The Journal of

C Language Translation

Volume 2, Number 4

March, 1991

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

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

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

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

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

Contents

28. Implementing Locales

P.J. Plauger

Abstract

The Standard C library includes a facility called *locales* that helps a program adapt to a number of local customs. Because it is new, the Standard leaves the machinery largely unspecified. Little prior art exists to guide the implementor.

This article describes the basic requirements for locales. It reviews several implementation considerations and proposes ways to deal with them. The proposed solutions have been tried as part of a complete Standard C library that is commercially available.

Background

Committee X3J11 added locales to Standard C as a pure invention. They were contrived to satisfy the stated needs of a number of European participants. These folk objected to several Americanisms that had become institutionalized in C—such as the dot for a decimal point and most aspects of the function `asctime` (including its name). The same group of people were also concerned with writing applications that could be sold in a single version for use in multiple cultures. Thus, it was not enough to remove the Americanisms. It must also be possible to inject a variety of other sets of conventions in their stead.

An applications program that uses locales wisely can adapt to local custom in many important ways. Yet the code can be written without detailed knowledge of all possible locales. And it can be maintained in a single version. Even Americans, with their large local market for software, can appreciate these benefits. Software is an important export, but producing foreign versions can add significant development costs.

Machinery has long existed in C to assist programs in adapting to changing conditions. UNIX added *environment variables* about fifteen years ago. These provide a set of names, each with a character-string value. Environment variables get smuggled among processes with no extra effort on the part of the programmer. A C program can call `getenv` to inspect the character string currently associated with a name. (In some implementations, a program can also call `setenv` to add or alter an environment variable. That capability is *not* a part of Standard C, however.) Such machinery has proved so useful that it has popped up in nearly all C environments, from MS-DOS to MVS.

A variety of conventions has grown up around environment variables. Time zone information can be found in TZ, for example, in a broad assortment of C implementations. Sadly, the format of the TZ value string is inadequate to represent all the time zones on the planet. Some implementations thus provide time zone information in an alternate form, sometimes by a different name. That typifies the problems with environments:

- Few standards exist for the names or meanings of environment variables.
- Those standards that do exist are often inadequate for the international marketplace.
- Many environments are cluttered with environment variable names used for varied and undocumented purposes.

In a real sense, a locale is simply a structured set of environment variables. It provides a standard way for naming the components of a related set of values. Some thought went into selecting and defining the components to meet the needs of programmers around the world. And a locale imposes a level or two of structure on the information to help control the proliferation of names.

For example, you can determine the current decimal point by writing:

```
#include <locale.h>
#include <stdio.h>
        ...
        struct lconv *p = localeconv();

        printf("numbers look like 3%c14\n",
                p->decimal_point[0]);
```

A program that never calls the function setlocale will assuredly display the sample number as '3.14'. The dot is what you get in the "C" locale. And the "C" locale is what you get at program startup. In this locale, the Standard C library behaves largely the way a generation of programmers has come to expect—Americanisms and all. The "C" locale is our island of stability in a torrent of new cultures.

On the other hand, the program might include one or more calls to the function setlocale. In this case, the test sequence above shows you what currently passes for a decimal point. What the printf call displays is exactly what that function uses to display floating numbers. You could just as well have written:

```
printf("numbers look like %.2f\n", 3.14);
```

The displayed lines should be identical. (Note, by the way, that altering the locale does *not* alter the character you use for decimal point when writing formats or floating constants. That way madness lies.) As you might expect, the function `strtod` also alters its notion of decimal point when the locale changes. That, in turn, alters the behavior of the functions defined in terms of `strtod`—the `scanf` family and `atof`.

COBOL has its '`DECIMAL POINT IS COMMA`' clause. A locale in C can specify comma or any other character for the decimal point. Of course, locales go far beyond the modest control of COBOL. Among other things, locales let you alter:

- How the `ctype.h` functions categorize characters, so that you can specify additional letters
- How the multibyte functions map between multibyte strings and wide characters
- What to use for a currency symbol, and how to format various currency amounts
- What to use as a digit separator (such as the thousands comma in the U.S.) and how to group digits for non-currency amounts
- How to display dates and times

Some of the information provided with a locale is purely advisory. A program can read it and act on it if it chooses. The Standard C library is otherwise unaffected. Still other bits of information alter the behavior of one or more library functions, as in the example above. Here is where locales can be most powerful, and most dangerous. Switching to an appropriate locale can help your program speak more kindly to the locals. It can also subvert existing program logic by adding unexpected letters or altering numeric conversions. Be careful.

You should, in fact, write every Standard C program with one of three styles in mind:

1. Stay in the `"C"` locale as in simpler days of yore. Never call `setlocale`.
2. Switch once and for all to the *native* locale and stay there. Place the call

   ```
       setlocale(LC_ALL, "")
   ```

   at the top of `main` and watch out for library functions that change behavior.
3. Adapt like crazy to different locales, or categories within locales, throughout the program. Call `localeconv` and `setlocale` in all its variations. Be ever alert to the sands shifting beneath your feet.

I recommend that you not even consider coding in the third style until you gain some experience in the second.

Requirements

As desirable as they may sound, locales remain largely untried. The earliest translators claiming ANSI conformance provided minimum support for locales. Only a few major corporations have so far announced more ambitious plans to support varied locales. The POSIX committee seems to be hammering out locales in excruciating detail. But there is little in the field today to give you a feel for how locales are supposed to work.

The C Standard also leaves certain aspects of locales intentionally unspecified. It mandates the existence of only two locale names, `"C"` and `""`. (The latter is for the native locale, whatever that may be.) It specifies six categories, but permits an implementation to add more. It says nothing about the way an implementation names a locale, defines its content, or makes it available to a C program.

I have heard a number of gripes about this lack of direction, but I feel it is appropriate. Think of locale names, for example, the same way you think of file names. The C Standard imposes a couple of length constraints on file names (`FILENAME_MAX` and `L_tmpnam`). It requires a minimum set of names for header files (such as `"abcdef.h"`). Beyond that, it says *nothing* about the form of a filename. It certainly doesn't mandate any specific file names. (The fifteen standard headers are not necessarily files any more.)

Of course, many conventions exist for naming collections of files. Nearly all systems require, or strongly presume, that C source files have names ending in `.c`. And include files have names ending in `.h`. Any significant project imposes even more constraints on file names and directory structure. But that is not the business of the C Standard. For it to say anything, even about C source file names, would be inappropriate.

The C Standard adopts much the same attitude toward locale names. As desirable as it may be to have common names for locales, the C Standard can't mandate them. It was risky enough to stick in locales with little or no prior art. To tackle the open-ended problem of coordinating names around the world would have been disastrous. So part of implementing locales is to establish rules for naming them.

There's an even nastier naming problem. A program can create a *mixed locale*, as in:

```
#include <locale.h>

static char *savestr(char *str) { ... }
        ...
        char *s0 = savestr(setlocale(LC_ALL, NULL));
        char *s1 = savestr(setlocale(LC_ALL, ""));
        char *s2 = savestr(setlocale(LC_NUMERIC, "C"));
```

(Here, the function `savestr` allocates space for the string argument and makes a copy of the string. Otherwise, `setlocale` might overwrite the string on a subsequent call.)

The first call returns the name of the locale currently in effect. You can undo the effect of the next two lines at a later point by writing

```
        setlocale(LC_ALL, s0);
```

The second call switches to the native locale. The last call to `setlocale` reverts the numeric category back to the `"C"` locale, leaving all others alone. This is a mixed locale.

For each of these calls, not just the first, `setlocale` must return a string that it can later accept as a second argument. A call such as

```
        setlocale(LC_ALL, s2);
```

must recreate the mixed locale. Whatever information is needed to do so can be stored only in the string returned by `setlocale`. That can pose some interesting challenges. (Several people objected to having the C Standard present such challenges.)

Implementation

About a year ago, I started writing a portable implementation of the Standard C library. I wanted to convince myself that a number of untested conjectures in the C Standard are correct. In particular, I wanted to see if locales and multibyte support could be implemented reasonably. If so, I wanted to provide an exemplary implementation for others to use, if only as a starting point. That library is now complete and will soon be published by Prentice-Hall in a book called *The Standard C Library.*[1]

The implementation of locales that I describe here is from that book. I don't represent it as the only way to do the job. It may well not be the best way to meet the goals of any given implementation of C. It does, however, meet the requirements of the C Standard. And it looks to be both usable and powerful.

Let's begin with the data structure. The standard header `locale.h` defines the type `struct lconv`. You call `localeconv` to get a pointer to such a structure. That gives you access to information in the monetary and numeric categories.

The C Standard strongly suggests that `localeconv` deals with a single static structure. Each call to `setlocale` rewrites the fields of this structure as needed. Presumably, you can call `localeconv` once in your program and trust that the returned pointer remains valid until program termination. I would discourage such a presumption as bad programming style, however. It is equivalent to

[1]The machine-readable source, over 9,000 lines of C code, will also be available at a reasonable price.

assuming that every call to `gmtime` returns the same pointer value. Maybe it's true. Maybe it even *has* to be true. But it's still a risky presumption.

From an implementation standpoint, using a single static structure makes the most sense. You have the usual problems with writable statics in the library. A system that wants to share library code among processes must work out how to give each a private copy of writable statics. The problem here is no better and no worse than elsewhere in the library. In most implementations, library statics present no problem. They can even be declared and initialized like any other static data in C.

One benefit of having an initialized static structure is that `localeconv` becomes a trivial macro. It expands to the address of the structure, as in:

```
struct lconv *localeconv(void);
extern struct lconv _Locale;
#define localeconv() (&_Locale)
```

The call overhead disappears and references to members of the locale structure become direct accesses to memory. What it costs you is more work on a call to `setlocale`. You can't avoid copying data into the locale structure when a category changes.

A locale contains information about other categories as well. How you represent this additional information is not spelled out in the C Standard. You can certainly add fields to `struct lconv`. Or you can make `struct lconv` one component of a larger structure. A small case can be made for favoring the latter approach.

Here's why. It seems desirable in principle to pack all locale-dependent data into a common data structure. That includes at the very least

- The three character maps for the functions in `ctype.h` (for `tolower`, `toupper`, and the `is*` functions)
- The tables that control the behavior of `mbtowc`, `wctomb`, and `strcoll` (and all the functions that call on them in turn)
- The day and month names, and possibly other data, used by `strftime`

But think what happens with a typical linker. Say you call only the function `isspace`. That drags in the character map for the `is*` functions, which is now a part of the locale structure. The locale structure in turn drags in all the other maps, tables, and strings whether or not they get used. Customers don't like to link in 5,000 bytes when they expect at most 300. I don't either.

My solution was to leave all these bits of data separate, just like in the old days. The first call to `setlocale` copies data as needed into the data structure for the `"C"` locale. Every call to `setlocale` copies data out to the separate bits whenever they must change. That's extra work for `setlocale`, but it's already an expensive function. You at least avoid the wasted storage space in programs that don't muck with locales at all.

Names

Implementing `setlocale` and naming locales are inseparable issues. The problems to solve, more or less in order, are:

- What do you choose for a native locale (with name `""`)?
- How do you represent an open-ended set of locales?
- How can you standardize locale names, and specifications, across multiple implementations?

For the native locale, you have three basic choices:

1. An implementation with minimum locale support can make the native locale the same as the `"C"` locale. No other locale names are defined.
2. An implementation with a strongly favored locale can build a special native locale directly into the library. You can switch to it quickly. Other locale names may or may not be permitted.
3. An implementation with more ambitious goals should select the native locale from an open-ended set. The selection should be made with little or no additional input from the person using the program.

I am in the business of implementing the third choice. (You can then get one of the first two by throwing away code.) It seems to me that the best mechanism for specifying the native locale is one I pooh-poohed earlier in this article—reading an environment variable. I believe that this is what environments do best, provide a small hint that leads to a significant amount of tailoring. Consequently, I commandeered the environment variable LOCALE to specify the native locale.

The first time the program calls `setlocale` with `""` as the second argument, the function calls `getenv("LOCALE")`. If that returns a value, it is taken as the name of the default locale. Otherwise, the default locale is `"C"`.

You can add any number of locales directly to the Standard C library. Each is represented by an initialized read-only static data structure. It is not terribly difficult to write the initializer, at least for the fields that commonly vary. Part of the initialization is to link the locale structures into a mono-directional list. A call to `setlocale` scans this list to find any locales already in memory.

Beyond a certain point, it is better to read in additional locales as needed at runtime. For that to happen, the implementation needs to know where to find the file of locales. It also needs to know how the file is formatted. Since it is extremely bad form to wire a file name into code, particularly library code, I commandeered yet another environment variable. (The entire library uses about half a dozen.[2]) The environment variable `LOCFILE` contains the name of a file to read if `setlocale` cannot find a locale already in memory.

[2]The library uses the following environment variables: `LOCALE`, `LOCFILE`, `TEMP`, `TIMEZONE`, and `TZ`.

