The Journal of

C Language Translation

Volume 1, Number 3

December, 1989

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

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

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

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

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

Contents

23. Diagnosing Quiet Changes

P.J. Plauger

Abstract

Special diagnostic tools have a long tradition in C. Indeed, the advent of *lint* under UNIX arguably offset much of C's bad reputation as an unsafe language. Additional diagnostic modes can aid the conversion of existing code to Standard C. Of these new modes, the most important is one that diagnoses the "quiet changes" between existing C dialects and Standard C. This paper explores some of the issues that must be addressed when diagnosing quiet changes.

Introduction

Everyone likes *lint*. It provides a way to keep project managers happy without getting in the way of programmers. You don't have to incur the extra checking overhead each time you recompile. You just ask for more meticulous checking of your program from time to time to keep it reasonably honest. And you can tailor the checks to suit your particular needs.

One way to tailor *lint* is to have it check for potential portability problems. A C program can be perfectly valid, yet have little or no chance of surviving a move to another machine. If you intend the program to be machine specific, however, you don't want to hear about portability lapses. Such gripes just obscure any legitimate complaints that you want to see.

Another way to tailor *lint* is to have it check for questionable practices. This is a way of making C a more strongly typed language without ruining the fun for everyone. It is well known that you can write all sorts of nonsense and stay within the sometimes lax rules of C. Unless you are being intentionally clever, you want to know about the nonsensical usages.

It has become common practice for some shops to mandate the use of *lint*. A typical guideline is to specify exactly the options that *lint* should be run with, then mandate that such a run produce *no* messages of any kind. You learn to spot the idioms (such as gratuitous `(void)` type casts) that programmers use to stifle some of the *lint* idiosyncrasies.

The process of conversion to Standard C introduces a whole new set of opportunities for special diagnostics. For example, you might have code written for an older dialect that has certain extensions. Compile it with a Standard C compiler and you get all sorts of diagnostics that miss the mark. A classic

extension is to tolerate the old-style assigning operators, at least where their usage is unambiguous, as in:

```
int f(i, p)
        int i, *p;
        {
        i=**p;
```

Here, only one interpretation of the character sequence `=*` makes sense, as the multiply assign operator. Better your new compiler should check explicitly for the older dialect and produce targeted diagnostics. (A really useful tool might even be able to automate some of the more obvious dialect conversions.)

If the extension is pretty unmistakable, you could make the check for it a standard part of your new compiler. If not, then you probably want the check turned off by default. A new generation of programmers endeavoring to write Standard C should mimic the extension only by accident.

The above example written by modern C programmers probably contains an error in the number of levels of indirection on a pointer. It has nothing to do with the previous existence of an alternate way to write multiply assign. You don't want to confuse programmers by accusing them of using a feature they shouldn't even know about!

The biggest conversion worry is not likely to be the elimination of past extensions, however. Much more worrisome is the introduction of "quiet changes." A quiet change is a place where at least one popular dialect of C tolerates a certain construct. It accepts it with no complaint and does something well defined. Standard C also tolerates the same construct. It accepts it with no complaint and does something well defined. The only problem is, the "something" is different.

A classic example of a quiet change occurred when C reversed the old way of writing the assigning operators. When the multiply assign operator changed, as shown above, the change was quite noisy. You can't write an expression that is valid both with and without the old spelling of the operator. The subtract assign operator is quite a different story, however:

```
int f(i, p)
        int i, *p;
        {
        i=-*p;
```

This has two distinct meanings before and after the change. In the old days, it decremented the value stored in `i` by the value stored in `*p`. Now, it assigns the negated value stored in `*p` to `i`. Even though this particular change occurred many years ago, it is documented as a quiet change in the Rationale. (See below.)

No project leader wants to hear that the introduction of a new compiler is

going to require that code be converted. Just recompiling everything, reading the diagnostics, and making the obvious changes can be a significant effort. Besides, it increases the risk that some of those changes will inadvertently introduce errors.

Now tell the project leader that the new compiler is going to change program behavior in some places *and it won't tell you when or where it makes the changes*. That's grounds for a plea of justifiable homicide in some states.

You can see where a new kind of *lint* is sorely needed. If you have any code that might suffer quiet changes when stuffed through a Standard C translator, you have a serious conversion problem. At the least, you must put your original designers and senior programmers on the job. At the worst, you must dispatch a horde of testers on the converted product. In the latter case, you can only hope that you catch the errors before your customers do.

You want a special checking option that looks for quiet changes and makes them noisy. As with the extension checking I described above, you want to be able to turn such checking on and off. When converting old code, you most definitely want to check for quiet changes. When checking converted or new code, you probably want to turn off such checks.

I can envision one situation where you might want to check for quiet changes indefinitely. If you have a product that must work under a mix of old and new translators, the dialect clash doesn't go away. Until the world (or your part of it) converts completely to Standard C, you must be careful of quiet changes. (The realists among us know that that day may never come.)

The purpose of this article is to alert you to the quiet changes. I have copied the descriptions almost verbatim from the Rationale that accompanies the C Standard. For each quiet change, I discuss how you can go about diagnosing its occurrence. (In some cases, you can only diagnose them to a certain degree.) Whether you are producing a modern *lint* or simply beefing up your in-house compiler, you should give serious thought to adding these checks as an option.

The Quiet Changes

- Programs with character sequences such as `??!` in string constants, character constants, or header names will now produce different results.

  This one is easy. If the translator detects any trigraph in an old program, it is a quiet change.

- A program that depends upon internal identifiers matching only in the first (say) eight characters may change to one with distinct objects for each variant spelling of the identifier.

  In principle, this is easy. Simply perform two lookups for each new name. One lookup uses the full name (or at least 31 characters of the full name) in the proper fashion. The other lookup uses only the prefix used in the

old dialect, but checks it against *all* names in scope. If you match any additional names over the first lookup, complain.

In practice, you can significantly slow down a translator with this check. Name lookup is often a significant contribution to translation execution time. You should profile your translator to determine if you need to retune name lookup as a result of this check.

- A program relying on file scope rules may be valid under block scope rules but behave differently—for instance, if `d_struct` were defined as type `float` rather than `struct data` in the following example:

```
typedef struct data d_struct {
        /* ... */ };
first() {
        extern d_struct func();
        /* ... */
        }
second() {
        d_struct n = func():
        }
```

  You have to maintain a file scope symbol table in addition to the usual table that reflects nesting of scopes. Lookup any symbol with linkage (external or internal) in both tables. If the symbol is not present in the file scope table, install it. If it is present, complain if the types or linkages are incompatible. The file scope table can be used only to complain, not to provide information useful to the program.

- Unsuffixed integer constants may have different types. In K&R, unsuffixed decimal constants greater than `INT_MAX`, and unsuffixed octal or hexadecimal constants greater than `UINT_MAX`, are of type `long`.

  In principle, you simply determine the type by both old and new rules and complain if you get different answers. In practice, this approach will generate a lot of noise. You must check enough context to see if the change in type is likely to make a difference in program behavior. For assignment and many arithmetic operations, it will not. The biggest area of concern is where the size in bytes of an argument changes. (For old code, there will probably be no prototype to coerce the argument to the proper type.)

- A constant of the form `'\078'` is valid, but now has different meaning. It now denotes a character constant whose value is the (implementation-defined) combination of the values of the two characters `'\07'` and `'8'`. In some implementations the old meaning is the character whose code is `078 == 0100 == 64`.

Look for any one or two digit octal escape followed by an `8` or `9`. It is bound to cause trouble.

- A constant of the form `'\a'` or `'\x'` now may have different meaning. The old meaning, if any, was implementation defined.

  Look for *any* undefined escape sequence, not just these, and complain.

- A string of the form `"\078"` is valid, but now has different meaning.

  Same as above for character constants.

- A string of the form `"\a"` or `"\x"` now has different meaning.

  Same as above for character constants.

- It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme may behave differently.

  This is a tough one. The kinds of expressions that you can easily test for this change are not ones that a programmer is likely to write. The kinds that a programmer is likely to write can easily evade even the most sophisticated of flow analysis.

  Some programs want string literals to be unique so they can be separately modified. You may be able to track down a store into a string literal. Standard C doesn't make this easy, however. For a variety of historical reasons, string literals do not carry the type qualifier `const`. Once again, you are reduced to flow analysis.

- Expressions of the form `x=-3` change meaning with the loss of the old-style assignment operators.

  In principle, you just scan for the old style operators and complain if you see any. In practice, you need worry only about add assign and subtract assign. It is impossible to pun on any of the others. Since this is such an old dialect, however, you may want to control it as a separate checking option.

- A program that depends upon unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered the most serious semantic change made by the Committee to a widespread current practice.

  Once again, the direct approach is almost certainly too noisy. (That was one of the justifications for accepting value preserving rules—they almost never alter the result of expression evaluation.) You must, of course, type each expression from the bottom up by both unsigned preserving and value preserving rules. Where they differ, one operator will have an unsigned type and the other will have a signed type of the same size in

bytes. That means you need set aside only a single bit to record whether a potential change exists.

Even so, you should check context carefully. Most differences disappear quickly before their effects can be felt. Problems typically arise when the subexpression with changed type is an operand of a divide, remainder, right shift, or comparison operator (other than equality or inequality). Even if you diagnose all doubtful situations, you should find that the message seldom occurs. (Encourage the addition of type casts to disambiguate expressions whose meaning could possibly change.)

- Expressions with `float` operands may now be computed at lower precision. The Base Document specified that all floating point operations be done in `double`.

  Again, you must type the expression tree both ways and look for changes. If there is a difference, one tree has type `double` and the other has type `float`. So again you need add only a single bit to track the change.

  Filtering out diagnostics is a bit trickier than for some of the quiet changes described earlier. Floating point arithmetic is notoriously perverse, particularly when you start reducing the precision you retain for intermediate results. It is hard to determine statically (for any operands other than floating literals) that the loss of precision will have no significant effect. Expressions of the form `a = b` *op* `c`, where all the operands have type `float`, are the safest to disregard.

  You may wish to key this check to a separate option. Barring that, you should probably warn of all possible changes.

- A program that uses `#if` expressions to determine properties of the execution environment may now get different answers.

  This one is a little tough. Never underestimate the cleverness of a Berkeley undergraduate. They have produced some pretty audacious expressions in their zeal to get the preprocessor to reveal its arithmetic properties. (They also rashly assume that preprocessor arithmetic is the same as target machine arithmetic. Not so.)

  You should be diagnosing overflow in translation time arithmetic. That will make many of the clever expressions go from quiet to noisy. If you also look for overflow in `unsigned` arithmetic, at least in `#if` expressions, you will generate even more useful noise. (This effectively replaces modulus arithmetic with truly unsigned arithmetic.) Diagnose any conversion of a negative signed value to unsigned and you can probably generate a bleat for every extant trick. Naturally, you should also disallow character constants in `#if` expressions if you want to warn of potential quiet changes.

- The empty declaration `struct x;` is no longer innocuous.

I have trouble believing that many programs have such declarations lying about. You may want to diagnose them all as likely deadwood. Otherwise, just complain if the declaration is effective in its new role. It is supposed to isolate the structure tag from any declarations in containing blocks.

- Code which relies on a bottom-up parse of aggregate initializers with partially elided braces will not yield the expected initialized object.

  Here again, you may be better off diagnosing essentially all aggregate initializers that are missing internal braces. The opportunities for ambiguity are ample. Only if that approach proves too noisy for your customers should you look at pruning the cases. I recommend, however, that you *not* try to parse the initializer two different ways and look for differences in interpretation. Better to err on the noisy side.

- Type `long` expressions and constants in switch statements are no longer truncated to `int`.

  You can probably diagnose all switch statements whose control expression has type `long` or `unsigned long`. If that proves to be too noisy, you can eliminate *all* such diagnostics if the target has the same representation for `int` and `long`.

  Don't forget to look also for `case` values that truncated in the past but not in Standard C.

- Functions that depend on `char` or `short` parameter types being widened to `int`, or `float` to `double`, may behave differently.

  Here is another case where you don't want to diagnose every instance of a function declared with suspect argument types. They occur far too often and are generally innocuous. Look for assignments to the arguments that may now truncate values but did not do so in the past. And pay close attention to anyplace the program takes the address of a suspect argument. It is almost certain to cause trouble. Just filtering for these cases will eliminate many spurious diagnostics.

- A macro that relies on formal parameter substitution within a string literal will produce different results.

  All you have to do is parse string literals within macro definitions just like in the old days. If any contain "identifiers" that match parameter names, complain. Remember, however, that this convention was far from universal in the past. You may want to control it with a separate checking option.

- A program which relies on size-0 allocation requests returning a non-null pointer will behave differently.

  Here is another tough one. Chances are, the code contains no explicit `malloc(0)` calls. The most sophisticated flow analysis is not likely to

catch all possible zero value arguments at translation time. It is easy enough to diagnose this change at runtime, but that takes you beyond the traditional realm of *lint*.

Conclusion

Fortunately, nearly all of the quiet changes can be detected at translation time. The exceptions are few:

- Assumptions about string literals being identical (or unique) can be very difficult to diagnose.
- Arithmetic in `#if` expressions can make subtle environment checks.
- Allocating an object of zero size can occur at runtime but evade checks at translation time.

All of these practices have long caused portability problems, however, since they differ among popular implementations. Even if you make no attempt to catch these particular trouble makers, diagnosing all the rest can be a real aid to upgrading C code.

If you want to read more about the quiet changes, see the Rationale. You can also read three of the "Standard C" columns I have written for the *C Users' Journal*. These are "Standard C Promotes Types According to Value Preserving Rules," August '88; "Quiet Changes, Part I," February '90; and "Quiet Changes, Part II, March '90.

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. He can be reached at uunet!aussie!pjp.

∞

24. Pragmania

Rex Jaeschke

Macro Expansion Revisited

In my last installment I incorrectly stated "The ANSI Standard says nothing about the expansion of macros in pragmas." As was pointed out by two respondents to the current electronic survey, the standard states in no uncertain terms what should happen. §3.8 page 87, line 15, of the December 88 draft states "The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated." §3.8.6 **Pragma directive** on page 94 does *not* indicate that macros are expanded so we must conclude they are not.

Having said all this I still think the idea of expanding macros in pragmas is worth considering, given some of the things implementers are, or plan to be, using pragmas for.

As I understand it, in strictly conforming mode, a pragma cannot change the outcome of a program. That is, the program must have the same semantics both with and without the pragma. This requirement provides very little creative latitude when inventing pragmas. On the other hand, many implementers having pragmas clearly require compilers to be run in "extended mode" for pragmas to have their desired affect. (Examples are pragmas that alter structure padding or argument passing due to language interface requirements.)

In any event, if you expand a macro in a pragma *and* you claim ANSI conformance, how could the formal validation suite ever tell? Clearly it cannot without a test specific to your implementation, and that's not what validation suites do.

Since I first proposed macro expansion privately early in 1989, I have had significant discussions with vendors on its benefits. However, the biggest problem presented thus far came from Walter Murray at Hewlett-Packard. It's one of namespace pollution. For example, assume an implementer provides the following pragma forms:

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

If macros are expanded, then `OFF` or `ON` (or even `OPTIMIZE`) would be replaced by unexpected (and therefore unrecognized) text if they were currently defined as macros. You could even have the situation that in some scopes these

macros would not be defined and in other places they would be, leading to the pragma sometimes being ignored and sometimes not. At the very least, such an implementation *must* warn about unrecognized pragmas.

The problem then is that all identifiers in a pragma whose names are in the user namespace are at risk of being expanded. To force implementers to protect such names, I think, is unacceptable. To do so would require something like:

```
#pragma _OPTIMIZE _OFF
#pragma _OPTIMIZE _ON
```

And the naming conflict comes not only from the source and headers provided by the user, it also comes from all the third-part library headers he includes as well.

One vendor suggested they might expand only certain identifiers inside particular pragmas. I suspect that will make their implementation more complicated, but since the pragma in question actually generates code conditionally, it's already being handled specially.

It seems this proposal is not yet settled, so I encourage you to let me know your thoughts. From this point on, though, I suggest we assume this proposal is restricted to being an extension.

One final thing. If macros were expanded, what if no macro `ON` (or `OFF`) was currently defined? Would you want the default value to be zero as in `#if` expressions, or null? Whichever, how could you tell which identifiers in the pragma are, or can be, macros? Those without leading underscores? I think you would want the preprocessing token left as is with no expansion to a default value.

DEC's VAX C Version 3.0

Various parts of this section are extracted from DEC's manual, *Guide to VAX C*, © Digital Equipment Corp., 1989.

Intrinsic Function Recognition

```
#pragma   builtins
#pragma nobuiltins
```

These directives enable or disable access to predefined functions. When enabled, these functions do not result in a reference to a function in the run-time library or in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site.

About 30 VAX instructions can be accessed directly in this manner. They include operations for: bit searching, queue manipulation, character searching,

block move, and read and write specific general and other registers. Apart from a family of general purpose functions, there are some that are specific to systems programming on the VAX. An example of an inline block move follows.

```
#include <stdio.h>

#pragma builtins

main()
{
        static char name[30];

        printf("name contains: >%s<\n", name);
        _MOVC3(10, "More Text", name);

/* Instruction generated is: movc3  #10, ..., ... */

        printf("name contains: >%s<\n", name);
}
```

By default, builtins are not recognized as such.

An interesting aspect of builtins is that some of them have more than one form of argument list, yet each form has its own implicit prototype in scope at each call. That is, you must call such functions with any one of the formats expected for that function. For example, the block move instruction MOVC3 is accessed in any one of the following ways:

```
void _MOVC3(unsigned short length, const char *src,
        char *dest);

void _MOVC3(unsigned short length, const char *src,
        char *dest, char **endsrc);

void _MOVC3(unsigned short length, const char *src,
        char *dest, char **endsrc, char **enddest);
```

Since these builtins behave much like keywords, DEC has wisely spelled them with a leading underscore and capital letter.

