The Journal of

C Language Translation

Volume 1, Number 4

March, 1990

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

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

The entire contents are copyright © 1990, Rex Jaeschke. No portion of this publication may be reproduced, stored or transmitted in any form, including computer retrieval, without written permission from the publisher. All rights are reserved. The contents of any article containing a by-line express the opinion of the author and are not necessarily those of the publisher nor the author's employer.

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

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

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

The following are trademarks of their respective companies: MS-DOS and XENIX, Microsoft; PC-DOS, IBM; POSIX, IEEE; UNIX, AT&T; TEX, American Mathmatical Society.

Contents

32. Standard C Library Name Space Control

P.J. Plauger

Abstract

The ANSI Standard for C imposes quite a number of constraints on how the library can be implemented. It restricts the space of names available to the implementor. It requires that library headers be carefully constructed in several nonobvious ways. On the other hand, the Standard also offers new opportunities to improve library performance and power.

This paper is an overview of the Standard C library from the viewpoint of both the user and the implementor. Its goal is to illuminate some of the subtler implementation issues in the library. It highlights issues that may not be apparent from a simple reading of the Standard or the accompanying Rationale.

Introduction

The ANSI C Standard is a treaty between programmers and implementors of the language. That's a principle that I first espoused back in 1983. It made it into the Rationale as one of the guiding precepts of X3J11. It's an important touchstone for evaluating every statement in the Standard. A statement must restrict either the user or the implementor (or both) in some obvious way. Otherwise, it is deadwood.

Nowhere is the Standard more clearly a treaty than in the library portion. Users are given stronger assurances than ever before about what is in the Standard C library and how it behaves. Users are also told quite clearly what they must *not* count on. Similarly, implementors are given clear guidance as to what facilities they must provide. And they are told what practices *not* to indulge in when writing the Standard C library.

Like any treaty, the C Standard has a number of gray areas. Some turf is not safe to tread by either party to the treaty. It forms a sort of demilitarized zone. Some critics of the C Standard view these gray areas as weaknesses in the Standard. They assume that X3J11 was too tired or too cowardly to eliminate them. Not so. By and large, the gray areas were conscientiously evaluated and intentionally left in. It is important to leave space for *some* dialect spread in C, to benefit both programmers and implementors alike.

A treaty is easy to misread. (The wars of the Twentieth century are ample proof of that.) If you try to implement Standard C just by reading the ANSI

Standard, you are in for a tough time. Reading the Rationale certainly helps, but even that worthy document is laconic at critical moments. Most of us in the C community can fall back on our shared cultural understanding to fill in the blanks. That works best for the language proper, because that is largely unchanged. It works somewhat less well for preprocessing, because that got tidied up a lot and altered somewhat. It works worst of all for the library, because that has never been well documented in the past.

The purpose of this paper is to provide additional guidance to implementors of the Standard C library. It does so by emphasizing the treaty aspects of the Standard. It focuses primarily on naming rules and conventions, because that's where most of the action is.

I will examine each of several major library topics from four perspectives. First I discuss what the user *must* do. I follow that with what the user *can* do. Then I discuss what the implementor *must* do. I end each discussion with what the implementor *can* do.

I find that those four views, taken in that order, build the most comprehensive image. Leave any out and you distort the picture. Take them in any other order and you distort priorities.

Headers

Standard C partitions its libraries into fifteen chunks. Each chunk consists of a set of functions, type definitions, and macros that are at least loosely related. You obtain all the definitions and declarations you need to use a chunk by including its associated standard header. One of the cleanups effected by Standard C is to ensure that *every* library function is declared in some header.

User Viewpoint

To use a given header, what you must do is include it with the preprocessing directive `#include`. Write the name of the header in angle brackets, such as:

```
        #include <stdio.h>
```

The header is probably a file, but it need not be. That means, for one thing, that you must use angle brackets. Writing the name in double quotes does not necessarily work on all implementations.

You must include the header, naturally, before you use any of its declarations or definitions. You must write the include directive at *file level.* File level is outside any other declaration, including its data initializer or function body.

You must not precede the include directive for the header with any macros that mask keywords. Nor may you have any macro definitions or declarations in scope that match any of the names defined or declared in the header. This rule has two important exceptions, however. You can include the same header more than once in the same translation unit. And you can include a header that defines certain macros or types also defined in other headers. The privileged names are `NULL`, `size_t`, and `wchar_t`.

You can include headers in any order. You can omit a header, in some cases, and still use one of its functions. The function must be one that you can declare explicitly with known types, such as:

```
        extern double cos(double);
```

Or it must be one that you can declare implicitly, as in:

```
        int atomag(char *s)
                {return abs(atoi(s)); }
```

(Both `abs` and `atoi` are implicitly declared here.)

You must assume that any function name declared within a header is masked by a macro of the same name. That means you must not use the function name as, say, a member or tag name. Nor can you safely redeclare the function name in an inner block. Macro names have no respect for program structure.

Finally, you can use any library name as you see fit *inside* a translation unit, provided you do not include its associated header. By inside, I mean as a macro definition, a type definition, or anything else that does not have external linkage. It is generally not wise for you to do so, except out of ignorance, but you can.

Implementor Viewpoint

As an implementor, you must of course provide all fifteen headers, at least for a hosted environment. Four of the headers must be present even in a freestanding environment: `float.h`, `limits.h`, `stdarg.h`, and `stddef.h`. (Note that none of these declare any functions.) In a freestanding environment, you can choose to omit any or all of the other headers. You can implement only part of the capabilities in a given header. Only market forces really limit you here.

Several of the restrictions imposed on users are needed to give you a fighting chance. You need the assurance that your header file will not be included inside a function body or partway through some other declaration. Otherwise, you could not safely write define preprocessing directives and file-level declarations within the header. You also need to know that type names such as `char` will have the meaning you expect.

You must write each header so that it can be included multiple times. Most function declarations can be repeated. A macro definition can be repeated if the new is essentially identical to the old. Barring that, you can always precede each define preprocessing directive with an `#undef` for the same name. Type definitions, however, cannot be repeated.

The safest practice to follow, therefore, is to protect the body of each header with a wrapper, as in:

```
/* stdio.h header */
#ifndef _STDIO
#define _STDIO
/* ... guts of stdio.h */
#endif
```

Note that the standard header `<assert.h>` is different. It must test `NDEBUG` each time you include it, as Rex Jaeschke described in a recent issue of *The Journal* (Volume 1, number 3 December 1989, page 235). Use the same trick to protect each definition of `size_t` and `wchar_t`.

You must not have any header include another header. If any two headers must share knowledge, you must introduce a *synonym*. Note, for example, three of the functions in `<stdio.h>`. The functions `vfprintf`, `vprintf`, and `vsprintf` each have an argument of type `va_list`. But `va_list` is not defined in that header. You can only declare these functions by using a synonym for the type that cannot be named, as in:

```
typedef char *_VALIST[2];
int vfprintf(FILE *, const char *, _VALIST);
/* ... etc. */
```

You can replicate the synonym as needed among headers. Or you can place all critical synonyms in a secret header that you include as needed in the standard headers. The former choice makes for faster compiles and a less messy visible interface. The latter is better for maintenance and retargetability.

You can mask any function declaration with a macro definition. You do this for some functions so that they expand to inline code. For example, the functions in `<ctype.h>` often use a translation table, as in:

```
#define isdigit(c) (_Ctype[c] & _ISDIG)
```

Evaluating this expression is almost always preferable to calling the actual function. Don't omit the actual function declaration, however. The program may want to pass its address as a function call argument, as in:

```
found = istest(str, isdigit);
```

You may want to mask a function declaration to access a builtin operation. True, a translator can know that all library names have special meaning. It can replace every call to the function `cos` (with external linkage) with a builtin instruction sequence. Your customers may want to control this translation, however. One way to do so is by requiring the masking definition:

```
#define cos(x) _COS(x)
```

That way, only the rather exotic name `_COS` is magic to the translator.

You must be careful what names you use when you write macros. Consider, for example, the definition:

```
#define log10(x) (_LN10 * log(x))
```

It looks safe enough, until some perverse customer writes:

```
#include <math.h>
#undef log
#define log(x) (printf(%d\n", x), x)
```

Any subsequent use of `log10` will be very difficult to debug.

Reserved Name Spaces

Standard C defines several hundred names. A relatively small number are predefined by the translator. You are best off using these names only as intended, even in the rare cases where you can redefine them. The rest of the names are defined in the various standard headers. I outlined what you can and cannot do with those names in the previous section.

Standard C also makes assertions about other names. This section discusses what the user and implementor can and cannot do with those names.

User Viewpoint

You can probably write a C program with no thought to any names not explicitly defined in the language proper or the Standard C library. After all, that's the way most of us have had to write C code for decades. Mostly, it works. If you inadvertently collide with a name used by the implementor, you just change it and try again. Once you get the program working this way, you can stop worrying. You can stop worrying, that is, until the implementation is upgraded. Or until you need to move the code to another implementation.

Standard C offers you an additional level of security, however. It is a level offered in no other language standard that I know. It promises that if you avoid certain sets of names, you will experience *no* collisions. Thus, Standard C makes it that much easier to write highly portable applications.

Please note that your program is not ill-formed if you use one of the contraindicated names. It is just not "strictly conforming," which is Standard C-ese for "not guaranteed to be portable." I lean on this point here because it has been the source of considerable confusion, and some hard feelings, among critics of the C Standard. I use "must" and "must not" in the remainder of this section only to describe strictly conforming programs. Ignore these rules and you are no worse off than in the days before Standard C.

So, to write a strictly conforming program, you must avoid four classes of names:

Hidden macros – These are names that the implementor uses for macros needed to flesh out the implementation. They consist of all names that begin with an underscore, followed by an underscore or a uppercase letter. Examples are `_STDIO` and `__log2`. Note that hidden macros can be defined even if you include no standard headers in a translation unit.

Hidden external names – These are names that the implementor uses for functions and data objects that have external linkage but are not for you to use directly. They consist of all names that begin with an underscore and are declared with external linkage. Examples are `_errno` and `_sinx`. Note that you can write macro names, names with no linkage, and names with internal linkage that match this class of names.

Added macros – These are names that the implementor uses for macros added to certain header files. Typically, they name things peculiar to a given implementation. The complete list is:

- An implementation can add error code names to `<errno.h>` if they begin with `E`, followed by a digit or uppercase letter.
- It can add locale category names to `<locale.h>` if they begin with `LC_`, followed by an uppercase letter.
- It can add signal code names to `<signal.h>` if they begin with `SIG`, followed by an underscore or uppercase letter.

Future functions – These are names that are reserved by X3J11 for future inclusion in Standard C. That makes them likely candidates for inclusion in avant garde implementations of the current standard. By avoiding these names, you can be tolerant of such extensions now and avoid maintenance changes in the future. Note that, like all library function names, each can also be masked in its defining standard header with a macro definition. The complete list of places future implementations may add functions is:

- `<math.h>`, by appending `f` or `l` to existing function names declared in that header.
- `<ctype.h>`, provided the names begin with either `is` or `to`, followed by a lowercase letter.
- `<string.h>`, provided the names begin with either `mem`, `str`, or `wcs`, followed by a lowercase letter.
- `<stdlib.h>`, provided the names begin with `str`, followed by a lowercase letter.

If you avoid the names listed here, and the ones explicitly defined in Standard C, you should never collide with the implementation. *All* other names belong to you.

Your only residual concern is the constraint imposed on external names by primitive linkers and assemblers. An implementation of Standard C might be able to preserve only the first six characters of significance in an external name. It might have to compress uppercase and lowercase letters in an external name to a single case. If you write code that may have to move to such an environment, you must create names that survive such mangling. Make sure that no two names you create collide as a result.

Implementor Viewpoint

As an implementor, your task is fairly clearcut. Confine your secret names to the classes listed above and you will not step on the toes of the user. The headers you write may not be as readable as in the past, but they will be safe.

One thing you should be careful of. Remember that the user can write macros that begin with an underscore, followed by a lowercase letter. These names overlap those reserved to the implementor for naming external functions and data objects. That causes no trouble if you use such external names only within library modules that you write. But don't write macros that refer to them if the macros can be expanded in user code.

For example, here is a not uncommon way to implement `cos` and `sin`:

```
extern double _sinq(double, int)
#define cos(x) _sinq(x, 1)
#define sin(x) _sinq(x, 0)
```

The second argument to `_sinq` is an offset in quadrants to be added to `x` (in radians). It works fine so long as the user doesn't inadvertently define a macro named `_sinq`. Better you should name the secret function `_Sinq`, assuming that uppercase letters survive the linker. Otherwise, a name like `__sinq` is safest.

I find that two leading underscores makes for difficult reading. The secret names I use almost always have a leading underscore, followed by an uppercase letter. For macro names, I use uppercase letters and digits. For other names, the remaining characters are lowercase letters and digits. Whatever you do, pick a convention and adhere to it rigorously. Standard C headers and support code needs all the help it can get.

If you must write code that survives shortened external names, I have one additional piece of advice. Don't try to contrive meaningful names as short as the linker might like. You may need two leading underscores to avoid collisions with the user. That leaves you only four significant characters in the name. In such straits, you need names with two components. The first is a four-character code that is unique but only suggestive of the meaning. The second is a tail that is as long as necessary to be properly descriptive. For example, you might well prefer `__prtx_Generic_Print` to the laconic `__prtx`.

Conclusion

None of the issues I discussed here is particularly onerous taken alone. In concert, however, the issues raised by the C Standard can intimidate users and implementors alike. I have tried to help users by clarifying what rules apply only to highly portable code. Not everyone needs to fret to the degree suggested by the Standard, even for code that must be portable to some degree. I have tried to help implementors by grouping the issues differently and showing a few techniques that work. I know from personal experience that Standard C is eminently implementable.

You just have to be a bit tidier than in the good old days.

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!plauger!pjp.

∞

33. Numerical C Extensions Group

Rex Jaeschke
Convener

In November 1989, a letter ballot was conducted among members of X3J11 regarding the admission of NCEG as a working group within X3J11. Below, is a copy of the cover letter attached to that ballot.

> Dear X3J11 Voting Member,
>
> Early this year I announced the formation of the Numerical C Extensions Group (NCEG) in a letter to all X3J11 members. As a result of this and other promotion, NCEG has had two successful meetings (in May and September) and is continuing on its mission to produce a Technical Report. At both meetings, the NCEG membership expressed desire to become affiliated with a formal standards group, and X3J11 was identified.
>
> Enclosed you will find a letter ballot which has been issued to all eligible voting members of X3J11. You are requested to state your position regarding the admission of NCEG as a working group (tentatively called X3J11.1) within X3J11.
>
> Also enclosed are copies of the minutes from the first two NCEG meetings so you may view the work achieved to date. The enclosed Project Proposal contains the formal specification of NCEG's mission. This will be forwarded to SPARC for their consideration provided this letter ballot results in a two thirds majority.
>
> Since the intent of NCEG is rather specialized (aliasing is a general issue but needs to be addressed by NCEG too) you may find NCEG's work to have little or no effect on you or your implementations. If that is the case I urge you to vote in favor of admitting NCEG rather than abstaining unless, of course, you have good reason to vote *NO*.
>
> The ballot will last 30 days and if admission is approved, the results will be summarized and forwarded to SPARC by the Chair of X3J11.
>
> *Rex Jaeschke, NCEG Convener*

The letter ballot period ended in mid-January with 23 of the 50+ eligible members actually voting. Of those 23, 22 voted "*YES*, I support the admission of NCEG and the forwarding of the project proposal to SPARC" and one person voted *NO*. As a result, X3J11 Chairman Jim Brodie will forward the results of the ballot to SPARC along with the project proposal for their consideration.

The one dissenting vote is worthy of comment. Essentially, the objection was based on a section of the project proposal that hinted NCEG might violate the ANSI C Standard. The section was "Expected Relationship with X3, etc." and contains the following passage.

> The stated goal of NCEG is to be upwards compatible with ANSI C. However, it is conceded that in at least one area (namely error handling by certain library functions) NCEG may have to use an approach not sanctioned by ANSI C.

Note that the key word here is "may." NCEG is definitely *not* advocating it set out to "fix" or "break" ANSI C in this regard. However, more than a few knowledgeable NCEG members have indicated `errno` is a major thorn in their (or their customers') sides. Note too, that NCEG's mission is to produce a technical report, *not* a standard. Even if NCEG recommends an alternative error handling mechanism X3J11 is under no obligation to adopt it. In fact, I expect numerous NCEG efforts will result in implementation-defined or undefined behaviors due to conflicting prior art or failure to agree. That's OK too.

I have established an informal liaison with X3J16 (ANSI C++) and attended their first meeting in Washington D.C. on December 15th. I also expect to be attending an ISO C meeting in Europe in mid-June and will brief delegates on our activities and solicit their input.

The next NCEG meeting is scheduled in New York City, March 7–8, 1990 immediately following X3J11. The meeting time has been extended from one and a half to two days. The meeting after that is being proposed in the Livermore, California area (an hour east of San Francisco), probably in 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. The deadlines for the next two mailings are (approximately) April 1 and August 1.

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

∞

34. C Standards Update

Jim Brodie

Abstract

The American National Standard for C has finally been approved and we can now look to the future activities of X3J11. Interpretations of the C Standard and the International standardization of C remain to be addressed. Also, what is the future of C? It's often said that standards committees take so long that they standardize a language about the time it becomes obsolete. Is this the case with C? Has C++ become the language of choice in traditional C language application areas? These questions are examined and some potential answers supplied.

