The Journal of

C Language Translation

Volume 2, Number 2

September, 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

10. Variable-Size Arrays in C

Dennis M. Ritchie
AT&T Bell Laboratories
Murray Hill, New Jersey, USA

Abstract

Both the original C language and ANSI C handle vectors of indefinite length, but neither caters for multidimensional arrays whose bounds are determined during execution. Such a facility is especially useful for numerical library routines, but is currently painful to program. Many of the issues, and some suggested language changes, were discussed by Tom MacDonald in *Volume 1, number 3* (December, 1989, pp. 215–233). Some compilers, for example the GCC compiler from the Free Software Foundation, already have extensions in this area. This paper discusses problems in these approaches, and offers a differing proposal for an extension.

Introduction

Previous thoughts (including my own) have fixed on the idea of allowing general expressions, and not merely constants, as bounds for automatic arrays. Although this extension, with careful restrictions, seems to fit into the structure of the existing language, and is useful by itself, it presents implementation difficulties in some run-time systems. Moreover, I argue that this proposal alone does not properly resolve the question of function parameters that involve varying arrays. The rules for both the GCC and MacDonald schemes are difficult to use and comprehend, and are difficult to formalize even to the level of the current ANSI standard; in particular, the type calculus for variable-sized arrays is murky for both.

In the existing ANSI C language, the type and value of an object `p` suffice to determine the evaluation of operations on it. In particular, if `p` is a pointer, the code generated for expressions like `p[i]` and `p[i][j]` depend only on its type, because any necessary array bounds are part of the type of `p`. In the MacDonald and GCC extensions, the values of non-constant array bounds are not tied firmly to its type.

My proposal, by contrast, is to avoid allowing variable-sized arrays as declared objects, but to provide pointers to such arrays; the pointers carry the array bounds inside themselves. This will preserve the property that pointer

arithmetic and dereferencing depend only on the type and value of the pointer. Allocation of varying-size arrays can be done by already existing facilities.

I intend that the proposed amendments to the Standard not change any currently conforming programs.

New Rules

To simplify the description, let us call the types 'array of T', 'array of array of T,' and so forth *matrix types*.

The first extension permits array declarators to be written

```
D [ ? ]
```

which imputes to the declarator `D` the type 'adjustable array.' Adjustable arrays are heavily restricted: no object, or structure or union member, may be a matrix type with any adjustable bound. However, pointers to matrices, some of whose bounds are adjustable, are permitted.

The second extension permits any integral expression to appear as a bound in an array declarator that occurs in a cast (but nowhere else). Such a cast may appear only in a function, and the expression is evaluated in the scope in which the cast appears. The type of the resulting array is the same as if the bound had been the constant expression of equal value. (The implied constraints here are not statically checkable.)

The final extension specifies that the `sizeof` operator, when applied to an expression involving a pointer to an adjustable matrix, need not yield a constant; there is a corresponding constraint that the value of such a `sizeof` expression is undefined unless the value of the pointer is defined. (This implies that the argument of `sizeof` must be evaluated when its type involves an adjustable array.)

Pointers to adjustable matrices (that is, matrices at least one bound of which is adjustable) are 'fat pointers' whose representation will include the space to store the adjustable bounds at run-time. It is crucial to arrange the rules so that these bounds can always be filled in properly. Therefore, the following restrictions are necessary.

- The type of an adjustable array of an element type is compatible only with another adjustable array, and only if the element types are compatible. In particular, an adjustable array is *not* compatible with an ordinary array or an incomplete array.

- Pointers to adjustable matrices may be assigned (or initialized or passed as arguments). If `p` is a pointer to an adjustable matrix, then `p = q` is permitted if the types are compatible, or if `q` fails to be compatible with `p` only because one matrix has a known array bound where the other has an adjustable bound. If the LHS `p` has a known bound at a

particular position and `q` has an adjustable bound, it is necessary (though not statically checkable) that the actual bound for `q` at that position be the equal to the known bound of `p`.

- Although the ANSI standard made `void *` pointers a generic type for pointers to objects, it exempted function pointers from this universality. Similarly, this proposal forbids automatic coercion of `void *` pointers to pointers to adjustable matrices; a cast specifying the bounds must be used.

These rules for casts and assignments are not simple generalizations of existing rules; they are formulated so that no pointer to an adjustable matrix can be constructed unless there is a way of finding the appropriate bounds at run time.

Examples

A function with a general two-dimensional array as a parameter might be declared:

```
void f1(int (*a)[?][?]);
```

and defined as:

```
void f1(int (*a)[?][?]) {
        int i, j;

        ... (*a)[i][j] ...
}
```

The type of the argument is 'pointer to adjustable array of adjustable array of `int`.' This formulation has the advantage that both array bounds are automatically available to the function; the first and second bounds are respectively:

```
sizeof(*a) / sizeof((*a)[0])
sizeof((*a)[0]) / sizeof((*a)[0][0])
```

This suggests that macros like the following be provided:

```
#define UPB1(x) (sizeof(*x) / sizeof((*x)[0]))
#define UPB2(x) (sizeof((*x)[0]) / sizeof((*x)[0][0]))
```

If the function needs a temporary array of the same size and shape as the argument, it can be allocated using:

```
int (*b)[?][?] = (int (*)[UPB1(a)][UPB2(a)])
                          malloc(sizeof(*a));
```

In order to call the function `f`, an ordinary array can be used:

```
int aa[100][100];

f1(&aa);
```

because the pointer to the constant-bound, complete array `aa` is converted to the argument's type.

The ugliest aspect of this formulation is that uses of the argument need to be explicitly dereferenced; we must write `(*a)[i][j]` inside of `f1`. One way of ameliorating the effect, within the function, is to generate a pointer to the first row of the array:

```
void f1(int (*a)[?][?]) {
        int (*ap)[?] = *a;
        ... ap[i][j] ...
}
```

This example was explicit in passing a pointer to the two-dimensional array. It can also be written in the more usual style, in which the pointer's explicitness is somewhat suppressed; this gives an alternate way of simplifying the references:

```
void f2(int a[][?]) { ... a[i][j] ... }
```

or equivalently,

```
void f2(int (*a)[?]) { ... a[i][j] ... }
```

Here the type of the argument is 'pointer to adjustable array of `int`,' and what is passed is a pointer to the first row of the two-dimensional array, instead of a pointer to the entire array. This function could be called using:

```
int aa[100][100];

f2(aa);
```

This formulation is in some ways more natural, but just as in ANSI C, the function loses the ability to compute the first bound of the array.

Discussion

Previous attempts to generalize C to adjustable arrays have taken the apparently straightforward approach of allowing array bounds (in carefully chosen circumstances) to be arbitrary expressions, instead of constants. I suggest that the straightforwardness is only apparent, and that the necessary rules are in fact quite complicated; neither the FSF nor MacDonald have really worked out their consequences. For example, the GCC language extension envisions function definitions like:

```
int f(int n, int m, int (*a)[n][m]) { ... }
```

in which the bounds are passed explicitly to the function (somewhat in the style of FORTRAN). Such proposals can possibly be made to work, but manifest unpleasant difficulties.

For example, what is the type of this function `f`, and how does one write a prototype for it? In GCC, one says:

```
void f(int, int, int (*)[][]);
```

while MacDonald makes the adjustability indicator more syntactic:

```
void f(int, int, int [*][*]);
```

The type `int (*)[][]` here, if one has only non-constant arrays, is strange; it is a notation that has to be introduced only for function declarations, and is of no use elsewhere. More seriously, one cannot determine, simply by looking at the declaration and at a call to the function, whether the call is type-correct, because there is no language-visible way to determine where the array bounds come from.

Other problems occur because of the way the function definition is written. In the example,

```
void f(int n, int m, int (*a)[n][m]) { ... }
```

the bounds come lexically first, before the array that depends on them. The arguments really have to be processed by the compiler left-to-right, in order that `n` and `m` be defined before the appearance of `a`. If the user wrote

```
void f(int (*a)[n][m], int n, int m) { ... }
```

then, in GCC, `n` or `m` would either be undefined, or would be captured by an outside declaration. (This is, in part, an unfortunate aspect of the ANSI function prototype syntax; the problem wouldn't occur with the old syntax.)

MacDonald suggests rules that cause all the arguments be processed simultaneously, or in two passes, but the rules would be messy, and dissimilar to the approach taken in the rest of the language. One example of the problems is arguments that depend on each other:

```
void g(int (*p)[sizeof(*q)], int (*q)[sizeof(*p)]) { ... }
```

Thus this approach seems fraught with difficulties. Either there are strange rules about the order in which parameters are mentioned, or there are strange rules about how parameter lists are parsed.

Conclusion

This paper proposes to extend C by allowing pointers to adjustable arrays and arranging that the pointers contain the array bounds necessary to do subscript calculations and compute sizes. In this way, the problem of handling variable-size array references is solved in a way that can be made consistent with the rest of the language. By contrast, the apparently most obvious language generalization (that is, allowing general expressions instead of constants in array bounds) complicates the type structure of the language, causes difficulties with the calculus of data types, and causes nonuniformity in discovering the sizes of arrays.

An earlier version of this paper was presented at the March 1990 meeting of NCEG and at the UKUUG Conference in London in July.

Dennis Ritchie is a Distinguished Member of the Bell Labs Computer Science Research Center. He is the principal designer of the C language and a major contributor to the UNIX operating system. Dennis actively participates in the Numerical C Extensions Group where he is the focal point of the variable dimensioned array subgroup. He can be reached at dmr@research.att.com.

∞

11. Standard C: What Barely Works

P.J. Plauger

Abstract

The ANSI standard for C perforce contains compromises. Many of these balance the conflicting needs of users and implementors. Some balance conflicts between different classes of users, some between different styles of implementation. Whatever the driving force, some of the compromises are better than others.

This article discusses a number of the less successful compromises in the C standard. It describes machinery that barely works, or that is easily misused. As much as possible, it explains how the compromise came about and why the language was not made better. It also suggests ways to minimize problems that can arise as a result of the compromise.

Users care about the weak spots in C because they need to avoid them. There is no point in stressing an already weak part of the language. Implementors care about the weak spots because they need to craft carefully around them. There is little point in trying to make them too sound, however. A program that depends upon a good implementation of a weak feature will only break when moved to another system.

Introduction

I know of nothing more complex, or harder to standardize, than a programming language. A general purpose language has a broad range of uses. Each use requires an extremely precise description of the language. Each use has a constituency with strongly held views about how the language should behave. It is hard to be at once ecumenical, precise, and accommodating for diverse applications.

One of the least understood aspects of making a programming language standard is the need for compromise. A user with a limited viewpoint sees only that the language is weakened for his or her particular application. How to fix it is patently clear. Should someone else object to the fix, it must be because that person has only a limited viewpoint.

The hardest to defend are the compromises that accommodate implementors at the expense of users. In a competitive marketplace, users are accustomed to making outrageous demands and having them met. A standard cannot be so brazen.

It's one thing to expect all PC compilers, say, to provide a certain service. Chasing a market measured in tens of millions of machines is a real incentive for implementors to work late into the night. To demand that a compiler for an eight-bit embedded microcontroller offer the same service, however, is quite another thing. Make too many silly demands and you encourage nonstandard subsets. Or you starve an already lean market.

Conflicts can occur between different implementors as well, or between different users. In all cases, the standard must compromise as best it can. If it cannot compromise creatively, it can only weaken the language. Where the language is weak, all users must beware. The chance of writing erroneous or nonportable code is high. Implementors, too, must step cautiously in these areas. It is too easy to step outside the bounds of conformance when the bounds are fuzzy.

For all its successes, the ANSI C standard contains a generous helping of unfortunate compromises. This article explores a number of the significant weak spots. It attempts to explain why they are weak, how they escaped better fixes, and why you should tread lightly around them.

Pointers and Addresses

Computers vary widely in how they address storage. It is remarkable that C permits the aggressive manipulation of addresses, yet remains efficient and portable. Nevertheless, here is an area that is fraught with dangers.

As recently as five years ago, I might have labelled pointer arithmetic in C an area of weak compromise. The C that grew up on the PDP-11 and VAX was facing serious portability problems:

- Early C assumed that pointers had the same representation as type `int`. More and more machines had pointers that were incommensurate with integers. You had to be much more careful about declaring function arguments and return values properly.

- Early C assumed further that pointers looked like unsigned integers that counted bytes in storage. More and more machines had pointers that were composites of two or more fields. You had to be much more careful about converting between pointer and integer representations.

Fortunately, many C programmers were willing to program more carefully. In the area of pointers, they got sophisticated in a hurry. Perhaps it was the surge in popularity of the IBM PC, with its bizarre 8086 memory addressing. Perhaps it was the need to port UNIX-based tools to multiple platforms to reach an adequate market. Whatever the reason, C programmers soon left behind their wild and wooly ways with pointers.

The situation has almost reversed itself. C started out on a machine with 16-bit pointers and 16-bit `int`s. That remains a valid implementation, but it has

become an uncommon one. Today, a 16-bit computer is more likely to be found in an embedded application. Serious computers for applications programming now need larger pointers, to address millions of bytes of memory. Try to move an application to such a small computer and it will likely break in a dozen places.

If the purpose of standards conformance is to maximize portability, then here is arguably a weak spot in the standard. What's the use of promising some form of portability if the promise isn't kept?

The question is, of course, rhetorical. Embedded systems and large applications represent diverse user communities. Both want to use Standard C. Both want to buy C translators with the cachet of standards conformance. Both deserve to have their interests represented in the C standard.

If you're looking for portability across implementations, however, it's not enough simply to ask for conformance to the C standard. You must also look for implementations from the same subculture as the original host. It would be nice if a standard could define these subcultures with some degree of precision. X3J11 tried in some areas, and mostly failed. In other areas, the committee didn't even try.

If you accept that pointers have inscrutable representations, your code will be largely portable. If you accept that a machine can simply be too small to host certain applications, you will be disappointed far less often. That leaves only one area where I feel that the standard still contains a weak compromise on pointers.

If you subtract two pointers, you get a signed integer result. The result counts the number of bytes between the places designated by the pointers. Naturally, both pointers had better point within the same data object. They should, in fact, designate elements from the same array. (Remember that any data object can be treated as an array of some character type.) The result can be positive, zero, or negative.

The result can also overflow, at least on some machines. Overflow is most likely to occur on 16-bit computers, since they are cramped to begin with. It can occur on a machine of any size, however. All the implementor has to do is choose the type of `ptrdiff_t` properly (or improperly).

If you can declare a data object sufficiently large, you can generate an overflow when subtracting two pointers. Say, for example, that `ptrdiff_t` has type `int` and `size_t` has type `unsigned int`. If the implementation lets you declare an object with more than `INT_MAX` bytes, you can get in trouble. The size of the object is well defined, but not the difference between any two pointers within it.

Nothing prevents an implementation from choosing a larger signed type for `ptrdiff_t`. Assuming one exists, that is. And assuming that old C code doesn't break because it assumes a certain type for the difference between two pointers. Sadly, at least one of these assumptions often fails.

In the days when 16-bit computers were more common, I habitually tested code for this class of failures. Contrive a data object that fills more than half

of memory and something invariably broke. It was a rare programmer who was always alert to this weak spot in C.

I predict that this problem will return to haunt us in the next few years. We will see more and more applications that strain the bounds of memory even on 32-bit machines. And programmers who are alert to the dangers of large objects will be rarer still.

Integer Sizes

If machines with small pointers are becoming rare, machines with small `int`s persist. The Intel 8086 and the Motorola 68000, to name two very popular chips, favor 16-bit integers. They can work with 32-bit integers, to be sure, but at a penalty. The penalty is always in time and frequently in code size.

Here is one of the most interesting compromises in the C standard. The language has recognized from the outset that each computer has a preferred computational type. With its 16-bit roots, C has always tolerated `int`s as small as 16-bits. X3J11 decreed, after only a brief debate, that `int` should never be smaller. It can certainly get much larger.

Today, great gobs of C code originate on machines with 32-bit `int`s. I would be astonished if many of those gobs survive porting to machines with 16-bit `int`s. At least not without some careful attention and a few judicious changes. You could argue that portability is not well served by tolerating machines with `int`s smaller than 16 bits.

I was one of the first people to push the limits of portability by writing commercial software in C. My goal was to write as much code as possible to run unchanged on half a dozen different architectures. Those architectures ranged from 8-bit Intel 8080s to 32-bit IBM System/370s. (I doubt that anybody has been quite that ambitious even to date.)

One of my earliest discoveries about C was most ironic. I found that the type `int` was the greatest impediment to portability. Wherever it appeared in code was a likely source of difficulty in moving code among machines. It was so bad, I learned never to declare a data object with type `int`. I also learned to contain places where the type crept into expressions willy nilly.

The irony, of course, is that `int` is a pervasive type. It is the type to which character and short integers gravitate when they appear in expressions. It pops up whenever you write a Boolean expression. It is the default type when you call a function that you do not declare.

The elasticity of type `int` is one of those fundamental aspects of C that is both a strength and a weakness. Many rules in C help an implementation generate good code by adapting to the peculiarities of each machine. But those same rules put an extra burden on programmers. Unless you learn to be wary of elastic types, you can easily sacrifice much of the portability that supposedly comes free with using C.

Integer Representations

Integers can also be elastic in another dimension. The C standard tries to be ecumenical about how an implementation encodes integers. It doesn't allow every form that has ever been wired into hardware, but it does allow a certain variety. My guess is that many C programs will fail if moved to some of these variations.

The one thing you can depend on is that integers have a weighted binary encoding. That means that a positive integer always has a least significant bit that contributes zero or one to the value. It has a next least significant bit that adds zero or two. And so on. The value zero always has all bits zero.

