The Journal of

C Language Translation

Volume 2, Number 1

June, 1990

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 2051 Swans Neck Way, Reston, Virginia 22091 USA. Telephone (703) 860-0091. Electronic mail address via *uucp* is *uunet!aussie!jct* or *aussie!jct@uunet.uu.net*.

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

1. Sequence Points and Parallelism

P.J. Plauger

Abstract

The ANSI standard for C introduced the concept of *sequence points*. These clarify the minimum ordering requirements on side effects within a program. They also emphasize the extent to which C overdetermines the order in which side effects must occur. Introducing parallelism by flow analysis is therefore made all the more difficult.

This article reviews the ordering imposed by sequence points both within and between expressions in a C program. It suggests several ways by which indeterminism can be added to C to aid generation of parallel executing code. It can be viewed as an alternate approach to solving the aliasing problem with pointers.

Introduction

A creative tension always exists in the definition of a practical programming language. On the one hand, programmers need to know with some certainty what a program will do. Give the implementor too much latitude and portability goes out the window. You have to experiment with each implementation to see what it does with a particular program.

On the other hand, implementors need some latitude in translating a program. Restrict the implementor too much and performance goes out the window. You have to experiment with each implementation to see how best to make it run fast enough. Almost invariably, you sacrifice readability and portability with this sort of tuning.

Programmers are very clear about how to resolve this tension. They expect an implementation to provide the ultimate in optimization for performance. Except when they don't want to lose determinism. A programmer can tell an implementor on a case by case basis what he wants. The only trouble is, he may want different behavior for the same code sequence in different programs. Or in the same program on different days of the week.

A translator can perform all sorts of flow analysis. It can keep track of the values stored in data objects and where pointers can point. It can endeavor to second guess the programmer's intent. But it can only go so far in rewriting a program. It still must produce visible results *as if* the program were unchanged.

And it must not spend too much time thinking, lest customers abandon it as too slow.

Even in a buyers' market, programmers do not always get their way completely. Beyond a certain point, programmers must provide some help. They may have to choose options at translation time that alter the semantics of the program. They may have to insert `#pragma` directives to convey extra information. They should certainly be called upon to decorate certain declarations with the `volatile` type qualifier.

Programmers can do all this—some already do—and find that the result is still not what they hoped for. Machines that perform vector operations are hard put to decide when it's safe to unleash their might. Programs written in FORTRAN fare better than the most carefully crafted versions in C. If C is to thrive on the growing number of parallel architectures, if it is to lure FORTRAN programmers in large numbers, it must parallelize better.

I believe that the fundamental problem with C is that it is overly deterministic. It evolved as a systems implementation language on a typical minicomputer. It flourished as a higher-level replacement for assembly language. As a consequence, C puts more emphasis on predictability than it does on optimizability. *Peephole* optimizations are strongly favored over global optimizations.

Global flow analysis has gotten steadily smarter over the years. The ANSI standard has provided optimizers a few assists. Any number of companies have added various ad hoc extensions to let the programmer indicate where parallel operation can occur. Nevertheless, C offers little latitude for introducing parallelism.

One approach to modernizing C is to take a hard look at the current level of determinism. That can highlight places where the language can be extended, or amended, for maximum benefit. It leads to solutions that are more general and less ad hoc. The goal is to find new ways for the programmer to help the optimizer. We certainly don't need to make existing C programs any less deterministic.

I begin by reviewing the concept of *sequence points* in C. The determinism they provide is a fundamental restriction on the semantics of the language. Existing opportunities for parallelism are interesting, but not widespread. Hence, the main thrust of this article is to explore ways to introduce more salutary indeterminism into C.

Sequence Points Within Expressions

Tom Plum forced committee X3J11 to think about sequence points. He did so by leading us down the garden path on several occasions. I review a few of those trips here.

We were all quick to agree that an expression of the form `X || Y` was rigorously sequenced. The program must first evaluate the scalar subexpression `X`. If it is nonzero, then the scalar subexpression `Y` is guaranteed not to be evaluated.

If it is zero, then `Y` is evaluated only after `X` is completely evaluated.

What do we mean by "completely evaluated?" A telling, if frivolous, test case is the expression `0 * putchar('\n') || getchar()`. Certainly, the program can determine the value of the left subexpression without first calling `putchar`. It could omit calling the function at all, or it could defer the call until after the call to `getchar`.

But both of these potential optimizations are perilous. Part of the complete evaluation of the left subexpression must involve writing the newline character. That action had better be complete before the right subexpression is evaluated. To be complete, it is not enough to determine the value of a subexpression. The program must also ensure that all side effects have occurred.

This example shows one kind of side effect, altering the state of a stream. An even more common side effect is an assigning operator that alters the value stored in a data object. A call to any library function has the potential of altering a value stored in some secret static location. In fact, a call to practically any function at all can do the same. It takes a pretty smart translator to track the effect of an arbitrary function call. Most translators simply assume that all function calls generate side effects.

Plum introduced the concept of sequence points to pin down when side effects must occur. The expression `X || Y` has a sequence point after the evaluation of its left operand. All side effects associated with that operand must occur before the sequence point. If the right operand is evaluated, all of its side effects must occur after this sequence point. The firewall is clean and well defined.

A few other operators also contain sequence points. The expressions `X && Y` and `X ? Y : Z` are well known examples. They are often used explicitly to control sequencing within an expression. Sometimes overlooked is the comma operator `X, Y`. It guarantees left to right evaluation of its operands. And the function call operator `X(Y)` has a sequence point after `X` and `Y` are evaluated, just before the function is called.

For some time, Plum felt that an assigning operator also had some kind of sequence point. You might think that `X = Y` cannot store a value until the right operand is completely evaluated. This is not necessarily so. Consider a variation on the earlier frivolous expression, `x = 0 * putchar('\n')`. The committee felt that it was proper for a translator to rewrite this as `x = 0, putchar('\n')`. So in the end, the committee decided that assigning operators should not contain sequence points.

The function call can also precede the assignment, of course. In fact, the operands can be evaluated in pieces and the pieces arbitrarily intermixed. If side effects interfere with each other, the behavior is labelled as undefined. The translator is not in error. That effectively means that the operands can be evaluated in parallel. In the example above, the function call can even occur in parallel with the assignment. (I admit that few translators exploit this license.)

Shuffling the evaluation of subexpressions may seem like a rash thing to do. The committee spent many hours trying to contrive horrors that are patently

unacceptable. We tried to word various constraints on evaluation order. We even toyed with having some way to control evaluation order with parentheses. Luckily, all those efforts eventually ran down and died. It was hard to find a smoking gun, a problem so clearly bad that it had to be fixed.

As a consequence, C permits considerable latitude in how a translator evaluates an expression. The operators with sequence points impose a partial ordering on the process, to be sure. Nevertheless, opportunities abound for rearranging the order of evaluation.

There are fewer real opportunities for introducing parallelism. Pipelined processors can often overlap operations. Smart translators routinely perform *instruction scheduling* to maximize such overlap. That is microscopic parallelism, however. The next step up in granularity these days usually involves vector operations or microtasking. Neither of those architectures helps much when climbing over a single expression tree. A translator usually has to consider several expressions at a time.

Sequence Points Between Expressions

Once you start looking at several expressions at a time, things get quickly more constrained. A sequence point exists at the end of every whole expression within a program. That makes at least four sequence points in a simple `for` statement:

```
for (i = 0; i < N; ++i)
        a[i] = b[i] * c[i];
```

What's good about this is that the state of the data object `i` is extremely well defined. You can even determine its value after the loop exits. It also lets you write all sorts of loop controls besides the usual arithmetic progression. You can walk a list:

```
for (p = first; p != NULL; p = p->next)
        ...
```

You can take nonlinear steps:

```
for (i = 0; i < MAX; i = 2 * i + 1)
        ...
```

Or you can iterate until convergence:

```
for (x = y; fabs(x * x - y) < 1E-6; x = xnew)
        xnew = f(x);
```

Many other procedural languages are far less helpful.

What's bad, of course, is that discovering vector operations involves optimizing across several firewalls. Even figuring out how many times the loop

executes can take nontrivial analysis. And tinkering with the loop control often involves a significant reordering of side effects. (A loop can terminate *only* as the result of a side effect that alters the value of the test expression.) Moreover, data objects used to control looping in C have no special status (unlike in other languages). It is permissible for the body of the loop to alter the stored value as well.

Still another problem is the nature of the repetition. It is precisely specified by the semantics of C. That means that all sorts of funny situations have a well defined answer, whether you want one or not. Consider the first example of a `for` loop above:

```
for (i = 0; i < N; ++i)
        a[i] = b[i] * c[i];
```

If the vector based at `a` overlaps either `b` or `c`, it is not safe to rearrange the order of execution of the iterations. That means it is not safe to perform them in parallel.

Most discussions of this situation treat it as an aliasing problem. If the three vectors are visibly declared as arrays, a translator usually can confirm that they do not overlap. When they are based on pointers, however, the analysis gets harder. If the pointer values are passed into a function as arguments, it can be impossible to check. Distinct arguments that are aliases for the same actual vector can cause problems.

Committee X3J11 tried and failed to solve this aliasing problem. The failure was rather spectacular, in fact. We introduced a third type qualifier called `noalias` at the eleventh hour. Then we tried to sort out its myriad and subtle semantics in haste. So great was the uproar that we yanked it right back out of the language. All that cost us an extra public review period.

Past experience indicates that really new concepts take several meetings to get debugged. I confess that aliasing semantics are still too murky for me to fathom. Even as one of the initial proponents of the machinery, I'm now appalled at the implications. I think it would take several years to sort it out properly. That's why I'm eager to explore any mechanism that sidesteps the need for machinery such as the `noalias` type qualifier.

Array Expressions

A natural way to help a translator vectorize C is to add vector expressions to the language. Several companies have done this already. The Numerical C Extensions Group is exploring quasi standards, to minimize dialects. I confine my remarks here to more general principles.

First, it is not that hard to promote arrays to first class objects in C. It is a simple extension to permit array assignment and functions returning array types. The only sensible way to pass arrays as arguments is in the scope of a function prototype.

Sadly, X3J11 felt moved to state explicitly what happens when you write `int f(char a[10])`. The array type is altered to a pointer type in the good old fashioned way. We weren't completely forced to do this, since prototypes are something new under the sun. It's just that arrays cause so much confusion to C programmers. We were afraid to plough new ground. But the fact remains that we did it. The conversion must be rescinded to permit the declaration of array parameters in the obvious fashion.

Some proposals sidestep the problem by introducing new notation for arrays that are to be treated as arrays in toto. You write something like `int f(char a[0:10:1])` to spell out lower bound and stride as well. This has the advantage of being a pure extension instead of a change in meaning for existing programs. It also lets you specify fancier slices of arrays. What it costs, of course, is considerably more complexity both at translation time and at runtime.

If arrays are no handier than structures and unions, you gain little. More useful than simple assignment is the ability to write `a = b + c` and have the operations performed on an element by element basis. (I don't propose wiring in the conventional mathematical rules for array multiplication and inversion. They are neither necessary nor sufficient for many applications.)

The benefit of writing vector operations this way is not just the shorthand, although some might love it. Rather, the notation gives the translator license to perform the operations on elements in whatever order it chooses. That means that overlapping vectors yield unpredictable results. So be it. That's the license a vectorizer needs to improve parallelism.

One additional component is required for this notation to be truly useful. It must be possible to declare arrays with dimensions that can be determined only at runtime. It is a rare function that knows the size of all its vector parameters at translation time.

Again, several people have proposed notation for such an extension. Dennis Ritchie has proposed one of the more simple and powerful approaches (no surprise). He suggests that flexible array declarations be confined to type casts. That avoids the messy business of allocating and freeing automatic storage for flexible arrays.

I find that none of the declaration problems are insurmountable for flexible arrays. Partly it is a matter of choosing a sensible notation. Partly it is a willingness to generate whatever code is necessary to support credible semantics. So long as good old fashioned fixed size arrays are not penalized, language designers have considerable latitude.

Parallel Statement Execution

Attractive as they are, array expressions solve only part of the problem. Many forms of parallelism don't happen to look like vector operations. Some vector operations are not easily captured as array expressions. C is still too deterministic in many situations.

Take a look at any large chunk of C code. Often you will find two or more statements enclosed in braces. They are written in a certain order, but the choice is arbitrary. Often you can rearrange a sequence of statements several different ways and achieve the same semantic effect. You may know that, but how hard is it for an optimizing translator to reach the same conclusion?

What I think we need is some way to write a set of statements without specifying an explicit order of execution. The effect would be similar to the `PAR` statement of Occam, though perhaps not quite so pedantic. For want of any better notation, I add vertical bars to the usual braces. (Think of them as parallel lines.)

```
if (done)
        {|
        printf("all done\n");
        done = 0;
        total += subtotal;
         |}
```

As with array expressions, the new braces come with a strong promise. None of the statements in the enclosed sequence has side effects that affect any of the others. If it does, the result is undefined. In other words, the translator is left off the hook in the event of failure. It is encouraged to be aggressive in reordering code or introducing parallelism. This may not be enough to encourage a translator to introduce microtasking, but it certainly helps.

An important variant is needed for loops. We need some way to encourage a C translator to reorder the iterations of a loop body or to execute them in parallel. Here, the semantics gets stickier because of the side effects in the loop control. Consider a notation something like

```
for (| i = 0; i < N; ++i |)
        a[i] = b[i] + c[i];
```

This promises to decouple the expressions that sequence the loop from the statement that forms its body. The translator determines the number of iterations, and the values stored in `i`, by the usual sequencing rules. It is then at liberty to execute the loop body for the iterations in arbitrary order. The value stored in `i` must be correct for each iteration, but it is essentially of a `const` type. Should the loop body change its stored value, the behavior is undefined.

The notation need not be confined to the `for` statement, by the way. It makes just as much sense to write (assuming that `i` is a signed type)

```
i = -1;
while (| ++i < N |)
        a[i] = b[i] + c[i];
```

The same is also true of the `do-while` looping statement.

I don't pretend to have thought through all the issues with these constructs. I'm certain, in fact, that hammering out the semantics will take much careful thought. I'm reasonably certain, however, that the problem is more soluble than declaring the aliasing properties of pointers.

Conclusion

C can benefit in several ways from the addition of a bit of indeterminacy. By letting the programmer show where sequencing can be relaxed, the language can ease the burden on translators to discover opportunities for parallelism. Given the proliferation of pipelined, vector, and multiprocessor machines, C needs all the help it can get in this area.

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. His latest book Standard C, *written with Jim Brodie, is published by Microsoft Press. He can be reached at uunet!plauger!pjp.*

∞

2. Parsing Initializers

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

Abstract

The requirements for initialization are hard to specify because of the many terms and rules that need to be defined before the specification is complete. Although the syntax is simple and straightforward, the semantic rules for interpreting initializers are complicated. Rules for initialization are given along with a consistent way of viewing what is being initialized. Finally, an algorithm is presented that provides the basic machinery needed to correctly interpret initializers.

Introduction

One problem with describing the rules for parsing initializers is that there are many terms that need to be defined before the rules can be defined, and these terms are often defined in terms of the rules. Therefore, a description of the problem being solved is a lengthy discourse. This subject is further complicated by the diversity of implementations that exist. Some implementations parse initializers *top down* and others *bottom up*. This article discusses the ANSI standard and its top down approach.

A description of initializers starts simply and gets complicated quickly. The best way to describe this complexity is through an example. What kind of initialization does the following produce?

```
struct rec {
   int m1;
   double m2[2];
} ary[2][2] =
   {
      {{1}, 2, {1.0, 2.0}},
      {3, {3.0}, 5, 7.0, {8.0}}
   };
```

The integral and floating-point constants give hints about their intended destination. The goal of this article is to describe an algorithm that correctly interprets both fully and partially bracketed initializers.