Parallel Programming Directives

You can tell the compiler that the specified variables within the next encountered `for` or `while` loop should not inhibit the decomposition of that loop. (*Decomposition* is DEC's name for parallelization.) The pragma format is:

```
#pragma ignore_dependence (id, ...)
```

This pragma affects only the next `for` or `while` loop and does not affect nested loops. *id* must be a previously declared pointer variable and the parentheses are optional if the list contains only one identifier.

The `safe_call` pragma tells the compiler that the specified function call (or calls) does not introduce data dependencies that prevent decomposition of a `for` or `while` loop. If you specify this directive outside of a function body, the functions named in the directive are globally safe in all `for` and `while` loops from the occurrence of the directive to the end of the source file. When specified inside a function body, only the next `for` or `while` loop is affected.

The format of the pragma is:

```
#pragma safe_call (id, ...)
```

where *id* is a function name or function pointer that is already declared.

This pragma should not be used if the function does any of the following:

- Has side effects that introduce data dependencies.
- Is not reentrant.
- Calls `longjmp`, or otherwise modifies the normal flow of control.
- Changes the executing process in some way.
- Takes an address as an argument, and that address points to memory that is not shared.

Loop decomposition can be disabled using the following directive:

```
#pragma sequential_loop
```

This pragma suppresses decomposition for the next `for` or `while` loop only.

Inline Code Generation

The `inline` pragma suggests to the compiler that it provide inline expansion of selected functions. By default, VAX C attempts to provide inline expansion for all functions. The compiler also uses the following function characteristics to determine if it can provide inline expansion: size, number of times called, and absence of various restrictions, as discussed below.

For a function to be expanded inline, the function definition must reside in the same source file as the calls to that function. The definition may appear either before or after the calls to it. Functions cannot be expanded inline if they perform any of the following tasks:

- Take the address of an argument.
- Use an index expression that is not a compile-time constant in an array that is a member of a structure argument. (An argument that is a pointer to a structure is not so restricted.)
- Use `varargs.h` or `stdarg.h` to access the function's arguments.
- Declare an exception handler.

The format of the pragma is:

```
#pragma   inline (id, ...)
#pragma noinline (id, ...)
```

where *id* is a function identifier.

Inlining can also be enabled and disabled using a compiler option.

Structure Member Alignment

By default, VAX C does not align structure members; they are stored on byte boundaries (with the exception of bit-field members). However, you can use the following pragma to specify member alignment.

```
#pragma member_alignment
```

When this pragma is used, the compiler aligns structure members on their natural boundaries. (The user can chose either no or full alignment only.)

Specifying the Warning Level

Some standard headers contain VAX C-specific keywords, and since the preprocessor does not distinguish between system and user headers, it produced a portability warning. Such messages can be suppressed using the following pair of pragmas:

```
#pragma nostandard
#pragma   standard
```

∞

25. Design of a FORTRAN to C Translator, Part I

Fred Goodman
PROMULA Development Corporation
Columbus, Ohio

Abstract

Waite and Goos in *Compiler Construction* [1] define a *compilation* as "a sequence of transformations (*SL*,*L1*), (*L1*,*L2*), ..., (*Lk*,*TL*) where *SL* is the source language and *TL* is the target language." This article describes and motivates the design of PROMULA.FORTRAN, a compiler whose source language is FORTRAN, whose intermediate language is a reverse polish pseudo-code, and whose target language is C.

The major objectives of the design are: (1) that the C output produce the same results as the FORTRAN original, (2) that the user be able to modify the FORTRAN dialect description, and (3) that the user be able to modify the look of the C output.

The bulk of the paper deals with objective (3). A *surface-form* description language is presented along with an example translation. Given that objective (1) above must always be met, there are three possible views of the C output: that it be as efficient as possible to compile, that it be as C-like as possible, and that it be as much like the original FORTRAN as possible. These views are discussed along with their impact on the design of the runtime library. Finally, surface-form representations are given for each in terms of the example translation.

Introduction

In the November/December 1987 *Micro/Systems Journal*, A.G.W. Cameron reviews a FORTRAN to C translator. In that review, he takes the position that the only difference between a translator and a compiler should be that the compiler translates the source code into assembly code while the translator takes it to a higher level language. This article discusses a FORTRAN to C translator, PROMULA.FORTRAN, which takes precisely this approach to translation. First, it compiles the FORTRAN source code into a low level pseudo-code, much like the pseudo-code produced by the first pass of contemporary compilers. Second, it optimizes that code, again using the same techniques as would be used by a conventional compiler. Third, it does code generation. But the code generated is not machine code, it is C.

What Cameron does not discuss, however, is that though the processes of translation and compilation are largely the same, the objectives of users of translators are much more varied than those of compiler users. The user of a compiler does not care what the output code looks like—it is merely an intermediate step. Though some users of PROMULA.FORTRAN also view the output C merely as an intermediate step, most view it as a final step. They care very much what the C looks like, because they intend to discard the FORTRAN original once the translation is completed. In general, users of PROMULA.FORTRAN can be divided into three broad classes:

1. Those who want to continue using their present FORTRAN dialect as their programming language. For these users the C output is of no importance as such. It should be designed to compile as quickly as possible.

2. Those who are presently FORTRAN programmers, but who want to become C programmers. For them the C output should be as close to the original FORTRAN as possible to ease the transition.

3. Those who are C programmers who must now take over a FORTRAN code. For them the C output should look as much as a standard C program as possible.

The techniques used to produce the target language are based on the work done in the area of re-creation of source code. In particular, P.J. Brown [2] discusses the re-creation of source code from reverse polish notation. This paper reviews that work and shows how it can be easily extended via a *surface-form description language* to allow not only for the surface-form variation needed to accommodate different user biases, but also to give the final user of the translator the ability to tailor the translation to his own needs. To motivate the discussion of the surface-form description language, a brief discussion of the history and design objectives of PROMULA.FORTRAN is given.

The discussion is presented in journalistic style. The purpose of the presentation is to describe how a translation is achieved, with emphasis on the output side. This emphasis is chosen because most readers of this article will already be familiar with computer language input processing. Each section becomes increasingly detailed. However, the basic ideas are presented early for those who only want an overview of the approach.

Background

PROMULA.FORTRAN has been under development since 1982. It was a child of necessity. The original FORTRAN compilers available for microcomputers were pathetic by comparison to their mainframe ancestors. This was not true of the early C compilers though, especially after large memory model versions became available. We are consultants, and were often asked to migrate mainframe FORTRAN codes to the PC. This migration task proved much easier

to achieve via translation to C. Given this need to actually use the translator as a substitute for the existing FORTRAN compilers, our first design goal was that the translated version had to produce the same results as the original FORTRAN version. Our original view of the C output was that it was merely an intermediate step. It had to be efficient and readable, but not necessarily maintainable.

The initial version of PROMULA.FORTRAN took the approach of examining each FORTRAN statement in the source program and then producing from it an equivalent statement, or statements, in C. Our view of translation was that it was basically a string manipulation problem—it was nothing like compilation. Given constructs like implied `DO` loops, assigned `GOTO` statements, and the hiding of characters in FORTRAN 66, it became clear that if any degree of completeness were to be achieved, then the program units had to be processed as a whole and then examined. Statement by statement processing simply did not suffice. The resultant C codes either would not compile at all or did not give the correct results.

Our second approach was based on the UCSD p-System [3], only with C as the base and not Pascal. If we could not compile our FORTRAN using existing FORTRAN compilers, and if we could not use the available C compilers directly, then perhaps we could design a C-based pseudo-machine. The FORTRAN codes could be compiled into a pseudo-code—a stack-oriented reverse polish notation. This pseudo-code could then be executed via C. This approach worked. By going to pseudo-code, it was possible to correctly process FORTRAN codes. The system produced the right results, but it was agonizingly slow. And since the entire runtime library had to be linked with the p-machine the executable was very large—this was not the way to go.

The solution came when we discovered the discussion in [2] about the re-creation of source code from reverse polish notation. Using this approach we could compile our codes using the p-machine compiler developed above, and then we could re-create its source code. Only the source code re-created was now C and not FORTRAN.

The Origins of the Approach

The approach taken in the current version of PROMULA.FORTRAN is based on the standard theories of compiler design. Waite and Goos in [1] define a *compilation* as "a sequence of transformations (*SL*,*L1*), *L1*,*L2*), ..., (*Lk*,*TL*) where *SL* is the source language and *TL* is the target language." The languages *L1* .. *Lk* are referred to as *intermediate* languages. PROMULA.FORTRAN, then, is a compiler whose source language is FORTRAN, whose intermediate language is a reverse polish pseudo-code, and whose target language is C.

The problem with the standard theories of compiler design is that though there is ample discussion of how to do the conversion from the source language to the intermediate language, there is little discussion of how to move from an

intermediate language to a higher level target language such as C. For compilers, the target language is always a low-level machine code. Fortunately, there has been some work done in the area of the re-creation of source code from intermediate code. Many BASIC interpreters need to be able to execute BASIC statements efficiently, while still giving the user access to the source code. This is a problem also faced by contemporary spreadsheet systems. The re-creationists solve this problem by compiling the source statements into a reverse polish notation and then rewriting the source code from that internal notation. An excellent discussion of this work can be found in [2].

The technique of source code recreation from reverse polish notation hinges on the observation that executing such notation via a stack-oriented pseudo-machine is identical to writing and combining the character sequences that perform those operations in some target language via a stack-oriented string-manipulation machine! The above can best be understood via a simple example.

Consider the following statement which we wish to both execute via a p-machine and re-create:

```
A = B * C + D
```

The intermediate form of this statement might be as follows:

```
PUSHADR   A
PUSHADR   B
GETVAL
PUSHADR   C
GETVAL
MULT
PUSHADR   D
GETVAL
ADD
PUTVAL
EOS
```

where for the execution pseudo-machine:

Op Code	*Meaning*
PUSHADR	push the address of the variable onto the stack.
GETVAL	pop the address from the stack, obtain the value at that address, and push it onto the stack.
MULT	pop the top two values from the stack, multiply them together, and push the result onto the stack.
ADD	pop the top two values from the stack, add them together, and push the result onto the stack.
PUTVAL	pop the value and address from the stack and then store the value at the address.
EOS	end the current statement.

And for the string-manipulation machine:

Op Code	*Meaning*
PUSHADR	enter the notation for a pointer to the variable onto the stack.
GETVAL	pop the top string from the stack, convert it to the notation for a value at the indicated address, and push that string back onto the stack.
MULT	pop the top two strings from the stack, concatenate them with the symbol '*' in the middle, and push that string back onto the stack.
ADD	pop the top two strings from the stack, concatenate them with the symbol '+' in the middle, and push that string back onto the stack.
PUTVAL	pop the top two strings from the stack, convert the lower one to a value string, concatenate the two strings with the symbol '=' in the middle, and push the resultant string onto the stack.
EOS	write the current string.

Let us execute these two machines and watch the symmetry between them. Assume that A is stored at address 45, that B is at address 50 and has value 2.0, that C at 55 has value 3.0, and D at 60 has value 4.0. It is both exciting and interesting to note that these address and value assumptions are needed for the execution pseudo-machine only. Alternately, the string-manipulation machine needs access to the symbol table, which the execution-machine does not.

Instruction	*Execution Stack*	*String-Manipulation stack*
PUSHADR A	45	"&A"
PUSHADR B	45, 50	"&A", "&B"
GETVAL	45, 2.0	"&A", "B"
PUSHADR C	45, 2.0, 55	"&A", "B", "&C"
GETVAL	45, 2.0, 3.0	"&A", "B", "C"
MULT	45, 6.0	"&A", "B*C"
PUSHADR D	45, 6.0, 60	"&A", "B*C", "&D"
GETVAL	45, 6.0, 4.0	"&A", "B*C", "D"
ADD	45, 10.0	"&A", "B*C+D"
PUTVAL		"A=B*C+D"
EOS		

Though this simple example only shows how to translate A = B * C + D back into itself, its extension to approximately 200 pseudo-operations, and its generalization via a surface-form notation, allowed us to construct a very fast and sophisticated translator. With a single extension for dealing with operator hierarchies, to be discussed below, the above is all that PROMULA.FORTRAN does. It produces pseudo-code and a symbol table from the source code, simplifies the code by using an execution machine, and then writes the code back out in user-definable C form via a string manipulation machine.

A serious problem faced by the re-creationists is that the production of the output string is completely independent of the original input made by the user. Though the original and the result mean the same thing, they might look very different. Fortunately, in the application to translation this is not a problem. The user wants semantic and not syntactic identity. The greatest weakness of the re-creation approach becomes the greatest strength of PROMULA.FORTRAN.

The Problem with Parentheses

A problem that was ignored in the above example has to do with parentheses or operator precedence. The reason reverse polish notation is typically used as an intermediate language, and even in the design of contemporary machine languages, is that it uses no parentheses. Consider the statement: A = B * (C + D) in which the addition operation is to be performed prior to the multiplication. Using the same notation as above the intermediate form of this statement would be as follows:

```
PUSHADR   A
PUSHADR   B
GETVAL
PUSHADR   C
GETVAL
PUSHADR   D
GETVAL
ADD
MULT
PUTVAL
EOS
```

Note that the effect of the parentheses in the intermediate form was simply to reorder the operations.

Using the same execution and string-manipulation rules as before, the following would result:

Instruction	*Execution Stack*	*String-Manipulation stack*
PUSHADR A	45	"&A"
PUSHADR B	45, 50	"&A", "&B"
GETVAL	45, 2.0	"&A", "B"
PUSHADR C	45, 2.0, 55	"&A", "B", "&C"
GETVAL	45, 2.0, 3.0	"&A", "B", "C"
PUSHADR D	45, 2.0, 3.0, 60	"&A", "B", "C", "&D"
GETVAL	45, 2.0, 3.0, 4.0	"&A", "B", "C", "D"
ADD	45, 2.0, 7.0	"&A", "B", "C+D"
MULT	45, 14.0	"&A", "B*C+D"
PUTVAL		"A=B*C+D"
EOS		

Obviously the execution result of 14.0 is correct, but the string-manipulation result of A = B * C + D , which is identical to the previous result, is wrong. The parenthesis have been omitted. The re-creationists solve this problem by entering special markers into the intermediate notation to indicate where the parentheses occurred. This technique obviously does not work in the translation application, because the placement of the parentheses in the target language is a function of the operator precedence of the target language and not of the source language.

The solution taken in PROMULA.FORTRAN is quite simple and general. Each rule in the string-manipulation machine has associated with it a precedence code. Whenever the output of an operation is written to the string-stack, the precedence code of the rule that produced the output is also written. Whenever an element is combined within a rule, if its precedence code is nonzero and

lower than the code of the rule, then that element is enclosed in parentheses as it is concatenated into the new output.

Applying this extension to the example above, the precedence codes for the various operations are as follows:

Operation	*Precedence*
PUSHADR	0
GETVAL	0
MULT	2
ADD	1
PUTVAL	0

Note that operations not involved in evaluations have a precedence of zero. MULT has a higher precedence then ADD because in C multiplication is higher in precedence than addition. Using the precedence rule from above we can again run the string-manipulation machine on the intermediate form of A = B * (C + D). The rule and stack precedence codes are shown in parentheses.

Instruction	*String-Manipulation stack*
(0) PUSHADR A	(0) "&A"
(0) PUSHADR B	(0) "&A", (0) "&B"
(0) GETVAL	(0) "&A", (0) "B"
(0) PUSHADR C	(0) "&A", (0) "B", (0) "&C"
(0) GETVAL	(0) "&A", (0) "B", (0) "C"
(0) PUSHADR D	(0) "&A", (0) "B", (0) "C", (0) "&D"
(0) GETVAL	(0) "&A", (0) "B", (0) "C", (0) "D"
(1) ADD	(0) "&A", (0) "B", (1) "C+D"
(2) MULT	(0) "&A", (2) "B*(C+D)"
(0) PUTVAL	(0) "A=B*(C+D)"
EOS	

The critical point in the above occurs when the output of the ADD operation, which has a precedence code of 1, is combined via the MULT operation which has a precedence of 2. Since 1 is nonzero and less than 2, the string C+D is enclosed in parentheses as it is entered into the output string of the MULT instruction.

The Translation Algorithm

To understand the surface-form description language, an overview of the entire translation algorithm is first needed. This algorithm has three requirements:

1. The source language must be translatable into an intermediate language all of whose operators are expressed in reverse polish notation and—be they unary, binary, ternary, etc.—produce either a single result or no result.

2. In the target language every *n*-ary operator can be expressed as an arbitrary but fixed concatenation of its operands and of other fixed character sequences.

3. It must be possible to gain access to all operations performed in the source language via the target language.

The first two requirements are easy to achieve via almost any contemporary language. The third is the killer. We were asked at one time, for example, if we could translate C into FORTRAN. The answer was *No* because C pointer operations cannot be expressed in FORTRAN.

The actual application of the requirements will become clearer in the following, though emphasis will be placed on target language production. Remember also that the world is not perfect. The algorithm works, but there are places where special processing is needed, especially in the area of the data definition component. PROMULA.FORTRAN is not a universal translator—it is a translator from FORTRAN to C. It is sufficiently general, however, to give the wide range needed to deal with the many FORTRAN dialects and the many user output biases.

The Components of a Translation Definition

A complete translation definition contains seven major components, of which the first five are general and accessible to the end-user and two are hardwired in the translator. They are as follows:

1. A definition of the basic operation codes which make up the pseudo-machine. These definitions form the glue which ties the other components together.

2. A definition of how the expressions of the language are broken down into operation codes. This component contains five subcomponents:

   - The operations needed to perform type conversions
   - The promotion hierarchies used when deciding what type conversions to perform
   - The unary operators
   - The binary operators
   - The functions available

3. A definition of the actual statements in the source dialect. Within this component statements are classified by type. Each type contains its own additional information requirements. Both data definition and executable statements are included in this component.

4. A description of the C surface-form to be taken by each operation. These surface-form descriptions look much like the control strings used by the C `printf` function, except that the conversion symbols refer to the elements on the translation stack.