Not all binary encodings are weighted, by the way. Gray code has the peculiar property that adjacent values have codes that differ in one bit position. N bits can represent 2^N distinct values, as usual, but not in the usual way. C does *not* permit Gray code for representing integer types.

Where the variety comes is in representing negative values. I know of three encodings. They are called twos-complement, ones-complement, and signed magnitude. In all cases, a negative number has its most significant bit set.

- Twos-complement gives the most significant of N bits the value -2^{N-1}. An unsigned integer would give this bit the value $+2^{N-1}$. To negate a number, complement all the bits and add one. (Ignore a resultant carry off the end.) Only the value -2^{N-1} overflows when negated.

- Ones-complement is similar to twos-complement, with negative values differing slightly. To negate a number, simply complement all the bits. No values overflow when negated, but zero comes in two flavors (all one bits and all zero bits).

- Signed magnitude multiplies the value by -1 if the most significant bit is set. To negate a number, simply complement the sign bit. As with ones-complement, you avoid overflow on negation at the cost of having a distinct code for -0.

The vast majority of modern computers use twos-complement arithmetic, for integers at least. Floating point numbers typically use signed magnitude, but that is much less of an issue. Floating point operations are more intricate, and more stylized, than integer operations. Even -0 causes few floating point surprises, provided comparison operators work sensibly.

X3J11 decided to permit all three encodings for integer arithmetic. Twos-complement is expensive to simulate if a computer doesn't do it naturally. And there are some important processors that can host C well in all other respects. It would be unacceptable to hamstring them, or to rule them out, on the matter of arithmetic alone.

Nevertheless, you can expect numerous and subtle surprises moving code to a machine that does not use twos-complement arithmetic for integers. Some can

come, as with floating point, if an implementation is shoddy about its treatment of −0. Most, however, come from the bitwise operators.

Most programmers learn the power of bitwise operations sooner or later. They let you manipulate integers as individual bits or groups of bits. You use bitwise AND to mask or clear specific bits. You use bitwise OR to merge or set specific bits. You use bitwise XOR to complement specific bits.

Stick with positive values for all operands and you're in good shape. Remember that C requires a weighted binary representation for all integers. That means zero and positive values always have the same bit patterns. Your only uncertainty is the number of bits. Even there, you have rigorous lower bounds.

Once an operand gets its sign bit set, however, you are asking for trouble. Mix it with an unsigned operand and it may well get converted to unsigned itself before the bitwise operation is performed. Twos-complement values suffer no change of bit pattern. The other two forms get curdled in various ways. The low-order bits may not be what you expect.

Smart programmers have also learned the dangers of right-shifting signed operands containing negative values. Implementations have altogether too much latitude in deciding what to do, even with the low-order bits. But even smart programmers tend to be cavalier about bitwise operations. Expect trouble if you have to move code away from a twos-complement implementation.

Argument Lists

Another notoriously spongy area concerns how arguments get passed on a function call. Most of the time, you don't really care. It is only when you need to move a pointer from one argument to the next that storage layout matters. Even here, you should only care when you write code that processes a varying number of arguments. The C standard encourages a fairly disciplined approach, but even this approach has plenty of soft spots.

In the earliest days of C, this was a nonproblem. Everyone knew how the PDP-11 compiler passed arguments. It laid them out in a solid brick on the stack. Successive arguments occupied increasing addresses. Arguments had only a handful of distinct representations—a character type became an `int`, for instance. It was easy to learn the possibilities and to write code that walked any argument list.

As C moved to other environments, various problems arose. In some cases, the implementor chose to match an existing calling sequence. That let you mix C code and, say, FORTRAN, to advantage. In other cases, the machine itself was unsympathetic to the PDP-11 way of doing things.

Perhaps it was easier to put arguments on the stack in reverse order. Or perhaps it was necessary to leave various sized holes between arguments. Or arguments got scattered through registers. In some cases, arguments had to be accessed through a chain of pointers.

Several of us settled on the obvious solution about ten or twelve years

ago. We each introduced a set of macros for accessing arguments and stepping through argument lists. Naturally, the various solutions differed in detail. But they were similar enough in spirit to make it possible to port code among the different schemes.

X3J11 was able to pick and choose among the best aspects of the various approaches. Since the Berkeley `<varargs.h>` was best known, it served as the basic prior art. We introduced enough changes to warrant renaming the standard header `<stdarg.h>`. That is the machinery you should use for walking varying length argument lists in Standard C.

The approach is basically flawed, however. Few operations require a more intimate knowledge of implementation details than walking an argument list. Prior art let you do so with macros, and X3J11 was committed to preserving prior art wherever possible. But that prior art barely worked.

Consider. You have to prime the macros with the address of the rightmost declared parameter. Some implementations require this information to determine where the varying arguments begin. That rules out functions with no fixed arguments. A perfectly sensible possibility is ruled out by an implementation kludge.

You also have to know the type of an argument after it has suffered default conversions, in the absence of function prototype information. That's not impossible, but it is often tedious and occasionally tricky. Thanks to the value preserving rules, for example, an argument of type `unsigned short` might become either type `int` or type `unsigned int`. If you think that's of no consequence, go reread the previous section.

Finally, you are constrained as to the types of arguments you can pass. Put simply, you must be able to write an asterisk after the type specifier for an argument to make the appropriate pointer type specifier. That is not always possible.

In summary, you had better not be ambitious in your use of the `<stdarg.h>` macros. Keep it simple or expect portability problems.

Nonlocal Jumps

The last item I will discuss is probably the biggest kludge in Standard C. It is the machinery for performing nonlocal jumps. These are control transfers that do violence to the normal stack discipline of nested function calls and returns. They are implemented by the functions `setjmp` and `longjmp`, defined in the standard header `<setjmp.h>`.

A call to `setjmp` memorizes information about the current function call environment in a data object provided by the caller. A later call to `longjmp` restores the calling environment back to that memorized in the data object. The net effect is that control once again returns from `setjmp`. Any intervening function calls are forgotten.

It is very hard for a C translator to optimize much in the presence of calls to

`setjmp` and `longjmp`. The last thing an implementor wants to worry about is having the stack wrenched about in the middle of a block of code. Nevertheless, that is exactly what happens.

As a sop to implementors, the C standard allows a scary bit of uncertainty. It leaves open what happens to dynamic data objects declared in a scope containing a `setjmp` call. The environment might snapshot one of these data objects when `setjmp` is called. In that case, the value gets rolled back by `longjmp`. Or a data object might be left out of the environment. In that case, the value remains unchanged.

You'd think that you could devise a simple rule for determining what happens. "`register` data objects are part of the saved environment, `auto` data objects are not." Sounds good, but it doesn't work. The translator may choose not to honor a `register` declaration. Or it may choose to promote an `auto` data object to a register.

X3J11 realized that it could seriously limit such optimizations if it insisted that `setjmp` be more predictable. Or it could effectively require that all implementations recognize `setjmp` and `longjmp` as magic functions. (Any implementation can do so if it chooses. The committee wanted to avoid requiring such behavior.) So it opted for a kludge definition.

As a result, programmers had better be careful using these functions. You should confine any `setjmp` calls to very simple expressions. Place these expressions in the smallest possible functions. Use them only where you must.

Like `<stdarg.h>`, the machinery in `<setjmp.h>` more properly belongs in the C language proper. Since it isn't there, it barely works.

Conclusion

I could add more items to the list of things that barely work. The error reporting machinery built around `errno`, for example, is notoriously klunky. And signal handling has been so emasculated that it has almost no portable semantics. I consider these so clearly flawed, however, that even naive C programmers soon learn to be cautious.

I confined my remarks in this article to more subtle issues. These are areas that at least look like they might be properly thought out. That is part of their danger. Were they as well engineered as many people think, they would cause less trouble. Or were they more clearly flawed, they would invite less trouble.

None of the machinery described here is outright broken. If you use it right, it will work for you. Just don't push it.

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.

∞

12. Electronic Survey Number 5

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 60 different people, with 16 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.

The GNU C Compiler

Do you actively use the GNU C compiler? Have you packaged it for general distribution to your customers? If so, do you support it as your primary or alternate compiler? Any comments on gcc?

- 9 – No
- 7 – Use it
- 1 – Ship it (FSF)
- 1 – Support it (FSF)
- Comments:

    1. I don't personally use it routinely, but some others here do, and seem happy enough with it.
    2. We only use it to compare against our own ANSI C Compiler.
    3. I use it as a reference against my compiler.
    4. We value it for its ANSI C-ness rather than its code generation.
    5. We don't compile our products with it, nor do we distribute it. Some people use it in-house. To a limited degree we support its use with our product (which, among other things, loads object files produced by compilers and reads the debugging information to learn types and line numbers).

6. gcc contains a number of extensions to the C language. Some are nice, some not so nice. But anyone who writes code that uses those extensions is insane, unless that code is to be thrown away. *[Ed: Let's wait and see how many of these extensions become commonplace. I think some will.]*
7. The compiler has not been ported to our hardware yet. We do use it on other vendors' hardware that we have in-house, sometimes as a primary compiler and sometimes as an alternate, depending on the application and the machine.
8. In many areas, the compiler seems to be a reasonable compromise between wanting a portable and effective implementation, and standard conformance. I think that more restraint could have been used in its extensions, but it does serve a useful purpose as a trial balloon for ideas.

Lexical Analysis

Do you use a commercial or third-party tool to produce your lexical analyzer? Which? lex, Flex, LALR, other, home-grown? Do you have any comments on the problem of tokenization of C in general or ANSI C in particular?

- 0 – *lex*
- 1 – *Flex*
- 12 – Home grown
- Comments:

1. No! Real compiler writers code up their own lexical analyzers.
2. I use a custom one since they are always faster, and lexers are rather simple, so no need for a tool. I have problems with the tokenization of ANSI C: 1) I hate the tokenization required between false conditionals. It's much faster to simply scan past characters until you see a line beginning with `#`. 2) Trigraphs slow down the scanner, and only will appear inside ANSI C test suites.
3. I used *lex* at first, but switched to a hand-coded lexer when it turned out to be a bit faster. The `typedef` ambiguity is solved by the lexer, of course, but this is easy. Some other things are tricky (continuation lines, keeping track of original line and position therein for diagnostics, trigraphs, etc.) but the actual tokenization is simply enough.
4. The lexical scan is performed with routines especially written for the C compiler. The only problem we can remember is the inconvenience caused by trigraphs.

5. The requirement to return type names and ordinary identifier as two different tokens adds feedback from semantic analysis to the scanner, and therefore adds a lot of unfortunate complexity.
6. My lexical analyzer is in straight C, custom written for the compiler. The most agonizing problem of tokenizing C in the way that I do is the problems of integrating the macro pre-processor into the lexical analysis phase. For most C programs, most of the source of a file is not in macro expansions, so if I interpose a complete layer of software that does the preprocessing and passes on any text to the scanner my compiler pays a penalty. So, the compiler tokenizes as much as possible reading text directly from the input buffer. Only when a macro expansion is encountered do I perform extra work to perform the substitutions. ANSI C has created many new problems because of the funky expansion rules for nested macros and stringizing.
7. I was using *lex*, then *Flex*, then went to an ad-hoc lexical scanner because of the speed issue and partially because of the size of the lexical tables.

   I did have some problems with *Flex*, I think it has some nasty bugs left in it, but it was considerably faster than *lex*. When we were using *Flex* it was necessary to port *Flex* along with our own software in some cases. *Flex* appears to want `char` to be `signed char` and this is portability problem.
8. In general, automatically generated lexical analyzers are too slow. The compiler spends a large fraction of its time in lexical analysis and a hand-generated lexer really pays off.
9. Our tokenizer is hand written. C tokenization (except for the problem with `<...>` `#include` header names) is straightforward. The pp-number grammar (first introduced in ANSI C) certainly simplifies the tokenization of numeric constants and allows for extensions, but doesn't seem too popular with a vocal portion of C programmers.

Parser Generation

Do you use a commercial or third-party tool to generate your parser for the compiler and/or preprocessor? Which? yacc, bison, occs, other, home-grown? Do you have any comments on the problem of parser generation for C in general or ANSI C in particular?

- 7 – *yacc*
- 4 – Home grown
- 2 – *bison*
- 1 – *LLgen*

- Comments:

  1. No! Real compiler writers code up their own recursive decent parsers.
  2. I use a custom recursive-descent one. Again, parsing is not a big problem for most languages, the problem is in the semantic analysis which tools provide no help on.
  3. Damned `typedef`s. I went through a lot to remove all conflicts but one (`if-if-else` ambiguity) from the grammar. Error recovery is hard, sometimes (especially at the top level: function definition versus declaration). Those braced-but-not-always initializers are really annoying, as are `typedef` names as an argument for a prototyped function.
  4. We use *LLgen*, a recursive descent parser developed at our university.
  5. We have our own tool that generates *yacc* output.
  6. We use *yacc* to generate LALR parsing tables, but use our own parser to interpret these tables in order to provide better error recovery than the *yacc* parser allows. Error recovery for C is difficult because many sequences of two or three tokens might be legal in some context.
  7. I use a recursive descent parser written in C. I think that ANSI C did create additional parsing problems for functions, but most of the problems I encountered implementing those changes had to do with the fact that I had to change an existing scheme that would no longer work and I also had to add extensive MS-DOS specific extensions like `near` and `far` that don't mesh well with the ANSI syntax.
  8. I use *yacc*. Speed has not been a problem (yet). It appears to me that some minor adjustments to the *yacc* table lookups (especially on one-to-one productions) could make it much faster. But again, this hasn't been a problem yet.
  9. No, and we are sorry we don't. Although some people complain that *yacc* output is too slow, the compiler does not spend enough time parsing for that to be a problem. The recursive descent parser we have now is extremely convoluted and we will probably replace it with a parser generated by *yacc* or from an attribute grammar.
  10. C has two parsing "problems". The obvious one is `typedef` names. Fortunately, the *yacc* we use does provide a limited retry mechanism that gives enough help to allow full handling for nested redeclarations of `typedef` names. The other problem is, strictly speaking, not a real parsing problem, but turns out to be the single biggest roadblock to naive processing of C code: function definitions require a complete parse to be recognized with confidence. ANSI C has had no negative or positive effects on these parts of C.

`float` and `long double` Math Libraries

Do you, or are you likely to, supply a `float` *and/or* `long double` *version of the math library?*

- 7 – Yes, `float`
- 6 – No, `float`
- 5 – Yes, `long double`
- 8 – No, `long double`
- Comments:

  1. On our machine `float` and `double` have the same representation.
  2. We're thinking about it.
  3. It is very unlikely that we will supply a `float` library. The majority of users interested in using a PC to do floating-point are willing to buy the floating point hardware, which does all of its computations in extended precision. The calling sequence we use allows one to write a math library that performs `long double` arithmetic internally and then narrow the results to `double` only when stored in the calling program. Thus the standard math library could be modified to be a dual precision library depending on how the function results are stored. I don't think that we have all of our math functions currently working internally in `long double` precision yet, however.
  4. There has been no demand for it from our customers. If that changes, we will provide whatever libraries they need.
  5. We have all the code in our FORTRAN library. However, we have no immediate plans to incorporate it in the C library but we might in the future.
  6. The `long double` functions will be available only when `long double` becomes distinct from `double`. To provide them now would only cause more compatibility problems down the road.

Pragma Recognition

If you support pragmas, what does "recognizing" a pragma mean? Must you recognize all tokens or is there a key first token that you recognize. That is, can you diagnose a pragma you believe is intended for you but is misspelled.

- 5 – No pragmas supported

- Comments:

  1. In our previous release no special token was permitted. In the current release all pragmas can optionally start with _*xxx* where *xxx* is a company-specific abbreviation. With the next release we will only diagnose unrecognized pragma directives if they begin with this token. Otherwise, they will be ignored.
  2. I recognize and parse pragmas that start with a company-specific abbreviation, and ignore all others.
  3. I key off the first token. Pragmas are a pain, anyway.
  4. We look at the first token to determine if the pragma is one we recognize. But, since unrecognized pragmas must be ignored, any unexpected parameters to the pragma give warnings, not errors.
  5. We have interpreted "recognizing" to mean that if the first token after `pragma` is an identifier in our list of pragmas, then it is tentatively recognized. If we are in strictly conforming mode (settable with a command-line option) then if there is any irregularity of syntax or semantics the pragma is unrecognized and is not diagnosed. In the non-conforming default mode of operation, a tentatively recognized pragma will be diagnosed if there are irregularities. Only pragmas with unrecognized first tokens will be ignored without comment.
  6. Currently all our own pragmas start with our company initials. No intelligence has been put into the pragma parser as far as diagnosing one of our own pragmas being spelled incorrectly. A warning message is printed out on all pragmas that are not recognized.
  7. We have added some System V Release 4 pragmas just so we can accept SVR4 source cleanly. (Of course, these do not have the company initials in them).
  8. We cannot tell the difference between a misspelling and a pragma intended for someone else's hardware. We print out a warning and continue the compilation.
  9. We mostly refuse to support pragmas.
  10. We ignore `#pragma` and rely on compiler switches.
  11. We have special purpose code in our scanner that recognizes the pragmas we know about. It warns if it doesn't understand the line and throws it away. We don't do any spelling correction.
  12. In general, the recognition is determined by the identifier that follows `pragma`. A warning is issued if the rest doesn't work out. Any otherwise unrecognized `#pragma` directive produces a warning anyway, so the difference is academic.

RISC System Support

Are RISC machines featured in your current or future plans? What do you see the biggest challenge in developing translators for RISC targets?