You describe locales in a text file, for ease of editing by people. Each line begins with a keyword, followed as needed by values and expressions. To minimize input, each locale begins as a copy of the "C" locale. All you need specify is any changes from this simple norm. A brief example of a locale is:

```
LOCALE france
NOTE sampler only, not complete
decimal_point ","
toupper['ê'] 'Ê'
tolower['Ê'] 'ê'
ctype['ê'] $L
ctype['Ê'] $U
currency_symbol "F"
int_curr_symbol "FFR "
```

You can also write more powerful expressions. The ASCII behavior of `toupper`, for example, can be expressed completely as:

```
toupper['\0' : $^] $@
toupper['a' : 'z'] $$+'A'-'a'
```

The first line sets each element (from `'\0'` to `CHAR_MAX`) to map to itself. The second adds to the range of lowercase elements the appropriate offset to map to the corresponding uppercase letter.

I have written just a few locales so far. That has been enough to convince me that the basic notation is economical and complete. In time, I hope to tackle an assortment of more complete specifications from POSIX or X/OPEN. That may well lead to additional refinements in the locale file format.

To name a mixed locale, `setlocale` bolts together these simpler locale names. Say the native locale is `"france"`. Then the example I gave earlier would have `s2` point at the string `"france;numeric:C"`. A semicolon separates category components within a name. A colon separates the category name from the locale name to which it is set.

Every string that `setlocale` returns has an unqualified name. It is determined on the last call of the form `setlocale(LC_ALL, name)` where `name` has an unqualified component. Any categories that differ from this one contribute a component of the form `;cat-name:locale-name`. You can call `setlocale` with even messier names, however. For each category that gets set, `setlocale` chooses, in order:

- The first qualified name for that category
- Otherwise, the first unqualified name
- Otherwise, the existing locale name for that category

The effect of these rules is that you can prefix any string returned by `setlocale` with one or more qualifiers. Those qualifiers will win where appropriate. Any unmodified categories shine through as needed.

The last item on the checklist is developing locale names and specifications that are portable across multiple implementations. This library is portable enough to solve that problem in the small. Anybody who chooses to make use of it can share locales with others who do the same. (A change in the underlying character set, such as from ASCII to EBCDIC, requires changes in the locale file, of course.)

Solving the problem in the large requires a degree of cooperation among implementors. I don't believe that implementors can make wise decisions until they get some hands-on experience with locales. That is a major purpose of this exercise. I am not in a position to dictate conventions among all C users and I don't want to be. My goal is to help people gain experience with using locales so more of us will one day know what we want.

Finite-State Machines

I conclude by mentioning a particularly nasty aspect of having locales that vary. You want a function such as `strxfrm` (and its companion `strcoll`) to be reasonably fast. That argues that you write a function tailored to a given collation sequence and include it in the library. To switch to another collation sequence, you switch to another function in the library. What goes in the locale structure is a pointer to the current version of `strxfrm` (and possibly `strcoll` as well).

That decision seriously limits your choices when specifying new locales. All you can choose from are the existing flavors of `strxfrm`. You want a new flavor, you write it in C and add it to the library. Many implementations will find such a limitation acceptable, but not all.

A more flexible solution is to write a single function that is table driven. It is relatively easy to specify the contents of a table as part of a locale. Certainly, that is easier than writing C functions and adding them to the library all the time. The function may run a bit slower, but for a well-designed table encoding it can be comparable to bespoke code.

Collating rules come in an astonishing variety. (So, too, do multibyte escape sequences and wide character encodings.) I tried several times to contrive a table format that encompassed a significant number of popular rules. I think I came close, but not close enough to be elegant. What I gravitated to in the end was the same basic machinery for `strxfrm`, `mbtowc`, and `wctomb`. I implement each as a table-driven finite-state machine. The variety of schemes they can implement at least borders on the astonishing.

Sadly, I couldn't contrive a way to use a single finite-state machine driver for all occasions. It seems that `mbtowc` converts one or more bytes to an `int`. `wctomb` does the opposite. And `strcoll`/`strxfrm` maps bytes to bytes. The methods are similar but different.

Coding tables for a finite-state machine is not for the faint of heart. I don't pretend to have eliminated the need for a programmer. What I have done is provide a way to represent a broad class of mapping functions as part of the text specification for a locale. I will happily provide implementation details, and a representative assortment of applications—in a later article.

P.J. Plauger serves as secretary of X3J11, convenor of the ISO C working group WG14, and Technical Editor of The Journal of C Language Translation. He recently took over the editorial reins of The C Users Journal. He is currently a Visiting Fellow at the University of New South Wales in Sydney, Australia. His latest book, The Standard C Library *will soon be available from Prentice-Hall. He can be reached at* uunet!plauger!pjp.

∞

29. Resolving Typedefs in a Multipass C Compiler

W.M. McKeeman
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract

A C compiler must resolve the ambiguity between variables and typedef names during parsing. This requires the parser take into account extra-syntactic information. The information is typically held in the compiler symbol table. This paper outlines a solution where the compiler symbol table is not available. The solution is to build a minimal symbol table in the parser itself.

Introduction

C fails to be LR(1) because of a conflict between *identifier* and *typedef-name*. The situation is illustrated by the following fragment:

```
        static X(Y)
```

This text starts a declaration of `Y` if `X` is a *typedef-name* and `Y` is not. It starts a function prototype for `X` if `Y` is a *typedef-name* and `X` is not. It starts an old-style *function-definition* if neither `X` nor `Y` is a *typedef-name*. There are similar conflicts for casts and parenthesized expressions and function calls.

C programmers have little difficulty resolving these conflicts—the declared attributes of the names are sufficient. A syntax-driven parser, on the other hand, makes all decisions based on the immediate syntactic context. A previous type definition is not part of that context, thus something additional must be done.

The following description assumes the reader is familiar with compiler structure and parsing methods.

The typical C compiler scans source text, parses it, and builds a symbol table in a single pass. The *typedef-name* resolution takes place in an enhanced scanner which builds a *typedef-name* token instead of an *identifier* token when the scanner finds that token in the symbol table marked as a *typedef-name*. The parser is unaffected by this collusion between the scanner and symbol table [2]. Alternatively, the parser can interrogate the symbol table in those places where the conflict arises [3]. In this case the scanner is unaffected.

Where parsing is done on a separate pass from interpreting declarations, the symbol table is not available to either the scanner or the parser. A solution for unavailable symbol information is presented here in terms of Standard C as defined in the American National Standard X3.159-1989 [3] [1]. The basis of the solution is a private symbol table built into the parser, capable only of resolving the *typedef-name*/*identifier* ambiguity.

It turns out that the parser needs to know only in a small number of places whether an *identifier* is a *typedef-name* [§A.2.2, 3.5.6] so as to apply the grammar rule

> *typedef-name:*
>
> > *identifier*

The issue must be resolved in deciding between rules [§3.3.4]

> *cast-expression:*
>
> > *unary-expression*
> > `(` *type-name* `)` *cast-expression*

because a *unary-expression* might start with a parenthesized *identifier* and also between the two cases for each of the rules [§3.5]

> *declaration-specifiers:*
>
> > *storage-class-specifier declaration-specifiers*$_{opt}$
> > *type-specifier declaration-specifiers*$_{opt}$
> > *type-qualifier declaration-specifiers*$_{opt}$

because if the last *type-specifier* is an *identifier* it might instead be meant to be redeclared as belonging to the following declarator. The choice between *declarator* and *abstract-declarator* is formally ambiguous and therefore requires a special elaboration in the standard [§3.7.1]. There is a similar ambiguity between new and old-style *function-definition*.

Finally, the issue must be resolved between rules [§3.6.2]

> *compound-statement:*
>
> > *declaration-list*$_{opt}$ *statement-list*$_{opt}$

because of the ambiguity between function calls and declarations mentioned for `X(Y)` at the start of this section.

The parser accesses its private symbol table to resolve the local ambiguity rather than relying on the scanner. With this solution separation of concerns is cleaner. There is no question of the feasibility of the proposed solution since a complete symbol table could be built in the parser. The problem to be solved is keeping the parser's private symbol table simple, small, and efficient. The

[3]Throughout this paper, the standard is implicitly referenced using the notation §x.y.z

idea has appeared in a C compiler [3]; it does not appear to be documented in the open literature.

Extra-syntactic decision-making in a parser requires *ad hoc* modifications to the parser. The modifications can be ugly if the parsing method is inflexible, as is the case if the parser is table-driven and parser sources are not available. The parsers in which these ideas have been tested have used recursive descent.

There are a number of simplifying assumptions in this presentation. It is assumed that the only objective of the parser is to report the shift/reduce sequence implied by the grammar. It is assumed that there is a lexical process that produces actions of type `Token`. It is assumed that the parser produces actions of type `Rule`. The time-merged sequence of `Token` and `Rule` actions is the shift/reduce sequence. These assumptions are neither theoretically nor practically limiting: any intermediate form can be efficiently constructed from the shift/reduce sequence. It is also straightforward to produce this same output from LALR-based parsers [4].

The requirements on the solution are that all syntactically correct C programs can be parsed and that all syntactically incorrect C programs can be diagnosed. The code added to resolve *typedef-name* need issue a diagnostic only when it cannot be assured that other diagnostic facilities will come into play. Specifically, the correct parse must be provided up to the point where a nonsyntactic error will be diagnosed, and some parse continuation must be provided so that later phases of the compiler will actually be invoked—the output of the parser must always reflect the parse for some correct program.

The Parser Symbol Table

Symbol tables for C are required to reflect a number of detailed requirements and constraints of the language definition. Where there is a common solution for the special parser symbol table and the general table, no details are given here based on the presumption that there are other sources of this information.

The actions for the private symbol table are interrogation, entry of a new *typedef-name*, obscuring a *typedef-name* with some other use of its name, entering and leaving a scope.

Scoping is complicated by the requirement that names in a *parameter-type-list* and *identifier-list* [§3.5.4] are in the scope associated with the *compound-statement* of the *function-definition* even though they are outside the opening '{' [§3.1.2.1]. The solution presented here has a second kind of scope entry which reopens a just-closed scope frame. This makes six functions altogether in the typedef-resolving symbol table mechanism.

Typedef names are in ordinary name space [§3.1.2.3]. If `typedef` is encountered in a *translation-unit*, its identifier becomes a *typedef-name* until either the current scope is finally left, or another declaration for the same name in an inner scope temporarily obscures it. There can be at most one *typedef-name* entered in any one scope. There can be multiple entries for other ordinary name space uses of identifiers (because of the rules for linkage [§3.1.2.2] and old-style function definitions).

Only the grammar rules

> *direct-declarator:*
>   *identifier*
>
> *enumeration-constant:*
>   *identifier*
>   *identifier* `=` *constant-expression*
>
> *primary-expression:*
>   *identifier*

can introduce names into ordinary name space [§3.5.4, 3.1.3.3, 3.3.1, 3.3.2]. These three cases can be treated one at a time.

A *direct-declarator* introduces a *typedef-name* only when it eventually participates in the grammar rule

> *declaration:*
>   *declaration-specifiers init-declarator-list*$_{opt}$ `;`

and reserved word `typedef` is among the *declaration-specifiers*. If, on the other hand, `typedef` was *not* in the *declaration-specifiers*, *direct-declarator obscures* any use for that *identifier* in enclosing scopes.

The rule above for *init-declarator-list* leads to

> *init-declarator:*
>   *declarator*
>   *declarator* `=` *initializer*

Via the above nonterminal *declarator*, nonterminal *direct-declarator* participates in four other rules in C:

> *function-definition:*
>   *declaration-specifiers*$_{opt}$ *declarator declaration-list*$_{opt}$
>   *compound-statement*
>
> *parameter-declaration:*
>   *declaration-specifiers declarator*
>
> *struct-declarator:*
>   *declarator*
>   *declarator*$_{opt}$ `:` *constant-expression*
>
> *direct-declarator:*
>   `(` *declarator* `)`

In *function-definition*, `typedef` is syntactically allowed but never valid in either *declaration-specifiers* or *declaration-list* [§3.7.1]. This occurrence of *declarator* therefore cannot enter a *typedef-name*. And, since *function-definition* is always in the outermost scope, neither can it obscure a *typedef-name*.

In *parameter-declaration*, `typedef` is syntactically allowed but never valid [§3.5.4.3, 3.7.1]. A *typedef-name* cannot be entered, but one can be locally obscured by a parameter.

In *struct-declarator* ordinary name space names cannot be defined. Therefore, all parse symbol table activity must be suspended for *struct-declarator*.

Whatever is said about *direct-declarator* also applies to the parenthesized *declarator*. Thus it may behave as any of the four uses of *declarator*. This concludes what must be done for *direct-declarator*.

The situation for *enumeration-constant* is much simpler—whenever it appears it obscures any other use of its identifier in outer scopes. There is no need to check for multiple definition in the current scope—later phases of the compiler will do that.