A good place to start is with the syntax productions in the grammar (§3.5.7):

initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* , }

The innocuous yet annoying trailing comma requires an extra production to support.

initializer-list:
initializer
initializer-list , *initializer*

The syntax is quite simple. From a purely syntactic point of view it is easy to parse. The complexity arises from the semantics of the { and } tokens. Sometimes the { token is interpreted as a redundant (and optional) brace and other times it starts an initializer list. Sometimes the } token is interpreted as a redundant (and optional) brace and other times it ends an initializer list which causes all remaining (and therefore, uninitialized) members to be implicitly initialized to zero.

Simple initializers

Every textbook on C contains examples of the following form. They are mentioned here for completeness.

```
int i = 3;                /* like simple assignment    */
float j = 2+3;            /* OK, assignment compatible */
char *p = "abcdefg";      /* p points to 'a'           */
int a[] = { 1, 2, 3, 4 }; /* a's dimension is 4        */
char c1[4] = { 'a', 'b', 'c', '\0' };
char c2[] = "abc";        /* same as c1 above          */
char c3[3] = { 'a', 'b', 'c' };  /* no trailing zero   */
char c4[3] = "abc";       /* same as c3 above          */
```

Textbooks that describe ANSI C also include examples similar to the following:

```
wchar_t wc1[4] = { L'a', L'b', L'c', L'\0' };
wchar_t wc2[] = L"abc";    /* same as wc2 above       */
```

```
union {
   int x1;
   float x2;
} x = { 2.0 };             /* x.x1==2 */
```

The last example shows that a union can be initialized via its first member. However, none of these examples demonstrates initializers with nested { and } tokens. That is the primary subject of this paper. Textbooks are oftentimes content with describing fully bracketed and minimally bracketed initializers. Since textbooks are oriented toward users this is justifiable in that existing C compilers interpret partially bracketed initializers differently. Therefore, a portable program should use full or minimal bracketing.

A word of caution about what you read in text books. It is not uncommon to find errors in text books. It is not uncommon to find errors about initialization. One book I read, that claimed to describe the ANSI C standard, contained the following example:

```
int example[2][3][4] = {
        5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
        { 2, 3, 7 } };
```

The book claimed this is equivalent to entering:

```
int example[2][3][4] = {
     { 5, 0, 0, 0 },
     { 1, 2, 0, 0 },
     { 5, 2, 4, 3 },

     { 9, 9, 5, 0 },
     { 2, 3, 7, 0 },
     { 0, 0, 0, 0 }
};
```

Not only aren't these equivalent, they are both invalid! Since the first initializer, 5, is not preceded by a {, it is assumed to be the initializer for `example[0][0][0]`. A conforming implementation must therefore diagnose the `{ 1, 2 }` because there are two values being assigned to `example[0][0][1]`. One or more pairs of braces can inadvertently be omitted and yet the example can still conform to the specified syntax but not, however, conform to either the constraints or semantics specified in the ANSI standard.

Zorts

One problem with understanding initializers containing nested { and } tokens has to do with nomenclature. The standard defines an aggregate as a structure or array. If what is being initialized is an aggregate or union then the { token

forces the translator to look inside the aggregate at its elements or members, and the } forces the translator to initialize remaining elements and members in the aggregate with zeros. If a remaining element or member is itself an aggregate or union then its elements and members are also implicitly initialized. Since these discussions center around objects, elements, and members of structures and unions, a term is needed to describe what is being initialized when looking inside an aggregate. This paper uses the term *zort*. The reason the term zort is used is to avoid the confusion with preconceived notions about names that the usual nomenclature presents. For instance (in §3.5.7 page 73, line 24) the ANSI standard contains the following sentence:

> "If the aggregate contains members that are aggregates or unions, or if the first member of a union is an aggregate or union, the rules apply recursively to the subaggregates or contained unions."

An aggregate contains subzorts (i.e., elements or members). When the translator looks inside the aggregate to initialize its subzorts, these subzorts turn into zorts. A zort then may contain its own subzorts. For purposes of this discussion, a union can be considered to be an aggregate that contains one initializable subzort. The above sentence is then replaced with:

> If the aggregate contains a subzort that is itself an aggregate, the rules apply recursively to the subzort.

My assumption is that there are no preconceived notions about zorts.

Rules and Definitions

Declared scalar objects and declared aggregate objects are called *declared objects*. A declared object is not a zort. The following declarations are used in many of the examples below.

```
struct st {
   int m1;
   float m2;
};

struct tag {
   struct st n1;
   int n2;
};
```

A declared object is not a zort because there are different rules for declared objects than for zorts. For instance, a declared object having automatic storage duration can be initialized with a nonconstant expression. However, a scalar zort must always be initialized with a constant expression.

```
{
   auto int i = j + k;
   struct st x2 = { i };        /* error */
}
```

Here then are the rules.

1. Only a declared object can be initialized with an = token followed immediately by an initializer.

```
typedef int INT = 3;    /* error */
```

2. A declared object is either a scalar or an aggregate (including unions).

```
int i;          /* scalar */
int a[3];       /* aggregate */
```

3. A brace enclosed list begins with { and ends with } and contains one or more initializers separated by commas.

```
/* brace enclosed lists */
{ 1, 2.0 }
{3}
{ }             /* error */
```

4. An initializer is either an expression or a brace enclosed list.

```
/* expression initializer */
int i = 3;

/* brace enclosed list initializer */
struct st x = { 1, 2.0 };
```

5. Array elements, structure members, and the first member of a union are collectively called zorts.

```
struct st x;
/* zorts:  x.m1, x.m2 */

struct tag y;
/* zorts:  y.n1, y.n1.m1, y.n1.m2, y.n2 */
```

6. A zort is either a scalar or an aggregate.

```
/* scalar zorts:  x.m1, x.m2, y.n1.m1,
        y.n1.m2, y.n2 */
/* aggregate zort:  y.n1    */
```

7. Declared objects and zorts are initialized with initializers, and the expressions associated with scalars must be assignment compatible.

```
int i = 2;      /* expression */
int j = "abc";  /* error - not compatible */
```

8. The initializers of aggregate zorts do not necessarily begin with a {.

```
/* initialization of zz begins with { */
/* initialization of zz.n1 does not   */
struct tag zz = { 1, 2.0, 3 };
```

9. Declared objects with static storage duration and zorts must be initialized with constant expressions.

```
static int i = j;  /* error */
```

10. If an aggregate is initialized with a brace enclosed list then all expressions in the list must be constant expressions.

```
{
   auto struct st z3 = { i, j };  /* error */
}
```

11. An initializer for an aggregate cannot be enclosed in redundant braces.

```
{  /* auto scope */

   struct st x = { 1, 2.0 };

   /* errors */
   struct st y1 = { x };
   struct st y2 = {{ 1, 2.0 }};
}
```

12. If the initializer for an aggregate is a brace enclosed list then the list contains (comma separated) initializers for the aggregate's subzorts.

```
struct tag a[2] =
{               /* a */
   {               /* a[0] */
      {               /* a[0].n1 */
         1,              /* a[0].n1.m1 */
         2.0             /* a[0].n1.m2 */
      },              /* a[0].n1 */
      3               /* a[0].n2 */
   },              /* a[0] */
   {               /* a[1] */
      {               /* a[1].n1 */
         4,              /* a[1].n1.m1 */
         5.0             /* a[1].n1.m2 */
      },              /* a[1].n1 */
      6               /* a[1].n2 */
   }               /* a[1] */
};              /* a */
```

13. If the initializer for an aggregate does not begin with a { then only enough initializers from the current list are consumed to initialize the aggregate's subzorts.

```
/* initializes subzort a[0] and subzort a[1] */
struct tag a[2] =
   {
   1, 2.0, 3, 4, 5.0, 6
   };
```

14. If there are too few initializers in an aggregate's brace enclosed list then remaining subzorts within the aggregate are initialized to zero.

```
/* x.m1==1 && x.m2==0.0 */
struct st x = { 1 };
```

15. The number of initializers in a brace enclosed list must be less than or equal to the number of zorts being initialized.

```
/* error - too many initializers */
struct st x = { 1, 2.0, 3 };
```

A large part of initialization is spent consuming a sequence of constant expressions followed by commas. When a { token is encountered a decision must be made. If the current declared object or zort being initialized is an aggregate then the { token causes the translator to start initializing the aggregate's subzorts. However, if a scalar is being initialized then the { is a redundant (and

optional) token, but there must be one scalar expression in the initializer list and a matching } token. This expression is used to initialize the scalar zort or declared object.

This means that the { token never forces the translator to skip the current scalar zort and all subsequent scalar zorts until the next aggregate zort is found. However, the } token does force the translator to advance through the remaining subzorts of the current aggregate being initialized. Each subzort is initialized to zero. For example:

```
struct xtag {
   int x1;
   int x2;
   int x3[2];
   int x4;
};

struct xtag x = { 1, { 2 }, { 3 } };
```

This initialization is equivalent to the following initializer:

```
struct xtag x = { 1, 2, 3, 0, 0 };
```

but is not equivalent to:

```
struct xtag x = { 1, 0, 2, 0, 3 };
```

because the { 2 } initializer does not force look ahead to the next aggregate zort. With these rules and examples the original example can be examined again.

Top Down View

The key to viewing any initializer is to look at the aggregate from the top down. An aggregate can be viewed as having a directory-style structure. There is a path to all of its subzorts. The original example is presented and followed by a top down-view of the initialization. All paths and initializers are identified.

```
struct rec {
   int m1;
   double m2[2];
} ary[2][2] =
   {
      {{1}, 2, {1.0, 2.0}},
      {3, {3.0}, 5, 7.0, {8.0}}
   };
```

```
ary
{ ary[0]
  { ary[0][0]
    { ary[0][0].m1 = 1
      ary[0][0].m2
        ary[0][0].m2[0] = 0.0 (implicit)
        ary[0][0].m2[1] = 0.0 (implicit)
    }  /* ary[0][0] */
    ary[0][1]
      ary[0][1].m1 = 2
      ary[0][1].m2
      { ary[0][1].m2[0] = 1.0
        ary[0][1].m2[1] = 2.0
      }  /* ary[0][1].m2 */
  }  /* ary[0] */
  ary[1]
  { ary[1][0]
      ary[1][0].m1 = 3
      ary[1][0].m2
      { ary[1][0].m2[0] = 3.0
        ary[1][0].m2[1] = 0.0 (implicit)
      }  /* ary[1][0].m2 */
    ary[1][1]
      ary[1][1].m1 = 5
      ary[1][1].m2
        ary[1][1].m2[0] = 7.0
        ary[1][1].m2[1] = {8.0}
  }  /* ary[1] */
}  /* ary */
```

This top down representation allows the initialization to be viewed in a straightforward manner. This approach is used to define an algorithm for initialization.

The Algorithm

The following algorithm is for consuming and interpreting initializer tokens, and is written in a pseudo C style. It provides the basic machinery needed to implement initialization. There is an assumption that the type `OBJECT` is a data structure capable of representing the object being initialized.

```
typedef ..... OBJECT;

init_object (OBJECT object)
{
```

```
   if ( object is scalar )
      if ( token == '{' )  /* redundant */
         token = get_next_token();
         init_object (object);
         if ( token == ',' )  /* trailing */
            token = get_next_token();
         endif
         if ( token == '}' )
            token = get_next_token();
         else
            error(" '}' token expected ");
         endif
      else  /* token != '{' */
         if ( auto scope )
            init_scalar_exp(object);
         else
            init_scalar_const(object);
         endif
      endif
   else  /* aggregate */
      if ( token != '{' )
         if ( auto scope )
            exp = get_expr();
            gen_assignment(object, exp);
         else
            error(" '{' init-list '}' expected ");
         endif
      else  /* token == '{' */
         init_zort(object);
      endif
   endif
}

init_zort(OBJECT zort)
{
   if ( zort is scalar )
      if ( token == '{' )  /* redundant */
         token = get_next_token();
         init_zort(zort);
         if (token == ',')  /* trailing */
            token = get_next_token();
         endif
         if ( token == '}' )
            token = get_next_token();
         else
```

```
            error(" `}' token expected ");
         endif
      else  /* token != `{' */
         init_scalar_const(zort);
      endif
   else  /* aggregate */
      if ( token != `{' )  /* open list */
      for ( every subzort in zort )
         init_zort(subzort);
         if ( last subzort in aggregate )
            break;  /* leave possible comma */
         endif
         if ( token == `,' )
            token = get_next_token();
         endif
         if ( token == `}' )
            break;  /* no more inits */
         endif
      endfor
      init_remaining_zorts_to_zero();
      else  /* token == `{' */
      token = get_next_token();
      for ( every subzort in zort )
         init_zort(subzort);
         if ( token == `,' )
            token = get_next_token();
            if ( token == `}' )
               break;  /* no more inits */
            endif
         else  /* token != `,' */
            break;  /* No more inits */
         endif
      endfor
      if ( token == `}' )
         init_remaining_zorts_to_zero();
         token = get_next_token();
      else  /* token != `}' */
         error (" `}' expected ");
      endif
   endif
}
```

```
init_scalar_const(OBJECT zort)
{  /* consumes expression tokens */
   exp = get_const_expr();
   gen_init(zort, exp);
}

init_scalar_exp(OBJECT zort)
{  /* consumes expression tokens */
   exp = get_expr();
   gen_assignment(zort, exp);
}
```

This algorithm will traverse the directory-style aggregates in a top down fashion through recursive calls to `init_zort`.

There are a few assumptions and omissions in this algorithm. It does not specify how to handle initializers such as:

```
char ca[4] = "abc";
```

where the string literal must be broken apart. The action, `for (` *every subzort in object* `)`, does not address how to compute the size of an array declared with an unknown size. It assumes that unnamed bit-fields are skipped when proceeding to the next subzort, that unions contain only a single initializable subzort, and that there is at least one initializer in a list.

Enhancements

There are many enhancements that can be added to initialization. The following is a list of some enhancements that can be added that are beyond what the ANSI standard requires:

- replicate a particular value N times
- initialize just the Nth element of an array
- initialize automatic array elements with nonconstant values
- initialize automatic structure members with nonconstant values

Finally, there is an initializer-related feature called *compound literals* that Dennis Ritchie is proposing to the Numerical C Extensions Group (NCEG). This feature allows initializer-like expressions to be used in statements. (This capability is already provided in GNU C and is called a *constructor expression.* Dennis has suggested that the term constructor should not be used for this capability since that name has different connotations in C++.) For example:

```
struct tag {
   int m1;
   float m2;
};

func () {
   int i = 3;
   float x = 2.0;
   struct tag rec;

   rec = (struct tag) { i, x };
}
```

The construct `(struct tag) { i, x }` creates a value with type `struct tag` that is computed at execution time.

Conclusions

Although there are many rules associated with initialization of the variety of data structures that C provides, they can be viewed in a consistent way. One straightforward approach is to use recursion, which is exploited in the algorithm presented. This algorithm is not particularly long nor complicated and provides most of what is needed to implement C initialization. C initialization can be extended in many ways while still exploiting the majority of this algorithm.

Finally, I would like to thank my colleague David Knaak, and Dave Prosser at AT&T, for their insights and observations about C initialization.

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.

∞

3. Electronic Survey Number 4

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 questions were posed to 50 different people, with 21 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.