ANSI C Standard Approved

This is an article that I have been waiting a long time to write. On December 20, 1989 X3J11 received formal notification that the American National Standard for the C Programming Language was approved by the Board of Standards Review (BSR) of the American National Standards Institute (ANSI). This was a unanimous decision. This is the final step in the approval process for C. **We now have a standard for the C language!**

The Board of Standards Review considered the issues raised by Mr. Hansberry in his earlier appeals of the X3J11 and X3 processes. (These appeals delayed the standards approval process by about nine months.) In response to Mr. Hansberry the BSR wrote:

> The members of the Board of Standards Review, in taking this action, carefully reviewed and considered all of the documentation regarding this subject standard and found that, in accordance with ANSI procedures, due process had been followed and consensus achieved in the development of this standard. However, it was the sense of the members, in determining this action, that:
>
> 1. The Secretariat was responsible for a serious oversight in its handling of Mr. Hansberry's original public review comments. It is the strong recommendation of the BSR that the Secretariat institute administrative procedures to prevent a similar recurrence.

2. The Secretariat, after recognition of this oversight, did act to correct the oversight and did contact Mr. Hansberry in an effort to resolve his comment, in accordance with procedures.
3. The BSR recognizes that step 2 was conducted by the Secretariat in a conscientious and earnest effort to resolve this issue and commends the Secretariat for the same.

The period for appeals before enactment of the standard has now passed and actual publication of the standard should occur shortly. (ANSI anticipates copies being available by late March.) The Standard's official designation is ANSI X3.159-1989. To obtain a copy, contact:

American National Standards Institute
Sales Department
1430 Broadway
New York, NY 10018
(212) 642-4900
fax (212) 302-1286

At press time, the price of the standard had not yet been determined.

The X3J11 committee will continue to exist and be active despite the approval of the standard. (The next X3J11 meeting will be held on March 5–6 in New York City.)

Interpretations Phase

X3J11 will now turn its attention to the Interpretations phase, where questions about the meaning of the standard document are addressed. Although the committee's responses do not have the force of the standard, they do serve as guidance to the C community. The committee already has a small backlog of interpretation requests. Requests for interpretations cover a range of issues. For example, one request for interpretation asks:

> Do functions return values by copying? The standard is clear (§3.3.2.2) that function arguments are copied, but it is not clear (§3.6.6.4) whether a function's returned value is also copied. This question becomes an issue in the assignment statement `s = f();` where `f` yields a structure: is the result defined when the structure `s` overlaps the structure that `f` obtained the returned value from?

A second request asks:

> Are multiple definitions of unused identifiers with external linkage permitted? The wording of §3.7 permits multiple definitions of identifiers with external linkage, so long as the identifiers are never used. For example, the following program is "strictly conforming" if you take the wording in §3.7 literally:

```
int F() {return 0;}
int F() {return 1;}
int V = 0;
int V = 1;
int main() {return 0;}
```

This *must* be a bug in the wording of §3.7. It *cannot* have been the Committee's intent, since it prohibits the commonly encountered linker model. For example, most linkers will flatly refuse to link the following "strictly conforming" program.

```
/* x.c */

int F() {return 0;}
int G(int i) {return 1;}

/* y.c */

int F() {return 1;}
int G(int);
int main() {return G(0);}
```

because `F` is defined twice.

As X3J11 prepares responses to these and other interpretation requests, we will share them with you. If you would like to formally submit a request for interpretation, do so to:

X3 Secretariat
CBEMA
311 First Street N.W.
Suite 500
Washington, DC 20001-2178
Attn: Manager of Standards Processing

If you like, you may also send a copy to the editorial offices of *The Journal*. Since interpretations meetings will probably occur only twice a year, if you submit a copy to us we can take an electronic poll of X3J11 members between meetings to gauge their (informal) opinions and pass them on to you. Another source of informal assistance is the comp.std.c electronic conference.

The ISO C Standard

X3J11 will also be involved in helping ensure that a quality International standard for C is produced. The International Standards Organization (ISO) standardization process includes several ballots. As I reported in last September's issue of *The Journal* the first of these ballots was completed without any negative votes. The second (six month) ballot was started in December 1989.

Despite the lack of negative votes on the initial ballot, all is not quiet on the International front. The Danish delegation is still unhappy over the lack of readable alternatives when the local character set does not include all of C's special characters. They do not consider trigraphs to be a readable[1] or acceptable solution.

The British delegation feels that the document, as it currently stands, is not explicit enough in several areas, most notably when identifying constructs which have undefined behavior. The British are driving an activity in the ISO C programming standards committee (WG14) which will produce a *Normative Addendum* for the draft currently being balloted (which *is* identical to the ANSI standard). This addendum would be part of the ISO standard (and have the *full* force of the standard). It would supply explicit editorial clarifications to some of the areas that the British feel need to be addressed. A risk is that one of these "editorial clarifications" will end up being a substantive change to the standard. X3J11 will be working to make sure this *doesn't* happen.

The current plans call for the next ISO C committee meeting to be held in London on June 18–19 of this year.

The U.S. Government FIPS

There is not yet a Federal Information Processing Standard (FIPS) for C. Referencing a FIPS significantly simplifies the process of using a language in a U.S. defense project. Now that the ANSI standard for C has been approved, the C FIPS should follow soon. Sometime in Spring 1990, the FIPS proposal will be issued for a 90 day public comment period. Assuming there are no objections, the FIPS is presented to the U.S. Secretary of Commerce for signing. It is expected this whole process will be completed during the Fall of 1990. This is then followed by a six month acceptance period, after which the FIPS becomes final. Note, however, that since the ANSI Standard exists right now, a U.S. Government agency *can* require conformance to that standard in Requests For Proposal (RFPs) right now! They do not need to wait until a FIPS is produced.

The Formal Validation Process

As reported in the *Volume 1, number 2, September 1989* article by Neil Martin of the British Standards Institution (BSI), Britain, along with several other European standards organizations, has established testing procedures and has actually adopted a commercial suite.

On the U.S. front, activity in that direction has also begun in earnest. According to the National Institute of Standards and Technology (NIST) they

[1]When trigraphs were invented by X3J11 they were not necessarily intended to be readable but, rather, to aid mechanical translation to and from ISO-646 environments.

issued an RFP in late January asking for submissions of C language validation test suites. To get a copy of the solicitation for C language validation suite RFP (Solicitation Number 52SBNB0C6O42) contact:

National Institute of Standards and Technology
Acquisition and Assistance Division
Building 301 Room B117
Gaithersburg, MD 20899

Note that the process of selecting or developing a validation suite is *not* related to the FIPS. That is, the suite may be finalized before or after a FIPS is produced.

Looking into the Crystal Ball

At this point, as we close one chapter in the history of C and open another, I think it is useful to take a look at what the future holds for the C language. It has been joked that standards committees take so long to complete their task that they standardize a language about the time it becomes obsolete. Since it took X3J11 six years to produce an approved standard, is this the case with C? Will the standardization of C be an item of more academic rather than practical importance? Has C++ become the language of choice in traditional C language application areas? If all of the object-oriented hype is correct, can a programmer developing in C hope to compete?

Many of you who read *The Journal* have made major commitments to the C language and the tools that support it. Now that you finally have a fixed target language, do you need to be moving off to support other languages?

To get a sense of the future of C and the C market, I talked to a cross-section of current language product developers (all of whom offer C related products) in the U.S. and Canada. I asked about the future of C and the demands of their marketplaces with regards to C and other languages, most specifically C++. In the remainder of this article I will share some of my own observations as well as those of the people I talked to.

Building from the Base

As a starting point, we should remember that even if no new C program development efforts were begun, starting today, there remains a very large body of existing code that must be maintained and enhanced during the coming years. There are many historical precedents to indicate that the massive investments in this C code will not be thrown away. It will not be abandoned any more than previous investments in BASIC, FORTRAN, COBOL, and Pascal have been thrown away.

Microsoft still has a very healthy business selling its BASIC interpreters and compilers. It is interesting to note too that the dominant language in the super computing world remains FORTRAN, despite all the wonders of C and other newer, “better” languages. There remain more lines of COBOL code than any other computer language.

Many large industrial efforts (e.g., the GTE GTD-5 telephone switching system and the Honeywell TDC 3000 Process Control system) which were begun in the late ’70s and early ’80s were written in Pascal. These products are still in the marketplace today and will remain there for years to come. (Some companies are now considering re-writing part or all of their Pascal programs in a language such as C++. However, the very size of the applications makes this a long and expensive process.)

If history is any indication of what the future holds (and it usually is), C *will* remain a significant player well into the 21st century.

New C Development Efforts

While there appears to be a secure “existing code” marketplace for C and C tools, this is not the exciting, growing market-place that companies typically like to be a part of. Will C continue to be used for new product development during the next 3–5 years?

The consistent answer appears to be *YES*. There are a variety reasons for this, many of which have little to do with the language’s features and capabilities.

The first reason for C’s continued use in new product development is that C remains the language that most programmers know and are comfortable with. People tend to stay with what they know unless there is a clear reason to change. In addition to the wealth of C experience there is a corresponding lack of experience with most of the new alternatives (e.g., C++). Right now it is simply much easier to find and hire experienced C programmers. This situation may change as universities begin to teach and use languages such as C++ on a consistent basis. However, this is likely to be a slow process.

C remains relatively easy to learn and use. One of the consistent comments about C++ is that there is a high learning cost (at least to become an “effective” or “real” C++ programmer). This will slow C++’s spread into many application area as a potential replacement for C.

Another reason for C’s continued use is that there is a wealth of high quality support tools from a wide variety of vendors. The competition in this arena has driven down prices and driven up quality. This has made the C market a difficult one for developers. However, the customers have benefitted tremendously.

C remains the language of choice when portability is a serious consideration. No other language can match C’s widespread availability. In addition, despite the fact that the standard has just been officially approved, the draft standard has been stable for over a year and a half. The C language, as defined by the ANSI Standard, is currently the de facto standard for much of the industry.

Several C products are already completely standard-conforming and many more, which are already quite close, will conform very soon. For example, the next release of UNIX from AT&T is reported to include a fully ANSI-compatible C compiler.

When considering how newer languages will impact the C marketplace we need to remember that change comes slowly in most parts of industry. Most engineering managers are very skeptical of new wonder cures like "object oriented whatever." They have all heard (and frequently paid for) the wonder cures of the past. They have spent big money, yet projects were still late and over budget.

Industry, unlike academia where it is relatively easy to move on to the next new idea, lives very closely with its past. The thought of moving to a new development language sends shivers down the backs of most software engineering managers. When they start to add up the costs of training, tools, lost productivity, replacement libraries, etc., they become very hard to convince that the gains will outweigh the costs. The existing inertia can be enough to kill any effort to move to a new language.

It is interesting that the role of C has changed so dramatically over the years. It has gone from being the upstart to being part of the conservative establishment.

The Object-Oriented Challenge

Perhaps one of the most interesting challenges to C will be the current industry emphasis on object-oriented programming. Many advocates argue that full blown object-oriented programming is the wave of the future that will engulf all of programming. If this is the case, then C might be doomed to extinction in the long term. I don't buy this scenario.

Another view is that object-oriented design and programming is largely an evolution of current practices. This view holds that the object-oriented approach supplies a new set of approaches and tools to address certain classes or kinds of problems. When these problems occur, the object-oriented approach can be a real help. In the cases where they do not occur, the object-oriented approach does not help and can even make things harder.

Larry Constantine, one of the leaders of the structured programming revolution, is an advocate of this alternative view. In a recent article *Objects, Functions, and Program Extensibility* (*Computer Language, January 1990*), Constantine shows a transaction processing example where an extension in the data domain (e.g., the kinds of transactions to be processed) was simpler if an object-oriented approach was chosen. However, an extension in the function domain (e.g., a new action supplied on each transaction) was simpler if a traditional function-structured approach was chosen. The key is to develop a program that supports the kinds of future changes which are likely to occur.

Once all the hype passes and we settle down to really understand the essence of the advantages that object-oriented programming brings, there will be a clearer understanding of exactly which application areas are well suited for the object-oriented paradigm and the associated languages. If there is no benefit in a particular application area, then the simplicity of C will probably ensure C's continued use.

What Markets will C Continue to Serve?

The question remains, "Are there marketplaces where C, because of features and simplicity, is simply the best language choice currently available?" Here again, the answer is consistently yes.

C remains the language of choice for much of the low level (e.g., device drivers) and system level programming market. C also appears to address the needs, while avoiding some of the complexities and high learning curves of some of the alternatives, of many tool applications. The full blown object-oriented programming paradigm does not appear to offer significant advantages in these areas (although many of the fundamental programming principles upon which object-oriented programming is based still apply, such as data abstraction and information hiding, just as they always have.)

C remains a strong choice in much of the embedded systems market. C remains the number one high level language choice in applications where efficiency, both in terms of time and space, is a high priority. This class of programs goes from real-time embedded applications (where response times are measured in micro-seconds) to non-embedded system applications (like Terminate and Stay Resident [TSR] utilities under DOS). This is partly due to the quality of the compilers that are available and partly due to the level at which C programmer's can specify their programs.

C will probably remain dominant for most application areas where the programs produced are "small." The C language facilities are completely adequate for small and medium size programs.

As you move into some of the large and very large systems, some of the limitations of C will make C++ an attractive alternative. C++ introduces a variety of complexities, but the additional facilities are very helpful when dealing with the inherent complexities of a large programming effort. At this point, it appears that C++ will make its first major inroads into the traditional C marketplaces among sophisticated users trying to build new large systems.

C has become a more viable alternative in some of the numerical application areas. The C Standard now allows true single precision arithmetic for `float` objects. This, along with the honoring of grouping parenthesis has allowed some programmers to move from FORTRAN to C for new applications. Although C is still a long way from providing a complete alternative for FORTRAN programmers, it offers enough to become a serious option for some. It will be interesting to see how the work of the X3J11's Numerical Subgroup expands this marketplace in the future.

The defense/government industry is an interesting marketplace for C. Although there has been a push for years to move to Ada there appears to be a renewed willingness by the Department of Defense to consider other language alternatives. This is particularly true with the adoption of IEEE's POSIX as an operating system standard.

C++ might be a better fit than C in many large defense software efforts. Here, however, the lack of an approved language standard will limit C++'s acceptance. It is interesting to note that the recently formed C++ language committee (X3J16) is currently planning to complete its work in about two years. Perhaps they will be successful in this aggressive schedule. However, past experience, including that of X3J11, is that it will take significantly longer than this before an approved C++ standard is a reality. One reason for my skepticism about a two-year schedule is that several key (and quite difficult) C++ areas remain to be developed before they can be standardized. These include exception handling and parameterized types. The lack of a body of existing code will make it difficult to evaluate the effectiveness or correctness of the proposed solutions.

One new area of application of C with considerable potential is that involving international applications. X3J11 spent considerable time and effort soliciting input from the international community before adopting the locale and multibyte support primitives. It is expected these hooks will boost C's use in Western European and Asian marketplaces considerably.

Conclusion

The C language has grown and prospered for the better part of 20 years. New challenges face the language and it will probably not grow into new application areas as rapidly as it has during the past 10 years. Some application areas, such as new, very large application developments will probably move to other language alternatives, most notably C++. However, C remains the language of choice in a wide variety of application areas, and will probably remain so for the foreseeable future.

Jim Brodie is the convener and Chairman of the ANSI C standards committee, X3J11. He is a Senior Staff Engineer at Honeywell in Phoenix, Arizona. He has coauthored books with P.J. Plauger and Tom Plum and is the Standards Editor for The Journal of C Language Translation. Jim can be reached at (602) 863-5462 or uunet!aussie!jimb. Rex Jaeschke contributed information about FIPS, NIST, and ANSI.

∞

35. A FORTRAN to C Translator, Part II

Fred Goodman
PROMULA Development Corporation
Columbus, Ohio

In Part I of this series, a *surface-form* description language was presented. In this, the second and final part, a detailed sample translation is presented.

A Sample Translation

This section gives a complete example, using the FORTRAN `OPEN` statement, of a translation and of the considerations needed to produce that translation for the various user output biases. The material presented shows the actual specifications and output results as produced by the current version (1.22) of PROMULA.FORTRAN. Before beginning this discussion it must be emphasized that though any user of PROMULA.FORTRAN has full access and control over the following specifications, the typical user accesses them in compiled form only and needs only specify his overall output bias.

The `OPEN` statement is selected because its operation should be clear to those not familiar with FORTRAN, while its syntax and implementation are sufficiently complex to make the points needed.