The occurrence of a *primary-expression* can introduce a local ordinary name space object with external linkage when the *primary-expression* is immediately used in the rule

> *postfix-expression:*
>   *primary-expression* `(` *argument-expression-list*$_{opt}$ `)`

and the *identifier* is not previously declared. The effect is to obscure other local uses of that name from the point of implicit declaration. In fact the parser can ignore implicit declarations. They cannot obscure a *typedef-name* because there can be no previous declaration (of any kind) for that name.

Implementation Details

While it violates a constraint to have two declarations for the same name in one scope [§3.5], it is not necessary to check this constraint to parse correct programs. In essence, C is extended during parsing so that, instead of diagnosing multiple declarations, the last declaration wins. This trick always results in a decision on *typedef-name* and makes the diagnostic correct with respect to the nearest declaration. This decision only affects incorrect programs, thus there is no substantial impact on efficiency of parsing.

Six parser-specific routines need to be implemented:

```
void ParseEnterScope(void);
void ParseExitScope(void);
void ParseReenterScope(void);  /* undo Exit */
void ParseEnterTypedef(Token t);
void ParseObscureTypedef(Token t);
bool ParseIsTypedef(Token t);
```

Function `ParseIsTypedef(t)` is what is needed by the parser to distinguish between ordinary *identifier* and *typedef-name*. Everything else is just support for this function. `ParseIsTypedef(t)` is called just in the situations where both a *typedef-name* and object are syntactically acceptable and the parse depends on which is actually found.

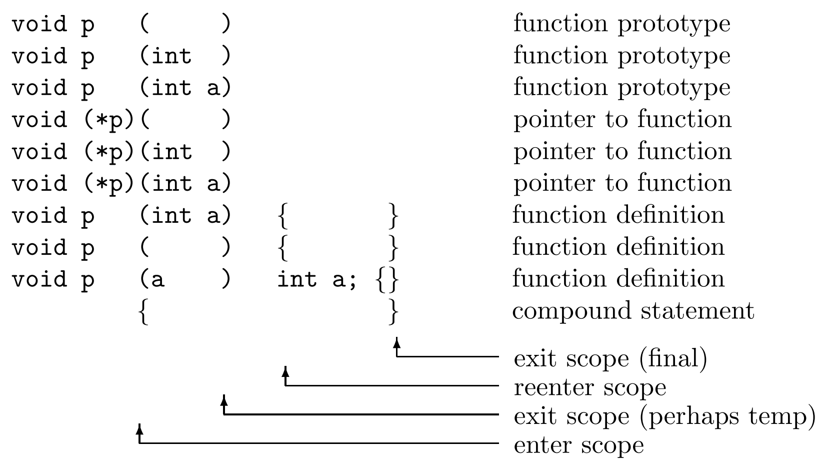
Functions `ParseEnterScope()` and `ParseExitScope()` are usually paired. The scopes in C are associated with one of file-scope, function body, compound statement, or prototype. These two functions are called as each scope is entered, and as it is exited.

The call of function `ParseReenterScope()` is immediately preceded (in time) by `ParseExitScope()` and restores the parse symbol table to the state it had just before it did the exit [5]. The trick is to merely set a global delay flag in `ParseExitScope()` and not do the exit action. If the next call is `ParseReenterScope()`, it has no effect except for the clearing of the delay flag. All other actions check the delay flag and if it is set, actually do the exit action prior to doing their own functions. The re-enter situation occurs for function definitions. Function definition is detected syntactically when a file-scope *declarator* is immediately followed by other than one of '`,;=`'. The effect is to include the formal parameters of the function in the scope of the function body.

`ParseEnterTypedef(t)` pushes `t` on the parse symbol table and marks it as a *typedef-name*. `ParseObscureTypedef(t)` pushes `t` on the parse symbol table and marks it as not a *typedef-name*. The obscuring action is effective until the end of scope occurs, causing `ParseExitScope()` to be called.

Invoking the typedef symbol table functions can be done in any of the conventional ways for complete symbol tables.

Scope Management



```
void p   (     )                 function prototype
void p   (int  )                 function prototype
void p   (int a)                 function prototype
void (*p)(     )                 pointer to function
void (*p)(int  )                 pointer to function
void (*p)(int a)                 pointer to function
void p   (int a)  {        }     function definition
void p   (     )  {        }     function definition
void p   (a    )  int a; {}      function definition
         {                 }     compound statement
```

Scopes are associated with *parameter-type-list*, *identifier-list* and *compound-*

statement. They interact [§3.1.2.1]. There are also some additional rules that allow the former status to be ignored [§3.5.6]. After a function header scope is closed (perhaps temporarily) scope is reopened for the function body since the parameters are in the same scope as the body. This forces `ParseEnterScope()` or `ParseReenterScope()` to be called before parsing compound statements.

Test Case

The syntax for C declarations permits writing declarations that are hard for programmers to decipher. Compilers have the same problem. The following C program compiles and runs. It contains multiple definitions of name `p`, each using a different combination of *typedef-name*s, to be analyzed for compatibility. At the same time the type definitions are being obscured in local scopes because the same names are used for objects. `J` appears 8 times, once as the name of a parameter. `I` appears 10 times, 4 as the name of a parameter. The reader might want to predict the number of times '`in p`' gets printed.

```
#include <stdio.h>

typedef int I;          /* I is a typedef-name */
typedef I   J(I());

I(i);                   /* i is an int variable */

extern J p;             /* p takes an int function */
extern I p(J), p(J I), p(J J);
extern int q(register J), q(J register), r(const J J);

I p(J I) {              /* I becomes a local name */
        puts("in p");
        return i++<0 ? I((J*)p(I)) : -17;
}

main() {
        i = -2;
        printf("%d\n", p(p));
}
```

Error Behavior

This mechanism must behave correctly in the face of source program errors. The principal issue is avoiding incorrectly classifying an identifier (*typedef-name* or not). The situation arises when the user incorrectly declares an identifier twice in the same scope. Since the only report from this mechanism is the single bit

`ParseIsTypedef()`, there are exactly two problems: TRUE overriding FALSE and vice versa.

The consequence of an incorrect value is a parsing error later in the source text. The compiler cannot know which declaration was intended. It is barely acceptable to report a parsing error at some point well beyond the declaration that caused the problem. The mechanism outlined here could be extended to detect and report redeclarations that could affect the value of `ParseIsTypedef()` without affecting the interface defined above.

Acknowledgements

The central idea of this paper was brought to the attention of the author by John Hamby in relation to the DEC C implementation. Since then many of my co-workers have contributed ideas and suggestions. The implementation described here is from a prototype incremental compiler for Standard C.

References

[1] *American National Standard for Information Systems – Programming Language C, X3.159-1989*, American National Standards Institute, 311 First Street, N.W., Suite 500, Washington, DC 20001-2178.

[2] Harbison and Steele, *C: A Reference Manual*, 2nd Ed. Prentice-Hall (1987). See section 5.10.3.

[3] John Hamby, private communication, June 1989.

[4] W.M. McKeeman, Shota Aki, and Scot Aurenz, "Parser-Independent Compilers," *Journal of C Language Translation*, Vol 2, No. 3 (December 1990).

[5] Randy Meyers, private communication, July 1990.

William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.

∞

30. A Parallel Extension to ANSI C

Rob E.H. Kurver
PACT
Foulkeslaan 87
2625 RB Delft
The Netherlands

Abstract

The ANSI X3J11 committee decided not to address the issue of parallel processing. Instead, it chose to wait until developers had come up with more prior art. This paper describes an effort to integrate the Communicating Sequential Processes (CSP) paradigm with ANSI C. The resulting ANSI C superset provides a familiar and powerful language for parallel processing. The concept has been tested in a compiler for the Inmos transputer.

Introduction

In the Communicating Sequential Processes (CSP) parallel programming model a *parallel program* is, in a nutshell, a collection of *sequential processes*. The processes can be distributed over any number of processors, with distributed or shared memory. Communication between pairs of processes is synchronized.

A process is *sequential* when there is a single flow of control. A process can spawn child processes, at which time the parent process is descheduled. That is, the parent process will be idle until all of its child processes are finished. This ensures the single flow of control in the sequential process.

Interprocess communication and synchronization are provided for via *channels*. For communication to take place over a channel, one process must be ready to send data over the channel and another one must be ready to receive that data. Both processes explicitly refer to the channel to be used for communication. Since no more than two processes can use a given channel at the same time, channels are typically allocated for every pair of processes wishing to communicate. If one process indicates its willingness to communicate over a given channel, it is descheduled until another process indicates its readiness to be the matching party.

It is possible for a process to guard a number of channels and select one available process to communicate with, or give up after a certain period of time.

Adding CSP support to ANSI C

The PACT Parallel C dialect is an attempt to enrich ANSI C with the CSP paradigm. For this purpose, two new constructs and one new type modifier were added to ANSI C. An implementation for the Inmos transputer proved that it is feasible to actually implement these extensions and put them to good use.

Perhaps the biggest problem with adding CSP to ANSI C was that of retaining the 'spirit of C.' Somehow, everybody just knows what feels like C and what doesn't. But how? Although it's possible to come up with some guidelines, some more intuitive than others, the final process of determining syntax and semantics for the extensions often depended more on aesthetics and gut feelings than anything else.

Processes

The basic building block of CSP is the sequential process. A *sequential process* is simply a piece of C code with its associated data. Each process has its own stack. In addition, child processes have access to the parent process' local data, in accordance with normal C scope rules. (Although conceptually very reasonable and simple, this does introduce the need for a static chain to be followed to the stacks of parent processes.)

A *task* is a collection of processes that have access to the same data segment, and thus can share data via more conventional means (shared memory, semaphores, etc.) in addition to channels. A task always starts as one process, very much like a conventional C program, with code and data sections. As this process spawns child processes, it is in the 'spirit of C' (re scope rules) to expect the child processes to have access to the parent's environment just as a normal C block would have. Hence the necessity for the static chain and the sharing of the data section. Although it is more in the spirit of CSP to use channels for all interprocess communication, it is convenient to be able to use other programming techniques if desired.

A *program* is a collection of tasks distributed over a number of processors. The allocation of tasks to processors may be static or dynamic. An implementation could even choose to move tasks about processors at runtime, depending on the workload.

Implementations may enforce particular restrictions regarding the distribution of processes and tasks. For instance, on a distributed-memory system, the sharing of the data segment pretty much forces all processes in a task to be run on the same processor. On a shared-memory system, however, it would be perfectly reasonable to allow a task to be distributed over multiple processors.

The `par` construct allows a process to start child processes and wait for their completion:

par-statement:
    `par` *replicator*$_{opt}$ *statement*

replicator:
    `(` *expression-1* `;` *expression-2* `;` *expression-3* `)`

This construct can be regarded as the parallel counterpart of the ANSI C `for` construct. Whereas `for` executes the loop body a number of times in sequence, `par` executes all loop bodies concurrently.

The replicator, if present, looks and feels just like the set of controlling expressions in a `for` construct. The replicator forms a loop in which the child processes are started, as follows:

```
par ( ... ; ... ; ... )
        start statement as child process

wait for child process(es) to finish
```

An example would be to calculate cube values concurrently as in

```
par (i = 0; i < 10; i++)
        printf ("%d**3 = %d\n", i, i * i * i);
```

In order to allow a number of different processes to be executed in parallel, if the body of the `par` statement is a compound statement (block), it is treated specially. Instead of starting this compound statement a number of times as a process, the compound statement is treated as a collection of declarations and statements, each of which is started as a separate process. Any declarations are declared global to the set of processes in this loop through the replicator, as follows:

```
par ( ... ; ... ; ... ) {
        initialize a copy of the local variables
        start every statement as a child process
}

wait for child process(es) to finish
```

The following code fragment starts three sets of two processes each. Each process calls a function `g()` with some shared parameter `x`, which is calculated from the replicator variable `i` before the processes are started. Each of the three sets of processes has a local copy of `x`, which is initialized before the processes using it are started.

```
par (i = 0; i < 3; i++) {
        int x = f(i);

        g(x);
        g(x + 1);
}
```

Because each process has its own stack, it is not possible to jump into or out of a process. Thus, `goto`s into and out of a process are disallowed, as are `continue`s and `return`s. Breaking out of a process transfers control to the end of that process, causing it to end. Of course, `break` and `continue` have their usual meaning in the appropriate constructs (`switch`, `for`, etc.) within the process.

The variables referred to in the replicator expressions are called the *replicator variables*. Each child process gets a private copy (a *replica*) of any replicator variables upon invocation. The main reason for this is to make absolutely clear which variables will be replicated, since this implicit copying of a variable is a new concept in C. Any variable can simply be replicated by merely mentioning it in the replicator expressions. If you do not want to replicate a variable because it must be shared, introduce a temporary copy of the variable. Using this copy instead of the original in the replicator expressions, causes the copy to be replicated and the original to be shared by all child processes.