Locales

Have you implemented or do you plan to implement locales other than the required `"C"`*? If so, for what purpose?*

- 13 – No plans at this stage
- 3 – Have them now
- 4 – Plan to in the future
- Comments:

  1. We are actively pursuing business in Europe and Asia.
  2. We plan to implement these soon after the native C compilers we stay compatible with do so. Reasons: compatibility and international sales.
  3. We will support a UK locale whatever that ends up being.
  4. We will implement other locales if customers request them and offer to help us "get it right."
  5. True internationalization requires locales, and that's what all the current ones are for. POSIX will require a POSIX locale which is an extended C locale.

     The POSIX locale is detailed in 1003.2 which is currently in ballot in IEEE. 1003.2 is big, and the stuff is scattered throughout it. (1003.1a

will also require a POSIX locale, but in that case it's [at least so far] just another name for the `"C"` locale, to allow a system that's both 1003.1a and 1003.2 to use a single locale name.)

6. We plan to implement a set of locales for various European and other languages.

A `sizeof`/Cast Puzzle

What do you think the following expressions should produce? Are the parentheses part of the `sizeof` *operand or do they represent the cast operator? Note that the tokens* `-` *and* `&` *can be either unary or binary operators.*

```
/* A */    sizeof (char *) 0
/* B */    sizeof (double) -1
/* C */    sizeof (unsigned) &p
```

- 12 – case A is an error and B and C contain binary operators (This is the correct answer.)
- 1 – All are errors
- 1 – All are correct
- 6 – Don't know for sure
- Comments:

  1. This a "How well do you know your standard?" question. These particular constructions were highlighted in K&R I's appendix A, page 188 in which the reader is told that `sizeof (`*type*`)` is taken as a unit. The K&R I grammar did not specify this behavior unambiguously. This is one of the changes that ANSI C's grammar includes. Consider the following:

     unary-expression:
     postfix-expression
     `++` *unary-expression*
     `--` *unary-expression*
     unary-operator cast-expression
     `sizeof` *unary-expression*
     `sizeof (` *type-name* `)`

     Since *unary-expression* cannot begin with a `(` *type-name* construction (*postfix-expression* can get to a parenthesized expression, of course) there is no ambiguity. Thus cases B and C above contain binary operators and case A cannot be constructed by the grammar and thus requires a diagnostic.
     Note that this was *not* the behavior of PCC.

2. These expressions are either illegal, or the parentheses are part of the `sizeof` operand; I'm not sure which. In either case, a really good translator would warn here about possible ambiguity or nonportability.
3. There is no ambiguity whatsoever in applying the Standard's grammar to example A, since there is only one production starting at *expression* and ending at that token sequence. However, for cases B and C the Standard's grammar leads to multiple productions, and Footnote 32 (Dec 1988 draft) or Footnote 35 (final Standard) explains how to resolve the ambiguity, namely:

   unary-expression `:= sizeof(` *type-name* `)`

   has higher precedence than

   cast-expression `:= (` *type-name* `)` *cast-expression*

   so the correct parse involves binary `-` for B and binary `&` for C. This happens to be consistent with the table on page 49 of K&R 1st Ed.; what a remarkable coincidence!
4. Very interesting question. As a user, I am not sure what I would like. As an implementer, I am. Currently, our compiler accepts the cast as the full operand of the `sizeof`, which is itself part of a larger expression. It does this consistently for all three expressions. Changing this would be a real beast, and probably slow our compiler down, as it would seem to involve significant look-ahead in parsing.

Saving Preprocessor Output

Do you permit the output of the preprocessor to be saved for inspection? Do you provide more information than the traditional pcc .I file? (A merged listing of output with source, lines selected by conditional compilation, for example.)

- 2 – Can't save output as a file
- 17 – Can produce equivalent of a .i file
- Comments:

  1. We do not provide anything beyond the preprocessed source, however, we do save all macro definitions for use by the debugger.
  2. There are no "listing" options. However, there are tools that, given a program compiled with debugging information, will produce listings with assembly language, etc. Not many programmers seem to use these, though.

3. During parsing, if we get a syntax error inside one or more macro expansions, we show all the macro definitions involved. Otherwise, no.
4. Yes. Currently just the pcc .I file. Because of the PP-tokenizing, I go out of my way to restore white-space and physical lines, etc. Lines selected by conditional compilation would be a nice extra.
5. The points you made on this issue in your *Quality of Implementation* on the VAX C compiler in *Volume 1, number 2* were good ones.
6. Error messages are inserted. Lines that are `#ifdef`'d out are not listed. Preprocessor lines that are executed are left in. (This is unusual.)
7. By default the listing file is the C source after the preprocessor has got at it. The information provided in the listing is a bit sparse and we have plans to improve it. There is a rather fancy scheme for connecting error messages to the tokens they are flagging. We have had a lot of positive user feedback on this feature.
8. I don't think that this is a reasonable constraint on a Standard-conforming implementation, because it slows down the translation process considerably, unless normal compilation uses a different mechanism, in which case one has to maintain a larger translation system just to support this one feature. However, many UNIX programmers have come to expect this capability and even abuse it for purposes totally unrelated to C! This makes it politically infeasible to eliminate the feature, although I think that would be the "right" thing to do.

   I don't believe in program listings in the conventional sense. The development environment should have strong enough symbolic support to permit interactive source-level debugging and even quick modification, in which case listings as such are pointless. While there are a few occasions on which it would be handy to view the result of translation phase 4, in general I can do fine without it, particularly when the development system accrues other benefits (such as faster operation) thereby.
9. We provide more information than the traditional pcc. It is, for example, possible to:
   - keep the control lines that evaluate to FALSE. The control lines with an empty body are useful to document where sections of code have been removed.
   - ignore all `#define` and `#undef` directives. In this case the define statements are printed as they stand.
   - don't include the contents of headers.
   - don't even scan headers.
   - and many more.

C++ Futures

Do you plan on supporting C++? (No interest; Maybe; Plan to; Already do.)

- 1 – No interest at this time
- 7 – Maybe
- 6 – Plan to
- 6 – Already do
- Comments:

    1. We plan to, but not until the language stops changing.
    2. Already do, but don't recommend it as a language due to its gratuitous incompatibility with C and its grammatical ambiguity.
    3. We would like to, but have not had demand to suggest it would be commercially worth our effort.

Narrow Function Arguments

Consider the following prototype:

```
void f(char, short, float);
```

Do you pass any of these arguments in their narrow form as permitted by ANSI C?

- 5 – Use old-style widening
- 5 – Keep all narrow
- 4 – Keep `float` only, narrow
- 1 – Irrelevant since same size
- Comments:

    1. It's optional. We want to be backwards compatible with existing libraries, etc.
    2. It's optional.
    3. "As permitted?" Isn't it required? *[Ed: Provided both the prototype form function declaration and definition are used and they specify the same narrow type for an argument, the implementation can please itself whether widening is done.]*
    4. We currently pass `char` in narrow form.

Handling of Ellipses Arguments

Do you use a different argument passing mechanism for fixed and variable length argument lists? Specifically, can a call to `printf` *or* `scanf` *go wrong with no prototype in scope? Can the programmer specify the calling mechanism (by using a pragma, for example)?*

- 16 – No difference
- 3 – Difference
- 2 – Can specify call mechanism
- 11 – Can't specify call mechanism
- Comments:

  1. We use the standard argument passing mechanism for `printf` and the like. The lack of a `printf` prototype being in scope does not cause any problems. It was felt that a simple program with a call to `printf` that didn't use `<stdio.h>` or have a prototype in scope should operate as expected.

     We do have pragmas to tell the compiler what calling mechanism to use on a particular function that is being called. We do not provide pragmas that allow the user to specify what calling mechanism a function definition should be created with `, ...` yet. Currently to get other than the normal calling mechanism the function to be called must be written in assembly language.
  2. Even for machines in which the argument passing would have been greatly simplified by behaving in a different manner given a `...` function, we cannot do so. Compatibility is too important. There are no user-available variations in calling conventions.
  3. Our mechanism for using variable length argument lists affects only the callee. Therefore, prototypes are not required for correctness.
  4. Unfortunately our calling standard, as dictated by 88Open, is an unmitigated disaster. The convention for deciding if aggregates are to be passed in a register, or on the stack, is complicated, and sub-optimal.

     As a consequence of our calling standard, we have trouble when derivative non-native language processors try to use `stdarg.h`. For example, Cfront can't pass structures by value in variable length argument lists. We do support aggregate-pass-by-value in run-of-the-mill C programs.
  5. Yes, for C++ or Pascal function types. No, for C function types. The mechanism can be controlled by command line switch, using the `cdecl` and `pascal` keywords, or using the C++ `extern "..."` syntax.

6. We went to a lot of trouble to make sure that the caller does not have to know that the function has a variable length argument list.

   Functions with fixed and variable length argument lists use the same calling sequence, which was designed primarily for efficiency of fixed argument lists and we went to some trouble to make it possible for functions expecting variable length argument lists to work.

   We pass the first few integral and pointer arguments in integer registers and the first few `float` and `double` arguments in double registers. But we always leave room on the stack for the maximum number of arguments that might be put in registers. Leaving room doesn't cost much since we bump our stack pointer only on entry to functions.

   The `...` callee stores the argument registers into the space reserved for them on the stack and can then walk through the argument list more or less normally. Our `va_list` is a structure containing a pointer into the argument list on the stack and counters of how many integer and floating values have been `va_arg`ed. An interesting consequence of this is that:

   ```
   printf("%d %f\n", 1.5, 5)
   ```

   prints `5 1.5`. `printf` doesn't know which order the 1.5 and 5 were specified in the caller, so it assumes they were in the "right" order. We certainly don't advertise this "feature," but as far as I know nobody has complained either.

7. In some of our environments we use a single argument passing mechanism, but in others we use multiple mechanisms. The motivation for multiple mechanisms is usually (a) cross-language call support or (b) performance gain from passing floating arguments in floating registers. We've made sure `scanf` and `printf` still work. So far we haven't needed to provide pragmas for programmer-specified calling sequences.

Future Polls

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, and publish them in a future issue. You don't need to have an e-mail address to propose topics, only to be polled.

∞

4. FORTRAN to C: Numerical Issues

Fred Goodman
PROMULA Development Corporation
3620 North High Street, Suite 301
Columbus, Ohio 43214

Abstract

MacDonald [1] contrasts numerical programming in FORTRAN with that in C. He gives extensive discussion of floating-point issues; however, he only mentions integer expressions. When converting numerical FORTRAN programs to C, real problems also occur in the conversion of fixed-point and mixed-mode arithmetic. The problems are threefold: 1) ensuring that the C code performs the computations correctly; 2) dealing with the nonstandardization of the FORTRAN conventions; 3) making the C translations look pretty despite the fact that C promotion conventions are not the same as those of FORTRAN.

The key issue in the discussion is that both C and FORTRAN perform mixed-mode arithmetic; however, the promotion conventions they use are different. If the final look of the C translation were unimportant, as it is for a compiler, then all C casting operations needed to get the correct result would be shown in the C output. But showing all the casts produces a very ugly translation.

To deal with this issue, the notion of a *casting level switch* is introduced which gives the user the ability to fine-tune the processing of mixed mode expressions to meet his own needs and desires.

Introduction

In Goodman [2] the design of PROMULA.FORTRAN (a FORTRAN-to-C translator) is discussed. In summary, the translator is a compiler whose output code is C rather than some machine language. A problem which must be faced with this design is that the output of the compiler must not only produce the correct results, it must also be readable and maintainable by the user if he so desires. In this paper, the objectives of correctness and readability are contrasted. To ensure correctness, the readability of the output must be sacrificed. To avoid this, the user can select translation options, but only at the cost of potential error.

The particular area to be focused on is fixed-point arithmetic, either by itself or in mixed-mode form (i.e., arithmetic in which operands of different

numeric types are combined via binary operators or via assignment operators). The rules for such combinations in C are fortunately very clear and easily controlled via casting operators and parentheses. Unfortunately, in FORTRAN the rules are less clear and in fact vary from implementation to implementation. All FORTRAN conventions can be implemented via C; however, readability suffers. The particular aspects of FORTRAN to be presented are short integer arithmetic and the typing of statement functions and function parameters.

The approach taken in PROMULA.FORTRAN is to keep track of the types of all operands involved in expressions and to determine when any conversions need to be performed. Normally, it only shows these conversions in the C output if they would not be performed automatically by the C compiler. In abnormal cases, however, the types of conversions to be shown are expanded. Even additional conversions can be requested. This process is controlled via *casting-level switches* which the user can select at runtime and which are also internally set in certain contexts by the translator.

The Usual Arithmetic Conversions

As described in Kernighan and Ritchie [3], in C when an arithmetic operator combines operands of the same type, then the result is of that type. When the operands are of different types, the lower type is promoted to the higher type prior to the evaluation of the operator. Using this terminology, the operand types from highest to lowest are `long double`, `double`, `float`, `long int`, `int`, `short int`, and `char`. The only complication has to do with `float` arithmetic in most C compilers now in use: when two `float`s are combined they are both converted to `double` and the computation is performed in `double` precision.

The usual arithmetic conversions in FORTRAN are largely the same as those of C except that in FORTRAN it is fixed-point arithmetic that is typically promoted rather than floating-point arithmetic. Thus, in FORTRAN when two floats (`REAL*4`) are combined, the operation is performed in single precision (`float`). When two short integers (`INTEGER*2`) are combined, however, they are both converted to long (`INTEGER*4`) and the computation is performed using long integer arithmetic. To keep life interesting, however, not all FORTRAN implementations follow this convention.

In summary, C is as precise or more precise than FORTRAN in doing floating-point arithmetic and can be less precise than FORTRAN in doing fixed point arithmetic when the "usual arithmetic conversions" are used.

Casting Level Switches

If the purpose of PROMULA.FORTRAN were simply to compile FORTRAN code via the intermediate language C, then all casting operators needed to obtain the correct result would be written explicitly into the C output expressions. Just as a machine-language compiler must generate all type conversions

explicitly, so would PROMULA.FORTRAN. Unfortunately, one of the goals of PROMULA.FORTRAN is also to produce readable code. C codes which show all casting operations explicitly are not particularly readable; therefore, the strategy used involves only showing explicit casts when the usual arithmetic conversions of C would not produce the correct result. To achieve this, various switches controlling the expression processing functions of the code generator have been implemented. These switches may also be externally controlled by the user via the command-line. To simplify the later discussion, the casting level switches are introduced here.

Explicit conversions in C can be forced in any expression with a casting operator which has the following general format:

(*type*) *expression*

This casting operator converts the expression to the indicated type. Casting operators can be ordered via parentheses to produce almost any desired computational effect. The C casting level option allows the user to control which casts are "forced" in the C generated and which are to be assumed at C compilation by using the usual arithmetic conversions. Both aspects of promotion are controlled via this switch. The switch itself may occur more than once on the command line. Its individual settings are as follows:

C0 Specifies that all promotions between operands of different types be forced in the C output.

C1 Specifies that all promotions between different operands involving any integer types be forced, but that conversions between float and double not be forced.

C2 Specifies that only those conversions between fixed-point and floating-point be forced, but that other conversions not be forced.

C3 Is the default and specifies that only those casts needed to maintain the integrity of a calculation be maintained.

Cs Specifies that short integer calculations are to be done using short arithmetic.

Cl Specifies that short integer calculations are to be done using long arithmetic.

As an example, consider the following simple FORTRAN code that computes the squares of a weighted mean:

```
      SUBROUTINE DEMO(VAL,NVAL,WEIGHT,SQ)
      DIMENSION VAL(*)
      INTEGER*2 NVAL,WEIGHT,POW
      XBAR = 0
      POW = 2
      DO 10 I = 1,NVAL
      XBAR = XBAR + VAL(I)
   10 CONTINUE
      XBAR = XBAR/(NVAL*WEIGHT)
      SQ = XBAR ** POW
      RETURN
      END
```

The default translation of this example looks as follows in a dialect of C in which `int` is a short integer:

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= ((long)nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

Notice that there are only two casts shown. The `long` cast on the product between `nval` and `weight` is needed in case this product exceeds the maximum value of a short integer. The other cast is on the variable `Pow` in the exponentiation. Though exponentiation is a binary operator in FORTRAN, it is not in C; therefore, this cast to `double` is necessary in order to make the parameter for the `pow` function have the proper type. `xbar` does not require a forced cast because C automatically promotes a `float` to `double` when it passes it by value.