- 6 – No
- 8 – Yes
- 1 – No comment
- Comments:

  1. Just another code generator!
  2. I'm just about to start an i860 back end. The problem is scheduling (imagine keeping the core unit, FP adder and FP multiplier of an i860 all busy at the same time!), with explicit timing information. Something akin to the Multiflow TRACE compiler, perhaps (dealing with the branches in a sensible way).
  3. We ship on a couple of RISC machines (SPARC, DECstation) and are porting to several others.

     The biggest challenge in producing compilers for RISC targets is making them reliable. Current compilers for RISC machines (I'm talking about the compilers shipped by the hardware vendors!) are not reliable. As a result, we ship our product compiled largely without optimization. Our product runs 10–15% slower as a result, but it's worth it to avoid compiler bugs.

     Compiler vendors should pursue reliability, even at the expense of optimization. They should also make it as easy as possible to detect when code will be broken by an optimizer.
  4. Good RISC compilers will need to do most of the optimizations (loop interchange, etc.) that are done by vectorizing and parallelizing compilers to allow enough data to be delivered to the chip to take advantage of its high speed.
  5. We only have RISC machines. They actually seem to be easier to optimize for than CISC machines because the pipelines are exposed to the compiler and because there is no need to choose between several complex addressing modes.
  6. Optimization.

∞

13. IEEE Floating-Point Arithmetic and C

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

Abstract

Currently, there is an effort to create a language binding between C and the IEEE floating-point standard. Several new linguistic features are needed to allow programmers to exploit the full capability of IEEE arithmetic. These features are intended to help numerical programmers but affect C, and other programming languages, by introducing performance issues and new concepts unfamiliar in traditional programming languages. IEEE features that have a major impact on C are discussed, along with other issues concerning programming environments and computer architecture. Many vendors may consider implementing a subset of the IEEE floating-point standard for both performance and linguistic reasons.

Introduction

Once upon a time, a group of computer scientists was explaining to a group of mathematicians how floating-point arithmetic would be implemented on computers. The mathematicians told them that this floating-point arithmetic implementation would never work because it violated too many mathematical rules and, in general, was error prone. The story goes that the computer scientists replied, "Just trust us and try it out; we know you'll like it."

The problems with floating-point arithmetic are as prevalent today as they were perceived to be back then. Someone relying on mathematically correct assumptions often stumbles into numerical inaccuracies. The *IEEE Standard for Binary Floating-Point Arithmetic* was defined to ameliorate some of the floating-point peculiarities that make it so difficult to write numerically stable algorithms. The *Numerical C Extensions Group* (NCEG) is attempting to make C more suitable for numerical and scientific programming.

This article addresses the incorporation of IEEE floating-point arithmetic into ANSI C, and explains the impact the IEEE floating-point standard can have on programming environments and computer architectures. Vendors may choose to implement a subset of the IEEE standard for both performance and linguistic reasons.

Floating-Point Arithmetic

Floating-point numbers are, in general, approximations for the mathematical real numbers. They are approximate because a real number contains an infinite amount of information, which cannot be represented in a finite floating-point format. The following is a small (and incomplete) list of mathematical identities that do not necessarily hold true for floating-point numbers:

$$
\begin{array}{rcll}
(X+Y)+Z & \equiv & X+(Y+Z) & \\
(X\times Y)\times Z & \equiv & X\times(Y\times Z) & \\
X\times(Y+Z) & \equiv & X\times Y+X\times Z & \\
1/(1/X) & \equiv & X & \text{for nonzero } X \\
(X/Y)\times Y & \equiv & X & \text{for nonzero } Y \\
(X+Y)-Y & \equiv & X & \\
X\times(1/X) & \equiv & 1 & \text{for nonzero } X
\end{array}
$$

This means that mathematical correctness does not imply numerical correctness. Consider the following simple C program run on a Cray Y-MP computer:

Example 1

```
#include <stdio.h>

double a, b;
double c = 524288.0;

main() {

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

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

}
```

The output from this program is:

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

This contrived example demonstrates the need to minimize the amount of relative error introduced by a floating-point operation to increase numerical accuracy. The next example introduces another issue.

Example 2

```
double x =  99999999999999.0;
double y = 999999999999999.0;

main(){
   printf("x = %18.1f\n", x);
   printf("y = %18.1f\n", y);
}
```

The output from this program is:

```
x =   99999999999999.0
y = 1000000000000000.0
```

The Cray C implementation converts a decimal floating-point number with 14 '9' digits to `double` format and back to ASCII without loss of information. However, the use of 15 '9' digits introduces an unsafe conversion. A different number is printed after it has been converted to binary and then back to decimal.

Both of these examples demonstrate mathematically incorrect results for the same underlying reason. That is, the benefits gained from the performance available with floating-point arithmetic outweigh the numerical inaccuracies that are introduced. There are formats other than floating-point that yield far greater accuracy but they are not widely used. The balance between performance and accuracy that a system offers is crucial to its viability in today's marketplace.

Performance Versus Accuracy

Performance sells computers. Vendors rarely claim their machines are numerically accurate. They claim that their systems can execute millions to billions of instructions every second. If an occasional digit is dropped on the floor, then that is the price you pay for all of these billions of floating-point operations every second. Most customer benchmarks do not directly test the accuracy of the vendor's floating-point implementation. Rather, an acceptable answer must be reached in a minimum amount of time. Performance will continue to sell systems until customers demand accuracy.

Of course there are two sides to the performance versus accuracy debate. If you lose enough accuracy then the next automobile you purchase may not be quite so aerodynamic, causing a decrease in your miles per gallon rating. It might rain even though the weather forecast called for sunny skies. You might drill the next oil well in the wrong location.

However, if performance is neglected in favor of accuracy then the design phase of the new automobile you are manufacturing may take an additional

18 months and increase development costs by several million dollars. Meanwhile, your competitor will beat you to the marketplace and undercut your price. The 24-hour weather forecast may take more than 24 hours to generate.

IEEE Arithmetic

The IEEE standard defines a way to perform floating-point arithmetic. To some extent this is a concession to performance. There are other formats that yield greater accuracy, but none that I know of that can be implemented efficiently. The foreward of the ANSI/IEEE Standard 754-1985 contains the following quote about its goals:

> "Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists."

The impact of IEEE arithmetic on programming languages will definitely be noticed by both users and vendors. Currently the impact on C is being explored by the NCEG committee. The committee hopes to produce a technical report that includes an IEEE binding for ANSI C. The following discussion does not address all implications of the IEEE binding; but rather those with the greatest impact on C, programming languages in general, and computer architecture.

Signed Zero

IEEE requires a floating-point representation for both $+0$ and -0. Although $+0$ and -0 have different representations they must compare equal! The intended purpose of the sign is presumably to record the direction of underflow. The linguistic implications are many. The compiler is no longer free to convert the expression $X + 0$ to just X anymore because $(-0) + (+0)$ yields $+0$. This has been a long-standing acceptable optimization that IEEE floating-point arithmetic forbids. It also adds the following identity to the list of mathematical laws that are no longer numerically correct:

$$X + 0 \equiv X$$

This forced a change in one of the draft versions of the ANSI C standard. The unary `+` operator was defined as:

> "The expression `+X` is equivalent to `(0 + X)`."

To accommodate the IEEE standard, this definition was changed to:

> "The result of the unary `+` operator is the value of its operand."

Similarly, the definition of the unary `-` operator changed. The definition used to be:

> "`-X` is equivalent to `0 - X`."

and was changed to:

> "The result of the unary `-` operator is the negative of its operand."

Since -0 represents a negative zero but $0 - 0$ evaluates to a positive zero the sign of the result appears to be arbitrarily chosen, unless the rounding direction is toward minus infinity. Since the intent of signed zeroes in IEEE arithmetic is presumably to record the direction of an underflow, `sqrt(-0)` should be an invalid operation. However, `sqrt(-0)` is defined to return -0, indicating that the rules are not consistent. It becomes unclear what a signed zero really is. If -0 is recording the direction of an underflow, then -0 really represents a negative number too small to represent, because underflow does not occur when the result is exactly zero. Clearly, `sqrt(-0)` should then trigger an invalid operation. An argument could then be made that

$$(+0) \; + \; (-0)$$

is invalid (and evaluates to a NaN which is defined below) because there is no way to deduce the correct sign of the result when adding a *too small* negative number to a *too small* positive number. IEEE floating-point arithmetic should admit to that, instead of arbitrarily picking a sign out of the hat. The ultimate failure of signed zeros is that now neither $+0$ nor -0 can be interpreted as a mathematical zero because they mean 'negligibly small' instead.

The presence of signed zeros in the IEEE floating-point standard is unfortunate. There is no basis in conventional mathematics for signed zeros and the implications are many. For instance, many floating-point implementations satisfy the following mathematical law:

$$X \; - \; Y \quad \equiv \quad -(Y \; - \; X)$$

However, IEEE floating-point arithmetic takes on the following interesting property:

$$1 \; - \; 1 \quad \equiv \quad +0 \quad \text{and} \quad -(1 \; - \; 1) \quad \equiv \quad -0$$

(at least for most rounding directions) creating another optimization barrier.

Another justification given for the presence of signed zeros in the IEEE floating-point standard is to support conformal mapping. Signed zeros permit certain complex functions to remain formally continuous in some end-cases. Some of these complex functions are: `sqrt`, `log`, `asin`, `acos`, `atan`, and `pow`. The square root of z, where z is a complex number, is discontinuous over the negative real axis. The straight line representing this discontinuity is called a

slit. Signed zeros allow a numerical programmer to omit tests as the value of z approaches the slit. The penalty that everyone pays by catering to these end-cases is, in some cases, degraded performance. But mostly, it is inconsistent and arbitrary specifications that require zero to have a sign. The trade-off is definitely questionable.

Another justification given for signed zeros is that the following relation:

$$1/(1/X) \quad \equiv \quad X$$

is true for both the signed zeros and the signed infinities. This is because the following relations are defined:

$$\begin{array}{lcl} 1/+\infty & \equiv & +0 \\ 1/-\infty & \equiv & -0 \\ 1/+0 & \equiv & +\infty \\ 1/-0 & \equiv & -\infty \end{array}$$

However, as mentioned earlier, due to the very nature of floating-point arithmetic, this same relation is not true for many finite values of X. This is not a compelling reason to support signed zeros, especially when unnatural architectural support must be present to force two different floating-point representations for $+0$ and -0 to compare equal.

Not a Number (NaN)

The IEEE floating-point standard defines an entity called *Not a Number* (NaN) that represents the result for certain exceptional computations such as $0/0$ and $\sqrt{-1}$. Traditionally, the evaluation of $0/0$ results in abnormal program termination, or transfer of control to a signal handler. Generating a NaN result allows normal program execution to continue for these exceptional conditions.

The introduction of a NaN representation means more mathematical laws are violated. The following mathematical relations do not hold true when X is a NaN but do apply to most other floating-point representations:

$$\begin{array}{l} X \times 0 \equiv 0 \\ X - X \equiv 0 \\ X == X \text{ is true} \\ X != X \text{ is false} \\ \text{exactly one of } X < 0,\ X == 0, \text{ and } X > 0 \text{ is true} \end{array}$$

Again, many classic compiler optimizations are no longer permitted.

IEEE arithmetic has introduced a fourth comparison state called *unordered*. That is, since a NaN does not compare equal to, less than, or greater than any X, the relationship between a NaN and X is said to be *unordered*. Much of the burden of this additional complexity is imposed on the programmer. There is some debate within NCEG as to whether the following additional operators are needed:

`!<=>`	unordered
`<>`	less or greater
`<=>`	less, equal, or greater
`!<=`	unordered or greater
`!<`	unordered, greater, or equal
`!>=`	unordered or less
`!>`	unordered, less, or equal
`!<>`	unordered or equal

Other possible solutions include defining library functions such as `isnan` and `isunordered`, or a more general function `isrelation` that handles all relations. The addition of so many new operators to the ANSI standard has been the subject of much debate. The argument against `isnan` is that it would require pervasive use of the logical operators:

```
if ((!isnan(X) && !isnan(Y)) && X < Y) { /*...*/ }
```

The current consensus seems to be that the addition of so many new relational and equality operators is too great a change. The `isnan` or `isunordered` alternative is not an attractive prospect. Therefore, I suspect that the function `isrelation` will be adopted by NCEG. The `isrelation` function would likely be declared in the header `<numeric.h>` along with the following macros:

```
#define FP_LESS
#define FP_GREATER
#define FP_EQUAL
#define FP_UNORDERED
```

The statement:

```
if (isrelation(X, FP_UNORDERED | FP_LESS, Y)) { /*...*/ }
```

could be written as:

```
if (X !>= Y) { /*...*/ }
```

if the additional relational operators are added. An equivalent way of writing this statement using the `isnan` functions is:

```
if (isnan(X) || isnan(Y) || X < Y) { /*...*/ }
```

Again, common mathematical practice does not admit to the presence of a NaN. This also requires special architectural considerations to ensure that two identical representations compare unequal (remember, NaN == NaN evaluates to false) Furthermore, the following example:

```
if (x < y) func1(); else func2();
if (x >= y) func1(); else func2();
```

could surprise the IEEE-naive programmer because function `func2` is called for both cases if either `x` or `y` is a NaN. That is, the equivalence:

$$X \;<\; Y \quad \equiv \quad !(X \;>=\; Y)$$

does not apply to IEEE floating-point arithmetic.

Infinity

IEEE arithmetic defines both a positive infinity and a negative infinity. In general, infinities are quite useful for determining the direction of overflow of operations on real numbers. The only unfortunate aspect of signed infinities is that they do not map onto Cartesian complex numbers in consistent ways because they can only capture the notion of infinity in one of four directions: $0, \pi/2, \pi$, and $3\pi/2$. Section 6.2 of the IEEE standard states:

> "Signaling NaNs afford values for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard."

This is an explicit admission that the rules do not extend to IEEE complex numbers. A projective infinity (without a sign) permits the use of polar coordinates where magnitude (absolute value) infinity can come in the directions (angles) 0 through 2π.

The IEEE notion of infinity is different from the geometric model described below. Imagine a sphere and place it on the complex plane right on the origin. A line drawn from the very topmost point of the sphere to any point in the plane will intersect the sphere at one point. In this way, each point in the plane can be mapped in a one-to-one way onto a unique point on the surface of the sphere. All the points in the plane infinitely far from the origin, no matter in which direction, map onto the topmost point of the sphere. The geometric model defines a unique infinity.

However, there are other ways to map the points infinitely far from the origin. Imagine a hemisphere placed on the complex plane at the origin. (Similar to a bowl sitting on a table). The top of the hemisphere is the *top circle*. Map the entire complex plane one-to-one onto the surface of the hemisphere by drawing a line from each point on the complex plane to the center of the top circle. In this model there are infinitely many infinities, all along the circumference of the top circle. This is the IEEE model where $+\infty$ and $-\infty$ are the two points on the real axis that are infinitely far from the origin. This model is well suited for real numbers but does not extend very well to a Cartesian representation of complex numbers. Polar coordinates are a natural way to exploit the geometric infinity because there is no sign. Unfortunately the IEEE standard does not define a projective infinity, making it an excellent candidate for a future extension. So far, the NCEG committee has declined to consider the possibility of directly supporting polar representations of complex numbers.

Input and Output

Adding NaNs, infinities, and signed zeros to C also affects the I/O and string conversion functions. There must be ways of reading and writing these new linguistic entities. For instance:

```
scanf("%f", &x);
```

could be made to accept any of the following input lines (in either upper or lowercase):

```
+NAN
-NAN
+INFINITY
-INFINITY
```

Similarly:

```
printf("%+e", x);
```

could be made to print any of the following output lines:

```
+nan
-nan
+infinity
-infinity
```

while:

```
printf("%+E", x);
```

could be made to print the same output but with the words spelled in uppercase.

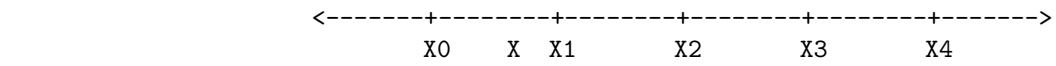
While it is clear that some mechanism for printing NaNs and infinities is required, there are still details to be worked out. For instance the format for a signaling NaN is proposed to be:

`+NAN(`*n-char-sequence*`)`

There are questions about allowing `INFINITY` (without an explicit sign) to be an acceptable numeric input line for `scanf`. The functions `atof` and `strtod` are also affected by NaNs and infinities in a similar way.

Decimal Conversions

Example 2 shows that the Cray Research Inc., implementation has a limit to the number of decimal digits that can be converted to floating-point format and back to decimal representation without losing any information. There has been considerable debate recently about just what conversion requirements should be imposed on IEEE floating-point implementations. The term *correctly rounded* is used to describe a number that has been converted to match the closest floating-point representation. Since a real number X is probably not exactly representable, it is desirable to pick the correctly rounded value. This can be viewed as:



where $X1$ is the floating-point number that is closest to the real number X. It is certainly possible to develop conversion functions that perform correct rounding, but again there is a conflict between performance and accuracy. The ANSI C standard (§3.1.3.1 Floating Constants) contains the following quote:

> "If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner."

This means that the ANSI C standard permits an implementation to choose either $X0$, $X1$, or $X2$. The IEEE standard requires correct rounding for a large but incomplete subset of decimal conversions. The current proposal before the NCEG committee, however, states that:

> "The value of a constant with `DECIMAL_DIG` or fewer decimal digits is *correctly rounded* (where `DECIMAL_DIG` is the number of decimal digits to which an implementation can correct round)."