5. A runtime library which performs those operations referenced in the target language but not implemented directly.

6. A set of hard-wired functions built into the translator that process the basic input structure of the source language and that build the various symbol tables.

7. A set of hard-wired functions built into the translator that process the basic output structure of the target language and that produce the needed data definition statements from the symbol tables.

As was said above, the basic flavors of FORTRAN and C are built into the translator via the last two components. These cannot be altered by the end-user. It is this hard-wiring that makes the translator fast and compact.

When new users first look at the PROMULA.FORTRAN translation definition file, they are usually surprised to see what appears to be just a long list of operation code identifiers. It is, however, in terms of these identifiers that the entire translation process is tied together. All executable FORTRAN statements are translated into these operation codes, and all executable C output statements are described in terms of these codes. PROMULA.FORTRAN is ultimately a pseudo-code compiler.

Via the second and third components, the user has considerable flexibility in defining his FORTRAN source dialect. Not much will be said about the notation used for describing the source dialect. It is interesting to note, however, that traditional language descriptions lump expression description in with statement descriptions. We have found that processing is much more efficient if the description of, and therefore processing of, expressions is separated from statement description and processing. Thus, component (2) describes expressions, while component (3) describes how those expressions are combined to form statements.

Components (4) and (5) will be discussed throughout the remainder of this paper. The actual surface-form description strings are contained in (4) and any functions referenced via those surface form description strings are contained in (5). Potential users of PROMULA.FORTRAN occasionally express surprise at the runtime library. Why, if we are translating into C, do we need a runtime library? To them it seems like cheating. Only C functions should be used if a

true translation is achieved. Clearly, there are operations performed in FORTRAN that are not directly accessible in C—complex arithmetic, FORTRAN style formatted I/O, etc. These operations must be performed via runtime functions, but the structure of these functions is largely determined by how the translation is performed. Thus, components (4) and (5) are closely tied.

The Surface-Form Description Language

For each operation code in the intermediate language, there is an entry in the surface-form description language. Each operation description has three components which may vary by the user output bias type:

1. A specification of the number of operands associated with the operation—that is, whether the operation is null, or unary, or binary, etc.—for the particular output bias. Remember that all operations are reverse polish. Therefore, when a given operation is encountered, its operand strings have already been placed on the string-machine stack. The operands are numbered starting with the oldest first. In other words, the operand deepest on the stack is argument 1 and the operand at the top of the stack is argument n, for an n-ary operator.

2. A specification of the precedence of the operator relative to others. As the output production proceeds, it is necessary to enclose certain operations in parentheses to achieve the proper order of evaluation. The current precedence of each operand is maintained. When two operators of lower precedence are combined via an operator of higher precedence, then they are enclosed in parentheses.

3. A pattern string that specifies an arbitrary but fixed concatenation of the operands and of other character sequences which can be described via a linear pattern string. This pattern string is, of course, the actual implementation of our re-creation algorithm requirement number (2) as presented earlier. The pattern string describes not only how the operands are combined but also how the various constants, symbol table entries, and miscellaneous special-purpose conversion routines combine to form the final output.

The bulk of the discussion below deals with pattern strings. They are deceptively simple.

The Example Revisited

Earlier, a set of six operation codes was presented along with a description of what each did when executed via the string-machine. This subsection presents the identical information using the surface-form notation in order to introduce the concept. The actual specification is as follows:

```
SURFACE-FORMS
PUSHADR   PATTERN 0, 0, "\v"
GETVAL    PATTERN 1, 0, "%1i"
MULT      PATTERN 2, 2, "%1d * %2d"
ADD       PATTERN 2, 1, "%1d + %2d"
PUTVAL    PATTERN 2, 1, "%1i = %2d"
EOS       PATTERN 1, 0, "%1d\c"
END
```

The first number parameter specifies the number of arguments and the second the precedence code. The string is the output pattern string. The basic operation of the pattern strings is straightforward. The output processor moves to the next pseudo-code in the intermediate language. It then looks up the pattern information for that code. It removes as many operand strings from the string stack as are specified in the first parameter of the specification. It saves pointers to these operand strings and the current precedence associated with each in a temporary buffer. Next, a new string is formed using the actual pattern string as a guide. Finally, the result string is pushed onto the stack and assigned the precedence specified in the second parameter.

Examining the pattern strings themselves, notice first that they resemble those used by C to specify output conversions. This similarity is deliberate, since the purpose of these strings is, in essence, the same. Within the pattern strings there are three types of specifications:

1. Special operation parameters which consist of a backslash followed by a letter. These parameters trigger special conversions. Thus, in the above rule for PUSHADR the `\v` specifies that a pointer to a variable whose symbol number is specified following the opcode is to be entered into the output string at the indicated location. The `\c` notation in the EOS string specifies that the current string stack is to be cleared and thus written to the output file.

2. Operand conversion parameters, which consist of a percent sign, followed by a numeric digit, followed by a conversion code. The numeric digit specifies which operand is to be entered at this point in the string, and the conversion code specifies any special operation to be performed. In the above, the `%1d` which occurs in the MULT, ADD, and EOS strings says "Enter the first operand without any additional editing other than any precedence editing which may be needed into the output string." The `%2d` says the same thing for operand 2. The `%1i` specification in GETVAL and PUTVAL says "Enter the string in instantiated form—that is, change it from a pointer representation to the representation of the value pointed to."

3. Simple character specifications are any characters not forming one of the two specifications above. Simple characters are entered into result strings exactly as entered.

This is basically all there is to know about pattern strings. There are obviously more specification codes; however, their description here is not important. As can be appreciated, the above notation is extremely powerful, especially when combined with the notion of user output bias.

Accounting for User Bias

As was discussed in the introduction, different users have different uses for the output of the translator. In particular, some want optimized code, some want C-like maintainable code, and some want FORTRAN-like code. This general issue is referred to as the user bias. To deal with this problem, the surface-form description language allows separate entries for each user bias. The next section will deal with some input-output statements in great detail. To introduce the topic let us look once more at the simple statement: A = B * (C + D). Let us now pretend that under certain circumstances the variables A, B, C, and D are complex. In this case, the complex functions `cadd` and `cmul` perform the arithmetic operations. We will assign this complex condition a bias code of C. Now, the surface-form description is as follows:

```
SURFACE-FORMS
PUSHADR   PATTERN 0, 0, "\v"
GETVAL    PATTERN 1, 0, "%1i"
MULT      PATTERN
          C  2, 0, "cmul(%1d, %2d)"
          *  2, 0, "%1d * %2d"
ADD       PATTERN
          C  2, 0, "cadd(%1d, %2d)"
          *  2, 1, "%1d + %2d"
PUTVAL    PATTERN 2, 1, "%1i = %2d"
EOS       PATTERN 1, 0, "%1d\c"
END
```

Whenever a given operation has an alternate form for one or more bias codes, the separate entries are entered on separate lines, preceded by the bias code. The '*' code is always last and specifies the default pattern. Note that the precedence codes are different between the two output biases. In a later example, even the number of parameters will differ.

Executing the string-machine using these two alternate biases looks as follows:

Instruction	*Complex Bias*	*Default Bias*
PUSHADR A	"&A"	"&A"
PUSHADR B	"&A", "&B"	"&A", "&B"
GETVAL	"&A", "B"	"&A", "B"
PUSHADR C	"&A", "B", "&C"	"&A", "B", "&C"
GETVAL	"&A", "B", "C"	"&A", "B", "C"
PUSHADR D	"&A", "B", "C", "&D"	"&A", "B", "C", "&D"
GETVAL	"&A", "B", "C", "&D"	"&A", "B", "C", "D"
ADD	"&A", "B", "cadd(C,D)"	"&A", "B", "C+D"
MULT	"&A", "cmul(B,cadd(C,D))"	"&A", "B*(C+D)"
PUTVAL	"A=cmul(B,cadd(C,D))"	"A=B*(C+D)"
EOS		

Note that though the translations look quite different, the code generated for the two biases is identical. The C output is controlled entirely via the the surface-form description language.

References

[1] Waite, W.M. and G. Goos *Compiler Construction.* Springer-Verlag, New York, 1984, p. 4.

[2] Brown, P.J. "More on the Recreation of Source Code from Reverse Polish." *Software Practice and Experience*, Vol 7, 545-551 (1977).

[3] *Internal Architecture Guide for the UCSD p-System (TM), Version IV.0*, IBM Personal Computer Language Series, January 1982.

In the second and final part of this article, a detailed sample translation willl be presented. This is focussed on the FORTRAN `OPEN` statement.

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

∞

26. Numerical C Extensions Group

Rex Jaeschke
Convener

The second meeting of the Numerical C Extensions Group (NCEG) was once again hosted by Cray Research at their Mendota Heights, MN facility. The meeting took place for one and a half days on September 19–20 and was attended by approximately 30 people. Several new companies were represented as were various academic users.

With the organizational aspects having been dispensed with at the first meeting, work started in earnest to propose, debate, and critique submitted papers, and generally to work toward the long term goal of publishing a Technical Report (TR) through ANSI.

Reports were received from various informal liaisons NCEG has with other groups: Parallel Computing Forum, IFIP TC2/WG 2.5 Numerical Software, X3J11 and ISO C. Since then I have agreed to attend the first meeting of X3J16 (ANSI C++) in Washington D.C. on December 15th.

Almost all of the technical agenda time was given to the following topics: IEEE arithmetic, exceptions, aliasing, nonzero-based arrays, complex arithmetic, variable dimensioned arrays, and vector and parallel support. To gather user feedback as well as provide additional capability, at least one vendor is known to be working on adding complex and variable dimensioned array support to their next release.

Wording for a formal project proposal was distributed and, with minor changes, was agreed upon. This document will be circulated to X3J11 voting members currently eligible to vote, along with a letter ballot. They will be asked to vote on admitting NCEG as a working group within X3J11, tentatively named X3J11.1. If they so approve, the vote and proposal will be forwarded to X3 for processing. We hope this is all completed by the March 1990 meeting. Since NCEG's aim is to publish a report *not* a standard, no formal public review period is required. In short, from an administrative point of view, a TR is much simpler than a standard.

Many thanks to Cray for agreeing to host the meeting at short notice once it was announced that the joint X3J11/NCEG meeting in Salt Lake City was cancelled. The next meeting is scheduled in New York City, March 7–8, 1990 immediately following X3J11. The meeting after that is being proposed in the Livermore, California area (an hour east of San Francisco) around September.

To submit a paper, call Tom MacDonald to get a document number. Please put the document number and title at the top of each page. Deadline for the final mailing for the March meeting is January 19.

For more information about NCEG, contact either Rex at (703) 860-0091 or *uunet!aussie!rex*, or Tom at (612) 681-5818 or *tam@cray.com*.

∞

27. Parallel Programming: Linda Meets C, Part III

Jerrold Leichter
Yale University

Abstract

Linda is a programming model for developing explicitly parallel programs. In earlier parts of this series, we introduced the model and discussed several different "injections" of it into C, which in effect defined several dialects of "Linda-in-C." In this part, we discuss the implementation of a compiler for our dialect, VAX Linda-C.

Introduction

Since VAX Linda-C is mainly VAX C, with some extensions, the simplest approach to compiling it is to preprocess VAX Linda-C code into VAX C code.

Preprocessors vary greatly in complexity and sophistication. Many—C's native preprocessor, for one—are text-to-text transformers having little or no understanding of the text they are processing. Others, such as RATFOR or C++, are essentially compilers which just happen to produce FORTRAN or C, rather than object code, as output.

`LCC`, the preprocessor for VAX Linda-C, falls somewhere in between. While `LCC` leaves all the C code in its input unchanged, its transformation of the Linda tuple operations requires that it be able to determine the types of arbitrary expressions. This in turn requires a full parse of the code, including the ability to understand declarators and expressions.

Overview of `LCC`

`LCC` must accept Linda-C programs as input and produce VAX C programs as output. However, as we discussed in Part II, Linda-C programs may use the `newtype` specifier to create new, globally-accessible types. The VAX C programs that `LCC` produces as the result of separate compilations must, when linked, resolve the various global new type references consistently.

Figure 1 illustrates the operation of `LCC`. In the Figure, three modules, `A.L`, `B.L` and `C.L`, form parts of a single Linda program. They are compiled into two distinct executable images, `AB.EXE` and `C.EXE`, which execute together.

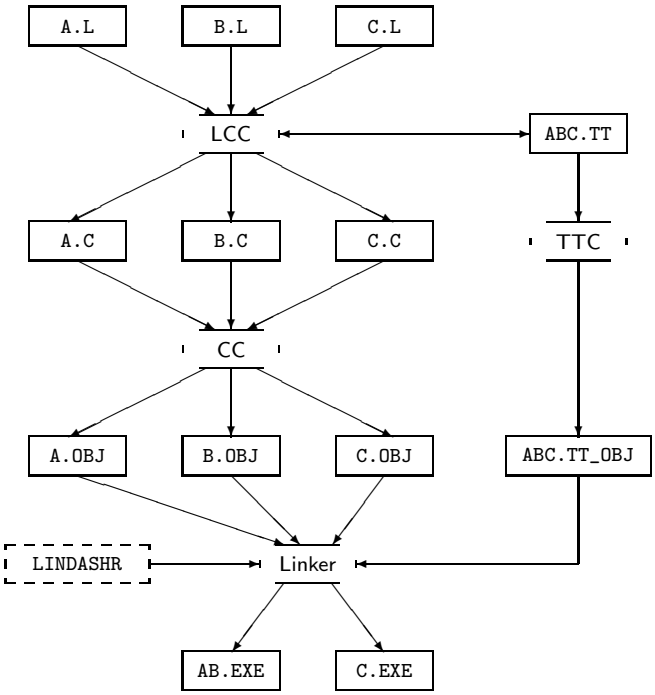


Figure 1: Overview of VAX LINDA-C Compilation

The Type-Tuple Context

In the Figure, the two images `AB.EXE` and `C.EXE` are able to communicate through tuple space because they share a common view of the type system and thus of tuples built upon it. Yet it's impossible for *all* LINDA-C programs ever written to share such a context. Independent programs will wish to use the same names for different types.

The *type-tuple context* within which a LINDA-C program is compiled defines the name space within which its globally-defined new types are known. Further, although this is not directly visible to the LINDA-C programmer, it also defines a set of numbers which uniquely identify the *tuple type signatures* which occur in all tuple operations within the program. The type signature of a tuple is simply the list of the types of its fields. Since matching depends on types, a tuple and template cannot match unless they have identical type signatures. A type-tuple context can extend across multiple source language modules. It can also extend across multiple executable images.

In Figure 1, the type-tuple context is stored in the file `ABC.TT`. Each `LCC` compilation reads the current version of this file and writes a new version containing new information gleaned from the current compilation.

The type-tuple context must also be available at run time. The `TTC` program

transforms the `ABC.TT` file into an object file, `ABC.TT_OBJ`, which is linked with each image that is to execute within the "`ABC`" context. We will see later that this object file actually contains executable code that will produce a run-time representation of the type-tuple information for use by the actual tuple operations.

Run-Time Support Code

Figure 1 shows one additional component, `LINDASHR`. This component is a shareable image containing the support code which implements and supports the tuple operations. There is another component as well, the LISTENER, which does not appear in the Figure because it is a stand-alone program that need not be linked into each LINDA-C program. The LISTENER implements the interface to an Ethernet, allowing a VAX LINDA program to consist of images running on multiple nodes on a local area network. We will not discuss the run-time code except in passing; a description can be found in a recent paper.[1] All the details can be found in our dissertation.[2]

LCC

The code in `LCC` descends from the front end of a C compiler developed by David Conroy. The original compiler supported standard C, essentially as defined in Kernighan and Ritchie, with a few minor extensions.

`LCC` requires a separate C preprocessor. Unfortunately, in the version of the compiler we worked with, VAX C's preprocessor could not be used separately.[3] Instead, we have used two different preprocessors with `LCC`. One, written by Martin Minow, is an independent implementation compatible with the proposed ANSI C standard. The other, `RPP`, is based on code provided to us by David Elins. `RPP` operates by running the VAX C compiler and requesting a listing file including the results of macro substitutions and conditional code and file inclusion. `RPP` then reads the listing file and extracts the preprocessed code.

A major goal of `LCC` is to accept, as a base, exactly the dialect of the C language that the VAX C compiler accepts. This allows most error messages to be generated by `LCC` in reference to the programmer's own code, rather than leaving that job to VAX C, which would refer to `LCC`'s output. Conversely, it ensures that correct VAX C programs are accepted. In developing `LCC` from Conroy's original code, we removed all of Conroy's extensions, minor as they were, and added code for the documented VAX C extensions to the language.

[1] "Implementing Linda For Distributed and Parallel Processing," in Proceedings of ICS89. Also available as Yale Department of Computer Science Technical Report TR-715, April 1989.

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

[3] Since Version 3.0, VAX C has provided a "preprocess only" option. Unfortunately, it appears to be impossible to get the preprocessor to retain comments.

To retain the compatibility between LCC and VAX C, the preprocessor of choice for LCC is RPP, which is inherently compatible with VAX C's preprocessing pass: It *is* VAX C's preprocessing pass!

The VAX C compiler is a moving target. At the time LCC was written, the current version was 2.1. VAX C has changed considerably since then. Starting with version 2.2, it began evolving toward compatibility with the proposed ANSI C standard—yet another moving target during the period in question.

The need for the programmer to interact with the generated code is often the weak spot of preprocessing systems. Sometimes it is possible to make such interactions very rare. In the case of LCC, we could not easily do that. Run-time error messages and debugging information are keyed to the text of the code eventually fed to VAX C for compilation. While it is possible to prevent this in principle, it is quite a complex matter.