The same translation using the **Cs** switch yields the following result:

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
```

```
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= (nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

It is identical to the above except that the cast on the short integer calculation is not shown. *Be careful, this translation might produce an incorrect result.*

Let us now go to extremes. The following is the same translation using the **C0** switch which requests that all casting operators be explicitly shown:

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L; i<=(long)nval; i++) xbar =
        (float)((double)xbar+(double)*(val+(short)i-1));
    bar = (float)((double)xbar/(double)((long)nval*(long)
        weight));
    sq = (float)pow((double)xbar,(double)Pow);
}
```

In all likelihood one would never want to run PROMULA.FORTRAN in this mode unless one were interested in seeing all of the promoting that actually goes on. Notice, for example, that all floating-point calculations are promoted to `double` and then reduced back to `float`. Note also that constants also get promoted; thus, in the `for` statement the value of `1` assigned to `i` is now shown as `1L`. The other effect of forcing the casts is that this process blocks the `DO` loop reduction algorithm. Thus, in the initial translation, the loop was reduced to start at zero; but now it starts at one. This approach was taken, because forcing the casting level probably means that one is very concerned about the arithmetic being performed. Since loop reduction performs additional integer arithmetic it is turned off.

Finally, note that the `long` cast forced on `nval` by the default **Cl** setting in fact causes a `long` cast on `weight`. This is how the convention forces short

arithmetic to be done as long arithmetic. It might be instructive to see this example again, not with **C0** but with **Cs**, which will not force these `long` promotions.

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L; i<=(long)nval; i++) xbar =
        (float)((double)xbar+(double)*(val+(short)i-1));
    xbar = (float)((double)xbar/(double)(nval*weight));
    *sq = (float)pow((double)xbar,(double)Pow);
}
```

As one might fear, the cast to `double` is too late. The division is done at the `INTEGER*2` level and the possibly overflowed result is promoted to `double`. Thus, using **C0** to fix the short arithmetic problem only shows what the problem is.

The following example shows the same translation using the **C1** casting level:

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=1L;i<=(long)nval;i++) xbar+=*(val+(short)i-1);
    xbar /= (double)((long)nval*(long)weight);
    *sq = pow(xbar,(double)Pow);
}
```

As specified, the `float`-to-`double` casts are no longer forced, but all fixed-point casts are forced.

At the **C2** casting level, the following is the result:

```
#include "fortran.h"
void demo(val,nval,weight,sq)
int nval,weight;
float *val,*sq;
{
static int Pow;
static long i;
static float xbar;
    xbar = 0.0;
    Pow = 2;
    for(i=0; i<nval; i++) xbar += *(val+i);
    xbar /= (double)((long)nval*weight);
    *sq = pow(xbar,(double)Pow);
}
```

Now only the mixed-mode cast is forced. Integer arithmetic is not forced so the loop reduction is also allowed.

FORTRAN Short Integer Arithmetic

When we began testing the initial versions of PROMULA.FORTRAN, we were very concerned about the floating-point arithmetic. One of our test suites was, therefore, taken from *Numerical Recipes* [4] which contains computationally intensive FORTRAN code. Our greatest fears were realized. In some instances, the C translations produced dramatically different results than their FORTRAN originals even when run on the same platform. For example the following subroutine, which computes Kendall's Tau:

```
SUBROUTINE KENDL1(DATA1,DATA2,N,TAU,Z,PROB)
INTEGER*2 N
DIMENSION DATA1(N),DATA2(N)
N1=0
N2=0
IS=0
DO 12 J=1,N-1
    DO 11 K=J+1,N
        A1=DATA1(J)-DATA1(K)
        A2=DATA2(J)-DATA2(K)
        AA=A1*A2
        IF(AA.NE.0.)THEN
            N1=N1+1
            N2=N2+1
            IF(AA.GT.0.)THEN
                IS=IS+1
            ELSE
```

```
                        IS=IS-1
                    ENDIF
                ELSE
                    IF(A1.NE.0.)N1=N1+1
                    IF(A2.NE.0.)N2=N2+1
                ENDIF
   11           CONTINUE
   12       CONTINUE
      TAU=FLOAT(IS)/SQRT(FLOAT(N1)*FLOAT(N2))
      VAR=(4.*N+10.)/(9.*N*(N-1))
      Z=TAU/SQRT(VAR)
      PROB=ERFCC(ABS(Z)/1.4142136)
      RETURN
      END
```

produced wildly different results when translated into the following C source:

```
void kendl1(data1,data2,n,tau,z,prob)
int *n;
float *data1,*data2,*tau,*z,*prob;
{
extern float erfcc();
static int n1,n2,is,j,k,D1,D2;
static float a1,a2,aa,var;
    n1 = n2 = is = 0;
    for(j=1,D1=1,D2=(*n-1-j+D1)/D1; D2>0; D2--,j+=D1) {
        for(k=j; k<*n; k++) {
            a1 = *(data1+j-1)-*(data1+k);
            a2 = *(data2+j-1)-*(data2+k);
            aa = a1*a2;
            if(aa != 0.0) {
                n1 += 1;
                n2 += 1;
                if(aa > 0.0) is += 1;
                else  is -= 1;
            } else  {
                if(a1 != 0.0) n1 += 1;
                if(a2 != 0.0) n2 += 1;
            }
        }
    }
    *tau = (float)is/sqrt((float)n1*(float)n2);
    var = (4.0**n+10.0)/(9.0**n*(*n-1));
    *z = *tau/sqrt(var);
    *prob = erfcc(fabs(*z)/1.4142136);
}
```

The problem was not in the floating-point arithmetic; rather, it was in the fixed-point arithmetic. During the computation of `var` the fixed-point multiplication of `n` times `n-1`, where `n` was 200, was overflowing in the C version but not in the FORTRAN version. Note that we added the explicit `INTEGER*2 N` statement in the FORTRAN version to remove the possible differences between C `int` and FORTRAN `INTEGER`.

The final blow came when we found that FORTRAN implementations varied in this respect. Most produced the correct result, some matched our C version, and some gave even a third set.

Once the source of the problem was discovered, the following FORTRAN program which does short integer additions, multiplications, and divisions in a variety of contexts was constructed:

```
      PROGRAM VARI2
      INTEGER*2 I1,I2,I3,I4
      INTEGER*2 ISUM,IPROD,IQUOT
      INTEGER*4 JSUM,JPROD,JQUOT
      REAL*4 RSUM,RPROD,RQUOT
      REAL*8 DSUM,DPROD,DQUOT
      I1 = 20000
      I2 = 30000
      I3 = 200
      I4 = 300
      ISUM = I1 + I2           ** addition overflow
      IPROD = I3 * I4          ** multiplication overflow
      IQUOT = (I1 + I2) / I4   ** intermediate overflow
      JSUM = I1 + I2
      JPROD = I3 * I4
      JQUOT = (I1 + I2) / I4
      RSUM = I1 + I2
      RPROD = I3 * I4
      RQUOT = (I1 + I2) / I4
      DSUM = I1 + I2
      DPROD = I3 * I4
      DQUOT = (I1 + I2) / I4
      WRITE(*,'(23H Short Integer Results ,3I13)')
     +   ISUM,IPROD,IQUOT
      WRITE(*,'(23H Long Integer Results  ,3I13)')
     +   JSUM,JPROD,JQUOT
      WRITE(*,'(23H Short Real Results    ,3F13.5)')
     +   RSUM,RPROD,RQUOT
      WRITE(*,'(23H Long Real Results     ,3F13.5)')
     +   DSUM,DPROD,DQUOT
      STOP
      END
```

In running this example with various FORTRAN compilers, always on machines with 16-bit short integers (VAX, IBM mainframe, IBM PC), we obtained the following three results:

1. Universal promotion to `long`

Short Integer Results	-15536	-5536	166
Long Integer Results	50000	60000	166
Short Real Results	50000.00000	60000.00000	166.00000
Long Real Results	50000.00000	60000.00000	166.00000

2. No automatic promotion to `long`

Short Integer Results	-15536	-5536	-51
Long Integer Results	-15536	-5536	-51
Short Real Results	-15536.00000	-5536.00000	-51.00000
Long Real Results	-15536.00000	-5536.00000	-51.00000

3. Selective promotion to `long`

Short Integer Results	-15536	-5536	166
Long Integer Results	50000	60000	166
Short Real Results	50000.00000	-5536.00000	166.00000
Long Real Results	50000.00000	-5536.00000	166.00000

In reviewing these results, we observed that a negative number means that a short integer overflow has occurred. The typical FORTRAN result is the first one. In this instance, the output of all integer calculations is a `long`. That result is then converted to the desired result type. Notice that even the intermediate addition in the division example is calculated as a `long`.

In the second case, the result of any short integer calculation is always also `short`, regardless of the surrounding context. This type of result is unusual for mainframe FORTRANs and is common for PC FORTRANs. Note that Microsoft FORTRAN allows the user to select which type of convention is to be followed as a side-effect of the `WORDSIZE` metacommand.

The third case is strange and difficult to deal with. The particular result above can be gotten from VS FORTRAN. Note that in an integer context, universal promotion to integer is followed. Also in all cases, the intermediate addition result is promoted to `long`. But for some reason the multiplication result is allowed to overflow while the addition result is not.

It must be pointed out that none of the FORTRAN manuals discuss how they perform short integer arithmetic. Someone moving a FORTRAN program, such as the one above, from one FORTRAN platform to another might well encounter problems with his results with no warning whatsoever.

In designing PROMULA.FORTRAN we allow the user to select whether he wants universal promotion to `long` or no automatic promotion to `long`, via the **Cs** and **Cl** switches described earlier. We have no provision for selective promotions. Note that selective and universal promotion differ only in overflow conditions, so users from such environments should use the universal promotion to long convention. The default convention is universal promotion. No automatic promotion is selected via the **Cs** command-line switch.

From a translation standpoint, the default condition is unfortunately not the cleanest one. In C, the results of all `short` binary operators are `int` (which can be only 16-bits on some machines). The only way to force C to produce a `long` result is to convert the arguments to `long` prior to the calculation. The translation of the assignments in the above FORTRAN program under the default universal promotion convention is as follows:

```
void main(argc,argv)
int argc;
char* argv[];
{
static int i1,i2,i3,i4,isum,iprod,iquot;
static double dsum,dprod,dquot;
static long jsum,jprod,jquot;
static float rsum,rprod,rquot;
    ftnini(argc,argv);
    i1 = 20000;
    i2 = 30000;
    i3 = 200;
    i4 = 300;
    isum = (long)i1+i2;
    iprod = (long)i3*i4;
    iquot = ((long)i1+i2)/i4;
    jsum = (long)i1+i2;
    jprod = (long)i3*i4;
    jquot = ((long)i1+i2)/i4;
    rsum = (long)i1+i2;
    rprod = (long)i3*i4;
    rquot = ((long)i1+i2)/i4;
    dsum = (long)i1+i2;
    dprod = (long)i3*i4;
    dquot = ((long)i1+i2)/i4;
    printf(" Short Integer Results %13d%13d%13d\n",
        isum,iprod,iquot);
    printf(" Long Integer Results  %13ld%13ld%13ld\n",
        jsum,jprod,jquot);
    printf(" Short Real Results    %13.5f%13.5f%13.5f\n",
        rsum,rprod,rquot);
```

```
    printf(" Long Real Results     %13.5f%13.5f%13.5f\n",
        dsum,dprod,dquot);
    exit(0);
}
```

Note that in each case the lefthand argument is converted to `long`, thus causing the entire expression to be evaluated in that manner. The result of running this translation is shown below:

Short Integer Results	-15536	-5536	166
Long Integer Results	50000	60000	166
Short Real Results	50000.00000	60000.00000	166.00000
Long Real Results	50000.00000	60000.00000	166.00000

Note that it agrees with the universal promotion to `long` result shown above.

The simpler alternative is shown below. No conversions to `long` are made. Thus, overflows occur in every expression.

```
void main(argc,argv)
int argc;
char* argv[];
{
static int i1,i2,i3,i4,isum,iprod,iquot;
static double dsum,dprod,dquot;
static long jsum,jprod,jquot;
static float rsum,rprod,rquot;
    ftnini(argc,argv);
    i1 = 20000;
    i2 = 30000;
    i3 = 200;
    i4 = 300;
    isum = i1+i2;
    iprod = i3*i4;
    iquot = (i1+i2)/i4;
    jsum = i1+i2;
    jprod = i3*i4;
    jquot = (i1+i2)/i4;
    rsum = i1+i2;
    rprod = i3*i4;
    rquot = (i1+i2)/i4;
    dsum = i1+i2;
    dprod = i3*i4;
    dquot = (i1+i2)/i4;
    printf(" Short Integer Results %13d%13d%13d\n",
        isum,iprod,iquot);
    printf(" Long Integer Results  %13ld%13ld%13ld\n",
```

```
            jsum,jprod,jquot);
        printf(" Short Real Results    %13.5f%13.5f%13.5f\n",
            rsum,rprod,rquot);
        printf(" Long Real Results     %13.5f%13.5f%13.5f\n",
            dsum,dprod,dquot);
        exit(0);
}
```

The result of running this translation is shown below:

Short Integer Results	-15536	-5536	-51
Long Integer Results	-15536	-5536	-51
Short Real Results	-15536.00000	-5536.00000	-51.00000
Long Real Results	-15536.00000	-5536.00000	-51.00000

This result agrees with the no automatic promotion to `long` output from above, and is probably wrong for most applications. Alternatively, this translation is the cleanest and would probably be preferred by someone who intends to maintain the C code.

FORTRAN Statement Functions and Function Parameters

The previous section showed why the short-long casting capabililty is needed in PROMULA.FORTRAN. This section shows why the mixed-mode casting switches are needed. In FORTRAN, a *statement function* is a computational procedure defined in the same routine in which it is used. It is defined by a single statement that has a syntax like the following:

sf `(` [*p* [`,` *p*] ...] `)` `=` *e*

where:

sf is the symbolic name of the statement
p is a dummy argument
e is an expression containing the dummy argument(s) *p*.

A statement function is syntactically identical to the C `#define` with two important differences. First, FORTRAN statement function identifiers have the same scope as local variables, while C `#define`s always have file scope. This is a minor difference which will not be discussed further here, though the C convention is clearly inferior. The important difference is that FORTRAN statement functions and their arguments are typed. C's `#define` is merely a syntactic device, while the FORTRAN statement function has type semantics as well. The only way to incorporate this information in the translation of

C statement functions to C `#define`s is to include the casting operations in the translation.

Consider the following FORTRAN fragment which contains two statement functions:

```
SUBROUTINE DEMO
SF1(A,B,C) = A ** B ** C
IF2(A,B,C) = A + B + C
ALPHA = SF1(3.0,2.0,3.0)
BETA = 1.0 - IF2(3.0,2.0,1.0) / 5
END
```

The `SF1` function is floating-point and the `IF1` function is integer.

The C fragment below shows a simplistic translation of the above:

```
void demo()
{
#define sf1(a,b,c) (pow((a),pow((b),(c))))
#define if2(a,b,c) ((a)+(b)+(c))
static float alpha,beta;
    alpha = sf1(3.0,2.0,3.0);
    beta = 1.0-if2(3.0,2.0,1.0)/5;
#undef sf1
#undef if2
}
```

In this version the statement functions are translated directly into `#define` directives. Since C was criticized above, notice first in the translation of the double exponentiation that in FORTRAN the exponentiation operator parses right-to-left rather than left-to-right (as do all other operators). The problem with this translation is that it gives the incorrect result for the `if2` function. When expanded, the calculation of `beta` looks as follows:

```
beta = 1.0-((3.0+(2.0)+(1.0))/5;
```

which produces a result of:

```
1.0 - 6.0 / 5 = 1.0 - 1.2 = -0.2.
```

But `if2` is an integer function. The correct expanded translation of the above is:

```
beta = 1.0-((long)(3.0+(2.0)+(1.0))/5;
```

which produces a result of:

```
1.0 - 6 / 5 = 1.0 - 1 = 0.0.
```

The only way to ensure that casting operators of this type end up in the expansions of the `#define`s is to force them into the translation of the statement function. The correct translation of the fragment above is as follows:

```
void demo()
{
#define sf1(a,b,c) ((float)pow((a),pow((b),(c))))
#define if2(a,b,c) ((long)((a)+(b)+(c)))
static float alpha,beta;
    alpha = sf1(3.0,2.0,3.0);
    beta = 1.0-if2(3.0,2.0,1.0)/5;
    exit(0);
#undef sf1
#undef if2
}
```

Again, as in the case of short integer arithmetic, the nicest looking translation and most readable translation does not produce the correct result.

Conclusion

This paper has shown that the goal of optimizing the readabilty of a translation can conflict with the goal of ensuring semantic correctness. There is no "best" translation of FORTRAN to C. Compromises must always be made. Also, as shown in the short-integer arithmetic example, to obtain a correct translation requires information about the source of the program. FORTRAN programs which run correctly on one platform will not necessarily run correctly on another.

The advantage of using a compilation via C approach to the processing of FORTRAN codes is that all of these issues can be addressed and dealt with explicitly. It is the user's responsibilty to decide what his objectives are and what the characteristics of the source platform are; it is the FORTRAN processor's responsibility to reflect these decisions in the results of its work.

References

[1] MacDonald, Tom, "C Language and Numerical Programming", *The Journal of C Language Translation*, Sample issue March 1989, p. 9.

[2] Goodman, Fred "Design of a FORTRAN to C Translator", *The Journal of C Language Translation*, Vol. 1, Nos. 3 and 4.

[3] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language* Second Edition, Prentice Hall, New Jersey, 1988, p. 42.

[4] Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes*, Cambridge University Press, New York, 1986, p. 493.

Fred Goodman is a mathematician/linguist and the author of PROMULA. FORTRAN and PROMULA, an applications development tool for modeling applications using databases. Fred is also the Vice President and Director of Technical Development at PROMULA Development Corporation. He is currently applying the translation methodology discussed here to BASIC, PASCAL, and COBOL as well as other special-purpose languages on multiple platforms. Fred can be reached at (614) 263-5454.

∞

5. C and Digital Signal Processing

Kevin Leary
Analog Devices
3 Technology Way
Norwood, MA 02062

Introduction

The road of application of C to Digital Signal Processing (DSP) chips has been a rocky one over the last three to four years. All the major vendors of DSPs have offered C compilers for their processors. These vendors include AT&T with the 16-bit fixed-point DSP16 and the floating point DSP32, Texas Instruments with the fixed-point TMS320XX family and the floating point TMS320C30, Motorola with 24-bit fixed point 56000 and floating point 96000, and Analog Devices fixed-point 2100 family and floating point 21000 family. Several Japanese companies offer DSPs as well, but because these have only a minor presence in the general purpose user programmable market, they are not discussed here. All these vendors have offered C compilers for their DSPs (with the exception of AT&T on their DSP16) and all have had a similar customer reaction—"Your compiler is not efficient enough." As a result C has suffered in the DSP user community. To understand why we will take a look at how C is used in DSP development and at how Analog Devices is working to improve the state of the art of C-based development for DSP.

Using C in DSP Development

Digital Signal Processing is the numeric processing of quantized time-sampled signals. The original signal can be a continuous analog signal. An example of a quantized time-sampled signal is the integer temperature reported hourly by the local radio station. A more useful example would be the periodic output samples from an analog-to-digital converter (like those stored on a Compact Disc).

Since the processing of the signal can be done numerically, the problem lends itself nicely to programmable processor and software solutions. DSP chips are simply programmable microprocessors that have fast data handling and compute capabilities in order to implement applications in real-time. A typical DSP chip can multiply two vectors of numbers in 80 nanoseconds per element (12.5 million elements per second).

Typically the algorithms for DSP are complex enough that they are prototyped in C (or with some DSP design tool like COMDISCO's SPW that generates C) on PCs and workstations. Their numeric performance is analyzed there in a simulation environment before they are moved to the DSP chip. The user in this workstation environment can tolerate almost any level of host compiler inefficiency—simulation of the algorithm will just take longer. (It already is not working in real-time.) When the user decides to move this C code into the DSP environment, compiler efficiency requirements get stiffer—the application must operate in real-time. Working against this scenario is the fact that DSP architectures often have such limited register sets and addressing modes that generating efficient code from C is difficult. Even the best optimizing compilers have limits in this circumstance. The result: often the C coded application will not run in real-time. This leads to the perception that C compilers for DSP chips are inadequate.

Generating More Efficient Code

To deal with this situation many users are starting with assembly language for the DSP chip directly, or they move to assembly language once they have prototyped in C. This solution works, but of course results in code that's more expensive, less reliable, less maintainable, less understandable, and less reusable. Users in software development environments that are similar to DSP (e.g., embedded systems control) have lived through this course already and now use C widely. It is easy to see that there is still the need for a high-level language solution for DSP.

More sophisticated solutions are necessary that can build on the foundation of C. Some of these solutions are highlighted below.

- Produce new generation DSP architectures with a sensitivity to C language support. Examples of this include easier access to local stack data and better stack control. This kind of support affects the hardware little, but can dramatically affect execution efficiency of C. Also, the creation of more general purpose register files (versus fixed function distributed registers) helps significantly.

- Apply target code global compaction in addition to traditional optimizations. Many C compilers already do traditional target independent optimizations like those outlined by Aho and Ulman in *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986. To supplement this, a pass over compiler generated code can be performed to do global compaction and loop pipelining. DSP chips can especially benefit from these techniques since DSP architectures typically have wide-field instruction words, distributed and parallel function blocks, powerful sequencing, and at least one level of explicit data pipelining. (Data fetched from memory is available on the next cycle.)

- Offer DSP function libraries. DSP function libraries (e.g., FFTs and digital filters) can be written by the chip vendor in assembly language so that the most compute intensive code runs as fast as possible. This will only work if the library contents match user requirements. Also, library functions must be made as general purpose and robust as possible, resulting in slower than optimal execution times.

- Extend C for DSP application. C was not developed as a numeric processing language and certainly not as a DSP language. Yet, given the ubiquitous nature of C and the fact that it performs well in other embedded applications, the user community has adopted C as a desired language for DSP (despite the issues put forth above). To take advantage of this popularity, Analog Devices is interpreting the work of the Numeric C Extensions Group (NCEG, ANSI X3J11.1) to produce what it calls DSP/C. DSP/C helps alleviate the code efficiency problem by better mapping the language to the application. As an example of this, DSP algorithms are typically dominated by references and operations on vectors and matrices of data. DSP/C allows for explicit vector referencing through the array selection operator. When entire arrays are referenced in this way the compiler has an easy job of generating the most efficient code possible for the operation without having to apply vectorizing techniques. DSP/C is a vector language (not a scalar language like C).

Analog Devices is committed to C and DSP/C by pursuing strategies in all four of the areas outlined above. The architecture of the ADSP-21000 family of floating-point DSPs was developed with C in mind. Also, Analog currently offers C function libraries for DSP.

DSP/C and ICASSP '90

Representatives of Analog Devices presented two papers at the April 1990 International Conference on Acoustics, Speech, and Signal Processing (ICASSP) in Albuquerque, New Mexico, on Global Compaction and DSP/C. At this forum Analog Devices encouraged (and continues to encourage) other DSP chip vendors to join NCEG and help make DSP/C an industry standard.

The discussions at ICASSP '90 were dominated by talk of development tools for programmable DSPs. Of particular interest was talk of C compilers for floating-point DSPs. It is clear that C will play a role in DSP application development. One observer noted that C in the DSP world is where C was in the general-purpose microprocessor world just 6 years ago.

Kevin Leary is Manager of Development Tools, DSP Division, at Analog Devices. He can be reached at (617) 461-3843.

∞

6. ANSI C Interpretations Report

Jim Brodie

Abstract

X3J11 has moved into the Interpretations phase of its duties. In this issue I will briefly review the Interpretation process and procedures. I will then discuss some of the interpretation requests that have been received as well as the X3J11 responses to them.

Process and Procedures

The interpretation activity is a valuable service that X3J11 supplies to the C community. It provides a forum for discussing and resolving issues and ambiguities (real and apparent) in the standard document. It is important to remember, however, that the time is past for making changes and refinements to the standard. It is appropriate for writers to ask "Does this mean ...?" but it is no longer appropriate for them to ask "Wouldn't it be better if...?"

It is time to carefully read the standard as it currently exists. Sometimes the meaning of the standard is not clear. (Reasonable people, given the surrounding context of the rest of the document, could read the words different ways.) In this case the prior intent of the committee comes more directly into play in preparing the Interpretation response. However, there are limits on what X3J11 can put forth as an Interpretation.

If the standard is clear on how a question should be answered, the committee is not free to change the standard, even if it does not like what the standard says. In fact, the meaning usually is clear from a reading of the document. Sometimes you have to know all of the appropriate places to look and how the pieces all fit together, but this is probably the fate of any highly interconnected technical document.

The Interpretations process has several safeguards in place to ensure that fair and reasonable interpretations are supplied by the committee. When a request for interpretation is received from the X3 Secretariat (all requests should be sent through the X3 Secretariat so that they are logged and tracked) it is distributed to the committee members and placed on the agenda for the next meeting. The person making the interpretation request is notified of the upcoming meeting and invited to participate in the discussion of the issue.

The committee cannot take a position on any Interpretation request unless there is at least a two thirds majority in favor of the committee's response.

This ensures that if the committee is almost evenly split on an issue then no position will be taken until further work is completed.

Any interpretation response that is not based solely on the current wording of the standard (because the standard is ambiguous and extrapolations or prior intent is used to determine the response) is sent out to the X3 membership (X3J11's parent body) for a 30-day approval ballot.

Interpretation responses are periodically published in Technical Information Bulletins. These responses, however, do not change the standard in any way. In fact the following notice *must* be included at the front of each Technical Bulletin.

> "This Bulletin, while reflecting the technical opinion of the Technical Committee which developed the standard, is intended solely as supplementary information to other users of the standard. That standard, ANS X3.159-1989, as approved through the publication and voting procedures of the American National Standard Institute, is not altered by this Bulletin. Any subsequent revision to the standard, ANS X3.159-1989, may or may not reflect the contents of this Technical Information Bulletin."

Although the Interpretations do not have the force of the standard, they will hopefully help guide the establishment of common practice in the confusing or "fuzzy" areas. The ultimate goal of the standard and the Interpretations activity is to supply enough common ground so that source code portability between C translators and tools is a practical reality.

Interpretations

With this background, let's look at some of the Interpretation requests that were handled at the March 1990 meeting in New York City.

Preprocessing Numbers

The committee has received several papers offering alternative grammars for preprocessing numbers.

The Standard's preprocessing grammar is the following:

pp-number:
- *digit*
- `.` *digit*
- *pp-number digit*
- *pp-number nondigit*
- *pp-number* `e` *sign*
- *pp-number* `E` *sign*
- *pp-number* `.`

where:

> *digit* is a digit between 0 and 9,
> *nondigit* is an underscore or upper- or lower-case letter, and
> *sign* is either `+` or `-`

The writers point out that there are "surprising" items that are accepted as preprocessing numbers. For example, the current grammar accepts:

```
0xEE+23   0x7E+macro     and  0x100E+value-macro
```

as preprocessing numbers. (If white space had been provided, then these would be expressions rather than single tokens.)

Alternative grammars that address the particular set of anomalies are provided.

The committee's first response to a "request for interpretation" such as this is to note that it is not a request for clarification at all, but rather a proposal for a change. As noted above, X3J11 *cannot* change the C language standard at this time. The standard has been accepted and no changes can be made until a new work item is approved to start developing the next C language standard. (As it currently stands, it will be at least 5 years before that happens.) However, this is only half of the answer.

The second half of the answer is that the committee intentionally selected the current form of the preprocessing number grammar. It was not the result of an oversight or laziness on the part of the committee. (The committee has many members with extensive backgrounds working with grammars. Two members in particular, Tom Pennello of MetaWare and David Prosser of AT&T, can think in, manipulate, and work with, grammars more effectively than anyone I've ever worked with before.)

In a response to one of the people requesting a different preprocessor number grammar, David Prosser notes:

> "For more background, you'd need to know the state of the pANS [proposed American National Standard] before this grammar was voted in. The Committee had stated its intent that 'garbage' character sequences that began like a numeric constant were to be tokenized as a single sequence. This was to prevent situations in which this 'garbage' would be turned into valid C code through obscure macro replacements, among more minor reasons. This was, unfortunately, very poorly stated in the draft. As I recall, it was placed in the constraints for §3.1. It was something like 'Each pair of adjacent tokens that are both keywords, identifiers, and/or constants must be separated by white space.'
>
> As you can see this constraint neither presented the intent of the committee nor caused implementations to behave in any sort of consistent manner with respect to tokenization.

> Finally, a letter writer understood the issue well enough to suggest a grammar along the lines of the current §3.1.8. It, unlike your opening remarks on this topic, is *not* a 'loose description,' and finally stated in a precise way the intent of tokenization rules.
>
> The benefits of this construction were that all tokenization for all implementations would now be the same, no 'garbage' character sequences would be able to be converted to valid C code, skipped blocks of code could silently be scanned without generating needless and unnecessary tokenization errors, the preprocessing tokenization of numeric tokens would be greatly simplified, and room for future expansion of C's numeric tokens was reserved.
>
> That's a lot of good. The down side was that certain sequences now would require some white space to cause them to be tokenized as the programmer intended. As noted in the Rationale, there are other parts in C that require white space for tokenization to be controlled, and this was found to be one more."

Function-Like Macros and Empty Arguments

Another request for interpretation dealt with a case of empty arguments to function-like macros. The interpretation centered around the handling of situations like:

```
#define macro(x) ...

macro()
```

The question is, does the constraint within section §3.8.3 **Macro Replacement** of the standard apply that reads:

> "The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition..."

(making the invocation `macro()` an error that must be diagnosed)? Or does the statement within the semantics portion of §3.8.3 apply that reads:

> "If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined."

(leaving the response to the `macro()` invocation totally up to the translator)?

The underlying issue is whether it is possible to have a macro invocation with a single empty macro argument (it is very hard to distinguish this from the case of a macro invocation with no arguments).

The committee felt that the undefined behavior statement from the semantics portion of the section **Macro Replacement** clearly gives translators freedom to respond as they choose to macro invocations where empty arguments are supplied.

In this example, the macro definition clearly establishes that a single argument is expected in the macro invocation. Given these two pieces of information and the fact that a reasonable interpretation of the supplied macro invocation is that an empty argument has been supplied, the committee's position was that this macro invocation fell under the umbrella of undefined behavior.

Deciding that it was a constraint error would have effectively limited empty arguments to cases where there were two or more macro arguments. This seemed an unwarranted limitation given the wording in the standard.

Although it was not discussed by the committee, the question arises as to how this ruling applies whenever there is one less argument supplied in the macro invocation than were specified in the macro definition. For example, will the following result in a constraint error or undefined behavior?