Unfortunately, the `par` construct as outlined above introduces a parser conflict. An opening parenthesis after the `par` keyword could indicate either a replicator or an expression statement. This conflict is resolved in favor of the replicator, thus forcing the following piece of code to be rewritten using extra braces:

```
par
        ( expression );
```

Instead, this must be changed to:

```
par {
        ( expression );
}
```

Note that this is a non-functional use of the `par` construct (starting one single child process), and this is the only case in which the parser conflict forces code to be rewritten to favor the expression. (More than one process can be started by using either a replicator or a compound statement—the conflict doesn't exist in either case.)

Channels

The channel provides for synchronized interprocess communication. Both processes using a particular channel to communicate must explicitly refer to the channel. This means we need some way to declare a channel as well as a way to read from and write to a channel.

The `channel` type modifier was added to the familiar ones (*pointer to, array of,* and *function returning*). Thus, a channel has objects of some principal type being passed through it. Casts can be used to send objects of a different type. The channel is said to be a *channel of some object.*

The '`|`' (vertical bar) token is overloaded to indicate channels in both declarations and expressions.

The syntax for *declarator* and *abstract declarator* was extended to include channels as follows:

declarator:
    pointer-or-channel$_{opt}$ *direct-declarator*

pointer-or-channel:
    `*` *type-qualifier-list*$_{opt}$
    `*` *type-qualifier-list*$_{opt}$ *pointer-or-channel*
    `|` *type-qualifier-list*$_{opt}$
    `|` *type-qualifier-list*$_{opt}$ *pointer-or-channel*

abstract-declarator:
    pointer-or-channel
    pointer-or-channel$_{opt}$ *direct-abstract-declarator*

An example of such declarations is:

```
int |ch;        /* channel of int */
T |*ch[10];     /* array of 10 pointers to */
                /* channels of some type T */
```

The qualifiers `const` and `volatile` can be combined with the channel type in the usual way. However, because a channel causes communication, it is volatile by definition, so the `volatile` qualifier is redundant. A `const` channel is a read-only channel.

An expression with channel type can be subjected to the `|` unary operator, in addition to the normal ones defined by ANSI C. So, *unary operator* now becomes:

unary-operator: one of
    `&  *  +  -  ~  !  |`

The operand of the unary `|` shall have channel type (i.e., "channel of *type*" where *type* is any object type).

The unary `|` operator denotes channel communication. If the operand is a channel of an object, the result is an lvalue designating the object. If the operand has type *channel of type*, the result has type *type*. If the operand is not a valid channel, the behavior is undefined. As a side-effect, this operator causes synchronized communication to take place. The receiving party is the process using the designated object, and the sending party is the process assigning to the object. The current process will be suspended until the other party is ready to exchange data. For example:

```
par {
        int |ch;                /* channel of int */

        |ch = 10;               /* send 10 */
        printf("%d\n", |ch);    /* report */
}
```

This starts two processes, one process sending an integer constant through a channel, and the other process reading this value from the channel and printing it.

Selection

In order to be able to choose between multiple processes to communicate with, or to probe a channel without immediately committing to communication, a mechanism is needed to select a process to communicate with. This is accomplished with the `alt` construct:

alt-statement:
    `alt` *replicator*$_{opt}$ *statement*

labeled-statement:
    ...
    `guard` *expression* : *statement*
    `timeout` *expression* : *statement*

The expression of each guard label has *pointer to channel* type. The expression of each timeout label has integral type. There may be at most one `default` label in an `alt` statement. (Any enclosed `alt` or `switch` statement may, of course, have a `default` label.)

An `alt` statement causes control to jump to or into the statement that is the `alt` body, depending on the states of the channels pointed to by the expressions on any guard labels, the timeouts indicated by the expressions on any timeout labels, and the presence of a default label. A guard, timeout, or default label is accessible only within the closest enclosing `alt` or `switch` statement.

If any of the channels pointed to by the guard label expressions is ready to communicate, control jumps to the statement following the matched guard label. Otherwise, if there is a default label, control jumps to the labeled statement. If no guard channel is ready to communicate, and there is no default label, the process is suspended until one of the guard channels is ready to communicate or one of the timeouts as specified in the timeout labels has expired.

If the guard expression evaluates to `NULL`, the guard label is effectively disabled and control is never transferred there. If the timeout expression evaluates to zero, the timeout label is effectively disabled and control is never transferred there. This makes it possible to dynamically disable guards or timeouts.

The replicator, if present, replicates the guards and timeouts. Unlike the `par` replicator, the `alt` replicator causes no implicit copies of replicator variables.

To illustrate, the following piece of code monitors an array of N channels. It waits at most one second for something to arrive and prints it:

```
#include <time.h>

extern int |ch[N];

alt (i = 0; i < N; i++) {

        guard &ch[i]:           /* guard channel */
                printf("from %d: %d\n", i, |ch[i]);
                break;

        timeout CLOCKS_PER_SEC: /* wait max 1 sec */
                printf("timed out\n");
                break;
}
```

If more than one process is ready to communicate, which label control is transferred to is implementation-defined. An implementation may choose to test the guards in the order in which they occur in the source. It may randomly choose one. The implementation could also choose to implement several algorithms, selectable with a pragma, for example.

As with the `par` construct, the optional replicator causes a parser conflict. Again, this is solved in favor of the replicator. This behavior is even less of a problem for the `alt` than it is for the `par`. The only way an opening parenthesis that follows the `alt` keyword does not constitute the start of the replicator is in the following construct:

```
alt
        ( expression );
```

This construct causes the process to be terminally suspended, and the expression is never evaluated, making it a rather useless construct.

Conclusion

In order to add the CSP paradigm to ANSI C, the language was extended with four keywords (`par`, `alt`, `guard`, and `timeout`). One operator (`|`) was overloaded.

The parser conflicts introduced by the optional replicators on `par` and `alt` are not seen as a big problem. Although it hurts a bit to add two conflicts to the grammar, there's no problem with real code, and the advantages of the optional replicator (as opposed to mandatory replicators which would not cause conflicts) far outweigh the minor disadvantages.

An implementation for the Inmos transputer family is currently in beta testing. Experience with the CSP extensions so far indicates that they are indeed intuitive to use and retain the 'spirit of C' rather well.

[Ed: The transputer implementation of the PACT Parallel C Compiler will be the subject of another paper to appear in a future issue.]

Rob Kurver is the founder and president of PACT. He can be reached electronically at rob@pact.nl. *Pact is a developer of transputer software development systems.*

∞

31. Electronic Survey Number 7

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 90 different people, with 23 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.

Compiler Exit Status Codes

Some compilers terminate with a useful exit status code (either via `exit` *or return from* `main`*) indicating the number of compilation errors detected. However, this behavior is not required by Standard C. Does your compiler provide this information? Given that this approach is probably the only way a validation suite can reliably tell if a compilation failed, would you support requiring such a feature in a future standard? What about making it a requirement for US (or other) Government acceptance now?*

- 13 – Compiler indicates none or at least one error
- 5 – Compiler returns error count
- 2 – Compiler returns one of a number of status values
- 1 – Compiler does not return a useful value
- 0 – Environment cannot supply a status code
- Comments:

    1. Since (on some systems) the exit status is limited to 8 bits (and perhaps even to 7), it is not possible to return the actual count of errors. So, given that this count can be truncated, truncating it down to 1 seems as sensible as any other maximum.

2. Some operating systems (e.g., VAX/VMS) assign certain bits in status values special meanings so it is hard to use the status to indicating the number of errors without changing those meanings. Standard C should be completely independent from any operating systems (host environment). If you try to add new meanings other than the Boolean value to the status, it will be impossible to provide a conforming implementation for some operating systems.
3. Our compiler exits with a status that depends on the severity of the errors encountered, if any:

   0 – no errors
   1 – 1 or more warnings
   2 – 1 or more errors
   3 – fatal error

   I'm not sure it's possible to standardize this behavior on all platforms, but it sure sounds like a good idea to me.
4. I take issue with the wording of your question. There are several ways that a validation-suite-driver could tell if there was a compilation failure. For instance, it could check to see if an output (object) file had been generated. If not, there was probably an error. If a core file was generated, there was probably an internal compiler error. If diagnostics were issued, then there were probably errors.

   The X3J11 committee went out of their way to avoid such operating system biases in their formulation of the standard. I see no good reason to undo their well-considered work for the sake of this one small point, especially when the desired outcome (i.e., a way to tell if the compiler found errors) is so easily obtainable in other ways.

   [Ed: You use the word 'probably' three times which is exactly my point. There is no way to be certain whether or not an error has occurred. Whether something was written to `stdout`*,* `stderr`*, or an object file is no guaranteed measure of compilation success or failure.]*
5. Our compilers return a count but I don't think it can be mandated in a language standard, because how errors are reported is an operating system-dependent feature.
6. We return some, but not much, information through the exit status. Zero means no errors, but there may have been any number of warnings. Otherwise, at least one error occurred. In particular, the only distinction—one not guaranteed to remain, either—is that a value of 10 means an 'early exit' (generally due to a signal) occurred while 2 means 'with errors' exit status.

   Test suites, strictly speaking, need not know the number of errors, just whether any diagnostics were issued. However, a warning is just as valid a diagnostic for conformance purposes as an error. We cannot modify the exit status to be anything other than zero when only warnings are issued, since Makes would then fail 'for no reason.'

The standard requires that the implementation define how diagnostics are recognized. I see no particular value in mandating that the exit status of the implementation be part of this recognition and there are good reasons not to do so. First of all, the means to acquire the exit status of a command is completely system dependent. Second, the bandwidth of the value can be very narrow (only 1 bit is necessary). Finally, since an implementation can issue diagnostics on a whim (as long as they do not cause a valid program to be rejected) and since the severity of a required diagnostic is unspecified, the requirement that "there must be at least one diagnostic issued" is at best an 'if', not an 'iff' testable.

To determine whether our implementation issued a diagnostic, read its standard error. If any characters were produced, there was at least one diagnostic. If at all possible, they are issued one per line and warnings are labeled as such.

7. This is clearly a quality of implementation issue. An exit code is not functionality different from any other form of diagnostic so there is little gain by insisting on an exit code in addition to a diagnostic. Also, on some systems (VAX/VMS comes to mind) a non-zero exit code is interpreted by the operating system and generates a system level message. I think that there are probably too many operating system problems that might make this an unrealistic imposition.

   For DOS and UNIX implementations I have always preferred to use some sort of error/no error exit code in addition to diagnostics. It's the main way to get Make to work properly.

8. This kind of thing is already a problem when interpreting the meaning of `system`'s return value. I can't see how it can be standardized.

9. The number of errors is seldom useful, especially because incorrect error recovery often yields spurious additional error messages. Quality of implementation? Yes. Requirement? No.

10. Our product returns a status code indicating the severity of the most severe message. This is common behavior for all our products that run on our platforms.

11. Having the compiler return the number of errors essentially usurps *all* non-negative possible return statuses. This is guaranteed to clash with just about every status value definition made by any OS that chooses to define such values.

12. A real problem occurs when a suite wants to cause an error. How do you tell if the detected error is correct. For example, a compiler that does not support prototypes will detect an error when the test was trying to create an error for an illegal prototype. (On some compilers

both cases would get the same error message; syntax error.) We solve this problem internally by comparing the compiler listing file to a *compare file*, but making an OS-independent form of this will be tricky.

Extended Characters in Identifiers

There are moves afoot at the ISO level to permit identifiers to include 'letters' from National character sets. For now, it would likely not include multibyte characters. Any support or objections? (Basically, this would involve a compile-time locale and any character testing True for `isalpha` *in that locale would be accepted, including such characters as '$', if you so desired.)*

1. No problem provided the standard specifies exactly which characters are letters in which locales.

2. As long as there is some way to easily translate these foreign identifiers back to (still unique) identifiers in English-based character sets, I see no reason for objection.

3. *[Ed: from Europe]* Why? What would be gained by this? How about translating the keywords into a different language? How about portability? Sounds like I would not be able to take source written in a character set (locale) that includes the '$' and compile it on a system without this compile-time locale. I take it a conforming program should not use these characters, then? And if you do use them, you can be sure no other compiler will be able to compile your code? That's even worse than pragmas!

4. This would reduce the portability of C code.

5. *[Ed: From Japan]* If we use special 'letters' from National character sets in identifiers, our identifiers will become Japanese-oriented and most foreign programmers will not understand their meanings. We want to maintain our programs in world-wide source form. I and all of the technical stuff in my company *cannot* find any necessity to use them and *do not like* to use them. And very few of our customers requested us to support this feature in our compilers. So I basically object to the feature but my objection is not so strong. But, if ISO permits them, multibyte characters *must* be included.

6. Am I required to support anything but the standard `"C"` locale? If not then it seems they are defining a variety of extensions rather than changing the standard.

