions could be generated automatically. That would mean that a user would have to be able to describe type conversions, perhaps as methods in a class definition.

Deciding what to leave out of a language is more difficult than deciding what to put in. As a user, any feature that can be exploited usefully is attractive. As a programmer I don't have to worry about abusing a feature, because I know I will use it properly. Operator overloading can be a valuable tool for defining new mathematical objects that are not in the basic set of data types supported by the language. At that point, the same algebraic notation used for built-in types becomes available to the class designer for new types. That said, I believe that operator overloading is so fraught with potential abuse that I did not implement it.

What makes the standard algebraic functions so flexible and powerful a notation is that centuries of mathematical experience have gone into refining the notation. Each of the numerical types that has an addition operator has been carefully defined so that as many of the properties of addition are preserved for all types that can be added. Operator overloading in programming languages almost never carries any constraint on the overloaded definitions. Some of the properties, such as commutativity, are even impossible to verify in general.

The input-output operators of C++ are a prime example of what I would consider inappropriate overloading. There is no mathematical relationship between binary shift operators and input-output. If compact notation for `printf`-like I/O was desired then I think an entirely new pair of operators would have been a better idea.

I think that the C++ I/O operators are illustrative of the tendency to overload unrelated meanings onto symbols. It is a general problem in programming. A programmer writing a large program must devise hundreds and thousands of names for objects, functions, types, and constants. Even though most programmers adopt some sort of personal naming conventions to distinguish the various

concepts being named, and programming teams often devise group-wide or even company-wide naming conventions, it is next to impossible to make sure that the same word is not used to mean different things.

The use of meaningful words and the willingness to distinguish long identifiers minimize those sorts of problems. (Parasol distinguishes 32 character identifiers throughout.) The relatively short list of specialized operators available for overloading invite abuse. Allowing the programmer to define new operator tokens is little solace since arbitrary combinations of abstract symbols lack the mnemonic content of an identifier.

Automatic conversion in the presence of overloaded operators is still more complicated. The conversion rules of C have been carefully crafted to preserve the value being computed. By strictly ordering the arithmetic types and demanding that widening the narrower type is the universal rule for all operators, potential harm due to inappropriate conversions is minimized. Since Parasol uses prototype-like function syntax, a common source of errors in C has been avoided. Bugs or inconsistencies in the specification of conversions and overloaded definitions can lead to some very unpredictable results.

I believe that readers and maintainers of programs are much better served by retaining as much overt meaning as possible in the program source. Already, you must look several places in a program to know completely what an expression does. Even so, since C operators have a certain uniformity of behavior, the general meaning of an expression is often obvious. With operator overloading, each token can have a unique meaning.

Parallelism

Parallel programming is a complicated subject. Hardware has been built over the years to support parallelism in a number of fundamentally different ways. Parasol does not try to provide support for all of these approaches. I have examined the kind of programming Parasol currently encourages and I have decided that the best marriage of parallelism and Parasol is fairly specific. Parasol supports coarse-grained parallelism such as that found in multi-processors and distributed memory multi-processors.

The basic model that Parasol uses assumes that the system being used is what is called a multicomputer. In a multi-computer some number of processors are each connected to a separate RAM memory. Each processor has exclusive access to anywhere from one to many megabytes of memory. The processors themselves communicate over some sort of network using variable-length messages.

This hardware environment actually describes local area networks of workstations as well as supercomputer-class multicomputers. Local area networks have the potential for a large computing resource. The Next workstations for example support an application called 'Zilla that can manage a whole group of processors for long overnight runs. It has been used to contribute original

knowledge by computing some mathematically significant numbers.

With Parasol I have attempted to address the programming needs of such an environment as just described. The central concept in Parasol's approach to parallel programming is to treat each object as a concurrently running entity. In fact, the overhead of running a process for each object in most hardware would be prohibitive. As a result, Parasol programs only create processes for selected objects. Most objects remain passive entities that are called upon by the active objects.