Since LCC cannot hide its output from the programmer in all cases, it attempts to retain the structure of its input to the degree that it can. Indentation, spacing, and comments are copied unchanged. Where the correspondence between line numbers may not be one to one—where a file has been included or conditional code omitted, for example—a `#line` directive is inserted as a guide.[4]

The Organization of LCC

The usual way to implement a pre-compiler is to parse the input text into some internal form, typically a syntax tree; modify the tree; and finally convert it back to textual form on output. This approach makes it difficult to retain the textual structure of the input, and is not used by LCC. Instead, the tokenizer has the responsibility for maintaining the correspondence between the textual structures of the input and output.

The lexical analyzer was implemented "bare hands," rather than using a lexical analyzer generator. We don't know Conroy's reasons for doing this, but it turned out to provide a good framework into which we could insert our own code. It is not clear that the kind of processing we added could be easily added to code generated by a lexical analyzer generator.

Figure 2 is a simplified picture of LCC. LINDA-C program text enters at the left side; VAX C code exits at the right. The line running from the top of the tokenizer box represents a bypass around semantic processing; spacing and comments flow through it.

The box labeled `s` represents the "current" token. Actually, the box has two halves. The upper half is used only by the tokenizer; it contains the actual text of the token, exactly as it appeared in the input. The lower half contains the internal representation of the token, and corresponds to several variables whose use is pervasive within LCC. Most important of these is `s`, which contains an

[4]This also causes any VAX C error messages to use a line number corresponding to the original file. Unfortunately, it has no effect on run-time error reporting.

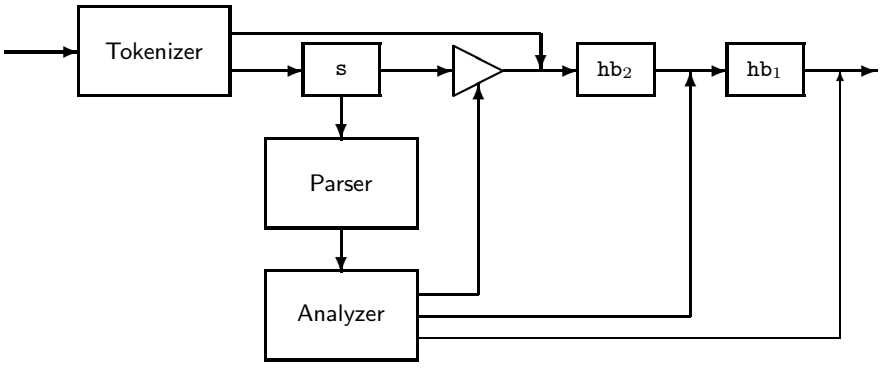


Figure 2: Overview of `LCC` Data Flow

internal code for the token. Other variables contain additional information for certain types of tokens, such as the value of an integer token.

The triangular shape to the right of the `s` box is a control point. Higher levels of the code may use it to discard the current token, or to insert text at the control point. Any such text appears in the output before the current token. In practice this mechanism is generally used to *replace* the current token.

Text flowing out of the control point is conceptually on its way to the output. However, another mechanism, the hold buffer, can intervene. Semantic processing routines may request a *hold* on the output by calling `hold`. This allocates a buffer (of unlimited size) and attaches it in front of the output stream. Figure 2 shows two hold buffers, though there can be any number. Usually, there are none. Hold buffers are stacked; hold buffer $\texttt{hb}_1$ in the figure was created by a call to `hold` made *before* the call that created $\texttt{hb}_2$.

Any text entering a hold buffer is retained there until `unhold` is called. At that point, the text is copied to the next hold buffer in sequence, if there is one, or to the output file.

When a hold buffer is active, the semantics processing code can insert text into the next hold buffer "outward." If there is no such hold buffer, the text flows directly to the output. This capability is represented in the Figure by the two lower arrows from the box labeled Analyzer. Because buffers nest, the lighter arrow at the bottom is inaccessible while the inner hold buffer, $\texttt{hb}_2$, is active.[5] The semantic routines can also erase the contents of the innermost hold buffer.

[5] Actually, `hold` returns a value that can be used to designate a particular buffer, regardless of subsequent nesting. This capability is not currently used.

Parsing and Semantic Analysis

Parsing at the statement level is by recursive descent, implemented directly as a series of C function calls. As they are recognized, expressions, declarators, and initializers are passed to special-case sub-parsers. Expressions are parsed using a precedence grammar. Declarators and initializers, with their idiosyncratic syntax and semantics, are parsed using algorithms to match. In flavor they are essentially recursive descent, however.

While Figure 2 shows separate Parser and Analyzer (semantic processing) boxes, in a recursive descent parser parsing and semantic processing are closely interwoven. There is no sharp line in the code corresponding to the two boxes, though some modules unambiguously belong in the Analyzer box; the constant-folding code is one example.

Common lore strongly recommends the use of parser generators, especially in the development of experimental compilers for languages whose grammar is subject to frequent change. Our experience is *not* consistent with this recommendation. We found the recursive descent parser in `LCC`–which, recall, was written by another programmer—extremely easy to understand and modify. Adding semantic actions and generating error messages was simple. Debugging was usually straightforward: The state of the parser is deducible from the point at which execution is taking place. In general, making modifications was very easy.

Of course, there is no denying that a parser generator would be useful in other circumstances—in checking a proposed grammar for ambiguities, if nothing else. Since we made only minor changes to C syntax, this was not an issue for us. For larger-scale changes, it might be important.

Parser generators are also claimed to produce faster code. That may be true, but in `LCC` parsing speed never became an issue. Since `LCC` was very fast—much faster than the VAX C compiler, for example[6] —we never saw any reason to do a detailed timing analysis. It seems highly likely, however, that the best way to speed up `LCC` is to re-implement the memory allocator, currently just a shell around `malloc` and `free`, not to begin reorganizing the parser.

The hold buffer mechanism's nesting action is consistent with the recursive descent parser's organization: When it is necessary to insert code dependent on a syntactic element *in front of* that element, one calls `hold` after recognizing the first token in the element, and `unhold` when the element is complete. The cleanest example can be found in a function, `readior`, which processes `in`, `out`, and `rd` statements into calls to the support routines `k_in` and `k_out`. The calling sequence for either of these functions makes use of the standard VAX representation of an argument list as a list of longwords, preceeded by a count. We call such a list, with the successive longwords containing a flag word; a type signature; a set of bits indicating the *polarities* of the fields, that is, which are formal and which actual; and then the fields of a tuple; the *external representation* of that tuple. By using a calling sequence to match, we

[6]Of course `LCC` doesn't have to generate code, an expensive operation.

```
hold();                         /* Hold field          */
if (readtyname(&attr))          /* A typename?         */
{       typnm = TRUE;           /* Remember that       */
        hberase();              /* Erase typename      */
        put('0');               /* Use 0 in its place  */
        .
        .
        .
}
else ...                        /* Non-typename field  */
unhold();                       /* Release field       */
```

Figure 3: Using the `hold` Function

get the compiler to build the external representation for us. A clever enough compiler might even discover that the calling sequence is, say, a loop invariant and construct it only once rather than at each iteration.

At least two of the first three arguments, and possibly all three, cannot be known until the entire tuple has been parsed. Hence, `readior` calls `hold` immediately after replacing the `in`, `out`, or `rd` token with the name of the appropriate function. It can then proceed to parse each field specification, building a record of field types and polarities.

It turns out to be necessary to call `hold` again for each field, since a field consisting entirely of a type name is replaced with a constant 0. While this could perhaps be accomplished by passing additional state information to the `readtyname` function, which reads type names, using `hold` produces cleaner code, as shown in Figure 3. The Figure has been considerably simplified by omitting all the actual semantic routines, but it is an accurate illustration of the code layout. The code in the Figure is, in effect, treating the entire type name as a large token—a natural thing to do in a recursive descent parser.

The Type-Tuple File

In order to support separate compilation, `LCC` must maintain the type-tuple file described earlier. Code within `LCC` maintains a secondary symbol table of Linda types and tuple signatures. The table is initialized with some built-in values and the data in the old type-tuple file. It is updated as new types are declared and tuple operations are performed. Finally, a new version of the file is written.

Type-tuple files are simple ASCII files—a more compact notation did not seem called for, as they are likely to be small. The files are line oriented, with each line specifying one of six operators, along with values to go with it.

LCC At Work

To get a feel for what LCC does, we illustrate its operation on a simple example program. The program consists of master and worker processes, implemented in two separate files. The master places a request tuple in tuple space; the worker eventually replies with a response tuple containing the factorial of the integer in the request tuple. There is nothing inherently interesting about the program. It was constructed to illustrate how LCC handles various constructions.

The figures on the following pages show the input code, the resulting C code and type-tuple file produced by LCC, and finally the C code generated by TTC.

Translating From VAX Linda-C To VAX C

Lines 1–12 of Figure 5 were created by LCC. Line 10, which describes the `nchar` type, is completely synthetic. No such declaration ever appears in the input to LCC: `nchar` needs special handling since string literals have type "array of `nchar`."

Line 11 makes the run-time type system available to the code.[7] Both the arrays declared here map a tuple type index to a tuple type number. The tuple type index is a value that the compiler can determine, while the tuple type number will not be known until run time. `LINDA_TU` maps "global tuple types:" The types of tuples all of whose fields have globally-known types. `LINDA_TU_0` maps tuple types at least one of whose fields is local to this module. The trailing zero is a module number; the corresponding line in Figure 7 would declare `LINDA_TU_1` instead. As it happens, `FACTMASTER.L` contains no local tuples, or types for that matter, so `LINDA_TU_0` is never used.

Starting with line 13, Figure 5 is essentially a line by line translation of Figure 4. Note how LCC preserves the format of the input file to the degree that it can. Where it has rewritten the input text, LCC often leaves the original text in the line as a comment. The `newtype` declarations on lines 3–5 of Figure 4, corresponding to lines 13–15 in Figure 5, illustrate this practice. Because including both input and output text causes the output lines to become unwieldy, LCC does not follow this practice for the Linda operations. Enough of the line is left unchanged that it is usually easy to understand.

An interesting special case can be seen in the lines just mentioned: A `newtype` declaration is generally translated into a `typedef`. However, for no particularly good reason, many C compilers, VAX C included, do not permit a declaration of the form:

```
typedef void VOID;
```

LCC must therefore do something different when a `void` type is declared. One possibility would be to declare the type as something else, say an `int`. However,

[7] The `globalref` storage class is a VAX C extension which is analogous to the `extern`. A corresponding definition is made using `globaldef`, examples of which appear in Figure 9. We used the VAX C extensions only because it makes generating the code in the latter Figure simpler—there is no analogue to `globaldef` for externals.

```
1  #ttcontext factorial
2
3  newtype void FACTORIAL;
4  newtype void FACTORIAL_RESULT;
5  newtype int PUBLIC;
6
7  LindaMain()
8  {   PUBLIC      n;
9      int          result;
10
11     n = 6;
12     out(FACTORIAL,n);
13     in(FACTORIAL_RESULT,?PUBLIC,?result);
14     printf("6! = %d\n",result);
15 }
```

Figure 4: File FACTMASTER.L

```
1  /* Created by lcc version 00.01,  9-AUG-1988 19:19:54.18 */
2  /* Source file USERCS:[LEICHTER.THESIS]FACTMASTER.P;1 */
3  # 1 "USERCS:[LEICHTER.THESIS]FACTMASTER.L;5"
4  # 2
5
6  #module FACTMASTER
7  # 3
8
9  # 1 LINDA_LCC_PREAMBLE
10 /* newtype */typedef char /*varying(void) */ nchar;
11 globalref short LINDA_TU[], LINDA_TU_0[];
12 # 3 "USERCS:[LEICHTER.THESIS]FACTMASTER.L;5"
13 /* newtype void FACTORIAL; */
14 /* newtype void FACTORIAL_RESULT; */
15 /* newtype */typedef int PUBLIC;
16
17 LindaMain()
18 {   PUBLIC     n;
19     int         result;
20
21     n = 6;
22     k_out(&0x40002,LINDA_TU[0],&0xFFFFFFFF,0,n);
23     k_in(&0x2,LINDA_TU[1],&0xFFFFFFF9,0,0,&(result));
24     printf("6! = %d\n",result);
25 }
```

Figure 5: File FACTMASTER.C

```
#ttcontext factorial

newtype void FACTORIAL;
newtype void FACTORIAL_RESULT;
newtype int PUBLIC;
/*
 * Compute factorial:  <FACTORIAL,n> => <FACTORIAL_RESULT,n,n!>.
 * In both cases, n has type PUBLIC; n! has type int.
 */
LindaMain()
{   newtype void PRIVATE;
    int         n, p;
    PUBLIC      input;

    /*
     * The outer loop reads request tuples, puts them into private
     * format with an initial p value, and sends them on.
     */
    for (;;)
    {   in(FACTORIAL,?input);
        out(PRIVATE,(int)input,1);

        /*
         * This loop does the calculation:  We compute a recursive
         * f(n) by computing F(n,p); p accumulates the result.
         */
        for(;;)
        {   in(PRIVATE,?n,?p);
            if (n <= 1)
            {  out(FACTORIAL_RESULT,input,p);
               break;
            }
            else
               out(PRIVATE,n-1,p*n);
        }
    }
}
```

Figure 6: File FACTWORKER.L

⋮

Fifteen lines omitted; see text.

⋮

```
1  /*
2   * Compute factorial:  <FACTORIAL,n> => <FACTORIAL_RESULT,n,n!>.
3   * In both cases, n has type PUBLIC; n! has type int.
4   */
5  LindaMain()
6  {   /* newtype void PRIVATE; */
7      int          n, p;
8      PUBLIC       input;
9
10     /*
11      * The outer loop reads request tuples, puts them into private
12      * format with an initial p value, and sends them on.
13      */
14     for (;;)
15     {   k_in(&0x2,LINDA_TU[0],&0xFFFFFFFD,0,&(input));
16         k_out(&0x80002,LINDA_TU_1[0],&0xFFFFFFFF,0,(int)input,1);
17
18         /*
19          * This loop does the calculation:  We compute a recursive
20          * f(n) by computing F(n,p); p accumulates the result.
21          */
22         for(;;)
23         {   k_in(&0x2,LINDA_TU_1[0],&0xFFFFFFF9,0,&(n),&(p));
24             if (n <= 1)
25             {  k_out(&0x80002,LINDA_TU[1],&0xFFFFFFFF,0,input,p);
26                break;
27             }
28             else
29                k_out(&0x80002,LINDA_TU_1[0],&0xFFFFFFFF,0,n-1,p*n);
30         }
31     }
32 }
```

Figure 7: File `FACTWORKER.C`

```
1  ! TT file created by lcc version 00.01,  9-AUG-1988 19:19:54.18
2  ! TT file created by lcc version 00.01,  9-AUG-1988 19:20:03.80
3  P 5 2 1 20 0
4  T 1 0 13 18   0 FACTORIAL
5  T 2 0 13 17   0 FACTORIAL_RESULT
6  T 3 0 5 16   4 PUBLIC
7  T 4 0 5 5   4
8  T 0 C000 1 15   1 nchar
9  U 0 2 1:A 3:1
10 U 1 3 2:A 3:A 4:A
11 M FACTWORKER 1 1 1
12 T 0 0 13 20   0
13 U 0 3 80000000:A 4:B 4:F
14 M FACTMASTER 0 0 0
15 E
```

Figure 8: File `FACTORIAL.TT`

since no objects can be declared with a void type, and the `void` type itself will never appear in a translated Linda operation, `LCC` can take the simpler approach of just omitting the declaration from the output entirely.

The first argument to `k_out` or `k_in` contains some flags and possibly a length field when the compiler can determine the total length of the tuple. All the Linda operations in the examples have tuple lengths that `LCC` can compute, so the bottom hex digit of the value passed in the generated calls always has the bit with value two set to indicate that the high-order 16-bit value (the high-order four hex digits) contains a length. For example, the `out` on line 12 of Figure 4 creates a tuple containing a 0-byte `void` field and a 4-byte `int`. The resulting length of 4 bytes can be seen in the first argument in the corresponding call to `k_out`, line 22 of Figure 5.

In all cases, the second argument to `k_in` or `k_out` is a member of `LINDA_TU` or `LINDA_TU_xx`, where `xx` is the current module's number. An example of the use of a local tuple type can be seen in line 21 of Figure 6, which translates to line 16 of Figure 7.

The third argument to `k_in` or `k_out` is a set of bits, one per field, indicating whether the corresponding field is formal or actual. Using a VAX C extension, it always appears as the address of a constant.

The remaining arguments are straightforward translations of the input fields specified. Where the input field is simply a typename, a value of zero is passed. For a formal passed to `k_in`, this will lead the binding code to skip the field.

The Type-Tuple File

Figure 8 illustrates the type-tuple file produced by `LCC` after compiling the code in Figures 4 and 6, in that order. Line 3 declares several global parameters,

such as the number of distinct global types and tuples seen so far. It also marks the beginning of a series of records that define global types and tuples.

The `T` records that follow declare global types. The information stored for a global type includes its type index, which once assigned can never change; the underlying "basic type," which essentially corresponds to one of the scalar types in C; modifiers, such as "pointer to" or "array of;" the length; and the name, if any. Lines 4–6 declare the global types created as a result of the `newtype` declarations in the code; line 7 declares a type that corresponds to `int`, needed for the result tuple. Note that this type is anonymous. However, `LCC` can determine that it is a built-in type from additional information on the line.

In general, `LCC` creates declarations in the type-tuple file only for types it actually needs. However, `nchar` is again an exception: A declaration is always created for it. Line 8 in the Figure is an example. (The third field, `C000`, is the hexadecimal representation of the `varying(void)` attribute.)

The two `U` records on lines 9 and 10 declare global tuple types. A `U` record consists of a tuple type index, an arity, and a set of field values. Each field value is a type index, a colon, and a set of flags indicating how this field is used.

