The Journal of

C Language Translation

Volume 1, Number 2

September, 1989

Publisher and Editor .... Rex Jaeschke
Technical Editor .... P.J. Plauger
Standards Editor .... Jim Brodie
Numerical Editor .... Tom MacDonald
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 1810 Michael Faraday Drive, Suite 101, Reston, Virginia 22090 USA. Telephone (703) 860-0091. Electronic mail address via *uucp* is *uunet!aussie!jct*.

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

11. Translating Pascal to C

P.J. Plauger

Abstract

The programming language C has largely displaced Pascal, at least in the commercial marketplace. As a result, interest has reawakened in the mechanical translation of Pascal to C. This article discusses the major technical problems that arise in constructing such a translator. It outlines solutions that have proved effective. It ends with a discussion of "make versus buy" considerations.

Introduction

When I began writing commercial C compilers a dozen years ago, Pascal was just coming into its own. At Whitesmiths, Ltd., we were relieved when potential customers stopped asking, "Why should I program in C instead of assembly language?" Then they started asking, "Why should I program in C instead of Pascal?" I solved the problem, from the viewpoint of Whitesmiths at least, by writing a Pascal-to-C translator.

That let us sell Pascal compilers to those who demanded them, and at a premium price. It also exposed all those Pascal zealots to a good quality C compiler. The subliminal message was that you could do in C anything you could do in Pascal, and then some. The only residual issue was which language better suited your taste for readability and error checking.

We sold a lot of Pascal-to-C translators as front ends for Pascal compilers. We sold a few more to those who wanted dual language systems that would let them freely mix Pascal and C. We sold only a few to people who wanted to convert their trousseau of Pascal code once and for all to C.

After awhile, however, we stopped selling Pascal almost completely. C won the battle for the hearts and minds of commercial programmers. Pascal freaks either threw in the towel, converted to Ada, or became Modula-2 freaks. I assumed the Pascal market was essentially dead, until very recently.

Lately, I have heard more and more people talking about writing Pascal-to-C translators. Perhaps the pending approval of the ANSI C standard is the cause. Maybe it's something in the water. For whatever reason, the folks now eager to convert all their Pascal to C are busy reinventing an old and well worn wheel.

One of the charters of *The Journal of C Language Translation* is to keep readers abreast of the technology used in converting other languages to C. If translating Pascal to C has become important once more, I am happy to share what I learned over the past seven years. If you are interested in translating a language similar to Pascal, you might pick up a few pointers as well. You might even learn enough to decide to buy existing technology instead of reinventing it.

What's Easy and What's Hard

Most of the job of translating Pascal to C is straightforward. Writing a recognizer for the Pascal language has been an undergraduate exercise for over a decade. Writing a code generator that utters C instead of assembly language or binary actually makes the job easier. It's just a bit tougher if you want to produce highly readable C, but still a relatively easy task. You don't even have to worry about optimizing the Pascal, as a rule, since C compilers do that job rather well these days.

Each scalar data type in Pascal has an obvious equivalent in C. You may want to allow the user to map `real`s either to `float` or to `double`. You may want `MAXINT` to be too large to represent as type `int` (for C implementations with smallish values for type `int`). You may have to anguish a bit over how big to make the base type for a set of some subrange of integers. None of these issues adds serious complexity to the job, however.

A Pascal record is a C structure. A Pascal variant record is a C union. A Pascal array is a C array. A Pascal function can return only scalar types (a procedure being a function returning `void`). The only annoyance you have to deal with is passing array arguments by value. When that happens, you can pass a pointer to the first element of the array, as usual, and generate code that makes a local copy of the array. A smart translator can eliminate the copy if nothing alters the value stored in the array argument.

Expressions map almost one-for-one to similar expressions in C. You have to add an occasional type cast, such as when you divide two integers:

```
        I / J      becomes      (double)I / J
```

You should represent small sets (of up to 32 members, portably) as integer types in C. That way, you can use the bitwise operators to perform set operations very efficiently in C. One nuisance is manipulating large sets. I discuss that in the next section.

Another nuisance is dealing with references to variables outside of a nested function or procedure. Pascal not only lets you write one function inside another (recursively, of course), it lets you peer outside the nested function to the innards of any of its containing functions. Moreover, Pascal has rather subtle rules for determining which activation of a function you can see when a function has been called recursively. All of this machinery is alien to C, of course, and not easily expressed. I discuss some of the subtleties, and some machinery that works in C, in a later section.

Still another nuisance in Pascal expressions is the handling of `file` variables. The machinery built into Pascal for performing input/output is idiosyncratic, to put it politely. If you want ordinary Pascal files to behave nicely when connected to interactive devices, you have to play games. And if you want to close files properly no matter how a function or procedure terminates, you have to be very careful. I devote a later section just to the topic of files.

Like expressions, control flow also maps almost one-for-one to equivalent statements in C. The Pascal `case` statement is more structured than the C version, and the `for` statement is rather more restricted. Only the Pascal `with` statement has no direct analog, but that is more of a problem in symbol table management than flow of control. A `goto` to a label in the same function or procedure is no problem, but a `goto` out of a nested function or procedure is a royal pain. I discuss the problems, and a potential solution, after the other problems of nested functions.

Manipulating Large Sets

You can represent a set as an unsigned integer provided the integer has at least as many bits as the set has members. Standard C requires that type `unsigned long` contain at least 32 bits. A particular implementation may have an integer type with more bits, but you shouldn't count on it. Any set larger than 32 members you will have to represent as an array of some unsigned integer type. You choose `unsigned char` for maximum storage efficiency, `unsigned int` for maximum speed (most likely).

In Pascal, the translator must know the size of each set when it translates a set expression. It is therefore an easy matter for the translator to convert each set operation to an inline function call. The arguments are the size of the set, pointers to the operands, and possibly a pointer to the result area. You need the following functions:

1. Construct a set – that either is empty, has one member, or has a subrange of members.

2. Copy a set – from source to destination.

3. Compute set intersection – between two sets.

4. Compute set union – between two sets.

5. Compute set difference – between two sets.

6. Determine set equality – to yield a Boolean result.

7. Determine set inclusion – to yield a Boolean result.

8. Determine membership in set – to yield a Boolean result.

For maximum performance, allocate temporary storage for all intermediate values in a set expression at the start of the block containing the expression. You can't do this in a translator that generates output in a single pass over the Pascal source code, however. At the very least, you need some form of "delay line" that lets you percolate declarations to the top of the current block of C code. And if you want to minimize the amount of temporary storage you allocate, you need machinery for reusing temporaries when they are no longer in use. Unless you expect to translate code that manipulates large sets extensively, I recommend against such heroic measures.

The scheme I devised let the runtime set functions allocate and free operands on the fly. Each function had at most two argument pointers plus a code value. The code was a bit count, for a set whose size was rounded up to some multiple of 8-bit bytes. That reserved the low three bits for additional instructions to the set functions. For most of the functions:

- `(code & 01)` freed the right operand after the operation.
- `(code & 02)` freed the left operand after the operation.
- `(code & 04)` allocated space for the result before the operation.

You will find that it is easy for the translator to determine how to set these bits when it generates inline calls to the set manipulation functions. You will also find several opportunities for minimizing the overhead of allocating and freeing sets. A simple caching scheme adapts quickly to the one or two set sizes that most programs manipulate. After the first few set operations, the runtime functions allocate essentially all temporary set results directly from the cache.

Nested Functions

The first thing you have to do about nested functions is to unravel them. Since all procedure and function declarations must precede the first executable statement in the containing function, this is not terribly hard. Just don't emit any code that goes inside a function until the translator encounters the first executable statement. What you have to save about the function inside the translator is not onerous.

Preserving Pascal names in the C source presents a few problems. You must avoid any names that are keywords in C but not in Pascal. You must alter the names of contained functions that match names at outer levels. If you extend your Pascal dialect to support separate compilation of source files for a common program, then you may face additional problems. External names may have to be shorter than your Pascal names, in some C implementations.

And some names may collide with the vast number of external names defined in Standard C. Even if you disallow separate compilation, you must deal with a Pascal function named `main`.

If your goal is to produce highly readable C code, then you should work hard at preserving the original Pascal names (and comments, by the way). Build a dictionary of reserved names. Add to it as you encounter Pascal names that don't otherwise collide. Then map only those names that you know will collide. Whatever mapping scheme you devise should preserve the original Pascal name as part of the C name, to give later maintainers a sensible hint about its purpose. For example, `t13_idcheck` is worlds better than the bald `t13`. Even if you're willing to muck over all names, rather than check for collisions, you should follow this principle.

You should map variables at the outermost block level to C data objects with static lifetime. (The storage class should be either `static` or `extern`.) Typical Pascal programmers tend to percolate widely used variables to the outermost block and expect them to have quick access. Don't disappoint those typical programmers, if you value good code performance.

Variables inside contained functions have, of course, automatic lifetime. (The storage class should be either `auto` or `register`.) You still have to worry about their names colliding with C keywords, but otherwise the name spaces you end up populating in C are nicely disjoint.

To make these variables available to a contained function, gather them into a structure. Each contained function has a corresponding structure declaration, with a unique tag, that contains all of its local variables. The structure also contains a pointer to the structure for the containing function. Say, for example, that the function with structure `t11` contains the function `x` with structure `t12`. The function prolog looks like:

```
struct t12 {
        struct t11 *P;  /* pointer to outer frame */
        int lcl1;       /* local variables */
        .....
        };

int x(  /* declare return type */
        struct t11 *AP; /* frame pointer argument */
        float f,        /* declared arguments */
        .....) {
        struct t12 a;
        a.P = AP;       /* setup back chain */
        .....
```

With this scheme, you refer to all local variables within a function as members of the structure `a`. If you have to refer to the variable `abc` in the containing function, you write `a.P->abc`. If you have to refer to a variable two layers out, you write `a.P->P->abc`. And so on. Only when you reach to the outermost block do you use the shortcut `abc`.

Note the use of a secret "argument zero" to convey the Pascal display information. You must also pass this pointer along with the function address when you translate a Pascal function or procedure argument. A contained function must know its containing environment either at the time it is called or when its name is passed as a parameter.

If you find this dizzying (and you probably should), here is some homework for you. Near the end of the ISO Standard for Pascal is an "example" program called `GLOBALONE`. It is cribbed from the validation suite first developed at the University of Tasmania and now widely used to bless Pascal implementations. If you can understand what `GLOBALONE` is supposed to do, you will understand how nested functions work in Pascal. If you can get your translator to pass this test, you have probably implemented the machinery correctly.

Nonlocal Gotos

A Pascal `goto` can transfer control out of a function or procedure. The transfer of control also closes any files opened within the function containing the `goto`. You can jump out through several layers of function calls, terminating execution and closing files along the way. The expression containing the function call in the target function aborts execution, and control finally transfers to the relevant label. Try doing all that in C.

A translator can easily spot a nonlocal `goto`. You are obliged to declare any labels within a function in a `label` statement. The `label` statement precedes any contained function or procedure declarations. If a `goto` within a contained function is not to a label in the innermost list, it is nonlocal.

Spotting it and handling it are two different matters, however. If a function can be the target of a nonlocal `goto` (and you know this before you encounter its first executable statement), you must turn on quite a bit of machinery. The mainspring of this machinery is, of course, the pair of Standard C functions `setjmp` and `longjmp`. You add a `jmp_buf` (called `ENV` here) to the structure for any function that you can jump into (or through). The function then begins something like:

```
int VAL = setjmp(a.ENV);
switch (VAL)
        {
case 0:
        break;  /* go execute body of function */
        return a.x;     /* return function value */
case 17:
        goto 17;        /* first nonlocal target */
.....
```

```
        default:
                /* close any files */
                if (VAL == -1)
                        return a.x;  /* return function value */
                longjmp(a.ENV, VAL); /* else percolate out */
                }
```

With this preamble, a jump to label `17` becomes:

```
        longjmp(a.ENV, 17);
```

You return from such a function by writing:

```
        longjmp(a.ENV, -1);
```

This gives the function startup code the opportunity to close any files that have been opened within the function before it returns control to its caller.

Files

Closing all those files may not be so easy. Unless you build a list of opened files for each function, you have no way of knowing whether a given invocation of a function actually opens a file. For this and other reasons, I settled on a primitive but effective scheme for keeping track of Pascal files.

The runtime library maintains a list of open files. Each file is characterized by the address of its corresponding buffer variable, plus the size of its buffer in bytes. A `text` file corresponds to the Standard C notion of a text file. Its size is recorded as zero bytes. All other files are binary files in Standard C. Reads and writes to binary files are data transparent, but for integral records only.

What this scheme lets you do is close all the files opened by a function just by writing:

```
        pascal_close(&a, sizeof a);
```

What it costs is a lot of scanning of the Pascal file list to find matching buffer addresses. I found it to be a reasonable tradeoff, but you might not if you have programs that are highly I/O bound.

The other trick you should consider with Pascal input files is a device known as "lazy input." It seems that Wirth's model of file I/O evolved in a batch processing environment. When you open a file for reading (with `reset`), the buffer variable is filled with the first record of the file. That's fine for a disk file, but lousy for a keyboard. The runtime is always one record ahead when you try to read a file interactively.

Some implementations have extended Pascal by adding files that are explicitly interactive. You can get acceptable behavior within the rules of Pascal, however, simply by deferring reads as late as possible. Translate each attempt to read a file buffer to a function call. The function checks to see if the buffer is full and performs a read only if it is not. That way, you get a chance to put out a prompt before you demand the answer.

You can optimize this basic approach, of course. If the translator can determine the control information that goes with a file variable, you can generate macro calls instead of function calls. The macro tests a status flag, calling the function that performs the read only if the buffer is truly empty. A smart translator can even eliminate some of these tests, but I'm not sure it's worth the added complexity.

Pascal contains quite a lot of semantics about files, but most of it translates to C fairly easily. I found it simpler to write parallels to the `printf` and `scanf` function family in Standard C, rather than contrive appropriate formats for calls to the existing functions. Either way, the job is more tedious than difficult.

Conclusion

My goal in writing a Pascal-to-C translator was to accept an arbitrary Pascal program. Programs ill-formed (by the strict rules of the Pascal standard) generated diagnostics. All others generated valid C programs. I required that the translator never crash on arbitrary input. I also required that all diagnostics be generated by the translator, not further downstream in the C compiler.

Your goals may be less ambitious. Perhaps you have a single large corpus of code you wish to translate once and for all. You expect to encounter few or no errors. You want a mechanical aid, but expect to intervene to make the code more readable. If so, then you may want to leave out some of the nastier bits I have outlined here. Just diagnose those parts in the translator and fix up the output by hand.

My guess, however, is that such an approach will get you into trouble sooner or later. It's amazing how a "one off" job keeps getting done over and over again. It's also amusing how a project with narrow scope broadens, particularly if it is at all successful.

My advice is that you plan on working to the same specifications I did. Don't settle for a translator that accepts less than a complete dialect of Pascal. (I haven't even touched the important issues surrounding nonstandard Pascal extensions, which are pretty standard.) Above all, don't omit tests for bad code because the cases will "never happen."

You should seriously consider buying an existing Pascal-to-C translator, along with any necessary runtime libraries (preferably written in Standard C). I know that Intermetrics Inc., will license the source code of the translator I wrote. But you might find a standard binary package that meets your needs for much less. Or you might find someone else willing to license source if you have the overwhelming urge to customize the product.

Know that translating Pascal to C is conceptually very easy, but filled with any number of practical pitfalls. Whether you make or buy, approach the job with caution.