The FORTRAN `OPEN` Syntax

In this presentation a slightly simplified version of the `OPEN` statement is presented. The purpose of `OPEN` is similar to the C `fopen` function though it cannot be directly translated into `fopen`. The primary problem is that in FORTRAN the user assigns arbitrary integer handles—referred to as *logical unit numbers*—which he then uses to reference the file. The handle is not assigned by the runtime system. The syntax of the `OPEN` is as follows:

`OPEN (` [UNIT=] *u* [,IOSTAT= *ios*] [,ERR= *sl*]
[,FILE= *fin*] [,STATUS= *sta*] [,ACCESS= *acc*]
[,FORM= *fm*] [,RECL= *rl*] [,BLANK= *blnk*])

Where:

u	unit number	*acc*	access type
ios	error code	*fm*	record format
sl	error label	*rl*	record length
fin	file name	*blnk*	blank treatment
sta	file status		

The Runtime Library Structure

Suffice to say that the FORTRAN file system is clearly different than the standard C file system. Therefore, a basic set of utilities is needed in the runtime library to deal with the `OPEN` statement. These are as follows:

```
void fiostatus(iostat, error)
long *iostat;   /* Address of error status variable */
int error;      /* Error testing switch */
```

Description: If the FORTRAN I/O runtime system encounters an error, it sets an error code and calls function `fioerror`. The behavior of that function depends upon how the code using the I/O system is doing error processing. This function establishes the error code return variable based on the error testing switch passed in. If that switch is zero, then an error on a successive I/O operation causes an abnormal termination. If the switch is nonzero, then an error condition is set and a normal return is executed.

```
int fiolun(lun, action)
int lun;        /* Logical unit number of file */
int action;     /* Action code for subsequent use */
```

Description: Before any action can be performed on a FORTRAN file, the logical unit number must be associated with an existing FORTRAN file structure. If there is no already existing structure for the unit number, then this function will attempt to create one. The form of this creation depends upon the type of the action to be performed. These will not be detailed here. For `OPEN`, `action` is zero.

```
void fiofdata(option, string, ns)
int option;     /* Says which data is being specified */
char *string;   /* String information */
int ns;         /* String length or integer info */
```

Description: This function is used to specify the various file data options associated with the current FORTRAN file structure. The particular data being specified is defined by the `option` parameter. These will not be detailed here.

```
int fioopen()
```

Description: This function opens the file associated with the current FORTRAN file using the current specifications as established via previous calls to function `fiofdata`.

```
int fioerror(clear)
int clear;      /* Should error control be cleared? */
```

Description: If the FORTRAN I/O runtime system encounters an error, it sets an error code and calls this function. This function either sets an error return value or exits to the operating system with an error message. In the case where an error code is returned to the calling function, the parameter `clear` specifies whether or not the error processing control variables should be cleared prior to the return.

The Optimized Translation

Given the above syntax and the above runtime library structure, the optimal translation of the following FORTRAN fragment is as follows:

```
  SUBROUTINE DEMO
  INTEGER ERRCOD
  OPEN(1,STATUS='NEW',FILE='d.o',IOSTAT=ERRCOD,ERR=5)
  OPEN(2,STATUS='OLD',FILE='d.i',ACCESS='DIRECT',RECL=50)
5 RETURN
  END

#include "fortran.h"
void demo()
{
static long errcod;
        fiostatus(&errcod, 1);
        fiolun(1, 0);
        fiofdata(1, "d.o", 8);
        fiofdata(2, "NEW", 3);
        fioopen();
        if(fioerror(1)) goto S900;
        fiolun(2, 0);
        fiofdata(1, "d.i", 8);
        fiofdata(2, "OLD", 3);
        fiofdata(3, "DIRECT", 6);
        fiofdata(5, NULL, 50);
        fioopen();
S900:   return;
}
```

First the `fiostatus` function is called if the user is performing his own error processing. Next, the `fiolun` function is called to establish the logical unit number of the file, and the `fiofdata` functions are called to establish the various parameter values. Finally, the actual open operation is performed and if so requested a possible branch is taken if an error occurred. The above translation is optimal in that it gets the job done in the most efficient and straightforward way. It is the translation for those users who have no direct interest in the intermediate C output.

The remainder of this subsection shows how the above translation is achieved and alternate translations can be produced which are less optimal but which are more readable and more maintainable.

The OPCODES Component

The first step in any translation specification is the establishment of the intermediate language and of its operation codes. These codes tie the translation specifications together. The basic operations involved in the `OPEN` are as follows:

Op Code	*Description*
LDA	Load the address of a variable onto the stack
LIC	Push an integer constant onto the stack
STA	Establish the current error processing status
LUN	Specify the logical unit number
LSC	Load a string constant onto the stack
DFA	Define file access type
DFN	Define current file name
DFR	Define file record length
DFS	Define current file status
DSP	Push a dummy string parameter onto the stack
DIP	Push a dummy integer parameter onto the stack
OPN	Open the current file
ERR	Perform error testing
NER	End without error processing
JMC	Jump on condition
GTL	Go to line address

The LDA, LIC, LSC, DIP, and DSP operation codes are generic operations that place values on the stack. The ERR, NER, JMC, and GTL operations are the branching control operations needed to define the error branching. The STA, LUN, DFA, DFN, DFR, DFS, and OPN operations are particular to the operations of FORTRAN I/O processing.

It must be emphasized again that PROMULA.FORTRAN is a p-code compiler. These operation codes were designed to make it possible to execute FORTRAN programs via a pseudo-machine. The translation to C works completely independently of the compilation to the pseudo-code.

The `OPEN` Statement Description

To illustrate the point that the transformation to the intermediate opcodes as listed above is independent of the output production, below is the actual statement description for the `OPEN` statement as entered in the translation description file.

```
"OPEN" "(" ["UNIT="] u [( "IOSTAT=" ios)|( "ERR=" sl )|
 ( "FILE=" fin )|( "STATUS=" sta )|( "ACCESS=" acc )|
 ( "FORM=" fm )|( "RECL=" rl )|
 ( "BLANK=" blnk ) ]... ")" .
Where:
        u = integer_parameter
        ios = long*
        sl = statementlabel
        fin = string
        sta = string
        acc = string
        fm = string
        rl = integer_parameter
        blnk = string
Emissions:
        IF(ios & sl) EMIT(ios LIC 1 STA)
        ELSEIF(ios) EMIT(ios LIC 0 STA)
        ELSEIF(sl) EMIT(LDA 0 LIC 0 STA)
        EMIT(u LIC 0 LUN)
        IF(fin) EMIT(fin DFN)
        ELSE EMIT(DSP)
        IF(ios) EMIT(ios DFS)
        ELSE EMIT(DSP)
        IF(acc) EMIT(acc DFA)
        ELSE EMIT(DSP)
        IF(fm) EMIT(fm DFF)
        ELSE EMIT(DSP)
        IF(rl) EMIT(rl DFR)
        ELSE EMIT(DIP)
        IF(blnk) EMIT(blnk DFB)
        ELSE EMIT(DSP)
        EMIT(OPN)
        IF(sl) EMIT(ERR JMC 5 GTL sl)
        ELSE EMIT(NER)
End
```

Very briefly, statement descriptions consist of four parts:

1. The left-hand-side recognition symbol. In this case the `OPEN` statement is recognized when the token OPEN is found at the beginning of a statement.
2. The actual syntax description, which consists of formal symbols, such as `(`, `)`, `[`, `]`, {, }, `.`, `|`, etc., terminal symbols enclosed in quotes, and non-terminal symbols. The syntax is designed to look as much as possible like

the syntax usually used in FORTRAN reference manuals, but formalized to make it equivalent to contemporary Backus-Naur notation.

3. The *where* section, which specifies what each nonterminal symbol is. On the right-side of these specifications is either a type specifier—meaning an expression of the indicated type—or the identifier of another statement component description defined like the above.

4. The *emission* specification which specifies how the actual intermediate code is to be structured.

We are omitting from this discussion the description of the basic expressions, which uses an entirely different notation. The linkage between expressions and statements is made in the *where* section above.

The bulk of the above notation should be at least readable by those who are used to formal language descriptions. However, the emission section is unique to PROMULA.FORTRAN. Its purpose is to specify how the actual code sequences associated with the nonterminal symbols are to be combined in the context of this statement. In the conditional portions of the notation, a nonterminal symbol is true if it has any code associated with it, and false if not. Testing a conditional does not affect its code. In the EMIT clauses constants and operation codes are emitted exactly as entered. For nonterminal symbols, the code associated with that symbol is emitted. This notation is very powerful and is able to deal with most of the FORTRAN syntax—in particular the I/O statements.

The Biased Surface-Form Descriptions

Once statements have been processed into the intermediate language as defined by the opcode list using specifications of the statement syntax and emissions, the intermediate language can be output as C using the biased surface-form descriptions. As was discussed in Part I, there are three distinct views of how the C output should look.

The first bias wants an optimal C which is quick to compile and which minimizes additional runtime code. The translation for this bias was presented earlier in order to introduce the runtime library. The translation gives an optimal use of this library with no additional overhead.

The second bias wants a C code that can be easily read and maintained, but which follows C conventions as closely as possible. Our default translation under this bias is shown below.

```
#include "fortran.h"
void demo()
{
static long errcod;
        fiostatus(&errcod, 1);
```

```
                ftnopen(1, "d.o", 8, "NEW", NULL, NULL, 0L,
                        NULL, 0L);
                if(fioerror(1)) goto S900;
                ftnopen(2, "d.i", 8, "OLD", "DIRECT", NULL,
                        50, NULL, 0L);
        S900:
                return;
        }
```

The error processing logic is the same as before, but an additional interface routine `ftnopen` is introduced which combines all of the possible parameters to the OPEN into a single function call with a fixed parameter list. This function itself then calls the `fioopen` and `fiodata` functions based on the parameter list. This implementation is clearly easier to read, but it requires slightly more runtime code.

The third type of user wants to retain as much of the original FORTRAN flavor as possible. He is used to maintaining the FORTRAN code and he wants to make the transition to C as painless and error-free as possible. For this user, the C output is as follows:

```
        #include "fortran.h"
        #include "ftnsymb.h"
        void demo()
        {
        static long errcod;
                if(IO_ERR(OPEN(1, O_IOSTAT, &errcod, O_FILE,
                  "d.o", 8, O_STATUS, "NEW", 0))) goto S900;
                OPEN(2, O_FILE, "d.i", 8, O_STATUS, "OLD",
                  O_ACCESS, "DIRECT", O_RECL, 50, 0);
        S900:
                return;
        }
```

Note first that an additional file `ftnsymb.h` is included. This file contains definitions of various constants intended to remind the user of the original FORTRAN. Also a new function `OPEN` is introduced, which is defined as `fiofopn` in the include file. This function takes a variable number of arguments, each preceded by an integer code with the last argument being zero. The integer codes themselves are represented by the symbols `O_IOSTAT`, `O_FILE`, etc., in the include file. In addition, the `OPEN` function itself returns the error exit condition so that it can be nested in the `IO_ERR` function which is simply defined as `fioerror` in the include file. The advantage of this translation is that there is a one-to-one correspondence between input statements and output statements and that the form of the C is at least reminiscent of the original. The disadvantages are that an additional runtime function is needed and extra time is needed to compile the additional symbols.

Having presented the three biased outputs, all that remains are the surface-form descriptions themselves. (See Part I for more details.)

```
DFS PATTERN
    F  3, 0, "%1d,O_STATUS,%2d"
    C  2, 0, "%1d"
    *  2, 0, "fiofdata(2,%1d,%2d);\c"
LUN PATTERN
    F  3, 0, "%2d%1d"
    C  2, 0, "%1d"
    *  2, 0, "fiolun(%1d,%2d);\c"
LSC PATTERN 0, 0, "\ks"
DFA PATTERN
    F  3, 0, "%1d,O_ACCESS,%2d,%3d"
    C  2, 0, "%1d"
    *  2, 0, "fiofdata(3,%1d,%2d);\c"
DFN PATTERN
    F  3, 0, "%1d,O_FILE,%2d,%3d"
    C  2, 0, "%1d,%2d"
    *  2, 0, "fiofdata(1,%1d,%2d);\c"
DFR PATTERN
    F  2, 0, "%1d,O_RECL,%2d"
    C
    *  1, 0, "fiofdata(5,NULL,%1d);\c"
DSP PATTERN
    C  0, 0, "NULL"
    *
DIP PATTERN
    C  0, 0, "0L"
    *
OPN PATTERN
    F  1, 0, "OPEN(%1d,0)"
    C  8,0,"ftnopen(%1d,%2d,%3d,%4d,%5d,%6d,%7d,%8d);\c"
    *  0, 0, "fioopen();\c"
DFF PATTERN
    F  3, 0, "%1d,O_FORM,%2d,%3d"
    C  2, 0, "%1d"
    *  2, 0, "fiofdata(4,%1d,%2d);\c"
ERR PATTERN
    F  1, 0, "O_ERR(%1d)"
    *  0, 0, "fioerror(1)"
NER PATTERN
    F  1, 0, "%1d;\c"
    C  1, 0, "\c"
    *  0, 0, "\c"
```

As described before, in the above descriptions those operations that have a single set of pattern specifications use the same pattern for all output biases. Those that have multiple patterns use the one specified for a given bias type if present, else they use the last pattern introduced by the '*'. Consider the OPN operation itself.

For the FORTRAN bias the following pattern is used:

```
1, 0, "OPEN(%1d,0)"
```

In this form, by the time OPN is executed by the string-machine, there is a single pattern string on the stack, which contains the concatenation of the other parameters as they occurred. This string is entered into the expression followed by a zero, which terminates the parameter string. The string thus formed is not yet cleared to the output file since it may yet be concatenated into the error function call.

For the C bias the following pattern is used:

```
8, 0, "ftnopen(%1d,%2d,%3d,%4d,%5d,%6d,%7d,%8d);\c"
```

In this form, by the time the OPN is executed by the string-machine, there are eight independent parameters on the stack. It is interesting, but not really important, that the order of the operators in the string-machine varies wildly depending upon the output bias. The pattern string concatenates the eight parameters into a single function call. The stack is then cleared, since the error processing is performed via a separate statement.

For the optimized bias, the following pattern is used:

```
0, 0, "fioopen();\c"
```

In this form, the stack has already been cleared prior to the execution of the OPN operation by the string-machine. There are no parameters. The operation itself merely writes the appropriate function call and clears the string stack.

The additional pattern strings may be analyzed in the same manner.

Other Bias Switches

In addition to the actual statement translations, which are controlled via the surface-form description language, there are several other translation features and optimizations which can be individually controlled by the user. Each has a default setting for the three output biases, though these settings can always be overridden by the user. These are best described via an example. Consider the following FORTRAN subroutine which computes the mean and variance of a set of values. This is followed by the optimized bias C translation. All flags are on for the optimized bias.

```
      SUBROUTINE EX001(VAL, N, XBAR, VAR)
      DIMENSION VAL(N)
      XBAR=0.0
      VAR=0.0
      DO 10 J = 1, N
      XBAR = XBAR + VAL(J)
   10 CONTINUE
      XBAR = XBAR/N
      DO 15 J = 1, N
      S = VAL(J) - XBAR
      VAR = VAR + S*S
   15 CONTINUE
      VAR = VAR/(N-1)
      RETURN
      END
```

```
void ex001(val, n, xbar, var)
int n;                                              (1)
float *val, *xbar, *var;
{
auto int j;
auto float s;
        *xbar = *var = 0.0;                         (2)
        for (j=0; j<n; j++) *xbar += *(val + j);(3)-(7)
        *xbar /= n;                                 (5)
        for (j = 0; j < n; j++) {                   (3) (4)
                s = *(val + j) - *xbar;             (7)
                *var += (s * s);                    (5)
        }                                           (8)
        *var /= (n-1);                              (5)
}
```

Note 1: The parameter `n` is not declared as a pointer, since it is not changed within the routine. PROMULA.FORTRAN uses *prototypes* of subprogram arguments so that it can generate optimal calling sequences. These prototypes may be specified by the user or may be determined internally by the translator. The above was internally determined by the translator. When this particular feature is turned off, all parameters are passed as pointers. This is the default for FORTRAN and therefore it is turned off for that bias.

Note 2: C allows multiple assignments of the same value to be written together. The translator looks for such assignments and combines them whenever possible. Since the FORTRAN bias wishes to maintain a one-to-one correspondence between input and output statements this feature is off for the FORTRAN bias.

Note 3: In FORTRAN the default base for a subscript is 1. Thus, all do loops which generate subscripts tend to start at 1. In C, however, subscripts start at zero. This fact makes for much more efficient code. The translator looks for `DO` loops whose only purpose is to move through array subscripts and reduces their range to start at zero, thus producing a very natural looking `for` statement and optimizing subscript expressions. This feature is left on even for the FORTRAN bias, since it makes the subscript expressions look more natural.

Note 4: C has `++` and `--` operators which take advantage of the fact that most computers have increment and decrement operators. The translator uses these operators whenever possible. This is turned off in the FORTRAN bias.

Note 5: C has operators like `+=`, `-=`, `*=`, `/=`, etc. The use of these operators ensures that the address of the left-hand-side of the assignment will only be computed as often as necessary. PROMULA.FORTRAN uses these operators, except in the FORTRAN bias.

Note 6: The `DO` loop running to statement 10 in the FORTRAN code is collapsed into a single compound statement, and that the now unneeded statement label is removed. This feature is retained in the FORTRAN bias.

Note 7: the subscript expression for the `val` vector is converted to the C pointer notation, which we have always preferred. The alternative notation using brackets is apparently preferred even by many C programmers. The flag is still by default on, for all but the FORTRAN bias, but we may just turn it off for everyone.

Note 8: Though the `DO` loop statements in loop 15 cannot be reduced to a single statement, the statement label can still be removed. Again this feature is the default in all biases.

Conclusion

The intent of this paper was to give a clear view of a problem faced by translation software which is not faced by other compilers. When the target language is human-readable, then humans want some say in its form. This fact greatly complicates the design of the software.

Fred Goodman is a mathematician/linguist and the author of PROMULA. FORTRAN. He is currently applying the translation methodology discussed here to BASIC, PASCAL, and COBOL. Fred is at (614) 263-5454.