```
#define   macro2( xx, yy ) xx + yy
          macro2( aaa )
```

In this case the macro invocation must generate a constraint error. This is because of a sentence earlier in the Semantics portion of the **Macro Replacement** Semantics section that reads:

> "The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments."

In the `macro2` invocation there are no commas separating arguments, so there can be only a single argument.

You would enter the realm of undefined behavior with either of the following macro invocations:

```
macro2 ( , aaa )
macro2 ( aaa , )
```

In these cases the first and second arguments, respectively, are empty sequences of preprocessing tokens.

Pragmas and Program Semantics

Another request for interpretation questioned what the limits were on the impacts of pragmas. For example, can a pragma change the semantics of a program?

As this issue was discussed, it became clear that the answer revolves around whether or not pragmas can appear in strictly conforming programs. In §3.8.6 **Pragma Directives** the standard states that a pragma

> "causes the implementation to behave in an implementation-defined manner."

It in no way restricts the areas that can be affected. This combined with the statement from §1.7 **Compliance** that reads:

> "A strictly conforming program shall use only those features of the language and library specified in this standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior ..."

made it clear to the committee that **a strictly conforming program can contain no pragmas.**

Given this situation, and no words to the contrary in the standard, it was the position of the committee that pragmas *could* change the semantics of a conforming program (remember a conforming program, as opposed to a strictly conforming program, only has to be accepted by a single conforming translator).

Declarations in Different Scopes

Perhaps the most interesting interpretation requests dealt with the issue of merging information from declarations in different scopes. The writer, Rich Peterson, asked the following question:

> "When more than one declaration is present in a program for an externally-linked identifier, exactly when do the declared types get formed into a composite type?
>
> "Certainly, if two declarations have file scope, then after the second, the effective type for semantic analysis is the composite type of the two declarations (§3.1.2.6 line 19). However, if one declaration is in an inner scope and one is in an outer scope, are their types formed into a composite type?
>
> "In particular, consider the code:

```
{
        extern int i[];
        {
                extern int i[10];
        }
        sizeof (i);
}
```

> "Is the use of `sizeof` in the above example a constraint violation because of the constraint in §3.3.3.4 **The sizeof Operator** that reads:
>
> > "The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type ..."
>
> or can a translator evaluate the operator using the remembered information from the now closed inner scope?"

The position of the committee (on a 25/2 vote) was that `i` has an incomplete type at the point where the `sizeof` expression is encountered and that a diagnostic is mandatory.

The issue of when composite types are formed was further clarified based on another example in this request for interpretation. The example given was:

```
{
        extern int i[10];
        {
                extern int i[];
                sizeof(i);
        }
}
```

The question raised was, again, whether the `sizeof` expression is valid. The underlying issue is whether a composite type is formed as a result of the second declaration of `i`. The alternative view is that the second declaration actually hides the information from the outer declaration. If the composite type is formed then the `sizeof` is valid. If it is not formed, then the `sizeof` will generate a constraint error because `i` will have an incomplete type.

After a careful reading of the standard (and much discussion) it became clear that the composite type is not formed and that a constraint error should be issued in this case.

There are two key sections in the standard that are brought to bear on this issue. The first is in §3.1.2.6 **Compatible Type and Composite Type**. After describing how composite types are built, it contains a single sentence that states when composite types are built. The sentence reads:

> "For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type."

The issue now becomes what does "same scope" mean. The standard has a definition in §3.1.2.1 **Scopes of Identifiers** that reads:

> "Two identifiers have the same scope if and only if their scopes terminate at the same point."

Since the outer and inner scopes do not terminate at the same point, they are different scopes and, by the first rule, no composite type is formed.

Peters asked yet another question relating to the flow of information in and out of scopes. He gives the example:

```
/* File scope */
static int i;                   /* declaration 1 */

main()
{
        extern int i;           /* declaration 2 */
        {
                extern int i;   /* declaration 3 */
        }
}
```

His interpretation of the rules given in §3.1.2.2 **Linkage of Identifiers** indicates that the second declaration specifies internal linkage for `i` while the third declaration specifies external linkage. He asked if this was what was intended?

The paragraph that addresses this case is:

> "If the declaration of an identifier for a object or a function contains the storage-class specifier `extern`, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage."

In the example above, the file scope declaration is not visible at the point of the third declaration. (The second declaration hides it.) In addition, the second declaration does not have file scope and therefore does not affect the linkage of the third declaration.

After reviewing this information, the committee agreed with the writer's interpretation. This means that this example program is in error, since it contains declarations for `i` with both internal and external linkage.

Whether this behavior was what the committee originally intended was a matter of some debate. Intent, however, is of little importance in this case because the standard is very clear in what it states.

Outstanding Issues

One final note. In *Volume 1, number 4*, I mentioned a couple of interpretation requests that I have not yet talked about.

The first one had to do with the question of whether functions return values by copying. This issue was debated at some length during the X3J11 meeting. However, there was no clear consensus or position taken. There is some on-going work in this area between meetings and I will pass on the status (and hopefully the committee's final position) after the next meeting, which is scheduled for September 24–25, 1990 in Pleasanton, California.

The second question was "Are multiple definitions of unused identifiers with external linkage permitted?" The answer is *No*. An editorial change that is

present in the final standard document clarifies this position. In §3.7 **External Definitions**, the applicable paragraph reads:

> "An external definition is an external declaration that is also a definition of a function or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a `sizeof` operator), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one."

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.

∞

7. Pragmania

Rex Jaeschke

Microsoft C V6.0 and QuickC

Microsoft's C compiler is available in two flavors: the full development system containing the optimizing compiler, and the QuickC interactive development environment. The pragmas available in either or both implementations are presented here.

Code and Data Placement

Executable code and certain objects can be placed in explicit text (code) and data segments, respectively, using the following pragmas.

```
#pragma alloc_text ( textsegment, function1 [, function2 ] ... )
```

The code generated for a given function can be forced into a named text segment. This directive must appear in the translation unit ahead of the functions' definitions but it need not appear before declarations for them or calls to them.

```
#pragma same_seg ( variable1 [, variable2 ] ... )
```

This directive declares that the named external variables are placed in the same data segment. (A compiler option is actually needed to force them in the same segment.) It must be placed after the objects' declarations and before they are used. The variables must be declared with the `extern` keyword. The compiler uses the list of variables to optimize common subexpressions involving data loads.

These directives can only be used at the file scope level.

Stack Overflow Checking

By default, the compiler generates code to call a stack probe function on entry to each user function. This probe function generates a stack overflow message if the stack space required for local variable allocation is unavailable. Calling this function increases both the code size and the execution time, so it may be useful to disable it. The stack probe can be enabled and disabled either by using a compiler option or one of the following pragmas.

```
#pragma check_stack ( on )
#pragma check_stack ( off )
#pragma check_stack ( )
```

The first two pragmas allow the programmer to enable and disable stack checking over a user-defined scope, whereas the compiler option approach covers the whole translation unit. The third example sets the stack checking status to that set by the compiler option.

A previous release of the compiler used the following alternate syntax.

```
#pragma check_stack+    /* equivalent to ( on ) */
#pragma check_stack-    /* equivalent to ( off ) */
```

Generating Inline Code

Certain library functions can be compiled inline using the `intrinsic` pragma which has the following form:

`#pragma intrinsic (` *function1* [, *function2*] ... `)`

This directive affects the specified functions from the point of the pragma until the end of the translation unit or the next `function` pragma that references the same functions. The `function` pragma allows you to escape the inline code generation by forcing a function call for the specified functions in all subsequent calls unless another `intrinsic` pragma follows further on.

`#pragma function (` *function1* [, *function2*] ... `)`

These directives can only be used at the file scope level. The set of intrinsic functions includes many from `math.h` and some copy and compare functions in `string.h` (such as `memset`, `memcpy`, and `strcmp`.)

Optimization

Loop optimization can be switched on and off on a per function basis using the following pragmas.

```
#pragma loop_opt ( on )
#pragma loop_opt ( off )
#pragma loop_opt ( )
```

It can also be controlled by a compiler option. And when used in conjunction with a relaxed aliasing option, can result in more optimization. Loop optimization can be switched on or off, or back to the mode set by compiler options.

Certain optimizations can be selected using compiler options. Some of these can also be switched on or off via the following pragmas.

```
#pragma optimize (  [ optstring ], on )
#pragma optimize (  [ optstring ], off )
```

optstring is a string literal containing a list of letters each of which corresponds to a particular optimization. For example, `"ace", on` *a*ssumes no aliases, enables local *c*ommon subexpressions, and *e*nables global register allocation. If the list is empty, the default settings are used. If the `on`/`off` switch is missing, the selected option is toggled.

Structure Member Packing

The alignment of structure members can be set using a compiler option. It can also be set (or overridden) using a pragma of the following form.

```
#pragma pack ( [ x ] )
```

where x is 1, 2, or 4 and indicates the byte multiple boundary to be used. When x is omitted, member packing reverts to that set by compiler options.

Bad Pointer Checking

You can enable or disable checks for null or out-of-range pointers at runtime using the following pragmas.

```
#pragma check_pointer ( on )
#pragma check_pointer ( off )
#pragma check_pointer ( )
```

To check selected pointers only, switch the checking on immediately before declaring those pointers and switch it off immediately after those declarations. Once checking is switched on it remains in force through the end of the translation unit unless it is explicitly switched off.

If the argument is omitted, checking is determined by the presence or absence of a compiler option.

Miscellaneous Pragmas

Comment records can be placed in an object file using the following pragma.

```
#pragma comment ( commenttype [, commentstring ] )
```

One special comment type allows the programmer to specify linker information in the object file, to direct the search of libraries for external references.

An Intel-specific keyword `_loadds` loads special data segment functions. This can be used in conjunction with the following pragma.

```
#pragma data_seg ( [ segmentname ] )
```

The `message` pragma writes its string literal argument to standard output. Compilation is not terminated.

```
#pragma message ( messagestring )
```

The title and subtitle of the source listing file can be specified with the following pragmas.

```
#pragma title ( titlestring )
#pragma subtitle ( subtitlestring )
```

A null string causes the title to be blank. The arguments may be either a string literal, a macro that expands to a string literal, or a combination of these, since adjacent string literals will be concatenated.

The dimensions of the source listing file can be set using the following pragmas.

```
#pragma pagesize ( [ linecount ] )
#pragma linesize ( [ columncount ] )
```

Form-feeds and newlines can be inserted in the source listing file using the following directives.