P.J. Plauger serves as secretary of X3J11, convener of the ISO C working group, and as Technical Editor of The Journal of C Language Translation. Dr. Plauger can be reached at uunet!aussie!pjp.

∞

12. Numerical C Extensions Group

Rex Jaeschke
Convener

At the first meeting of the Numerical C Extensions Group (NCEG) in Minnesota, we identified the hot topics and formed subgroups for each of them. Each subgroup has a primary (and for some a secondary) leader who is coordinating that group's activities. The intent is that anyone interested in one of these topics keep in touch with the leader(s) and that only the distillations of these groups actually be included in the formal mailings.

The subgroups established in May, and their leaders, are as follows:

NCEG Subgroups and Leaders

Topic	*Primary*	*Secondary*
aliasing	Bill Homer	
variable dim arrays	Dennis Ritchie	Tom MacDonald
array syntax	Frank Farance	Tom MacDonald
parallelization	Austin Curley	Frank Farance
complex	Tom MacDonald	Bob Allison
IEEE issues	Jim Thomas	David Hough
exceptions	David Hough	Jim Thomas
new math functions	Frank Farance	
ANSI math.h issues	Walter Murray	

They may be contacted as follows:

How to Contact Subgroup Leaders

Person	*e-mail*	*telephone*
Bob Allison	uunet!microsoft!bobal	(206) 882-8413
Austin Curley	curley@convex.com	(214) 497-4529
Frank Farance	decvax!farance!frank	(212) 486-4700
Bill Homer	homer@cray.com	(612) 681-5606
David Hough	dgh@sun.com	(415) 336-7702
Tom MacDonald	tam@cray.com	(612) 681-5818
Walter Murray	walter@hpda.HP.COM	(408) 447-6129
Dennis Ritchie	dmr@research.att.com	(201) 582-3770
Jim Thomas	uunet!apple.com!jimt	(408) 974-3266

For more information about NCEG, contact Rex at (703) 860-0091 or by e-mail at uunet!aussie!rex.

∞

13. Aliasing Issues in C

Tom MacDonald
Cray Research, Inc.
1345 Northland Drive
Mendota Heights, MN 55120

Abstract

Many modern optimization techniques, such as automatic parallelization, are difficult to apply to C programs because of the unconstrained nature of C's pointers. Complete interprocedural analysis may one day be the solution, but currently this kind of analysis is beyond our technological capability. More localized solutions are needed. Although scalar optimizations are also affected by unconstrained pointers, this article concentrates on parallel issues. The aliasing problem, and the advantages and disadvantages of several solutions, are examined in this paper.

Aliasing occurs when different expressions can reference the same object. Several examples of aliasing are shown in the following program fragment.

```
int a[10], i, j;
extern int *p;

/* ... */

a[j] = a[i];    /* alias if i == j       */
*p = a[i];      /* alias if p == (a + i) */
*p = i;         /* alias if p == &i      */
```

In many cases the compiler cannot determine if these expressions are aliases that reference the same object. Therefore, worst case assumptions must be made when optimizing. It should be noted that a `volatile` object is assumed to have aliases that the compiler cannot know about. Therefore, the compiler inhibits all optimizations with respect to that object.

A number of terms are used throughout this paper. *access* refers the value of an object is obtained, which is sometimes referred to as a *load.* The term *modify* means the value of an object is changed, which is sometimes referred to as a *store.* The term *reference* means that an object is either accessed or modified, and the term *lvalue* is an expression that references an object.

Considerable research has been conducted on how to use parallelism as a way to decrease the execution time of compute-bound applications. Automatic vectorization and multiprocessing are two common optimization techniques that have been exploited to extract parallelism from programs. Aliases must be resolved before a compiler can determine that a loop is safe to parallelize. Consider the following loop.

Example 1

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

void f() {
        int i;

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

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

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

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

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

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

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

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

Example 2

```
#include <stdio.h>

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

void blackbox(int *p1, int *p2, int *p3, int n);

main( ){
   int i;

   blackbox(c, b, a, 6);             /* no aliases */
   for (i = 0; i < 6; i++)
       printf(" c[%d] = %d ", i, c[i]);
   putchar('\n');

   blackbox(&a[1], &a[1], a, 5);     /* aliases */
   for (i = 0; i < 6; i++)
       printf(" a[%d] = %d ", i, a[i]);
   putchar('\n');
}  /* main */

void blackbox(int *p1, int *p2, int *p3, int n) {
   int i;

   for (i = 0; i < n; ++i)
      *p1++ = *p2++ + *p3++;
}  /* blackbox */
```

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

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

The function `blackbox`, whose definition starts on line 23, appears to add the corresponding elements of two arrays together, storing the results in a third array. This is exactly what happens when `blackbox` is called without any aliases at line number 12. The resulting array `c` contains the sum of `a` and `b`. This makes the loop inside `blackbox` appear to be a candidate for parallelization. However, when `blackbox` is called with aliases at line number 17 something different happens. Each element of the resulting array `a` contains partial sums of the values in the preceding elements. This time the loop must be executed as a scalar loop to obtain the correct results. Since `blackbox` is not declared `static`, it can be called from a separately compiled module. Therefore, the compiler must make the worst case assumption that this loop might contain aliases. It is interesting to note that FORTRAN does not permit aliasing through formal parameters if the actual object is modified. C, on the other hand, provides no way of restricting formal parameters that are pointers.

Aliasing and Standard C

Somehow the compiler needs to determine if two pointers are potential aliases. The aliasing rules specified in the ANSI C standard are defined in terms of an expression's type (§3.3 EXPRESSIONS). An object can be referenced by an expression only if the expression has the right type. The expression must be an lvalue and must have one of the following types:

- the declared type of the object,
- a `signed` or `unsigned` version of the declared type,
- a `const`-qualified or `volatile`-qualified version of the declared type,
- an aggregate (structure or array) or union type that contains a member with one of the aforementioned types, or
- a character type.

These rules allow compilers to make some assumptions about potential aliases. The following example contains a loop with pointers to different types.

Example 3

```
void f(int *p, int *q, double *a, int n) {
        int i;

        for (i=0; i<n; ++i)
                *a++ = *p++ - *q++;
}
```

An implementation is allowed to assume that the pointer `a` is not an alias with either `p` or `q` because the type of `a` is *pointer to* `double` and the types of `p` and `q` are *pointer to* `int`. Therefore, the assumption that the array elements modified through `a` are never accessed through either `p` or `q` is supported by the standard. That assumption permits vectorization or parallelization of this loop.

However, the aliasing rules do allow a *pointer to structure* or *pointer to union* to be an alias with pointers to the types of all their members (including, recursively, a member of a subaggregate or contained union.) The following example contains a pointer to a structure and a pointer to the type of one of the structure's members.

Example 4

```
struct st {int m1; /* other members */} *pst, x;
extern int *p;

/* ... */

        int i;

        for (i = 0; i < 10; i++) {
                pst[i] = x;
                *p++ = i;
        }
```

The pointer `p` is an alias with the pointer `pst` if the expression `p == &pst->m1` is true.

Character pointers can be used to access any object, which means a *pointer to* `char` is a potential alias with any object pointer. Therefore, any object can be viewed as an array of characters. A *pointer to* `void` is identical to *pointer to* `char` for aliasing purposes, except that a *pointer to* `void` cannot be dereferenced. The following example is a (probably) inefficient, but nonetheless portable way of doing a structure assignment.

Example 5

```
struct st {int m1, m2;} a, b;

char *pa = (char *)&a,  *pb = (char *)&b;

main() {
        unsigned long i;

        for (i = 0; i < sizeof(struct st); i++)
                *pa++ = *pb++;
}
```

Finally, it should be noted that aliasing is not restricted to "pointer meets pointer" combinations. A pointer `p` is an alias with variable `i` if the expression `p == &i` is true. This kind of potential alias affects scalar optimizations such as common subexpression elimination. However, implementations trying to perform automatic parallelization need a mechanism to resolve "pointer meets pointer" aliases. If an acceptable mechanism is found, it should also apply to scalar optimizations.

Directive to Ignore Aliasing

One common solution is to invent a directive that instructs the compiler to ignore any optimization problems that might arise from aliasing inside a loop. The loop inside the `blackbox` function (Example 2) might be written as follows:

Example 6

```
#pragma Ignore_Aliasing
for (i= 0; i < n; ++i)
        *p1++ = *p2++ + *p3++;
```

This directive tells the compiler to ignore any aliasing issues and vectorize or parallelize the loop anyway. It provides the user with a way to control which loops are optimized. However, it is not fully automatic nor portable, and if a harmful alias does exist the results might be incorrect.

Two Versions of a Loop

Another approach is to generate code that checks for aliases during execution. If no alias problems exist then a parallel version of the loop is executed, otherwise a scalar version is executed. For parallelization purposes, a harmful alias exists if either `abs(p1 - p2) < n` or `abs(p1 - p3) < n` is true, because elements modified in a particular iteration of a loop are accessed by a different iteration of the loop. (This check assumes a linear address space.) The compiler could turn the `blackbox` loop into something similar to:

Example 7

```
#define CHK(P,Q,N)      (abs(P - Q) < N)

if ( CHK(p1,p2,n) || CHK(p1,p3,n) )
        for (i = 0; i < n; ++i)
                *p1++ = *p2++ + *p3++;
else
        #pragma Ignore_Aliasing
        for (i = 0; i < n; ++i)
                *p1++ = *p2++ + *p3++;
```

which would check for harmful aliases and execute the appropriate version of the loop. The advantage is that the compiler can perform fully automatic parallelization, thereby eliminating the need for the programmer to add a directive. The biggest disadvantage is that the additional execution time overhead of performing the alias-check is introduced. Another disadvantage is that the executable image is bigger because of the alias-check and generating two versions of the loop.

Parallelization pays off only when a loop iterates enough times. If it iterates too few times, the additional overhead introduced by "starting up" a vector register or parallel processor increases the execution time of the parallel version of the loop beyond that of the scalar version. The smaller an implementation can make this Minimum Iteration Count (MIC) and still make parallelization pay off, the more loops that benefit from automatic parallelization. Generating the alias-check increases the MIC and decreases the number of loops that benefit from automatic parallelization.

Example 8

```
for (i = 0; i < n; ++i) {
        *p1++ = *p2++ + *p3++;
        *q1++ = *q2++ + *q3++;
}
```

The loop in Example 8 is slightly more complicated. More complicated loops require more complicated alias-checks to guarantee that no harmful aliasing exists.

Example 9

```
#define CHK(P,Q,N)      (abs(P - Q) < N)

if (CHK(p1,p2,n) || CHK(p1,p3,n) ||
    CHK(p1,q2,n) || CHK(p1,q3,n) ||
    CHK(q1,p2,n) || CHK(q1,p3,n) ||
    CHK(q1,q2,n) || CHK(q1,q3,n) ||
    CHK(p1,q1,n))

        for (i = 0; i < n; ++i) {
                *p1++ = *p2++ + *p3++;
                *q1++ = *q2++ + *q3++;
        }
else
        #pragma Ignore_Aliasing
        for (i = 0; i < n; ++i) {
                *p1++ = *p2++ + *p3++;
                *q1++ = *q2++ + *q3++;
        }
```

Example 9 demonstrates that the alias-check becomes quite complicated very quickly. The amount of execution time spent evaluating the alias-check of a complicated loop can easily increase the MIC beyond a useful value. Generating two versions of a loop is most beneficial for simple loops.

Disjoint Directive

Ideally, there should be some method of indicating that different pointers reference different objects. The idea is captured by the following two statements.

```
/*1*/   p[i] = p[i - 1];        /* explicit alias */
/*2*/   p[i] = q[i - 1];        /* potential alias */
```

In statement 1 an explicit alias exists. In statement 2 a potential alias exists. The concept that needs to be expressed is to state that `p` is not an alias with `q`. That is, the objects referenced through `p` are never referenced through `q` inside a certain section of the program. This could be accomplished by a directive such as:

```
#pragma disjoint (p, q)
```

The scope of the directive depends upon its location in the program.

Example 10

```
#pragma disjoint (p1, p2), (p1, p3)
for (i = 0; i < n; ++i)
        *p1++ = *p2++ + *p3++;
```

Again, pragmas are not portable, but this allows the compiler to detect explicit aliases and avoid generating the alias-check. However, it does become the programmer's responsibility to inspect the loop and completely specify all disjoint pointers. For a complicated loop this is not a simple task.

Example 11

```
extern double *apd[];

for (i = 0; i < n - 1; i++)
        for (j = 1; j < m; j++)
                apd[i][j] = apd[i + 1][j - 1]
                /* ^  potential alias here  ^ */
```

Example 11 declares an *array of pointers to* `double` named `apd`. In this example `apd` is used to access different arrays whose elements have type `double`. The `disjoint` directive must be expanded to allow expressions such as:

```
#pragma disjoint (apd[i], apd[i + 1])
```

to specify that no aliases exists between `apd[i]` and `apd[i + 1]`. This demonstrates that it is insufficient just to allow `disjoint` variable names.

Constrained Pointers

The ANSI C standards committee, X3J11, attempted to solve the unconstrained pointer problem with a new `noalias` keyword. This effort failed because of technical problems with the specification of `noalias` semantics, and insufficient time to correct these problems. However, the effort that went into the `noalias` proposal shed considerable light on the aliasing problem. Much of what follows is derived from that experience.

Example 12

```
extern n;
int a[100], b[100], c[100];

main() {
        int i;

        for (i = 0; i < n; i++)
                a[i] = b[i] * c[i];     /* no aliases */
}
```

The loop in Example 12 can be parallelized because a subscripting expression such as `a[i]` can never access an element of array `b` or of array `c`. Arrays are constrained by their bounds. A desirable solution to the aliasing problem is to allow a pointer to be constrained in a similar way.

An interesting perspective on subscripting is to count the number of references to objects.

```
int a[10], aa[10][10];
int *p, **pp;
```

Given the above declarations the following is true in the abstract C machine.

Expression	*Number of References*
`a[2]`	1
`p[2]`	2
`aa[2][3]`	1
`pp[2][3]`	3

In all four expressions only one identifier is present. When a pointer expression is used in a subscripting operation, the pointer object is accessed to obtain the base address and then the array element is referenced. However, when an array expression is used in a subscripting operation the base address is already known. Therefore, the only reference is to the array element itself. Another way to state this is that when an array expression is used in subscripting, the array element being accessed is known. When a pointer is used the element is not known because the base of the array is hidden in the pointer. An access through an array expression is a constrained access. An access through a pointer expression is unconstrained. This is why aliasing exists in C. C permits an object to be accessed through another pointer object.

The following concept needs to exist in C to constrain pointers:

An object referenced through an lvalue expression involving a pointer type is not the same object that is referenced through a different lvalue expression.

This concept must be defined such that the expression `p[i]` references a different element than `q[i]` but not necessarily different than `p[j]`. Part of this concept is already defined by the C aliasing rules that were previously discussed, (i.e., *pointer to T1* is not an alias with *pointer to T2*). Another part (*not* in Standard C) involves defining syntax that allows pointers to be declared as *constrained pointers.* Constrained pointers behave like arrays when used to reference objects. The final part involves the definition of when two lvalues are *different.* In this context different means specifying how the compiler can tell that lvalue *L1* must be referencing different objects than lvalue *L2.* The intent is shown in the following table.

`p[i]`	`p[j]`	same lvalues
`*p`	`**pp`	different lvalues
`a[b[i]]`	`b[i]`	different lvalues

Two lvalue expressions are different if they have different *handles.* The lvalues `p[i]` and `p[j]` have the same handle `p`, while the lvalues `*p` and `**pp` have different handles.