Some aspects of the parallel programming support are still being designed as I write this. As a result, fewer concrete things can be said about the parallel constructs of Parasol. I cannot, for example, report much about the level of performance of these concepts. I do have the basic messaging functions working, and they appear to be fast enough to make an operating system written in Parasol and using messages slightly faster than MS-DOS running on the same hardware.

Arenas

Objects in Parasol exist within arenas. An *arena* is a self-contained region of memory in which all addressable memory locations have unique addresses. A sequential program runs in a single arena, while a parallel program runs in several arenas.

Parallelism is achieved by combining two different strategies. In one strategy (which has been traditionally associated with parallel programming) the programmer writes a single program source that specifies how many processors should work on a single task. A very common approach in this style of program is to divide a problem into many distinct slices of data. Each slice is manipulated using the same logic. I call this strategy *replicative distribution.* Massively data-parallel computers are specifically designed to solve this kind of problem.

In the other strategy, distinct programs are written with published interfaces. New programs can be added later that call on existing code to perform work. I call this *functional distribution.* Graphics or I/O co-processors are good examples of hardware designed to support functional distribution. Operating systems, through their system-call interfaces are good software examples of functional distribution.

Parasol is designed to support either style of distribution. Replicative distribution is achieved by creating many arenas, each of which has essentially the same logic. Functional distribution is achieved in two ways. First, programs can be separately linked and special interface library units exported from one program to another. Second, a programmer can construct subtypes derived from arena type. Code and objects can be associated with arena-derived types, thereby avoiding the problem of replicating everything.

Arenas are represented by arena objects within a Parasol program. Arena objects may be declared as any other objects are. Since they can be dynamically

created, a Parasol program can vary the number of different processors it runs on. With some effort, it can do this as it is running.

When an arena is created, it is dormant. It is activated by attaching it to a processor. By default, each Parasol program starts with one active host arena on some processor. Typically, a parallel program will ask the operating system for a processor set, called its *team*. Programs can also ask for processors piecemeal, but since allocating processors is a somewhat time-consuming task probably involving communications over the network, allocating teams is recommended.

An active arena can allocate objects dynamically, like the local heap. Instead of returning a normal pointer, however, an object allocated in an arena returns a far pointer. A *far pointer* is simply a pointer that has been augmented to include not only the object's address within the arena, but also the arena's name as well. Since the encoding of far pointers is operating-system dependant, such pointers should be regarded as unique 'things.'

Functions and static objects exist in all arenas of a given type. The entry functions of an arena type are executed only when an instance of that arena type is started.

Unless specified otherwise, all code and static data is assigned to an arena type called `hostArena`. Each program is started with one instance of `hostArena` type running. This instance is named `HostArena`. You can specifically declare an object to be part of a specific arena type. For example:

```
outputArena:   type inherit arena {};
inputArena:    type inherit arena {};

Infile:   static.inputArena ref stream;
Outfile:  static.outputArena ref stream;

Output:   outputArena;
Input:    inputArena;

main: entry () =      // hostArena code
{
        myTeam:   ref team;

        myTeam = team create(2, 2, PT_VIRTUAL);
        myTeam start(Output);
        myTeam start(Input);
        ...
}

omain: entry.outputArena () =
{
        ...
}
```

```
imain:    entry.inputArena () =
{
        ...
}
```

This code creates two named arena types, which will presumably be responsible for I/O activities. The `Infile` and `Outfile` stream pointers are then allocated to their respective arenas. Two arenas are then created to be instances of their respective arena types. The same main-line code then allocates a team of two processors (the two arguments are the minimum and maximum number of processors needed by this program).

The `PT_VIRTUAL` flag indicates that if enough real processors are not available, the system should create virtual processors to run the arenas. A virtual processor is simply a separate task running on some real processor. You may choose to wait for enough real processors to become available, or you can choose not to wait and simply abort the call if not enough processors are available. Management of processors and tasks is a very operating-system dependent mechanism.

The start calls then allocate a processor from the team to each arena. The correspondingly designated entry functions then are started for each respective arena. Because the above code is willing to work with virtual processors, this code will actually function on a single-processor computer. This sort of breakdown in a program may make sense when the inputs and outputs of a program need significant format conversion. On a single-processor machine this program will function like a UNIX pipeline.