∞

36. Electronic Survey Number 3

Compiled by **Rex Jaeschke**

Introduction

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

The following questions were posed to 50 different people, with 19 of them responding. Since some vendors support more than one implementation, the totals in some categories may exceed the number of respondents. Also, some respondents did not answer all questions, or deemed them 'not applicable.' I have attempted to eliminate redundancy in the answers by grouping like responses. Some of the more interesting or different comments have been retained.

Definition and Representation of NULL

How do you define `NULL`*? Is the null constant pointer actually represented as all-bits-zero? What different internal pointer representations do you support?*

- 3 – `#define NULL (void *) 0`
- 9 – `#define NULL 0`
- 1 – `#define NULL 0L`
- 16 – All-bits-zero
- 1 – Not all-bits-zero
- 10 – Only one pointer representation
- 3 – Two pointer representations
- Comments:
  1. As an extension, we support long pointers, declared with `^` instead of `*`. They are 64-bit space-offset pairs.
  2. `NULL` must be `#define`d as `0` because some existing applications have misused this definition, and would break if it changed.

3. On my word-addressable architectures, pointers to aligned objects are word addresses while `char *` typically adds a byte index within the word via additional high-order bits (low order still being the word address).
4. All pointers visible to the user are byte addresses, but internally we use word addresses to pass large structures and unions by value.
5. Although not necessary for any of our present implementations, there are (at least) two pointer sizes understood by the common front-end/code-generator.
6. On our 24-bit word machine we support two formats, one for character pointers, and one for pointers to everything else. All pointers are 24 bits in size, but in a character pointer the most significant two bits indicate which byte in a word to use.

Importance of the ANSI C Standard

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.

- 0 – Not at all concerned
- 6 – Somewhat concerned
- 11 – Very important or absolutely necessary
- Comments:

1. Provable? Nonsense! Demonstrable via some kind of test suite? Important. Claimable? Absolutely necessary.
2. Somewhat, but people are still confused about what ANSI standard conformance means.
3. Our applications require either ANSI C or System V C support; dealing with other C variants would be too much of a support burden. Computer procurement specifications will start requiring ANSI C once vendors have had a fair amount of time to prepare, and *no* deviation will be tolerated.
4. Conformance is probably less important than efficiency (in our parallel target environment.)
5. As soon as there is really a standard (either ANSI or ISO) it will be important. When there is a FIPS (US Federal Information Processing Standard) it will be vital even to non-government sales.
6. Very important to absolutely necessary (depending on the particular customer.)

7. ANSI C conformance used to be a big deal. Now I'm looking toward conformance with AT&T C++ 2.0.
8. Having an ANSI C conforming compilation is very important for the market, but in practice, is not the compilation mode used. The problem is the name space restrictions which make it impossible to compile existing code. Practical conformance to other "standards" (e.g., POSIX) also makes strict adherence to the ANSI name space rules essentially impossible.
9. It's very necessary since we want to bid on government contracts.
10. Somewhat but mostly for government bids only.

Validation Suites Used

Do you have a home-grown or commercial validation suite? If so, which?

- 2 – No suite
- 10 – Home grown
- 2 – ACE/HCR
- 1 – Metaware
- 4 – Perennial
- 10 – Plum Hall
- Comments:
    1. No. That is what users are for.
    2. We also have Rex Jaeschke's preprocessor and header test suite (available from Rex under license at no charge).
    3. Not yet but we're looking at the ACE/HCR suite and using Plum Hall's Sampler.
    4. I've found that commercial suites tend to only test the parser, not the optimizer or code generator.
    5. We also use the X/OPEN XPG3 Suite.

POSIX and IEEE Floating-Point Standards

Is POSIX conformance an issue for you? What about IEEE floating-point standards support?

- 13 – POSIX conformance is important

- 4 – POSIX conformance is not important
- 12 – IEEE floating-point conformance is important
- 4 – IEEE floating-point conformance is not important
- Comments:
    1. IEEE 1003.1 conformance will become more important in the future. Our applications rely much more heavily on SVID than on POSIX. I think the other 1003 so-called standards are completely berserk.
    2. Our applications neither need nor want IEEE floating point features. As an experienced numerical software developer, I disagree with the whole philosophy behind IEEE 754.
    3. Isn't life hard enough for us ANSI compiler writers without having to worry about other standards too?
    4. POSIX is vital because of the FIPS, and corresponding standards in the European market.
    5. Floating-point speed has been more important than IEEE support, though this is starting to change.
    6. POSIX conformance is just as important as ANSI C conformance. Our math libraries go out of their way for accuracy in supporting IEEE fp standards.

Shortcomings of ANSI C

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)?

1. *Editor's note: Each of the following 16 comment sets represents the input from one person. As far as I can tell, at least 13 of the respondents were either represented on X3J11 or followed its activities closely. As the marketing person said, "We have to leave something for the next release."*

2. The worst thing about it is that nearly all extensions are prohibited.

   The fact that `const **` pointers cannot be assigned without a cast from ordinary pointers is annoying. Many users are confused by this.

   Then there is the fact that a function that returns a structure value which it gets through a pointer must use `bcopy` to copy it into the return area, because the function value is not an lvalue and is allowed to come from a place that overlaps the place where the value is being stored.

3. I think that most of our headaches will relate to the fact that ANSI C is a slightly different language from our current PCC implementation.

4. Not many shortcomings. The various weasel-words to avoid making decisions is the worst source of trouble. One gets used to them. Also, the preprocessor stage could have used grammars more effectively. The one in the X3J11 report is misleading but this is not a big deal.

5. The inability to partially initialize data structures. UNIX I/O seems to be important among our users (and we're not UNIX-based), but missing from the standard. Also the `sleep` and `fgetname` functions.

6. The use of a global state in library functions, particularly `errno` in the math library, causes a variety of problems.

   Lack of adequate array support, particularly for matrix parameter variable dimensions, unduly burdens numerical applications.

   Lack of `typeof(object)` makes generic macros hard to write.

   The six-character limit on portable external identifiers (more than the monocase aspect) really gets in the way of programming. I understand why it's specified that way, but it *is* a deficiency in the language specification.

   Multibyte character sequences are unwarrantedly difficult to program for and should not have been sanctioned; a simple scalar data type for large character code would have been far better. To avoid having to have a parallel set of `str`* functions or a `strcoll` kludge, that data type should have been `char`, with another type such as `short char` introduced to allow access to smaller objects such as bytes or bits. As it now stands, the traditional, straightforward way to write text-handling programs doesn't work in an international environment, but it would have worked if a better approach had been adopted. I think this is a case of system analysis failure due to taking too literally what the users' detailed perceptions of their needs were, coupled with vendor commitments to existing poorly engineered solutions.

   The constraint on the character code set that `'0'..'9'` be represented in a certain way was unwarranted and outside the proper scope of X3J11. Other solutions could have been found to address that issue.

   Note: We do *not* need standardized access to special hardware features such as multithreading, vector processors, etc. I think rigid standards in such areas are harmful to the evolution of computing as a whole. Certainly these are not essential for the production of portable applications.

7. The biggest problem I see has to do with modularity, data hiding, and the like. I find it increasingly hard to write good modular, self-documenting code, with everything hidden, etc. I wouldn't mind nested functions, operator overloading, macros with a variable number of arguments, and macros that can deal with every possible type of argument (a `typeof` operator, perhaps). Hmmm ... I guess what I'm saying is that I want C

to become C++. I've never used C++ but feel the need for something like it.

The second biggest problem would be the lack of support for multiprocessors, vector processors, and the like. Very much understandable though, given the lack of common practice.

And, of course, the issue NCEG is tackling: making C a replacement for FORTRAN in the scientific world.

8. Biggest: Not a standard yet because of the blatant unresponsiveness of X3J11 to international input (or anyone else's, for that matter!).

   Second: Waffles too much around the edges on the functions.

   Third: Namespace rules are too tight to allow for backwards compatibility, yet not even guidance was provided in that area. (`__STDC__` isn't done well enough.)

9. It's very complicated—it will be hard for beginners to learn things like `const` and `volatile`.

   Also, the automatic coercion on function call is dangerous. For example, if you have

   ```
           extern int abs(int);
   ```

   then `abs(-5.4)` very unexpectedly yields 5 or 6.

10. Its internationalization capabilities.

11. As a programmer: `varargs` macros.

    As a compiler writer: the grouping rules (which say they are like other languages but aren't like FORTRAN), and the lack of something like `noalias` makes optimization with pointers almost impossible.

12. The biggest problem is that it is now impossible to make a non-tokenizing preprocessor. The preprocessor now must have far more complexity than before. The committee stepped out of bounds on this one, by inventing features and standardizing them before trying them out in an implementation.

13. It would be nice if there were a way to handle the great optimization needs that the "`noalias` wars" made so evident, but I'm hoping for a clear direction to come from future implementations. Also, I'd have been happier with a system that allowed type qualifiers to be added as necessary at any type depth instead of the existing one level scheme. I'm convinced that the next version of the standard will have to loosen the rules along these lines.

14. Too many machine- or implementation-dependent features and poor floating point support.

15. The standard never solved C's aliasing problems. Ultimately this limits the effectiveness of global optimizers since they have to make very pessimistic assumptions about the effects of pointer indirections. This in turn forces the use of non-standard extensions to support vectorization and just to improve global optimizations. Something like the proposed `noalias` extension (but better thought out) would have been useful.

    Left too many area's with implementation-defined behavior.

    Preprocessing numbers are broken.

16. Since the naming conventions don't match existing practice (at least in UNIX environments) they might as well have gone further towards making all implementation support routines whose names start with strange characters (e.g., use a `__` prefix for everything).

17. While I see no major shortcomings in the Standard, I would have liked to see something in the Rationale justifying the change in the behavior of the preprocessor from that of the "standard" UNIX character preprocessor.

Representation of `long double`

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?*

- 5 – `float`, `double`, `long double` all different
- 10 – `float` and `double` different, `long double` maps to `double`
- 1 – `float` and `double` the same, `long double` different
- 0 – `float`, `double`, `long double` all the same
- Comments:

    1. My guess is we are going to provide a lot of compiler flags and do everything.

Future Polls

Some of the topics planned for future polls are:

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

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

```
sizeof (char *) 0
sizeof (double) -1
sizeof (unsigned) &p
```

- Do you permit the output of the preprocessor to be saved for inspection? Is all white space (including comments) retained? Do you provide a way to merge or easily compare this output with the original source? Do you have a way to indicate which source lines were selected and skipped over as a result of conditional compilation?
- Do you recognize `main` as a special function and treat it differently than other functions?
- Can `main` call itself recursively in your implementation?
- Given that trigraph processing can slow down compilation do you (plan to) have a compiler option to disable trigraph recognition?

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

∞

37. Parallel Programming: Linda Meets C, Part IV

Jerrold Leichter

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, as well as the implementation of a compiler for our dialect, VAX Linda-C. In this final part, we discuss the `eval` operator, which is used to dynamically create new processes.

Introduction

In Part I of this series, we described a parallel programming language as one supporting process creation and coordination in the same way that traditional languages support sequential operations like looping. The operators we have discussed in detail up until now — `in`, `out`, and `rd` — support communication and coordination, but not process creation.

In Part I, we mentioned the `eval` operation, which is used for exactly that. Recall that `eval` is a variation on `out`. Like `out`, it inserts a tuple into tuple space. However, the tuple is *unevaluated*. For example, if `g` is a function producing a `float` then

```
        eval(g(x)+5.)
```

produces an *active tuple*:

$$\boxed{\langle \texttt{g(x)+5.}_{\texttt{float}} \rangle}$$

This tuple cannot be matched by any `in` or `rd` operation, and begins evaluating as soon as it is created. In our example, when `g` completes execution and returns (if it ever does), the active tuple becomes an "inactive" tuple, specifically,

$$\langle \texttt{(g's}\ \text{value}\ \texttt{+ 5.)}_{\texttt{float}} \rangle$$

This tuple can be matched in the usual way.

The Difficulty With `eval`

The `in`, `out`, and `rd` operators cannot be implemented directly as function calls as they depend on the types of the tuple field to which they are applied. Once that information is available — whether provided explicitly by a programmer or determined by a compiler — it can be passed in a function call along with the tuple fields. With minor variations, this is just the strategy chosen by all the implementations of which we are aware.

This strategy, however, cannot be generalized to `eval`. Any C function call will evaluate its arguments before passing them, and the whole point of `eval` is to avoid such evaluation. We'll need a different approach.

Our inability to use a function call should be seen as a hint to search for other problems; after all, function evaluation is central to the semantics of traditional languages. Such a search quickly reveals a difficult semantic issue involving variable scoping. In LINDA-C, we had to make some minor adjustments to the semantics of C expressions used as tuple fields. In particular, we had to reverse C's bias toward addresses and away from arrays. The problems with `eval` are more pervasive: C at least supports arrays, not just pointers; but there is nothing at all in C corresponding to an unevaluated expression tuple.

While these problems may seem obvious in retrospect, in practice they were ignored in favor of quick experimentation. We'll begin by examining some of those experiments.

Investigators

The earliest implementation of `eval` was done by Robert Bjornson at Yale University several years ago. He has since done several additional implementations, which will form part of his upcoming dissertation. Some of Bjornson's recent work has been done in conjunction with Nicholas Carriero. As far as we are aware, none of this work has yet been published.

We explored the problems surrounding `eval` in our dissertation[2] and described, but did not implement, the stack frame approach we describe later. We also suggest that `eval` is too blunt an instrument and that better alternatives are available. Discussing them here would take us too far afield, however.

Simple Approaches: "Let Joe Do It"

On UNIX systems, a simple approach is available to Linda implementers: punting the problem to the implementers of the operating system. We simply rewrite

```
        eval(tuple);
```

as

[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.

```
if (fork() == 0)
{       out(tuple);
        exit(0);
}
```

A variation of this theme is available on some systems, particularly shared-memory multiprocessors. Rather than using `fork()`, we can use some equivalent call from a light-weight threads package. This is likely to be much faster. Bjornson's experiments using `fork()` gave reasonable results with tens or perhaps hundreds of `eval`'s. His experiments using threads worked well with many thousands.[3]

All such implementations are obviously limited to a single shared-memory system. They also leave effective control of the created processes in the hands of the operating system.

Semantics

The semantics of an implementation based on `fork()` is fairly straightforward. We need a bit of language to describe it, however. Call the totality of variables accessible to a program, and their values over time, an environment. There are two environments of interest here: The *creating* environment within which the `eval` statement itself is executed, and the *execution* environment within which the field values are evaluated. For a `fork()` implementation, the two environments are initially identical, but disjoint. Any subsequent changes to either environment have no effect on the other.[4]

When we move on to thread implementations — which at first appear to be simple optimizations of `fork()` implementations — things become much more complex, since threads generally share at least part of their environments. Typically, variables on the stack are copied, and all others are shared. The exact details are implementation-dependent.

The result is implicit and sometimes unexpected communications paths. Writing code for such an implementation requires that the programmer either avoid using any parts of the shared (read-write) environment, or write a program that uses shared memory communication in addition to the tuple space operations.

Joe Doesn't Do It Too Well

While `fork()` and thread implementations are simple, they have few other virtues. In the case of `fork()`, the semantics are clear but an efficient implementation is difficult. Thread implementations have complex and varying

[3]Bjornson eventually implemented a fairly portable, if rudimentary, threads package using `sigstack` to create and switch among thread contexts.

[4]This description ignores "external" parts of the environment, such as files. The fact that UNIX-level file descriptors are shared but `stdio`-level file pointers are not in any practical sense leads to no end of confusion.

semantics, to the point that Linda implementations using them, in the time-honored tradition of experimental lash-ups, have a semantics best described as "it does what it does — try it and see."

Neither style of implementation extends beyond shared-memory multiprocessors, and both cede control over critical issues such as how much of the creating environment to pass, how many actual processes to create, and so on, to an external package.

Explicit Environment Passing

An alternative approach is to make all environment passing explicit. In this style, we re-write

`eval(`*tuple*`);`

as (roughly) the following:

`out("eval-request",`*func-id*`,`id_1`,`id_2`,...);`

Here, the id_i's are all the identifiers occurring anywhere within the original tuple, and *func-id* somehow identifies a unique piece of generated code which must be added to the program. The code itself is simply a function with arguments id_1,id_2,. . . , and having the single line

`out(`*tuple*`);`

Additional code is needed to create a "server program" that repeatedly `in`'s `eval-request` tuples, determines the function identified by the *func-id* received, and calls the function with the arguments received in the tuple. There are many details to be dealt with, beginning with the such trivialities as the need for a means to receive tuples containing a variable number of id_i's. None are particularly challenging, and it is beyond our scope to explore them.

This approach has much to recommend it, and has been used by Bjornson and Carriero in their recent work. Since all communication is through tuple space, it will work anywhere the underlying tuple space implementation works — there is no implicit communication through a shared memory. Conversely, there are no possible hidden communication paths between the creating and execution environments. The implementer retains complete control over what in the environment will be passed, and over the processes to be used to provide execution environments.

There remain, however, questions of semantics.

What Should Be Transferred?

Consider the Linda program in Figure 1. Which of the `eval` operations do we

```
 1  int             x1;
 2  int             x2;
 3  static int      s;
 4
 5  LindaMain()
 6  {       int     l;
 7
 8          eval(l);
 9          eval(s);
10          eval(x1);
11          eval(f());
12  }
13
14  f()
15  {       return(x2);     }
```

Figure 1: Examples of the `eval` Operation

wish to support? Ideally, *all* of them — and, indeed, the `fork()` and thread approaches would allow us to. But other approaches have problems.