7. I am a little surprised by this development. I recall that when we [X3J11] deliberated over locales and character sets, the issue of identifiers was specifically discussed and we felt that there was no interest in allowing

identifiers to include extensions in a quiet way. Any program using extended characters in identifiers is not portable. I am afraid that the current language about conformance would be in danger because if National characters in identifiers are considered locale-specific, and not implementation-defined behavior, then a conforming program would not be portable!

I have no specific objections to this sort of extension. I believe that we could relax the rules about extensions to syntax (since this would involve changing how a source is tokenized) so that added identifier characters could be accepted without forcing a diagnostic as long as the added identifier characters did not include anything from the defined source character set. (Identifiers including an ! character, for example, would require a diagnostic.)

8. X3J11 has already answered this through an informal request. DEC asked if 'ñ' could be part of an identifier. (They didn't want to ask whether $ could be.) They presented a situation in which a strictly conforming program could distinguish between an implementation that accepted 'ñ' as a letter and one that did not. (See paper 90-015.) The committee voted 23/1/1 that such an extension is valid as long as a diagnostic (warning) is issued if the extension is used. This seems to cover this issue completely.

9. We effectively permit this, although without actually having a national character set compile-time locale as such. However, it should be noted that the multibyte people (the Japanese, in particular) are insistent that this simple solution (which, to them, is to permit 'a few' national characters) is rather insular, and that quite a lot of really interesting variable names are still not possible, or rather, have to be written in katakana or Romanised. I think if people in the West understood quite what we are asking the Japanese to swallow, there would be more sympathy for their position.

10. Some compilers have already permitted letters, including multibyte characters, in identifiers, for several years (known to me for more than three years). I believe that quality standards should permit quality implementations to do so. I have the impression that quality implementations are already permitted to do so, though they might have to issue a warning message. I support the idea that if a user specifies an option to allow additional letters without a warning message, then `__STDC__` should not have to be turned off.

11. What on earth do they propose to do to allow transportation of such programs to ANSI C environments? If the answer is that they are willing to sacrifice portability (and I see no other), then what they are writing is an incompatible extension to ANSI C, and I for one will be sorry to see it. Though I certainly understand the motivation I think worldwide portability is even more important.

12. *[Ed: from Scandinavia]* In general, I believe the use of national characters would be a welcome addition for many non-English speaking programming shops.

13. It would not be difficult to add this support to our product. However, how would the compile-time locale be specified by the user? *[Ed: Perhaps via a compiler option or maybe via conditionally compiled pragma.]*

14. I would have no problem with a designation of some fixed group of characters from the 'upper half' of ISO Latin 1 as 'letters.' I would have severe problems if the set of letters changed from locale to locale, whether by choosing different subsets or, even worse, by using National Replacement Characters. Allowing this would make source code impossibly non-transportable. At least in the case of trigraphs, I could (if I was willing to put up with ugly code) write a program to 'en-trigraph' an arbitrary source file; it could then be compiled everywhere. However, if I want to edit a source file written using some strange NRCS, there is no general way for me to rename variables to avoid problems—even ignoring the impossible problems of modifying just one of a number of modules.

    It is also worth keeping in mind that names known to the system linker might have to be restricted to the current set of letters, just as today they may have to be short and case-insensitive. This will add yet another potential portability problem (since of course some system linkers will accept names with these 'letters').

15. This has also come up in X3J16 in a different context. As a major supplier to companies and universities in Europe we like this idea and it doesn't seem to be too difficult, but maybe that shows my naiveté.

Guidelines for Extensions

Both the ISO C group WG14 and the NCEG are inventing extensions to Standard C. And so too are numerous implementors. What guidelines should they use regarding defining new tokens, headers, etc? Comment on the following and/or add your own categories.

General Comments

1. All new operators, punctuators, and keywords should be compatible with C++, to encourage the sharing of extensions to C and C++.

   No new symbol should be a legal initial prefix of a sequence of ANSI C tokens. For example, `<-` should not be a token, because `a<-b` could no longer be tokenized properly.

   Any new characters (e.g. `@`) should have trigraphs. New operators should not run afoul of trigraphs.

2. Most important is to retain the 'spirit of C.' This is more than just having simple rules for keywords, operators, etc. The extensions should have the look and feel of C and should be natural to use. Needless to say, a conforming program should *always* be accepted, and the extensions must be optional (i.e., can be turned off with a pragma or command-line switch). I have added parallel extensions to our compiler and have thought a lot about their potential impact.

3. New operators or new syntaxes can be anything that does not make any existing construct potentially ambiguous.

4. I do not want new punctuators or operators added at this stage. Here in Japan, NCEG is not yet so popular since the application of C in the numerical area is not big. However, now that DSPs are getting popular the requirement to program in C for DSPs is increasing. This may change my attitude.

5. A language without extensions is a language without users. It is vital to the evolution of C that extensions be implemented. However, it does not serve the future users of the language to make extensions appear ugly, awkward to use, or too different from the base language. In extending a language your first duty is to the integrity of the language. Only second should you try to preserve existing code, even though preserving existing code is still very important. You should spend a lot of time designing an extension, making sure that you can preserve as much integrity and as much code as possible.

   An example of a terrible way to extend C is the `near`/`far` keywords of PC C compilers. These keywords were added around the time that `const` and `volatile` were being added to ANSI C, but instead of using the same syntactic position and analogous semantic interpretations, the keywords have a completely different kind of binding. As a result, all PC C parsers are more complicated and PC C users are more confused than they need to be. The spelling of the keywords is not the problem, but the way in which the syntax was butchered to give the new keywords their significance.

   The only kind of extension that is unpleasant to me is one that changes the behavior of a program silently. It is equivalent to the kind of integer overflow problems one encounters when moving code between a 16-bit and a 32-bit computer. It takes a lot of time to find those sorts of incompatibilities and fix the offending code.

6. I think the source program should `#define` `__STDC_1__` or `__STDC_2__`, etc., to specify which level it wants. An old program wants unsignedness preservation. A current program wants to use various identifiers without worrying about additional future namespace pollution. A new program will want to use new features, and will take responsibility for avoiding new namespace pollution.

7. *[Ed: from a member of X3J16]* Compatibility with C++, in a very broad meaning of the word. Apart from obvious pitfalls like reusing a word reserved in C++, I think it would be desirable to standardize a functionality that could be expressed well both in C and using the more advanced structuring mechanisms of C++. Specifically, one should ensure that a new C library could be nicely encapsulated by one or more C++ classes.

8. The Swiss NRC replaces the underscore ('_') character. Because this is a non-ANSI replacement, the ANSI C Standard does not include a trigraph for '_'. Furthermore, the current specification in the standard does not *allow* an implementor to define a new trigraph. I believe an implementor should be able to define a trigraph for '_' or anything else that is appropriate to the environment in which their product is to be used. However, no one, including myself, is very excited about trigraphs.

9. I am against any new feature that conflicts with C++ or could be done with existing C++ constructs. These things would be very divisive of the C/C++ community.

10. New operators, punctuators, and keywords should be used as though they are a scarce resource and only when an acceptable syntax for the extension cannot be designed without them.

New operators and punctuators

1. I'm not sure where to draw the line between overloading an existing operator and defining a new one. It depends *very* much on the meaning and function. No simple rules for this one.

2. WG14 should not produce any.

3. New tokens or syntax should avoid the kinds of problems that the template syntax has caused in C++. They added `<` and `>` as paired angle brackets, despite the knots it puts in the language. Preferably new operator and punctuator tokens should avoid overloading existing ones.

4. Only if they: 1) match the 'spirit of C' (exponentiation doesn't, for example); 2) are spelled so that all existing conforming programs are unaffected; and 3) (operators) sit well with the rest of C's operators.

5. New operators should be invented only for new data types (e.g., complex numbers).

6. Whenever possible overload existing operator tokens. Do not invent a new operator token unless absolutely necessary. However, there are times when a new token makes the most sense.

7. If you really need a lot of operators, there's little choice but to use variable-like operators such as `sizeof`.

8. *[Ed: from a C++ implementor]* Avoid new operators like the plague. That's what C++ and functions are for!

9. Probably a bad idea, short of full overloading. Extending existing syntax should be enough for the burning issues (extended range integers, for example, can be handled by generalizing bit-field syntax).

   No new operators! C has enough operators. Isn't the language large enough? If there are new operators they should be 'keyword type' so that national character set issues do not come up.

New keywords

1. Keep them short, but not too short. Lowercase only, of course. Possibly prefix them with one or more underscores for things not used so often (e.g., `__asm()`), but preferable not for the 'real' extensions. *[Ed: This is not an unpopular view. Keywords specific to a particular implementation might have such a prefix but keywords such as* `complex` *would not. One hopes to extend the language to include these some day, so let's make them look like 'real' keywords now.]*

2. New keywords should start and end with `__`.

3. New keywords should start with `__`.

4. Moving code into a new compiler system that has extended keywords that conflict with user variables is rarely going to be an insurmountable problem. The extended compiler will complain about every misuse of an identifier, and development tools exist that can substitute one identifier for another throughout a body of source.

5. This is always a sticky problem. The best keywords are those that are less than 8 characters in length, and consist of only lowercase letters. Its too bad that just about every sensible identifier like that has been used in somebody's code somewhere. I really dislike introducing keywords with underbars or some such silliness. I would rather, however, use a good keyword and break a few programs than use a bad keyword and preserve most code.

   Introducing a keyword with bad spelling as an extension for one implementation only creates trouble if the feature is ever to be incorporated into the language as a universal feature. A standard keyword in the future will almost certainly have to have a good spelling.

6. Generally, I am against these unless they can be introduced as 'hidden.' For example, you include the header `xyz.h` and this includes a `#define` of the name `xyz` as a synonym for the hidden `__xyz` keyword.

   On the other hand, new keywords will naturally be added to an evolving language. Since new keywords break programs in a loud, straightforward manner (usually), I will not vote against allowing new keywords in the next version of the standard, but that does *not* give NCEG license to include new, unhidden keywords.

7. This one is difficult. One good approach is to invent a new header that contains something like:

   ```
   #define newname __protected_name
   ```

   This means the name does not affect C code that does not include the header. This is the approach used by NCEG for the new complex types. However, there are times when the only sensible solution is a new keyword, like `fortran` or `asm`. NCEG is also exploring the addition of a new keyword `restrict` that is not protected by a unique header. I'm not convinced that there can be hard and fast rules because it is too subjective.

8. With careful design, you can often avoid reserving new keywords. Few people seem to remember that PL/I, with its long list of keywords (many available in both full and abbreviated form) had *no* reserved words!

New headers vs. extending existing headers

- 8 – Allow enhancement of existing headers
- 9 – Create new headers
- Comments:

  1. If the extension is really essential and it builds upon standard features, add it to existing headers. If it's something completely new and different, use a new header which can be included only when this feature is used.
  2. You should avoid extending standard headers as much as possible (just like the proposed Multibyte Support Extension [MSE]). *[Ed: At the last WG14 meeting in Copenhagen it was agreed that the MSE should come in a separate header. So, rather than augment* `printf` *and the like to handle wide characters, a new version of* `printf` *would be defined, perhaps called* `wcprintf`*.]*
  3. As long as there is a documented way to disable extensions in the standard headers, I see no problem with adding declarations to the standard headers.
  4. Generally, my principle is that new (not currently reserved) names should not be added to existing standard headers whenever there is a reasonable alternative. But, there are circumstances where the only reasonable choice is to clash with the programmer's name space. For

example, it would seem reasonable to me that the next version of ANSI C could include `popen` and `pclose` in `stdio.h` even though they are not currently reserved names.

5. OK to extend existing headers if the extensions concern the same topic (i.e., new I/O related macros should be in `stdio.h`). New header should be added only for new functionality.
6. It is definitely preferable to create new headers. However, there may be cases where extending the existing headers makes more sense. For instance, if there was enough sentiment for an `isodigit` function (is octal digit) then it might make the most sense to extend `ctype.h`.
7. As long as new macros and typedefs are in the implementor's name space I think extensions which obviously 'belong' in an existing header should go there. Where there is doubt, they should be in new headers.
8. The programmer's namespace is polluted enough. Don't change standard headers!
9. I believe restricting things to new headers only will result in a fragmented and confused collection of headers. Adding to headers will cause some backwards incompatibility but, assuming the additions are fairly limited, I believe the cost of updating code would be worth it.

Standard pragmas

- 12 – Pragmas are, by definition, implementation-specific.
- 2 – OK to have standard pragmas.
- Comments:

  1. I prefer to prefix every pragma with a vendor-specific prefix. How about 'standardizing' this as an approach?
  2. There should be no standard pragmas. A pragma has the disadvantage of not being allowed within a macro. This is an arbitrary and foolish limitation on whatever feature pragma will be used for. It is better to define a new keyword or a new syntax that can be used in macros, instead of a pragma.
  3. I think that pragmas have become a safety valve where implementors pile tons of extensions without much care to what they look like. The effect has been to create a 'Tower of Pragma Babel.' You really do have to conditionalize the inclusion of pragmas because there is no reliable method for coding a pragma that is intended for just one or a few implementations.