Note that specific checks are not needed in this sample code. If these calls fail, an exception is raised. Since this code does not catch exceptions, any exception raised will abort the main process. When a process aborts, any running team members attached to it are also terminated. Exceptions are explained below.

Messages

The compiler must analyze the code to determine which references to data are local to the arena and which must be resolved across arenas. A reference to a static object in the same arena type will access the local copy of the object. A reference through a simple pointer is also assumed to refer to a local instance. References to static objects attached to a different arena type, or references through far pointers, are assumed to need operating-system assistance.

Any call, either a member call or a call to a simple function, across arena boundaries is resolved using messages. By default a Parasol message is treated like a remote procedure call. A message is sent and the calling process waits until a reply is received. The source code looks exactly like a normal function call. Breakdowns in the communications network, an abort in the called arena, or some other failure will generate an exception.

Asynchronous sends will probably be added, but certain problems must be addressed (such as how to obtain the reply value). The low-level operating-system interface allows for the creation of a future (actually a message id handle) that is used later when the process finally wants to wait for the reply. The problem is that to the operating system, futures all look alike. There is no constraint on the reply value when it finally arrives. In Parasol code, one would want a future to be constrained so that it can only accept a reply value of the same type as the function originally called.

I may also add a capability to send a message with no reply at all. Void functions still have a reply (with no data in it) for synchronization purposes. Occasionally, one may want to generate a message such as a mailgram or low-priority log message where you don't care about any reply. In such a case, the caller is not stopped at all and no exception can be fed back to the caller. Currently, that is not possible.

Exceptions

I have mentioned exceptions a couple of times so far without explaining them. Parasol exceptions are modeled on the exception mechanism detailed in the language CLU. The syntax currently used incorporates ideas from the syntax employed in C++.

Operating-system code, hardware, or programmer-written code can raise exceptions. These are synchronous, unexpected events usually associated with errors in a program. Exceptions are coordinated through objects of type `trap`. Currently, code raises an exception by calling a trap object. Code also catches exceptions by catching the same object.

A trap object contains hardware specific state information (like the contents of machine registers) needed to diagnose a problem.

The existing syntax for catching an exception uses a new kind of statement called the `try` statement, which exists in two forms:

```
try
        statement
except
        statement
```

or

```
try
        statement
except ( expression )
        statement
```

The second form has an expression that names a pointer object. This pointer is a pointer to trap type and when an exception occurs, the address of the offending trap is stored in the pointer for future reference.

When a `try` statement is executed the first statement clause is executed. If it completes without any exception being raised, the second statement clause is skipped. If an exception is raised during the execution of the first clause, then the second clause is executed.

The second clause is like a switch in that it is typically a block with one or more cases and possibly a default case. Instead of each case having an integral value, as a switch does, a case in an except clause has an expression that names a trap object. If the exception that was raised matches one of the traps named in a case, control is transferred to that case. Otherwise, if a default case is present, control transfers there. If no case matches and no default case is present, the exception is propagated out to the next `try` statement in the call stack. `break` statements can be used to drop out of an except clause just as in a switch.

When a `try` statement catches an exception it can continue executing, raise a new exception, or re-raise the trap that was just raised. Re-raising a trap preserves the original execution context for some outer `try` statement.

Currently, enough context is preserved during an exception to be able to restart code after an exception. I have not actually written any code that needs to restart from an exception, and others have raised significant theoretical problems with restarting after an exception. I have no fundamental objections to supporting restartable exceptions, although there may be difficulties in supporting such a thing on some hardware.

I am strongly considering a revamp of the exception mechanism so that `try` statements catch types, rather than objects. There are a number of advantages to such a scheme, and the cost of finding a match in a case is little different.

So far, the Parasol exception mechanism has proven to be fairly easy to implement and provides a simple degree of control over error handling. Overhead is little different from the `setjmp/longjmp` mechanism of C and is considerably more structured. Whether the exception mechanism can withstand the stresses of more complex environments remains to be seen. In particular, when an exception must be propagated across an arena boundary there are numerous problems. Currently, there is no provision for propagating meaningful context across an arena boundary, and restarting after such an exception is likely to be impossible.