Handles

A handle is always the name of a data object (usually a variable.) For aliasing purposes, a handle only has meaning for lvalue expressions. The following is a list of lvalue expressions followed by the handles (if any) in the expression:

Expression	*Handles*
`x`	`x`
`(x)`	`x`
`"string"[2]`	handle is unique
`*(p - i)`	`p`
`a[i]`	`a`
`a[b[i]]`	`a`
`*(i < 0 ?  p :  q)`	`p`, `q`
`z.m`	`z`
`sp->m`	`sp`
`z.m1[i].m2`	`z`
`(z = y).m`	`z`

The handle in all these lvalue expressions is the identifier used to yield the address of the object. The handles of an lvalue are those identifiers found by recursive application of the following rules:

- If an expression is an identifier, the identifier is the handle.
- If an expression is a constant or function call expression, it contains no handle. If an expression is a string literal, its handle is unique (and secret.)
- If an expression is a parenthesized expression, cast expression, or expression with a unary operator, the handles (if any) are contained in the expression operand.
- If an expression is a conditional expression, the handles (if any) are contained in both the second and third operands.
- If an expression is an assignment expression or member access expression, the handles (if any) are contained in the left operand.
- If an expression is a comma expression, the handles (if any) are contained in the right operand.
- Otherwise, an expression is a subscript expression or an expression with a binary operator, and the handles (if any) are contained in the operand with pointer type.

With these rules, the handle of an expression such as `z.m1[i].m2` can be found. This expression is written in the following prefix notation with the operands indented on the line below their operator.

```
.         (handle contained in left operand)
  m2      (right operand – no handles)
  []      (left operand – handle is in its pointer operand)
   i      (non-pointer operand – no handles)
   .      (pointer operand – handle is in its left operand)
     m1   (right operand – no handles)
     z    (left operand – this is the handle)
```

New Syntax

The last detail that needs to be resolved before pointers can be constrained is to define new syntax for pointer declarations. New syntax is often an emotional issue because it involves aesthetics. One solution is to create a new keyword called `constrain`. A declaration such as:

```
double * constrain p;
```

would constrain expressions that use pointer `p` as a handle in an lvalue expression in the same way that arrays are constrained. That is, expressions with a constrained pointer and a handle `p`, such as `p[i]` and `*p`, cannot reference objects that are referenced by lvalue expressions with a different constrained handle. Certainly, other syntax exists which would permit this concept to exist, but at this point the concept is more important than the syntactic details.

Conclusions

This paper is not intended to address every design detail that must be specified before an actual implementation is attempted. It does point out that simple solutions to the aliasing problem do not solve the general problem. A general solution must address several issues: constrained pointers, handles, and new syntax. If a general solution is found, then compilers will be free to perform automatic parallelization. Until the solution is found, directives and compiler options will have to be used to improve performance.

Tom MacDonald is the Numerical Editor of The Journal of C Language Translation. He is Cray Research Inc's representative to X3J11 and a major contributor to the floating-point enhancements made by the ANSI standard. He specializes in the areas of floating-point, vector, array, and parallel processing with C language and can be reached at (612) 681-5818, tam@cray.com, or uunet!cray!tam.

∞

14. Electronic Survey Number 1

Compiled by **Rex Jaeschke**

Introduction

Occasionally, I'll be conducting polls via electronic mail and publishing the results. (Those polled will also receive an e-mail report on the results.)

The following six questions were posed to 53 different people with 20 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.

Definition of `__STDC__`

How do you define `__STDC__` in non-conforming or extended mode? Do you ever define it to be greater than 1? Equal to 2?

- 9 – Undefined in K&R mode, and 1 in ANSI-conformant (or superset) mode.
- 4 – 1 always.
- 1 – 0 in extended mode.
- 3 – 1 in extended mode with a separate preprocessor macro being defined to indicate extended mode.
- 1 – 2 in extended mode.
- 2 – 1 even though the mode is not quite strictly conforming.
- Comments:
    1. I don't believe `__STDC__` should be defined at all in non-conforming mode. If one decides anyway to define it, 0 is the best value. The positive values are reserved for present (1) and future (>1) versions of the C standard. Since one cannot at this time predict the content of future standards, values >1 are inappropriate and will eventually cause application porting difficulties.

2. Our system provides three modes: compatible transition behavior (-Xt), essentially ANSI behavior (-Xa), and conforming behavior (-Xc). Only in -Xc will `__STDC__` be defined to be 1. In -Xt and -Xa, `__STDC__` is defined to be 0. We have found that the features provided by -Xt and -Xa are the ones people wish to distinguish between: prototypes, new keywords, etc. The only real difference between -Xc and the others is the namespace guarantee, and most of our customers really don't care about this particular issue.

3. GCC defines `__STDC__` as 1 unless running under `-traditional`. GCC has several standard-wise modes. The default is what you call 'extended mode'; it has several nonstandard extensions. With the `-ansi` flag, GCC disables nonstandard things which would make an ANSI conformant program fail, and also predefines the symbol `__STRICT_ANSI__`, which makes GNU C Library header files not use any non-ANSI symbols. With `-pedantic`, uses of non-ANSI things produce warnings. With `-traditional`, some ANSI things are not accepted and some things are done differently.

4. Currently, it's not defined at all. I support some dpANS extensions which can be requested by a compiler switch, but they cause no change to any predefinitions. At first I thought of having this cause `__STDC__` to be defined as 0 but then realized that would not be portable anyway. I think it would be a bad move to define `__STDC__` as anything but 1 until such time as a revised standard is adopted by ANSI.

5. We will define `__STDC__` to be 1 for a slightly non-conforming extension. The extension will be to allow `asm`s and `long long` integer types, and probably to omit some required messages.

   If there were a consensus on how to indicate an "ANSI C like" compiler we would not abuse `__STDC__`. Our research leads us to believe that people will use `__STDC__` not to ask "Is this a conforming implementation?" but "Does this compiler support prototypes?", "Is this a UNIX pcc compiler?" and such questions.

6. Only conforming implementations should define `__STDC__`. Other implementations should come with some kind of list describing those ANSI items they do support.

   A conforming implementation with extensions could define `__STDC__` to be 1 if the implementation can assure that strictly conforming code is not affected by the extensions.

Alignment of Members in Structures

What is your default alignment of structure members? Can a programmer change it and if so, how? (pragma, compiler option, etc.)

- 15 – Fixed (possibly can be changed when compiler rebuilt or for different data types using a built-in algorithm).
- 3 – User-definable via compiler option.
- 3 – User-definable via pragma.
- Comments:

  1. Are there good reasons for wanting to change alignments?
  2. Our alignment cannot be changed although we see the need for this functionality in the future for systems with heterogeneous processors that share data with each other.
  3. It is simplest to align all structures on the same boundaries as the worst basic data type, typically `double`. I think allowing this to vary between separately-compiled modules is a horrible mistake, so there should be *no* way to override the built-in alignment.
  4. 2 vendors had: 1 byte objects (`char`) on 1 byte bounds. 2 byte objects (`short`) on 2 byte bounds. 4 byte objects (`int`, `long`, `float`) on 4 byte bounds. 8 byte objects (`double`) on 8 byte bounds.
  5. The default is word aligned (`sizeof(int)`) unless the member is a structure or union containing only bit-fields and having `total_size <= (sizeof(int)/2)`, in which case an attempt is made to align that member on an appropriate boundary. We do a similar thing with arrays of these structures or unions.
  6. Structure members are aligned on their natural boundaries. That is, the structure is aligned on the strictest member boundary.
  7. On PDP-10 machines, which have 36-bit words, using 9-bit `char`s, we have the following alignment requirements: `struct/union`, `float`, `double`, `long double`, `int`, `long` – word boundary; `short` – halfword boundary; `char` – byte boundary. Arrays start on a word boundary.
  8. Bit-fields do not cross `int` boundaries (except when, by extension we allow `long long` bit-fields).
  9. For a structure or union, alignment is the most restrictive of any of its members.
  10. We do not support changing the alignment but we do have an undocumented pragma for doing so.

Pragma Detection Messages

Do you produce an informational/warning message for each pragma accepted and rejected?

- 1 – Diagnoses all pragmas.
- 10 – No warnings for any pragma.
- 1 – Warnings are optional (using lint-like option).
- 8 – Warn about rejected ones only.
- Comments:

  1. The general UNIX philosophy is not to issue diagnostics when things are going well, so accepted pragmas should be silently accepted. "Ignored" pragmas are more problematic, although according to the standard I would say they're perfectly valid and should therefore also produce no diagnostics.
  2. If pragmas are used by the compiler itself in the future, unrecognized ones may generate warnings.
  3. I have no plans to recognize any pragmas at the moment. I think they are a terrible idea from a portability viewpoint, because there is no way to ensure that your pragmas are disjoint from those of others. It is safest to simply use none at all. System- or compiler-specific information belongs in the makefile or command line or whatever you use outside of the source. `#pragma` would be okay for "compiler hints" but *only* when enclosed within `#if` directives that have some firm assurance of doing the right things.

String Literal Attributes

Do you pool string literals and are they writable?

- 3 – writable and shared.
- 8 – writable and distinct.
- 6 – read-only and shared.
- 1 – read-only and distinct.
- 4 – User-specified (either or both attributes).

- Comments:

    1. I believe that writable pooled literals are exceedingly dangerous and must be avoided. Existing UNIX-based applications unfortunately require writable string literals, therefore no pooling. I would recommend pooling, therefore read-only, for implementations that do not need to worry about such backward compatibility.
    2. String literals are not pooled but I generally write my programs to pool them myself. In general they are writable—for compatibility—but if used in a context that includes an appropriate `const`, they are placed in a read-only portion of the program. For example:

        ```
            const char *msg = "read-only string";
        vs.
            char *msg = "writable string";
        ```

    3. String literals are generated at the end of whatever top-level declaration defined them—i.e., on the spot if file-scope `static`, at the end of a function otherwise. They go into the code space, which the user is at liberty to write-protect. It is definitely the intent that they not be writable. There is no attempt to share the use of literals. Three different instances of `"foo"` will each generate their own strings. Judging from our code mix, the cost of sharing such literals isn't going to be worth the minuscule few words gained, and the programmer can always do this sharing by defining a `static char` array.
    4. In non-ANSI mode string literals are writable; in ANSI mode, they are read-only.
    5. Identical string literals up to a length of 11 characters are pooled. String literals are writable.
    6. String literals share memory with executable code and, therefore, are not writable. We are still undecided whether to pool them and at what level.
    7. String literals are pooled in writable data segments. However, the linker has the ability to load these data segments in read-only segments.

Monitoring/Tracing Memory Allocation

Do you provide functions to monitor/debug the heap?

- 6 – Yes, but may be optional. (This might be achieved by using different versions of a library routine, or by using a library built differently.)

- 10 – No.
- Comments:

1. The `malloc` implementations I maintain all can be built specially for heap integrity checking when necessary, although that is not enabled by default (for efficiency reasons). I believe the form of such monitoring is best left up to applications, since it is not obvious what action should be taken when a problem is discovered.
2. More than one version of `malloc` (and family) is provided. Some provide information about the allocation arena. With some you can have extra checks performed at each call to the allocation functions.
3. This is not within the scope of a hosted C implementation. It is an issue for operating systems and debuggers.
4. `malloc` is compiled with `#if DEBUG` turned on.
5. For heap monitoring and debugging we make use of the "plumbers' `malloc`" package, which I believe is generally available.
6. We don't in the core release of GCC. On our contributed software tape, we will be providing `malloctrace` from comp.sources.unix, which helps track down memory leaks, and `malloccheck`, which checks for modifying memory after it's been deallocated, and also checking to see if a block is ever freed more than once.
7. There is a way to ask how much storage is allocated in the heap. More importantly, we do full runtime checking of pointers so that interpreted C code cannot corrupt the heap—the user gets a runtime error message instead.
8. `_heapchk`, `_nheapchk`, `_fheapchk` perform consistency checks on heap. `_heapset`, `_nheapset`, `_fheapset` perform consistency checks and set all free entries to a specified fill pattern. The functions `_heapwalk`, `_nheapwalk`, `_fheapwalk` walk through all entries in the heap one at a time
9. None planned so far. It sounds like a good idea though.

Extended Keywords

What extended (conforming or non-conforming) keywords do you support?

This question was posed simply to identify what identifiers were currently in use. I intend to document some of these (and others) in future issues. For the time being I will simply list (in alphabetical order) those identified by the respondents.

- Extended keywords.

  `_Abs`
  `__alignof`
  `_Alloca`
  `__asm`
  `asm`
  `atomic_add` *(builtin)*
  `atomic_bit_clear` *(builtin)*
  `atomic_bit_invert` *(builtin)*
  `atomic_bit_set` *(builtin)*
  `atomic_bit_test` *(builtin)*
  `atomic_cas` *(builtin)*
  `_cdecl`
  `cdecl`
  `_Complex`
  `_Far`
  `far`
  `fortran`
  `huge`
  `inline`
  `interrupt`
  `_KCCtype_charN`
  `long long`
  `_Max`
  `_Min`
  `_Near`
  `near`
  `no_code_motion` *(builtin)*
  `_Offsetof`
  `pascal`
  `pragma`
  `_ss`
  `static_register`
  `typeof`
  `unshared`
  `wait_until_bit_one` *(builtin)*
  `wait_until_bit_zero` *(builtin)*
  `wait_until_eq` *(builtin)*
  `wait_until_eq_and_store` *(builtin)*
  `wait_until_locked` *(builtin)*

- Comments:

  1. `asm` provides the behavior in the common extensions appendix, but also acts like a function storage class. If a function is defined as

having `asm` storage class, the tokens between the {}s are taken to be inline substitutable assembly language code. One can specify different possible sequences by matching the "shapes" of the function parameters, and the parameters are recognized and replaced as appropriate within the assembly language.

2. Currently all non-conforming features come through added semantics of the existing syntax or pragmas.
3. `pragma` is like `#pragma`, with function syntax, and can be used within macros.
4. `asm` lays down assembly code. You can specify input and outputs in terms of variables, and the compiler will substitute the appropriate register/offsets. `asm` is also used to change the external name of global items, and to hard-wire a variable to a global register.

Future Polls

Some of the topics planned for future polls are:

- Do you expand macros in pragmas?
- What predefined macros do you have? What is their purpose?
- If you support case ranges, what syntax do you use?
- Do you take notice of `register`? If so, how many registers do you make available and what is their size? Can an object use multiple registers? What types are accepted? If there are more requests than you have registers, what is your allocation strategy? Is yours a hosted or free-standing implementation?
- Do you have some sort of a 'stack probe' to check that space is actually available for automatic objects before allocating them?
- Do you allocate automatic storage on entry to each block or do it all on entry to the function?

If you have any topics to add to a poll please send them to me. I will provide the responses to you as soon as they are collated, as well as publishing them in a future issue. You don't need to have an e-mail address to propose topics, only to be polled.

∞

15. A Parallel Processing Implementation

Mike Holly
Cray Research Inc.
1345 Northland Drive
Mendota Heights, MN 55120

Abstract

One way to support the creation of parallel programs is via directives added to a standard programming language such as C. In this paper I will describe one such approach in a C compiler that runs on Cray supercomputers.

Introduction

In recent years there has been a strong trend for computers to be designed with increasing numbers of CPUs, in an attempt to pack more and more computing power into a single machine. While such machines typically have operating systems that support running multiple jobs simultaneously, systems that apply multiple CPUs to a single job are less common.

There are at least two reasons to want to apply more than one CPU to a single job. One is to attain the absolute maximum speed in a compute-intensive application such as weather forecasting. The other reason is to make better use of the multiple-CPU machine even in a multiple job stream environment by using up the machine cycles which would otherwise be wasted, thereby decreasing the elapsed time of execution for the parallel application.