We cannot completely control how pragmas are implemented now. There is already an enormous amount of variety in the field. It would probably be appropriate to survey the field and publish the de facto specifications of any common pragmas. At least then an implementor can use compatible syntax for similar pragmas.

I am afraid the pragmas are the wild frontier of C, where the law has little to say.

4. There are some things that work best with pragmas. However, there are lots that just cannot be wrestled into the pragma strait jacket. If there are some capabilities that are found to be necessary and fit the pragma model well, I see no reason to disallow their standardization. However, there must be a strong justification, and the syntax used must not be such that it gets in the way of the most common pragma forms.

5. A clearing-house is mostly a good idea. It would be a good idea to hold votes in some reasonable manner, on suggested names and meanings.

6. `#pragma` is too tightly tied to the preprocessor to get standard linguistic semantics out it.

7. As long as pragmas are *not* conformance issues, I see no problem with publishing a 'standard' set—if it can be agreed on.

8. If you want to know whether I think there should be some kind of list of 'widely-implemented' pragmas, I'd say yes. Then again, the whole pragma business was poorly thought out to begin with.

9. I think that standard should have said anything goes with pragmas *except* that pragmas *cannot* change the planned and expected result of a program. For example, a pragma that says a certain optimization is legal because the programmer planned that the code should be optimizable in that fashion and someone reading the code doesn't have to know what the pragma does because removing it will get the same answer (a day later perhaps). On the other hand, a pragma that says all `++` operators really mean increment by 2 instead of 1 (granted, it's a silly example) might result in the program working as planned by the programmer but if someone comes along and removes it the answer will be different. By the way, I recently got a copy of a C benchmark program and it had pragmas all through it. This strikes me as very unfair.

∞

32. ANSI C Interpretations Report

Jim Brodie

Abstract

This is the third in an ongoing series of articles addressing the interpretation activities of X3J11, the standards committee that developed the ANSI Standard for C. In it I discuss a variety of interpretation requests dealing with translation limits, valid expressions for accessing array elements, the compatibility of pointer and array declarations in prototypes, the completion of incomplete types, and the relative priorities of constraint violations and undefined behavior.

In this article I continue to review C Standard interpretation requests being addressed by X3J11. In particular, I discuss some of the requests that were addressed at the September, 1990 meeting of X3J11 in Pleasanton, California.

Translation Limits

One request for interpretation asks about the translation limit related to the minimum nesting that must be supported in a standard-conforming translator. The minimum in question (stated in §2.2.4.1, Translation Limits), is:

> 15 nesting levels of compound statements, iteration control structures, and selection control structures

The writer points out that the term *iteration control structure* is not defined in the Standard. In attempting to determine what was meant, he asks:

> Is it:
>
> 1. a `for` loop header excluding its body, i.e., `for (;;)` or,
> 2. a `for` loop header plus its body, i.e., `for (;;) {}`?
>
> Does it make a difference if the compound statement is a simple statement without {}?

The response from the committee clarifies the issue:

> The committee's opinion was that the term *iteration control structure* is the same as *iteration statement*, which is defined by the standard. As a result, the statement that is the loop body *is* considered part of the *iteration control structure*. Similarly, a *selection control structure* is the same as a *selection statement*.
>
> In discussing nesting levels, the consensus of the committee was that the fragment
>
> ```
> for ( ... )
>         for ( ... )
> ```
>
> contained 2 nesting levels while the fragment
>
> ```
> for ( ... ) {
>         for ( ... )
> ```
>
> contained 3 nesting levels (i.e. the introduction of the { implied another nesting level).

Over the course of the development of the C Standard, there has been considerable debate centered on the Translation Limits section. It has been argued that the basic requirement of this section, which is that

> The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:

didn't give the Standard any "teeth." At best they were "rubber teeth." Some of the specific limits and numbers in this section, such as '509 characters in a logical source line,' were clearly the result of committee compromises.

The requirements of the Translation Limits section give the translator developer considerable leeway in actually establishing the test case that must run to prove compliance. This was, at least in part, a concession to the small environments where memory space limitations make very large programs difficult to handle. It also, quite frankly, was a protection from devious developers of Standards-compliance test suites.

One of the goals of the Translation Limits section was to help establish a translator-developer mind-set where arbitrary limits on translation complexity were avoided. In the end, despite the weak wording of the section, it seems to have accomplished this goal. It is possible to build a translator that meets the Translation Limit criteria, but that does not meet the intent of this section. However, the work to do so is probably just as great as building a usable system that complies with the spirit of the standard. In practice, the Translation Limits section has helped establish limits so that reasonable-sized programs can be written with a level of confidence that they will be acceptable to any conforming translator.

Accessing Array Elements

One of the more interesting requests for interpretation handled at the meeting had to do with the accessing of array elements. The interpretation request can best be understood with the following example. Given the declarations:

```
typedef char row[5];
typedef row matrix[4];
matrix A;
```

is the assignment

```
A[1][7] = 0;
```

valid or will it result in undefined behavior? In particular, is the expression `A[1][7]` valid. The second subscript in the expression clearly exceeds the size limit for the row (that you would access with subscript values 0–4). However, can the programmer portably count on the address, which exceeds the end of the second row, to safely locate an element in the third row?

Note that the use of the `typedef`s is simply to emphasize the multi-dimensional nature of `A`. The same question arises after the more direct declaration

```
char A[4][5];
```

The issue of whether `A[1][7]` is valid is of importance to interpreters that want to diagnose array-bound violations for their customers. Many interpreter developers and their customers feel that the above assignment statement probably indicates a programmer's misunderstanding of the structure of the array. Since this is indicative of an error, they want a diagnostic to warn them.

Others take the position that there is a long history in C of writing expressions that access multi-dimensional arrays as if they were single-dimension arrays. In particular, they point out the long-standing practice of initializing multiply-dimensioned arrays by walking them as if they were long single-dimension arrays.

If the assignment is valid then the interpreters *cannot* issue a diagnostic. (They could, however, issue a non-diagnostic message that warns of this.) If the behavior is undefined, then any translator behavior is acceptable, including issuing a diagnostic or, in the case of a vectorizing compiler, assume no overlap when one actually exists.

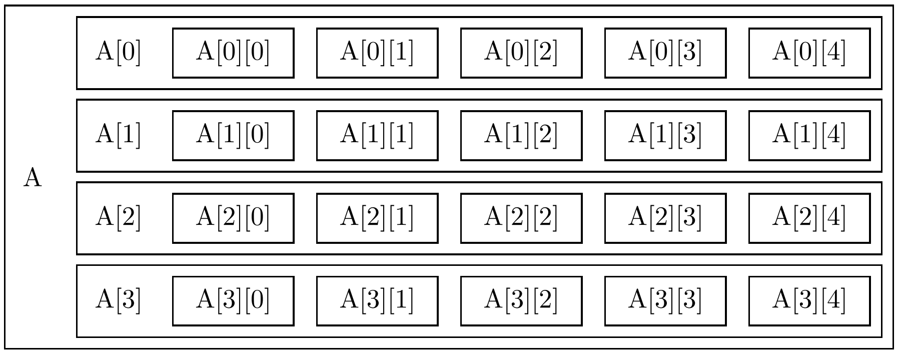
When we try to answer the question of the validity of `A[1][7]`, we start with the fact that array subscripting is actually defined in terms of pointer arithmetic. In §3.3.2.1, Array Subscripting, the subscript operator `[]` is defined so that the expression `E1[E2]` is identical to `(*(E1+(E2)))`.

Perhaps we can determine the validity of the subscript operation by looking at the corresponding pointer expression and the limitations that are placed on it. The Standard establishes limitations on what range of values can be safely generated by arithmetic on a pointer. In the semantics portion of §3.3.6, Additive Operators, a discussion of pointer arithmetic includes the statement:

> If both the pointer operand and the results [of the addition of the pointer and the integer] point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

We've taken a step towards the answer, but now the key questions become "What array object are we referring to in `A[1][7]`?" And, "Is the subscript operation causing a pointer to be generated that points more than one past the end of that array object?"

Lets take a look at the conceptual layout of the array `A` as it was defined above.



In an array such as `A`, we are actually dealing with 3 different types of objects. At the outer level, we have one array object (designated by `A`). It has 4 elements. At the first inner level, we have one of these 4 elements (designated by `A[`*i*`]`). It, in turn, has 5 elements. At the innermost level, we have one of these 5 elements (designated by `A[`*i*`][`*j*`]`). It is an object of type `char`. Therefore, we have 25 objects in all $(1 + 4 + 20)$. As to which of these objects is being referenced in `A[1][7]`, the answer comes from looking at the layout of the arrays and the expressions that access them.

In C, there are no true multi-dimensioned arrays—there are only arrays whose elements may be other arrays (i.e., arrays of arrays). Therefore, when we evaluate the expression, `A[1][7]`, we first evaluate `A[1]`. In the expression `A[1]`, we are referencing the encompassing array and selecting the second element of that array (which is a row-array object). The expression `A[1]` is a pointer to the second row array object in the encompassing array. It has type *pointer to array of 5* `char`. The second computation called for in the evaluation of `A[1][7]` takes this *pointer to row* type pointer and the subscript 7 and calculates the address of the underlying `char` element. It is in this second calculation that we run into trouble with the above limitations, from §3.3.6, Additive

Operators, on the original pointer and the pointer resulting from arithmetic operation. In the second calculation, the object that is being referenced by the pointer is the row, *not* the entire encompassing array. The range of the object is only 5 items long, so an offset of 7 is more than one beyond the end of that object.

In the reply to the request for Interpretation X3J11 wrote:

> For an array of arrays, the permitted pointer arithmetic in §3.3.6, Semantics, [the relevant section is excerpted above] is to be understood by interpreting the use of the *object* as denoting the specific object determined directly by the pointer's type and value, *not* other objects related to that one by contiguity.

Therefore, the result of the computation involving the `[7]` subscript violates the limitation from §3.3.6, Additive Operators, that the original and resulting pointers must point to the same array.

The behavior of the expression is, therefore, undefined and interpreters are free to issue a diagnostic. Since this falls into the category of undefined behavior, translators are free to continue to ascribe whatever meaning they wish, including the obvious meaning of the third element of the next row.

Compatible Prototypes

Another request asked whether the following prototypes are compatible:

```
int f(int a[4]);
int f(int a[5]);
int f(int *a);
```

This question deals with the issue of whether the conversion of array typed parameters to pointers in function prototypes occurs before or after compatibility checks are performed.

If the compatibility checks are performed before the conversions, all three of the above prototypes would be incompatible. This is because the compatibility rule for arrays, as stated in the Semantics portion of §3.5.4.2, Array Declarators:

> For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, they shall have the same value.

In this case, the arrays have different size specifiers.

For pointers the compatibility check is stated in the Semantics portion of §3.5.4.1, Pointer Declarators:

> For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

There is no statement for compatibility between a pointer and an array type.

The automatic conversion, in the function parameter context, of a declaration with an array type into a declaration with a pointer type is defined in §3.7.1, Function Definitions:

> A declaration of a parameter as *array of type* shall be adjusted to *pointer to type*.

The paragraph that addresses the issue of compatibility of parameter lists in function declarations is in §3.5.4.3, Function Declarators (Including Prototypes). It starts with:

> For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of ellipsis terminator; corresponding parameters shall have compatible types.

The paragraph goes on to describe several special cases where old style function declarations (i.e., those without argument type information) are encountered. The paragraph then ends with the parenthetical statement that describes the timing for the parameter conversion rules.

> (For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in §3.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

X3J11 felt it was clear that this parenthetical comment applied to the entire paragraph (including the case where complete prototypes are provided). Since the statement clearly states that the conversions are applied before the comparison takes place, the committee took the position that the above three prototype declarations are compatible.

Remember that the conversion of array-typed declarations to pointer-typed declarations is a special feature of declarations that occur in function parameter lists.

If the following three declarations were found in a file, at file scope

```
    int a[4];
    int a[5];
    int *a;
```

they would be considered incompatible. In §3.1.2.6, Compatible Type and Composite Type, it is stated that:

> All declarations that refer to the same object or function shall have compatible type, otherwise, the behavior is undefined.

Since the behavior is undefined in this case, no diagnostics need to be generated by the translator.

Incomplete Structure Types

The next interpretation deals with incomplete structures. The specific question is whether the following translation unit is valid:

```
struct foo x;
struct foo {int i;};
```

In particular, the requestor wanted to know if the first declaration, that leaves the contents of `struct foo` undefined and declares an object with that incomplete type, was valid.

The committee's response is that the construct is completely valid.

This falls into the same category as incomplete array types, which are probably more common. For example, in

```
int arr[];
int arr[17];
```

the first declaration declares `arr` as an array of unknown size. The second declaration then completes the information on the type of `arr`. The type of `arr` goes from being an incomplete type to being an object type.