Processes

Each running arena has at least one process executing in it. A process in Parasol's terms is not the same as a UNIX process. (A UNIX process is actually both a Parasol arena and a Parasol process.)

When an arena is started, it has one process which executes the entry functions of the arena, if any. When any entry functions have been executed, instead of calling `exit` as the main arena does, if one of the entry functions does not explicitly call `exit`, a spawned arena remains intact and the executing process is suspended.

When a message is sent to an arena or to an object in an arena, any suspended process running in the arena is awakened to receive and process the message.

An arena can spawn additional processes, effectively acting as threads, to execute more than one message concurrently. A multi-processor can assign a separate processor to each process running in an arena. Library functions are supplied that synchronize processes within an arena.

Array Syntax

I made an early decision to remove the identity relation between arrays and pointers that is found in C. (This identity relation says that the name of an array used in an expression is converted to a pointer to the first element of the array.) As a result, the subscripting operator can be described as a simple abbreviation for combined pointer addition and indirection.

In Parasol, an array name in an expression has array type and is *not* converted to a pointer type. As a result, the subscripting operator is sensitive to the types of its operands. I added another constraint on subscripting that the expression inside the brackets must be the integral expression. The expression in front of the brackets must be a pointer or array type expression. The effect of the subscript operator is the same as that in C.

Array operands cannot be used in arithmetic expressions as they can be in C, however. Adding an array name to an integer is an error in Parasol.

I made this change to provide for an extension to support arrays as first class objects. I have not done anything to implement vector operations in Parasol, but the capability is there. A few things are supported in this version of Parasol. Array objects can be assigned like structures, and arrays can be passed by value in function calls and returns (just like structures).

One of the issues that arose from breaking the array/pointer symmetry in C was the type of `&array_name`. In C, the unary ampersand operator applied to an object yields a pointer to the object and has the type of "pointer-to-type-of-the-object." It turned out in Parasol, since an array name by itself was no longer a pointer to the first member, I needed expressions of the form `&a[0]` to produce the same effect. These were so frequent and tedious that I decided to change the rules for the unary ampersand so that when applied to array objects, it produces a pointer to the first member with the type of a pointer to member type. The only way to produce a "pointer-to-array-type" is through a cast. Such a cast has not been necessary in the code I've written so far, but literally hundreds of instances of `&a[0]` have been simplified.

In retrospect, these were not really the best choices. Since I was not adding vector operations to Parasol, there was very little offseting the inconvenience of writing code that scans arrays. I not only had to break the array/pointer symmetry to make extension to vector operations possible, I also had to gum up the address-of operator to make traditional C-like scanning and string manipulation code moderately convenient.

Scanning arrays with pointers in Parasol is just as efficient as C, so the issues here are lexical and somewhat aesthetic. I have source code that looks about the same as corresponding C code (a few extra ampersands don't intrude much), but I have a language definition with some unfortunate exceptions in the operator semantics.

Ironically, changing array syntax and semantics has given an open door to scientific and engineering programmers (provided vector extensions are actually implemented), while only complicating the lives of systems programmers (but not by much). Being a dyed-in-the-wool systems programmer I would have to consider this change a mistake.

Variable Length Arrays

One issue that arose early on in the process of adding distributed computing support to Parasol was the problem of passing data between arenas. Pointers are no longer a valid way of passing arbitrary-length data in a function call. A pointer simply does not contain enough information about the length of the array being passed to allow the compiler to pass information properly.

As a result, I added the capability of specifying variable length arrays in function calls, either as function arguments or as return values. In order to store return values, some way was needed to define an array object of variable length.

The following syntax is used to declare a variable length array:

```
name: [;length] sometype;
```

The length expression is the maximum length of the array. In function parameters the expression can be omitted. For function parameters, the length is obtained from the array argument that is passed in the call.

Note that the semicolon is a separate token. No expression appears in front of the semicolon.