An `M` record, such as line 11, introduces a module. Each module compiled in a particular type-tuple context will be represented by a single `M` record in the type-tuple file. Any local types or tuples are declared following the `M` record, using the same `T` and `U` records we have already examined. The only novelty can be seen in line 13, which declares the tuple type corresponding to the tuple operations on lines 21, 28, and 34 in Figure 6. Since a local tuple type can contain fields with a mixture of local and global types, it is necessary to mark the fields appropriately. The top bit of the type index, a 16-bit value stored in a 32-bit integer, is used for this purpose. Hence, the first field of the tuple being declared is *local* type 0, which corresponds to `PRIVATE` in the source code, and not *global* type 0, which corresponds to `nchar`.

As `LCC` reads a type-tuple file it discards the previous `M` record for the module being compiled and all declarations associated with it. It re-creates the local records, and all the global records, from its internal symbol tables. The `M` records and associated declarations for other modules are copied unchanged.

`TTC`'s Translation of the Type-Tuple File

Figure 9 shows the result of applying `TTC` to the type-tuple file of Figure 8. The code consists of a basic shell within which a series of declarations and calls to the run-time type system are inserted. The call to `Linda_init` initializes the run-time support code. It also parses, interprets, and removes from the `argv` vector any command line options intended for it.

Line 10 creates the `LINDA_TU` vector used to map global tuple indices to tuple type numbers. It is initialized by lines 22–25. First, however, run-time types must be created to pass to the `NewTupleType` function. This is done in lines 13–21. All these lines are essentially direct translations of lines in the

```
/* Created by ttc version 00.01,  9-AUG-1988 19:20:47.23 */
/* TT file USERCS:[LEICHTER.THESIS]FACTORIAL.TT;2 */
#module LINDA_MAIN_PROGRAM
#include "LINDA_LIBRARY:linda.h"
main(argc,argv,envp)
int argc;
char **argv;
char *envp;
{       globaldef TYPE LINDA_T[5];
        globaldef TYPE LINDA_TU[2];

        argc = Linda_init(argc,argv,envp);
        LINDA_T[1] = NewType(TYPE_IMMEDIATE|TYPE_FIXED,0,
                        "FACTORIAL");
        LINDA_T[2] = NewType(TYPE_IMMEDIATE|TYPE_FIXED,0,
                        "FACTORIAL_RESULT");
        LINDA_T[3] = NewType(TYPE_IMMEDIATE|TYPE_FIXED,4,
                        "PUBLIC");
        LINDA_T[4] = NewType(TYPE_IMMEDIATE|TYPE_FIXED,4,"");
        LINDA_T[0] = NewType(TYPE_IMMEDIATE|TYPE_FIXED,1,
                        "nchar");
        LINDA_TU[0] = NewTupleType(
                                LINDA_T[1],LINDA_T[3]);
        LINDA_TU[1] = NewTupleType(
                                LINDA_T[2],LINDA_T[3],LINDA_T[4]);
        {       /*** Module FACTWORKER ***/
                TYPE LINDA_T_[1];
                globaldef TYPE LINDA_TU_1[1];

                LINDA_T_[0] = NewType(
                        TYPE_IMMEDIATE|TYPE_FIXED, 0,"");
                LINDA_TU_1[0] = NewTupleType(
                        LINDA_T_[0],LINDA_T[4],LINDA_T[4]);
        }
        {       /*** Module FACTMASTER ***/
                TYPE LINDA_T_[1];
                globaldef TYPE LINDA_TU_0[1];

        }
        return(LindaMain(argc,argv,envp));
}
```

Figure 9: File FACTORIAL.TT_C

type-tuple file. However, where the type-tuple file was language independent, the output of `TTC` is specific to C. For example, the type-tuple file encodes the fact that `PUBLIC` corresponds to a four-byte integer `TTC`, in generating lines 17 and 18 of Figure 9, adds the information that, in C, such objects are passed by immediate value.

Each `M` record, together with the declarations that follow it, translates to a block of code in `TTC`'s output. The block creates the `LINDA_TU_xx` vectors that individual modules use to map local tuple type indices to tuple type numbers.

Local types are stored in vectors local to each module block, since they will never be needed again once the block is complete. Global types are stored in the globally-known vector `LINDA_T`. This is a historical artifact—no reference to `LINDA_T` is ever made outside of the code generated by `TTC`.

Conclusions

We consider our implementation for `LCC` as generally successful. While ideally we would prefer a full compiler rather than a preprocessor, the code generated by `LCC` is generally easy to understand when necessary.

We've also found that `LCC` itself is reliable and easy to modify and extend. Much of the credit must go to David Conroy, who defined the overall structure of the code. While we were working on `LCC`, we watched others painfully making analogous changes to a LEX- and YACC-based compiler. We came away convinced of the virtues of a carefully designed hand-built lexer and, even more so, of a recursive-descent parser.

Availability

Our implementation of Linda-C for VAXes has been made available to several laboratories on an experimental basis. We are beginning the development of a commercial version and hope for an initial release early in 1990. Contact the author for further information.

Jerrold Leichter has recently completed his doctoral research, which includes an implementation of Linda for shared-memory and networked VAXes, at the Yale University Department of Computer Science. He was a long-time employee of Digital Equipment Corporation, whose Graduate Engineering Education Program supported him during some of his work. He may be reached electronically as `leichter-jerry@cs.yale.edu`*; at 24 Old Orchard Lane, Stamford CT 06903; or by phoning (203) 329 0921.*

∞

28. Electronic Survey Number 2

Compiled by **Rex Jaeschke**

Introduction

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

The following six questions were posed to 52 different people, with 14 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.

Expanding Macros in Pragmas

Do you expand macros in pragmas? If not, any reason not to?

- 3 – Yes
- 9 – No
- Comments:

    1. No. First, the pANS implies that this does not occur. (§3.8, page 87, lines 15–16.) Second, if macros were allowed to "monkey" with `#pragma` directive contents, they wouldn't be as safe to use. In other words, something like

        ```
        #pragma __DATE__ = "May 5 1887"
        ```

        would be unsafe if macro replacement occurs.
    2. I ignore pragmas. If and when I recognize them, I will expand macros in them.
    3. No. We only have one pragma.
    4. Currently no. See no reason not to though.
    5. Not yet but we plan on doing this in the next release.

6. No. Pragmas are currently ignored.
7. I do expand macros in pragmas. I think unrecognized pragmas should elicit a warning, and recognized ones should elicit one at the user's request. This would prevent strange things from happening when pragmas and macros are mixed. I don't think doing macro expansion makes pragmas any more dangerous then they already are.
8. I think they should be expanded, to allow convenient configuration parameterization. For example, Orca/C has a pragma for specifying the slot number in the Apple IIGS in which a floating-point accelerator is installed. This obviously should be parameterizable.
9. The fact is that we do currently expand macros in pragmas and have been thinking of it as a desirable feature. For reasons I won't go into, however, one of our compiler labs recently suggested that we should stop doing it and, upon investigation, I came to the conclusion that our current behavior is wrong. ANSI §3.8: "The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated."

   So to me it looks as though macro expansion in pragmas is clearly a non-conforming thing to do. Do you know of any arguments to the contrary? I am curious why the question came up on your survey. I haven't been on X3J11 long enough to know whether this subject was debated there.

   I came up with one reason not to expand. Suppose I have written a program which uses:

   ```
   #define OFF 0
   #define ON  1
   ```

   Now I want to port to a system where, in order to invoke the optimizer on a per-function basis, I have to write:

   ```
   #pragma OPTIMIZE ON
   ```

   If I want to use this pragma, and if macros are expanded in pragmas, I have a problem.

   I would appreciate any light you can shed on the question, since we plan to make a decision soon on whether to change our behavior.

Predefined Macros

What predefined macros do you have? What is their purpose? I'm mostly interested in those that conform with regard to namespace requirements (i.e., begin with __ or such).

`_ADDR32`	`__FUNCTION__`
`_CRAY`	`__GNUC__`
`_CRAY1`	`__GNU_LIBRARY__`
`_CRAY2`	`__m88000__`
`_KCCsymfnd(a, b)`	`__m88k__`
`_KCCsymval(a, b)`	`__mc68000__`
`_MV`	`__ORCAC__`
`_RELEASE`	`__RAND__`
`_UNICOS`	`__sparc__`
`__`*company*`__`	`__STRICT_ANSI__`
`__`*machine*	`__unix`
`__CLASSIFY_TYPE__`	`__unix__`
`__COMPILER_KCC__`	`__vax__`
`__DGUX__`	`__ZTC__`

- Comments:

  1. None of these names (other than those in the standard) are provided for the user. A different method is used for machine/system preprocessing inquiry.
  2. GCC always predefines `__GNUC__`. If you compile in strict-ANSI mode (in which extensions which conflict with the Standard are turned off), `__STRICT_ANSI__` is predefined. `__GNUC__` is intended to be used to test for the presence of GCC extensions. The macro `__STRICT_ANSI__` is tested by the headers of the GNU C Library; they don't define any non-ANSI macros, types, or functions if it is defined. `__STDC__` is defined as 1 except in traditional mode (in which `const` and `volatile` are not recognized, etc.). Various macros such as `__unix__`, `__vax__`, `__sparc__`, `__mc68000__`, etc., are predefined depending on machine and OS. Forms of the same macros without the leading and trailing underscores are defined except in strict-ANSI mode.

     The headers of the GNU C Library define `__GNU_LIBRARY__`. Several other macros are defined and/or tested by the headers of the GNU C Library to determine what to declare (from ANSI, POSIX, BSD, System V, miscellaneous historical Unix things, and GNU extensions). These should not generally be of interest to application writers. There are some feature-test macros that an application writer may define to determine what will be declared:

`_POSIX_SOURCE`	IEEE Std 1003.1 only
`_BSD_SOURCE`	All except GNU, with conflicts in favor of 4.3BSD
`_SVID_SOURCE`	All except GNU, with conflicts in favor of the SVID
`_GNU_SOURCE`	All plus GNU extensions, with conflicts in favor of POSIX

3. We predefine `__`*machine* and `__unix` as well as some flag specific macros (i.e., macros are defined when certain flags are turned on to allow conditional compilation based on presence/absence of certain features).
4. I have a bunch of macros that tell the user things about the mode he's in (which has been selected with command line options or pragmas). The C compiler I've written supports parallel extensions for transputers, and has to deal with various transputer models and programming models.

`auto` Allocation Checking

Do you have some sort of a 'stack probe' to check that space is actually available for automatic objects before allocating them? If so, what happens on allocation failure? Can this probe be disabled?

- 6 – Yes (4 can turn it off)
- 6 – No
- Comments:

1. Yes. If no stack space is available the program aborts (not very nicely), but we would have to run out of memory completely before that would happen. No, the stack overflow check cannot be disabled unless you write in assembly.
2. Well, there are two variants depending on whether the code (on a PDP-10) is using extended addressing or not. If not, then the LH of the stack pointer is a count, and the hardware automatically generates a stack overflow interrupt when this count runs out (so subroutine calls could trigger this as well as `auto` allocation). But if extended, there is no count, so what I do instead is put a read-only page at the end of stack space so that any attempt to store stuff there will trigger a memory protection interrupt. The stack overflow interrupt can be disabled with an appropriate `signal()`; the memory protection one cannot. But there really isn't any point in doing so,

although I suppose someone might want to `longjmp()` back up the stack—nothing else would be able to recover.

3. No. Given that the same compiler is used for stand-alone and hosted environments, there is no general way to do this. In a hosted environment, it is assumed that the operating system mediates when the stack overflows.
4. Yes. The program is aborted with a message on failure. It can be disabled (it is off by default).
5. The GNU system is intended for virtual memory machines, so this is not necessary.
6. You get a signal when you try to access the unavailable memory.
7. Yes, upon entry to the function a check is made to see if sufficient space is available. If not enough space exists the program terminates abnormally and a "Stack overflow" message is issued. No it cannot be disabled. Another interesting aspect of our stack is that it might be non-contiguous. This is our way of allowing multiple processors to allocate stack space simultaneously.
8. The instructions that allocate stack space check against a stack limit. The check can be disabled, but the language runtimes (notably malloc) might not work, since they adjust the stack limit downwards to allocate more memory for the heap.
9. The stack probe is enabled by default. (Transputers are kind of strange, with their multiple processes in the same heap.) Can be disabled, though, for efficiency reasons. This is potentially dangerous, though. When the process runs out of stack space, a new stack is allocated for this function and everything it calls. Whenever this happens, it should be flagged somehow, so the programmer can find out about it, so he knows things are less than optimal.
10. In a version of Ritchie's PDP-11 C compiler, I forced the stack to grow immediately upon entry to the function, rather than relying on segmentation violation handling by the OS when the actual data access occurred. This was because certain instructions could not be properly restarted from the context information available to the OS.

Semantics of `register`

Do you take notice of `register`*?*

- Comments:

1. Yes, but only for the purpose of checking for `&`. All `register` variables are actually `auto`. If this ever changes, there would probably

be about 10 word registers, with any type up to 2 words long allowed. Actual allocation would hopefully depend on the optimizer making a judgment call as to which variables were used most often.

2. The compilers I maintain are all based on PCC and, as such, allocate a batch of registers reserved for programmer-denoted `register` variables, reserving another batch for compiler-generated temporaries. Generally only word-sized (`int`, pointer) or smaller data actually get registers this way.
3. The only effect `register` has is to cause error messages if you take the address of a `register` variable. Actual register allocation is done by coloring, as many as will fit. Usage counts are done of variables to determine priority for getting in registers. Usages inside loops are multiplied by 10.
4. When not optimizing, GCC obeys `register` declarations. When optimizing, GCC can figure this out much better than the programmer. With the exception of the MIPS C compiler (for the MIPS processor only), GCC produces better code than almost all other C compilers.
5. We pay attention to `register` if and only if the function has `asm` statements. In other circumstances we perform global register allocation and instruction scheduling which is more productive than listening to programmer `register` suggestions.
6. We only notice `register` to prevent the variable's address from being taken. We attempt very aggressive register assignment and believe that the compiler can do a better job than the programmer of deciding which variables should be assigned to a register.
7. Our compilers do use `register`. The old ones use it alone, the newer ones use such a declaration as one more bit of information about whether to registerize an object.
8. Yes, if not optimizing (optimizing throws away the `register` hint, and uses flow analysis to determine what goes in a register).
9. No. If supported at all, the compiler should find out about often used objects, and only take the `register` keyword as a hint. Transputers don't really have registers in the traditional sense. It is possible to have a better workspace (stack frame) arrangement by looking at frequency of use of objects, though. On transputers, it's not important enough to make a big fuss about.
10. Yes we take notice of `register`. The number of registers available is different depending on the code situation. All registers are the same size as `sizeof(int)` (which `== sizeof(long) == sizeof(short) == sizeof(double)`).

If so, how many registers do you make available and what is their size?

- Comments:

  1. However many registers the machine has and are not used are available for variables. The size is whatever the machine has; all of the machines that GCC currently runs on have 32-bit registers.
  2. 13 32-bit registers that are preserved across calls, 10 32-bit registers that are not preserved across calls, and 2–3 32-bit additional registers will be available soon.
  3. Objects of sizes 1, 2, 4, and 8 can be stored in registers. On some machines, objects of size 16 could be stored in them.

Can an object be stored in multiple registers?

- 3 – Yes

- Comments:

  1. The only use of multiple registers is that forced by the architecture. For example, two adjacent 16-bit registers can hold a long.

What types can actually be stored?

- Comments:

  1. Whatever will fit.
  2. All types whose size `<= sizeof(`register`)` are candidates for storing in registers.
  3. 32- and 64-bit integer and floating-point, and soon, 1 and 2 word structures.
  4. The available types is machine-specific. Basically, if a machine has registers (some of our targets do not), all types that are usable as registers with the instruction set can be so allocated.

If there are more requests than you have registers, what is your allocation strategy?

- Comments:

  1. The most frequently used variables go in registers.
  2. Linear allocation for registers needed for a single basic block. I think graph coloring is used for registers whose lifetimes span more than one basic block.
  3. Extra `register` requests are ignored (first few, typically 3 to 5, are honored).
  4. The compilers that only use the keyword for allocation do exactly what Hansberry [the thorn in X3J11's side] wanted: lexical ordering takes precedence. The others use heuristic schemes.

`auto` Allocation Timing

Do you allocate automatic storage on entry to each block or do it all on entry to the function?

- 12 – On entry to the function.
- Comments:

  1. It is very difficult to allocate `auto` storage on entry to each block: a `goto`, `switch`, or `longjmp` may jump into the middle of a block, bypassing such allocation code. The only implementation I have used that tried to do this got it wrong.
  2. GCC supports dynamically allocated local storage via `alloca` or through variable dimensioned arrays. In these cases, the space as allocated at the call site.
  3. All on entry. Simpler and more efficient. Optimization nibbles away at this from both the entry and exit points of a function, though, since there are instructions that atomically push and pop words.
  4. We allocate it all upon entrance to the function. I've never figured out how it could be done any other way because `goto` and `setjmp/longjmp` make this a nightmare. I'd be curious about an implementation that solved this problem.
  5. All done upon entry to the function (though `alloca` and dynamic sized arrays are available to dynamically allocate automatic storage.
  6. I used to do it on entry to each block (or even expression, if it needed temp space), but decided the act of (de)allocating took more time than what I gained by having shorter offsets.
  7. All upon function entry. I would hope that internal block use is free, because it's widely used just to delimit the scope of identifiers and to aid in reuse of `register`s.
  8. One addition to the stack pointer on entry, one on return. Some of the machines make any other approach almost impossible.

Case Ranges

If you support case ranges, what syntax do you use? If you don't, can you point to any prior art here?

- 12 – Not supported.
- Comments:

  1. If a common syntax is decided upon (by ANSI or practice), I'd consider supporting it.

2. No. As a programmer I'd like the extra functionality, but as an implementer it gives me a headache.
3. I don't have case ranges available. I would think anyone who added them would use

   `case` *const-int-exp* `..` *const-int-exp*`:`

   due to Pascal's prior art.
4. Certainly, we on X3J11 have seen a number of suggestions, some that conflict with other language enhancements (such as C++'s `::` operator).