In a similar way, the second structure declaration in the interpretation request

```
struct foo {int i;};
```

provides the information to complete the incomplete structure type associated with the tag `foo` and (indirectly) the object `x`. This is described in §3.1.2.5, Types:

> ... A structure or union type of unknown content (as described in §3.6.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

Typedefs and Incomplete Types

Another somewhat related question was asked. Given the following:

```
typedef int table[];
table one = {1};
table two = {1, 2};
```

the writer asked several questions. First, he asks "Is the `typedef` of an incomplete type valid?" He then asks "How is the incomplete type completed given the two differing declarations that follow? Is the type completed to have a size of 1 or a size of 2?"

To answer the question of incomplete types in `typedef`s, we can look at the definitions for typedefs in §3.5.6, Type Definitions:

> In a declaration whose storage-class specifier is `typedef`, each declarator defines an identifier to be a typedef name that specifies the type specified for the identifier in the way described in §3.5.4. A typedef declaration does not introduce a new type, only a synonym for the type so specified...

Since there is no limitation on the kinds of types that can be specified, typedefs that specify an incomplete type (which is a perfectly valid kind of type) must be allowed.

This leads us to the second question about how the incomplete type is completed. The answer to this question is found in §3.1.2.5, Types:

> An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage).

The emphasis here is on the clause "for an identifier of that type." This means that the incomplete type is completed for the individual names. The name `one` has an array type with size 1 (its type is completed based upon the initializer). The name `two` has an array type with size 2 (again based on the initializer). The typedef name `table` remains an incomplete type (it still has no size associated with it). Therefore an expression such as `sizeof(table)` is invalid, even after the declarations of the arrays `one` and `two`, since you cannot take the size of an incomplete type.

Constraints and Undefined Behavior

The next requests cites the following two references.

In §3.1.2.6, Compatible Type and Composite Type, the following statement is made:

> All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

In the Constraint section of §3.5, Declarations, it is stated:

> All declarations in the same scope that refer to the same object or function shall specify compatible types.

The writer then asks:

> The constructs covered by these sentences overlap. The latter is a constraint while the former is undefined behavior. In the overlapping case who wins?

After some discussion X3J11 decided that the constraint wins. In its response, the committee references the following paragraph from §2.1.1.4, Diagnostics:

> A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint.

The committee response notes:

> When a construct violates a constraint, §2.1.1.3 page 7, lines 15–17, requires a diagnostic, with no exceptions described. That a particular construct might also be in the category of undefined behavior does not release a conforming implementation from issuing a diagnostic.

This interpretation can be generalized to handle any case where a general statement holds that a group of constructs are undefined, but in certain specialized cases, within this broader scope, a specific diagnostic must be generated.

Jim Brodie is the convenor 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.

∞

33. Initializers and Finalizers: A Proposed C Extension

Jerrold Leichter
LRW Systems
and
Columbia University

Abstract

A running program goes through a number of predictable operational phases. Often, it is important to a programmer to gain control of execution at the transitions between these phases.

The most elementary transitions, common to all programs, are startup and shutdown. We discuss the fact that C currently does not provide the programmer with adequate techniques for gaining control at these transitions, and propose extensions to provide the necessary mechanisms.

Introduction

A C program begins execution at its `main` function. Before `main` begins executing, standard input, output, and error files are opened. After all user code terminates, whether by a return from `main` or a call to `exit`, any remaining data buffered for these (or other) files is flushed and the files are properly closed.

A familiar interface. And yet there is a subtlety hidden here which can be seen immediately if one considers duplicating this interface within the standard hosted C environment. No explicit calls are coded into `main` to open the standard files, yet there they are. Similarly, no explicit calls flush and close the standard files.

Many mechanisms can be used to provide such an interface. All have limitations and costs that the standard libraries avoid. We propose that the C language be extended to provide a better interface.

Why Have Initializers and Finalizers?

Suppose we wish to write a natural-language database package. The following template outlines such a package:

```
/*
 * Simple database package.  Environment variable DB_LANGUAGE
 * controls how various character combinations are treated.
 */

DB_setup()
{       static language_def;

        language = getenv("DB_LANGUAGE");
        ... initialize language_def ...
}

DB_open(...)        { ... }
DB_close(...)       { ... }
DB_lookup(...)      { ... }
DB_getnext(...)     { ... }
DB_update(...)      { ... }

Other routines ...

DB_finish()
{
        for each open database file
                DB_close(file);
}
```

The `DB_getnext` function is intended to access the next record in alphabetical order. ‘Alphabetical order’ depends on the particular language—each has its own rules for dealing with accented letters and even some pairs of letters. The appropriate rules are set up by the `DB_setup` function. In order to ensure consistency, it is essential that all opened databases be properly closed. The `DB_finish` function ensures this by calling `DB_close` for each open database.

We call a function like `DB_setup`, which must be called before any other function in its package, an *initializer*. Conversely, we call a function like `DB_finish`, which must be called after all usage of the database is complete, a *finalizer*.

Context For the Problem

If our database package were part of a single program written by a small group of programmers, initialization and finalization would not be a major issue. The program would call `DB_setup` early on in `main`, and would make sure to call `DB_finish` before exiting. Many programs today, however, are written under very different circumstances. They are written by large numbers of programmers and incorporate standardized modules. Suppose that the database package were part of a pre-written library. It would then no longer be possible to just assume that the appropriate calls to the initializers and finalizers would be present. Rather, if access to the database is to be reliable, the module's initializers and finalizers, and the rules for when they are to be called, would have to become part of the module's documented interface. There are several problems with this obvious approach, however:

- It makes the interface larger, and increases the degree of coupling across the interface. Use of the module becomes more complex—there is more to learn about it, and there are additional ways in which the client of the module can cause it to fail.
- The *nature* of the additional entries in the interface is inappropriate. The initializers and finalizers are internal to the module, and have no particular meaning to the client of the module. They make visible to the client an unpleasant bit of internal mechanism that it would better for the module to keep private.
- Client-invoked initializers and finalizers cannot be added gracefully to an existing interface. Suppose that an earlier version of the database module supported only English databases. It would thus have no need to calculate language-specific ordering rules, and presumably no need for an initializer. Clients coded to the original interface would include no call to an initializer. Now we add support for additional languages, and with it the need for such a call. Suddenly, all existing code needs to be changed, at an unacceptable cost if the module has been popular and is widely used.

So, we approach the issue of initializers from the point of view of developers and users of modules. We assume that modules are widely used, so that incompatible changes—changes that require changes to existing clients—are very costly. But *compatible* changes are a fact of life. Conversely, we assume that any client of a module is likely a client of any number of other modules, and that we as developers of one module can know little about either our clients or any other modules they may use.

Given these assumptions, we can list a number of goals for an initialization and finalization facility:

1. Each module can declare its own initializers and finalizers;
2. Separate modules need not be aware of each others' initializers and finalizers. In particular, the client need not be aware of any module's initializers or finalizers.
3. The overhead for use of initializers and finalizers, especially for modules that don't need them, should be minimal.

Goals 1 and 2 simply state that the facility should be consistent with the module structure defined by the language. Goal 3 should be viewed in relative terms—the cost of the facility must be consistent with the cost of other things in the language. Since C is a low-level language, designed for the production of efficient code, an initializer and finalizer facility for use with it must be very efficient.

Implementation Techniques for Initializers

There are several ways to implement initializers.

Client-called Entries
This is the approach we discussed and rejected in the previous section. While it satisfies goals 1 and 3, it does poorly on goal 2.

Special-case Code
This is the approach taken by most language run-time libraries. For example, in most UNIX implementations, the *cc* command will link executables in such a way that the actual entry point will be not at `main`, but in the support code, which will initialize the process and then call `main`. Other compilers—VAX C on VMS, for example—generate special code for a function called `main`. C++, which provides its own powerful initializer facility, uses a combination of techniques to allow its own run-time code to gain control at the beginning of execution, so that it can in turn invoke the programmer's initialization code.

This approach can satisfy goals 2 and 3, but in and of itself—that is, without additional linguistic and support mechanisms such as those in C++—cannot deal with goal 1.

Per-Entry Initialization
In this approach, each entry point accessible to a client is modified to call a one-time initialization routine. Logically, the initialization routine itself checks to see whether it has been called before and simply returns if so. In practice, it is generally worth it to avoid the extra function call overhead by testing the 'already initialized' flag before making the call, resulting in functions of the following form:

```
DB_open( ... )
{
        if (!initialized)
                DB_setup();
        ...
}
```

This approach is probably the most widely used. It satisfies goals 1 and 2 easily. Often, it can also satisfy goal 3. This is not always the case, however. Many compilers today can in-line expand function calls. This makes it practical to use a function call interface for operations that do very little computation—simply providing access to a module-private value, for example. Adding a test to every such call can have a significant effect on its cost. It will certainly significantly increase the size of the in-line expanded code for simple functions.

Beyond the possibly minor cost, this approach can be a rich source of maintenance bugs. Most modules have internal functions not visible to clients. Since such code could only be reached after a call to a client-accessible routine, it is common to leave off the initialization test, since it is certain to fail. However, as modules evolve, it's common for once-internal routines to be made visible to clients. Should this happen without the initialization check code being added, a subtle bug will have been introduced into the code. It will only be triggered should the newly-visible routine be the first one called.

Implementation Techniques for Finalizers

While per-entry initialization is not ideal, it at least does the job reasonably well, and can be implemented without any special support from the compiler or run-time code. No analogous technique exists for finalization. The only way that the transition to 'program completion' takes place is by a return from `main` or a call to `exit`, both of which result in execution of code over which neither client nor module author has any control. So the only viable approach without special support is through client-called finalization entries. In the case of initializers, we at least know that there is only one place—at the top of `main`—where the appropriate calls must be placed. Termination, however, can occur in many different places within the code. It is extremely rare to see code that reliably makes explicit calls to finalizers, even in the face of errors. This is particularly unfortunate since robust code often needs a reliable way to do finalization. It must flush buffers, close out files, restore terminals to their initial settings, and generally clean up after itself.

Since user-written techniques are unavailable and few languages have chosen to provide appropriate support, some operating systems have taken the task onto themselves. For example, both DEC's VMS and Apollo's Aegis provide what they call 'exit handlers,' routines that the operating system will call upon termination of the image. Unfortunately, such extra-linguistic mechanisms are not portable.[4]

The ANSI C committee also recognized the finalization problem and defined the `atexit` library function. `atexit` supports a simple version of exit handlers. Unfortunately, it was specified to support only a fixed (and relatively small) number of registered finalizers. This makes the use of `atexit` by a module in-

[4]VMS also provides a little-known initializer facility through the `LIB$INITIALIZE` psect. We are unaware of any other operating system initialization facility, although some probably exist.

tended for widespread use problematical—modules that use up sparse resources are not good neighbors. (Note that while it is perfectly possible to register a single finalizer with `atexit` and then have it in turn register any number of additional routines that it will call, this doesn't solve the problem: We can't guarantee that other modules that need finalizers will use our special registration procedure rather than calling `atexit` themselves. It's also worth noting that this problem could have been avoided by examining existing art. Rather than requiring `atexit` to provide the necessary memory, let the caller create, whether as a `static` variable or in `malloc`ed memory, a defined `struct` with sufficient space for a pointer to the finalization function and an additional object pointer to be used by `atexit` to maintain a linked list of finalizers. VMS does things this way, for example.)

It's interesting to note in passing that a finalization facility based on an explicit call, such as `atexit`, immediately creates its own need for an initialization facility. After all, the call to `atexit` must be made exactly once as part of package startup. Where is that call to be placed? Obviously, in an initializer!

An Alternative Approach

As an experiment, we implemented an initialization and finalization facility as part of the DECUS C[5] system a number of years ago. At the time, DECUS C already provided a simple finalization facility. The run-time system contained, within its 'end of execution' code, a call to a function named `wrapup`. The system library contained a dummy version of `wrapup`, which simply returned. A program could provide its own function which the linker would use in preference to the one in the system library. In the interests of compatibility, we retained support for `wrapup`. As far as we are aware, this facility remain in the DECUS C implementation. The following is extracted from the DECUS C documentation. It defines this facility as it is seen by a programmer:

Name: `initia.h` – Specify Initializers and Finalizers

```
#include <initia.h>

INITIAL
{ initial-code-block };
FINAL
{ final-code-block };
```

Description: The macros defined in this module provide a facility for module-specific initialization and finalization code. The code in the *initial-code-block* is called as a normal C function before `main()` is called; the *final-code-block* is called on exit, just after `wrapup()`.

[5]DECUS is the international DEC Users Society and DECUS C is a popular PDP-11-based C compiler made available to the public through the DECUS library.

Neither call passes any arguments.

Any number of modules—separately compiled files—in an image may independently declare initializers or finalizers; all of them will be called at startup or exit. However, it is impossible to predict what order the calls will be made in, and programs should not rely on any particular ordering.

A typical use of initializers and finalizers is the following: Suppose you have a package that supports access to some on-disk data base. The file containing the data base must be opened before any operations on it can take place, and it should be closed when the program is finished.