A variable-length array object is stored as a descriptor followed by the array elements. The descriptor is a single unsigned integer containing the current length of the array. The space allocated is enough to hold the descriptor plus the maximum number of elements. For a function argument, a variable-length array is stored as a fat pointer, with the descriptor plus a pointer to the array elements.

When `sizeof` is applied to a variable-length array, the value is not a compile-time constant. Rather, it is computed at runtime based on the current value of the array length.

When doing array assignment, the length of the destination is adjusted to be the length of the source. Fixed-length arrays can be copied to variable-length arrays. A variable-length array can only be copied to a fixed-length array if the fixed-array is at least as long as the maximum length of the source. String literals (which normally have pointer to `char` type as in C) can be assigned to a

character array, and the length of the string is determined as in a C initializer. That is, the ending null byte is counted in the size of the array if the target array can hold it and it is not otherwise.

In order to allow pointers to participate in array assignments, Parasol also has the concept of array slices. The syntax of an array slice is as follows:

```
array_or_pointer[;expression]
```

This converts the array or pointer expression outside the brackets to an array with the length given in the bracketed expression. In a future extension it will be possible to include an expression in front of the semicolon to mean the starting index into the array.

A number of issues still remain to be resolved. Multi-dimensional arrays currently store a descriptor for each element of the sub-arrays. In other words, like C, in Parasol there are no multi-dimensional arrays, per se. Instead there are arrays of arrays. So a two-dimensional variable-length array is stored as a variable-length array of variable-length arrays. One of the by-products of this property is that individual rows can be of different lengths. Is this a good property? It consumes a lot of extra space for all the extra descriptors. Scientific programmers are more accustomed to thinking of matrices where each row has the same length.

There is a significant issue concerning dynamic allocation of variable-length arrays. For such a thing, the C convention of using a `sizeof` expression in the call to allocate an object would imply that `sizeof` should include the descriptor size. Since I have not prescribed the exact size of the integer descriptor it may be possible for an implementation to customize the descriptor size for the maximum length of the array. What this means is that there is no portable way to calculate the array dimension from the `sizeof` expression, as can be done for fixed-length arrays. All this implies that Parasol may need a `rankof` operator that returns the actual dimension of an array object, not a size in bytes.

Miscellaneous Changes

There are a few changes in the syntax of Parasol that can hardly be called compelling. They are small changes to the C syntax that are intended to improve minor aspects of the language. They could easily have been avoided, but since I was already committed to a non-compatible language there was no overwhelming reason not to change things.

Enumerators

Parasol has no `enum` type. Instead, when defining a named type, you can include a set of constants that are the equivalent of an enumerator type. For example:

```
enum_a: type short = {
        ENUM_0,
        ENUM_1,
        ENUM_4 = 0x04
};
```

This declaration defines a named type `enum_a` and three constants. The enumerator constants appear in the initializer part of the declaration (since types otherwise do not have initializers). Like C enumerators, the first name is assigned zero and each subsequent name is assigned a value one greater than the previous one. Also like C, an enumerator can have an explicit value assigned. Each constant has the named type.

I added this capability to satisfy a concern voiced during the process of standardizing C. Several proposals were put forward to allow for control of the size of an enumerated type. The syntax I have defined here covers this capability quite nicely.

This syntax has the advantage that even floating-point types can be defined with associated constants.

The `offsetof` Operator

Since Parasol has no preprocessor, the C `offsetof` macro cannot be used. Instead Parasol defines an `offsetof` operator. The exact syntax follows:

```
offsetof type_name.member_name
```

The `type_name` must be a structure type and `member_name` must be a non-static member. This form is limited to simple members. Subscripted array members or nested structures cannot appear in this operator. It would be possible to extend the `offsetof` macro so that the right side of the dot could be any valid addressing expression involving dots and subscripting (with no restriction on the subscript expression). Such an `offsetof` expression would be a constant if all subscript expressions are constant. I haven't expanded the capability of this operator because it would require a considerably more sophisticated internal treatment of the expression.