The operations of lines 8 through 11 in Figure 1 successively require larger and more comprehensive definitions of what should be transferred between the creating and execution environment. Transferring more may make things simpler for the programmer, but in general will be more difficult to implement efficiently. We will consider the various operations on a case by case basis, attempting to find compromise semantics that are clear and useful while remaining implementable.

- Line 8 requires the value of an integer stored in a local variable be transferred. The actual value in this case is small, a single integer, but the same principles would also have to be applied to "large" values. If `l` were instead an array of 1000 integers, the entire array would, in general, have to be transferred.

  Clearly any useful implementation must support the ability to use some kind of variables in fields of `eval` operations; the ability to create new processes to evaluate constants seems rather pointless! Local variables are at least as easy to support as global ones, as we will see, so we might as well consider the ability to use local variables in `eval`'s a basic requirement.

- Lines 9 and 10 refer, respectively, to static and globally-known variables. We could transfer them as well. However, the utility of doing so is less than for local variables. Static and global variables are used for two main purposes in C programs. To provide implicit communication among a set of procedures (whether within a single file or across an entire program), and to retain state across successive calls to a procedure. Transferring the values of such variables serves the first of these purposes only partially, and the second purpose not at all. Except in a thread implementation, communication can only be one-way. From the creating to the execution environment. If programmers try to use static and global variables in the usual ways, the lack of communication back to the creating environment is likely to cause obscure bugs.

We don't argue that there is any *fundamental* difference between local and static or global variables here; but in practice there *is* a difference in the way they are perceived and used by the programmer. In C, a variable local to a procedure can be changed only by code lexically within that procedure[5], and the value is meaningful only as long as the procedure remains in execution. Neither of these are properties of static or global variables. As a result, it seems reasonable to distinguish between local variables on the one hand and static or global variables on the other.

- So far, we have dealt with situations in which a static analysis of an `eval` expression is sufficient to determine the entire context potentially required to evaluate it. Line 11 poses a more difficult problem: The `eval` expression refers only to `f`, but the evaluation of `f` requires access to a global (equivalently, static) variable, `x2`. This example makes it clear that providing access to static or global variables explicitly referred to in an `eval` expression is merely the first step. It is difficult to rationalize a design in which the values of global or static variables are available within `eval` expressions — but only within the expression themselves, not in procedures called from them. In C, it is possible to replace any expression by a call to a function containing the same code, with perhaps trivial re-writing to allow modified variables to be passed by reference. Conversely, any function call can be replaced by the code for the function, again with trivial re-writing to avoid name clashes. If we chose to support the examples of the previous item, but not examples like this one, both of these properties would be lost for functions containing references to static or global variables not otherwise referred to in the `eval` statement. This would mean that replacing a macro definition by a call to a real function could break working code.

  Given the nature of C it is impossible, except in unusual circumstances, to determine a restricted set of procedures, and thus global or static variables, accessible as `x2` is here via a procedure call chain starting at the `eval` expression. Hence choosing to support `eval` expressions such as the call to `f` that we have here would require copying the entire global and static data space of the executing program. Also, it means that it is impossible for the compiler to even determine that code is vulnerable to the kind of bugs we discussed above. On the other hand, it can easily diagnose references to static or global variables within `eval`'s.

  Note that, because C does not support nested procedures, local variables

[5]Unless its address is passed to another procedure.

behave very differently from statics and globals: Locals in `f` are not a problem, since they can receive no value from the environment at line 11. Further, `f` cannot refer to any of the locals, such as `l`, that are in scope at line 11.

A Semantics For `eval`

Synthesizing these ideas, we obtain the following semantics:

- The values of all local variables in scope at the point of the `eval` operation are copied to the execution environment.
- The values of static and global variables are taken exclusively from the execution environment. No value is passed for them from the creating environment. While the *values* of static and global variables are not passed between the environments, it is guaranteed that the same *set* of such variables is available in both environments.
- The creating and execution environments execute the same code. Further, at some point, they start out with the same initialization code. That is, the same initializers for static and global variables were executed in both environments at some point. If there is any code within either environment that can modify those initialized variables, it is unpredictable how many times that code has been executed since the initialization was done.

  Note that function names in C are static or global read-only variables, initialized to point to the entry points of the functions they name. The rules of the previous paragraph ensure that the functions they name are always the same in both environments.
- In no case is any information passed back from the execution environment to the creating environment, except by explicit tuple space operations.

Environments As Stack Frames

If we accept the semantics of the previous section, another implementation alternative exists. Rather than passing the individual variables actually used in the environment, we can copy the entire stack frame. In many cases, the difference in the amount of data passed will be small. Further, on many hardware architectures we can pass the saved program counter as part of the stack frame. Where this can be done, there is no need to generate a function to evaluate the tuple — it can be done "in place." In sketch, such an implementation translates

```
eval(tuple);
```

into

```
if (SendEval())
{       out(tuple);
        Idle();
}
```

The functions `SendEval` and `Idle`, which are similar to `setjmp` and `longjmp`, constitute the control code of the evaluator. `SendEval` packs up its caller's stack frame, including the saved program counter, sends it off in a tuple of the appropriate form, and returns `0`. In the creating environment, this return value causes the `if` statement to skip the `out`.

A server program in its idle loop retrieves a tuple sent by `SendEval`, pushes the contained stack frame, and simulates a return of `1`. In the resulting execution environment, the `if` statement proceeds to evaluate the `out` and then calls `Idle`. `Idle` clears both the call to itself and the frame pushed by `SendEval` from the stack and jumps to the idle loop.

Passing Pointers

The implementation we have just outlined has one significant limitation. If some local variable is a pointer, the value transferred will be an address that will not generally be meaningful in the execution environment. If we had used explicit environment passing in LINDA-C, the generated `out` would have either converted the pointer to an array reference and transferred the array — if the pointer had been declared `varying` — or produced an error message. A complete stack frame implementation should check for pointers and handle them specially, perhaps by falling back to explicit environment passing.[6] The ability to fall back in this way would in any case be useful as an optimization when the compiler can determine that the stack frame is large while the actual environment required is small.

Conclusions

We can't claim to be completely satisfied with the semantics we have defined for `eval`. Given the nature of C, however, we feel that no completely satisfactory solution can exist. `eval` is just too different from the other constructs in the language. Ultimately, only experience with actual programs will determine how useful the different possible interpretations are, and up to this point that experience is limited.

[6]Depending on the hardware, it may be possible to copy the data onto the stack, extending the stack frame, and change the pointer transferred so that it points to the new copy. This limits the special-case code necessary.

Summing Up

In this series, we've examined the Linda programming model and some of the alternatives available for embedding it into C. We've also had a fairly detailed look at one implementation.

Linda has proved to be a useful tool in a variety of problem domains, and is beginning to move from the laboratory into the work of practical computing. There are many issues we have not been able to explore, ranging from experience to programming techniques and beyond. The interested reader may wish to examine the papers we've referenced along the way.

Availability

Our implementation of LINDA-C for VMS VAXes has been made available to several laboratories on an experimental basis. We are developing a commercial version for initial release early this year. Later versions will support different hardware, as well as judicious additions to the language as experience dictates.

Jerrold Leichter recently received his doctorate from the Yale University Department of Computer Science. His dissertation includes an implementation of Linda for shared-memory and networked VAXes. 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.*

∞

38. Complex and VLAs Revisited

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

Abstract

In previous articles I have written about numerical deficiencies in C, and have proposed solutions for some of these deficiencies. Two of the proposals were to add *complex* and *variable length array* types. In light of comments I have since received, I wish to revisit these topics. I'll discuss changes to the original design along with some alternate viewpoints of those who communicated their opinions to me.

Introduction

In the March 1989 issue of *The Journal*, I wrote an article, *C Language and Numerical Programming*, that contained a discussion of some deficiencies in traditional C and ANSI C in supporting the requirements of numerical and scientific programmers. Subsequent articles contained proposals for two of the deficiencies mentioned: complex arithmetic and Variable Length Arrays (VLA). These proposals were based on plans by Cray Research to extend their Standard C implementation in these directions. Since these articles were written the topics have been debated by numerous readers, by the Numerical C Extensions Group (NCEG), and the actual implementors. This paper describes some of the differences between the original design and the actual implementation. I will also mention other proposals that have been made.

Complex Arithmetic

The original article titled *Adding Complex Arithmetic to C* appeared in the June 1989 issue of *The Journal*. The issues to be reviewed here are: the new keyword, the new arithmetic type, the new operator, and the *usual arithmetic conversions*.

Complex Types

The original article proposed a new header file `<complex.h>` which contained a `typedef` something like the following:

```
        typedef _Complex complex
```

However, in the actual implementation the `typedef` was replaced with:

```
        #define complex _Complex
```

The primary motivation for this change was due to two concerns raised at NCEG meetings. The first concern was that three different complex types should be accommodated: `float complex`, `double complex`, and `long double complex`.

The second concern was that a declaration containing the keyword `complex` should always be accompanied by either `float`, `double`, or `long double`, because then there is less chance for ambiguity about the underlying floating-point type of the real and imaginary parts. Making `complex` a macro name instead of a `typedef` name provides the needed flexibility for adding additional type-specifiers in a declaration.

```
        #include <complex.h>

        complex cx1;            /* error */
        float complex cx2;
        double complex cx3;
        long double complex cx4;
```

The initial Cray Research implementation only supports the types `float complex` and `double complex`.

Complex Constants and the Complex Operator

A new constant was proposed in the original article. It had the form:

```
        2.1i
```

which expands upon the existing suffix notation for constants to create a new complex constant. This complex constant specifies that the imaginary part is 2.1 and the real part is zero (0). Since the real part is always zero in a complex constant this is viewed by some as a deficiency. However, in my opinion it is the very nature of the mathematics involved in describing the complex plane that complex numbers are mapped onto. Since it takes two numeric values to represent one complex number, I believe that an expression such as:

```
        2.3 + 3.4i
```

is a natural way of representing a complex number.

A new operator was proposed to create a complex number out of two floating-point values. The following is an excerpt from the original article:

> The complex operator creates a complex number out of two floating-point values. Several different approaches are possible:
>
> ```
> cmplx(real, imag)       /* new keyword */
> <real, imag>            /* grouping */
> real %% imag            /* infix */
> ```
>
> New keywords are, again, controversial because of the potential to break existing code. Both the grouping and infix approach can be made to work.

However, I received a comment from Walter Bright at Zortech that a new complex operator was not necessary because the following:

```
real + imag * 1.0i
```

converts the two floating-point numbers, `real` and `imag` into a complex number. This eliminates the need for a new keyword or a new operator. To make this idea as easy as possible to use, an extension to the complex constants makes `1i` equivalent to `1.0i`. Thank you Walter.

Usual Arithmetic Conversions

The original article proposed adding the following:

> If either operand has type `complex`, then the other operand is converted to `complex`.

to the *usual arithmetic conversions.* However, several people commented that this simple rule allows precision to be lost for expressions like the following:

```
#include <complex.h>

long double x;
float complex cx;

cx + x          /* loses precision */
```

Losing precision is not in the best interest of numerical programmers. The usual arithmetic conversions need to be modified to preserve both precision and value when complex and floating types are converted. Therefore, the following is proposed as a replacement for §3.2.1.5 **Usual arithmetic conversions**.

3.2.1.5 Usual arithmetic conversions

Many binary operators that have operands of arithmetic types cause implicit conversions of one or both operands. The purpose of the conversions is to yield a common type for the two operands, which is also the type of the result. These implicit conversions of the operands are called the *usual arithmetic conversions*.

The conversions shall preserve the original magnitude and precision of both operands, except that precision may be lost when an integral type is converted to a floating type. This will occur if the magnitude of the integer is too great for the mantissa of the floating type to represent it exactly.

The result type is determined by three type attributes of the operands, called the *dimension*, the *format*, and the *length*. The dimension attribute specifies whether the values of the type can be represented on a one-dimensional line, i.e., real numbers, or on a two-dimensional plane, i.e., complex numbers. The format attribute specifies whether the values of the type are represented with an exponent part, i.e., floating numbers, or without an exponent part, i.e., integral numbers. The length attribute specifies how many bits are used to represent the absolute values of the type. The values for each of these attributes are ranked, from highest to lowest, as shown below. For example, complex ranks higher than real for the dimension attribute.

Dimension	`complex`, real
Format	Floating, Integral
floating length	`long double`, `double`, `float`
integral length	`unsigned long`, `signed long`, `unsigned int`, `signed int`

The rules for the usual arithmetic conversions are:

1. The dimension of the result type is that of the higher ranking dimension of the operands.
2. The format of the result type is that of the higher ranking format of the operands.
3. If the format of the result type is floating then the length of the result type is that of the higher ranking floating length of the operands; else the format of the result type is integral, and the integral promotions are performed on both operands. The length of the result type is that of the higher ranking integral length of the promoted operands, with one exception. The exception is that if one operand has type `signed long` and the other has type `unsigned int` and if a `signed long` cannot represent all the values of an `unsigned int`, the length of the result is `unsigned long`.

4. After the result type is determined and before the binary operation is performed, any operand that does not already have the same type as the result type is promoted to the result type.

As long as the result of the binary operation is mathematically equivalent to the result as if the promotions had been done, an implementation is not required to actually promote the operands.

Mathematical Equivalency

The notion of mathematical equivalency is applied to the *as if* rule to permit optimizations when one operand of a binary operator has a complex type and the other operand has a floating type. Consider the following example:

```
double x;
double complex cx;

x + cx;         /* convert x to double complex? */
```

From an optimization point of view it is more efficient to add only the real part of `cx` with `x`. However, implementations supporting IEEE arithmetic have both a positive and negative zero to worry about. If `x` is converted to `double complex` then the imaginary part becomes zero. The question that arises is, "Which zero?" Since adding positive zero to certain operands yields different results than adding negative zero, there is a concern.

```
-0 + +0 == +0
-0 + -0 == -0
```

This example shows that the age old axiom that `x + 0 == x` for every `x` is no longer true from an optimization point of view. Certainly there is no basis in mathematics for signed zeros that can be used for guidance. This is another reason for explicitly stating that in this case the *as if* rule applies to the mathematical concept of zero and the conversion from real to complex does not dictate which kind of zero is used.

Variable Length Arrays

Several sections of the *Variable Length Arrays* article that appeared in the December 1989 issue of *The Journal* were controversial and generated interesting discussions. Since this topic will receive additional time at the March 1990 NCEG meeting in New York City, there will certainly be more to report in the future.

One controversy surrounds the independent order of parameter names in function definitions containing a prototype. This independent ordering allows VLA function parameters in prototypes to be declared in any order. The following is an excerpt from that article:

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.

The controversy surrounds the burden placed on the implementation, and how to specify such a feature in a future standard.

The first implementation burden is that the compiler must now be able to parse a VLA declaration without knowing the types of variables found in between the `[]` tokens that specify the length of the array. The second implementation burden is that the compiler must now remember the token sequence that constitutes the entire declaration of the VLA parameter and parse it again when the `)` token that terminates the function prototype is detected. I certainly agree that C compilers have never in the past had to worry about *typeless* parsing nor rescanning an arbitrary number of tokens.

The first problem was solved in the Cray Research implementation by introducing the concept of typeless parsing. This technique permits the parsing of a VLA declaration such as:

```
void f(double a[xyz.m1 & n + *p >> count])
```

Since all identifiers in between the `[]` tokens (`xyz`, `m1`, `n`, `p`, `count`) might appear in subsequent parameter declarations, there is no way to detect that the types of operands are incompatible with their operators. Therefore, a *universal* type is assigned to every operand and operator found. The universal type is compatible with every operator. This permits syntax errors such as:

```
void f(double a[n +])
```

to be detected but semantic checking is delayed.

The second problem can be solved in several ways but requires a rescan of an arbitrary number of tokens. This can be accomplished by scanning just the tokens (Cray Research approach), or by scanning the original source characters again. Neither is trivial but neither is insurmountable. The second pass over the tokens exposes any semantic problems with the types of operands and operators. Since the names of all parameters in the prototype are now known, they can be used to resolve the VLA expression. Since VLA parameters are *incomplete types* until the `)` token that terminates the function prototype is parsed, VLA parameters *cannot* be used in certain contexts. This avoids ambiguities in pathological cases such as:

```
void f(double a[n][sizeof(b[0])],
       double b[n][sizeof(a[0])])
```

This is consistent with the current requirement that the operand of `sizeof` cannot have an incomplete type.

One reader commented that this additional compiler machinery is unnecessary because the old style function definition already permits parameter names to be specified in an arbitrary order. However, the ANSI standard specifies, "... the use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature." Perhaps a future committee will decide that the old style function definition still provides a useful service after all.

Dennis Ritchie favors an entirely different approach. His approach involves the introduction of a descriptor that contains all of the information about the array bounds. For example:

```
void f(double desc[?][?])        /* tentative syntax */
```

declares a parameter `desc` that is a descriptor with three pieces of information: an address, length of first dimension, and length of second dimension. Dennis is working on a full specification of this approach and plans to present this idea at the March meeting of the NCEG.

The `sizeof` Operator

When the `sizeof` operator has a VLA operand, the expression is no longer a constant expression. This is a fundamental change to the language. As someone who teaches C seminars, Rex Jaeschke has urged caution in this area. Teaching that `sizeof` generates a compile-time integer constant expression except when it doesn't, will, likely add considerable confusion. He suggests that since you can always compute a VLA's size from the dimension expressions used in its definition, perhaps `sizeof(VLA)` should be disallowed on the first pass. It can always be added later on.