The ultimate goal for a compiler in a multiple-CPU environment is to automatically use all the available resources, and in particular to cause as much of the application program as possible to run in parallel with no user intervention.

One way to achieve maximum parallel operation is for the application to be written in a language specially designed for parallel operation. Usually this approach requires the programmer to learn an unfamiliar language, and the resulting code is not very portable.

An alternative is to provide extensions to a standard language such as C or FORTRAN. While using an extended standard language also requires some initial work by the application programmer, this approach can often result in programs which are both highly parallel and somewhat portable. Using an extended language also allows for the parallelization of existing code with relatively little rewriting.

The Primary Parallelization Directive

For the project at hand it was decided to provide directives to allow the C programmer to dictate the parallelism in the C code. The directives chosen were modeled after a subset of a similar group of directives already available in an existing FORTRAN compiler.

The primary possibility for finding parallelism in C code was judged to be in the `for` loop. Therefore the main directive was designed to allow the programmer to designate a `for` loop whose various iterations could be executed in parallel. The approach taken was to implement some new pragmas in C, and the name chosen for the primary directive was `#pragma taskloop`.

The syntax of the `#pragma taskloop` is:

```
#pragma taskloop private (var [,...])                \
        [shared (var [,...])]                        \
        [if (scalar expression)]                     \
        [savelast]                                   \
        [chunksize (integer expression) or vector]
```

The arguments of the pragma, which are all optional except for the list of private variables, are:

`private`: Allows the programmer to specify the names of variables that will be treated as private data (further described in *Data Considerations*).

`shared`: Allows the programmer to specify the names of variables that will be treated as shared data (further described in *Data Considerations*).

`if`: Allows the specification of an expression which is evaluated at run-time, with parallel execution of the `for` loop occurring only if the expression is true.

`savelast`: If specified, variables whose values are set inside the loop will be guaranteed to have the values produced by the final loop iteration when the loop is exited. This is not guaranteed by default.

`chunksize`, `vector`: These are scheduling options. By default, loop iterations are handed out one at a time to the several processors. If `chunksize` is specified, then the iteration space is broken into chunks of size n, where n is a loop-invariant integer expression. If `vector` is specified, an optimization algorithm is employed to determine how many iterations will be assigned to the next available processor, with a minimum size of 64 iterations per processor. (Of course, in either case there may be a final chunk with fewer iterations.) The `chunksize` and `vector` arguments are mutually exclusive.

The `taskloop` pragma as currently implemented does not allow macros anywhere inside the pragma; this facility would be a useful future extension. The pragma may be continued across multiple source lines using the backslash/newline continuation sequence. Note also that the pragma in essence represents executable code, since it will be expanded into code by the compiler. This is a slightly unconventional role for a C pragma.

Constraints on Using the `taskloop` Pragma

In order for the `taskloop` pragma to function properly, the `for` loop to which it is applied must satisfy certain restrictions. Foremost among these is the requirement that the loop be parallelizable. This determination is left up to the programmer. There are also requirements on the syntax of the `for` loop. A significant portion of the job of adding this pragma to the compiler involved checking the `for` loop to ensure that these restrictions are met.

Given that the general form of a `for` loop is:

```
for ( exp1; exp2; exp3 )   { /*** loop body ***/ }
```

then the restrictions are:

- Expression `exp2` must have the form *LCV op expression* where the loop control variable *LCV* is an integer variable, *op* is from the set { `<` `<=` `>` `>=` }, and *expression* is any valid C expression whose operands are all loop-invariant.

- Expression `exp3` must be a single[1] assignment, `++`, or `--` expression, whose lvalue is the *LCV*. If `exp3` is an assignment, then the *LCV* must be incremented or decremented by a constant.

- The *LCV* must not be redefined inside the loop (or be `volatile`).

- The loop may not contain constructs such as `p++;` or `p = p + 10;` which would not increment properly in a parallelized loop. (The user can convert these to a form such as `p = pinit + k *` *LCV*, which can be parallelized.)

There is no restriction on `exp1`, and in fact it may be omitted. The restrictions on `exp2` and `exp3` allow the compiler to know that the loop will be executed in an orderly fashion, with the *LCV* changing in an arithmetic progression, so that the assignment of loop iterations to the several processors may be done correctly.

The loop body restrictions are much less severe for parallelizing than they are for vectorizing. For example, in parallelizing the loop body may contain `if` statements, function calls, and `goto`s which branch to other portions of the loop.

[1]Expressions of the form `a = b = c` and `a = b, c = d` are not permitted.

Data Considerations: Shared vs. Private

Each of the variables referenced inside the parallelized `for` loop must be either `shared` or `private`. `shared` variables have a single instance which is accessible from all of the processors working on the loop in parallel. `private` variables have separate independent instances for each of the processors.

The data-scoping rules of C are helpful in allocating variables to the proper category. In this design, all variables with external or internal linkage are automatically `shared`. All variables declared in inner scopes of the parallel `for` loop are automatically `private`. Variables not in either of these automatically-determined categories must be explicitly made `shared` or `private` by including their name in the appropriate list in the pragma. Some helpful guidelines are:

1. Variables whose values are passed in from outside the loop are `shared`. These will usually include the bounds of the *LCV*s.
2. Variables that receive new values inside the loop are usually `private`. This must include the *LCV* of the parallelized loop, and will usually include the *LCV*s of nested inner loops. The exception to guideline 2 is guideline 3:
3. Generally the loop will be calculating the elements of a result array which is indexed by the *LCV*. This array must be `shared`.

Note that the *LCV* of the parallelized loop is in fact a special case of a `private` variable. Its initial value is passed in from outside the loop if it is not initialized in the `for` statement itself.

As the design proceeded, it seemed unnecessarily complicated to allow structures and unions to be declared in the `shared` or `private` lists. Thus the only aggregates allowed in these lists are arrays (which must not contain structures or unions). Structures and unions may be made `shared` or `private` by using the default rules mentioned previously.

Variables which are not shared or private (either by the default rules or by inclusion in the shared/private lists) are out of scope for the duration of the parallelized `for` loop. Any attempt to reference such variables inside the loop results in an error message.

Other Directives

Sometimes it is necessary to allow a shared variable to be written while inside the parallelized loop, although in such cases it is almost always desirable to have only one of the processors writing at a time. An example of this is the updating of a pointer which is used to access some allocated memory. Another example is any call to `printf` inside the parallel loop, where we want to insure that the processors do not try to write to the `printf` buffer simultaneously. The `guard` and `endguard` pragmas were designed to handle this.

The programmer inserts `#pragma guard` to designate the beginning of a guarded region, and `#pragma endguard` to end that region. It is guaranteed that only one of the parallel processors will enter the guarded region at a time. Since a guard forces non-parallel behavior, its use should be minimized for best efficiency.

The `guard` and `endguard` pragmas must always be used in pairs, and may not be nested.

Example

Here is a simple example of a matrix initialization loop and a matrix multiply loop, which have been parallelized by using the `taskloop` pragma.

```
/* Matrix Multiply Example */

#define SIZE 50

float a[SIZE][SIZE], b[SIZE][SIZE];

main()
{
int i, j, k;
float c[SIZE][SIZE];

#pragma taskloop private( i, j )

/* Initialize arrays a and b.  Because of the
preceding pragma, the i iterations of the outer loop
are done in parallel. */

for ( i = 0; i < SIZE; i++)
        for ( j = 0; j < SIZE; j++) {
                a[i][j] = 1.0;
                b[i][j] = 2.0;
        }
#pragma taskloop private( i, j, k ) shared( c )

/* Calculate array c as the matrix product of a and b
Because of the preceding pragma, the i iterations of
the outer loop are done in parallel.  Each processor
in turn is assigned one iteration of the outer loop. */

    for ( i = 0; i < SIZE; i++) {
        for ( j = 0; j < SIZE; j++) {
            c[i][j] = 0.;
            for ( k = 0; k < SIZE; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }


/* print a few selected values */

    if (i % 20 == 0)
#pragma guard

        printf("c[%d][40] = %7.3f\n", i, c[i][40]);

#pragma endguard

    }
}
```

Note that arrays `a`, `b`, and `c` are shared, but `a` and `b` are not listed in either `taskloop` pragma since they are declared at global scope and therefore, are automatically shared.

Implementation

The pragmas described were implemented in the C compiler directly, rather than in the C preprocessor or some other preprocessor. By using this approach, we could take advantage of the existing lexical analysis and parsing code, as well as using the existing symbol table and a number of lower level utility routines. One drawback, as compared to a preprocessor approach, is that the result is relatively difficult to port to another C compiler.

It took about 1400 lines of (commented) C code in two new routines, plus about 1000 lines of modification/addition to existing routines, to implement the parallelizing pragmas. Fortunately, since this effort had been preceded by similar work in a FORTRAN compiler, the underlying library routines were already in place to do the work of interfacing with the operating system, allocating and de-allocating extra processors, etc.

The design requires that a separate subroutine be created for each parallelized loop in the C source program. This subroutine is then executed by each of the auxiliary parallel processors to accomplish one or more iterations of the loop. One of the more difficult tasks encountered was to get this subroutine properly generated in the midst of an ongoing compilation. It seemed that the choices were either to save all of the necessary symbol table and dimension information for each parallelized loop until after the original program had been compiled, or else to save the complete state of the compiler whenever the subroutine was to be created, and then restore the compiler and resume compilation of the original program. The latter approach was chosen. It required saving and restoring about 150 individual variables and 17 arrays or tables.

Future Directions

The work described has been implemented in Cray's version of the Portable C Compiler (PCC) as a test bed. The next major goal for this project is to implement the parallelizing pragmas in an ANSI Standard C compiler. At the same time, there are a number of enhancements to consider:

- Have the compiler automatically find and rewrite constructs such as `p++` into parallelizable forms.
- Allow unions and structures declared at intermediate scopes to be listed in the shared or private lists.
- Allow macros in the `taskloop` pragma.
- Extend the `taskloop` pragma to `while` and `do while` loops.
- Make the *LCV* of the parallelized loop automatically private, so that it need not be included explicitly in the private list.

Finally, a worthwhile goal, but beyond the scope of work currently planned, would be to automatically recognize parallelizable loops, and cause them to execute in parallel, without any need for user-generated pragmas.

Conclusions

The C compiler described is due for release in Q3 '89. Meanwhile it has been applied to a number of application programs for testing purposes. Among the programs which have been successfully parallelized with the C compiler are a ray-tracer, a Mandelbrot set generator, a matrix-multiply routine, and any number of smaller test programs.

As expected, the total CPU time of the parallelized applications goes up (perhaps 5 or 10%) while the elapsed (wall-clock) time usually decreases, sometimes substantially. Improved wall-clock times of almost 8× have been observed for the most straightforward programs running on an 8-processor Cray Y-MP. More typically, improvements of 1.5× to 2.5× on a 4-processor X-MP have been observed, even when the machine was shared with other users.

The performance of the parallelized code varies widely, depending on how much of the total program can be parallelized, and on how much parallel machine resource is available at the time. It is almost always true that execution times are improved, however.

Our experience has been that the most important factor in successfully parallelizing a C program is a thorough understanding of the algorithm. Once the opportunities for parallelism are understood, the mechanics of inserting the proper directive, and of sorting the variables into the shared and private categories, are relatively straightforward and painless.

Mike Holly is a member of the C compiler development team at Cray Research Inc., in Mendota Heights, MN. He is currently adding support for parallel computing to Cray's Standard C Compiler. Dr. Holly also has a long-standing interest in the field of computer graphics, which he teaches at the College of St. Thomas in St. Paul, MN. He may be reached at uunet!cray!hollyma or hollma@cray.com.

∞

16. C++ Standardization Project Report

Dmitry Lenkov
Hewlett-Packard Company
California Language Lab
Cupertino, CA 95014

Abstract

At their meeting in July, the ANSI X3 Standards Planning and Requirements Committee (SPARC) unanimously recommended that the X3 Secretariat create a new standards committee for the C++ programming language. This paper covers background information and important issues raised in the proposal submitted to SPARC, which served as the basis for that recommendation.

Need for Standardization

C++ is a C-based object-oriented language invented by Bjarne Stroustrup at AT&T Bell Laboratories, which is now widely perceived as one of the languages best suited for many types of software development. C++ is going through a much faster public acceptance path than most other languages. C++ is already available on almost all machines where C is used. Projects choosing C++ for development vary from small to very large and ambitious.

While the growth of C++ popularity is enthusiastically supported by the C and UNIX communities, the main reason for such growth is that many companies are finding C++ of strategic benefit for their product development. The requirements for the portability of C++ programs, for a full specification of C++ as a language, and for a specification of C++ libraries and supporting environment have reached a level of importance similar to that of other widely used and mature languages such as C and Pascal.

The question of C++ standardization has been debated in the industry and academic world for at least a year. Despite the growing support by major language vendors and C++ users, until very recently many C++ users and several large companies (including AT&T) resisted initiation of the C++ standardization process because of language immaturity and lack of some features important for modern software development. Bjarne Stroustrup, the language creator, had some concerns as well. However recent events led him and AT&T to change their opinion.

The announcement of AT&T C++ Language System Release 2.0, in June 1989, has definitely played an important role. This release represents the fullest language definition currently available, and will provide the base for new C++ products as well as for the development of a C++ standard.

Another important factor is that several major companies (including Digital Equipment Corporation, Hewlett-Packard Company, and Microsoft Corporation) have committed to C++ as vendors and/or users. This has made C++ stability a critical issue.

Hewlett-Packard proposed a C++ standardization project to SPARC in April, 1989. In June, AT&T decided to support this proposal, and Stroustrup agreed to participate in the work of the C++ standard committee when it is approved by the X3 Secretariat.

Scope of C++ Standard

The recommended scope of the proposed standard includes three layers of the language and associated libraries and tools.

Most of the C++ implementations are based on the AT&T translator implementation of C++ and Stroustrup's book, papers, and other documents. AT&T C++ Language System Release 2.0 contains the C++ reference manual by Stroustrup with the latest and most complete definition of the language, the translator version implementing C++ according to this definition, and three libraries. The first layer of the standard should encompass the features and libraries available in this version of the AT&T C++ Language System. These features and libraries have been heavily used by various development projects.

The second layer includes primarily libraries, header files, support environment features, and certain run-time mechanisms that have been implemented and sometimes used extensively, but that still involve experimentation and have problems or are being implemented and need more experience. By the time the C++ committee prepares a draft of the C++ standard, this layer of the language will be significantly more stable.

The current implementation of C++ by AT&T does not provide certain important features that are needed for large development projects. Among them are exception handling and support of generic functions and types. Some internal implementations already support these features in various ways. Implementations that are commercially available can be expected in the near future. In addition, almost every development project using C++ develops its own libraries. Many of these additional features and libraries are wanted by a large group of C++ users. These additions constitute the third layer. It will take some work to decide which of them should be included into the standard.

C++ and C

An important factor in wide acceptance of a C++ standard is its compatibility with ANSI C. The community of C users and product suppliers has spent a large amount of effort to arrive at the ANSI C definition. It will take a continuing effort and investment from C product suppliers to conform to ANSI C and from the developers to port existing C code to ANSI C. To minimize future investment in similar efforts, it is very important to provide compatibility of the future standard C++ with ANSI C .