The need for this operator came up in connection with a data-entry package. Admittedly, this operator is not in the spirit of object-oriented programming (in which members are hidden), but the available OOP mechanisms were awkward. A package that must access data in a generalized way and map between external data formats and internally defined structures has a need for this capability. The C address operator (`&`) needs an instance of the structure to bind, which a general data dictionary entry may lack.

Operator Precedence

Perhaps the least important change I made in Parasol was to alter slightly the precedence of a few of the operators. At some early stage of the development of Parasol I recalled a statement in the original K&R about the shortcomings of C. It mentioned in passing that the operator precedences weren't right, but didn't elaborate.

I decided to try an experiment. I analyzed the operator precedences of C and how I wrote C code. The following table compares the operator precedences of C and Parasol:

Comparison of Operator Precedence

C	*Parasol*
`* / %`	`* / %`
	`<< >>`
`+ -`	`+ -`
`<< >>`	
	`&`
	`^`
	`|`
`< > <= >=`	`< > <= >=`
`== !=`	`== !=`
`&`	
`^`	
`|`	
`&&`	`&&`
`||`	`||`
`?:`	`?:`
`*=`, etc.	`*=`, etc.
`,`	`,`

The first change is to move the shift operators to a precedence just below that of addition. An expression of the form `a<<b+c` is usually intended to group as `(a<<b)+c`. In C, because of the precedence of shift, the actual grouping is as `a<<(b+c)`. Since the shift operator is often used as an optimized form of a multiply or divide by a power of two, it seemed more reasonable that it bind more like the multiply and divide operators themselves.

The other change was to raise the bit-wise operators as a group above the comparison operators. Again, experience tells me that bitwise operators are rarely, if ever, used as booleans. The `&&` and `||` operators have the much more commonly desirable property of short-circuit execution. As a result of this change, bit testing expressions like the following need no parentheses in Parasol:

```
if (field & BIT_MASK == 0) ...
```

In C this would group as:

```
if (field & (BIT_MASK == 0)) ...
```

which is rarely a desired grouping.

Was changing operator precedences important? No. Is it going to get in anybody's way when learning Parasol? I doubt it. Bit mask testing is easier to write, and if I forget and use my old C habits and parenthesize these expressions it doesn't hurt. Going from Parasol back to C might cause some problems.

I maintained a C compiler for seven years that printed a warning if it ever encountered these operators together without parentheses. I never got a complaint that users liked writing code without the parentheses. My experience is that between addition and assignment, people don't usually remember the precedences well enough and parenthesize liberally.

The Future

This article is a snapshot of a language in evolution. I have been making design changes even as I have been writing this article. I have tried to be intellectually honest by stating what I actually have designed. I have tried to indicate where the rough edges of the language are and where it is changing most.

At any moment throughout this project I have held a vision of how the entire Parasol language and environment should interact. That vision has shifted over time. The original vision was definitely rooted in my experience with Turbo-C. The addition of parallelism has caused me to revise my vision in major ways.

The Parasol compiler I am now using is an integrated environment with an editor, compiler, and debugger linked into a single executable program. This integration means that it is very fast, but it is also very closed. Adding extensions means recompiling the environment.

I concluded some time ago that a truly powerful development environment must admit the addition of extensions. The use of messages exchanged between separate programs offers the low-level mechanisms for designing such an extensible environment. So the new Parasol development environment will be centered around a project-description data base. The debugger, compiler, editor, and browser plug into that database in various ways. I am exploring ways to generalize the project database structure so that third-party modules can be added.

This environment is still very much vaporware. Nevertheless, it is the needs of this environment that are driving the current evolution of Parasol. Depending on the overall efficiency of the system, I am even looking into the idea of parallel compilation. Source files can be parsed into syntax trees independently of one another and functions can be compiled into object code without regard to other functions. The potential is tantalizing.

If Parasol is to deserve a place among the programming languages people

use every day, it must provide capabilities that other languages lack. The OOP extensions, the units, and the syntax changes of Parasol are nothing really unique. It is the parallel programming features of Parasol that will really contribute something of value.

Conclusion

How has Parasol worked in practice? Going to all the trouble of modifying syntax and adding features isn't worth it unless you get some benefits.