I feel that the semantics of `sizeof` is not that severely affected because the size of a VLA is still invariant as long as it is visible. This preserves the current behavior with respect to `malloc`, which is one of the primary uses of `sizeof`. Our implementation supports the use of `sizeof` with VLAs as operands.

Variable Length Objects

A Variable Length Object (VLO) is a structure or union that contains a VLA member. The original article exposed a major weakness of a VLO type: it cannot be passed to a function. For that matter, a pointer to a VLO cannot be passed either. For these reasons, several people expressed reservations about implementing this feature. As a result, we did not add it. We may, however, if an acceptable solution is found and the merits of this feature are significant.

Conclusions

The position taken in my original article was motivated by the needs of programmers wanting to write numerical and scientific programs in C and not necessarily the needs of compiler writers. However, C compilers exist on many small computers, so features that might unduly burden these implementations should be scrutinized.

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.

∞

39. Interfacing Microsoft C and FORTRAN

Bob Allison
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052-6399

Abstract

Microsoft has incorporated a variety of attributes, keywords, and compiler invocation switches to aid in developing applications with components written in multiple languages. These conventions solve most, but not all, interfacing requirements between languages such as C and FORTRAN. This article describes these conventions and limitations, as well as future implications for possible extensions to C and FORTRAN.

Introduction

Earlier in this century, each European country had its own distinctly sized railroad lines. This required people and goods to be transferred to a new train at each border. Not so coincidentally, it also made military attack by rail nearly impossible. Modern times now require the rails to be of uniform size and separation to allow for efficient trade.

Procedure calling conventions within programming languages (not so coincidentally, I sometimes think) provide similar barriers to transferring flow of control and data between program components written in different languages. Unfortunately, we have not yet gotten to the point where a single procedure calling and argument passing mechanism is accepted for all programming languages. Nevertheless, plenty of applications wish to utilize existing libraries or components that have been written in another language, so various implementations have come up with ways to bridge the gap.

At Microsoft, we decided to solve the problem by introducing keywords and type attributes that describe most of the possible procedure calling and argument passing conventions. This has been a fairly successful strategy, with its only shortcomings in areas where the languages do not have similar data types. For the purposes of this paper, I will restrict my description to C and FORTRAN, although Pascal and BASIC are also supported at an equal level.

In this paper, I am using the term *calling convention* as a global term covering all the areas of procedure naming requirement, procedure calling method, argument passing order, argument passing method, and value-returning method.

Defining the Problem

Before getting to the solution, let's try to fully describe the problem. Form tends to follow function and C and FORTRAN specify the function of procedure names and argument lists explicitly. Most implementations of a language have settled on many of the same conventions, and Microsoft is no exception. Fortunately, the Microsoft implementation uses a stack-based calling convention for FORTRAN, so interfacing is simpler.

In C, the external name of a function is an underscore, followed by the name the user-specified, case sensitive, truncated to 31 characters. The user-specified name must consist of an alphabetic character, followed by alphabetic characters, numbers, or underscores. In FORTRAN, procedure names are all upper case and limited to six characters in length (although in the latest release this was increased to thirty-one characters). The first character must be an alphabetic character or dollar sign, followed by alphabetic characters, numbers, underscores, or dollar signs.

In C, arguments are passed by value and placed on the stack in reverse order from their specification. In FORTRAN, arguments are passed by reference and placed on the stack in the order declared. FORTRAN character strings are passed by address, with the length of the string placed in a global vector.

In both C and FORTRAN, a function returns its value in register(s) if the value is four bytes or less in size and not a floating point value. In C, floating point values are returned in a global variable, and objects larger than four bytes in size are allocated a location, whose address is returned in one or two registers. In FORTRAN, floating point values and values whose size is larger than four bytes are stored into a location allocated by the calling routine whose address is passed as a hidden argument. Character values have their length returned in the global vector for character arguments. In C, the calling routine adjusts the stack frame. In FORTRAN the called routine adjusts the stack frame.

New Keywords and Type Attributes

To provide a generalized calling convention that allows mixed-language programs in C and FORTRAN, with calls in each direction, one could try to whittle down the defined interface to areas of commonality. Unfortunately, there aren't any in the area of procedure naming convention and procedure calling convention. There are only a few areas in terms of argument passing and they aren't practical (you can't require C programmers to reverse the arguments and always pass by reference). So, Microsoft implemented the reasonable parts of both calling conventions in both languages and enabled the programmer to switch between the calling conventions via keywords, type attributes, and compiler invocation switches.

In C, Microsoft supports the `fortran` keyword in `extern` statements. This indicates that the prototype represents a FORTRAN function and should be called using the FORTRAN procedure naming, procedure calling, argument ordering, and value returning conventions. The code must explicitly pass the address of the arguments, if desired. The function name is not implicitly truncated by the compiler (except to 31 characters). An example of a declaration might be:

```
double fortran sum(double *vector, int len);
```

In FORTRAN, the closest similar functionality is to declare a name to be an external function via the `EXTERNAL` statement, and use the type attribute extensions (`[C]`, for example) to declare it as a procedure to be called using the C calling convention. For example:

```
DOUBLE PRECISION SUM[C]
EXTERNAL SUM
```

This indicates that the procedure is to be called using the C procedure naming (well, the name will be `_sum` since the attribute is defined to make everything lower-case as a reasonable approximation of typical C coding style), procedure calling, argument ordering, argument passing (by value), and value returning conventions. However, this is a limited method, since the name might not be all in lower case, and not all of the arguments may be intended to be passed by value.

To support a more generalized calling scheme, more keywords and type attributes were added. In Microsoft FORTRAN, there is an equivalent of the C function prototype statement, called the `INTERFACE TO` block. It allows the programmer to define the prototype of a procedure via a stub-like mechanism. For instance, in C, a prototype might be declared as:

```
int foo(double d, int i);
```

In Microsoft FORTRAN, a similar function would be prototyped as:

```
INTERFACE TO FUNCTION FOO(D,I)
INTEGER FOO
DOUBLE PRECISION D
INTEGER I
END
```

In Microsoft FORTRAN, these can only occur outside of other procedures. This allows for detailed description of the procedure name and each of the arguments via type attributes. For instance, in the `EXTERNAL` statement above, if the function had in fact been named `_Sum`, and it had two arguments, one passed by value and another by reference, then the `INTERFACE TO` block could be specified as:

```
INTERFACE TO FUNCTION SUM(VECTOR, LEN)
DOUBLE PRECISION SUM[C, VALUE:'_Sum']
DOUBLE PRECISION VECTOR[REFERENCE]
INTEGER LEN[VALUE]
END
```

In Microsoft FORTRAN, the `VALUE` type attribute can be used on external identifiers to precisely specify the desired external spelling of the identifier. The C attribute is still desirable to specify the other attributes of the C calling convention. The `REFERENCE` attribute specifies that calls to the function should pass this argument by reference. The `VALUE` attribute is not required in this example, due to the existence of the C attribute on the function, but is provided to introduce it. There is also a `VARYING` function type attribute which can be used in conjunction with the C type attribute and specifies that a varying number of arguments are allowed when calling the function.

Of course, keywords and type attributes require a lot of tweaking of the code by hand. Most cases are pretty straightforward and could be solved by using some form of switch during the invocation of the compiler. The C compiler supports the switch `/Gc` to indicate that the FORTRAN calling convention should be observed. In both compilers the significant number of characters on external names can be set with the `/H` switch. Note that when compiling a C module with `/Gc`, the default calling method for all functions in that module is that for FORTRAN. If one of the functions defined or declared inside that module really needs to use the C conventions, its declarations must include the `cdecl` keyword to override the default.

Some Problem Areas

Using the various keywords, type attributes, or compiler invocation switches allows most arguments to be passed across languages, assuming that a corresponding type exists in both languages. Unfortunately, FORTRAN's `COMPLEX` and `CHARACTER` data types do not have a corresponding type in C. The `CHARACTER` data type can be worked around by cautiously passing it to C as a character array of known length, but issues like null-terminated strings must be watched carefully. Similarly, `COMPLEX` can be worked around by using a structure in C, but the solution is not entirely satisfactory. Microsoft FORTRAN supports structures in its latest release, which relieves the reverse problem of user-defined types in C. Also, multiply dimensioned arrays in FORTRAN must be declared as pointers to the scalar object in C, not pointer-to-pointer-etc., which is moderately inconvenient. At the moment, there are no satisfactory solutions to these problems on the horizon.

However, there are two significant developments expected in the future which could substantially change current approaches to inter-language calling between C and FORTRAN. First, is the Numerical C Extensions Group (NCEG) work currently in progress. One area which seems to have a very good chance of appearing in the final proposal from NCEG is a complex data type for C. Also, some interest has been displayed in making arrays "first-class objects," but those efforts are less likely to result in a final proposal. Second, Fortran 8X is winding its way through the standards process (it is finishing its second public review), and once implementations appear it will be extremely difficult to pass arrays between C and FORTRAN, since the new array type in Fortran 8X is strictly oriented around a descriptor whose internal representation will obviously be implementation-specific. There is a pointer type in Fortran 8X as well, but it is also oriented around a descriptor (although the descriptor may not be required for pointers to scalar objects). However, none of these areas is a serious issue for at least two years.

As it happens, FORTRAN `COMMON` blocks present no problem: you just declare an external structure in C that has the same name and mapping as the `COMMON` block.

Conclusion

At Microsoft, we solved the problem of inter-language calling conventions by implementing as much as possible of each calling convention in each language and then providing various mechanisms for activating the various conventions. This limits the problem to those areas where there isn't any decent congruence between the languages, and the fundamental problem of requiring human intervention. Overall, though, it has been a very effective strategy and people have worked successfully within the boundaries of the solution. Unfortunately, future definitions of C and FORTRAN will probably both simplify and complicate inter-language calling conventions.

Bob Allison works at Microsoft as Development Manager, FORTRAN. He is also their representative to ANSI X3J3 (the Fortran 8X standards committee) and to the Numerical C Extensions Group. Bob may be contacted at (206) 882-8080 or uunet!microsoft!bobal.

∞

40. Pragmania

Rex Jaeschke

Intel's iC-86/286 C Compiler Controls

The information in this section is reprinted with permission of Intel Corporation. This material is extracted from their *iC-86/286 Compiler User's Guide*, © Intel, 1989.

The exact operation of the compiler may be specified by compiler controls which specify such things as the model of segmentation used by the compiler, the type and format of listings produced by the compiler, the destination of the object file, composition of the object module, etc. Compiler controls may be specified as part of the command invoking the compiler or within `#pragma` preprocessing directives.

There are two types of compiler controls: primary and general. Primary controls may be specified in either the command line or in a `#pragma` preprocessing directive. However, the only thing which may precede primary `#pragma` directives in a source file are other preprocessing directives and white space, including comments. Primary controls may not be changed within a module. General controls may be specified either in the command line or via a `#pragma` preprocessing directive. A `#pragma` directive for a general control may be located anywhere within a source module and may be changed freely within a source module.

A primary compiler control specified within the command line has precedence over the same control specified by a `#pragma` preprocessing directive. If the compiler finds a primary control within a pragma that has been specified on the command line a warning is issued.

A primary control duplicated on the command line causes a fatal error and results in termination of the compilation.

The following apply to compiler controls:

- Most compiler controls have a two-character abbreviation which may be substituted for the full control name.
- Most compiler controls have a default that reflects the most frequently used value. Therefore, although there are many controls, in most cases few are actually needed to compile a module.
- Arguments are required by some controls.

- Every control name itself in all forms is case-insensitive. Control arguments may be case-sensitive depending on the host environment.

The following table summarizes the compiler controls. As shown, some controls have an optional *no* prefix.

Intel's iC-86/286 Compiler Controls

Control Name	*Abbreviation*	*Control Type*	*Affected File*
no`align`	*no*`al`	general	object
no`code`	*no*`co`	general	listing
`compact`	`cp`	primary	object
no`cond`	*no*`cd`	general	listing
no`debug`	*no*`db`	primary	object
`diagnostic`	`dn`	primary	listing
no`extend`	*no*`ex`	general	–
`fixedparams`	`fp`	general	object
`interrupt`	`in`	general	object
`large`	`la`	primary	object
no`list`	*no*`li`	general	listing
no`listexpand`	*no*`le`	general	listing
no`listinclude`	*no*`lc`	general	listing
`medium`	`md`	primary	object
`mod86`	–	primary	object
`mod186`	–	primary	object
no`mod287`	–	primary	object
`modulename`	`mn`	primary	object
no`object`	*no*`oj`	primary	object
`optimize`	`ot`	primary	object
`pagelength`	`pl`	primary	listing
`pagewidth`	`pw`	primary	listing
no`print`	*no*`pr`	primary	listing
`ram`	–	primary	object
`rom`	–	primary	object
no`searchinclude`	*no*`si`	general	–
no`signedchar`	*no*`sc`	primary	object
`small`	`sm`	primary	object
`subsys`	–	primary	object
no`symbols`	*no*`sb`	primary	listing
`tabwidth`	`tw`	primary	listing
`title`	`tt`	primary	listing
no`type`	*no*`ty`	primary	object
`varparams`	`vp`	general	object
no`xref`	*no*`xr`	primary	listing

A brief description of each control follows. Unless otherwise shown, the control takes no arguments.

no`align` – Aligns structures on a specified boundary of 1, 2, or 4 bytes.

> `#pragma align` [`(`*structure_tag*[`=`*size*][,...]`)`]
> `#pragma noalign` [`(`*structure_tag*[,...]`)`]

no`code` – Generates or suppresses pseudo-assembly language code in listing file.

`compact` – Specifies the compact segmentation memory model.

no`cond` – Includes or suppresses uncompiled conditional code in the listing file.

no`debug` – Includes or suppresses debug information in the object file.

`diagnostic` – Specifies the level of diagnostic messages.

> `#pragma diagnostic (`*level*`)`

no`extend` – Recognizes extended keywords from previous versions of C compilers from Intel.

`fixedparams` – Specifies fixed parameter list calling convention.

> `#pragma fixedparams` [`(`*function* [,...]`)`]

`interrupt` – Specifies the function is an interrupt handler.

> `#pragma interrupt` [`(`*function*[`=`*n*][,...]`)`]

In the iC-86 compiler, the argument *n* can be used to associate the handler with an interrupt number from 0–255.

`large` – Specifies the large segmentation memory model.

no`list` – Includes or suppresses source text in the listing file.

no`listexpand` – Includes or suppresses macro expansion in the listing file.

no`listinclude` – Includes or suppresses header contents in the listing file.

`medium` – Specifies the medium segmentation memory model.

`mod86` – Specifies the 8086/8088 instruction set.

`mod186` – Specifies the 80186/80188 instruction set.

no`mod287` – Generates code for an 8087 or 80287 numerics coprocessor. Includes or suppresses FWAIT instructions.

`modulename` – Names the object module.

`#pragma modulename (`*name*`)`

no`object` – Generates and names, or suppresses object file.

`#pragma object [(`*filename*`)]`

`optimize` – Specifies the level (0, 1, 2, or 3) of optimization.

`#pragma optimize (`*level*`)`

`pagelength` – Specifies the lines per page in the listing file.

`#pragma pagelength (`*lines*`)`

`pagewidth` – Specifies the line length in the listing file.

`#pragma pagewidth (`*chars*`)`

no`print` – Generates or suppresses the listing file.

`#pragma print [(`*filename*`)]`

`ram/rom` – Specifies the placement of constants in the object module.

no`searchinclude` – Specifies the search paths for headers.

`#pragma searchinclude (`*pathprefix* `[,...])`

no`signedchar` – sign-extends or zero-extends `char`s when promoted.

`small` – Specifies the small segmentation memory model.

`subsys` – Reads a subsystem specification.

no`symbols` – Includes or suppresses the identifier list in the listing file.

`tabwidth` – Specifies the characters per tab stop in the listing file.

`#pragma tabwidth (`*width*`)`

`title` – Specifies the listing file page title.

    #pragma title ("*string*")

no`type` – Includes or suppresses debug information in the object file.

`varparams` – Specifies variable parameter list calling convention.

    #pragma varparams [(*function* [,...])]

no`xref` – Includes or suppresses symbol table cross-reference in the listing file.

Apple's MPW 3.0 Compiler

The following material is extracted from Apple's manual *Macintosh Programmer's Workshop 3.0 C Reference*, © Apple, 1988.

Segmentation

Generated code can be placed in named code segments instead of the default segment `Main`.

    #pragma segment *segment-name*

Since segment names may contain spaces, leading and trailing spaces around the segment name are significant except for the single space following the token `segment`.

Precompiled Headers

Two pragmas are provided to save and restore precompiled versions of headers. A precompiled header allows for more efficient processing and is stored in a disk file in a compressed format.

    #pragma dump "*filename*"
    #pragma load "*filename*"

Unused Identifiers

Normally, unused identifiers are flagged by the compiler. Such warnings can be suppressed using the following directive.

    #pragma unused (*identifier* [, *identifier* ,...])

Since only local variables and formal parameters are flagged, this directive must appear inside a function definition.

∞

41. Cray C: C in a 64-bit World

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

Abstract

The C implementation on Cray Research supercomputers exposes many of the erroneous portability assumptions that exist in many C applications. Most of these non-portable assumptions reflect the behavior of many existing implementations. A description of some aspects of the Cray Research C implementation is given along with a discussion of how they affect some non-portable applications.