```
#pragma page ( [ ejectpagecount ] )
#pragma skip ( [ linecount ] )
```

If the count argument is omitted, only one form-feed or newline is inserted.

∞

8. A Language Compatible Arithmetic Standard

Martha Jaffe
Digital Equipment Corp.
MLO 1-3/B68
146 Main Street
Maynard, MA 01754

Abstract

This article explains the proposed Language Compatible Arithmetic Standard (LCAS), whose purpose is to facilitate, encourage, and support the development of portable numerical software written in high-level languages. The LCAS requires integer and floating-point arithmetic to be accurate, and to provide a notification when a serious arithmetic exception occurs. The LCAS also includes some useful floating-point manipulation functions and machine parameter constants.

Introduction

The Language Compatible Arithmetic Standard (LCAS) [1] was developed by Mary Payne and Craig Schaffert from Digital Equipment Corp., and Brian Wichmann from the National Physical Laboratory (UK), to be the first in a series of standards to facilitate porting numeric software across different architectures. The LCAS is a work item in ANSI X3T2 and in ISO/SC22/WG11. As part of the standards process, version 2.2 was recently published in SIGPLAN Notices [2] and the SIGNUM Newsletter [3] in order to solicit feedback from the Languages and Numerics communities. The present article hopes to be a gentle introduction to this document.

The ultimate goal of the LCAS is to support the production of high-quality, robust, portable numerical software. The LCAS identifies those aspects of floating-point and integer arithmetic that turn out to be important in writing portable programs. A conforming implementation must provide them to the programmer as machine parameter constants and floating-point manipulation functions. The LCAS also tries to ensure that the computing environment works accurately and predictably. The underlying arithmetic of a conforming implementation must provide a certain level of correctness and a certain predictability in the presence of exceptions. On the other hand, the intent is to be as general and inclusive as possible across the diverse arithmetic implementations currently in wide-spread use—so the LCAS is a little prescriptive and a little descriptive.

The authors of the LCAS have tried to make it compatible with the standard languages currently used for scientific software. After all, a user who is writing numerical software sees the computational environment through the filter of a high-level language. The machine's computational objects become data types, the arithmetic operations become operators or functions, comparisons are clothed in syntax, and explicit and implicit type conversions abound. (Actually, one of the goals in this series of standards is to separate out the effects of the high-level language.) Most languages already offer some of the LCAS constants and functions. Conforming implementations can provide the others through external libraries and data files.

Later standards in this series will consider other problems of writing numeric software in a high-level language. The next standard will specify accuracy and exception behavior for decimal/float conversions and for the elementary (and not so elementary) math functions. The following standard will address complex arithmetic and complex functions—of course, languages without the complex data type will not be required to add it. Other possible future topics: user control of precision of computation, control of optimizations, a uniform interface for exception handling.

Numeric Computing, Software, and Portability

Moving numerical software around from machine to machine is a fact of modern life. Physicists develop software on their workstations, test it out on the lab VAX server, and then let it fly on the Cray. Commercial software vendors sell portable libraries or application programs that are expected to run on all IEEE machines (all UNIX machines, all PCs, etc). In fact, a lot of effort goes into transporting code from one environment to another and then making sense of the transported results.

Much of the difficulty in porting code has nothing to do with numerical properties—the user might find different system calls, different libraries, unsuspecting use of local extensions to the high-level language, even subtle differences in declarations, storage formats, side-effects, and optimizations [4]. The compiler and operating system do influence the following numeric aspects:

- constructs in the language itself: data types, initialization of variables and constants, procedures, parameters, functions, etc.
- compiler strategies for evaluation of mixed precision expressions
- compiler strategy for register allocation, short-circuit evaluations, reversing conditionals, in-lining, and other optimizations
- decimal/float conversions, and I/O in general
- operating system intervention in cases of exceptions

Of course, the basic arithmetic and architecture of the machine have the greatest effect on accuracy and performance:

- the computational types (integers, floating-point) available, their sizes, and language data types correspondence
- number and size of registers
- memory management, caches, virtual versus direct mapping
- the basic arithmetic operations available, their accuracy, intermediate computations rounding to machine precision
- the overflow and underflow thresholds and behavior
- even the relative speeds of the arithmetic operations, and therefore, the actual operations selected for computation

An experienced number cruncher knows not to expect bit-for-bit identical results. Even on the same system, this can fail at different optimization levels. A reasonable expectation: a numerically stable algorithm for a reasonably well-conditioned problem will give reasonably comparable results on different systems. In fact, small perturbations in the data and computations giving rise to wildly different results is a clue that the problem is numerically unstable.

Some differences between implementations are very visible: the overflow threshold, the underflow threshold (usually less obvious), the response to exceptions. The separation between machine-representable numbers (precision) is more subtle—on a machine with higher precision, a Taylor's polynomial approximation might require more terms, or a float-to-decimal conversion might give a few extra significant digits. Even very subtle differences in machine rounding might show up in convergence parameters.

In fact, porting between IEEE 754 [5] implementations is a good example of how fragile the concept of portability is. Although the single and double precision floating-point formats are the same on all IEEE implementations, and the accuracy of basic arithmetic operations is specified, some IEEE implementations lack support for gradual underflow, NaNs, infinity, control of rounding direction, etc. Even when all the IEEE features are available, there is no portable way to access them from a high-level language. (The Numerical C Extensions Group [NCEG], a subcommittee of the ANSI C committee, is actively working on this issue.) Some IEEE implementations have extended precision registers and fast extended precision arithmetic. Actually, the most noticeable difference in porting code comes from compiler optimizations and register usage, not from the underlying arithmetic [6].

The LCAS wants to help the programmer separate out the numeric aspects from the rest of the system components (language, compiler, operating system) and allow him to concentrate on the algorithm.

Computational Quantities in the LCAS

The LCAS requires a conforming implementation to provide integer and floating-point types, at least one kind of each. A conforming implementation may also include other integer or floating-point formats that don't conform to LCAS.

Integers

No surprises here. LCAS allows both signed and unsigned integers. (The ANSI C language [7] requires both types.) The underlying machine integers that are visible in a conforming implementation depend on the high-level language's stipulations. (In C, the maximum and minimum available integers for each integral type are listed in `<limits.h>`.) Basic integer arithmetic operations—add, subtract, multiply—must be exact unless the result overflows. Integer division yields a quotient which must be rounded to an integer, either as the floor or as the chopped algebraic quotient. LCAS defines a *rem* function, which returns the remainder after division (C's `%` operator), and a *mod* function, which computes the positive remainder (C's `div` function).

LCAS requires conforming implementations to be able to signal that an integer arithmetic error (overflow, divide-by-zero) occurred. Languages sometimes suppress this information. Although LCAS doesn't specify a particular action on error, a conforming implementation must provide at least a mode of execution in which notifications do occur. (C's unsigned integer arithmetic is really integer arithmetic modulo a large N. In this system, overflow wraps around, is no longer exceptional, and does not produce a notification. So the unsigned integer data type in C is not the integer data type that the LCAS requires). In any case, an implementation's behavior in the presence of errors must be documented.

Floating-Point

The LCAS presents a floating-point model—the same model as in the ANSI C Standard—as a convenient way of describing precision and range. The model is sign-magnitude, with a p-digit fraction and an exponent that varies between $emin$ and $emax$. LCAS allows a general radix r, which in practice is 2 (e.g., IEEE 754, VAX), 8 (e.g., Unisys), 16 (e.g., IBM), or 10 (e.g., IEEE 854, HP).

The hardware need not really be sign-magnitude, of course; the fraction in a two's-complement format can be mapped to sign-magnitude with only minor problems. But the LCAS does not apply to some other computational models, like logarithmic, whose spacing and range properties are too difficult to fit into a floating-point precision/range description.

The LCAS model describes floating-point numbers as either normalized numbers (the first fraction digit is not zero) or IEEE-style denormalized numbers. Many hardware implementations have additional non-ordinary numbers with useful and predictable properties (e.g., IEEE NaNs and infinity, VAX

reserved operand, Prime unnormalized numbers), but of course these special non-numbers aren't portable across all architectures, so the LCAS can't use them.

Note that the LCAS/ANSI C floating-point model is a tiny bit different from the IEEE 754 model, leading to some confusion in notation. The LCAS/ANSI C model's fraction digits range in value from $1/r$ to $1/r^p$, while the IEEE fraction digits start at place value 1 and go down to $1/r^{(p-1)}$. A typical IEEE 754 floating-point number, written in the ANSI C model (radix 2), will have the same fraction digits but the unbiased exponents will differ by 1. For example, 3.5 looks like:

$$3.5 = (\tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8}) * 2^2 \text{ using binary LCAS}$$

and like:

$$3.5 = (1 + \tfrac{1}{2} + \tfrac{1}{4}) * 2^1 \text{ using the IEEE model.}$$

In both cases, the fractions are 111000...00, but the exponents are displaced by 1.

The IEEE maximum exponent represents NaN and infinity, and the minimum exponent is used for denormalized numbers. Given a normalized IEEE single precision number, x, its exponent can range from +127 to -126 in IEEE format. Writing the same x using the LCAS format, its exponent ranges from +128 to -125.

The LCAS specifies a few relationships between $emax$, $emin$, and the precision p that ensure that floating-point numbers can be large (10^6) and small (10^{-6}); that they can separate 6-digit decimals; that the product of $fmax$ and $fmin$ is reasonably sized; and that nearly every floating-point number (except zero and a few numbers very close to $fmax$ or $fmin$) can be inverted. Actually, some of these relationships may be too strong for a "shall" requirement—a few floating-point implementations have quite unsymmetric exponent ranges.

A wide range of floating-point implementations do satisfy the LCAS, and results of numeric computations on the different machines don't vary much. In any case, an implementation's floating-point format is one of its least changeable aspects. The LCAS does not expect to change existing implementations, but it does hope to encourage future implementations to limit their diversity, so that portability becomes easier instead of harder.

FP Operations—Accuracy and Rounding

A conforming implementation must provide basic floating-point operations (add, subtract, multiply, divide, comparisons) between operands of the same data type, and also conversions among the various floating-point and integer formats.

Correctness of basic arithmetic is surprisingly difficult to define. Take two machine-representable numbers. An arithmetic operation on the two numbers is

well-defined mathematically, but the result is usually not machine representable. So the result must be rounded: mapped to a machine-representable point (unless the result is too small or too large—more on this later). Although the computer arithmetic is approximate, the LCAS requires it to satisfy identities:

- commutativity: $a + b = b + a$ and $a * b = b * a$
- addition/negation: $a - b = a + (-b)$
- scaling: $a * r^n + b * r^n = (a + b) * r^n$ and $a * r^i * b * r^j = (a * b) * r^{(i+j)}$

The LCAS also requires several monotonicity conditions to hold:

if $a < b$ then $a + c <= b + c$, and if $c >= 0$ then $a * c <= b * c$.

Monotonicity is known to fail in only a few cases, on implementations in which the hardware optimizes the arithmetic operations for maximum speed. The jury is still out on the question of how important those few missing monotonicity cases are to numeric programmers.

The LCAS requires that:

- if the true result is exactly representable, then the computed result must be the true result.
- otherwise, if the true result doesn't underflow and lies between two neighboring machine-representable numbers, the computed result will be one of the neighbor points.
- otherwise, the true result underflows or overflows, and this will not pass unnoticed. (More on this later.)

Another difficult problem: validating that an implementation's basic arithmetic meets the LCAS's stringent accuracy requirements. Validation test suites for floating-point will be needed. Verifying that error reporting works as advertised is also an interesting problem, because once an error happens, the subsequent behavior of the test program may be unpredictable.

The LCAS allows a conforming implementation to offer several different ways of mapping the intermediate computed result to the machine-representable answer (rounding rules) but requires descriptions of the default and any other rounding rules as part of the implementation-provided documentation. For example, IEEE 754 offers four rounding directions (to-nearest, up, down, chop). Rounding on the VAX is the traditional round-to-nearest (halfway cases round away from zero). Other implementations just chop (IBM), and some systems provide a mixture of truncation and rounding away from zero. LCAS does not specify how any non-default rounding choices are communicated to the program.

Overflow happens when the computation's intermediate result would round to a number that is larger than the largest machine-representable number

($fmax$). The LCAS requires conforming implementations to provide a notification on overflow.

Underflow happens when the intermediate result would round to a number that is smaller than the smallest normalized machine-representable number ($fminN$). Underflow is potentially dangerous because non-zero suddenly becomes zero (e.g., in a denominator).

Underflow can be detected in a number of ways. IEEE 754 allows three possible definitions that involve combinations of exponent size and loss of precision—a loss of information when the fraction is rounded to fit the machine precision. On underflow, IEEE systems create denormalized numbers, which gradually underflow to zero. (Actually, in special cases, the IEEE underflow signal can occur with a normalized result, and denormalized numbers can be generated without the underflow signal.) Non-IEEE implementations often return zero, issue an underflow warning (or maybe not), and allow the computation to go forward. LCAS permits all of these actions but recommends that a notification be available on request.

The LCAS presents a formal description of the process of checking whether underflow or overflow is happening and then either rounding the intermediate computed result to a machine-representable number or raising the exception (and whatever happens after that), using a family of rounding $rndF$ and checking $chkF$ functions.

Notification: Who? What? When? How?

Notification is a central part of LCAS, and a very tough issue. The LCAS basic premise is that if something bad happens, it will not pass unnoticed. (And conversely, if nothing bad happens, the result will be correct.)

What bad thing happened? Either the computed result was out of range (overflow, underflow, divide-by-zero) or mathematically undefined (like the IEEE *invalid*, e.g., zero/zero).

Who gets notified? the LCAS does not specify exactly, but someone must find out. Something bad happens; the operating system is informed; it notifies the executing program (maybe the programmer anticipated the problem and included exception handlers) or perhaps notifies the user (error message)—and maybe the angry user notifies the programmer!

What does a notification look like? The LCAS wants notifications to be hard to ignore, impossible to overlook accidentally. Some recommendations: a change in control flow (e.g., to an exception handler—and maybe back), error messages, program termination.

Conformance to the LCAS is a matter of degree: how hard is it to avoid receiving a notification? An implementation can conform to LCAS if vigorous notifications are provided or can be enabled. (So an IEEE 754 implementation with trapping enabled will certainly conform.)

In the chaos of the world, the reactions of a typical system to an arithmetic

error can range from the strong error reporting required by the LCAS, through weaker alternatives (error codes, status bits, NaNs, infinities, VAX reserved operands), down to complete suppression of notification. The actions taken depend on many constraints, including the kind of error (e.g., underflow is usually invisible) and the stipulation of the high-level language. Actually, in both C and FORTRAN [8], post-error program behavior is undefined, although traditional FORTRAN practice is to terminate on the more serious errors. An ANSI C implementation can conform to LCAS by ensuring that user-visible notifications are raised after an error occurs in expression evaluation or data type conversion.

Even serious arithmetic error need not always provoke a catastrophic response, if the programmer has found a reasonable fixup strategy:

- in an interactive program, asking the user for more (or better) data and restarting the procedure
- scaling the data and recomputing
- setting a fraction with an infinite denominator to zero and continuing the computation

Of course, in any implementation, a willful user could refuse to heed all notifications—throw away error messages unread, disable all trapping, catch all signals and ignore them, ignore all the special non-ordinary number error returns, write exception handlers that just continue execution. This is certainly irresponsible programming and the results could be garbage, but no standard can prevent it.

LCAS Constants and Functions

The LCAS requires conforming implementations to supply a set of LCAS-defined constants and functions. Some of the constants could be determined by environmental inquiry programs (MACHAR [9], Paranoia [10]), but this process is not very reliable—it is extraordinarily difficult to ask the right questions and to interpret the results for some architectures. All the constants and functions must be documented. In addition, a conforming implementation will make constants available as run-time parameters in headers (or however the high-level language requires) and without using decimal/float conversions.

ANSI C provides many of the LCAS constants already. *maxint* and *minint* are in `<limits.h>`. The radix, precision, *emin* and *emax*, $fmax$ and $fminN$, *epsilon*, and some information about the rounding function are all in `<float.h>`.

epsilon, the distance between 1.0 and the successor of 1.0, is a good example of a constant that is hard to determine dynamically. One approach is to look for the smallest x such that $1.0 + x <> 1.0$, but the result depends very much on the rounding rule: on IBM (chop), $x = epsilon$; on VAX, $x = epsilon/2$;

on IEEE with round-to-nearest, x = the successor of $epsilon/2$, while on IEEE with round-up, even the smallest denormalized number works.

The LCAS recommends a number of floating-point manipulation functions. Some of these are familiar ANSI C functions from `<math.h>`; others are found in the Appendix to IEEE 754. A conforming implementation can provide the functions in external libraries, as intrinsics recognized by compilers, as macros in headers, or however the high-level language specifies. The functions must raise no undeserved exceptions. In the interest of maximum portability, the LCAS says nothing about NaN or infinity as input or output.

One of the most useful LCAS functions is $exponF$, which returns the IEEE-style exponent, like the IEEE `logb` function and `<math.h>`'s `frexp`. $exponF$ computes the order of magnitude of the number, relative to the radix: for a denormalized number, this is the IEEE minimum exponent minus the number of leading zeros in the fraction. The LCAS $signifF$ function returns the fraction, like `frexp`. Other `<math.h>` functions in LCAS: $scaleF$ (like `ldexp`), $intF$ and $fractF$ (like `modf`), and $sqrtF$.

LCAS contains some interesting new functions: $succF$ and $predF$ (like the IEEE $nextafter$); $roundF$ and $truncF$ (round or truncate the fraction at the nth digit and fill the trailing digits with zeros); $ulpF$ (the distance to the neighbor point in the direction away from zero). $ulpF(x)$ gives the place value of x's least significant digit. Without denormalized numbers, the $ulpF$ of very small numbers will underflow.

Why is the LCAS a Good Thing?

The integer and floating-point parameters and functions provide convenient and reliable ways to describe convergence criteria, to scale floating-point numbers, to take apart floating-point numbers into exponent and fraction and then put them back together.

An implementation that conforms to the LCAS provides a secure, predictable arithmetic environment:

- the integer and floating-point arithmetic is accurate for in-range computations
- the limits are documented
- the behavior in the presence of errors is documented
- serious errors (overflow, divide-by-zero, invalid) will not pass unnoticed.

Thus, the LCAS provides an arithmetic basis for a computing environment in which a programmer can feel safe moving arithmetic code from one conforming LCAS implementation to another.

References

[**1**] Mary Payne, Craig Schaffert, Brian Wichmann, *Proposal for a Language Compatible Arithmetic Standard*, version 2.2, September 26, 1989, ANSI Doc. No. X3T2/89-179, ISO Doc. No.: ISO/IEC JTC1/SC22/WG11 N144

[**2**] M. Payne, C. Schaffert, B. Wichmann, *Proposal for a LCAS*, in Sigplan Notices, vol. 25, no. 1, January 1990, pp. 59–86.

[**3**] M. Payne, C. Schaffert, B. Wichmann, *Proposal for a LCAS*, in Signum Newsletter, vol. 25, no. 1, January 1990, pp. 2–43

[**4**] Rex Jaeschke, *Portability and the C Language*, Hayden Books, 1989

[**5**] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985, 1985

[**6**] Tom Ochs, *Theory and Practice*, in *Computer Language*, vol. 6, no. 3, March 1989, pp. 67–81

[**7**] *American National Standard Programming Language C*, ANSI X3.159-1989, 1990

[**8**] *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, 1978

[**9**] W. J. Cody and W. Waite *Software Manual for the Elementary Functions*, Prentice-Hall, 1980

[**10**] R. Karpinski *Paranoia: A Floating-Point Benchmark*, in *BYTE*, vol. 10, no. 2, February 1985, pp. 223–224

Martha Jaffe trained as a mathematician at Brandeis. She worked on math libraries at Prime Computer, and recently joined the Computational Quality Group at Digital Equipment Corp. If you have questions or comments on LCAS, you may contact Martha at jaffe@rdvax.enet.dec.com.

∞

9. Miscellanea

compiled by **Rex Jaeschke**

Encrypting C Source for Distribution

Early in 1990, an interesting commercial product was announced called *The C Shroud, A source Protection Facility*. According to the product documentation, "The C Shroud is a software package that will translate a C program consisting of one or more modules into an equivalent C program that will be much more difficult to comprehend."

Since I've not seen such a product before, I though it was worth looking at to see just what such a tool can do and how it can be used. The obvious use is to enable vendors to distribute in source form rather than binary. They still have the distribution media format problem but at least they don't have to have one of every hardware and software platform their customers have. In fact, it allows them to sell to marketplaces they could not otherwise support. It also occurred to me that secure sites might also be able to use it to shroud their code when logging problems to compiler vendors and the like.

If you distribute source, that makes it easier to distribute fixes; you simply read the fix to customers over the phone or send them a hardcopy and they can apply it themselves. However, to make this work, the source must not only be encrypted but hidden in such a way as to still be readable, at least from a maintenance and vendor interaction viewpoint. If no one ever need actually read the encrypted source the problem is easy; you mangle it up as much as you can.

I have contrived the following simple source file to demonstrate the main aspects of the shroud program.