The rationale given for this extension to the IEEE standard is that "practical methods are now available." This is a source of controversy. This rationale does not address performance. It is easy to develop perfect conversion routines, but it is not clear that the performance of these methods will compare favorably with the current techniques. For instance, Cray floating-point format has eight times the dynamic range of the IEEE 754 64-bit floating-point format. The worst case requires simulating exact integer arithmetic with operands exceeding 8,000 bits. Even the proposed `DECIMAL_DIG` restriction is controversial. This is an obvious concession to performance. If perfect conversion is the overriding concern, then why not require a 1000 digit floating-point number to be correctly rounded? The answer is that performance always wins.

Conclusions

There are certainly many good things to be said about the IEEE floating-point standard. It has done a great deal to promote portability for numerical and scientific applications across implementations. It raises many thought provoking questions, however. We will learn more about the impact of IEEE arithmetic on programming environments by creating a C binding.

I do believe, though, that blind adherence to the IEEE standard is not necessarily in the best interest of all vendors and their customers for both performance and linguistic reasons. Vendors need to question the effects of certain aspects of the standard when performance is an issue. Some of these considerations are:

- signed zeros
- round to $+\infty$
- round to $-\infty$
- perfect rounding for decimal to binary conversions

Problem solving requires creative techniques and tools. Blind adherence to everything in the IEEE floating-point standard could be the cause of losing an important benchmark and thus an important sale. On the other hand, numerical instability is a major source of irritation for customers. The ability to balance these conflicting needs is a challenge. We may have to incur the burden of providing multiple arithmetic environments to support diverse customer needs. One environment provides slower but more accurate arithmetic and the other provides high performance but less accurate arithmetic.

I would like to especially thank Tim Peters from Kendall Square Research, Inc., for his insight and valuable opinions in preparing this article.

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) 683-5818, tam@cray.com*, or* uunet!cray!tam.

[Ed: Note that Tom's postal address and telephone number have changed.]

∞

14. Signal Handling and C

Gordon A. Sterling
Analog Devices, Inc.
P.O. Box 9106
Norwood, MA 02062-9106

Abstract

The ANSI C language standard contains information on how to process signals directly from C. Since signal handling is machine dependent the requirements of the standard are broad enough to allow compliance in a general manner. Implementors are free to add extensions to support the special characteristics of their target machines.

The C compiler and runtime library developed at Analog Devices contains extensions that increase applicability of the C language to embedded Digital Signal Processing (DSP) applications. In this article, we will look at some of these extensions.

Introduction

The power of C has been available to developers of interactive software for many years. However, the world of real-time embedded systems such as radar, telecommunications, graphics engines, and control systems had been consigned to the fate of hand-coded assembly language.

The past few years have seen a dramatic increase in the number of C compilers available for embedded systems development. These compilers are often used for developing control code for an application, but a lack of efficient code generation and access to a processor's hardware features has hindered the development of complete DSP applications in C. While code generation is improving, work must still be done to provide access to the hardware.

Signal handling enhancements can be used to access some of the special purpose hardware provided with today's processors. Applications that require IEEE floating-point hardware, interprocessor communications, and external communications need a way to interface to these functions. These resources are often supported through the use of signals.

This article covers some of the work being done at Analog Devices to provide a software interface to the hardware signals of the host processor.

A Refresher Course in Signals

For the purpose of this article, *signals* will refer to two things, interrupts and flags. An *interrupt* is a signal used to indicate a situation that requires the immediate attention of the processor. (A common example of an interrupt is the Control/C key combination on an ASCII keyboard.) If an interrupt occurs that is not being ignored (masked out), normal program flow is altered. The processor begins to execute an interrupt service routine.

An interrupt service routine is a small routine that is designed to deal with a particular interrupt. It may take some corrective action, sample data, or perform some other task. These routines are characterized by their small size and clearly defined purpose. A service routine to handle an inadvertent Control/C might ask the user for confirmation before exiting the program, for example. When the service routine has completed, the interrupted program resumes execution. It is not obvious to a program that the interrupt ever happened.

A *flag* differs from an interrupt in that it does not indicate a critical condition. The situation that activates the flag does not require the immediate attention of the processor. A flag is activated and the program can test it during normal program flow.

A flag might also be used for interprocessor communications. One processor might inform another that data is available in common memory. The second processor can test the state of the flag and act on the data when it has time.

Hardware Support for Interrupts

There are many different schemes for supporting interrupts in hardware. One of the most common approaches, where a vector table is placed in memory, is described in this section.

Interrupts generally come in two flavors, internal and external. *Internal interrupts* are those that are generated within the processor itself. Two examples are an internal timer and a floating-point operation that generates an exception.

External interrupts come from outside the processor. They are indicated by a change of state on an external pin. A chip can provide separate pins for each interrupt, or a single pin that connects to every interrupt source.

The vector table approach relates a specific memory range to each interrupt. The range can be a single location, or several contiguous locations. When an interrupt occurs, the processor jumps to the designated memory address. In some cases, the service routine can fit within the memory range, otherwise the location contains a jump to the actual routine.

There is an established protocol that the processor follows concerning how and when an interrupt service routine can itself be interrupted. The simplest method is to shut off (or mask out) other interrupts while executing a service routine.

Ignoring interrupts has some disadvantages. If a more important interrupt occurs (such as a stack overflow) while another is being serviced, information

may be corrupted. It is also possible that an interrupt may lock-up or never return. Since other signals would be masked out, the system might have to be powered-down, or reset, to regain control.

The problems described above can be minimized by prioritized interrupts. This technique assigns a rank or priority to each interrupt. A higher priority interrupt will be serviced even if another (lower priority) interrupt is active.

There are other signals that may be included in the vector table but are not usually considered interrupts. The reset signal often has a vector table entry. Once the chip has completed the hardware reset operation, the first instruction executed is located at the reset vector address.

This article ignores reset because it is not used to signal an event that a program would handle. The reset signal may occur at system power-up or some other extraordinary event. Reset will preempt all other actions of the chip, and is usually referred to as *non-maskable*, meaning that it cannot be ignored.

Signal Handling in ANSI C

The signal handling capabilities of the ANSI C standard are open-ended to allow implementors to customize the library to their particular environment. The required support for interrupts is covered by two routines: `signal`, which is called to identify a particular routine to be executed upon receipt of the defined signal, and `raise`, which provides a mechanism to invoke the handlers through software.

The `signal` routine has several drawbacks. The most significant problem encountered by a real-time application is that `signal` only provides handling for one invocation of an interrupt. After an interrupt has been serviced once, handling is returned to the default state.

An application that is only interested in catching and handling a Control/C in a clean fashion may not need to deal with multiple interrupts. Once the first interrupt occurs, the program jumps to a cleanup routine and exits. The world of real-time systems, however, does utilize repetitious interrupts. A periodic interrupt might be designated as a sampling clock. The signal would indicate that a converter or other device needs to be serviced. For example, an application involving speech processing might sample a converter at 8000 times per second.

A system for supporting interrupts that requires the interrupt service routine to be re-designated after each invocation would incur an undesirable overhead. Support for continuous handling of a signal can be provided by the addition of a single routine. The new routine, referred to as `interrupt`, operates like `signal`. The only difference is that once a handler has been designated by a call to `interrupt`, it will continue to be invoked every time the specified interrupt occurs. A routine that reads a sample from a converter at a specified interval can now be used with little overhead.

Analog Devices' Implementation

The Analog Devices family of DSPs supports a vector table with prioritized interrupts in hardware. Currently only non-prioritized interrupts are supported by the library, but the library and C compiler in development for the ADSP-21000 will support prioritized interrupts.

Analog Devices has two processor types with production grade C compilers at this time. The ADSP-2100 microprocessor supports four external interrupts (IRQ0, IRQ1, IRQ2, and IRQ3) that have a pin dedicated for each. A single memory location is provided for each interrupt. The ADSP-2100 does not generate any internal interrupts.

The ADSP-2101 microcomputer provides memory, serial ports, and a timer on-chip. The timer and serial ports are supported by several internally generated interrupts. The interrupt vector table provides additional space to support these interrupts, and also increases the space dedicated to each interrupt. There are four memory locations assigned to each interrupt. Assembly routines that are three cycles or less can be placed entirely in the vector table. No addition jump is needed.

The software implementation begins with an interrupt controller. This code is executed before the handler. The controller has three primary responsibilities: saving registers, determining what will happen when this same interrupt occurs next, and calling the service routine.

Whenever a C subroutine is entered, most registers are saved on the stack. (Several scratchpad registers would not normally be saved during a subroutine call. They must be saved on the stack before calling the server.) There are also registers that the language assumes will contain specific values. Since the interrupt may have occurred during a section of the user's assembly code, where their values may have been changed, the registers are saved and set to the default values.

The last step before calling the service is to determine what action will be taken the next time this interrupt occurs. This is accomplished by dedicating two memory locations to each interrupt. The first location contains the address of the handler for this invocation. The second contains the handler's address for subsequent invocations.

The first location is read and used for the CALL instruction. The second location is copied into the first. The importance of this step can be seen by the way `signal` and `interrupt` set these locations.

The `signal` routine sets the first memory location to the address of the handler specified. The second location holds the address of a known Return-From-Subroutine (RTS) instruction. The first invocation of an interrupt will cause the handler to be executed. Subsequent invocations will be a call to an RTS, which has no net effect.

`interrupt` sets both the first and second memory location to the address of the handler. The interrupt controller re-loads the address of the handler (from the second location), causing the same handler to be executed repetitively.

Assembly Language Interrupt Support

The routines provided in the extended signal library are fast enough to be used in many real-time applications. Those that cannot tolerate any overhead can be hand coded in assembly language. Although these routines would not be written in C, the language must still provide support for the programmer.

Special purpose hardware often has dedicated registers that can be used for high speed operations. The ADSP-21xx microcomputer supports high speed serial port/memory transfers called autobuffering. During a port write operation, autobuffering copies a single value from a memory buffer and sends this value to the serial port. An interrupt is generated only after the entire buffer is transferred.

The single instruction needed to execute this operation is generated by the processor internally. The programmer has no way of altering the code (such as saving registers). This requires that certain registers be dedicated to the autobuffering operation. The three dedicated registers contain a pointer to the next memory location, a step pointer, and an array length pointer.

When an entire buffer of data has been written (or read), a serial port interrupt is generated. The program can provide a new buffer for output. Program interruption is shifted from one interrupt per sample to one interrupt per buffer (which can be up to 16k samples).

This operation provides such a dramatic increase in efficiency over a C language service routine that there needs to be a way to support it. The ADSP-21xx C compiler provides a pragma directive that reserves specific registers for use in autobuffering operations.

When an application requires the use of autobuffering, the pragma directive defines which registers will only be used for that purpose. The C compiler will not use those registers during code generation. The efficiency of the compiler is slightly reduced due to the register limitations, but the overall speed increase can be significant.

Special Purpose Interrupts

When a processor contains an internal timer, it is often supported as an interrupt. Although any support provided for interrupts in general will be applicable to timers, there are other extension that are necessary for a program to be able to utilize a timer.

A timer is different from the clock that maintains system date and time. There may or may not be an actual clock associated with a timer. The timer is also distinct from a clock in that it counts processor cycles. Since the compiler is developed for the processor, not a complete system, the clock speed and other architectural features are unknown.

The extensions developed for the ADSP-21xx library include three routines that provide access to the internal timer. The activation/deactivation routines

are `timer_on` and `timer_off`. These are very fast routines. They start or stop the processor and return the current timer count. The parameters of the timer are set using the `timer_set` routine.

Other Signals of Interest

Not all signals of interest to an application generate interrupts. A flag input signal is one such example. A flag might be connected to an external pin. This provides a mechanism for the outside world to signal a program that an event less critical (than an interrupt) has occurred.

The event signaled by a flag does not require the program flow to be interrupted, but may be of interest for the program when time is available. A flag may be generated by an internal event. A computational unit (such as the adder) might signal that some input has exceeded a specified limit. The program may, or may not, take action on this information depending on the situation.

The extension provided in our compilers includes a routine to test the state of various flags supported by the processors. The routine, `poll_flag_in` takes as input the flag name (such as `FLAG_IN_0`), and a condition to test. The routine checks the specified flag and does not return until the specified condition is satisfied.

There are five basic conditions against which a program might wish to test a flag. The test may be for a flag state, such as low or high or a certain transition might be desired, such as low-to-high or high-to-low. Finally, the program might wish to test for any transition at all.

Conclusions

The ability to control a variety of signals directly is one important step in the acceptance of C as a development language for embedded systems. The ANSI C standard provides a very general framework which can be enhanced to provide a useful signal handling environment.

The library and language extensions described in this article are being used for some DSP projects. Further extension to the signal library are being considered by the *Numerical C Extensions Group* (NCEG). NCEG is expected to include support for the floating-point exceptions that can generate runtime signals.

Gordon A. Sterling is a Signal Processing Engineer at Analog Devices, Inc., a designer and supplier of DSP hardware and software development systems. He can be reached at (617) 461-3076. Analog Devices is active in NCEG.

∞

15. Concurrent C: A Language for Multi-Processing

Steven Lally
AT&T Bell Laboratories
200 Laurel Avenue
Middletown, NJ 07748

Abstract

The C language has prospered for both its expressiveness and its brevity. That expressiveness includes its suitability as a systems language, used to express (among other things) the UNIX operating system. Yet, while the UNIX System V development effort turns to supporting multi-threading facilities, its "native language" C provides no explicit language facilities for instantiating and controlling multiple threads. Concurrent C, a language developed by Narain Gehani and William D. Roome at AT&T Bell Laboratories, provides a concise notation for multi-processing on multiple processors as well as multi-threading on a uni-processor, while supporting full and correct C.

Concurrent C is a superset of the C language that provides facilities for abstracting events into defined processes. These processes are created dynamically, execute in parallel, and interact with each other to carry out some unified and purposeful activity. The purpose of this article is to provide both a conceptual and syntactical overview of the language, as well as to demonstrate its orthogonality to another C extension, C++.

A C Extension for Multi-Processing

The guiding principles underlying UNIX system design—"small is beautiful" and "less is more"—apply to the C language as well. Indeed, C, with a keyword list of 32 words, might qualify as a "little language." The simplicity of its design and flexibility of its syntax rules can lead to concise and expressive code.

In keeping with other small languages supported under UNIX, C-related semantics have been expanded by preprocessors and functions, leaving the base language unchanged. For example, C has been extended grammatically to support explicit facilities for data abstraction, as described by Stroustrup [7]. (Implemented as a preprocessor, C++'s *cfront* has assisted rapid language evolution, obviating repeated revisions to users' compilers.)

C's expressiveness includes its suitability as a systems language. Yet, as UNIX System Laboratories, Inc. turns its attention to multi-threading, UNIX System V's "native" language C provides no explicit facilities for instantiating

and controlling multiple threads. These semantics, however, have been provided by the extension Concurrent C.

Concurrent C is an upward-compatible superset of C that provides facilities for multi-processing as well as for multi-threading on a uni-processor. In addition, Concurrent C supports multi-processing on a loosely coupled multiprocessor over a Local Area Network (LAN). Implemented as a C language extension, it supports facilities for process creation, process control, and synchronous and asynchronous transactions. Concurrent C transactions are modeled after the extended rendezvous of Ada.

Like C++, Concurrent C is implemented as a language extension rather than as a library of functions for concurrent tasking. Consequently, it is possible to provide a notation that is intuitive to the underlying problem domain. Gehani [5] stakes out the position in favor of language extensions as follows:

> Library functions for specifying concurrency are in general awkward to use, for the following reasons:
>
> 1. Providing the needed functionality is often not straightforward.
> 2. Using them can have a negative impact on program readability because they do not make the concurrent parts of a program easily identifiable.
> 3. A compiler might not be able to perform type checking and optimization because it would not be able to tell the difference between concurrent and sequential programming facilities.
> 4. Function call overhead might result in a certain degree of inefficiency.
> 5. It might be difficult to design a standard set of functions that can be implemented efficiently on a wide class of machines.

Beginnings

Multiple CPU environments, both loosely and closely coupled, are becoming increasingly common. Concurrent C has been designed to exploit these processors. The language's inventors (in [2]) list four primary motives for having designed Concurrent C:

1. Concurrent programming facilities are notationally convenient and conceptually elegant when used for writing systems in which many events occur concurrently, for example in operating systems, real-time systems and database systems.

2. Inherently concurrent algorithms are best expressed when the concurrency is stated explicitly; otherwise, the structure of the algorithm may be lost.

3. Efficient use of multiprocessor architectures requires concurrent programming.

4. Concurrent programming can reduce program execution time even on uniprocessors, by allowing input/output operations to run in parallel with computation.

Language Overview

As the function is the basic building block in C, so the *process* is the basic atom in a Concurrent C program. Under Concurrent C, processes are created dynamically, execute in parallel, and interact with each other to carry out some unified and purposeful activity. Software construction, then, begins by isolating and analyzing the *events* that must be carried out.

Each event may be defined in terms of a separate Concurrent C process. The interface to each process is declared in a *specification* (using the keyword `spec`). The implementation of each process is defined within a process *body* (using the keyword `body`).

Process Interaction

Processes may interact by way of named *transactions*. Each process can initiate a transaction (by naming it) and/or accept a transaction (using the keyword `accept`). Information exchange can be bidirectional; transactions, therefore, can return values (using the keyword `treturn`). Transactions are declared using the keyword `trans`.

A transaction may be accepted according to a rich set of conditions set forth in `suchthat` clauses, which set forth tests for acceptance, and `by` clauses, which establish priority for acceptance. A timeout mechanism allows transactions to be withdrawn if they are not processed within a specified interval (using the keyword `within`).