Future Polls

Some of the topics planned for future polls are:

- How do you define `NULL`? Is the null constant pointer actually represented as all-bits-zero? What different internal pointer representations do you support?
- Assuming you implement most of the goodies required/defined by ANSI C, how important is provable ANSI-conformance to your market place? Not at all, somewhat, absolutely necessary.
- Do you have a home-grown or commercial validation suite? If so, which?
- Is POSIX conformance an issue for you? What about IEEE support?
- What do you see as the biggest shortcoming of the ANSI C Standard, as a language standard or in some missing functionality (in the library or preprocessor, for example)?
- Do you or will you implement `long double` with a different representation than `double`? If so, will that make three different floating-point representations or are `float` and `double` mapped the same?
- Have you implemented or do you plan to implement locales other than the standard `"C"`? If so, for what purpose?

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

∞

29. Understanding Expressions in C

Jim Brodie

Abstract

In this paper, I will attempt to provide a consistent and complete set of rules which can be used to understand and decode complex C expressions. This is necessary if you are going to provide effective documentation on how C works. You need a framework which does not rely on black magic, hand waving, and "Well, you know" during the discussions of the complex or difficult cases. For the most part my discussion will cover the expression attributes: type, value, and class; and the concept of rvalues, lvalues, and their conversion.

Introduction

Expressions are at the heart of any C program. (All computations in a C program are expressed by writing expressions.) Expressions appear in a wide variety of contexts. (They range from initializers to the argument lists for function calls.) So users need a firm foundation if they are to analyze complex expressions in a methodical, reliable. It turns out that the real complexities in C expressions almost always have to do directly or indirectly with pointers. When is the result of an expression (or subexpression) the address of an object and when is the result the value of the contents of an object?

One of the primary attempts to deal with some of the difficulties of complex expressions was the introduction of the concepts of *rvalues* and *lvalues*. Any moderately experienced C programmer is familiar with the simple application of the rvalue and lvalue concepts. The origin of these names goes back to the assignment operator. The expression `x = y` means to take the value from the data object designated by the subexpression `y` and store it into the data object designated by the subexpression `x`. In the left hand (lvalue) context of the an assignment operator, the expression is used to designate an object. In the right hand context (rvalue) context the expression is used to access the stored value. Unfortunately, when we move away from the simple case of assignment, the use of the lvalue and rvalue concepts have become less clear. In fact, much of the confusion about complex expressions can be traced to insufficient information in these non-assignment uses of the lvalue and rvalue concepts.

In the remainder of this article I will present the rules, based upon the proposed ANSI standard for C, which will allow you to document and use

the concepts of lvalues and rvalues so that you can understand even the most complex expressions.

Simple Expressions

Let's start with a little background. Although there are specific restrictions which apply in some contexts, the general form of expressions is the same everywhere in a C program. An expression is made up of one or more operands and zero or more operators.

When you write an expression, with only a few exceptions, the operands are also expressions. (The primary example of a context where an expression is not allowed is the right hand argument to the select member operator, "`.`". The right hand argument is limited to a structure member name. This limitation is the reason you cannot dynamically select which member of a structure you will reference.)

The simplest form of an expression which can appear as an operand is called a *term*. The terms which can appear in a C program, along with a simple example of each, are shown below.

Terms	
Kind of Term	*Example*
name	`main`
string literal	`"my string"`
integer constant	`123`
floating-point constant	`3.14`
character constant	`'a'`
parenthesized expression	`(a + b)`
`sizeof(`*declaration*`)`	`sizeof(int)`

The only unusual thing here (compared to most presentations) is the inclusion of the `sizeof(`*declaration*`)` case. You can think of this as a way of writing an integer constant which is determined by the C translator, using implementation-specific rules.

Expression Classes

Despite having the same general form, there are actually four *classes* of expressions in C.

1. Expressions whose results designate data objects – These are the lvalue expressions. After these declarations:

```
long int li;
float *pf;
```

the expressions `li` and `*pf` are lvalue expressions. A string, such as `"Hello world"` is also an lvalue expression. The name used to designate a data object *always* represents the address of the data object. It is always an lvalue expression.

When an lvalue expression is evaluated the result is an lvalue. Note here that we make a distinction between the expression and the result of the expression. The expression always has the same class, given the declarations which were used to declare the names which appear in the expression.

The result, because of automatic conversions performed by the C translator, may change class during the course of the evaluation of an expression. There are several rules, which we will discuss later, which describe the conversion from the address of a data object (the lvalue) to the value stored in the data object (the rvalue). Whether these rules are applied, however, is based upon context (the operator this operand is being used with).

2. Expressions whose results designate functions – These are called, cleverly enough, function designator expressions. For example, after these declarations:

```
void func(int);
int (*pfunc)(double);
```

the expressions `func` and `*pfunc` are function designator expressions.

3. Expressions which result in values that do not designate a data object or a function – These are the rvalue expressions. Constants are examples of rvalue expressions. For example, `1234`, `'a'`, and `3.14` are all rvalue expressions. In addition, the general form `sizeof(`*declaration*`)`, as in `sizeof(int)`, is an rvalue expression. The result of an rvalue expression is an rvalue. All of the arithmetic operators form expressions which result in rvalues. For example, after this declaration:

```
int abc, def;
```

the expression `abc + def` produces an rvalue result.

4. Void expressions – These are unusual in that they do not designate any value, data object, or function. They are specified only to cause a set of instructions to be performed. With only three exceptions (involving the comma, and conditional, operators and void casts), void expression

results cannot participate as operand expressions. For example, after these declarations:

```
void func(int);
void (*pfunc)(double);
```

the expressions `func(22)` and `(*pfunc)(3.14)` are void expressions.

Other Expression Attributes

In addition to class, each expression has an associated type. Type information is used by the translator to determine which instructions to use when translating the expression to machine language. The type is also used by the translator to ensure that the only valid operations are specified in the program. For example, the type information is used by the translator to diagnose invalid operations such as the addition of two structures.

An lvalue, function designator, rvalue, or void expression result inherits the class and type from the expression that was used to generate it. The one exception is that an rvalue result always drops any type qualifiers from its type. For example, after this declaration:

```
const int i = 5, j = 7;
```

the type of the rvalue result of the expression `j + i` is simply `int`—it is *not* `const int`.

Each expression result, other than void, also has an additional attribute beyond type and class.

An rvalue result has an associated value attribute. This attribute holds the value calculated when the expression is evaluated. Depending upon the form of the expression used to generate the rvalue, this attribute may take on a different value each time the expression is evaluated during program execution. For example, the value attribute of an expression such as `abc + def * i` will depend on the current values stored in each of the designated data objects. On the other hand an expression such as `762 + 97 * 19` can be evaluated at translation time and will not change during the execution of the program. Any expression whose resulting value can be determined, once and for all, at translation time is called a constant expression. (`762 + 97 * 19` is an example.)

An lvalue result has an address attribute associated with it. This attribute holds the address of the data object which it designates. A function designator result also has an address attribute associated with it. This attribute holds the address of the function which it designates. Once you understand that the result of an expression is actually made up of three components (class, type, and value or address (if not void)) then you can start to make sense out of the rules for how expressions can be constructed in C.

Each operator in C has a specific set of rules which limit the class and types of the operands that it accepts. Each operator then produces a result, when it is evaluated, which has a class, type, and value or address (if not void).

Most of you are probably already fairly familiar with the type restrictions for operators and how a value or address is generated from an expression such as `5 * i`. However, you may be unfamiliar with the expression class rules. The following tables summarize the expression class rules for the operands and results of the C operators. They show the valid classes for the operands to each of the C operators and the class of the result.

Notice that some operators, such as the function call and select member operators, allow various classes of operands to be used. In these cases the class of the result depends on the particular classes of operands which are used.

Expression Class Rules for Binary Operators			
Operator	*Left Operand*	*Right Operand*	*Result*
Function call	function designator rvalue	rvalue list rvalue list	rvalue/void rvalue/void
Subscript	rvalue	rvalue	lvalue
Arrow	rvalue	member name	lvalue
Select member	lvalue rvalue	member name member name	lvalue rvalue
Binary Arithmetic, Relational, Equality, Bit-wise, Logical	rvalue	rvalue	rvalue
Assignment	modifiable lvalue	rvalue	rvalue
Comma	void void	rvalue void	rvalue void

Expression Class Rules for Unary Operators		
Operator	*Operand*	*Result*
Unary Arithmetic, and Logical	rvalue	rvalue
Increment and Decrement	modifiable lvalue	rvalue
Type cast	rvalue void	rvalue void
sizeof	lvalue rvalue	rvalue rvalue
Address of	lvalue function designator	rvalue rvalue
Indirection	rvalue rvalue rvalue	lvalue function designator void

Expression Class Rules for the Conditional Operator				
Operator	*Left Operand*	*Middle Operand*	*Right Operand*	*Result*
Conditional	rvalue	rvalue	rvalue	rvalue
	rvalue	void	void	void

Class Conversions

C is actually very forgiving when you write expressions in your programs. In situations where one class of operand result is required and a different class has been supplied, the C translator will make certain implicit class conversions. This is done to make it easier to write your programs.

The most common implicit class conversion is:

> If the expression context requires an rvalue and the result of the expression is an lvalue, that does not have an array type, then an implicit lvalue to rvalue conversion is performed.

When this lvalue to rvalue conversion is performed, the rvalue value attribute is set to the value stored in the data object designated by the address attribute of the lvalue. The type remains unchanged.

It is this conversion that comes into play in expressions such as `x * y`. The multiply operator requires rvalue operands. The names `x` and `y` (assuming appropriate declarations, such as `int x, y;`) are lvalue expressions that result in lvalues. In this situation, the C translator performs the implicit conversion to access and use the values stored in the data objects designated by `x` and `y`.

One context where an rvalue is frequently required is the "full expression" context. A full expression is an expression which is not an operand expression. It stands alone. Full expressions that require rvalues occur in the following contexts:

- In a declaration, to specify the number of elements in an array.
- In a declaration, to specify initializers for data objects and enumeration constants.
- In a declaration, to specify the number of bits in a bit-field.
- In the controlling expression of the `if`, `while`, `switch`, and `do` statements.
- In each of the three optional expressions of the `for` statement.
- In `case` labels.
- In the optional expression in a `return` statement.
- In argument lists for function calls.
- In the `#if` and `#elif` preprocessing directives.

In each of these cases, if an lvalue is created when the expression is evaluated it is implicitly converted to an rvalue.

Note that if the context requires an lvalue then no conversion is performed. This situation occurs, for example, in the left operand of the assignment operator and the operand to the "address-of" operator. If the context allows either an lvalue or an rvalue, then no conversion is performed. This situation occurs, for example, in the left operand of the select member operator "`.`".

Another implicit class conversion rule applies to the special case where the lvalue has an array type.

> If the expression context requires an rvalue and the result of the expression is an lvalue with an array type, the lvalue is converted to an rvalue with a *pointer to type*. If the original type was *array of type* the converted type is *pointer to type*. The value attribute of the rvalue is set to the starting address of the array.

For example, after these declarations:

```
int iarr[5];
int *pi;
```

the expression `pi = iarr` causes this conversion to take place. The result of the right operand expression has an lvalue class, an *array of* `int` type, and address attribute equal to the beginning of the array designated by `iarr`. The implicit class conversion changes this to have an rvalue class, a *pointer to* `int` type, and a value attribute equal to the address of the beginning of the array designated by `iarr`. The assignment operation can now take place.

Another class conversion allows function designators to be specified in contexts where rvalues are required. This works in a manner which is similar to the array type lvalue to rvalue conversion.

> If the expression context requires an rvalue and the result of the expression is a function designator with type *function returning type*, the function designator is converted to an rvalue with type *pointer to function returning type*. The value attribute of the rvalue is set to the address that designates the function.

For example, after these declarations:

```
int func(void);
int (*pfi)(void);
```

the expression `pfi = func` causes this conversion to take place. The result of the right operand expression has a function designator class, a *function taking no arguments and returning* `int` type, and an address attribute equal to the address associated with the function designated by `func`. The implicit class conversion changes this to have an rvalue class, a *pointer to function taking no*

arguments and returning `int` type, and a value attribute equal to the address of the beginning of the function designated by `func`. The assignment operation can now take place.

A final implicit class conversion deals with contexts where void class expressions are required.

> If the expression context requires a void class result and the result of the expression is an lvalue, rvalue, or function designator, then an implicit conversion to a void result is performed.

This conversion is performed by discarding the value or address attribute. For example, the complete expression contexts which require a void class result occur in the following contexts:

- Expression statements
- The first and third expression in the `for` statement
- The left operand of the comma operator

In each of these contexts lvalue, function designator, and rvalue results are implicitly converted to void class results.

These rules, along with the operator precedence table, allow you to decipher complex expressions.

An Example

Let's look at a moderately complex example.

```
int x;
int *arr[2];
int **pi;

x = 5
arr[1] = &x
pi = &arr[0];
```

Now consider this expression:

```
*pi[1] + 2
```

According to the precedence rules, the subscript operator is applied first. The operands of the subscript operator are the lvalue expression `pi` on the left and the rvalue expression 1 on the right. If you look in the class tables shown earlier you will see that the subscript operator requires rvalues for both of its

operands. The automatic lvalue to rvalue class conversion is therefore applied to the lvalue result of `pi`. This produces an rvalue result with a value attribute equal to the contents of the data object designated by `pi` (the address of the array `arr`).

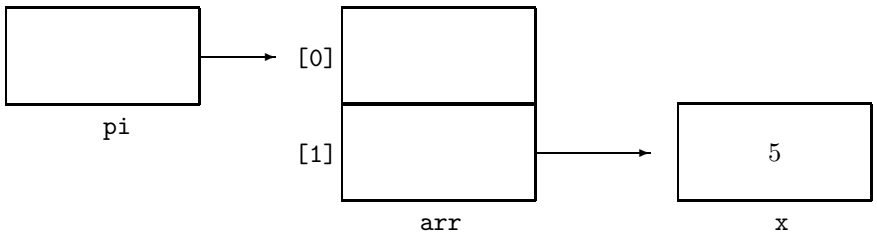
Subscripting is essentially an addition (except that the subscript is scaled by the size of the objects pointed to and the result is an lvalue), so the value from the data object designated by pi is added to a scaled 1. The address attribute of the subscript lvalue result is set to this calculated value (the address of the second element of array, remember we start at 0).

Next the indirection operator is applied. It takes an rvalue operand. We have an lvalue (the result of the evaluation of the subscript expression). Therefore, we again apply the automatic lvalue to rvalue conversion. This time, the value from the data object designated by the operand address attribute (the second array element) is accessed.

Indirection is an interesting operator. Its primary function in life is to take an rvalue and convert it into an lvalue (or a function designator if working with pointers to functions). The lvalue which results from the evaluation of the indirection expression has an address attribute equal to its operand's value attribute. This is the address contained in the second element of the array (the address of the data object designated by x).

Next we apply the add operator. The add operator requires rvalue operands. Therefore, once again we apply the lvalue to rvalue automatic conversion to the left operand to obtain an rvalue. This causes the value in the data object addressed by the contents of the second element of the array (the data object also designated by `x`) to be retrieved. This value (5) is then added to 2. The result is an rvalue with a value attribute of 7.

Graphically the data structures accessed are as follows:



Test out your understanding of these rules and this approach to deciphering expressions by trying to decipher what these expressions mean.

```
*arr[1] + 2
(*pi)[1] + 2
```

Try them out with your C translator. As a hint, you may want to add the

following declaration and statements to your program when trying out the last expression.

```
int arr2[2];

arr2[1] = 9;
arr[0] = &arr2[0];
```

Good luck!

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) 789-5462 or uunet!aussie!jimb.

∞

30. Variable Length Arrays

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

Abstract

The inability to declare arrays whose size is known only at runtime is often cited as an impediment to using C as a numerical computing language. Eventual adoption of some standard way of declaring runtime arrays is considered crucial for C's acceptance as a useful language in the numerical computing world. Since standards attempt to standardize existing implementations and prior art, an initial implementation is needed. This paper describes an implementation of variable length arrays which Cray Research, Inc. has chosen to implement. A description of the linguistic issues encountered during this development are presented along with some rationale for the chosen resolution.

Introduction

Currently, C only supports the declaration of *Fixed Length Arrays (FLA)*. It does not support the declaration of *Variable Length Arrays (VLA)*. This omission is a major obstacle to using C for numerical and scientific applications. Some VLA support exists, in that an array is converted to a pointer before it is passed as an argument to a function. Similarly, a formal parameter declared to be an array has its type adjusted to be a pointer. The pointer can be used to access any element of the array, which means the function can process arrays of variable length. However, this implicit conversion only applies to the first dimension. This means that the second dimension of a two dimensional array is not implicitly converted. Therefore, C does not, for instance, provide a straightforward way to define a function that performs a matrix multiply on arbitrary sized matrices.

It would be quite convenient to be able to define a function something like:

```
void mxm(int n, int m, double a[n][m],
    double b[m][n], double c[n][n]) {
    /* ... */
} /* mxm */
```

to perform a matrix multiply on arbitrary sized matrices. Since this widely used feature has been in FORTRAN for many years there is substantial prior art. The justification is easy, but attempting closure within the language raises many issues. This paper attempts to clarify those issues, and offers guidance for resolution based on an implementation of C that runs on Cray supercomputers. It should also be noted that the GNU C compiler supports the notion of a VLA declaration.

VLA Support Through Dynamic Allocation

Before exploring all the issues, it is appropriate to note that a solution exists without requiring any changes to the language. Essentially, this solution involves using the library function `malloc` to dynamically allocate an array of pointers. Each element of the array points to a different row of a matrix. This array of pointers permits access to the entire matrix through multiple indirection, as the following example demonstrates.