C++ is not strictly compatible with either C, as defined by Kernighan and Ritchie, or ANSI C. The differences between ANSI C and C++, as defined by the AT&T 2.0 reference manual, are minimal and do not present a practical barrier for converting ANSI C programs into C++ programs. However it is important:

1. to develop a standard conversion procedure that may include conversion routines and a set of documented steps to be taken,
2. to establish a coordinating procedure between X3J11 and the C++ committee that will assure the same level of compatibility between ANSI C and C++ when any new extensions to ANSI C are considered.

The Next Step

Now we must get approval of the C++ committee by the X3 Secretariat and to get C++ experts together to organize this very important effort. For a successful completion of the C++ standard, appropriate representation by implementers and users in the process is very important. Please get in touch with me if you want to attend the first meeting (probably two days long).

Dmitry Lenkov is a software design engineer at Hewlett-Packard's California Language Laboratory. He is a principal instigator in the effort to establish a C++ Standard. He may be reached at H-P, 19447 Pruneridge Avenue, MS 47LE, Cupertino, CA 95014, (408) 447-5279, or electronically at dmitry%hpda@hplabs.hp.com

∞

17. Parallel Programming: Linda Meets C, Part II

Jerrold Leichter
Yale University
and
Digital Equipment Corporation

Abstract

Linda is a programming model for developing explicitly parallel programs. In Part I of this series, we introduced the model. In this part, we discuss the injection of Linda's primitives into C.

Introduction

In Part I of this series, we introduced the Linda programming model, with its single data structure, tuple space, and its four operators, `in`, `rd`, `out` and `eval`. While there is more to be said about the abstract model,[2] we will not examine it further here. Rather, in this article we will discuss several embeddings of Linda in C that have been defined as part of Linda implementations at Yale University.

C-Linda

C-Linda is the name given to a series of Linda systems developed by Nicholas Carriero. Early versions of C-Linda constituted the first running implementations of Linda. C-Linda continues to evolve. We will describe in detail only the two earliest versions of the language, the only ones for which documentation is readily available.[3]

C-Linda$_0$: Linda as Support Library

The easiest and most obvious way to support Linda is to leave the underlying language unchanged and provide function calls to implement the tuple space

[2]See our recent dissertation, "Shared Tuple Memories, Shared Memories, Buses and LAN's — Linda Implementations Across the Spectrum of Connectivity," which is available as Yale Department of Computer Science Technical Report TR-714, July 1989.

[3]See Carriero's dissertation, "Implementation of Tuple Space Machines," available as Yale Department of Computer Science Technical Report TR-567, December 1987.

operations. Several implementations of this general sort have been reported. Charles Fleckenstein's Master's Thesis[4] is one example. Brenda, a Linda-like library developed by Moshe Braner as part of the Trollius project at the Cornell Theory Center, is another.[5]

In developing this kind of implementation, a fundamental choice has to be made as to how to specify tuples, which vary in the number and types of their elements. Brenda avoids this entirely by essentially requiring all tuples to have exactly two fields: An integer index used in matching, and an arbitrary array of bytes that are simply copied in or out of the "tuple." It's obviously possible to make such an approach a bit more sophisticated by having, say, a fixed set of fields—one integer, one floating point, one string—and placing dummy values in any fields that aren't needed. Such an approach is very simple and can be very fast, but at a large cost in flexibility.

Carriero's implementation[6] chooses flexibility over performance. The first argument to any type operation is a `scanf`-like format string. Only three types are supported: `d` specifies a `long`, `s` specifies a zero-terminated string, and `b` specifies a block, a contiguous array of `long`'s the first of which is the number of additional `long`'s that follow. A formal is specified in the format string by using `f` as a modifier. For example, to create the 2-tuple

$$\left\langle \texttt{123}_{\texttt{long}}, \texttt{"dog"}_{\texttt{string}} \right\rangle$$

we might execute:

```
        out("d s",123,"dog");
```

Later, we might match it using

```
        in("d fs",123,&animal);
```

C-Linda

While viewing Linda as support library is certainly possible, it is not ultimately a good approach for a language such as C, which does not have a run-time type system. It's too error-prone (the compiler can provide no error checking), too inefficient (the control string must be parsed repeatedly, and it's impossible to make any compile-time optimizations), and ultimately too inflexible (adding user-defined types is impractical). In practice, it's also annoying to use. After all, the compiler already knows the types of all the expressions used — why must we repeat ourselves and write a control string?

Instead, we would like the compiler to make use of its knowledge of types to generate the appropriate data structures. Along the way, we can let it make optimizations as well.

[4] "A Distributed System Using Logically Shared Memory," Write State University Department of Computer Science and Engineering, 1988.

[5] See Braner's "Brenda — A Tool For Parallel Programming," available from the Center.

[6] Fleckenstein's approach is roughly similar.

Carriero's second implementation accomplishes this using a pre-compiler. The pre-compiler, originally based on `pcc`'s front end and later re-implemented using `gcc`, re-writes a C-Linda program into a C program. The Linda operations are seen by the compiler as expressions.[7] The pre-compiler determines the types of all the fields and whether they are formals or actuals. It builds a structure analogous to the control string of the previous section, but organized for better run-time access. Note that to determine the types of expressions, the pre-compiler must understand just about all of C. That's why we call it a pre-compiler rather than a pre-processor.

The examples of the previous section would appear in C-Linda as:

```
out(123,"dog");
⋮
in(123,? &animal);
```

The ugly `&` was removed from the syntax for formals in later versions of the language.

The C-Linda system supports a few more types than C-Linda$_0$ did: `int` and `long` are both available as distinct types and `double` was added. The "string" type remains unchanged, while the "block" type is now based on a standard `struct` consisting of a length and a pointer to arbitrary data. Since C doesn't really have either a string or a block type, this pre-compiler actually treated any `char *` field as a string, and any `struct` as a block.

As originally implemented, the C-Linda pre-compiler did not support separate compilation: While a C-Linda program could consist of more than one module, only one module could perform any Linda operations. However, the compiler does perform a number of important optimizations. As a very simple example, if it can determine that a given field always appears as a constant in all possibly-matching tuples, it can use that field to subdivide tuple space *at compile time.* The field can then be discarded.

The most recent versions of Linda-C support separate compilation and allow fields of tuples to have any legal C type. The details are not stable at this writing, so we will not explore them further.

Linda-C

Linda-C represents a closer coupling between C and Linda than previous implementations. The goals we had in mind in developing the language included:

- Better support for types. Not only should all the native C types be supported fully, but programmers should be able to define their own types

[7]The reason they are expressions, which happen to return no useful value, rather than statements, is that this implementation supports non-blocking variations of `in` and `rd`, known as `inp` and `rdp`. The non-blocking operations return `0` if they fail to locate a match, `1` if they succeed.

and have them used in matching. The reasoning here is simple: C is distinguished from BCPL by its use of typed objects, rather than simple untyped memory words. Where a BCPL programmer must specify "integer add" or "floating add" as part of an expression, the C programmer specifies the types of the variables and lets the compiler determine which underlying machine operations are required. The native types in C are those naturally tied to machine operations. In Linda, we think of tuple space operations as primitive, and so wish the richer set of types natural to *them* to be natural to LINDA-C.

- Automatic support for aggregates. In many cases, the compiler knows the size of aggregates — `struct`'s or arrays with fixed dimensions are examples. In these cases, it should be able to determine the data that should be moved to or from tuple space. In other cases — zero-terminated strings, for example — the programmer should be able to provide this information in a natural way.

- Support of separate compilation consistent with the previous two goals.

To achieve these goals, LINDA-C adds two new constructs to C, the `newtype` specifier and the `varying` modifier.

The `newtype` Specifier

The `newtype` specifier is a variation on the `typedef` specifier. Where `typedef` introduces synonyms for existing types, each instance of `newtype` creates a new, unique type. For example,

```
newtype int    INT, *PINT;
newtype int    OTHERINT;
```

creates two new types, one named by `INT`, the other named by `OTHERINT`. These types are distinct: A field of type `INT` does not match one of type `OTHERINT`, and neither matches a field of type `int`. `PINT` names a type for a pointer to an `INT`; that is, if `pi` has type `PINT`, then `*pi` has type `INT`.

In LINDA-C, every type name and expression has associated with it two distinct types: Its C type and its Linda type. Unless a type created with `newtype` occurs somewhere in an objects definition, the two types are always identical. However, as soon as `newtype` is involved, they differ. For example, after

```
INT    I;
```

`I` has C type `int` and Linda type `INT`.

C types are used in all LINDA-C contexts except tuple matching. A C expression has exactly the same semantics whether viewed as C or as LINDA-C. However, in determining whether fields match, only the Linda type is significant. The C type is ignored.

The previous paragraph outlines the most important points. There are many details. For example, what should the type of

```
I + 1;
```

be? Its C type is clearly `int`, but its Linda type is problematical. Even if we decide to make it `INT`, we would be left in a quandary by an attempt to add to objects with *different* Linda types. As a result, LINDA-C assigns the special type `!notype!` to any such combination. The compiler reports an error if an attempt is made to use a field with Linda type `!notype!`. The programmer can always correct the problem by using a cast, which modifies both C and Linda types.

An interesting problem is illustrated by the following example:

```
INT    A[10];

A[2];
```

What is the type of the array reference? Under the usual C rules, an array reference is simply a shorthand for an addition followed by a de-reference, so this array reference should be the same as:

```
*(A + 2);
```

However, the Linda type of

```
(A + 2)
```

is `!notype!`, which is also then the Linda type of the whole expression. Trying to preserve the Linda type in the inner addition won't work, since addition is commutative. But clearly we must insist that the type of

```
A[2];
```

be `INT`!

LINDA-C gets around this problem by discarding C's notion that an array reference is just a shorthand for explicit address arithmetic. The Linda type of an array reference in LINDA-C is determined entirely by the type of the array. The type of the subscript is ignored. We consider this incompatibility with C minor, and in any case we would argue that insisting on this equivalence does more harm than good. For example, it is easy to define a notion of "an array of bitfields," provided that you don't insist that a reference to an element of such an array go through the usual shorthand mechanism.

Global Types

If separate compilation is to be supported for LINDA-C, it must be possible for separately-compiled modules to share types. LINDA-C allows this as follows: All native C types — `int`, `double`, and so on — are implicitly global, and have the same meaning everywhere. New Linda types created with `newtype` are global unless they are declared within a block. Declarations within a block have a scope restricted to that block. Globally-known types are matched across separately-compiled units by name.

These rules are directly analogous to the scope rules for C objects. One thing that is missing is an analogue of file scope. The obvious

```
static newtype int INT;
```

runs into a syntactic problem, since both `static` and `newtype` are by definition storage classes. Once again, an elegant simplification present in C — that `typedef` can be seen as a storage class — turns out to cause problems when we attempt to extend the language.

Aggregates and `varying`

C is heavily biased toward addresses, rather than data values. However, it is generally meaningless to pass addresses in tuples, since the different participants in a parallel computation may not share address spaces. If `a` is an array of ten `int`'s, then interpreting

```
out("a",a);
```

as inserting `a`'s address into a tuple is not very useful. Rather, what we would like to see inserted into the tuple is `a`'s *value.*

In tuple fields, LINDA-C reverses C's usual address-biased conversions. It will in fact insert `a`'s value — ten `int`'s — into the tuple. Further, if a tuple field as given has type "pointer to ...", LINDA-C will re-write this into "array of ..." — the exact inverse of C's usual conversions.

There is, however, a problem with this conversion. While the compiler can determine the size of an array declared with explicit bounds, it has no way of determining the size of the array to which it must convert a pointer. Put another way; the compiler has, in general, no information concerning the size of the data object to which a pointer points.

LINDA-C allows the programmer to provide the compiler with the means of computing the length of an object. The mechanism is the type modifier `varying`. For example, we might declare the type `STRING` by

```
newtype varying(void) char *STRING;
```

A `STRING` is a pointer to `char`, but it carries along with it the additional information that the length of the object being pointed to can be determined by scanning to a zero byte. A LINDA-C compiler will produce code from

```
STRING s;

s = "Hello, world";
out("hi!",s);
```

that determines the length of `s` using `strlen`, and then inserts that number of bytes into the tuple.

There are several classes of `varying` modifiers besides `void`. For example, `int` indicates that the length of the data object is stored as an `int` at the location pointed to. The actual data follows.

`varying` modifiers do not affect whether types match, so in principle a field can be `out`'ed with one class and `in`'ed with another, effecting a change of representation. The current LINDA-C pre-compiler does not support such uses, insisting that all instances of a given type be declared with the same `varying` class. For pure C programs, this is a minor inconvenience. Data representation conversions will become important with tuple spaces accessed by programs written in different languages, however.

While we used `newtype` in our example, `varying` is independent of `newtype`, and can be used with built-in types. Since it is a first-class part of the LINDA-C type system, it can also be used in casts, allowing a programmer to select the appropriate representation in the actual tuple operation.

Conclusions

We have given only an overview of the C-Linda and LINDA-C languages, and only a bit of the flavor of the tradeoffs necessary in defining them. Much more detail can be found in the two dissertations we cited earlier. These dissertations also discuss matters we have not discussed: the semantics of `eval`, and the practical implementation of these ideas. We will return to these in Part III.

Availability

Newer versions of Carriero's implementations of C-Linda are available commercially from SCA in New Haven. Our implementation of LINDA-C for VAXes has been made available to several laboratories on an experimental basis. The form of commercial availability of LINDA-C is undetermined at the time of writing. Contact the author for further information.

Jerrold Leichter has recently completed his doctoral research, which includes an implementation of Linda for shared-memory and networked VAXes, at the Yale University Department of Computer Science. He is also a long-time employee of Digital Equipment Corporation, whose Graduate Engineering Education Program supported him during some of his work. He may be reached electronically as leichter-jerry@cs.yale.edu.

∞

18. ANSI/ISO Status Report

Jim Brodie

Abstract

This article discusses what is happening in the efforts to get final ANSI approval of the draft proposed C standard developed by X3J11. It explains the most recent delays and discusses the currently pending appeal against the procedures used by X3J11 in the development of the standard. It also gives the latest "best guesses" of when the process will produce an approved standard.

Introduction

Whenever I talk to people about C, the most common question I'm asked is "Is it a standard yet?" The members of X3J11 believed that the approval of the standard by the American National Standard's Institute (ANSI) would be finished by now. In the last issue I made the bold statement "We are hopeful that we will have an approved American National Standard by July of this year [1989]."

Unfortunately, the schedule has slipped. There is still no approved ANSI standard for the C language. In the remainder of this article I will try to outline what has happened over the past several months and give my latest thoughts on what is most likely to happen next.

One Step Forward, Two Steps Back

Following X3J11's April meeting in Seattle the committee's responses to a late public response letter were put into the form of a formal response document. These responses were sent to both the letter writer and to the X3 Secretariat.

The letter writer, however, did not accept the committee's responses to his issues. Not only did he reject the responses, he wrote a formal appeal to X3 challenging the work of X3J11. And until that appeal is processed, the draft standard will *not* be submitted to ANSI for final approval.

The rules for the standards formation process have been designed to ensure that the rights of a minority (in this case a minority of 1) are upheld and protected. To this end, there is a detailed set of procedures that must be followed to carefully consider, respond to, and hopefully resolve an appellant's complaints. The delay introduced by following these procedures is regrettable. However, like the American justice system, protecting the rights of the people involved is important if the results are to be considered fair and unbiased.

Progress towards an approved ANSI standard has not been stopped completely during this appeal processing period. An X3 reconsideration ballot was held. The appellant's public review letter and the X3J11 responses were distributed to the X3 members. They were given an opportunity to change their vote based on the issues raised in the letter. This reconsideration ballot closed without any vote changes (the original ballot closed with 34 in favor, 0 against, 2 abstaining, and 4 not voting.)