```
/* This is a C Shroud demo file */

#include <ctype.h>              /* protected */
#include <stdio.h>              /* protected */
#include "private.h"

main()
{
        char *pc1 = NULL;       /* protected */
        char *pc2 = "Hello";
        int f(int arg);
```

```
        int i, j[END + 1];

        i = tolower(i);         /* protected */

        for (i = START; i <= END; ++i)
                j[i] = f(i) * VALUE;
}

int f(int i)
{
        return (i + ' ');
}
```

By default, all headers are brought inline and all macros are expanded. Clearly, you do not want this to happen with the standard headers such as `stdio.h`, since an updated compiler may change the way `EOF` or `errno` is defined, for example. This is handled by flagging headers and/or identifiers as being protected. When a header is protected, all nested headers and identifiers automatically also become protected. (You can also designate a macro as being protected so that code that is conditionally compiled based on the macro's existence or value, can also be protected.) The only protected identifier by default is `main`.

With appropriate planning, these capabilities allow a product's source to be encrypted so that it is suitably protected yet allows the user to reconfigure it to fit their particular environment.

Here then is the shrouded output.

```
#include <ctype.h>              /* protected */
#include <stdio.h>              /* protected */
main()
{
char *a286484 = NULL;
char *a72796 = "\x048\x065\x06c\x06c\x06f";
int a117876(int);
int a106688, a77946[9 + 1];
a106688 = tolower(a106688);
a106688 = 0;
a258533:
if( a106688 <= 9) goto a130958;
goto a277383;
a89785: ++a106688;
goto a258533;
a130958:
a77946[a106688] = a117876(a106688) * 20; goto a89785;
```

```
a277383: ;
}
int a117876(int a298979)
{
return (a298979 + '\040');
}
```

Comments and indenting are removed unless they are part of a protected section of code. Newlines are retained. As mentioned above, the `#include` directives for the standard headers are retained as are references to their contents (`NULL` and `tolower`, in this case).

Identifiers are converted into long names starting with the letter 'a' followed by random (but unique) digit sequences. However, command-line options are available to specify shorter names, sequential number suffixes, and user-defined prefixes. You may even include the old name as part of the new name allowing you to add a prefix or suffix to identifiers in existing code without mangling it up. To confuse the reader even more, the members within each structure and union can be forced to reuse the same member name set. String literals and character constants can optionally be rewritten using their octal or hexadecimal counterparts, independent of each other.

The control structures `for`, `while`, `do/while`, `if/else`, and `switch` can be retained or converted to `if/goto` on an individual basis.

Special lint-like comments can be placed in the source to provide fine tuning over a user-specified set of lines. Such comments have the form:

`/*shroud` *option1 ... optionN* `*/`

Clearly, you might also wish to encrypt external identifiers. This is done by passing all source files to the shroud tool as a list so it can perform inter-module encryption. The width of lines in the output file can be specified, although tokens are never broken across output lines. Using a narrow width helps to make the source look even more obscure.

If you need to support customers, you will need to know just which identifiers and constructs they are talking about when they read their version of the source. To assist with this, a conversion symbol table can be generated when the source is shrouded. The table produced for the demo program is as follows:

new Name	scope	old name	file	line
a286484	local	pc1	test.c	9
a72796	local	pc2	test.c	10
a117876	global	f	test.c	11
a106688	local	i	test.c	13
a77946	local	j	test.c	13
a298979	local	i	test.c	21

Shrouding source code for general distribution is an interesting idea. Only time will tell now whether there really is a future for such a tool. In the meantime, Gimpel itself is one of its own customers. The C Shroud is used to distribute their commercial version of lint (called PC-Lint or FlexeLint, depending on the platform). For more information, contact:

Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426
(215) 584-4261
fax (215) 584-4266

Extensions

static_register

The `static_register` is a feature of the ACE EXPERT C compiler (developed by ACE Associated Computer Experts bv., Amsterdam, The Netherlands). It allows the programmer to store global variables in registers without the local save/restore mechanisms that apply to the `auto` storage class registers. This provides for such usage as required for base-registers, etc. The semantics of `static_register` are especially appreciated by those working in a realtime environment. It also has its use in case every possible optimization is required.

The syntax for a `static_register` variable is:

`static_register` *type-specifier identifier* `=` *constant*

where *type-specifier* denotes a type that can be held in a register and constant denotes the internal register number used for this variable. Internal registers should be allocated from high to low in steps of 1, and are, of course, target machine-specific. The initial value of a `static_register` variable is undefined.

The effect of the `static_register` declaration is that from that point on, the identifier is known as a register variable, and that register is no longer used in register allocation by the compiler. An example of the use of `static_register` follows:

```
/* Rather artificial example to demonstrate the
use of static_register.  Purpose: count total
occurrences of 'A' and 'B' combined, in "buf". */

#include <stdio.h>

#define BUF_SIZE 100
char buf[BUF_SIZE];

static_register char *pool = 13;
```

```
static_register long count =  7;

void count_A(void);
void count_B(void);

main()
{
        pool  = buf;
        count = 0;

        count_A();
        count_B();

        printf("Grand total is %d\n", count);
}

void count_A(void)
{
        register int i;

        for (i = 0; i < BUF_SIZE; i++)
                if (pool[i] == 'A') count++;
}

void count_B(void)
{
        register int i;

        for (i = 0; i < BUF_SIZE; i++)
                if (pool[i] == 'B') count++;
}

/* Each register decl.  of i in the above
functions will cause the usual save/restore code,
whereas no such code is generated for pool and
count.  Yet they are register variables. */
```

This example was run on an MC68020. For the MC68k family the first and the last available data registers are 7 (d7) and 2 (d2) while the address registers range from 13 to 10 (a5 to a2). (For the VAX the first register number which can be used is 11 (r11); the last is 6 (r6).)

The GNU C compiler also allows global variables to be placed in selected registers. The syntax it uses for this is:

`register` *type identifier* `asm ("`*register-name*`");`

Charize: Character Version of Stringize

ANSI C invented the stringize preprocessor operator `#`, to construct a string literal from a macro argument. Since that time various implementors have talked about providing a counterpart operator for constructing character constants. Microsoft has done just that in their latest DOS compiler, Version 6.0. The operator they invented is `#@`, which you will, no doubt, observe uses a character not in the required character set and for which there is no existing trigraph. In any event, here's a simple example of its use.

```
#include <stdio.h>

#define M(arg) printf("'" #arg \
   "' has an ASCII value of %d\n", #@arg)

main()
{
        M(0);
        M(A);
        M(?);
}
```

The relevant output from the preprocessor is:

```
printf("'" "0" "' has an ASCII value of %d\n", '0');
printf("'" "A" "' has an ASCII value of %d\n", 'A');
printf("'" "?" "' has an ASCII value of %d\n", '?');
```

and the program's output is:

```
'0' has an ASCII value of 48
'A' has an ASCII value of 65
'?' has an ASCII value of 63
```

Very Long Integers

Over the years, a non-trivial amount of code has been written that assumes a `long int` to be 32 bits. Unfortunately, vendors of 64-bit machines have to make a choice of whether to make `long int` map to a 32-bit integer thus preserving such code when ported, or to break that code by mapping `long int` to 64 bits. CONVEX C uses the former approach while Cray C uses the latter. As a result of their choice, the authors of CONVEX C had to provide a non-standard way to get at 64-bit integers. They did so by introducing new signed and unsigned integer types called `long long int` and `unsigned long long int`, respectively.

To accommodate these new types certain support was needed.

- The usual integral promotions in CONVEX C include an extra step of promoting to 64-bit integer types as necessary.
- `long long int` constants are specified explicitly using a suffix of `LL`, `ll`, `Ll`, etc. Constants without a suffix and which are too big to fit into `long int` are also treated as `long long int`, provided, of course, they can be represented as such.
- The edit masks for `printf` and `scanf` were extended to include the modifier `ll` as in `%lld`.
- `long long int` bit-fields are permitted.

Name-Space Reserved by GKS-3D

An ISO draft standard currently exists for *Computer Graphics – Graphic Kernel System for Three Dimensions* (GKS-3D) and it includes a C language binding. This standard reserves certain name-spaces for identifiers and, as such, may be of interest to developers of C language translation and support tools. The following information is extracted from the draft standard ISO/IEC DIS 8806-4:199x(e), January 1990.

The function names of GKS-3D are all mapped to C functions which begin with the letter `g`. Words and phrases used in the GKS-3D function names are often abbreviated in the representation and are always separated with the underscore character (`_`).

Standard C requires that compilers recognize internal identifiers which are distinct in at least 31 characters. It also requires that external identifiers be recognized to a minimum of 6 characters, independent of case.

Implementations that run in environments where two distinct C internal identifiers would be equivalent, if they were both external identifiers, *shall* include a set of `#define`s in `gks.h` which equate the long names to a set of short names.

In the construction of the several data types, function names, etc., the following policy is applied:

- All identifiers in the C binding are abbreviated using the same abbreviations for every component and using underscores to denote blanks;
  - Function names: `g` followed by abbreviated function name in lower case.
  - Data types: `G` followed by abbreviated data type in lower case.
  - Function-like macros: `Gfn_` followed by abbreviation of function name.
  - GKSM item types: `Gksm_` followed by abbreviation of item name.

- Error macros: `GE_` followed by some abbreviated expresion.
- Fields of enumeration types: `G` followed by a prefix followed by an abbreviation of the field name; this prefix is constant for each enumeration field; all the fields are in upper case.

AE: A Program Tracer

AE is a system for efficiently tracing events during the execution of C programs. It incorporates several new ideas that greatly reduce the overhead of collecting and storing this trace information. For example, AE can collect full address traces and only slow the measured program by a factor of 2–4 times. In addition, the resulting trace files are compacted 10–50 times (or 50–600 times after they are compressed) with respect to the full trace file.

AE consists of a small addition to the GNU C compiler (GCC) and an auxiliary compiler. The modified version of GCC produces code to record some events when the compiled program runs and generates a description of the program known as a schema. The other compiler, AEC, translates schemas into C programs that can read the condensed trace record and generate a full address trace.

AE has successfully compiled large programs on a DECStation 3100 and Sun 4. It should work properly for other computers that use the MIPS or SPARC processors. In addition, AE is parameterized in a manner similar to GCC and should easily be retargetable to any processor for which GCC produces code.

AE currently produces a full address trace of a program and identifies instructions that begin or end program loops and loop iterations. It could easily be extended to identify other interesting program events.

AE is documented by a University of Wisconsin Computer Sciences Department Technical Report #912 entitled *Abstract Execution: A Technique for Efficiently Tracing Programs*. The postscript version of this report is available for anonymous ftp from *primost.cs.wisc.edu* in the file ~ftp/pub/ae-tr.ps.Z. If you cannot print postscript, a printed version of this report is available at no charge from: Technical Report Librarian, Computer Sciences Department, 1210 West Dayton Street, University of Wisconsin, Madison, WI 53706, (608) 262-6616.

The patches to GCC and the code for AEC are available for anonymous ftp from *primost.cs.wisc.edu* (128.105.8.17) in the file ~ftp/pub/ae.tar.Z. The file is about 300K. The patches have been tested with GCC versions 1.36 and 1.37. They would probably work with earlier versions, but this is not guaranteed. If you cannot ftp this file, then either copy AE from another site or contact the author (whose address is shown below).

If you obtain a copy of AE and wish to be informed of future updates and bug fixes, send your electronic mail address to the author of AE. AE is copyrighted and distributed under the terms of the GNU General Public License. For further

information, contact the author of AE at: James Larus, Computer Sciences Department, 1210 West Dayton Street, University of Wisconsin, Madison, WI 53706, *larus@cs.wisc.edu*, (608) 262-9519.

Calendar of Events

- July 9–13, **ANSI C++ X3J16 Meeting** – Location: Microsoft to host somewhere in the Seattle, Washington area. For more information, contact the convener Dmitry Lenkov at Hewlett-Packard, 19447 Pruneridge Avenue, MS 47LE, Cupertino, CA 95014, (408) 447-5279, or electronically at *dmitry%hpda@hplabs.hp.com*.

- August 13–17, **International Conference on Parallel Processing** – Location: Pheasant Run resort in St. Charles, Illinois. Call David A. Padua on (217) 333-4223 or *padua@a.cs.uiuc.edu* for more information. (David is the contact for software-related papers.) The deadline for submitting papers is Jan 10.

- September 19–21, **International Workshop on Attribute Grammars and their Applications** – Location: Paris, France. For information, contact: INRIA, Service des Relations Exterieures, Bureau des Colloques, B.P. 105, F-78153 LE CHESNAY Cedex, France. Telephone: [33] (1) 39.63.56.00; Telex: 697 033 F; FAX: [33] (1) 39.63.56.38; E-mail: *waga@minos.inria.fr*.

- September 24–25, **ANSI C X3J11 Meeting** – Location: Pleasanton, California (about an hour east of San Francisco). Lawrence Livermore National Labs and SSI are hosts. This two 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*.

- September 26–27, **Numerical C Extensions Group (NCEG) Meeting** – The fourth 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 (see above entry) and will run for two full days. For more information about NCEG, contact the convener Rex Jaeschke at (703) 860-0091 or *uunet!aussie!rex*, or Tom MacDonald at (612) 681-5818 or *tam@cray.com*.

News, Products and Services

- **Microsoft** has released V6.0 of their DOS and OS/2 C compiler. It includes the first edition of their **Programmer's WorkBench**, an extensible framework for cooperating development tools. The tools shipped

include an editor, compiler, source browser, linker, and symbolic debugger. The interface specs have been published to allow third-party vendors to integrate their products.

- **Nu-Mega** is shipping **Bounds-Checker**, a tool that can detect out-of-bounds pointers in Microsoft C programs. Using the symbolic information produced by the compiler, the offending source line can be identified. (603) 888-2386.

- **Gimpel Software** has announced V4.0 of their PC version of lint, called **PC-lint**, for both DOS and OS/2. (215) 584-4261.

- **Computer Innovations, Inc.** has ported Gimpel's **FlexeLint** static analysis tool to the QNX operating system. (201) 542-5920.

- **Paracom, Inc.** is providing a family of parallel processing products built around the Inmos 32-bit RISC-based Transputer. Three different C compilers are available: **3L**, **Logical systems**, and **Norcroft**. (708) 293-9500.

- **AT&T** is making a research version of its **Concurrent C** multiprocessing environment available for $700. AT&T, 1776 On the Green, Morristown, NJ 07960, (800) 828-8649.

- **Peritus International** announced an ANSI C/C++ compiler for 386/486 systems running SVR3 UNIX and SunOS 4.0. (408) 725-0882.

- **LALR Research** is shipping V4.0 of their LALR parser generator. Included with the kit are grammars for 13 languages including C and C++. (714) 832-2274.

- **Sun** introduced a new product, Sun C 1.0; its first C compiler sold separately from SunOS. By unbundling the compiler, Sun can provide more frequent updates and enhancements independent of operating system releases. A version of the C compiler will continue to be bundled and supported with SunOS, but feature enhancements will be made to the unbundled version only.

- **Intel** announced iC-386 V4.2. This ANSI-conforming compiler generates code for the Intel386 architecture provides extensive support for developing embedded applications. The library also contains POSIX and SVID functions. (503) 696-4096.

∞