The object-oriented extensions have worked out very well. Parasol offers better encapsulation of data and polymorphism has proven to be a valuable way to write code that would be awkward to express in Standard C. If you believe in the value of object-oriented programming, Parasol gives it to you in a manner that does not involve a lot of complexity in the language or the compiler.

Incremental compilation has proven more elusive. The problem of minimizing the computation required to rebuild a program after changes have been made in the source has been very difficult to conquer. The syntax changes have helped in some ways, but there is an intrinsic complexity to programs that makes incremental compilation a challenging task.

Parasol has managed to preserve the low runtime overhead and efficient execution that C enjoys. The techniques used by C programmers for writing efficient expressions, loops, data structures, and functions still apply with Parasol. The data model employed by Parasol means that code generation on most machines should be about the same as for C. A few of the semantic changes involving aliasing will probably improve code generation on many machines.

The parallel constructs are new, and given the limited experience of wedding object-oriented and parallel-programming concepts in the research community, I expect that there will be considerable evolution in this area of the language. I have tried to keep Parasol simple enough so that evolution will be possible.

I believe that Parasol is successful from a technical perspective. Commercial success is impossible to predict. The factors that decide whether a given language will prosper often involve issues well outside the technical arena. Who knows, in ten years there may be a *Journal of Parasol Language Translation*.

Bob Jervis was author of the Wizard C compiler and the chief architect for Borland's popular Turbo C compiler. He is now self-employed and is developing software using Parasol. Bob can be reached at uunet!bjervis!rbj.

∞

35. Miscellanea

compiled by **Rex Jaeschke**

Calendar of Events

- April 29–May 2, 1992 **Workshop on Parallel Compilation** – Location: Kingston, Ontario Canada. Researchers are invited to submit an extended abstract of about 2,000 words by January 13th 1992. Invitations will be made by February 15th, 1992, and final versions of papers will be required by March 15th, 1992. Workshop Chair: David Skillicorn (613) 545 6050, Fax: (613) 545 6453, or *skill@qucis.queensu.ca*.

- May 11–12, 1992 **Numerical C Extensions Group (NCEG) Meeting** – Location: Salt Lake City, Utah. For more information about NCEG, contact the Convenor Rex Jaeschke at (703) 860-0091 or *rex@aussie.com*, or Tom MacDonald at (612) 683-5818 or *tam@cray.com*.

- May 13–15, 1992 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Salt Lake City, Utah. WG14: Contact the US International Rep. Rex Jaeschke at (703) 860-0091, or *rex@aussie.com*, or the Convenor P.J. Plauger at *uunet!plauger!pjp* for information. X3J11: Address correspondence or enquiries to the Vice Chair, Tom Plum, at (609) 927-3770 or *uunet!plumhall!plum*.

- May 24–26, 1992 **X3H5: Parallel Model and Language Bindings** – Location: Ithaca, New York.

- Jun 22–24, 1992 **Intensive Tutorial on High Performance Compiler Back Ends** – Location: Portland, Oregon. Contact *mwolfe@cse.ogi.edu* for technical inquiries. For registration and other information contact *lpease@admin.ogi.edu*.

- Jul 6–10, 1992 **Intensive Tutorial on High Performance Compilers** – Location: Portland, Oregon. Contact *mwolfe@cse.ogi.edu* for technical inquiries. For more information contact *lpease@admin.ogi.edu*.

- Jul 12–17, 1992 **Joint ISO C SC22/WG21 and X3J16 Meeting** – Location: Toronto, Canada. Contact Dmitry Lenkov at *dmitry@cup.hp.com* or (408) 447-5279 for information.