Beyond the conditional acceptance of transactions supported by the `accept` statement itself, a higher level of abstraction may be applied to transaction acceptance using a `select` statement. This statement allows one of multiple `accept` statements (as well as alternatives to transaction acceptance) to be selected based on the return value of Boolean expressions, or *guards*. The `select` statement is specified using the keywords `select` and `or`.

Process Scheduling and Assignment

One of Concurrent C's strengths is its highly intuitive interface to both tightly coupled multi-processors and loosely coupled processors connected via a LAN. (Concurrent C has been ported to loosely coupled systems of unlike machines.)

While process scheduling is handled by an underlying scheduler (process priority and the duration of each process run are left to the implementation of each Concurrent C port), individual processes may be delayed (using the keyword `delay`) and timed out (using the keyword `within`). In addition, a

process may specifically be assigned a processor (using the keyword `processor`), provided that multiple processors exist.

Process Model

Posing a basic question at this point brings to light several important issues: "What is a process in the context of Concurrent C?" In general, we would say that it is a single "thread of execution" with its own stack and machine registers. But this answer begs further questions: "How do we differentiate between a dynamic instance of a Concurrent C program, which is a process, and the multiple, concurrent processes instantiated within each program?" "Are the concurrent processes within a Concurrent C program independent `a.out`s?" Answers vary depending upon the environment to which the language has been ported.

Heavy-Weight vs. Light-Weight Processes

The UNIX system (under which Concurrent C was developed) supports multi-processing, of course. UNIX system calls (e.g., `fork`, `exec`, and `pipe`), however, are not primitive to Concurrent C's parallel programming facilities [4]. The UNIX pipe, expecting interprocess communication (IPC) to be linear and unidirectional, does not support generalized IPC. In addition, the use of UNIX system calls to create and control processes imposes a significant performance penalty. Each such process occupies its own address space and is known as a *heavy-weight process*. By contrast, Concurrent C processes running on a uni-processor can occupy the same address space. Not being bound to a particular address space, these *light-weight processes* allow for fast context-switching. Because the UNIX system kernel must change address spaces when it changes processes, a UNIX process context switch can be quite expensive ($\approx$1 ms on a VAX 11/780). A Concurrent C process switch is significantly more efficient ($\approx$50 μs).

Processes in a multi-processor environment, however, may be instantiated as independent `a.out`s. Each such process may be assigned its own processor, carry out separate tasks, and later rendezvous to exchange information.

Concurrent Programming Model

The programming model selected for Concurrent C is based on message passing. (A shared memory mechanism, called *capsules*, has been proposed for a future release of the language [6].)

Message Passing Facilities

Multiple processes communicate with each other using both synchronous and asynchronous transactions. The model of information transfer during a synchronous transaction is that of the *extended rendezvous*. That is, two processes interact, exchange information bidirectionally, and then part, all within in a single rendezvous.

The relationship is frequently described in terms of a client and a server. The client requests the services of another process via a transaction. The process furnishing the service, called the server process, rendezvous with the client, carrying out the client's request while the client waits for (blocks on) the server process. When the server has completed its task, the client receives the results, and the two part (though they may rendezvous again).

Concurrent C also supports asynchronous transactions. Here, processes can compute and perform message sends in any way they want. Asynchrony maximizes parallelism; potential performance is further enhanced because no process synchronization is required, and information exchange is only unidirectional. A process sending a message does not block until the receiving process gets around to accepting the transaction.

A Concurrent C Program

It is time for a taste of Concurrent C. The following example expresses the client-server process relationship notationally (though its ambition as a program is modest). A process `client` reads a character stream from `stdin` and asks another process `server` to convert all lowercase alphabetical characters to uppercase, then print the result on `stdout`.

The example programs below are a review of the concepts and keywords introduced thus far.

Process Specifications

The Concurrent C keyword for declaring a process is `spec`. The following two process specifications are stored in the file `cli_serv.h`:

```
process spec client(process server Client);
process spec server()
{
    trans void serv(int ch);
};
```

These expressions show the two primary roles a process may play—initiating and accepting transactions. In fact, a single process may play both roles.

Processes that initiate transactions, such as `client`, must declare the processes with which they will transact. Here, `client` specifies in its parameter list that it will initiate a transaction with process `server`. In addition, it specifies that the point of rendezvous will be called `Client`. Consequently, `Client` will be defined within the `client` process, not the `server` process.

By contrast, processes that accept transactions declare those transactions within braces. Process `server` specifies that it will accept a transaction (using the keyword `trans`) named `serv` and that `serv` will involve the transfer of integer-valued objects. We do not know the name of the process that will initiate this transaction. This is appropriate, as many clients may call upon a server process. Transaction `serv` does not return information to process `client`, so it is declared to be `void`. (Transactions can be declared to return information: all synchronous transactions are potentially bidirectional.) Note that all transactions are synchronous by default.

Process Bodies

The keyword for defining a process is `body`. The bodies of the two processes, stored in the file `cli_serv.cc`, are as follows:

```
#include <stdio.h>        // getchar, putchar, and EOF
#include <ctype.h>        // islower and toupper
#include "cli_serv.h"     // process declarations
process body client(Client)
{
    int ch;
    while ((ch = getchar()) != EOF)
        Client.serv(ch);
    Client.serv(EOF);
}

process body server()
{
    int do_up;
    for (;;) {
       accept serv(ch)
          do_up = ch;
       if (do_up == EOF)
         break;
       if (islower(do_up))
         putchar(toupper(do_up));
       else
         putchar(do_up);
    }
}
```

```
main()
{
    create client (create server());
}
```

This program could have been simply a C program; Concurrent C is a superset of C. As such, it includes preprocessor directives and C language constructions. In addition, Concurrent C is orthogonal to another C extension: C++. Consequently, C++ constructions, such as the comments in the preceding example, are valid input to Concurrent C.

The processes in this program are defined using a facility similar to the one for defining functions in a C program. Two processes are defined: `client`, which rendezvous with a process `server`. The point of rendezvous is named `Client`: process `client` sends characters from `stdin` to `Client`, expecting acceptance via the transaction `serv` (`Client.serv`); process `server` accepts (using the keyword `accept`) these integer-valued objects via its transaction `serv` (`serv(ch)`).

How do the two processes synchronize? They are created in `main` such that the process identification number (PID) of process `server` is passed to process `client`.

Compilation and Execution

Concurrent C programs are named with the suffix `.cc` and are compiled with the command `CCC`. For example:

```
$ CCC -o cli_serv cli_serv.cc
$ cli_serv
cli_serv.cc:
$ cli_serv
hello, world!
HELLO, WORLD!
control-D
$
```

Instantiated Processes

We have said that a process, in the context of a uni-processor, is one of (potentially) many threads of execution in a single address space. In the context of a multi-processor, it is one of (potentially) many cooperating `a.outs`. Notationally speaking, we would say that a process is the instantiation of a process definition. It is the creation (using the keyword `create`) of a process that has been defined in the same program.

We have commented on the similarities of Concurrent C process definitions to C function definitions. These similarities extend beyond the merely formal. For example:

- As a function body is a static block of code, so a process body is a passive, static entity.
- As a function body is unique within a C program, so a process body is unique within a Concurrent C program.
- As a function body can have variable declarations and receive copies of variables, so a process body can declare and receive copies of variables.
- As a function body can be activated, so a process body can be instantiated. Each can terminate and begin activation/instantiation again.
- As there can be multiple activation records of the same function on the stack (as in recursive calls), so multiple instantiations of the same process body can occur at any time during the life of the program.

Functions such as `printf` may be called within an instantiated process; such a function is said to be executing on behalf of the process.

Improving Parallelism

The client-server example is very simple. Some parallelism is provided: `server` accepts transactions, performs case conversion, and prints while `client` reads from `stdin` and initiates a new transaction. The two, nonetheless, must block on each character. A program optimizing parallelism here must allow the two processes to work independently and communicate via a process suited to that purpose.

Another artificiality of the "hello, world!" program is the one-to-one relationship enjoyed by the `client` and `server` processes. In real terms, concurrent processes would compete with other concurrent processes for a single transaction. That is, the process accepting such transaction calls would *select* among the pending transactions.

The following program, therefore, introduces a third process `buffer` to intermediate between `client` and `server` and to demonstrate how transactions can be accepted from multiple processes. Consequently, it uses the Concurrent C `select` statement for choosing among multiple, pending transactions. The program is written in C++ to demonstrate the orthogonality of the two C extensions.

Process Specifications

The following file, called `proc_specs.h`, contains specifications for the three processes:

```
#include "String.h"

process spec client(process buffer Buffer);
process spec server(process buffer Buffer);
process spec buffer(int full)
{
    trans String& serv();
    trans void request(String s);
};
```

There is not much to add here. The point of rendezvous is named `Buffer`. It will be made at process `buffer`. It follows that the only process accepting transactions is process `buffer`. Process `buffer` will accept the transactions `request` and `serv`, both participating in the transfer of integer-valued objects. Transaction `serv` will carry back data from the completed service, so it is declared to return a `String&` value (defined as a C++ class below). Transaction `request` accepts objects of type `String`, but it returns nothing. Processes `client` and `server` will initiate transactions with `buffer`.

Process Bodies

The four programs below define the processes. The file `buffer.cc`, which follows, implements a process that will stand as a circular buffer between `client` and `server`. Another notable change is that we reduced the total number of I/O operations by handling strings instead of characters. (These strings are abstract data types defined in C++.)

C++ Classes and Member Functions

First, here is class `String`, defined in the file `String.h`:

```
class String
{
   char str[80];
public:
   String();
   String(const String&);
   String& operator = (const String&);
   int write(), read();
   void allcaps();
   friend int operator != (const String&, const String&);
};
```

Next, `String` member functions and friends are defined in the file `String.cc`. (Note that while Concurrent C accepts both C and C++ constructions, it requires its input files to use the suffix `.cc`.)

```
#include <string.h>    // strcpy
#include <ctype.h>     // islower and toupper
#include <stdio.h>     // puts and gets
#include "String.h"

String::String()    { str[0] = '\0'; }
String::String(const String& rhs) {strcpy(str, rhs.str);}
String& String::operator = (const String& rhs)
{
   if (this != &rhs)
     strcpy(str, rhs.str);
   return *this;
}

int String::write() { return puts(str); }
int String::read()  { return
        (gets(str) != (char *) NULL) ? 1 : 0; }
void String::allcaps()
{
   for (char *ptr = str; *ptr != '\0'; ptr++)
      if (islower((int) *ptr))
        *ptr = (char) toupper((int) *ptr);
}

int operator != (const String& s, const String& t)
{
   return strcmp(s.str, t.str);
}
```

Processes Accepting Objects as Arguments

The processes `client` and `server` have changed in two ways, primarily. First, transactions now involve objects of type `String`. Second, they both call transactions, each rendezvousing at a point `Buffer`. In the following program, `client.cc`, the transaction `request` reaches process `buffer` via this point:

```
#include "proc_specs.h"

process body client(Buffer)
{
   String s, eof;
   while (s.read())
       Buffer.request(s);
   Buffer.request(eof);
}
```

In the following program, `server.cc`, the transaction `serv` also synchronizes via `Buffer`:

```
#include "proc_specs.h"

process body server(Buffer)
{
   String s, eof;
   while ((s = Buffer.serv()) != eof) {
      s.allcaps();
      s.write();
   }
}
```

The main program `main.cc` tells us, among other things, that `Buffer` is of type process `buffer`:

```
#include "proc_specs.h"
#define MAX 32

main()
{
   process buffer Buffer;
   Buffer = create buffer(MAX);
   create client(Buffer);
   create server(Buffer);
}
```

As the process specifications suggest, processes `client` and `server` do not interact directly: no PIDs are exchanged between them. They interact via another process, `buffer`. What we have called a "rendezvous point" is actually the PID of the created process `buffer`. Because `client` and `server` accept no transactions, they establish no rendezvous points.

Selecting Among Transactions

The new process, `buffer`, allows for greater parallelism between `client` and `server`. The following program, `buffer.cc`, selects among competing transactions from either process `client` or process `server`:

```
#include "proc_specs.h"

process body buffer(full)
{
   String *buf;           // circular buffer
   String eof;            // null String
   int n = 0;             // number of strings in buffer
   int in = 0;            // index of next slot
   int out = 0;           // index of next string
   buf = new String[full];
   for (;;)
      select {
         (n < full):
            accept request(s)
               buf[in] = s;
            (in++) % full;
            n++;
      or
         (n > 0):
            accept serv()
               treturn buf[out];
            if ( buf[out] != eof ) {
               (out++) % full;
               n--;
            } else
               break;
      }
}
```

Process `buffer` accepts transactions from both `client` and `server`. It receives `String` objects from `client`, via the transaction `request`, which it stores in a circular buffer of `String` objects (`buf`). And it receives the transaction `serv` from `server`, through which it returns objects from the same array, so `server` can complete the task of case conversion and printing.

Process `buffer` accepts transactions from multiple processes. To distinguish among them, a `select` statement is used. This statement has two *alternatives*, separated by the keyword `or`. Each alternative begins with a Boolean test called a *guard* identifying conditions under which an `accept` statement can be executed. A `select` statement can have an arbitrary number of alternatives.

Returning Transaction Values

To return a value we need a new keyword: `treturn`. This statement causes the transaction `serv` to return objects of type `String`.

Deadlock

The term *deadlock* describes a state in which the program has not completed, yet it is unable to continue execution. Given the preceding program, we might wonder if the process `buffer` ever completes. The `client` process is able to detect end-of-input through the member function `read`; the `server` process terminates when it receives the object `eof`. Deadlock has been avoided in this program by enabling the process `buffer` to detect `eof` coming out of the buffer `buf` (lines 22–26).

New Directions: Shared Memory Facilities

Synchronous and asynchronous transactions also apply in a shared memory environment. The implementation of Concurrent C in that environment will very likely exploit shared memory resources. In addition, Gehani [6] has proposed an explicit facility for shared memory access: the *capsule*. The capsule blends syntactically and semantically with the C++ data abstraction facility called the *class*.

In that capsules encapsulate shared variables, they are similar to monitors. By contrast to monitors, they satisfy Bloom's [1] criteria for expressiveness of synchronization conditions, support inheritance in the sense that a class does, allow operations to execute in parallel, and allow operations to time out.

Summary

This paper has reviewed some fundamentals of a language that provides an intuitive notation for abstracting and implementing concurrent events. As an intuitive notation, Concurrent C makes a significant contribution in the area of problem-solving. Few real-world problems are characterized by purely sequential phenomena. A well-abstracted notation for describing concurrent events allows the programmer to think more closely in terms of the problem studied, rather than in terms of a general implementation language.

As a language for instantiating concurrent processes, Concurrent C can deliver significant performance gains. In a multi-processor environment, concurrently processable components can be isolated and distributed for parallel processing. In a uni-processor environment, concurrently processable components can experience some parallel processing, as individual, cooperating tasks must wait for resources, etc.

Because Concurrent C is orthogonal to C++ (it derives from the same grammar underlying *cfront*), the modeling facilities supplied by the superset language, Concurrent C/C++, are impressive. Using this language, we can decompose problems both in terms of user-defined events and in terms of user-defined types.

References

[1] Bloom, T. "Evaluating Synchronization Methods." *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM-SIGOPS (September 1979).

[2] Gehani, N. H. and W. D. Roome. "Concurrent C." *Software — Practice and Experience* 16 (September 1986): 821–844.

[3] Gehani, N. H. and W. D. Roome. "Concurrent C++: Concurrent Programming with Class(es)." *Software — Practice and Experience* 18 (December 1988): 1157–1177.

[4] Gehani, N. H. and W. D. Roome. *Concurrent C.* Summit, NJ: Silicon Press, 1989.

[5] Gehani, Narain. "Working in Concurrent C." *UNIX Review* 7 (June, 1989): 60–70.

[6] Gehani, N. H. "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++." Bell Labs, 1990.

[7] Stroustrup, Bjarne. *The C++ Programming Language.* Reading, MA: Addison-Wesley, 1986.

Steven Lally is a Distinguished Member of Technical Staff at AT&T Bell Laboratories. He received his Ph.D. at Johns Hopkins University, where he studied language theory and, in particular, the history of the English language. He is the author of books and papers on both Computer Science and linguistic topics. He can be reached at uunet!att!mtfmi!lally.

∞

16. Extended Multibyte Support

Jim Brodie

Abstract

This article reviews the Japanese Multibyte Support Extension (MSE) proposal. That proposal defines extensions to Standard C for languages requiring more than a single byte to represent a character (e.g., Japanese, Chinese, and Korean). Some background into the current status of the multibyte character support is presented. This is followed by a discussion of the major components of the MSE proposal and some of the rationale for their inclusion.

Normative Addenda and ISO C

It is very likely the approval cycle for the International Standards Organization (ISO) C programming language standard will be complete sometime late in 1990 or early in 1991. At this point, it is expected the ISO standard will be identical (except for various required formatting differences) with the ANSI C standard. However, as the C standard has moved through the international approval process, several issues have arisen. I have discussed in an earlier column (*Volume 1, Number 4*)) the British desire for a Normative Addendum to explicitly call out undefined behavior. The Normative Addendum, which will be developed over the coming months or years, will be balloted and will extend the ISO Standard currently being established. It is not just commentary. **It will have the full force of a standard.**

Now the Japanese have submitted a proposal that will extend the content of the Normative Addendum to include more complete support for languages requiring more than a single byte to represent characters (e.g., Japanese, Chinese, and Korean). Unlike the British proposal, which was aimed at editorial clarification, the Japanese proposal is a request for a large and substantive change to the language. The Japanese's stated goal, however, has been to present the new language support features as a strict extension to the C language and libraries. It is *not* intended to change the meaning of any of the existing features of the language.