Example 1 – User Allocated

```
#include <stdlib.h>

double a[10][20], b[20][10], c[20][20];
void mxm(int n, int m, double **a, double **b,
    double **c);

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

    /* allocate arrays of pointers */
    pa = malloc(10 * sizeof(double *));
    pb = malloc(20 * sizeof(double *));
    pc = malloc(20 * sizeof(double *));

    /* setup array of pointers */
    for (i = 0; i < 10; i++)
        pa[i] = a[i];   /* pa[i] -> i-th row of a */

    for (i = 0; i < 20; i++) {
        pb[i] = b[i];   /* pb[i] -> i-th row of b */
        pc[i] = c[i];   /* pc[i] -> i-th row of c */
    }
```

```
    mxm(20, 10, pa, pb, pc);

    free(pa);
    free(pb);
    free(pc);

} /* main */

void mxm(int n, int m, double **a, double **b,
    double **c) {
    int i, j, k;

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

Using this technique, a library of utilities can be provided to create, convert, and free matrices of variable length. The calls to `malloc` and setting up the array of pointers is handled by a `create_mx` utility, and freeing up the matrix is handled by a `free_mx` utility. The `create_mx` utility might also allocate the space for the matrix itself. The following example shows how the `create_mx` and `free_mx` utilities can be used.

Example 2 – Library Allocated

```
main(){
    double **pa, **pb, **pc;

    /* allocate arrays of pointers */
    pa = create_mx(a, 10, 20);
    pb = create_mx(b, 10, 20);
    pc = create_mx(c, 10, 20);
```

```
    mxm(20, 10, pa, pb, pc);

    free_mx(pa);
    free_mx(pb);
    free_mx(pc);

} /* main */
```

The book *Numerical Recipes in C: The Art of Scientific Computing* by Press, Flannery, Teukolsky, and Vetterling (Cambridge University Press), gives an excellent description of the framework needed for this support.

This paper deals with the issues introduced by providing direct compiler support for VLA declarations. These are arrays whose length can vary each time the function or block containing the declaration is entered. Providing direct compiler support for VLA declarations eliminates the need for explicit calls to `malloc` and `free`. It is my belief that this lends itself to a more natural view of the problem being solved. The *User Allocated* and *Library Allocated* approaches, discussed above, will be compared with a *Compiler Allocated* approach presented in the next example.

Direct VLA Support

Permitting formal parameters to be VLA declarations is the primary motivation for this feature. There is no need to dynamically allocate any memory for a formal VLA since the address of the first element is passed, just like a formal FLA. The following example shows how the matrix multiply example appears using VLA formal parameters.

Example 3 – Compiler Allocated[8]

```
double a[10][20], b[20][10], c[20][20];

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

main(){
    mxm(20, 10, a, b, c);
} /* main */
```

[8]Technically speaking, the parameters `n` and `m` should have the type `size_t` which is defined in the `<stddef.h>` header.

```
void mxm(int n, int m, double a[n][m], double b[m][n],
    double c[n][n]) {
    int i, j, k;

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

Now that the three approaches, *Example 1 – User Allocated, Example 2 – Library Allocated*, and *Example 3 – Compiler Allocated*, have been examined, some comparisons can be made. From a readability viewpoint, both the *Library Allocated* and *Compiler Allocated* approaches are clearer and easier to understand than the *User Allocated* approach. The *User Allocated* approach burdens the programmer with writing extensive allocation, setup, and deallocation code. The *Library Allocated* approach requires a library of support utilities. Although this eliminates much of the burden, the library does not directly support more than two dimensions. The *Compiler Allocated* approach has an advantage over the *Library Allocated* approach in that the library of utilities is not needed and support for more than two dimensions is provided directly by the compiler.

The following terms are useful for discussing the issues surrounding the use and declaration of a VLA and to help explain new syntax that is being introduced. A *prototype definition* is a function definition declared with a prototype, while a *prototype declaration* only contains the prototype and not the function body.

```
void f(int, double *);     /* prototype declaration */

void f(int n, double *p) { /* prototype definition */
    while (--n >= 0)
        p[n]++;
}
```

A *type* consists of zero or more *modifiers* and a *basic type*. A modifier is a part of the typing mechanism and is one of *pointer, array*, or *function*. The following declaration:

```
int *a[10];
```

declares `a` to have the type *array of ten pointers to* `int`. This type is composed

of two modifiers, *array* and *pointer*, in which the order is important, and a basic type of `int`. (A basic type is either an arithmetic, structure, union, or void type. An arithmetic type is either an enumeration, integral, or floating type.)

Currently, all array modifiers have a fixed length. Introducing variable length arrays also introduces a *VLA modifier* into the typing mechanism. This requires that the entire language be reviewed for issues surrounding the declaration of a VLA modifier, and the interaction between two compatible types, with one containing an FLA modifier and the other containing a VLA modifier.

Formal Parameters

Example 3 introduces a prototype declaration with a VLA modifier as part of the type of a formal parameter.

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

The `[*]` notation is used to indicate that an array modifier inside a function prototype is a VLA. The `[*]` notation is *only* permitted inside a prototype declaration and *not* inside a prototype definition. This is only used for type compatibility checking with the function definition and other prototypes for the same function. The following prototype declarations are also equivalent.

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

because the first array modifier is converted to a pointer. Similarly, for the prototype definition the following definitions for `mxm` are equivalent.

```
void mxm(int n, int m, double a[ ][m], double b[ ][n],
    double c[ ][n]) {
    /* ... */
}

void mxm(int n, int m, double (*a)[m], double (*b)[n],
    double (*c)[n]) {
    /* ... */
}
```

VLA Expressions

The VLA modifier can contain an arbitrary integral expression. There is no linguistic reason to place restrictions on which operators can appear as part of the *length expression* in a VLA modifier. All operators, including function calls, short circuit operators (`&&` and `||`), and conditional operator (`?:`), are permitted. The use of assignment operators is also permitted, but is of dubious value.

```
double a[n][n + m];  /* + in length expression */
double b[n][m++];    /* use of ++ is questionable */
```

If the declaration of a VLA contains multiple assignments, the order in which the assignments occur is unspecified.

The value of the length expression in a VLA modifier is stored inside a compiler generated temporary after it is evaluated at execution time. This prevents the size of a VLA from changing after it has been allocated.

Example 4

```
void f(int n, double a[n]) {
    while (--n)
        a[n] = 0;
}
```

Note that the size of `a` does not change when `n` does. This makes the size of a VLA such as `a` an invariant value as long as it is visible, but it is not a compile-time constant—its size can change every time it becomes visible.

All of the examples used so far have declared the parameter `n` before the VLA parameter. This is an undesirable restriction to place on programmers since they currently do not need to worry about the order in which formal parameters are specified. Consider the following prototype declaration and old style function definition:

Example 5

```
/* prototype for an old style definition */
void f(double a[*][*], int n);

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

The order in which the names are specified in the parameter list is independent of the order of the declarations for those parameters. The accompanying prototype declaration is compatible with the definition. For this reason it seems appropriate to allow a similar prototype definition such as:

Example 6

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

to exist with the same meaning. That is, the order in which the parameters are specified remains unimportant. In order to accomplish this temporary suspension of honoring the lexical ordering, the concept of an incomplete type is expanded. An *incomplete type* is a type whose size is unknown. A VLA modifier inside a prototype definition is an incomplete type. That is, the size is not known at the time of the declaration. An attempt is made to complete the type when the `)` that terminates the function prototype is encountered. If at that time the VLA modifier is still incomplete it is undefined behavior. (A quality implementation will likely produce an error message.)

Example 7

```
extern int n;           /* file scope variable */

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

```
extern int m;           /* file scope variable */

void g(double b[][n * m], int n) {
    /* ... */
}


void h(double c[][x], int n) {  /* ERROR */
    /* ... */
}
```

For function `f` the formal parameter `a` is an incomplete array type until the `)` is encountered. At that time, the type of `a` is completed by first searching for `n` in the formal parameter list and then as a file scope variable. In this case, formal parameter `n` is used to complete the type. For function `g` the type of `b` is completed by using the formal parameter `n` and the file scope variable `m` in the expression `n * m`. Finally, for function `h` the type of formal parameter `c` cannot be completed because it uses the undeclared identifier `x` for its size. A compile-time diagnostic is issued for this error.

Automatic VLA declarations

A declaration containing a VLA modifier is only permitted for formal parameters, block scope `typedef`s, `auto`, and `register` declarations. This means that file scope declarations, and block scope `static` and `extern` declarations cannot contain VLA modifiers. A function cannot return a VLA.

Example 8

```
extern int n;
double glob[n];         /* ERROR - file scope VLA */

void f(double a[n], int n) {

/* declare a temporary with the same
   length as parameter a */

    double tmp1[n];
    static double tmp2[n];      /* ERROR - static VLA */

    /* ... */
}
```

A declaration of an `auto` variable containing a VLA modifier can appear

inside an inner block. Each time the block is entered, the length expression of the VLA modifier is evaluated. The memory for a VLA is allocated upon entrance to the block containing the declaration and deallocated when the block is exited. This means that the size of the VLA can change each time the block is entered.

Example 9

```
void f(double a[n], int n) {
    int i;

    for (i = 0; i < n; i++) {

    /* the size of tmp changes for each iteration */
        double tmp[n - i];

        /* ... */
    }
    /* ... */
}
```

Pointers to VLAs

When the address of a VLA is taken, it yields a pointer to a type that is a VLA. Since types such as *pointer to VLA* exist, objects can be declared with such types. The declaration:

```
register int (*pvla)[n];
```

declares a *pointer to array of* `n int`*s*. Since `pvla` is a pointer and not a VLA, a `register` declaration is acceptable. The assignment:

```
int a[n];
int (*pvla)[n];

pvla = &a;
```

is perfectly acceptable because the type of `pvla` is compatible with the type of `&a`. However, this does raise an issue about loss of type information when corresponding FLA modifiers meet VLA modifiers. Consider the following:

```
int a[10];
int (*p)[n];

p = &a; /* Are these assignment compatible? */
```

If `&a` can be assigned to `p` then a VLA modifier is always compatible with any FLA modifier. If they are compatible then the *fixed length* information (i.e., size of the array is ten) is replaced with the *variable length* information. This means that information is lost because the compiler is no longer able to diagnose inconsistent usage. This loss of information is accepted because it allows for much greater flexibility and usability. The following example shows how information is lost when a VLA modifier and FLA modifier are compared for compatibility. No compile-time diagnostic is issued for these incompatible types, but the program is erroneous.

Example 10

```
int n = 10;
int a[10];

main() {
    f(&a);  /* pointer to array of ten ints */
}


g(int q[ ][11]) {
    /*  ...  */
}


f(int (*p)[n]) {
    /* incompatible types are not diagnosed */
    /* at compile time:  10 != 11 */
    g(p);
}
```

Similarly, two different VLA modifiers are always compatible.

```
int (*p1)[n];
int (*p2)[m];
int (*p3)[x*y+7];

p1 = p2 = p3;   /* All are assignment compatible */
```

When checking for type compatibility, the length of a VLA modifier is assumed to be the same length as a corresponding FLA or VLA modifier. The burden is placed on the programmer to ensure that the two lengths are identical. If at execution time the lengths are different, it is an erroneous program with undefined results. However, corresponding FLA modifiers must still have

the same lengths, which can be detected at compile-time.

```
int (*p1)[n][4];
int (*p2)[m][6];

p1 = p2;        /* ERROR - because 4 != 6 */
```

Finally, the following example shows that a pointer to a VLA is assignment compatible with a pointer to an FLA.

```
int (*p1)[n];
int (*p2)[10];

p1 = p2;
p2 = p1;        /* both are permitted */
```

This means that a VLA modifier is always compatible with an FLA modifier. Providing execution time checks that ensure that the lengths of corresponding VLA modifiers are identical is a desirable debugging aid, but is left as a quality of implementation issue.

Making `typedef` Work

A VLA modifier can appear as part of a block scope `typedef` declaration. In this case a decision has to be made about when to evaluate the length expression of the VLA modifier.

Example 11

```
void f( ) {
    int i;
    int n = 10;
    typedef int An[n]; /* length expression is n */
    An x;              /* array of 10 ints */

    n = 11;
    {
        An y;          /* array of 10 or 11 ints? */

        /* ... */
    }
}
```

Since a `typedef` declares a synonym for a type, there is no storage allocated for `An`. The length expression `n` could be evaluated when the `typedef An` is declared or when the arrays `x` and `y` are declared. The decision was made to store the size into the temporary when `An` is declared because this means that the following expressions are always true:

```
sizeof(An) == sizeof(x)
sizeof(An) == sizeof(y)
sizeof(x) == sizeof(y)
```

Essentially this means that `x`, `y`, and `An` have the same length. That length may change each time function `f` is called, but the sizes of `An`, `x`, and `y` will always be equal. The type of a VLA (e.g., `An`) is considered to be a *Variable Length Type (VLT).*

The `sizeof` Operator

The size of an object is such an important concept in the language that it is important to define the semantics of the `sizeof` operator when its operand is a VLA or VLT. Currently, `sizeof` can be used in any constant expression (except for preprocessor directives such as `#if`). Since the length of a VLA is not known until execution-time, the compiler must generate code that computes the size of a VLA. This means that the `sizeof` operator does *not* always yield a constant expression. If the operand of a `sizeof` operator is a VLA or VLT then the result is not a constant. Although this seems controversial, it is also important in that the notion of size is an integral part of other operations. The operations involving pointer increment, subscripting, and pointer difference are closely tied to the size of an object. Specifically, given an object type `T` and the following declarations, the three pairs of expressions shown below represent equivalent results:

```
T a[N];
T *p = &a[0], *q = &a[3];
int i;


/ * pointer addition */

/*1*/  p + i
/*2*/  (T *)((char *)p + i * sizeof(T))
```

```
/* subscripting */

/*1*/  a[3]              /*  *(a + 3)  */
/*2*/  *((T *)((char *)p + 3 * sizeof(T)))


/* pointer difference */

/*1*/  p - q
/*2*/  ((char *) p - (char *) q)/(ptrdiff_t) sizeof(T)
```

The definition of `sizeof` should always maintain these equivalences. The requirement of storing the value of the length expression of a VLA modifier in a compiler temporary preserves these equivalences. However, the use of `sizeof` where constant expressions are required can now cause a diagnostic, as shown in the following example.

Example 12

```
void f(int n) {
    int fla[2];         /* fixed length array */
    int vla[n];         /* variable length array */

    static int i = sizeof(fla); /* OK */
    static int j = sizeof(vla); /* ERROR */

    auto int k = sizeof(vla);   /* OK */


    enum { A = sizeof(fla),     /* OK */
        B = sizeof(vla) };      /* ERROR */

    struct {
        int b1 : sizeof(fla);   /* OK */
        int b2 : sizeof(vla);   /* ERROR */
    } x;


    int *p = malloc(sizeof(vla));/* OK */
    int num_elem = sizeof(vla)/sizeof(int);/* OK */
```

```
    switch (i) {
        case sizeof(fla):       /* OK */
        case sizeof(vla):       /* ERROR */
    return;
    }
}
```

Although the `sizeof` operator can now produce a value computed at runtime, there is still a consistency in its definition that preserves its most useful properties. For instance, it can still be used as the operand to `malloc` and to compute the number of elements in an array.

Pointer Difference

Before two pointers can be subtracted (portably) they must conform to the following two rules:

- They must be pointers to the same type.
- They must point into the same object.

The same rules apply when a pointer points to a VLA. The following example shows several pointer subtractions involving pointers that point to a VLA. Comments are present to indicate if the expression violates either rule.

Example 13

```
int a[10][10];
int b[3][5];

int n0 = 10, n1 = 10;
int m0 = 5, m1 = 3;

void f( ) {
    int i;
    int (*p0)[n0] = a;
    int (*p1)[n1] = a+1;
    int (*q0)[m0] = b;
    int (*q1)[m1] = (int (*)[m1]) b+1;
    int (*p10)[10] = a+2;
```

```
    /* OK:  p1 and p0 point into same object */
    i = p1 - p0;        /* OK:  i == 1 */

    /* OK:  q1 and q0 point into same object */
    i = q1 - q0;        /* ERROR because 3 != 5 */


    /* OK:  p10 and p0 point into same object */
    i = p10 - p0;       /* OK:  i == 2 */

    /* ERROR: p10 and q1 do not point into same object */
    i = p10 - q1;       /* ERROR because 10 != 3 */
}
```

The "errors" identified in Example 11 are not diagnosed at compile-time. Whether or not execution-time checks are generated is left as a quality of implementation issue.

Variable Length Objects

Up to now this paper has concentrated on arrays and pointers to arrays. However, a structure or union member can also be declared with a VLA modifier. A structure or union with a VLA member is considered to be a *Variable Length Object (VLO)*. Similarly, a structure or union type that contains a VLA member is considered to be a VLT. Essentially, the same rules apply to a VLO that apply to a VLA with one exception: formal parameters *cannot* be declared to be variable length structures or unions. If they were, it is not clear how such a function could ever be called. For example:

Example 14

```
void f(int n) {

    struct tag {        /* struct tag is VLT */
        int len;
        int a[n];
    } x;                /* x is a VLO */

    x.len = n;

    /* ... */
}
```

```
/* Can this function ever be called? */

void g(int n, struct vtag { int m1; int m2[n]; }) {
    /* ... */
}
```

The size of a variable length structure type is computed by evaluating the size of all variable length members at execution time and adding it to the constant fixed size.

The size computation of a variable length union type is different than for other VLTs because either the largest fixed sized member or largest variable sized member can be the size of the type. For example:

Example 15

```
void f(int n, int m) {

    union tag {         /* union tag is VLT */
        int a[n];
        int b[10];      /* largest fixed size member */
        int c[m];
        char d[20];
    } x;                /* x is a VLO */

    /* ... */
}
```

The size of a variable length union type is computed by first computing the size of all variable length members. Then computing X, the maximum of these variable length sizes; and then computing the maximum of X and the largest fixed size member.

It is important to note that the size computation of a VLT can be complicated.

Variable length structure and union types will probably not be implemented in the initial release. There are no linguistic reasons preventing such implementation, however, just pragmatic issues such as limited resources and deadlines.