The Appeal

The appeal document is fairly lengthy, raising about 40 different technical and procedural issues. According to X3's ANSI-approved rules, only procedural issues can be addressed in an appeal. However, each procedural issue must be responded to, in writing, by X3.

A response to the appeal was prepared and sent out by X3 in late July.

Appeal Issues and X3's Responses

To give some feel for the appeal, I will discuss a couple of the major complaints which the appellant raises. Along with the complaints I will also indicate the X3 responses.

One complaint raised in the appeal is that several letters that the appellant wrote were lost and not responded to in a timely fashion.

In the response, the X3 Secretariat recognized that:

> "... through clerical, postal, and/or simple human error(s), materials may have been lost. Subsequent to receiving notice of your comments, actions were taken by us to notify X3J11 and by X3J11 in order to process your comments."

X3J11 believes that every reasonable effort was made to give a fair and thorough hearing to the issues raised in the appellant's public review letter (See *ANSI/ISO Meeting Report* in *Volume 1, Number 1, June 1989*).

Another major point raised in the appeal centered on the appellant's belief that the ANSI requirement that a standard's committee have a "balance of interest groups" had been violated in the formation of X3J11. In particular, he claimed there was insufficient freestanding "embedded systems" expertise on the committee.

The "balance of interest groups" requirement is intended to ensure that a standard adequately addresses the needs of the broader community, not just a portion of the community, such as producers of products (e.g., compiler developers).

The X3 response points out that both the understanding of the requirement for a balance of interest groups and the claim of a lack of expertise are mistaken. The X3 response says:

> 'The term "balance of interest groups" (as used in ANSI and X3 procedures) refers to the Producers, Consumers, and General Interest categories. The approved procedures apply in this categorization at the X3 level only. You have taken this term and attempted to apply it to specialized areas, such as "embedded systems," "numerical programming," and "graphics." Any general-purpose standard has dozens of identifiable specialized areas; it would be unworkable to require that all of them are "balanced" in any particular specialized area and our procedures do not require this. Instead, the Technical Committees are open for membership to any and all qualified persons; it is this very openness which supports the fairness of the process. The open Technical Committee's results are then examined by a balanced accredited committee, X3, which is itself open to all materially interested organizations. We believe this approach is both workable and fair.'

The X3 response further points out that many of the companies which deal in the freestanding embedded systems marketplace are represented on X3J11. The response notes:

> "Though the Secretariat is not expert in this area and without intending to be all-inclusive, we suggest that AT&T, Computer Innovations, Control Data, Data General, Digital Equipment Corporation, Intel, National Semiconductor, NCR, Texas Instruments, Unisys, etc., would qualify as having expertise in embedded and numerical applications."

Another basis for the appeal is that X3J11 is unwilling to seriously consider changes to the standard. The appellant writes:

> "Many of the shortcomings of the draft ANSI C had been pointed out in other public comments and the X3J11 Technical Committee was not willing to spend the time to develop solutions, choosing instead to quickly issue a deficient draft ANSI C."

The X3 response is:

> "X3J11 spend over six years developing the draft standard including making revisions based on two public reviews that produced over 1700 specific responses, plus a third public review that produced no changes. The Committee adopted procedures to propose, discuss and vote, on each proposal or comment. These procedures comply with the X3 SD-2 [the X3 document describing Policies and Procedures]."

The fundamental X3 position is that the appeal is unwarranted and that no additional remedial action is required by X3 or X3J11.

Addressing the Technical Issues

The technical issues raised in the appeal centered on the appellant's claim that the standard was not adequate to allow C programmers to do freestanding embedded systems work. Since it is not appropriate to deal with the technical issues as part of the appeal process, they are instead handled by X3J11 and the X3 membership review (in this case by the X3 reconsideration ballot, discussed earlier.) X3J11 addressed them during its April meeting. The technical issues raised by the appeal are now considered closed since there were no negative X3 votes during the reconsideration ballot.

What's Next

If the appellant accepts the X3 response (or does not respond within 15 days), the proposed draft C standard will be forwarded, during August, to ANSI for consideration.

If the appellant does not accept the X3 response to his appeal, then a formal appeal board must be formed. This board is made up of three members who are experienced in the rules and procedures of X3. One member is selected by the X3 Secretariat, one by the Appellant, and one is mutually agreed upon. If necessary, the process for forming and convening this board will probably take at least two months. The findings of this board will govern what further steps must be taken.

If the board finds in favor of X3J11, the standard will be forwarded to ANSI at the conclusion of their work.

The ANSI committee, BSR, that reviews draft proposed standards meets in the middle of October and the middle of December. If the appeal is resolved and the draft proposed C standard is submitted to them in time to be considered during the October meeting, we would, if there are no further problems, have an approved ANSI standard for C by November, 1989. If the October meeting is missed but the information is available for the December meeting, it will probably be January, 1990 before we formally have an approved standard.

If the appellant is not satisfied with the findings of the X3 Appeal board, he has the option of appealing yet again at the ANSI level. However, in most cases this appeal is processed after the draft proposed standard receives the American National Standard status.

The bottom line on all of this is that no one can give a firm date on when a C standard will be formally approved and in place.

September X3J11 Meeting Cancelled

Because of the continuing appeal process, the September X3J11 meeting in Salt Lake City has been cancelled. Since the standard has not been approved, it is inappropriate to begin the interpretations phase. This means that the primary reason for the meeting has been lost. At this point X3J11 does not plan to re-schedule the meeting; it will simply be skipped. The next scheduled X3J11 meeting is in March, 1990 in the New York City area.

Progress on the International Front

On the international front, progress is being made towards an International Standards Organization (ISO) C standard that is identical to the draft proposed C standard developed by X3J11.

The ISO standardization process requires two separate ballots. The first of these two ballots has been completed with no negative votes. This was a major milestone. This means that the document will be submitted for the second ballot (to be accepted as a Draft International Standard [DIS]). The DIS ballot has a 6 month voting period. At the current pace, it looks like there will be an international standard for the C language sometime during 1990.

The Passing of a Friend

Finally, I would like to pay tribute to Joan Hall who passed away on August 17th. As the wife and business partner of Tom Plum, Vice-chair of X3J11, Joan was a staunch supporter of our activities. Her uncanny insight and understanding of group dynamics helped X3J11 work its way out of, and sometimes even avoid, untold problems and difficult situations. She made life a lot easier for many of us and her contributions will be sorely missed.

Jim Brodie is the convener and Chairman of the ANSI C standards committee, X3J11. He is a Senior Staff Engineer at Honeywell in Phoenix, Arizona. 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) 863-5462 or uunet!aussie!jimb.

∞

19. Validation Suite Report

compiled by **Rex Jaeschke**

Introduction

With the final ANSI C Standard approaching reality the issue of conformance testing escalates in importance. As a result, I have invited vendors of commercial C language validation suites to participate in an electronic questionnaire. Those companies invited were Perennial, Plum Hall, and HCR/ACE. All three agreed to participate. Their collective replies follow in alphabetical order of company name.

Questions and Responses

Validation suite product name:

HCR/ACE: SuperTest

Perennial: CVS-A, C Compiler Validation Suite.

Plum Hall: The Plum Hall Validation Suite for C.

Company details:

HCR/ACE: Jointly developed by HCR Corporation and ACE Associated Computer Experts bv.

North America	*Europe*
HCR Corporation	ACE Associated Computers Experts bv
130 Bloor St. W.	Van Eeghenstraat 100
Toronto, Ontario	1071 GL Amsterdam
Canada M5S 1N5	The Netherlands
(416) 922-1937	(31 20) 6646416

Perennial:

Perennial
4677 Old Ironsides Drive
Suite 450
Santa Clara, CA 95054
USA
(408) 727 2255

Plum Hall:

Plum Hall Inc
1 Spruce Avenue
Cardiff NJ 08232
USA
(609) 927-3770

Contact person:

HCR/ACE: North America: Lynda Williams (uucp ... !hcr!lynda);
Europe: Technical – John van Brummen (john@ace.nl),
Marketing – Marco Roodzant (marco@ace.nl).

Perennial: Barry Hedquist, President; e-mail: ...sun!practic!peren!beh

Plum Hall: Thomas Plum; Fax: (609) 653-1903.
e-mail: plum@plumhall.uu.net or uunet!plumhall!plum.

Distribution media formats:

HCR/ACE: tape (1600 bpi), or special arrangements.

Perennial: tar tape, cartridge or 9-track.

Plum Hall: tar tape (1600 bpi), PC/AT diskettes, or special arrangements.

Pricing/licensing arrangements:

HCR/ACE:

- 60 day trial: US$4,500.
- Site license US$32,000, 1st additional site US$16,000. Corporate license: US$56,000.
- Trial period fee credited against purchases if decision made within 60 days.
- Technical support (annual) fee is 20% of product price, 60 day warranty.

Perennial:

- Source code license for site/machine type is $10,000. Additional sites and/or machine types are $5,000 each. (A machine type is a common system architecture or designated series, such as Sun 3 series, all 80386 based systems, VAX series, etc.) Custom licensing arrangements are available.

- Potential licensees may obtain a trial-use license, at no cost, for the purpose of evaluating the Suite for a specified period of time, usually 10-30 days, after which they may either purchase or return the Suite.
- New licensees (after Jun 1, 1989) are guaranteed to receive a full ANSI-C Validation Suite after adoption of the standard, whether or not it falls within the 12 month support window.
- Special pricing and licensing is provided to government agencies via the GSA Schedule.

Plum Hall:

- CONFORM section $4500 (tests all requirements of draft ANSI).
- Full Suite $9500 (also contains EGEN random expression generator, EXIN test-driver, and intensive compiler-test scripts). Proprietary source code.
- Licensed to a two-mile-radius site, unlimited number of machines.
- Maintenance provided for one year, and then available annually for no more than 25% of license fee.
- Even without maintenance, everyone will receive the Suite version that corresponds to the eventual finally-approved ANSI Standard C.

Technical support policy:

HCR/ACE: Call-in support (via fax, e-mail, or mail) for all licensed customers.

Perennial:

- Initial license includes 12 months of free technical support via electronic mail, telephone support, US mail, and FAX.
- After the first year, technical support and update service can be purchased for 25% of the paid up licensing fee per year.
- Perennial will also provide a low level of assistance, at no charge, for interpretation and analysis of the test results. Additional levels of assistance, including an independent validation test effort, can be obtained on a contract basis.

Plum Hall:

- Four hours of free consulting time.
- Any call that reports a bug is free.
- Responds to calls as soon as possible and always within 72 hours.

Update policy (frequency, cost, etc.):

HCR/ACE: Support includes updates (expected to be approximately once per year). Customers who do not acquire support will receive a single update at the end of the first year after original purchase.

Perennial: Supplied at no charge for twelve months. (See above.) Frequency varies, but licensees can expect at least one update during that period. As of June 1, 1989, Perennial guarantees licensees a fully compliant ANSI C Validation Suite, whether or not the final approval of the ANSI Standard and the availability of the Suite fall within the twelve month window.

Plum Hall: At least one major release per year. Notice given within two weeks of bugs reported by others.

How long has the suite existed?:

HCR/ACE: SuperTest is a combination and enhancement of test suites developed by HCR and ACE. The HCR suite was initially developed in 1985 and the ACE suite in 1984.

Perennial: Since 1984.

Plum Hall: Since May 1986.

Have the suite's developers been involved in ANSI/ISO C activities? If so, in what capacity and for how long?:

HCR/ACE: HCR has had staff representation on the ANSI C committee since December 1984. ACE joined the ISO/JTCI/SC22/WG14 committee in 1986.

Perennial: We have been involved with ANSI since 1986 primarily as observers to track the committee's activity. We get copies of all committee documentation.

Plum Hall: Ralph Ryan chaired the Environments Subgroup of X3J11 and Thomas Plum has been Vice-Chair since X3J11 started.

How many real paying customers do you have installed (not counting those "on trial")? 10 or less, 11–25, 26–50, more than 50?:

HCR/ACE: The total installed base of the HCR and ACE test suites is in the 26–50 range. The combined test suite, SuperTest, will have first customer shipment late this summer.

Perennial: More than 50.

Plum Hall: More than 50.

How many different translators have you exercised the tests on?:

HCR/ACE: This process has not been completed. Historically the number exceeds 10.

Perennial: We've lost count of the number of compilers we've developed on and run on in house. Current in-house configurations total is 7 systems with 9 compilers. (All of these systems are UNIX or UNIX-like systems.)

Plum Hall: More than 50.

What is the format of your tests? Do they include user-generated scripts as well as hard-coded tests? Compile-time, link-time and run-time tests?:

HCR/ACE: Conformance tests are hard-coded with embedded macro and function calls that support the interface. Depth tests have the same hooks but are machine generated. Compile-time, link-time, and run-time testing is supported.

Perennial:

- The Suite is run by a driver that is written entirely in C source code and tailored for UNIX and UNIX-like environments. It creates several processes and makes use of `exec()` in running the Suite. The driver allows the user many options to control the test run, such as performing compilation or execution only testing; selecting which test or groups of tests to run; measuring CPU performance while running the Suite; restarting the Suite at the last test executed, or any other location; running a single test, or the entire Suite, continuously; sending the results to a specified directory; and others.
- All test programs are in C source code, and test for compile-time, link-time, and run-time results. The test programs use a set of library functions provided with the Suite to control the format and generation of the test results. These allow exact and concise reports to be produced for the features being tested.
- The tests are grouped into several directories to provide focused testing of specific compiler functionality. The groupings are: basic data types, compound data types, control flow, combinations of data types, conversions, functions, libraries, operators and expressions, preprocessor, syntax, and environmental limits.
- The Suite performs both positive and negative testing, including negative compile-time testing, as well as providing performance measurements of individual test programs.
- The Suite also contains a template and instructions so that users can easily add their own tests to the Suite to test proprietary features of their own, or others', implementation of the language, such as

optimization. The driver may then run any set of tests the user chooses.

- There are no user-generated scripts or expression generators in the Suite since the use of such a tool is not validation per se.

Plum Hall: In CONFORM (standards-conformance only): No user-generated scripts. All are C source files: one per draft section ("positive-tests"), one demonstration program for translation limits ("capacity-test"), plus one per each required diagnostic message ("negative-tests"). Tests are compile-time (valid code must be produced) as well as run-time (expected answers must be produced). In Part 2 (compiler-testing sections): several hundred megabytes of user-generated scripts.

Are the tests completely automated or is "significant" customization needed for each particular compiler? Can specific or individual groups of tests be run?:

HCR/ACE: Essential details such as compiler options are prompted for up front by the user interface. All additional configuration by the customer is strictly optional. Tests may be run in groups, individually, or all at once. Failed tests from a previous run are grouped by the system for easy resubmission.

Perennial: The Suite is totally automated, including the installation process. No significant customization is required. The user may insert certain flags in the Suite's makefile such as BIGENDIAN, or change the compiler call from `cc` to something else if called for by the compiler under test, such as testing a cross compiler or a third party compiler. The driver allows the user to run the entire Suite, a selected group of tests, or an individual test. The selected tests can then be run once or continuously until stopped by the user.

Plum Hall: To validate a bug-free compiler that completely conforms to the ANSI Standard, no customization is needed. Simply compile, link, and execute the positive-tests (celebrating the absence of run-time error outputs) and compile the negative-tests (noticing that a diagnostic message is produced in response to each bad file). To the extent that the compiler is buggy or non-conforming, selective customization may squeeze out further information about the problems. It is extremely easy to split the positive-tests into the subsections of the Standard, to test only Sections 3.1, 3.2, ..., 3.8, or 4.1, 4.2, ..., 4.12, for example.