In the remainder of this article I will present some background and then look at the specifics of the the Multibyte Support Extension proposal.

[Ed: The Danish alternate trigraph proposal is also part of the normative addendum. At the June 1990 ISO C meeting in London, it was suggested that

the UK, Danish, and Japanese proposals each be in a separate normative addendum. Since they are unrelated, this makes sense and, I suspect, is likely.]

Background

Before we discuss the MSE proposal, it is useful to briefly review the current state of the C Standard relative to multibyte character set support.

Standard C provides a basic level of support for language character sets that require multibyte characters. The support is centered around the locale facilities, the `wchar_t` defined type, support for multibyte characters in C source code (in string literals, character constants, comments, and header file names), and a small collection of functions that support conversions between potential encodings (representations) of multibyte characters.

Standard C recognizes two fundamental approaches to encoding characters that require multiple bytes. The simplest is to have a character set that uses the same number of bytes for each character. For example, you could support a character set that has over 65,000 distinct characters in an encoding that uses two 8-bit bytes per character. Characters that are encoded using these constant, fixed length byte sequences, are called *wide characters*. (A wide character is an atomic entity.) Each wide character representation is unique and can be identified independent of any characters that may have preceded it in a sequence of wide characters. Wide character strings are convenient to work with because you always know where a character starts and stops. You can conveniently perform operations such as splitting or concatenating strings. They, however, have some drawbacks. Perhaps the most notable is that they do not represent a very space-efficient encoding.

An alternative approach is to allow potentially variable-length sequences of bytes to be used to represent characters in the extended character set. This form of encoding may use shift codes so that the same byte value (or sequence of bytes) may represent different characters, depending upon which shift state has been entered using preceding shift codes. Characters encoded using one of these encodings are known in Standard C as *multibyte characters*. (This naming is somewhat confusing, since the constant fixed length wide character encoding is also a multiple byte encoding.) The standard describes multibyte characters and their encodings in the following way:

> "A multibyte character may have a state-dependent encoding, wherein each sequence of multibyte characters begins in an initial shift state and enters other implementation-defined shift states when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state."

For example, characters from the English alphabet may be represented by their ASCII single byte representation. The language encoding may then use multibyte sequences to represent the additional required characters (e.g., the Kanji characters).

Multibyte encodings that use either variable length byte sequences and/or shift states are frequently more compact than the corresponding wide character representation. They, however, bring their own collection of problems. For example, you cannot begin reading in the middle of a string. Without the context from the beginning of the string you do not know what shift state may have been entered.

Standard C supplies a small collection of basic functions for working with and translating between wide character and multibyte encodings for characters. It also supplies a defined type, `wchar_t`, that is the integer type used to hold a single wide character. The primary facility supplied by each of the multibyte character functions is as follows:

`mblen` – returns the number of bytes, from a string, that constitutes the next multibyte character

`mbstowcs` – converts a string from multibyte to wide character encoding.

`mbtowc` – converts a character from multibyte to wide character encoding.

`wcstombs` – converts a string from wide character to multibyte encoding.

`wctomb` – converts a character from wide character to multibyte encoding.

All of the wide character and multibyte functions are under control of the `LC_CTYPE` category of the current locale (which is established by the `setlocale` function).

When this basic support for large character set languages was proposed, the Japanese responded favorably. However, they continued to ask for a more complete package of functions and capabilities. At the time (several years ago), X3J11 argued that a more complete set was inappropriate because no existing practice had established what this set of functions should be.

The MSE Proposal

In the meantime the Japanese have, as you might expect, continued to do a lot of work in this area. They have continued to look for a single common set of facilities and functions so that portability is ensured. They have worked with a number of organizations, including the Japanese UNIX System Advisory Committee to AT&T, Hewlett-Packard, IBM, X/OPEN, the Open Software Foundation, UNIX International, and several User groups (such as Japan UNIX Society and UniForum) to prepare a coordinated proposal based upon current practice and experience. They now believe sufficient existing practice has been

accumulated. The time is now ripe for complete language support for character sets that require multiple bytes. The MSE proposal is this unified proposal. It is a set of generic extensions that can be applied to support *any* language requiring a large character set.

The MSE proposal builds upon the existing functions and capabilities already supplied by Standard C. The proposal adds several major components. It includes a recommendation that the defined type, `wchar_t` be promoted to a basic type (like `char`, `int`, `double`, etc.). It attempts to clarify several parts of Standard C that the Japanese feel are ambiguous or incomplete with regard to multibyte character sets. Finally, it recommends a set of wide-character-oriented functions that parallel many of the current character oriented functions in the standard library.

Let's look at some of the details of the MSE proposal.

The rationale accompanying the MSE proposal argues for making `wchar_t` a full-fledged type, rather than a defined type (which is simply a pseudonym for one of the basic types). They argue several technical points, but the most compelling argument is that C should have fundamental support for a "character" data type that is *not* tied to single byte units. Making a character type that supports multibyte characters as conveniently as single byte characters is part of the full integration and recognition of the multibyte character sets.

The adoption of another basic type has many problems and ramifications throughout the language. For example, promotion rules and rules for what operations can be performed using this new type must be established. This may be somewhat difficult given the fact that there is no obvious "natural" position in the type hierarchy for a `wchar_t` type. X3J11 spent several years working out the subtle ramifications of introducing the `signed char` type into the language. There will undoubtedly be serious concerns as to whether the ISO or ANSI committees can get all of the corresponding language changes completed correctly, within a short period of time.

In the area of clarifications, the MSE proposal adds the clarification that a shift-sequence (used in multibyte character encodings) does *not* have a corresponding wide character encoding. Shift-sequences are used to establish a state so that other characters can be determined.

The bulk of the proposal addresses the need for a set of library functions that support the input, processing, and output of characters from the multibyte character set. It also addresses a wide range of the support issues that arise with the inclusion of the new library functions.

With Standard C, text handling programs that work with multiple character encodings on input and output have little support from the standard library, particularly if the character encoding uses shift states. The program must explicitly interpret and handle shift-sequences to determine the shift state for any text streams. These dependencies make the programs character-encoding-specific and therefore non-portable. Even when dealing with strings, the shift sequences can cause problems. Consider the following example, where .X and .Y represent multibyte characters and SI and SO stand for the Shift In and

Shift Out sequences.

```
char *s1 = ".X.Y";       /* SI .X .Y SO \0 */
char *s2 = ".X"".Y";     /* SI .X SO SI .Y SO \0 */
int result

result = strcmp(s1, s2);
```

The Shift In and Shift Out sequences can be treated as elements of the multibyte character set. Therefore, shift-sequences in the string literal may remain after the string concatenation. This means that, despite the intuitive expectation, the result of the `strcmp` function may be that the strings do not compare equal.

Because of the many problems related to handling the multibyte character encodings (with their variable length characters and shift states) the committee developing the MSE proposal decided to recommend a complete set of functions that support the wide character encodings. The multibyte character encodings are still allowed, they are simply not supported by library functions. This, in essence, will make the wide character encodings the "favored" encoding for most C language operations.

The following lists enumerate the MSE proposal's recommended functions. These functions fall into a number of categories ranging from wide character testing functions to wide string time conversion functions. As a general rule, the functions map very closely to the Standard library's character oriented functions with similar names. For example, `iswalnum` is the wide character version of the `isalnum` function.

- Wide character testing: `iswalnum`, `iswalpha`, `iswcntrl`, `iswdigit`, etc.
- Wide character case mapping: `towlower` and `towupper`.
- Wide character classification: `set_wctype` and `is_wctype`.
- Wide character input/output: `fgetwc`, `fgetws`, `fputwc`, `fputws`, `getwc`, `getwchar`, `getws`, `putwc`, `putwchar`, `putws`, and `ungetwc`.
- Formatted wide character I/O: `vwsprintf`, `wsprintf`, and `wsscanf`.
- String Conversion (all of the string-oriented function names are formed by replacing the `str` with `wcs`): `wcstod`, `wcstol`, and `wcstoul`.
- Wide string handling: `wcscpy`, `wcsncpy`, `wcscat`, `wcsncat`, `wcscoll`, `wcscmp`, `wcsncmp`, and `wcsxfrm`.
- Wide string search: `wcscrt`, `wcscspn`, `wcspbrk`, `wcsrchr`, `wcsspn`, `wcswcw`, `wcstok`, and `wcslen`.
- Wide string time conversion: `wcsftime`.

These functions reflect the libraries that are being developed in implementations that support multibyte character sets. These functions provide a standard programming environment where a programmer can work consistently in a wide character environment, rather than needing to constantly drop back into the byte level operations supplied by functions such as `getc` or `putc`. This is important if programmers who work in multibyte character set environments are going to develop correct code in a highly productive manner.

As noted above, almost all of the wide character functions correspond to existing character oriented functions. The exceptions to this are the wide character classification functions `set_wctype` and `is_wctype`. These functions are used to handle the implementation-defined character classification classes. The proposal describes these functions in the following way:

> The functions `set_wctype` and `is_wctype` classify characters in the same style as the traditional `ctype.h` functions (e.g., `isspace` and `isalpha`), according to the rules of the coded character set defined by character type information in the program's locale.
>
> **The `set_wctype` Function**
>
> **Synopsis**
>
> ```
> wctype_t set_wstype(char *property);
> ```
>
> **Description**
>
> The `set_wctype` function is defined for valid property names as defined in the current locale. `property` is a string identifying a generic character class for which codeset specific type information is required. The function returns a value of type `wctype_t`, which can be used as the second argument to a call of `is_wctype` ...
>
> **The `is_wctype` Function**
>
> **Synopsis**
>
> ```
> int is_wctype(wchar_t wc, wctype_t wc_prop)
> ```
>
> **Description**
>
> The `is_wctype` function determines whether the wide character `wc` has the property `wc_prop`, returning true or false ...

This pair of functions provides an open ended character classification mechanism that can be customized to the classification needs of each language environment. While the complete list of property names is implementation-defined

for each locale, a standard set (`"alnum"`, `"alpha"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"`) is reserved for the standard character classes. For example, the following call would be used to determine if the wide character in the data object associated with the name `c` is a numeric digit.

```
is_wctype(c, set_wctype("digit"));
```

When using the standard property names, these functions duplicate the wide character testing functions (e.g., `iswdigit` and `iswalpha`).

In addition to the collection of new functions, the following functions are modified to support additional format specifiers (`%ws` and `%wc`) that support wide character encoded strings and characters: `printf`, `sprintf`, `fprintf`, `vprintf`, `vsprintf`, `vfprintf`, `scanf`, `sscanf`, and `fscanf`.

There are a variety of issues surrounding the introduction of these wide character format specifiers. An interesting issue that the proposal deals with is the difficulty of controlling how many print positions will be taken up by a character (in the real world some characters may span several printing positions). This becomes very important, for example, when trying to output columns of information in a database or spreadsheet. The question is "When you specify a field width or precision, are you specifying it in terms of `wchar_t` characters, (independent of how many print positions are required for the representation) or single byte character (`char`) print positions?" The answer is "It depends!"

The following table shows that the MSE proposal uses different meanings for the field width and precision numbers, depending on the function and the presence or absence of the `#` flag.

Functions	*Field width*	*Precision*

`printf`
`sprintf`
`fprintf`
`vprintf`
`vsprintf`
`vfprintf`
	byte `char`	byte `char` (as default)
		character `wchar_t` (with the `#` flag)

`scanf`
`sscanf`
`fscanf`
	byte `char`	N/A

`wsprintf`	character `wchar_t`	character `wchar_t` (even if `#` flag is specified)
`wsscanf`	character `wchar_t`	N/A

Consider the case where you want to print out a string of wide characters that contains several characters that require double width printing (they are designated as .A, .B, .X, and .Y). The other characters, a, b, and c, are single width characters. The following five calls to `printf` and `wsprintf`:

```
printf("1234567890123\n") /* gives a reference line */
printf("%13ws\n", L".A.Babc.X.Y");
printf("%13.6ws\n", L".A.Babc.X.Y");
printf("%#13.6ws\n", L".A.Babc.X.Y");
wsprintf("%13.6ws\n", L".A.Babc.X.Y");
```

produce the following output:

```
1234567890123
.A.Babc.X.Y
.A.Bab
.A.Babc.X
.A.Babc.X
```

Despite the large number of functions, the MSE proposal does *not* recommend the introduction of any new headers. Instead, it recommends these new function prototypes be added to the existing headers.

Conclusion

The MSE proposal provides a comprehensive approach to extending the C language's support for languages that have multibyte character sets. It makes a concerted effort to address the wide variety of issues that arise when making this large change to the C language and library.

The proposal will be reviewed by X3J11 as well as by other C users throughout the world[1]. Problems will be addressed and changes will likely be made. However, since there is considerable world-wide support for the "complete" internationalization of the C language, it now appears likely that some variant of this proposal will become a part of the International standard for the C language.

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.

∞

[1]This proposal will be discussed in detail at the ISO C meeting in Copenhagen this November.

17. FORTRAN to C: Character Manipulation

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

Abstract

To translate a source language into a target language, the syntax of the target language must be able to express the semantics of the source language. This paper discusses the major mismatch between FORTRAN semantics and C syntax which makes a clean translation of FORTRAN to C difficult—that of character manipulation. From the standpoint of FORTRAN, the problem is that there is no real unified approach to character manipulation. With C, the problem is that the C null-terminated string construct is useless insofar as the semantics of FORTRAN are concerned.

Introduction

In Goodman [1], 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. Using this design, two problems must be solved. First, the resultant C program, once it is compiled and linked, must behave correctly. Second, the program must be readable and maintainable in its C form. A translation from FORTRAN to C is referred to as 'correct' if the final C program when compiled and executed on its platform produces the identical results as the FORTRAN original would produce on its platform. Such a translation is referred to as 'clean' if the final C program can be read and maintained as easily by a C programmer as the FORTRAN original would be by a FORTRAN programmer.

In Goodman [2], various numerical issues associated with ensuring the correctness of the C translations are discussed. In this paper, FORTRAN character manipulation is studied. The conclusion drawn is that cleanliness must sometimes be sacrificed to achieve correctness. Though all FORTRAN codes can be correctly translated into C, not all FORTRAN features can be translated cleanly. The alternative of producing an incomplete or, even worse, incorrect translation is completely unacceptable. Fortunately, given prior knowledge of the techniques used in the program, the user may select a cleaner alternative;

however, that selection may cause a syntax error either from the translator or from the C compiler.

FORTRAN Character Manipulation

By far the most difficult task to be faced by the translator is that of FORTRAN character manipulation. Despite the efforts of the various standards committees, the problem is that FORTRAN as a living computer language has no unified approach to character manipulation.

FORTRAN was originally designed to do 'formula translation.' It had little use for characters other than to label the results of calculations. FORTRAN 66 has no `CHARACTER` data type and has no character manipulation statements other than formatted I/O operations. Character data is simply hidden in whatever variable is convenient. Once a programmer stores character data in a numeric variable, it is up to him to ensure that he does not use that variable for anything else.

As the language matured, a `CHARACTER` type was added to the language and some minimal machinery (such as initialization, assignment, concatenation, and I/O) was added to manipulate these types of variables. However, since there were many FORTRAN 66 programs still in existence, the use of the new `CHARACTER` type did not preclude using the old character management techniques as well. As the final blow, programs were then written which mixed the old and new machinery. Things such as equivalencing numeric variables and character variables were completely valid and are still in common practice.

When the FORTRAN 77 standard was established, the committee rightly concluded that the way in which these FORTRAN dialects had implemented characters was clearly not machine transportable and was unacceptable. Therefore, they made all FORTRAN 66 techniques of hiding character data in numeric variables illegal. In addition, they placed several restrictions on the placement and handling of character variables, all intended to improve the portability of the language. The standards committee, however, had totally ignored the problem of what to do with existing programs. An aside here is that we first attempted to write a translator which would take older dialects of FORTRAN to the 77 standard. This effort proved impossible. The languages are simply incompatible. FORTRAN 77 does not have the openness of C.

After the FORTRAN 77 standard came out, however, no compiler vendor felt that he could afford to lose his customers by telling them that they had to convert all of their FORTRAN programs into the new standard. Therefore, all compiler vendors put features into their compilers that allowed existing programs to still operate. Since there was no standard mixed FORTRAN, all the vendors created their own extensions. Still today, as new compilers are created, even for the PC market, they differ in how they deal with this problem of FORTRAN 66 versus FORTRAN 77.

The problem is now compounded in that particular programs are being writ-

ten not for either standard but rather for the incompatible hybrid combinations. Every request for information about PROMULA.FORTRAN asks if we support VAX FORTRAN, or Prime FORTRAN, etc.

Dealing With FORTRAN Conventions

In designing PROMULA.FORTRAN we decided that mixed FORTRAN dialects were the major barrier to using existing FORTRAN programs in a new machine environment. Our treatment of `CHARACTER` data attempts to deal with all of the dialects with which we are familiar. The basic FORTRAN programming conventions, all of which violate the 77 FORTRAN standard, that had to be dealt with were as follows:

1. Via `DATA` statements, noncharacter variables of any type may be initialized with character data.

2. Via `READ` and `WRITE` statements, character data may be stored and transmitted into or out of noncharacter variables of any type.

3. Character and noncharacter variables may be mixed in `COMMON` blocks or across subroutine parameters.

4. The length of a character variable may be mixed arbitrarily with its structure. That is, a `CHARACTER*20` has the same structure as 20 `CHARACTER*1` or 5 `CHARACTER*4` or 2 by 2 `CHARACTER*5`.