Introduction

When someone plans to port a C application to a Cray supercomputer, his initial attitude typically is, "We have ported this application to several different computers and all of the porting problems have been solved. Our standard porting package allows us to get it working in *x* weeks." However, I always fear the worst because a truly portable application is a rare find. The same porting problems are encountered over and over again. These problems are never easy to find and yet I know they are there. When a porting problem is finally tracked down and identified, there is a good chance it is one of the non-portable constructs that I will describe in this article.

C-WORLD

The inhabitants of C-WORLD are called C programmers. Many C programmers live on *Main Street* in C-WORLD which consists of a byte-addressable machine with two's-complement arithmetic and a linear address space. It also has 8-bit `char`s, 16-bit `short`s, and 32-bit `long`s. All pointers have exactly the same representation, and characters are represented in ASCII.

C programmers have heard that some implementations only support six character external names, but these implementations are considered to be deficient. Many choose to ignore the REF/DEF model for external names that the ANSI standard imposes on portability. When a C programmer leaves *Main Street* his environment often changes, and drastically.

The Cray Research Implementation

Before describing the most common non-portable constructs encountered while porting codes to the Cray Research implementation of C, I will explain something about that implementation.

Currently, all of Cray Research's computers are 64-bit word-addressable machines. They have three different sizes of registers: Address (A), Scalar (S), and Vector (V) registers. The S registers are 64 bits wide, and the A registers are 24 bits wide on older machines and 32 bits wide on newer machines. The V registers have 64 elements, each 64 bits wide. However, V registers are not really relevant in this article. The following is the data mapping for various types:

Cray C Type Representations

Type	*Precision*	*Register*
`char`	8 bits	S
`short`	24/32 bits	A
`int`	64 bits	S
`long`	64 bits	S
`float`	64 bits	S
`double`	64 bits	S
`long double`	128 bits	2 S
byte pointers	64 bits	S
word pointers	24/32 bits	A
aggregates	N/A	V

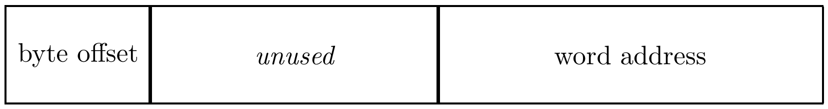
One important aspect of the `short` type and word pointers is that even though there are only 24/32 bits of precision in A registers, they still occupy a full 64-bit word of memory. Therefore, the following expression evaluates to true with the Cray C implementation:

```
sizeof(short) == sizeof(long) && SHRT_MAX < LONG_MAX
```

This means that even though the `short` type and word pointers are mapped onto the A registers they are not packed in memory. For these types, all traffic to and from memory is through an A register, which leaves the bits in the upper half of the word unused. The non-`short` integral types (including `char`), floating types, and byte pointers travel to and from memory through S registers, thus their upper bits are not affected by register length.

Byte Pointers

The type *array of* `char` is packed in memory with eight bytes per 64-bit word. This means there are no unused bits. Since the `char` type is packed, a byte pointer must represent the byte in the word the `char` is stored in. The information contained in a byte pointer represents both the word address and byte offset. This uniquely identifies all possible byte addresses. A 64-bit S register is used to hold the values of byte pointers. The format of the byte pointer has the byte offset (B/O) stored in the left most (high) 3 bits and the word address stored in the right most 24/32 bits, with unused bits in the middle. The following figure depicts how this would look in an S register.



Pointer arithmetic on byte pointers is done by performing the following instructions:

- Left circular shift the B/O around to the lower part of the word.
- Perform the arithmetic.
- Right circular shift the B/O back around to the left-most three bits.

Using this representation for a byte pointer has several advantages. First, both the word pointers and byte pointers have the same representation when they reside in memory. Second, since the word address is right justified, a single instruction can transfer it to an A register. Finally, there is no change in bit pattern when converting the value of a word pointer to a byte pointer.

These advantages outweigh the disadvantages encountered when byte pointers are converted to an `int` and used in non-portable ways. Consider the following program:

```
#include <stdio.h>

main(){
        char *cp = "abcdefghijklmnopqrstuvwxyz";

        int i = (int) cp;
        printf("<%s>", (char *)(i + 2));
}
```

When this program is compiled with a Cray C compiler, the generated output is `<qrstuvwxyz>`. This shows that converting a byte pointer to an `int` does not change the bit pattern. The expression `(char *)(i + 2)` causes the word address to be incremented by two. This is a common portability problem. On many byte-addressable machines this causes the byte address to be incremented by two.

Another common porting problem occurs when an application attempts to compute the byte offset of a member of a structure. Consider the following example:

```
#include <stdio.h>

struct tag {
        int m1;
        float m2;
        char m3;
};

int offset = (int) &((struct tag *)0)->m2;

main() {
        printf("<%d>", offset);
}
```

When this program is compiled with a Cray C compiler the generated output is `<1>`. The value that is stored in `offset` is a word offset, not a byte offset. Again, on many byte-addressable machines the value would have been a byte offset. The ANSI standard specifies that an implementation-defined macro named `offsetof` must exist in the header `<stddef.h>`. This allows the following:

```
int offset = offsetof(struct tag, m2);
```

to be used to find the byte offset of a structure member. Unfortunately, applications exist that require the first argument to a macro similar to `offsetof` to be a pointer to a structure. One example of this is the X-Windows package. X-Windows requires a macro similar to `offsetof` except that the first argument must be a pointer to a structure. Something like the following is used:

```
int offset = Xoffset(struct tag *, m2);
```

Unfortunately this macro cannot be defined in terms of the ANSI `offsetof` macro. Technically speaking, X-Windows is *not* portable.

Integral Types

The type `short` is 16 bits on most systems. However, it is *not* 16 bits on any Cray supercomputer. Unfortunately, there are applications that read two bytes from a device and store those two bytes into the elements of an *array of* `short`*s*. On a Cray, this inevitably ends up writing into the unused bits of the memory word used to store a `short`. There is a surprising amount of code that exploits this non-portable technique.

Similarly, the type `long` is 32 bits on most systems, and again this is not true for the Crays. One networking code changed a declaration from:

```
        char a[4];
```

to:

```
        long a;
```

because, *after all*, both types *are* 32 bits, right? Again, the assumption that certain types are a certain size makes the code non-portable.

Other Porting Issues

Some less common porting problems are due to the following assumptions:

- A character pointer has the same representation as a function pointer.
- A pointer can represent negative values.
- An `int` bit-field designates a signed value.
- Floating-point arithmetic with integral values is exact (For example, assuming that the expression `(int)(6.0/3.0) == 2` is true.)
- The `sizeof` operator yields a signed type.

These assumptions may be true on some machines but they are not portable.

There are many aspects of Cray's C implementation that make it easy to port certain codes that are technically speaking, non-portable. The following is a list of features that comply with common beliefs of the C programmers from C-WORLD:

- 8-bit `char` type.
- ASCII representation.
- Two's-complement representation.
- Linear address space.
- 255 character external names.
- External names reside in FORTRAN style `COMMON` blocks (no REF/DEF model is supported).

Summary

The following is a list of portability issues with the Cray C implementation raised by this article. Hopefully, they will help lead to more portable code being written.

- Byte pointers converted to an `int` do not necessarily represent a byte offset.
- Converting a pointer type to an `int` type to perform arithmetic is unwise.
- A `short` is not necessarily 16 bits.
- A `long` is not necessarily 32 bits.
- Pointers can be unsigned values.
- There can be unused bits in an object for certain types.

Conclusions

Many decisions must be made about how a particular C implementation is developed. Some of these decisions are influenced by the non-portable assumptions made by C programmers from C-WORLD. For example, if at all possible, an implementation should strive to provide external names that are longer than six characters even though the ANSI standard does not require it. Too many applications have been written with non-portable external names that are longer than six characters.

Therefore, my advice on this subject is, "Do what makes sense." It certainly is not profound advice, but it comes from years of experience with a C implementation on a 64-bit word-addressable supercomputer that is outside the mainstream of C-WORLD. A supercomputer is built to provide performance. If a decision must be made between performance and supporting non-portable coding practices, Cray Research will undoubtedly decide in favor of performance. Most portability issues involve solutions that please some but not others. The best solution to these portability issues is the one that is best for your market place.

And perhaps you can do me a favor. Since no one buys a Cray as their first machine, code that runs on Crays inevitably comes from other systems. As such, I urge you to mention these issues in the portability section of your manual set, not just for Cray's sake but for vendors of like architectures.

Tom MacDonald is Cray Research Inc's representative to X3J11. He can be reached at (612) 681-5818, tam@cray.com, or uunet!cray!tam.

∞

42. Miscellanea

compiled by **Rex Jaeschke**

`size_t`, `ptrdiff_t`, and Some Magic

Intel's Memory Models

Anyone with more than a little experience in developing software on Intel's 80x8x segmented architecture family will have learned all about *memory models* and will probably have come to hate them. (Just in case you don't recognize this chip family, it's the one IBM made famous by using it for its PCs in 1981.)

Simply stated, when you compile a program you must select one of a number of memory models for your code and data. Since the low end of the CPU family has only 16-bit registers, it's much more efficient to use 16-bit (near) pointers rather than 32-bit (far) pointers. (`int`s are always 2 bytes and `long`s are 4.) The four basic memory models then are:

Intel 80x8x Memory Models		
Model	*Code pointer*	*Data pointer*
Small	near	near
Compact	near	far
Medium	far	near
Large	far	far

Different compiler vendors use different names for these memory models. However, in this article Microsoft's names will be used. One restriction on the Large memory model is that no object can exceed 64K bytes. To have objects larger than this you must either use the `huge` keyword explicitly in the declaration or compile using a *huge* option (the latter having been used in the following examples).

Some Hard Choices

This CPU family is interesting because it is a 16-bit architecture *and* it can directly address more than 64K bytes of memory. (Most 16-bit machines require various disk and/or memory overlay techniques to shoehorn large programs into a 64K byte address space.)

This gives rise to the problem that `size_t` could be of type `unsigned long` since 16-bits isn't large enough to store the size of the largest possible object. The real problem is that 32-bit operations are not directly supported by the low-end CPUs. The choice then becomes one of the following:

1. Make `size_t` the (2 byte, very efficient) type `unsigned` and forbid objects larger than 64K bytes.
2. Make `size_t` the type `unsigned long` and pay the price of the extra data and code space and execution speed all the time *even* though very, very few objects are (typically) ever larger than 64K bytes.
3. Conditionally compile the `typedef` for `size_t` based on the memory model chosen at compile-time. Also, make `sizeof` behave differently for various models and have a corresponding set of matching library functions for each version.

Similarly, `ptrdiff_t` could be of type `long` since 16-bits isn't large enough to store the size of the largest possible pointer difference. Note though that while some standard library function prototypes use `size_t`, none uses `ptrdiff_t`.

One Vendor's Solution

Consider the following program, which was compiled using Microsoft C V5.1.

```
#include <stdio.h>
#include <stddef.h>

char c[100000];

main()
{
        unsigned long size1;
        unsigned size2;
        long diff1;
        int diff2;

        printf("     sizeof(unsigned): %lu\n",
                (unsigned long) sizeof(unsigned));
        printf("sizeof(unsigned long): %lu\n",
                (unsigned long) sizeof(unsigned long));
        printf("       sizeof(size_t): %lu\n",
                (unsigned long) sizeof(size_t));
        printf("     result of sizeof: %lu\n",
                (unsigned long) sizeof(sizeof(int)));

        printf("         sizeof(c) #1: %lu\n",
                (unsigned long) sizeof(c));
        printf("         sizeof(c) #2: %u\n\n",
                sizeof(c));
                size1 = sizeof(c);
                printf("                size1: %lu\n", size1);
                size2 = sizeof(c);
                printf("                size2: %u\n\n", size2);
                printf("          sizeof(int): %lu\n",
                        (unsigned long) sizeof(int));
                printf("         sizeof(long): %lu\n",
                        (unsigned long) sizeof(long));
                printf("    sizeof(ptrdiff_t): %lu\n",
                        (unsigned long) sizeof(ptrdiff_t));
                printf("    result of ptr sub: %lu\n",
                        (unsigned long)sizeof(&c[90000]-&c[0]));
                printf(" &c[90000] - &c[0] #1: %ld\n",
                        (long) (&c[90000] - &c[0]));
                printf(" &c[90000] - &c[0] #1: %d\n\n",
                        (&c[90000] - &c[0]));

                diff1 = &c[90000] - &c[0];
                printf("                diff1: %ld\n", diff1);
                diff2 = &c[90000] - &c[0];
                printf("                diff2: %d\n", diff2);
        }
```

The output produced is:

```
             sizeof(unsigned): 2
        sizeof(unsigned long): 4
               sizeof(size_t): 2
             result of sizeof: 2
                 sizeof(c) #1: 100000
                 sizeof(c) #2: 34464

                        size1: 100000
                        size2: 34464

                  sizeof(int): 2
                 sizeof(long): 4
            sizeof(ptrdiff_t): 2
            result of ptr sub: 2
         &c[90000] - &c[0] #1: 90000
         &c[90000] - &c[0] #1: 24464

                        diff1: 90000
                        diff2: 24464
```

As you can deduce from the output, Microsoft C types `size_t` as `unsigned` and `ptrdiff_t` as `int`. (They do this in all memory models.) However, the correct size of a large object can be found using `sizeof`, and a large pointer difference can also be correctly determined. How do they do it? Well, they pack 32 bits of information into 16 bits. What? You don't believe it?

The critical part of the code is as follows:

```
unsigned long size1;
long diff1;

(unsigned long) sizeof(c)
size1 = sizeof(c)

(long) (&c[90000] - &c[0])
diff1 = &c[90000] - &c[0]
```

Even though `sizeof` produces an `unsigned int` result, by casting that to `unsigned long` you magically get 32 bits of precision out of the 16-bit unsigned value that you started with. As the output also shows, the same occurs when a `sizeof` result is directly assigned to an `unsigned long` object.

Similarly, casting (or assigning) a huge pointer difference to `long` produces the correct answer.

Clearly, this is *not* intuitive to the programmer and it's certainly *not* ANSI-conforming. However, the approach does work, it's a reasonable trade-off given the host environment, and very, very few programmers will ever care since they rarely have objects larger than 64K bytes. About the only time you get into trouble is when you use `printf` with `%u` and `%d` to display `size_t` and `ptrdiff_t` expressions, respectively, as shown in the example.

Extensions

In this issue we debut a new section. It covers extensions to compilers, preprocessors, or run-time libraries. To have your product considered for coverage in this section, send a manual set to the editorial offices or submit a technical description of its interesting features by electronic mail or magnetic media. The extensions discussed will not necessarily by upwards-conforming to ANSI C.

Pixar's *Chap C*

Pixar builds a special purpose graphics processor called the *Chap*^{TM} Channel Processor. Their *Chap C Compiler* enables graphic algorithms to be developed for the Pixar Image Computer^{TM} using the C language.

The hardware has a 4-way Single Instruction Multiple Data (SIMD) architecture with tesselated access modes to the picture memory. The C compiler supports extensions for this hardware and its four ALUs.

The Chap's memory is addressable in 16-bit units. As a result, a byte, word, `char`, `int`, and all pointer types are synonymous. `long`, `float`, and `double` all use 32 bits. (Like `float`, `double` maps to IEEE's single-precision.) By selecting 16-bit `char`s, character pointers have the same representation as word pointers and a string literal is equivalent to an array of `int`s with a trailing null word.

The type modifiers `parallel` and `component` provide parallel access to the graphics-oriented data format used by the hardware. A *pixel* is a vector of four `int`s (representing the red, blue, green, and alpha elements) and all parallel data types are composed from pixels. Consider the following example:

```
void f(pixel, ppixel)
register parallel int pixel;
register parallel int *ppixel;
{
        ppixel->chan[0] = pixel.chan[3];
        ppixel->chan[1] = pixel.chan[2];
        ppixel->chan[2] = pixel.chan[1];
        ppixel->chan[3] = pixel.chan[0];
}
```

`pixel` is a vector of four `int`s each of which is accessible by the predefined name `chan[`*x*`]` where *x* is 0–3. While a pixel is not really a structure, structure notation is conveniently used to access its elements.

A *component* is a group of four pixels. It looks like an array of four parallel objects. Also, an array of parallel objects can be treated as a component simply by casting the address of the first array element to a component pointer. A pixel in a component is accessed using the following notation, where `comp` is a predefined name.

```
component int compon;
parallel int pixel;

pixel = compon.comp[2];
```

Other interesting aspects of the implementation include: Arrays, structures, parallel objects, and component objects are aligned on 4 word boundaries. `sizeof` returns 1 for `short`, `int`, and pointer types as well as for `char`. It returns 2 for `long`, `float`, and `double`.

Several interesting operators exist. According to the *Chap C Compiler Technical Summary*, Feb 1988:

> The state of parallel processing may be checked through the special functions `any()` and `all()`. These expressions return a scalar Boolean value based on the evaluation of the parallel Boolean value given as an argument. The `any()` function returns *TRUE* if the argument expression evaluates to true on any processor. The `all()` functions returns *TRUE* if the argument expression evaluates to true on all processors.

> The `force()` operator is used to modify the runflags within a block of code. It takes as a parameter a scalar expression whose low four bits will become the new runflag (which indicates the subset of the four parallel channels that are to be active). The effect of the force will persist until the end of the following statement.