How much code/how many tests are provided?:

HCR/ACE: The tests are in two groups, a breadth suite and a depth suite. The breadth suite consists of more than 1000 files performing over 6000 tests. The depth suite has over 100,000 machine generated tests.

Perennial: The Suite contains over 600 test programs with over 3000 individual test cases. The Suite requires slightly less than 5 megabytes of disk space prior to installation, and can expand up to 10 megabytes after the first run, depending on the number of errors found during the test and the volume of reports generated. We advise users to allocate 15 megabytes of disk space for the Suite to handle multiple runs, report output, etc.

Plum Hall: CONFORM: over 20K lines of carefully hand-coded C, plus over 30K lines of widely-tested machine-generated C, together comprising 1.9 megabytes. About 27,000 run-time tests are performed. Full Suite comprises over 3.3 megabytes, and generates hundreds of megabytes from scripts provided.

What is the format of the test results? Do I get a listing of pass/fail, a table in order of the sections of the Standard or what?:

HCR/ACE: Four flags control the style of output. In the most verbose mode, each test generates a line giving a brief description followed by either FAILED or PASSED. Multiple test results are listed under a paragraph heading identifying the directory where the tests are found, which corresponds to the matching paragraph of the standard where applicable. (A few tests involving behaviors such as quiet changes or "torture testing" do not map to a single section of the standard). Tests which failed to compile, dumped core, compiled when they should have failed, or correctly failed to compile are appropriately flagged.

Perennial: Test results are generated in several types of reports all located in the 'Results' directory (default), or a directory name selected by the user. The report types are:

- report_file: one file that lists each test program, and whether the compilation and/or execution passed or failed.
- test_name.co: a file created for each test that generates compile-time or link-time output of any kind. If a test fails to compile or link properly, the error messages would be found here.
- test_name.ex: a file created for each test that generates an execution-time message that could be categorized as standard output or standard error.
- test_name.log: a file created for each test as a result of output intentionally generated by the test program execution. These files contain specific information on what is being tested, and why a test failed during execution.
- test_name.pcc: a file created for each test program when running in the the 'profile' mode, a driver option to measure CPU utilization, containing any error messages created during the compilation phase.

- test_name.pro: a file created for each test program when running in the profile mode to measure CPU utilization. This file reports the total CPU time used by the test program, lists the functions that were called, how many times they were called, and the percentage of CPU time each function contributed to the total CPU time used.

Each test program contains a cross reference to the ANSI draft and/or K&R, as applicable. A manual is also provided with a complete cross reference for each test program for the appropriate ANSI and/or K&R reference.

Plum Hall: Positive tests: four brief pass/fail summaries (number of tests passed, list of tests failed). Capacity-test: one pass/fail message. Negative tests: one checklist of required diagnostics.

Are POSIX C extensions conformance tests also provided/available?:

HCR/ACE: There are no immediate plans for support of POSIX extensions.

Perennial: Active in IEEE POSIX P1003 since 1986. Now working with P1003.3 Conformance Testing Subgroup. As such, we will support testing of the POSIX C extensions, and make those tests available in a future version of our C Validation Suite.

Plum Hall: Not at this time.

How do you test environment-specific things such as multi-byte character and locale support? How do you test `system()`*?:*

HCR/ACE: Environment-specific things, in the breadth section, are tested as far as their behavior and syntax is defined by the ANSI standard.

The depth component copes with certain crucial machine dependencies by generating target-specific code, which is installed in the suite before shipping.

Perennial: Testing for multi-byte characters and locale support is under development, and not yet supported. Tests for these areas will be available in a future release.

Because the Suite is aimed at UNIX and UNIX-like environments, we test `system()` by using it to call a UNIX command, such as `ls`, checking the return status of the call, redirecting the command output to a file, and checking the contents of the output.

Plum Hall: We only validate the (sometimes rather minimal) conformance requirements of the Standard itself. We test the syntax and perform those tests which must work in every environment. Strict conformance requires only the presence of a `system()` function.

Describe the documentation set. How do I use it to interpret test messages?:

HCR/ACE: SuperTest comes with a manual describing various aspects of the different tests, how to interpret the results, how to install the validation suite, and how to interact with the user interface.

Perennial: The documentation set consists of both hard copy and on-line documents. Hard copy documents are:

- C Compiler Validation Suite Description Document—provides an overview of the Suite, how it is structured, what's needed to run it, what's tested, a list of the test programs, sample test code, sample output, and a walk through of a sample test run.
- User's Guide—provides specific instructions on how to install the Suite, how to run it, the various options available for running the Suite, how to interpret the test results, and how to make use of tools provided with the Suite, such as the template and `gthdrs`, a shell program that gathers the test program headers into a single file.
- Test Programs/Standards Cross Reference Manual—cross reference for each test program with the ANSI Standard, and K&R as applicable.
- Configuration List—complete list of each test program in the Suite, its version number and last date of modification. This list is provided with each release of the Suite.
- Release Notes—documents changes to the Suite since the last release, and is accompanied by the Configuration List.

On line documentation includes:

- User's Guide—same as hard copy version but not as pretty.
- Schematic of Suite layout/directory structure.
- README files in each major directory describing its content and purpose in the Suite.
- Header information in every test program which provides the test name, last date of modification, modification history, a description of what is being tested and how, any known dependencies for the test, and the ANSI and/or K&R reference for the features tested. (an example is "Must have BIGENDIAN flag set, if your system is BIGENDIAN, in order for this test to pass.") A program supplied with the Suite allows the user to gather these headers into a single file which may then be printed out.

Plum Hall: The 38-page manual describes how to use each section. From the quality assurance point of view, messages are just Boolean—"Something is wrong in the handling of section *x.y.*" To interpret the underlying compiler bug or feature that caused the reporting of any particular error, the compiler developers themselves need to read the source code for the case that failed, compare the expected result with the actual result, etc.

A source code test suite can validate a compiler. However, all implementation-defined behavior must be documented by a vendor to actually be conformant. What assistance do you provide to help with this task?:

HCR/ACE: Tests exist for implementation-defined behavior where the possible set of results is small and pre-determined, or gross misconduct of an unacceptable nature can be easily spotted (e.g., core dump). There are currently no plans to further assist the developer in this area.

Perennial: Perennial will provide a documentation audit to any Suite licensee on a contract basis to verify that all implementation defined behavior is documented. This effort would result in a report generated by Perennial listing all such implementation-defined behavior, the documentation reference, a description of the behavior as documented, and a report of omissions or deficiencies for those behaviors not properly documented.

Plum Hall: No direct assistance. This is somewhat outside the scope of a source code suite.

OK, now it's your turn to brag. Describe the attributes of your suite that make it the best available. No need to write complete and correct sentences, just give me the major points:

HCR/ACE: Merger of two successful existing suites, one specializing in ANSI conformance and one in torture testing, revised to support ANSI.

An interface which produces informative reports and demands minimal user configuration. At the same time it allows the user to exercise considerable control through optional parameters. Failed tests may be resubmitted conveniently.

Supported by two software houses with extensive experience in building compilers. HCR's test suite is a proven commodity, both in-house and on the market. ACE's experience in building C compilers and supporting its test suite customers has fed an ongoing improvement process, especially sensitive to practical features. The suite has been exercised on mc680x0, iAPX 386, VAX ND500, ND5000, ns16000 and many other architectures, as well as cross compilation environments.

Tests for correct handling of quiet changes from ANSI C Rationale as well as covering the Standard section by section.

Perennial: Precise—Broad yet concise validation testing of K&R and ANSI features. Totally automated.

Robust—The Suite will not 'hang' or stop prior to completion, regardless of test failures. Longevity—existed since 1984, used world wide, and continues to evolve.

Portable—To any UNIX or UNIX-like environment. Flexible—Users may run single tests, selected groups of tests, or the entire Suite. Expandable—Tools provided to facilitate development and testing of user generated tests for addition to the Suite.

Concise Results—Test reports allow immediate identification of failure cause without having to wade through volumes of output.

Test Methodology—Tests are true 'validation' methodology, using only known and concise input parameters, and looking for only known and concise output in a total system environment.

Suite supports cross compiler testing, performance measurements of CPU utilization.

Plum Hall: Accurate. Thorough. Impartial. Not connected with any vendor of compilers. Up-to-date. Authoritative.

Designed from the very beginning for the purpose of validating the ANSI C Standard. Comes with strong technical support. Written to cover the Standard sentence-by-sentence. Doesn't favor any particular environment or operating system. Provides negative-tests (error-tests that test for required diagnostics).

We won the British Standards Institution's competition for the Suite to be used officially in Europe.

Used for several years by over a hundred diverse compiler implementers, specifically for the purpose of tracking the ANSI C Standard.

∞

20. Pragmania

Rex Jaeschke

Macros in Pragmas

The ANSI Standard says nothing about the expansion of macros in pragmas, so we have the expected divergence of implementations. Some do, some don't.

Having looked at quite a few pragmas, it occurs to me that expanding macros in pragmas would likely be a useful feature. For example, consider the following pragma:

```
#pragma pack ( n );
```

where n indicates a structure member alignment value of 1, 2, 4 or 8. (This pragma, or variations of it, has been implemented by several popular compilers.) If n is permitted to be a macro that expands to any compile-time integer constant expression, it might provide more utility.

Taking the idea one step further (as users always seem to do), expanding macros in pragmas could then give rise to pragmas such as:

```
#pragma DO_IT
```

where `DO_IT` was conditionally compiled or provided via a compiler option. Now, it no longer is obvious from the source which pragma is being specified. In fact, the macro may expand to nothing, in which case it is presumably ignored as it is not recognized.

If you are contemplating expanding macros in pragmas, I suspect you will also need to provide an information message when a pragma goes unrecognized since incorrect macro definition could easily result in an unintended outcome.

Cray Standard C Compiler 1.0

The following section is reprinted with permission from Cray Research, Inc. This material is extracted from their manual, *Cray Standard C Programmer's Reference Manual*, © Cray Research, Inc., 1989.

Vector Dependencies

```
#pragma ivdep
```

This directive tells the compiler to ignore vector dependencies for the immediately following loop; the loop then vectorizes if all other conditions are satisfactory. This directive is especially useful for use with loops that have pointer references. Specifying the `-h ivdep` option on the command line that invokes the compiler, however, takes precedence over any `#pragma ivdep` directive encountered during compilation.

Each of these directives control vectorization attempts only for the loop it immediately precedes.

Loop Vectorization

```
#pragma novector
```

This directive tells the compiler not to vectorize the immediately following loop. The `#pragma novector` directive overrides any other vectorization-related directives as well as the `-h vector` and `-h ivdep` compiler options.

Code Optimization

You can control optimization of your source code by specifying a line of either of the following forms:

```
#pragma opt
#pragma noopt
```

This directive tells the compiler to optimize (`opt`) or not optimize (`noopt`) the source code following the directive. This directive can appear only outside function bodies.

Vectorizing Reduction Loops

You can control whether reduction loops are vectorized with a preprocessor directive of the form:

```
#pragma noreduction
```

Suppressing Register and Local Memory Use

```
#pragma suppress
```

This directs the compiler to store all variables in memory rather than retain them in a register or local memory.

Identifying Short Vector Loops

You can identify vector loops that execute in 64 iterations or less with a preprocessor directive of the form:

```
#pragma shortloop
```

Use of this directive allows the compiler to generate more efficient vector code.

The Tasking Pragma

The `taskloop` pragma discussed elsewhere in this issue (see the article by Holly) has been implemented in Cray's version of pcc as an experiment. If it proves to be successful, it will also be implemented in their Standard C compiler.

∞

21. A European Conformance Testing Service for C

Neil Martin
British Standards Institution

Abstract

The British Standards Institution (BSI), building on the experience of its established Pascal Validation Service, is currently leading a joint project to develop the framework for a European conformance testing service. This service will validate C language translators against the requirements of the ISO/ANSI Standard for C. The basis of this service, as in other language validation services offered by BSI, is a Compiler Validation Suite. This paper outlines BSI's involvement in compiler testing and the process used to select a C Validation Suite for the harmonised European validation service.

Compiler Services at BSI

BSI's major function is to support British industry by providing both national standards and a range of quality assurance services which contribute fundamentally to the trading success and well-being of the British business community. It also offers information services on standards and regulations world-wide and a special advisory service for exporters. BSI originated in 1901 and was the first national standards body in the world. There are now more than 80 similar organisations which belong to the International Organisation for Standardisation (ISO) and the International Electrotechnical Commission (IEC). BSI represents the views of British industry on these and on other bodies which work towards harmonising world standards.

In 1984, aided by the National Physical Laboratory (NPL), BSI set up a validation service for Pascal compilers. This service has since become truly international and is used in the USA, Japan, Italy, France, and numerous other countries. Bilateral agreements between the various international test houses ensure that certificates are recognised all around the world, with the result that BSI carries out a substantial number of validations in North America for companies in need of validation certificates under federal procurement requirements.

Since the development of the Pascal Validation Service, BSI has gained extensive experience in the testing of compilers. In addition to conventional language conformance testing, BSI also uses and distributes test suites for quality assessment of compilers. These include static checkers, floating point analysers and the even more conventional benchmarking. With its prior experience in compiler testing, BSI was quick to recognise the commercial significance of C in the market place and the need for formal verification procedures following the publication of a standard. BSI had the practical experience to develop such procedures. Hence the development of the necessary infrastructure began prior to the finalisation of the standard.

The CTS-2 Programme

During the early eighties, a programme was established by the Conference of European Postal and Telecommunications Administrations CEPT and the Joint European Standards body CEN/CENELEC to harmonise the development and use of Information Technology (IT) standards within Europe. (The organisation CEN/CENELEC is essentially the European equivalent of ANSI, sometimes producing standards and other times adopting suitable international ones.)

It is generally recognised that the new generation of IT standards will not be effective without verification methods and independent agencies to operate them. Standards without formal independent verification tend to lack credibility. The European Commission, in support of this need, set up a conformance testing service (CTS) programme to provide harmonised test services throughout Europe. This programme supports the provision of testing services and the development of test tools when necessary. The programme covers OSI protocols, software quality, programming language validation, and graphical systems (GKS and CGI).

Harmonisation is achieved by ensuring the use of the same or equivalent test tools and reporting procedures by all the testing laboratories. The first phase of CTS concentrated mainly on the OSI arena, however the second phase widened the scope of the CTS programme. It was under this second phase that BSI applied, and obtained funding, for a joint European C conformance testing service, the so called *CTS C project*. In addition to BSI, this project includes compiler testing laboratories from two other European Nations, namely Association Francaise de Normalisation (AFNOR) of France and Instituto Italiano del Marchio di Qualita (IMQ) of Italy. Both of these labs also participate in Pascal validation.