5. Character and noncharacter variables and arrays may be freely equivalenced and must have the same structure. That is, an array of `REAL*8` variables and an array of `CHARACTER*8` with the same structure require the same amount of memory and correspond exactly on an element by element basis.

6. Equality and inequality tests between noncharacter variables and character constants must produce the expected result, with the assumption that the noncharacter variable contains character information.

7. A FORTRAN subprogram can define any of its arguments as being variable length character strings. It is up to the calling routine to specify the actual length of the character strings which it is passing.

The Translation Approach

A character variable is treated simply as a sequence of `char`s. It has no other structure. `CHARACTER*4` is identical to `CHARACTER*2(2)` or `CHARACTER*1(4)` or a sequence of characters hidden in an `INTEGER*4`. The elements of a `CHARACTER`

array are stored one after another with no intervening null characters or character counts or intermediate character pointers. The C notion of a string as being a null-terminated sequence of `char` is useless. Consequently, the C character concatenation and string comparison functions, which depend on the null-byte for their termination condition, cannot be used. This storage of character data is simple and allows the easy simulation of the various tricks allowed by the various dialects.

Unfortunately this simple approach produces dirty translations, as the following sections will show.

Initializing `CHARACTER` Values

The first example shows how character variables are initialized. It presents a `CHARACTER` type declaration with associated initializations.

```
      SUBROUTINE DEMO
      CHARACTER*10 NAME(4:8)/'Charles','Frederick'
     +  ,'Andrew',2*'Mary'/
      RETURN
      END
```

The default translation produced looks as follows:

```
void demo()
{
static char name[50] = {
        'C','h','a','r','l','e','s',' ',' ',' ',
        'F','r','e','d','e','r','i','c','k',' ',
        'A','n','d','r','e','w',' ',' ',' ',' ',
        'M','a','r','y',' ',' ',' ',' ',' ',' ',
        'M','a','r','y',' ',' ',' ',' ',' ',' '
};
        return;
}
```

Note first that in the C output `name` is now a singly dimensioned array whose size, 50, is 5 times 10. The first value is 5 since the FORTRAN definition specifies 5 entries; $8 - 4 + 1$. The second value is 10 since each character string contains 10 characters. The user has the option to show `name` as a two dimensional array. This produces the following alternative translation:

```
void demo()
{
static char name[5][10] = {
        'C','h','a','r','l','e','s',' ',' ',' ',
        'F','r','e','d','e','r','i','c','k',' ',
        'A','n','d','r','e','w',' ',' ',' ',' ',
        'M','a','r','y',' ',' ',' ',' ',' ',' ',
        'M','a','r','y',' ',' ',' ',' ',' ',' '
};
        return;
}
```

This translation looks a bit better, but still does not solve the real problem, which derives from one of the quirks of C. There is no way to specify a sequence of characters without using cumbersome notation, because the notation

```
"Charles   "
```

would generate a null-terminated string and the notation

```
'Charles   '
```

is not accepted by most compilers, even though it seems completely clear and unambiguous. *[Ed: Note though that in* `char c[3] = "abc";`*, the null character is* not *stored in the array.]*

Next, note that the definition of 'Mary' must be repeated twice, since C has no equivalent of initializer repeat counts. Note also that FORTRAN character strings are always padded with blanks and not null characters as is the convention in C. It is this final fact that makes C and FORTRAN character manipulation incompatible.

The case involving the initialization of character variables with character constants is simple by comparison to the problem of initializing noncharacter variables with character constants. The following FORTRAN fragment shows a common use of a double precision variable.

```
 SUBROUTINE DEMO
 DOUBLE PRECISION NAME(4:8)/'Charles','Frederic'
+  ,'Andrew',2*'Mary'/
 DOUBLE PRECISION WAGE(4:8)/48900.0,51300.0
+  ,37890.0,2*61230.0/
 RETURN
 END
```

In this example there are two double precision arrays, one of which contains the first name of each employee and the other their annual wage. The semantics of the two initializations are identical. The values are stored by the FORTRAN compiler in the initial data space so that they are available when the program

begins execution. One can say that the above is bad programming style and nontransportable. The FORTRAN 77 standards committee did that, but there are still many many lines of code being written right now that use logic such as the above. It is a widely accepted FORTRAN convention.

There is no possible direct translation into C which captures the semantics of the above FORTRAN code. We have tried many tricks, including the use of hexadecimals, to initialize noncharacter variables with character data. None have succeeded. The following is the default translation for the above:

```
void demo()
{
static double name[5];
static double wage[5] = {
  48900.0,51300.0,37890.0,61230.0,61230.0
};
static int ftnsiz[] = {1,1,5};
static namelist DATAVAR[] = {
  "name",name,2,ftnsiz
};
static char *DATAVAL[] = {
  "$DATAVAR",
  "name='Charles ','Frederic','Andrew  ',2*'Mary    ',",
  "$END"
};
static FIRST = 1;

        if(FIRST) {
                FIRST=0;
                fiointu(DATAVAL,0,1);
                fiornl(DATAVAR,1);
        }
        return;
}
```

The initialization of the `wage` variable is quite natural and clean. The 2*61230.0 is lost, but at least the semantics of the initialization are the same since no runtime device is needed to get the values into the variable. The machinery introduced to initialize the `name` array is based on two other FORTRAN features: internal files and namelist. In FORTRAN, groups of variables may be read and/or written as groups. The syntax used is free-form and looks almost identical to that used by the `DATA` statement. Internal files are character arrays (yes, noncharacter arrays are also used) whose members are treated as records in a formatted file. The device used to initialize the `name` array is as follows:

1. Create a namelist definition for the variables requiring runtime initialization. The `ftnsiz` and `DATAVAR` arrays contain this definition.

2. Create an internal file which contains the actual initialization values in the namelist format. The `DATAVAL` array is this internal file. Note that the actual syntax of the initialization record is almost identical to that of the original `DATA` statement. Even the repeat count syntax `2*'Mary'` may be used.

3. Create a static local variable `FIRST` which ensures that the initialization occurs only once, when the subprogram is initially called.

4. Add the logic to the C translation to actually use the runtime internal file facilities, `fiointu`, and the read namelist facilities, `fiornl`, to perform the required initialization.

This device does work. Using the syntax of C, we are not able to capture the precise semantics of the FORTRAN original. In the above, `name` and `wage` are not initialized at the same time or in the same manner; however, they are semantically equivalent to the FORTRAN intent.

Variable Length Strings

If FORTRAN had no subprograms, there would be no problems beyond those discussed above. Unfortunately, FORTRAN subprograms allow for variable length character strings, and the various system operations—such as string concatenation, string comparison, string assignment, and string input/output—allow for the total mixture of character strings of any length.

To deal with a variable length character string, the length of that string must be known. In C, the storage technique itself makes this information available. In FORTRAN, since the storage technique for character variables does not have this information, it must be kept separately. This is an especially difficult problem, since the length of a given character string is not defined globally, but by the context of its use (see condition 4 above).

The solution is difficult to accept, but appears to be dictated by the needed generality of the translation. All references to character variables in the FORTRAN program are translated into an ordered pair of values—a pointer to the start of the characters being manipulated and an integer value which defines the length of the character string at that point in the code. Similar solutions are used by other FORTRAN compilers as well.

Prime FORTRAN [3], makes the following statement about the number of arguments that may be passed to a subprogram.

> "F77 allows up to 254 arguments to be passed to and from subroutines. There are two exceptions to these limits: If all the arguments are of type `CHARACTER`, F77 only allows 127 arguments. If the arguments are of mixed data types (`CHARACTER` and other types), the maximum number of arguments is between 127 and 254."

Obviously the Prime compiler is passing two things for each character parameter.

VAX FORTRAN [4], in describing the default argument types for the different data types assigns `REF` to all types except character which are assigned a `DESCR` type. The `REF` attribute implies a call by reference, while the `DESCR` attribute causes a string descriptor structure to be built which contains a pointer and information about the size and type of the element being pointed to. This descriptor is then passed by reference.

To show the effect of this solution on the look of translations, consider the following simple example.

```
SUBROUTINE DEMO
CHARACTER*13 ALPHA/'     32      '/
I = IVALUE('123456')
J = IVALUE(1H6)
K = IVALUE(ALPHA)
END

FUNCTION IVALUE(SVAL)
CHARACTER*(*) SVAL
READ(SVAL,'(BN,I5)') IVALUE
END
```

In this example, a function `IVALUE` is being used to decode an integer value from a variable length character variable. Note that the FORTRAN function `IVALUE` has only one parameter. The default translation for the above looks as follows:

```
void demo()
{
extern long ivalue();
static char alpha[13] = {
    ' ',' ',' ',' ',' ','3','2',' ',' ',' ',' ',' ',' ',' '
};
static long i,j,k;

        i = ivalue("123456",6);
        j = ivalue("6",1);
        k = ivalue(alpha,13);
}
```

```
long ivalue(sval,P1)
int P1;
char *sval;
{
static long ivalue;

        ftnread(INTERNAL,sval,P1,VFMT,"(BN,I5)",
                INT4,&ivalue,0);
        return ivalue;
}
```

In the translation, the function `ivalue` now has two parameters. The first is the character string and the second is the length of that string. It is this length, `P1`, which is then passed to the library function `ftnread` which performs the actual decoding operation. The addition of this extra parameter obviously degrades the cleanliness of the C output.

There are, of course, many FORTRAN codes that do not make explicit use of the variable length character string. The following is a variant of the example above.

```
SUBROUTINE DEMO
CHARACTER*13 ALPHA/'     32      '/
I = IVALUE('123456')
J = IVALUE(5H    6)
K = IVALUE(ALPHA)
END

FUNCTION IVALUE(SVAL)
CHARACTER*5 SVAL
READ(SVAL,'(BN,I5)') IVALUE
END
```

In this example, the parameter `sval` is declared as a `CHARACTER*5`. It does not matter that some of the actual parameters are longer than 5, but none may be shorter. Thus, the second call was changed to a '`5H    6`'. There is a command-line switch that tells the translator that variable length strings are not needed. The use of this switch produces the following translation.

```
void demo()
{
extern long ivalue();
static char alpha[13] = {
    ' ',' ',' ',' ',' ','3','2',' ',' ',' ',' ',' ',' '
};
static long i,j,k;
```

```
        i = ivalue("123456");
        j = ivalue("    6");
        k = ivalue(alpha);
}
long ivalue(sval)
char *sval;
{
static long ivalue;

        ftnread(INTERNAL,sval,5,VFMT,"(BN,I5)",
                INT4,&ivalue,0);
        return ivalue;
}
```

Except for the initialization of `alpha`, which was discussed earlier, this translation is clean. Unfortunately, if this switch is used with the earlier version, the following is produced:

```
SUBROUTINE DEMO
CHARACTER*13 ALPHA/'     32      '/
I = IVALUE('123456')
J = IVALUE(1H6)
K = IVALUE(ALPHA)
END

FUNCTION IVALUE(SVAL)
CHARACTER*(*) SVAL
READ(SVAL,'(BN,I5)') IVALUE
```

> FORTRAN Syntax Error Number 284 occurred on input record number 9 of file *demo.for*. Explanation: A string's variable length is being used in a raw string context. Symbol = 'SVAL', Column = 21

The processor waits as long as it can, but ultimately it must pass the length of the `sval` parameter to the `read` function. This length is not available and a syntax error must be generated.

This is a primary design goal of PROMULA.FORTRAN. If the user makes a command line selection to improve the cleanliness of the C output and if that selection makes a correct translation impossible, then a clear syntax error is issued. The user does not have to debug the translated program.

Conclusion

We must strive to achieve correctness and cleanliness when doing translation. When the syntax of the target language blocks this dual goal, then cleanliness must be sacrificed for the sake of correctness. The early implementations of FORTRAN had a very limited view of character manipulation. This is something which all FORTRAN implementations must deal with, even translators.

References

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

[2] Goodman, Fred "FORTRAN to C: Numerical Issues", *The Journal of C Language Translation*, Vol 2, No 1.

[3] Hasse, Camilla and Jerry Ornstein, "FORTRAN 77 Reference Guide Fifth Edition", Prime Computer Inc, 1988, Page 8–12.

[4] "Programming in VAX FORTRAN, V4.0", Digital Equipment Corporation, 1984, Table 10-1.

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

∞

18. Miscellanea

compiled by **Rex Jaeschke**

Extensions

Assertions in UNIX System V

[Ed: This section was contributed by David Prosser of AT&T.]

UNIX System V Release 4 provides a new preprocessing capability called assertions. The intent of assertions is to have a completely separate name space from macros for the typical environmental queries made by programs. Historically, the preprocessor would have names such as `unix` and `vax` predefined as macros that could surprisingly "rewrite" your program. Other vendors seem to have taken the straightforward approach of adding __-prefixed versions of these macros as their ANSI C conforming approach. This only compounds the problem since now there are twice as many predefined macros.

In contrast, the assertion mechanism allows for a separate pool of names as well as the extra dimension of predicates that can be associated with multiple token sequences.

Syntax

```
# assert   identifier
# unassert identifier
# assert   identifier ( pp-tokens )
# unassert identifier ( pp-tokens )
```

Semantics

The first directive causes the identifier to be known as an assertion predicate (if it is not one already). The second causes the identifier no longer to be known as such. (No diagnostic is produced if it wasn't already one.)

The third directive behaves as the first, and also associates the given token sequence with the predicate. (The token sequence is handled just like a replacement list—the existence of white space in the midst of the pp-tokens is significant.) There can be any number of token sequences associated with a given predicate. The fourth directive disassociates a given token sequence (if any such existed) from the identifier.

Predicates can be tested only with `#if` or `#elif` directives. The expression syntax for these directives is extended to include unary expressions of the form

`#` *identifier* `(` *pp-tokens* `)`

that evaluate to 1 if the identifier is a predicate associated with the given token sequence; otherwise 0.

The use of the `#` causes predicate testing to be a pure extension since it does not interfere with any existing expressions. Of course, a conforming implementation must produce at least one diagnostic message if one or more of these constructs is found.

The set of preasserted predicates varies according to the target machine and operating system. Some of the possibilities are:

```
#assert machine(u3b2)
#assert system(unix)
#assert cpu(M32)
#assert cpu(i386)
```

Assertions can be specified on the cc(1) command line. The −A option has the following syntax:

`-A-`	forget all predefined macros and assertions
`-A` *name*	same as `#assert` *name*
`-A` *name*(*pp-tokens*)	same as `#assert` *name*(*pp-tokens*)

White space on the command line is handled the same as in a directive, but the subject of the −A option must be one argument (quoted as needed to get around special characters recognized by your shell).

Examples

If you have code that is dependent on the target operating system, you might use assertions as follows:

```
#if #system(unix)
        #define SEPCHAR '/'
#elif #system(MS-DOS)
        #define SEPCHAR '\\'
#else
        #error Cannot guess pathname separation character
#endif
```

A means to specify byte order might be handled in a crude manner as follows:

```
#if #cpu(vax) || #cpu(i386)
        #assert byte_order(little-endian)
#else
        #assert byte_order(big-endian)
#endif
```

```
#if #byte_order(big-endian)
        /*...*/
#endif
```

Pointers in a Segmented World

The Intel 80x86 architecture is, perhaps, the single most widely used C development platform. And it can also be one of the most hostile, particularly if the size of your program approaches 640KB.

In order to be somewhat backwards compatible with their 8-bit product line, Intel designed the 8086 chip memory to be organized into segments each of 64KB and to use segment registers to access multiple segments. Registers were made 16 bits. This design decision was later seen to have several shortcomings although, at the time, few people would have projected the enormous popularity the chip (and its successors) now enjoys.

Basically, an executable program consists of pieces of code and data each of which belongs to a particular 64KB segment. Essentially, there are four primary ways to construct a program:

8086 Segment Structure	
Total Code Size	*Total Data Size*
64KB	64KB
1MB	64KB
64KB	1MB
1MB	1MB

These groupings are called memory models. When you compile a program using one of these models the compiler generates 16-bit (near) or 32-bit (far) addressing as required. The more interesting situation is where a programmer wishes to mix 16- and 32-bit code (or data) pointers in the same program. In this case, they need to explicitly declare near and far pointers using the keywords `near` and `far`, respectively. For example:

```
#include <stdio.h>

char c1 = 'A';
char near *pnc = &c1;
char c2 = 'Z';
char far *pfc = &c2;
main()
{
        printf("pnc:  %Np, %c\n", pnc, *pnc);
        printf("sizeof(pnc) = %u\n\n", sizeof(pnc));
        printf("pfc:  %Fp, %c\n", pfc, *pfc);
        printf("sizeof(pfc) = %u\n", sizeof(pfc));
}
```

```
pnc:  0085, A
sizeof(pnc) = 2

pfc:  1544:0088, Z
sizeof(pfc) = 4
```

Note that the `printf` edit masks have been extended to handle the different pointer sizes. (If `near` and `far` are not used, the compiler uses defaults depending on the command-line options specified.)

Normally, each function and object must be completely contained in a 64KB segment. Therefore, address arithmetic on an object need not be concerned with the segment's base address. The address of each element (or label) has the same base. To support objects (but not functions) larger than 64KB, the `huge` keyword was added. As a result, far pointers were no longer adequate and huge pointers were provided. A huge pointer is much like a far pointer except that arithmetic on huge pointers is normalized to handle segment base differences.