Miscellaneous Issues

The `offsetof` Macro

The `offsetof` macro expands into an implementation-defined constant expression that yields the byte offset of the member-designator of a struct or union type. Since `offsetof` currently yields a constant expression, the same issues

that surrounded `sizeof` exist. There are no problems with members that have a constant offset. The problem is with members that follow a variable length member. For example:

```
struct tag {
    int m1;
    int m2[n];
    int m3;
};


offsetof(struct tag, m1)        /* OK */
offsetof(struct tag, m2)        /* OK */
offsetof(struct tag, m3)        /* Undefined behavior */
```

Since the expansion of `offsetof` is implementation-specific, it seems impractical to guarantee any behavior for members that follow a variable length member. The `offsetof` macro can only be used if the header `<stddef.h>` is included in the program. Unlike `sizeof`, the `offsetof` macro is not part of the language proper. The actual expansion of `offsetof` varies from implementation to implementation, making it more difficult to specify.

Initialization

Initialization of a VLO is not permitted. However, a pointer to a VLO is still a scalar and not considered to be a VLO.

```
int a[n] = {1, 2};      /* ERROR */
int (*p)[n] = &a;       /* OK */
```

Jumping into blocks with VLO declarations

It is not permitted to jump into a block that contains a declaration with a VLA modifier because the size expression evaluated at execution time will be skipped. This applies to the `goto` statement, `switch` statement, and `setjmp/longjmp`.

Jumping into a block containing a VLA declaration causes a compile-time diagnostic. Jumping out of a block that contains a VLA declaration is permitted, but it might mean that memory for the VLA object is not freed.

Example 16

```
void f(int n) {
   int j = 4;
   goto lab3;   /* ERROR - going into scope with VLA */
   {
      double a[n];
      a[j] = 4.4;
lab3:
      a[j] = 3.3;
      goto lab4;    /* OK - same scope */
      a[j] = 5.5
lab4:
      a[j] = 6.6
   }
   goto lab4;   /* ERROR - going into scope with VLA */
}
```

Conclusions

Adding a variable length array feature to C raises many issues. The changes to the language to support this feature must be weighed against the need for this feature. It is hoped that this initial implementation will give valuable insight into the usefulness of providing variable length types.

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

∞

31. Miscellanea

compiled by **Rex Jaeschke**

Constant Types in `float.h`

Some time ago I came across an implementation whose header `float.h` contained the following macro definitions. The compiler runs on Intel 80x86 machines using IEEE floating-point representation (either Intel's 80x87 chip or emulation). The types `double` and `long double` are mapped to the same representation.

```
#define FLT_MIN         1.17549435e-38F
#define DBL_MIN         2.225073858507201e-308
#define LDBL_MIN        2.225073858507201e-308
```

The interesting thing here is the presence of the `F` suffix on `FLT_MIN` and then, the lack of an `L` suffix on `LDBL_MIN`. When I looked at six or seven other implementations (several of which are actively claiming ANSI conformance) I found none with such suffixes. Are these suffixes required or not?

According to §2.2.4.2, page 14, line 31, of the December 1988 draft, "... except, for `CHAR_BIT` and `MB_LEN_MAX`, the following [`limits.h`] macros shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integral promotions." That's fine for macros in `limits.h` but what about those in `float.h`? No such statement is made.

Instead, §2.2.4.2, page 16, line 19, states "The values given in the following list shall be replaced by implementation-defined expressions ..." No information is provided as to the permitted type of these expressions. However, it does say the value must be equal or greater in magnitude to the values shown.

When I spoke with David Prosser (the redactor of the draft) his (unofficial) response was " I think the pANS implies that the `F` suffix should be present by the two examples given at the end of §2.2.4.2. An implementation that provides a `long double` type that's larger than its `double` would require an `L` suffix, if only to eliminate the required diagnostic (see §3.1.3, page 26, line 37, **Constraints**)." The constraint is required since there is no hierarchy of constant typing for floating-point constants like there is for integers. That is, a constant that is too big for `double` is not automatically typed as `long double`.

The only place where I could see that the macro types would matter is if they were used in an argument to `sizeof` or in certain function arguments where no prototype was in scope at the call site.

As far as I can tell, the only macros in the header `<float.h>` affected are: `FLT_EPSILON`, `FLT_MIN`, `FLT_MAX`, `LDBL_EPSILON`, `LDBL_MIN`, and `LDBL_MAX`.

It appears that all the integer minima and maxima defined by the standard can be represented by type `int`. Certainly, that is the type inferred from the examples on pages 17–18. As such, I'm assuming that if `FLT_RADIX` is defined as `16L` or `16U`, it is not conforming. Any opinions on this?

Of course, the standard permits most of the macros in this header to expand to non-constant expressions. However, this discussion is still relevant in such cases since the expressions are still required to have a given type.

Idiosyncrasies of Designing `assert.h`

From a functionality point of view, the header `assert.h` is one the simplest in the standard library. However, to make sure it's ANSI-conforming, you must design it very carefully. Very few versions I have seen to date have gotten it right including several that define `__STDC__` to 1. (Note, there are several ways to design the header correctly.)

According to ANSI C, `assert.h` must be self-contained. That is, you are not required to include any other standard header to use `assert.h` correctly.

`assert.h` is the only standard header that is permitted to be included multiple times *and* to behave differently from one inclusion to the next. Therefore, you must *not* put this header's contents inside a protective wrapper. And since on each inclusion the definition of the `assert` macro may change, you *must* explicitly `#undef assert` each time.

Assuming `NDEBUG` is defined, `assert` *must* expand to a `void` expression. It *cannot* expand to nothing since this will fail when `assert` is used as the second or third operand of the conditional operator. The macro must have the form:

```
#define assert(arg) ((void) expression)
```

When `NDEBUG` is not defined, `assert` must still expand to a `void` expression, which eliminates the use of an `if`. You can use the conditional operator, but in doing so, you are forced to have both a second and third operand when you really only want to do something when the assertion is true.

```
#define assert(arg) (((arg) == 0) ?     \
  __assert(#arg, __FILE__, __LINE__) : ((void)0))
```

To eliminate this the logical OR operator can be used instead, as follows:

```
#define assert(arg) \
  ((void)((arg) || __assert(#arg, __FILE__, __LINE__)))
```

The above definitions have `assert` expanding into a call to the "private" function `__assert`. Note that according to §4.1.2.1 **Reserved Identifiers**, page 98, all private externals defined by the implementation *must* be spelled with two leading underscores or an underscore and a capital letter. (Whether two underscores are needed, or only one, depends upon whether the symbol has file scope or block scope. The simplest and safest course is to use two leading underscores on your private function name.)

You may include a prototype for `__assert` if you wish but there is no requirement to do so, provided you can define the macro to work since the function has a fixed-length argument list. Of course, if you do have a prototype, any optional argument identifiers *must* be in your namespace as well.)

The big question is, "Can `assert` expand directly to a call to `fprintf`?" If the answer is *Yes* you would need a prototype for `fprintf` since to call a function with a variable argument list without a prototype in scope is undefined behavior. However, for most (if not all) implementations, `assert.h` cannot include `stdio.h` since the latter contains `typedef`s. These identifiers have file scope, there is no way to undefine them, and they are not defined in the section describing `assert.h`. Therefore, they are not permitted there. That leaves the possibility of duplicating the required bits from `stdio.h` in `assert.h`.

To write to `stderr` *without* defining the names `stderr` or `FILE` is challenging, although I believe it can be done at least for some implementations. However, the amount of effort is hardly worth it considering a more overt and probably just as efficient method is available using something like `__assert`.

I have noticed some implementers call `__assert` as follows:

```
__assert((arg), #arg, __FILE__, __LINE__)
```

Here, `__assert` checks whether the argument is true or false, eliminating the test code from the macro expansion, and hence from all the places it is called. The cost though is that the function is always called, not just when it needs to abort.

Whatever approach you use you probably should call `fflush(stderr)` to make sure the output is actually written since `abort` is not required to do so itself.

One last thing. The preprocessor stringize operator `#` has to be used to construct the expression text since the old-style method of recognizing formal macro arguments inside strings (and character constants) is not sanctioned by ANSI C.

Here then, is a correct version of `assert.h`:

```
/* assert.h -- designed by Rex Jaeschke for public use */

#undef assert

#ifdef NDEBUG
   #define assert(arg) ((void)0)
#else
   void __assert(const char *, const char *, int);

   #define assert(arg) (((arg) == 0) ?  \
     __assert(#arg, __FILE__, __LINE__) : ((void)0))
#endif
```

To check your version, compile the following test program. No errors should be produced.

```
#define NDEBUG
#include <assert.h>

void f(int i)
{
        assert(i);

        i ? assert(i - 4) : assert(i + 4);

        if (i > 24)
                assert(i * 3);
        else
                assert(i * 24);
}


#undef NDEBUG
#include <assert.h>

void g(int i)
{
        assert(i);

        i ? assert(i - 4) : assert(i + 4);

        if (i > 24)
                assert(i * 3);
        else
                assert(i * 24);
}
```

Quality of Implementation

Strict Prototype Checking

ANSI C requires function arguments to be assignment compatible to the corresponding prototype. However, this can still permit logical errors to go undetected. For example, consider the following code. (The prototypes may be in the source file or an included header.)

```
void f(double);
void g(char);

main()
{
/*6*/   f('a');         /* char confronts double */
/*7*/   g(1234.56E26);  /* double confronts char */
}
```

Silently converting the `char` expression to type `double`, and truncating the `double` constant to `char`, will likely produce surprising results. The ability to selectively activate more strict checking is highly desirable.

Microsoft C V5.1 provides this capability. For example:

```
test.c(6) : warning C4051: data conversion
test.c(7) : warning C4051: data conversion
```

The DOS-based DeSmet compiler also provides this capability. (Error locations are identified by this compiler using `$$`.)

```
6     f('a' $$ );
       warning:argument type conversion
7     g(1234.56E26 $$ );
       warning:argument type conversion
```

Lattice's V6.01 compiler produces:

```
test.c 6 Warning 89: constant converted to required type
        f('a');         /* char confronts double */
            ^
test.c 7 Warning 89: constant converted to required type
        g(1234.56E26);  /* double confronts char */
                  ^
```

An even more subtle situation exists in the following case:

```
void f(int);
void g(unsigned);

main()
{
/*6*/   f(100U); /* unsigned confronts signed */
/*7*/   g(100);  /* signed confronts unsigned */
}
```

In this case, Microsoft C and Lattice C did not complain while DeSmet produced messages similar to those shown in the previous example. In both compilers, the extra checking is activated using a compiler option—it is not the default mode.

Actually, flagging certain type "mismatches" can be most distracting. For example, warning about `malloc(100)`, simply because `100` has type `int` when an unsigned integer type is expected, is undesirable. However, warning about `malloc(-10)` or `malloc(i)`, where `i` is an `int` variable (or expression involving other than constants), is not. Similarly, if a signed integer type is expected and you pass an unsigned integer constant expression whose value can be represented in that signed type, no warning should result.

Machine Code Listing

Many implementations either generate assembly code directly or have an option to disassemble the object produced. However, since most of them cannot produce a listing file, the assembly code is written to its own file. Many DOS-based compilers can produce a source listing file containing merged assembly code. These include: Microsoft, Borland's Turbo C, WATCOM, Zortech, DeSmet, and Lattice. VAX C also has this capability.

Automatic Prototype Extraction

In the March 1989 Sample Issue of *The Journal of C Language Translation*, I described details of a prototype extraction tool. This generated quite some interest in the user and implementer community. There were also numerous suggestions for a tool to rewrite programs changing to/from old and new styles of function definitions as well as declarations. (See the section following.)

The DOS-based compilers from Microsoft, WATCOM, and Lattice provide a compiler option for prototype extraction. The Manx Aztec system also provides a separate tool for this purpose. For example, the source:

```
#include <stdio.h>

void f(FILE *fp)
{
}

void g()
{
}
```

causes the following prototypes to be generated:

```
extern void f(FILE *);           /* WATCOM */
extern void g(void);

extern void f(struct _iobuf *fp);/* Microsoft */
extern void g(void);

void __near f(struct _iobuf *);  /* Lattice */
void __near g(void);

void f(FILE*fp);                 /* Manx */
void g(void);
```

Note, however, that Microsoft and Lattice do not preserve the `typedef` name, whereas WATCOM and Manx do. (It is interesting that WATCOM also provides an alternate option to emulate Microsoft's [limiting] behavior. I guess that's what competition does to you.) The identifier `__near` produced by Lattice C refers to the an Intel memory model in which function addresses are 16-bit "near" pointers.

Lattice provides several prototype-related compiler options. You can request all functions be extracted or just the `extern` or `static` ones. You can also specify whether identifiers are to be preserved. Another option allows you to generate function declarations using the old or the new format. When used with the earlier example above, it produces:

```
#ifndef __NOPROTO
#ifndef __PROTO
#define __PROTO(a) a
#endif
#else
#ifndef __PROTO
#define __PROTO(a) ()
#endif
#endif

void __near f __PROTO((struct _iobuf *));
void __near g __PROTO((void));
```

Simply by defining (or not defining) the macro `__NOPROTO` using a compiler option, you can compile with or without full prototype checking. This utilizes an old trick that I first came across in the Whitesmiths C compiler on the PDP-11.

Both Manx and Aztec retain `register` in formal parameters, with Manx making it optional.

ANSIfying Old Programs

The following text was taken from a note posted to the comp.std.c newsgroup by Ron Guilmette late in mid-1989.

Many people have noted the need for tools to automate the conversion of old C code (in non-prototyped form) to new ANSI C and C++ prototype form. Well, I noted it too and I decided to do something about it.

I would like to announce the availability of an automated prototyping assistant tool. This tool can automate much (but definitely not all) of the otherwise tedious work of converting a large system of source files to ANSI C (or C++) prototype format. This assistant can convert most of the obvious cases, leaving you to handle only the occasional tricky case manually.

This prototyping assistant tool comes in two separate parts. The first part gathers prototype information, and the second edits this information into the proper places (i.e., function declarations and definitions) within a set of existing source files (both base files and include files).

The information gathering tool is really a modified version of the GNU Project C compiler (GCC) Version 1.35. This was the most expedient base for an information gatherer, because it already had a complete (debugged) parser for full ANSI C.

The other half of the automatic prototyping system is the tool that actually merges the prototypes back into the source code at the right places. This is called PROTOIZE. It is written in (reasonably portable) C. It is about as intelligent as it can be, given the limitations of its input. For instance, it knows

the difference between `static` and `extern` functions, and about many other subtle points.

Since PROTOIZE must be used with GCC, the current version (1.00) is being distributed as a set of patches against the (virgin) GCC 1.35 sources.

You may obtain the current version of PROTOIZE via anonymous FTP from *yahi.stanford.edu* (36.83.0.92). The compressed set of GCC 1.35 patches is in:

```
~ftp/pub/protoize-1.00.Z.
```

This software is distributed with the same terms and conditions as other GNU software distributed by the Free Software Foundation.

For further information, contact:

Ron Guilmette
MCC - Experimental Systems Kit Project
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3740
Arpa: rfg@mcc.com
uucp: rutgers,uunet,gatech,ames,
pyramid!cs.utexas.edu!pp!rfg

In October an updated message was posted, excerpts of which are reprinted here.

PROTOIZE Version 1.04 is now available for anonymous FTP from ics.uci.edu (128.195.1.1). It is stored in the following file:

```
~ftp/pub/protoize-1.04.Z
```

This version is compatible (only) with GCC version 1.36.

Note that I have plans to produce a modified version of PROTOIZE called UNPROTOIZE when time permits. This new program will convert ANSI C programs back to old K&R code (without prototypes). Once this is available you will be able to automatically convert code back and forth between ANSI C and K&R C at will.

Calendar of Events

- January 22–26, **Winter 1990 USENIX Technical Conference** – Location: Washington, D.C. at the Omni Shoreham Hotel. Call Judith Des Harnais on (714) 588-8649 or *judy@usenix.org* for details.

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

- March 7–8, **Numerical C Extensions Group (NCEG) meeting** – Location: New York City. The third meeting will be held to consider proposals by the various subgroups. It will follow the X3J11 ANSI C meeting being held at the same location earlier that week and will run for two full days, *not* the usual one-and-a-half. For more information about NCEG, contact the convener Rex Jaeschke at (703) 860-0091 or *uunet!aussie!rex*, or Tom MacDonald at (612) 681-5818 or *tam@cray.com*.

- April 9–11, **USENIX C++ Conference** – Location: San Francisco Marriott. Contact Jim Waldo at *waldo@apollo.com* for more information.

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

- 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. The deadline for extended abstracts is Mar 15.

News, Products and Services

- Developers of C compilers for new architectures will find the **Edison Design Group** C Front End an effective basis for development of a high-quality translator. It supports both traditional K&R and ANSI C, and features rapid compilation, full error checking (with clear error display and excellent error recovery), and a high-level tree-structured intermediate form. Host and target computer characteristics are configurable, allowing use in a variety of environments and as a cross-compiler. Licensing is in source form, with complete internal documentation. For information, contact Steve Adamczyk at (201) 744-2620.

- Conversion service: If you have a large system that you would like to have converted from old C to ANSI C or to C++, please contact me via e-mail. I may be willing to perform the conversion (using the PROTOIZE tool described elsewhere in this column) for a modest fee based on thousands of lines of code (or something like that). Ron Guilmette, Austin, Texas, (512) 338-3740, *rfg@mcc.com*.

- **PROMULA Development Corporation** is licensing the source to PROMULA.FORTRAN its Fortran to C translator. V1.22 is now available for MS-DOS, VAX/VMS, and Sun/UNIX platforms. Also available are consulting and conversion services. Call George Juras on (614) 263-5454 for information.

- **Lattice, Inc.**, is now shipping V6.01 of their DOS compiler with claims it is ANSI-conforming. Contact John Nelson on (312) 916-1600.

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

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

∞