Ed: Mike Paquette, a principal contributor to the Chap C Compiler contributed the following information.

One thing worth noting is that the parallel types are first order objects, *not* aggregates. That means a parallel type can participate in any operation that a scalar type can, with the result being promoted to a parallel type. This lets us speed up operations on small matrices significantly. For example, here is a vector combiner from a ray tracer:

```
typedef double Vec[3];

VecComb(A, a, B, b, c)
 double A, B ;
 Vec a, b, c ;
{
        c[0] = (A * a[0]) + (B * b[0]);
        c[1] = (A * a[1]) + (B * b[1]);
        c[2] = (A * a[2]) + (B * b[2]);
}
```

Using the parallel data types we can collapse the six multiplies and three adds into two multiplies and one add, executing on multiple channels of the hardware. Here is the same function, written using parallel doubles.

```
typedef parallel double Vec;

VecComb(A, a, B, b, c)
 double A, B ;
 Vec a, b, c ;
{
        c = (A * a) + (B * b);
}
```

For more information on their hardware and/or C compiler contact Pixar at 3240 Kerner Blvd., San Rafael, CA 94981, (415) 258-8100.

Free Software Foundation

The following information is taken from *Using and Porting GNU CC* © 1988, 1989 Free Software Foundation, Inc. As required by the the copying terms of the manual, the following text is included here.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU General Public License" is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled "GNU General Public License" and this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.

The Journal recognizes and respects the GNU Manifesto and the intent and effort put into all software and documentation produced by the Free Software Foundation. As such, this section is *explicitly not included* in the copyright claim for this issue. The text was taken verbatim from the GNU TEX sources. However, it has been translated into the LATEX typesetting package. Any error or change introduced during that conversion is unintentional.

GNU is a licensed (at no charge) software project involving an operating system, compilers, linker, debugger, and other language development tools. It is run by Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA and was instigated by Richard M. Stallman. Their E-mail address is: *uunet!tower@wheaties.ai.mit.edu*. The GNU project sources and documentation are available electronically from numerous public sites around the world, including *uunet*.

GNU Extensions to the C Language

GNU C provides several language features not found in ANSI Standard C. (The '-pedantic' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is always defined under GNU CC.

Statements and Declarations inside of Expressions

A compound statement in parentheses may appear inside an expression in GNU C. This allows you to declare variables within an expression. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
```

```
           else z = - y;
           z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo()`.

This feature is especially useful in making macro definitions “safe” (so that they evaluate each operand exactly once). For example, the “maximum” function is commonly defined as a macro in Standard C as follows:

```
        #define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either `a` or `b` twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let’s assume `int`), you can define the macro safely as follows:

```
        #define maxint(a,b) \
          ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don’t know the type of the operand, you can still do this, but you must use `typeof` or type naming.

Naming an Expression’s Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *name* as a type name for the type of *exp*:

```
        typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe “maximum” macro that operates on any arithmetic type:

```
        #define max(a,b) \
          ({typedef _ta = (a), _tb = (b);  \
            _ta _a = (a); _tb _b = (b);     \
            _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers. This will be a more reliable way to prevent such conflicts.

Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax for using this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`, with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions. The type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointer to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typedef` instead of `typedef`.

A `typeof`-construct can be used anywhere a `typedef` name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares `y` as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

  It is equivalent to the following traditional C declaration:

  ```
  char *y[4];
  ```

  To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

  ```
  #define pointer(T)  typeof(T *)
  #define array(T, N) typeof(T [N])
  ```

  Now the declaration can be rewritten this way:

  ```
  array (pointer (char), 4) y;
  ```

  Thus, 'array (pointer (`char`), 4)' is the type of arrays of 4 pointers to `char`.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at that time and deallocated when the brace-level is exited. For example:

```
FILE *concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len])
{
  ...
}
```

The length of an array is computed on entry to the brace-level where the array is declared and is remembered for the scope of the array in case you access it with `sizeof`.

Jumping or breaking out of the scope of the array name will also deallocate the storage. Jumping into the scope is not allowed. You will get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

Quality of Implementation

Borland's Turbo C V2

Ever since Borland released the first version of Turbo C I've been recommending it as a front-line quality assurance tool. Not only is the interactive development environment very easy to work with, the number of code quality checking options available is significant. This is not surprising considering that the original author of the compiler, Bob Jervis, also had a commercial version of the static analysis tool lint. (This program was called Pre-C and is no longer sold.) In fact, prior to selling his compiler and company to Borland, Bob's version (called Wizard C) actually had lint integrated in it pretty much as Turbo C has now.

To demonstrate Turbo C's quality checking I contrived the following test program:

```
/* 1*/  struct tag {
/* 2*/          int i;
/* 3*/          double d;
/* 4*/  };
/* 5*/
/* 6*/  f(int i, double d, long l)
/* 7*/  {
/* 8*/          char *pc;
/* 9*/          char c;
/*10*/          int *pi = &i;
/*11*/          int (*pf)(int, double, long);
/*12*/          struct tag st = {10, 1.2};
/*13*/
/*14*/          c = *pc;
/*15*/          i = l;
/*16*/          pf = &f;
/*17*/          *pi;
/*18*/          test(st, 123456);
/*19*/          goto label;
/*20*/
/*21*/          pi++;
/*22*/  label:  ;
/*23*/          if (i = 10)
/*24*/                  ;
/*25*/
/*26*/          if ((i = 10) != 0)      /* no warning */
/*27*/                  ;
/*28*/  }
/*29*/
/*30*/  g(int j)
/*31*/  {
/*32*/          if (j + sizeof(int) > -1)
/*33*/                  return;
/*34*/          else
/*35*/                  return (20);
/*36*/  }
```

The messages produced when this was compiled were:

```
line 14: Possible use of 'pc' before definition
line 15: Conversion may lose significant digits
line 16: Superfluous & with function or array
line 17: Code has no effect
line 18: Constant is long
line 18: Structure is passed by value
line 18: Call to function 'test' with no prototype
line 21: Unreachable code
line 23: Possibly incorrect assignment
line 28: 'pf' is assigned a value which is never used
line 28: 'c' is assigned a value which is never used
line 28: Parameter 'd' is never used
line 28: Function should return a value
line 32: Constant out of range in comparison
line 35: Both return and return of a value used
```

Other options available when configuring the compilation and linking environment include:

- Plain `char` signedness selection.
- Alignment on byte or word boundaries.
- Merging of duplicate strings.
- Testing for stack overflow at run-time.
- Identifier significance length from 1–32 characters.
- Case-sensitive external names.

By the way, more than a few implementers had told me over the years that a C compiler has no business doing all this checking—that's what lint is for. To avoid a repeat of this argument let me say that: There are many environments for which lint is not available at all. In those (outside UNIX systems) where it is, it comes from a vendor other than that of your compiler. As such, incompatibilities are bound to occur. Also, users prefer to deal with fewer vendors since that reduces the finger pointing when interfacing problems occur. Since anyone buying a C compiler can benefit greatly from a lint-like tool, **why not built lint into the compiler to begin with?**

Calling Undeclared Functions

When a function is called and no prototype is in scope, the behavior is well defined. No argument list count or type checking can be done and the function is assumed to return an `int` value. As such, ANSI C does *not* require any diagnostic. However, this is an occasion to seriously consider issuing an informational message.

By calling a function without having a prototype in scope, the following may occur:

- If the function's return type is *not* `int` and the return value is used, bad code may be generated. (For example, if `int` and `long` are different sizes.)
- If the number and/or types of the arguments is incorrect, this could lead to a non-trivial debugging effort.
- If the function has a variable argument list, the behavior when calling it is undefined.
- Type conversion of arguments to prototype declarations cannot occur. (Even with a declaration of `double sqrt(); sqrt(10)` will produce an interesting result.)
- Arguments with narrow types will always be widened.

I have seen a few implementations that warn of this situation and **I strongly encourage the rest of the implementers to do likewise.** I find that this single message can save programmers many hours of debugging in each project. The main cause of the message is the failure to include the correct header.

Calendar of Events

- April 9–11, **USENIX C++ Conference** – Location: San Francisco Marriott. Contact Jim Waldo at *waldo@apollo.com* for more information.
- June 18–19, **SC22/WG14 ISO C Meeting** – Location: British Standards Institute, London, England. The expected major work items will be a discussion on the UK's *Normative Addendum* proposal, Denmark's alternate trigraph proposal, and numerous wide-character function proposals from the Japanese group ITSCJ.
- July 9–13, **ANSI C++ X3J16 Meeting** – Location: Microsoft to host somewhere in the Seattle, Washington area. For more information, contact the convener Dmitry Lenkov at Hewlett-Packard, 19447 Pruneridge Avenue, MS 47LE, Cupertino, CA 95014, (408) 447-5279, or electronically at *dmitry%hpda@hplabs.hp.com.*
- August 13–17, **International Conference on Parallel Processing** – Location: Pheasant Run resort in St. Charles, Illinois. Call David A. Padua on (217) 333-4223 or *padua@a.cs.uiuc.edu* for more information. (David is the contact for software-related papers.) The deadline for submitting papers is Jan 10.

- September, **ANSI C X3J11 Meeting** – At press time the date had not been set but Lawrence Livermore National Labs and SSI have offered to host somewhere near Livermore California (an hour east of San Francisco.) 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*.
- September, **Numerical C Extensions Group (NCEG) Meeting** – The fourth meeting will be held to consider proposals by the various subgroups. It will follow the X3J11 ANSI C meeting being held at the same location earlier that week (see above entry) and will likely run for two full days. For more information about NCEG, contact the convener Rex Jaeschke at (703) 860-0091 or *uunet!aussie!rex*, or Tom MacDonald at (612) 681-5818 or *tam@cray.com*.
- September 19–21, **International Workshop on Attribute Grammars and their Applications** – Location: Paris, France. For information, contact: INRIA, Service des Relations Exterieures, Bureau des Colloques, B.P. 105, F-78153 LE CHESNAY Cedex, France. Telephone: [33] (1) 39.63.56.00; Telex: 697 033 F; FAX: [33] (1) 39.63.56.38; E-mail: *waga@minos.inria.fr*.
- 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.
- November 12–16, **ANSI C++ X3J16 Meeting** – Location: Hewlett-Packard to host in the San Francisco, California area.

News, Products and Services

- Since late in 1989 we've heard much of the upheaval in the Eastern European Bloc. Until you can relate to it personally, it's just another news item. What's this to do with *The Journal?* Well recently I received a letter from a Senior Software Engineer in Romania. The relevant extract follows:

  > Prior to, and during, the recent revolution in Romania we at the Research Institute for Computers suffered much. Much hardware was damaged or destroyed and we also lost considerable software and many books.
  >
  > **I would be very grateful if you could help me, by appealing to your readers, rebuild our collection of books and documentation on C, C++, and expert systems. Also software and even hardware if possible.**

I have put together a shipment of books and manuals that I intend to send. I encourage you to do the same. If you have any doubts about export restrictions on certain items (hardware, for example) consult your national Trade Commission or relevant government agency. Since I know they are using Borland's Turbo C it appears they at least have IBM-PC compatibles.

The address to send computer care packages is:

Doru Turturea
Research Institute for Computers
Strada Clucererului No 1, Sector 1
Bloc 40, Scarad
Bucuresti 71308
ROMANIA

I'm sure that, apart from doing a good deed for our colleagues abroad, someone in your marketing department would love the press coverage a generous donation could bring. Think about it. Wouldn't your company like to get a foot in the door of the newly opened Eastern European marketplace?

- **C lives [again] on the venerable PDP-11.** DEC has announced and is shipping PDP11C, their ANSI-conforming compiler for DEC PDP-11s. The compiler can be hosted on VAX/VMS, RSX, and RSTS, and supports these targets as well as POS and RT-11.

- **BCPL anyone?** Various implementations of BCPL, one of C's ancestors, are available from RimStar Technology, 1 Commodore's Court, Suite 503, Hull, MA 02045. (617) 925-2718. A DOS hosted kit runs $79.95, which includes the BCPL text by the language's designer, Martin Richards. Source to the compiler and library is available for $229.95. Code generators are available for VAX, MC680x0, IBM 370, and Data General machines. *[Ed: I have recently installed this product and plan to report on it and the language in a future issue.]*

- Available from ACE is the 1989 edition of "Benchmarking UNIX Systems," which includes the results of the execution of the ACE Benchmark Suite on more than 100 different major UNIX systems. The ACE Benchmark Suite integrates most of the established benchmark programs and provides versions of these in all of C, Pascal, Fortran77 and Modula-2.

  ACE has completed the validation and benchmarking phase of ACE EXPERT C for a variety of systems. Like all members of the ACE EXPERT Compiler family (C, Pascal, Fortran77, Modula-2, and COBOL) the new release employs heavy global optimization at intermediate-code level (UNCOL).

ACE also offers a **portable IEEE Floating-Point Emulation Package**, ACE Libf80, written in C. For more information contact: John van Brummen or Doeke Harkema at ACE Associated Computer Experts bv., Amsterdam, The Netherlands, (31) 20 6646416 (phone), (31) 20 750389 (fax), 11702 (ace nl) (telex), *uunet!ace.nl!john.*

- ACE, along with HCR of Canada, recently announced the latest version of their combined **ANSI C validation suite SuperTest**. The site license fee is US$32,000. For information contact ACE (see entry above) or HCR in Toronto on (416) 922-1937 or *uunet!attcan!utzoo!hcr!paul.*

- **More on validation suites.** Early in February, Plum Hall held a press conference announcing support for the final ANSI C Standard in their validation suite. The license fee varies from one country to the next, but in the U.S. is US$9,500. Contact: U.S: Terri Chandonnet at Plum Hall (609) 927-3770; U.K: Neil Martin at BSI (908) 220-908; Japan: Takashi Kawahara at Advanced Data Control Corp., Tokyo (03) 576-5351.

- Quantitative Technology Corporation offers a C language library package that provides a **full IEEE implementation of floating-point arithmetic**, using only integer operations. The routines include the arithmetic operations (add, subtract, multiply, and divide), utility routines, and a full suite of translation routines to and from ASCII, IEEE single and double precision, and integer. The precision of the numeric representation can be set by the user, and is limited only by memory size. Rounding mode and rounding precision can be set as required, and full exception handling can be added. A companion software package provides elementary transcendental functions to the selected precision (up to 512 bits, more upon special request). For more information, contact QTC, 8700 SW Creekside Place, Beaverton OR 97223, or 1-800-234-0168. E-mail: *uunet!sequent!qtc!law.*

- **ANSI C Quick Reference Guides** by Rex Jaeschke. This $8\frac{1}{2}$ inch by $3\frac{1}{4}$ inch 16-page booklet contains the complete language and preprocessor. Library identifiers are listed alphabetically within each header and across headers. Everything from escape sequences to trigraphs. Contact Rex at the *The Journal* for copies or to license the artwork for your own customization.

- Liant (parent company of Ryan-McFarland and LPI) announced NEW C, an ANSI- and X/Open-compliant implementation for Motorola's 88000 RISC-based systems. Contact Jillian Harvey-Asquith on (508) 626-0006. Liant Software Corp., 959 Concord St., Framingham, MA 01701-4613.

- **Perennial**, vendor of a C language validation suite, has announced a validation suite for C++ Release 2.0 that runs on UNIX systems. Perennial, 4677 Old Ironsides Drive, Suite 450, Santa Clara, CA 95054. (408) 727-2255. *uunet!sun!practic!peren!beh.*

- Amalgamated Software of North America has released **DIPLOMA/C for the IBM AS/400**. Anne Ferguson, ANSA, 4041 McArthur Blvd., Suite 300, Newport Beach, CA 92660. (714) 752-7991. Not to be outdone, **IBM has released IBM C/400** for the same machine.

- Late in 1989, Ken Thompson and Dennis Ritchie of AT&T Bell Labs received the 1989 Computers and Communications Prize, sponsored by NEC Corp. (Japan) for their work on UNIX and C.

- **To get a copy of AT&T's C++ Release 2.0** manual call them on (800) 432-6600. Price is $25 and the product select code is 307-146.

- Jensen & Partners International have released **TopSpeed C**, a highly optimizing compiler for DOS and OS/2. There's an unconditional 30-day, money-back guarantee. 1101 San Antonio Road, Suite 301, Mountain View, CA 94043. (415) 967-3200.

- Late in 1989, a self-paced training textbook called *Mastering Modern C* was published by Professional Press. It is written in ANSI Standard C, covers the complete language and is designed as a training book suitable for seminars. Written by Rex Jaeschke, copies are available from the publisher at (215) 957-1500. To license the companion training materials contact Rex at *The Journal*.

- Chorus Supercomputer, Inc., is now shipping version 1.1 of its **Linda C language implementation for the Macintosh**. Cost $450. Chorus also sells a line of hardware called ComputeServer which also supports Linda C development. 100 Varick Street, New York, NY 10013. (212) 925-1715.

- Silicon Valley Software announced SVS C for UNIX (on MC68K, Intel 386/486, and N32x32) and 32-bit DOS-based systems. 1710 South Amphlett Blvd, Suite 100, San Mateo, CA 94402. (415) 572-8800. Has K&R, pcc, ANSI, and POSIX modes.

- **Note:** Copies of final ANSI standards are available from:

  American National Standards Institute
  Sales Department
  1430 Broadway
  New York, NY 10018
  (212) 642-4900
  fax (212) 302-1286

  The ANSI C Standard's official designation is ANSI X3.159-1989.

∞