The keywords `near`, `far`, and `huge` are supported by all mainstream C compilers running under DOS on the Intel architecture. In its recent release (V6.0), Microsoft has also added the alternative keywords `_near`, `_far`, and `_huge`, whose spelling conforms to ANSI's requirement for new keywords.

Apparently, all this was not complicated enough, so various vendors invented yet another addressing scheme. Programs having more than 64KB of code or data require far pointers. Since these are twice the size as near pointers and require more instructions to manipulate, they are less efficient than near pointers. The solution was to have special near pointers whose base address was not fixed by the compiler but, instead, is specified by the programmer. As such, if you wished to access many objects, all of which resided in the same data segment, you only had to load the segment base register once and then use near pointers (with offsets only).

The TopSpeed compiler from Jensen and Partners provided this mechanism using what they call *relative pointers*. Such pointers are specified as follows:

```
unsigned int segment_base = 0xB800;

char <segment_base> *ptr;
```

At runtime, `segment_base` can be changed causing all subsequent references to `ptr` to be considered offsets from that new base address. When used correctly, relative pointers can significantly reduce code size and improve performance. TopSpeed supports the use of relative pointers with function pointers as well, although this is of less use since arithmetic operations are not permitted with these pointers.

Microsoft's V6.0 compiler also supports this notion but calls them *based pointers*. They use a different approach, which includes four new keywords and an operator. Here's how they do it:

```
int _based(_segname("_DATA")) i = 10;
int _based(_segname("_DATA")) *pi = &i;
```

The keyword `_based` has the general form `_based(...)`. In this case its argument is one of the four standard segment names. Using this approach, you can control which segment data is placed in. The `_segname` keyword is used to specify the segment name.

```
_segment base = 0XB800;
char _based(base) *pc;
```

In this case, `pc` is based on the segment variable `base`. `base` can be changed at runtime. `_segment` is another new keyword.

```
double *pd;
double _based(pd) *pd2;
```

Here, `pd2` is based on another pointer, `pd`.

```
_segment base = 0XB800;
int _based(void) *px;

main()
{
        int i;

        i = *base:>px;
}
```

In this case, we have a `void` base address. As such, `px` is a relative offset that can be combined with a base address at any time using the new operator `:>`. (The precedence of this operator is below the postfix operators but above the unary operators. It associates left to right.)

```
char _based((_segment)_self) *ps;
```

Finally, we have a way to make a pointer contain an offset based on its own base address. The utility of this might not be obvious, but it can be useful when used with forward and backward pointers in a linked list.

How much machinery must we invent to compensate for a "deficient" architecture? Thus far, it appears the answer is, "Lots." Fortunately, the 80386 and 80486 machines are not limited to 64KB segments. Compilers that take full advantage of these processors can treat the address space as being flat. However, these compilers need special support system software. At this time, they account for a very small fraction of installations.

Come to think of it, I don't remember having all this trouble programming the PDP-11! Now there is a *real* machine.

f2c: A FORTRAN to C Converter

[Ed: This article was recently published in SIAM News *and is reprinted here with the kind permission of the authors S. I. Feldman (Bellcore), D. M. Gay (AT&T Bell Laboratories), M. W. Maimone (Carnegie-Mellon University), and N. L. Schryer (AT&T Bell Laboratories).]*

We have produced a program that automatically converts ANSI standard FORTRAN 77 [1] to C [8]. It has converted many FORTRAN programs without manual intervention. It is easily available—free of charge (and of warranty)—by electronic mail and *ftp*.

Automatic conversion of FORTRAN 77 is desirable for several reasons. Sometimes it is useful to run a well-tested FORTRAN program on a machine that has a C compiler but no FORTRAN compiler. At other times, it is convenient to mix C and FORTRAN. Some things are impossible to express in FORTRAN 77 or are harder to express in FORTRAN than in C (e.g., storage management, some character operations, arrays of functions, heterogeneous data structures, and calls that depend on the operating system), and some programmers simply prefer C to FORTRAN. There is a large body of well tested FORTRAN source code for carrying out a wide variety of useful calculations. It is sometimes desirable to exploit some of this FORTRAN source in a C environment. Many vendors of computing platforms and operating systems provide some way of mixing C and FORTRAN, but the details vary from system to system. Automatic FORTRAN to C conversion lets one create a *portable* C program that exploits FORTRAN source code.

A side benefit of automatic FORTRAN 77 to C conversion is that it allows tools like *lint* [3] to provide FORTRAN 77 programs with some of the consistency and portability checks that the Pfort Verifer [10] made available to FORTRAN 66 programs.

Starting from Feldman's original *f*77 compiler [5] (the original UNIX FORTRAN 77 compiler), we created a program called *f2c* that converts FORTRAN 77 source code into C. Our report [4] describes *f2c*'s conversions in considerable detail. The purpose of this note is to point out the availability both of our report and of the source for *f2c*. On request, we are happy to send out paper copies of the report. You can also obtain a PostScript version of it by electronic mail by sending the message `send f2c.ps from f2c` to *netlib@research.att.com*. Source for both *f2c* and the support libraries assumed by the C it produces is available by electronic mail and by *ftp*. For details, send the electronic-mail message `send index from f2c` to *netlib@research.att.com*. You can also *ftp* to *research.att.com*. In subdirectory `dist/f2c` you will find a copy of the `index` file and of source for *f2c* and its support libraries. (As of this writing, August 1st 1990, 520 different people have obtained *f2c* source from *netlib*, i.e., by electronic mail, and 1093 have obtained it by *ftp*.)

We have used *f2c* to convert various large programs and subroutine libraries to C automatically (with no manual intervention). These include the

PORT3 subroutine library [2] (PORT1 is described in [6, 7]), MINOS [9], and Schryer's floating-point test [11]. In addition to our own testing, we have been helped by feedback from many people and by observing *f2c*'s behavior on some 21 megabytes of FORTRAN submitted to *netlib*'s experimental "`execute f2c`" service. This feedback and our observations have led us to isolate and repair numerous bugs and to extend *f2c* in various ways.

Although we tried to make *f2c*'s output reasonably readable, our goal of strict compatibility with *f*77 implies some nasty looking conversions. I/O statements, in particular, generally get expanded into a series of calls on routines in *libI77* (*f*77's I/O library), and complex arithmetic often results in messy C. Thus the output of *f2c* may be difficult to maintain as C. It may be more sensible to maintain the original FORTRAN, translating it anew each time it changes. Some commercial vendors of conversion services, e.g., those listed in an appendix to our report, seek to perform translations yielding C that one might reasonably maintain directly. In general, these translations require some manual intervention.

f2c optionally emits special files called *prototype* files that summarize the calling sequences of the translated subprograms. A side benefit of *f2c* is that it can read prototype files and check the consistency of calling sequences across files. We have found this to be a useful debugging aid. Use of prototypes is slightly more convenient than invoking *lint*, but *lint* warns of other possible errors, such as variables that may be referenced before they are set.

References

[1] *American National Standard Programming Language FORTRAN*, American National Standards Institute, New York, NY, 1978. ANSI X3.9-1978.

[2] *The* PORT *Mathematical Subroutine Library*, AT&T Bell Laboratories, Murray Hill, NJ, 1984. Third edition.

[3] *UNIX Time Sharing System Programmer's Manual*, AT&T Bell Laboratories, 1990. Tenth Edition, Volume 1.

[4] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "A FORTRAN-to-C Converter," Computing Science Tech. Report No. 149 (1990), AT&T Bell Laboratories, Murray Hill, NJ.

[5] S. I. Feldman and P. J. Weinberger, "A Portable FORTRAN 77 Compiler," in *Unix Programmer's* Manual, Volume II, Holt, Rinehart and Winston (1983).

[6] P. A. Fox, A. D. Hall, and N. L. Schryer, "Algorithm 528: Framework for a Portable Library," *ACM Trans. Math. Software* **4** (June 1978), pp. 177–188.

[7] P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," *ACM Trans. Math. Software* **4** (June 1978), pp. 104–126.

[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1988. Second Edition.

[9] B. A. Murtagh and M. A. Saunders, "MINOS 5.1 User's Guide," Technical Report SOL 83-20R (1987), Systems Optimization Laboratory, Stanford University, Stanford, CA.

[10] B. G. Ryder, "The PFORT Verifier," *Software Practice and Experience* **4** (1974), pp. 359–377.

[11] N. L. Schryer, "A Test of a Computer's Floating-Point Arithmetic Unit," in *Sources and Development of Mathematical Software*, ed. W. Cowell, Prentice-Hall (1981).

Reader Feedback

Ed: The following letter is a response to Jim Brodie's discussion of an X3J11 interpretation, in his Volume 2, number 1 *column.*

I was disappointed by the one-sided presentation of the issue in my letter to X3J11.

The Standard's grammar for a pre-processing number is:

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

A simple correct grammar for this purpose is:

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

pp-decimal:
digit
`.` *digit*
pp-decimal digit
pp-decimal `.`

where:

digit	is `0--9`
nondigit	is `_`, `A`–`Z`, or `a`–`z`
sign	is `+` or `-`

A correct grammar only allows the [`eE`][`+-`] sequence when there has not been a preceding nondigit in the sequence, thus making a simple distinction between probable hexadecimal constants and probable floating point constants.

The arguments that were published against using a correct grammar, and my replies, are summarized as follows:

1. Garbage sequences are not intended to become valid C tokens due to pre-processing.

   My correction to the grammar has nothing to do with turning garbage sequences like `2.mac` into legal C by such preprocessing as `#define mac e+03` and other such nasties that can be constructed.

2. Each pair of adjacent tokens that are both keywords, identifiers, and/or constants must be separated by white space.

   This quote is superfluous to the point at issue. In the examples which ANSI C does not allow (for example, `0xEE+23` there is always an operator token (a `+` or `-`) between the two numbers or number and identifier.

3. The character sequence space is reserved for future expansion.

   The correct grammar also reserves character sequences for future expansion. The two reserved sets of character sequences are identical except for character sequences which contain `e+` or `e-`, in which case the revised grammar offered above will only accept one instance of the exponent escape and then only if it has not been preceded by a nondigit. This, for example, corrects the processing of `0xEE+value-macro`, excludes `0b011011e+23`, and makes no difference to `0b011011`.

4. White space was already found to be required to control tokenization, so what harm is another situation?

   This is self-justification.

Sincerely,

Terence Carroll, Independent Consultant, Kirchenstrasse 28, D-8000 Munich 80, West Germany.

New Compiler Construction Tools Available

Ed: The following material was taken from the comp.compilers conference.

OMA

OMA is an object oriented system for constructing pattern matchers, or lexical analysers. The system is object oriented in both its design and implementation, and in the programs it generates. OMA input is similar to the input used by the *lex* lexical analyzer system. Unlike *lex*, OMA produces a class description for the pattern matcher. This class description can be specialized or altered by the programmer in a number of ways to achieve unique effects. For example, multiple lexical analyzers can be included in a single system.

OMA is part of a suite of object-oriented tools for compiler construction. Other tools in the collection include the OPA parser generator, the AWESOME symbol table system, and the OCHER code generation routines.

OMA is written in C++ and has been compiled under GNU C++.

Source code and manual can be *ftp*'ed from *cs.orst.edu*, file

pub/budd/oma/oma.tar.Z

OPA and the other tools are still under development.

Tim Budd

Purdue Compiler-Construction Tool Set

The Purdue Compiler-Construction Tool Set (PCCTS) is a set of public domain software tools designed to facilitate the implementation of compilers and other translation systems. The tool set is continuously being upgraded and extended, hence the current release—which is merely a Beta release—only includes those portions which we believe are reasonably stable.

PCCTS is functionally similar to the combination of *lex* and *yacc*, although not interchangeable with them. The primary advantages of PCCTS are:

- Lexer and parser are both generated from a single specification.
- Has extended BNF grammar notation, allowing complex subrules.
- Has powerful attribute handling, both upward and downward inheritance.
- Generates pure C code, implementing fast LL(1) parsers.
- Has documentation/maintenance features (e.g., print grammar w/o actions).
- Sample grammars (e.g., Pascal) are included with the system.

The tools (written in relatively portable C) and several examples are available via anonymous *ftp* from the carp/PCCTS directory on en.ecn.purdue.edu. Documentation is not currently available on-line, but is available free of charge as Purdue University School of Electrical Engineering Technical Report TR-EE 90-14, "Purdue Compiler-Construction Tool Set." Contact: Prof. Hank Dietz, School of Electrical Engineering, Purdue University, West Lafayette, IN 47907, *hankd@ecn.purdue.edu*, (317) 494 3357.

PCCTS is the creation of Terence Parr, Hank Dietz, and Will Cohen. The development of some portions of the system was partially supported by NSF. The authors and Purdue University provide this software on a strictly *as is* basis, without warranty or liability. Despite this, bug reports and fixes are welcome.

Calendar of Events

- 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) 683-5818 or *tam@cray.com*.

- October 8–10, **Frontiers of Massively Parallel Computation** – Location: University of Maryland, College Park, MD (greater Washington D.C.). Call Prof. Joseph JaJa on (301) 454-1808 for more information.

- November 12–16, **ANSI C++ X3J16 Meeting** – Location: Cupertino, CA. For more information, contact the Vice-Chair William M. (Mike) Mille, P.O. Box 366, Sudbury, MA 01776-0003, (508) 443-7433. Email: *wmmiller@hplabs.HP.com*.

- November 16, **Supercomputing '90** – Location: New York City Hilton. Tom MacDonald of Cray Research, Inc. (and NCEG) will present a tutorial on 'Numeric and Scientific C Programs' from 1–5 pm. For more information contact Tom at (612) 683-5818 or *tam@cray.com*.

- November 26–27, **ISO C SC22/WG14 Meeting** – Location: Copenhagen, Denmark. Contact the US International Rep. Rex Jaeschke at (703) 860-0091 or *uunet!aussie!rex* or the convenor P.J. Plauger at *uunet!plauger!pjp* for information.

- December 3–5, **International Workshop on Compilers for Parallel Computers** – Location: Paris, France. This workshop is a follow-up to the *Workshop on Compiling Techniques and Compiler Construction for Parallel Computers*, held at Oxford in September 1989. Contact: Compilers for Parallel Computers, CAII, Ecole des Mines de Paris, F77305 FONTAINEBLEAU Cedex, FRANCE. Telephone: +33 1 64.69.47.08; Telex: MINEFON 694736 F; Email: *workshop@ensmp.fr*

- March 4–5, 1991 **ANSI C X3J11 Meeting** – Location: Norwood, Mass.

- March 6–7, 1991 **Numerical C Extensions Group (NCEG) Meeting** – Location: Norwood, Mass.

- March 11–15, 1991 **ANSI C++ X3J16 Meeting** – Location: not yet established.

- June 17–21, 1991 **ANSI C++ X3J16 Meeting** – Location: Lund, Sweeden.

News, Products, and Services

- **Compiler Design in C** by Allen I. Holub, Prentice Hall 1990, 924 pages. Contains a complete implementation of *lex* and two versions of *yacc* (called *occs*). Also, the source to an "almost-ANSI" C compiler is provided. Complete source code is available from the author for $60. It runs on the IBM-PC (Microsoft 5.1) and BSD UNIX (4.3). The code is written in ANSI C. Contact *holub@violet.berkeley.edu* for more details.

- **ANSI C Transition Guide** by AT&T from Prentice Hall, 1990, 64 pages, 0-13-933698-2. (201) 767-5937 or your local bookstore.

- **The Annotated C++ Reference Manual** by Margaret A. Ellis and Bjarne Stroustrup. Addison-Wesley 1990, 51459, 447 pages, $37.75. Covers Version 2.1 and is serving as one of the base documents for X3J16, the group working on an ANSI C++ standard.

- **C Portability Verifier** from Mindcraft, Inc. Checks application portability to POSIX.1, FIPS 151.1, POSIX.2, ANSI C, X/Open standards using static analysis and other tools. Extensible. Supports both ANSI and K&R C and is available for a wide variety of UNIX systems. Call them at 410 Cambridge Ave., Palo Alto, CA 94306, (800) 537-6749 or (415) 323-9000.

- **Measuring Parallel Performance** is the subject of Technical Report #929 by James R. Larus from the University of Wisconsin, Madison. It explains a new technique for estimating and understanding the speed improvement resulting from execution on a parallel computer. The results from testing with six substantial C programs are presented. Contact *larus@cs.wisc.edu*.

- King Computer Services, Inc., has announced a C cross-development package for the **Tandy 102 Laptop** system. It provides support for dialer access, modem, RS-232 port, sound, and graphics. 1016 North New Hampshire, Los Angeles, CA 90029, (213) 661-2063.

- Matthew Dillon has written a **Freeware C development system for the Amiga**, called DICE. It includes all source for the front-end, preprocessor, compiler, code generator, 68000 assembler, linker and library. Contact him at *dillon@Overload.Berkeley.CA.US*.

- Jim Roskind has written a pair of *yacc*-able **grammars for C++ and C** and is making them available publicly. One feature is that they handle all the ambiguities provided in the ANSI C standard that involve `typedef` names. The files can be obtained from: ics.uci.edu (128.195.1.1) in the ftp/pub directory as c++-grammar.1.0.tar.Z or, from c++gram.tar.Z, holding them all available for anonymous *ftp* at:

  mach1.npac.syr.edu:~ftp/pub/C++ and labrea.stanford.edu:~ftp/pub

- Vern Paxson has announced **Release 2.2 of flex**, a replacement for *lex*. You can get it via anonymous *ftp* to svax.cs.cornell.edu (128.84.254.2, East coast) or ftp.ee.lbl.gov (128.3.254.68, West coast). Retrieve flex-2.2.alpha.tar.Z, using binary mode.

∞