- Aug 26–28, 1992 **Fourth International Symposium on Programming Language Implementation and Logic Programming** – Location: Leuven, Belgium. Contact *maurice@cs.kuleuven.ac.be* for information.
- Sep 20–Oct 2, 1992 **Second Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors** – Location: Boulder, Colorado. Send abstracts to Carolyn Mich at *carolyn@cs.colorado.edu* by May 8.
- Oct 5–7, 1992 **International Workshop on Compiler Construction** – Location: Paderborn, Germany. The presentation language is English. For more information contact P. Pfahler at *peter@uni-paderborn.de*, telephone 05251/603069, or Fax 603836.
- Nov 8–13, 1992 **Joint ISO C SC22/WG21 and X3J16 Meeting** – Location: Boston, Mass.
- Dec 7–8, 1992 **Numerical C Extensions Group (NCEG) Meeting** – Location: Washington D.C.
- Dec 9–11, 1992 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Washington D.C.

News, Products, and Services

- The U.S. Department of Commerce National Institute of Standards and Technology **(NIST) has announced the beginning of its C Validation Service for the FIPS 160 C Standard**. They are now accepting requests to schedule a validation. Scheduling will be based on the order in which requests are received. Applicants must have a valid license for use of Perennial's ACVS suite by the scheduled test date.

  The validation service is available to both government agencies and industry organizations on a cost-reimbursable basis. Demonstrated compliance to FIPS 160 will result in the issuance of a Validation Summary Report (VSR), a Certificate of Validation, and inclusion in the quarterly Validated Products List (VPL). These items are all used by government agencies to confirm that the requirements for FIPS 160 have been met. The earliest date for certificates will be April 1, 1992.

  The official test suite for validation of C compilers is the ACVS, Version 3.0 available from **Perennial**, 4699 Old Ironsides Drive, Santa Clara, California 95054, telephone (408) 748-2900, E-mail *uunet!peren!acvs*.

  For further information or to request an information package, contact Kathryn A. Miles, NIST, Building 225, Room A266, Gaithersburg, Maryland 20899, telephone (301) 975-3156, Fax (301) 590-0932. Her E-mail address is *miles@ecf.ncsl.nist.gov*.

- Release 5.00 of **SAS/C** for IBM mainframes is now shipping for MVS and CMS. Contact the SAS Institute at (919) 677-8000 or Fax (919 677-8123 for more information.

- Microsoft Press has released *ANSI and ISO Standard C: A Programmers Reference,* by **P.J. Plauger and Jim Brodie**. This is a rerelease of their earlier *Standard C* in a larger format. It also corrects known errors and adds more diagrams and explanatory material.

- The **C Model Implementation** developed by Knowledge Software Ltd., for verifying the European C validation suite (and now the U.S. one as well) is available to end-users. For information within North America contact Plum-Hall at *plum@plumhall.com*. For other countries contact Knowledge Software at *derek@knosof.uucp*.

- Rational Systems, Inc., is shipping version 5.0 of their **Instant-C** development system for PCs. Contact them on (508) 653-6006, Fax (508) 655-2753.

- Professional Press has released *The Dictionary of Standard C* by **Rex Jaeschke**. This book can be bundled with a compiler manual set to supply or augment the glossary of terms. For information about quantity discounts contact Annette Nelson at (215) 957-4287. Her E-mail address is *nelson@proeast.propress.com*. To get information about licensing the book (or his **Standard C Quick Reference Guide**) in electronic form for direct inclusion into a manual set or on-line database, contact the author, Rex Jaeschke, at (703) 860-0091 or *rex@aussie.com*.

∞

36. Cumulative Index

Index by Volume and Article

Sample Issue – March, 1989

Vol 1, Num 1 – June, 1989

Vol 1, Num 2 – September, 1989

Vol 1, Num 3 – December, 1989

Vol 1, Num 4 – March, 1990

Vol 2, Num 1 – June, 1990

Vol 2, Num 2 – September, 1990

Vol 2, Num 3 – December, 1990

Vol 2, Num 4 – March, 1991

Vol 3, Num 1 – June, 1991

Vol 3, Num 2 – September, 1991

Vol 3, Num 3 – December, 1991

Vol 3, Num 4 – March, 1992

Index by Author

Each entry in this section contains a cross-reference of the following form: *1/4* **39.** 300. The first column represents the volume and number (in this case *1/4*). The second column represents the article number (in this case **39.**) and the third column represents the page number (in this case 300).

∞