At the outset of the project it was obvious that C, though still without a formal standard, was a relatively mature language. As such, more than a few test suites were already in existence. Some of these test suites were following the (draft) ANSI X3J11 standard as it emerged. Therefore, it was decided not to finance the development of a new test suite, but to carry out an evaluation procedure to select an existing suite. The initial task was simply to identify as many commercial test suite vendors as possible. After much searching of various journals, software reviews, and contact within the C user community, a list consisting of the following companies was drawn up:

Validation Suite Vendors	
Name	*Location*
Softlab Gmbh	W.Germany
HCR Corp.	Canada
Metaware Inc.	USA
Nixdorf AG	W.Germany
ACE BV	Netherlands
Plum Hall Inc.	USA
Perennial Inc.	USA
R.A.P.A Inc.	USA
Softron Inc.	USA
Xopen	Multinational

BSI approached each of these companies, giving details of the proposed European Validation Service, and inviting them to participate in the selection process. Most of the companies were forthcoming with literature, which was usually followed up with lengthy discussion. It must be added that in some ways this process was disappointing as the majority of the test suites were in fact in-house compiler regression test suites dressed up as products. However, BSI had carefully established comprehensive criteria for the selection process. This aimed to ensure fairness to the test suite vendors, to establish long term acceptability to the C user community, and to utilise the experience of the Pascal Validation Service. The BSI criteria can be broken down into two lists, loosely classified as commercial and technical, as follows:

- Commercial

  1. The cost (to a customer) of the C test suite must be suitably low so as to encourage as many implementers/users as possible to purchase the suite.
  2. The owner of the test suite must be willing to modify the test suite, as necessary, in order to make it suitable for validation purposes. In addition, an indefinite maintenance programme for the CVS must be agreed upon.
  3. The owner of the test suite must be willing to take part in an open, independent review process.
  4. The owner must have no vested commercial interests in compiler products.
  5. Title and ownership of the suite must be clear under all circumstances.

- Technical

  1. The suite must give comprehensive coverage of all aspects of the (draft) C Standard.

2. There must be clear correlation between the standard and any particular test in the suite.
3. Both the test suite and the supporting documentation must be clear, concise, and easily maintainable.

These criteria, along with a demonstrably broad user base and the willingness of the vendor to become involved in a formal verification process, produced two prime candidates. At this point BSI enlisted the help of a leading UK compiler house, Lattice Logic Ltd. This company had a track record that included validating Pascal compilers on 12 hosts. They were also in the process of developing a C compiler. Lattice Logic carried out a thorough technical review of the two prime candidates, concentrating on the suitability of the test suites as a base for a Validation Service.

The outcome of this process might not be considered a politically sound choice for a European Validation Service, since the suite selected was that of Plum Hall Inc., of the USA. However, this clearly was the outstanding suite seen during the selection process and, not surprisingly, appeared to be the only suite developed from scratch for the purpose of testing for conformance to the ANSI C standard. In addition to the suitability of the actual test suite, Plum Hall had the added advantage of having been intimately involved in the standardisation process. (Their Chairman, Dr. Plum, serves as the Vice-Chair of X3J11.)

Since the selection process was completed, the Plum Hall suite has greatly expanded its visibility in Europe, adding to its already large user base. In addition, the suite has undergone several revisions and improved markedly on the version that was originally reviewed. The current release is 1.09, and version 2.00 will be that which matches the final version of the standard. Furthermore, in an attempt to ensure complete coverage of the C standard by the suite, BSI is in the progress of repeating a "completeness" exercise such as was carried out on the Pascal test suite. BSI has enlisted the help of a C compiler vendor, Knowledge Software Ltd, to produce a model implementation C compiler. This compiler will allow us to obtain execution profiles when running the test suite. It is also intended that the compiler issue messages whenever any portability issues arise either at compile- or run-time.

Requirements for Validation

Application for validation may be an alien process to some C compiler vendors, especially those not involved with other languages. However, there is nothing particularly sinister about the process. The first and most important step is that the vendor obtains a copy of the test suite prior to testing, and ensures that their compiler completely passes the suite. Once the vendor is satisfied that this is the case, the appropriate test laboratory is informed and arrangements made for a full witness testing. The full test involves a qualified member of the test laboratory staff formally identifying the test suite and witnessing the successful passing of all the test suite by the compiler.

The complete details of the compiler and run time system under which the test took place are recorded for entry onto the validation certificate. In addition to passing the test suite, C compiler vendors will be required to demonstrate that they have satisfied the documentation requirements of the standard. All implementation-defined features, along with any features which are undefined in the standard but have been defined by the vendor, must be fully documented. The final requirement is that the vendor makes a formal statement of compliance, which takes the following form:

Validation Suite Compliance Example

Doodle C Version 9.1 Fast Clone model 1, (Intel 80286 plus 80287 floating point processor) OS USDOS 2.0
The above C language processor complies with the requirements (by means of a compiler switch) of ANSI X3J11/88-165/ISO 9899.

C System Components

Compiler:	Doodle C version 9.1
Interpreter:	
Linker:	Doodle Link version 10.5
Preprocessor: (if separate)	

Mode of Use

Compiler options used:	"Standard" switch, medium memory model, hardware floating-point

Hardware Identification

Host Computer:	Fast Clone Model 1
CPU:	Intel 80286
Operating System:	USDOS 2.0

Target Computer

Host Computer:	Fast Clone Model 1
CPU:	Intel 80286
Operating System:	USDOS 2.0

Validation Suite:	Plum Hall CVS 2.00

Conclusion

Compiler validation often is a procurement requirement by large agencies. The most important of these is the U.S. Government, which has already announced by means of the Federal Register Vol. 54 (Jan 27, 1989) their intention to require validation for C compilers. This means that in the near future it will be necessary for a C compiler to be validated before it can be sold to any U.S. Government Agency. However, although the US government (in the form of the National Institute of Standards and Technology, NIST) intends to validate C compilers, it has yet to select a test suite. BSI already has close ties with NIST and has offered to help NIST in any way it can, but as yet NIST has not made clear what criteria it will use to select a test suite for validation. However, from our perspective the C vendor community already appears to be leaning towards the Plum Hall suite. And those that can pass it advertise the fact, making it not only a standards issue but also a marketing one.

At BSI we are confident that the Plum Hall C test suite is the most appropriate suite available for a formal validation service. This, along with its widespread usage, would seem to make it an ideal choice for an international harmonised testing service. It will be surprising to us if NIST comes up with any reasonable criteria that leave the Plum-Hall test suite as other than the front runner.

Neil Martin is a Senior Software Engineer in the BSI Quality Assurance group at BSI in Milton Keynes, United Kingdom. He may be reached electronically via neil@uucp.bsiqa. In late June he represented the UK at the CTS-2 meeting held at NIST. This meeting was attended by delegates from NIST, UK, France, and Italy.

∞

22. Miscellanea

compiled by **Rex Jaeschke**

Quality of Implementation

Frequently during ANSI C deliberations a proposal failed to make it into the standard because it was felt the issue was "one of quality of implementation." One example is the format and spelling of diagnostic messages. Another is whether or not informational or warning messages are produced.

From time to time I'll be documenting interesting and useful examples of quality enhancements, primarily to entice other implementers to follow suit. Since this information will be based on my own experience and I only get to see and use a small subset of implementations, I encourage vendors to send me a copy of their documentation set so I may become aware of their quality extensions as well.

DEC's VAX C

I've worked with numerous of DEC's compilers since 1974. They have all had one thing in common—they produce compilation listings. And their VAX C compiler is no exception.

Now I don't want to get into the whole argument about how a compiler's job is to generate code and (possibly) diagnostics only, and that anyone can write a listing formatter. Sure they can, but users are in the business of writing applications *not* tools. If they have to write them themselves or buy them from another vendor, the chances are they will go without. Also, certain very useful tools require you write tokenizers, parsers and even preprocessors. So the more components contained in a development toolkit, the better off the user will be.

Writing a listing formatter is easy. Writing a C preprocessor is not, and that's where implementers can help. Here's how. When a C programmer starts to use complex macros that expand to call other macros, etc., wouldn't it be nice to be able to see both the original and the expanded output at the same time. (I've used several implementations that provide this capability and I know I love it.)

Many preprocessors allow their output to be saved in an intermediate disk file. However, this file is designed such that it can be submitted to the C compiler. As such, it cannot contain any formatting or other information unless it is contained in C style comments. So the approach a user must use to view both source and preprocessor output is to look at two files, the source and the intermediate file produced by the preprocessor. Unfortunately, in the real world most developers don't have adequate multi-window screen capabilities. At best they have a 24-line screen with limited split screen editing. And even if you can look at both files at once, it's hardly the same as having the information properly merged in one file.

Enter the compilation listing file. Not only can it contain the formatted source listing, it can also provide useful information about preprocessor flow. For example:

```
CC DEMO1/LIST/SHOW=EXPANSION

 1         #define M1    5
 2         #define M2    (M1 + 10)
 3         #define M3(a) (M2 * a)
 4
 5         void f()
 6         {
 7    1          int i;
 8    1
 9    1          i = M3(10);
         3       i = ((5 + 10) * 10);
10    1    }
```

The /LIST compiler option causes a listing file to be produced and the qualifier /SHOW=EXPANSION causes the final output from the preprocessor to be merged in the listing, as shown. (The number 3 at the start of the expanded line indicates the number of intermediate expansion steps.)

I have used one other implementation that provided this capability, and that is Power C from MIX, a DOS-based compiler that sells for something like $30.

Once users get a taste of this capability, they inevitably want to be able to see the intermediate levels of expansion too. After all, if they are debugging a complex, nested macro they need to see how the preprocessor expands at every step. Again, VAX C has this capability. For example:

```
CC DEMO1/LIST/SHOW=INTERMEDIATE

 9    1          i = M3(10);
         1       i = (M2 * 10);
         2       i = ((M1 + 10) * 10);
         3       i = ((5 + 10) * 10);
```

And since all diagnostic, warning, and informational messages also are merged at the appropriate source lines, locating the corresponding problem is much easier.

Another benefit of a listing file is that it can be used to identify which conditional compilation path was used for a given set of compilation criteria. For example:

```
CC DEMO2/LIST

 1       #ifndef M1
 2             char c[10];
 3       #     ifdef M2
 4   X                 int i = 5;
 5       #     else
 6                     int i = 10;
 7       #             ifdef M3
 8   X                         int j[5];
 9       #             else
10                             int j[10];
11       #             endif
12       #     endif
13       #endif
```

Lines preceded with an X were not included in the compilation.

Another feature provided by VAX C is the ability to define function-like macros on the compiler command-line. (All other implementations I have seen allow only object-like macro definitions.) For example:

```
CC DEMO3/LIST/SHOW=EXP/DEFINE=(VALUE=5,SQR(a)=((a)*(a)))

  1      #include <stdio.h>
142
143      main()
144      {
145    1       int i = VALUE;
          1    int i = 5;
146    1
147    1       int j = SQR(i + 10);
          1    int j = ((i + 10)*(i + 10));
148    1
149    1       printf("i = %d, j = %d\n", i, j);
150    1 }
```

To be sure, you have to play a few tricks to circumvent the VAX/VMS command-line processor (to pass in quotes, preserve casing, and handle white space, for example) but at least you do have the capability. I see no technical reason why this capability cannot be provided by other implementations.

Another advantage provided in VAX C's listing file is the ability to get a list of all identifiers in alphabetical order, a listing of all lines where identifiers are referenced, and an English description of the identifier's type. For example:

```
double d[3][5][6];
int f2(int);
void (***pf)(void);
int *ap[5];
int (*pa)[5];
int (*signal (int sig, void (*func)(int)) ) (int);
```

`ap`	Array [5] of pointer to `long int`
`d`	Array [3][5][6] of `double`
`f2`	Function returning `long int`
`pa`	Pointer to array [5] of `long int`
`pf`	Pointer to pointer to pointer to `void` function
`signal`	Function returning pointer to function returning `long int`

Note that, on the VAX, `int` and `long` map to the same representation, so any `int` type is actually reported (incorrectly) as `long`.

When structures and unions exist, information is also provided about each member as follows:

```
struct tags {
        long int i;
        double d;
} s;

union tagu {
        long int i;
        double d;
} u;

s          Struct tags
tags       Structure tag
 i         Member (offset = 0), long int
 d         Member (offset = 4 bytes), double
tagu       Union tag
 i         Member (offset = 0), long int
 d         Member (offset = 0), double
u          Union tagu
```

Calendar of Events

- September 19–20, Minneapolis, MN, **Numerical C Extensions Group (NCEG) meeting** – The second meeting will be held to consider proposals by the various subgroups formed at the Minneapolis meeting in May. **NOTE THAT THE LOCATION HAS BEEN CHANGED FROM SALT LAKE CITY.** Contact Rex Jaeschke at (703) 860-0091 or uunet!aussie!rex.

- September 21–22, Salt Lake City, **ANSI C X3J11 meeting** – **THIS MEETING HAS BEEN CANCELLED** due to the processing of an appeal lodged by a member of the public. The appeal questions certain procedural issues.

- October 10–12, **Frontiers of Massively Parallel Computation** – To be held at George Mason University, Fairfax, Virginia. Cosponsored by IEEE. Conference chair is James Fischer at NASA, (301) 286-9412.

- November 13–17, the **Supercomputer 89** conference – Reno, Nevada. Tom MacDonald of Cray Research, Inc., is organizing a workshop on Friday the 17th. The topic is **Scientific and Numerical Programming in C**. The focus of the workshop is to discuss issues involved with using C for numerical and scientific applications. This includes programming techniques used to achieve high performance on supercomputers and any issues involved with using C on supercomputers. This also includes C derivatives such as C++, C*, and Objective C.

  Please contact Tom at (612) 681-5818, tam@cray.com, or uunet!cray!tam. if you have any interest in making a presentation or if you know of someone else that might.

- January 22–26, **Winter 1990 USENIX Technical Conference** – Location: Washington, D.C. at the Omni Shoreham Hotel. Call David Klein on (412) 268-7791 or wash-usenix@sei.cmu.edu for details.

- March 5–6 1990, New York City, **ANSI C X3J11 meeting** – Sponsored by Farance, Inc. This one-and-a-half day meeting will handle questions from the public, interpretations, and other general business. Address correspondence or enquiries to the vice chair, Tom Plum, at (609) 927-3770 or uunet!plumhall!plum.

- March 7–8 1990, New York City, **Numerical C Extensions Group (NCEG) meeting** – The third meeting will be held to consider proposals by the various subgroups. It will follow the X3J11 ANSI C meeting being held at the same location earlier that week. Contact Rex Jaeschke at (703) 860-0091 or uunet!aussie!rex.

News, Products and Services

- The *Journal of Parallel and Distributed Computing* is a bi-monthly publication from Academic Press, Inc. ISSN 0743-7315. Subscription cost within USA and Canada is $186, elsewhere it's $234. Publisher's address is 1 East First Street, Duluth, MN 55802. Telephone (800) 543-9534 or (218) 723-9828.

- The public uucp node UUNET is offering its extensive collection of source on tape. The set includes X-Windows, TEX, and the complete GNU system. Call them on (703) 876-5050 or info@uunet.uu.net for details.

- Copies of proposed and final ANSI standards (including our own X3J11 ANSI C effort) are available from:

  Global Engineering Documents, Inc.
  2805 McGaw Avenue
  Irvine, CA 92714
  (800) 854-7179
  (714) 261-1455
  Telex: 62734450

- **Lattice, Inc.**, is shipping its ES68 C Development System for embedded programming on the 68K family. It's hosted on numerous systems. Contact John Nelson on (312) 916-1600.

- **Cobalt Blue** is shipping FOR_C++, which converts Fortran-77 to either C or C++. Runs on Sun-3 and Xenix/386. Call them on (408) 723-0474.

- **Silicon Valley Software** is shipping its SVS/C compiler for Sun-3 and Sun386i systems. Call Beau Vrolyk on (415) 572-8800.

- **King Computer Services, Inc.** has announced a C cross development system for Radio Shack's Model 100/102 portable computer. Call Helen Budlong on (213) 661-2063 for information.

- **OASYS** is distributing **Green Hills'** compilers to the **NeXT** machine. Call Ann Bishoff on (617) 890-7889.

- Looking for a **cheap** C development system for DOS? Try **Power C** from Mix Software. The full-blown compiler is $19.95 as is the symbolic debugger. Source to the run-time library is $10 as is the BCD library, prices that are hard to beat. (214) 783-6001.

∞