The solution using these macros is straightforward. The defining module includes the calls:

```
INITIAL
{ open-the-data-base };

FINAL
{ flush-buffers-and-close-the-data-base };
```

The DECUS C Implementation

The implementation of our initializer and finalizer facility is simple. The output of the DECUS C compiler is linked by one of two programs, the RT-11 linker or the RSX task builder, both of which support the ability to declare large numbers of *psects*, or program sections. A psect can be viewed as a separate link-time memory allocation domain. Compilers place code and data in psects, beginning at address 0 relative to the psect. The linker or task builder collects all code or data placed in each psect and then places all psects within the address space of the program. Psects come in two forms. The form of interest to us is the *concatenated* psect, used for code and global data. Such psects are built by concatenating the contributions from various inputs to the linking program.

DECUS C contains an extension to the C language that allows the programmer to control which psects the compiler will use. The `INITIAL` and `FINAL` macros expand to two things: the names of `static` functions, whose bodies are the corresponding `INITIAL` or `FINAL` blocks; and declarations for pointer variables in a pair of special psects, initialized to the addresses of these `static` functions. We added code to the C run-time startup code that scans through the psect containing initializer addresses, calling each routine pointed to. Similarly, the C run-time exit code scans the list of finalizer addresses. Note that no code need be inserted into the code the programmer has provided. The overhead for this facility is minimal in both time and space.

In fact, almost no code had to be inserted into the C run-time library to support this facility either. The C run-time code already contained calls to routines that played essentially the role of built-in initializers and finalizers. Our implementation simply generalized this code somewhat. As a result, the built-in code is called using the same mechanism as programmer's initializers and finalizers. Careful control of the order of evaluation guarantees that the system initializers are called *before* the programmer's, while the system finalizers are called *after* the programmer's. `wrapup` is implicitly declared as the first programmer finalizer.

This implementation easily meets all three of our goals.

Discussion

To our three goals, it is possible to add two others. The order in which initializers and finalizers are called should be specifiable. And initializers and finalizers should only be called if their module is actually used. Our implementation satisfies neither. (We don't believe they are particularly important, however.)

Specifiable Order

The main reason that a programmer might want to specify the order in which initializers and finalizers are invoked is that some modules depend on others *within their own initializers and finalizers.* For example, we require that the standard I/O functions be available to initializers and finalizers, thus implicitly requiring that their initialization be completed before the programmer's initializers are invoked, and their finalization after programmer finalizers complete. This is fine as a special case, but in general it's a morass. The dependency graph among modules may be arbitrarily complex. In fact, it may contain loops, in which case *no* workable ordering exists. Any attempt at a general solution is likely to be complex, and we question the utility gained. Perhaps the simplest approach would be give initializer and finalizer blocks explicit names, so that they could be called explicitly from other initializers and finalizers should the need arise.

Execution Only On Use

Per-entry initialization has the interesting property that if the client never calls any entry in a module during a particular run, the module's initializer will never run. If the module sets up its finalizers in its initializers, they will not run in such circumstances either. For some programs, this can be a benefit. Balanced against it, however, is a cost. Since initialization may not take place until some arbitrary point during program execution, any errors during initialization will also be delayed until later. This may make the program harder to debug, and can introduce unexpected dependencies. An interesting example can be seen in many implementations—including both UNIX and VMS—of the standard I/O package. While much initialization takes place at program startup, some is delayed until the first operation on a file. This includes, in particular, allocation of memory for buffers. Programs that require tight control of memory allocation can run into problems if the standard I/O package allocates memory at some unexpected time.

Execution only on use does have its advantages, and some other languages provide direct support for it. Mesa, for example, will invoke user initialization code at the first call to any entry point in a package. The implementation uses an 'initialized already' flag. However, it is the compiler's responsibility to insert the appropriate tests and calls.

Jerrold Leichter recently received his doctorate from Yale University. He is the founder of LRW Systems, a company developing state-of-the-art software for distributed and parallel processing. He can be reached electronically as leichter@lrw.com*; at 24 Old Orchard Lane, Stamford, CT 06903; or by phoning (203) 329 0921.*

∞

34. Cray C and Fortran Interlanguage Communication

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

Abstract

This paper describes the interlanguage programming conventions between C and Fortran for programs executing on Cray Research, Inc. computer systems. The emphasis is on recognizing a common environment that minimizes the need for extensions. The execution environment, interlanguage data mapping, and function calling sequences are examined to identify those areas that are common enough to avoid explicit interfaces, and to identify where an explicit interface mechanism is required. This examination led Cray Research to choose their Fortran conventions as the interlanguage communication conventions. C and Fortran can easily communicate with each other using these conventions.

Introduction

There is no industry-wide standard for interlanguage communication. The absence of such a standard is an impediment to transferring data and flow of control between separately compiled modules written in different languages.

This paper focuses on interlanguage communication between Cray Research's Fortran compiler (CFT77) and Standard C Compiler (SCC), the primary compilers used on Cray Research computer systems. There is also some discussion about the requirements imposed by the ANSI standards for Fortran-77 and C.

The solutions offered here are intended to provide insight into the general problems that multilingual implementations face. And even though the process of standardizing interlanguage communication is very desirable, it is probably not attainable.

Throughout, I generally use upper-case to represent Fortran names and keywords and lower-case to represent C names and keywords.

The Environment

A comparison of SCC and CFT77 shows that there is an intersection of the two languages where interlanguage communication is relatively straightforward.

Both languages represent external names in a straightforward fashion. There is no attempt to prepend or append special characters (like `$` or `_`) to external names.

Both languages pass arguments in the order that they are specified, in a *calling list* that is on the stack. A header word in the calling list provides the number of words in the calling list and the source line number of the call site. The prologue and epilogue sequences are the same between the two languages. The calling sequence is also the same with the calling routine preserving some of the current invocation environment, and the called routine allocating stack space and saving the rest of the calling routine's environment. Function return values are:

- Copied to a memory location specified by a special argument if the return type is `struct`, `union`, or `CHARACTER`
- Returned in two 64-bit registers if the return type is `float complex`, `double complex`, `long double`, `COMPLEX`, or `DOUBLE PRECISION`
- Returned in a single 64-bit register otherwise

This common environment constitutes a large part of the interlanguage communication conventions. Another part of the intersection is the language-independent data types where there is a straightforward mapping from one language to the other.

The languages diverge, however, because of language-specific data types, different global name spaces, and language-specific calling conventions. A close examination of these areas resulted in our choosing the CFT77 calling conventions as the *interlanguage conventions*. This puts the burden of communication on the C programmer to conform to the Fortran conventions. Furthermore, it is never necessary to modify any Fortran source code to communicate between the two languages. The following guidelines should be kept in mind when attempting communication between SCC and CFT77:

- Fortran names are upper-case while C names are mixed case.
- Fortran uses the call-by-reference convention while C uses call-by-value.
- Multidimensional Fortran arrays are stored column-wise, while multidimensional C arrays are stored row-wise.
- Fortran data declared to be type `CHARACTER` is incompatible with C data declared to be either `char` or `char *`.
- Fortran data declared to be type `LOGICAL` is not necessarily compatible with C's values for true and false.

The primary goal of the Cray Research interlanguage communication conventions is to address the issues outlined here.

Language-Independent Data Types

The following data types are considered to be language-independent:

- 64-bit signed integers
- Single-precision floating-point numbers
- Double-precision floating-point numbers
- Complex numbers
- Single dimensioned arrays of the aforementioned types
- Addresses of word-aligned data
- Addresses of functions and procedures

It is straightforward to pass arguments and return scalar values that have these language-independent types. The scalar types map exactly, while the single dimensioned arrays differ only in their bases. C provides only zero-based arrays. Fortran arrays, on the other hand, are one-based by default but can be declared to have any base. For example, the following two declarations are equivalent.

```
INTEGER A(0:9)          int A[10]
```

Another issue raised by these conventions is compatible data mapping between the languages. For example, the mapping between Fortran and C types is as follows:

Fortran	*C*
`INTEGER`	`int`, `long`
`REAL`	`float`, `double`
`DOUBLE PRECISION`	`long double`
`COMPLEX`	`float complex`, `double complex`
`POINTER`	all word-aligned pointers

Clearly other implementations have chosen different interlanguage data mappings. The data mappings specified here were chosen because they mapped very nicely onto the underlying machine architectures. Other implementations make different decisions for similar reasons. For instance, on many implementations it is natural to map type `float` onto type `REAL`, and type `double` onto type `DOUBLE PRECISION`. This is one of the issues that makes it very difficult to standardize interlanguage communication.

SCC supports two complex types that map onto the same underlying representation. These are additional arithmetic types that are not part of the

ANSI C standard. (As far as I am aware, SCC is the only C compiler that supports such an extension.)

Fortran-77 allows formal parameters to be declared as variable-length arrays (*dummy arrays*). This is part of the ANSI standard. The CFT77 implementation also allows local arrays to be declared as variable-length arrays. These are called *automatic arrays*. Although neither of these features is part of the ANSI C standard, we have extended SCC to include both of these features. Our implementation is based largely on the GNU C implementation. Therefore, single-dimensioned variable-length arrays are considered to be language-independent data types.

The CFT77 implementation supports a `POINTER` type that is a Cray Research extension to the Fortran-77 standard. Although this is important for our needs, it is a questionable candidate for standardization.

Language-Dependent Data Types

Some data types require an implicit or explicit interface to convert their format to the format required by the interlanguage conventions. The following are the *language-dependent* data types: character, logical, and multidimensional arrays.

Fortran `CHARACTER` addresses contain two pieces of information, a character address and a length. For this reason they are often referred to as *character descriptors*. C character pointers just contain a character address and, by convention, the character string is terminated with a zero byte. Interface functions are provided that convert one to the other. Due to the underlying implementation, it is possible to pass arrays of character strings between CFT77 and SCC, although not in an obvious way. (An example at the end of this paper shows the actual method.)

CFT77's `LOGICAL` values are not necessarily compatible with C's notion of true and false. Because Fortran compilers were developed on our architectures before any C compilers, different values were chosen for true and false. Fortran-77 does not specify the exact values for `.TRUE.` and `.FALSE.` in contrast to C which dictates that false is zero and true in nonzero. New architectures allow us to eliminate this difference, but as long as the original architecture is supported this difference will exist. Again interface functions are needed to convert one to the other.

The interlanguage communication rule for multidimensional arrays is easier to state than to work with. Several of our library functions operate on multidimensional arrays and expect Fortran column-major order. Therefore, it is necessary to reverse the order of the dimensions in the array declaration and subscript accordingly. For example, a Fortran array declared with:

```
      INTEGER A(20,0:10)
```

could be declared in a C routine as:

```
int A[11][20];
```

and an element from this array that is referenced in Fortran as:

```
A(2,7)
```

is referenced as:

```
A[7][1];
```

in the C routine. Unfortunately, reversing the dimensions is not always intuitive.

Multidimensional variable-length arrays are also considered to be language-dependent data types with the only concern being row-major versus column-major order.

C-Specific Data Types

There are C-specific data types that do not have any interlanguage mapping at all. This is primarily due to the rich typing mechanism present in C. The following C-specific data types do not have a corresponding type in CFT77.

- All unsigned types
- All `char` types
- All `short` types
- All structure types
- All union types
- Arrays of the aforementioned types
- Arrays of word-aligned pointers
- Arrays of function pointers

Fortunately, it is easy to convert most values of C-specific scalar data types into the same values of language-independent data types. Arrays of pointers do not exist in CFT77 and are best avoided altogether when attempting interlanguage communication. Structures have no analogy in CFT77, however, many global structures can be mapped onto CFT77 `COMMON` blocks as described in the next section.

Interlanguage Names

Our convention requires interlanguage names to start with an upper-case letter followed by upper-case letters, digits, or underscores. A maximum of 31 characters is permitted in an interlanguage name. The Fortran-77 standard limits names to 6 alphanumeric characters. However, the CFT77 implementation has an extension that allows names up to 31 characters with embedded and trailing underscores. The ANSI C standard defines portable external names as being unique within the first 6 characters, single case, and no leading underscore. SCC supports mixed-case names up to 255 characters. The interlanguage conventions, therefore, require names to be 31 characters or less, and all letters must be upper-case.

It is possible to share global data by placing it in `COMMON` blocks. In fact, SCC stores externally defined data in such blocks. Therefore, the following simple declarations will map onto the same global memory locations:

```
COMMON /I/ I            int I;
```

The following is a more intricate example that demonstrates how to map SCC external definitions onto CFT77 `COMMON` blocks.

```
        PROGRAM F

        COMMON /GLOB/ M1, A(100), N(10,20)

        DO 10 I = 1, 100
  10    A(I) = I

        M1 = 7
        DO 30 J=1, 20
           DO 20 I = 1, 10
              N(I,J) = I+J-2
  20       CONTINUE
  30    CONTINUE
        CALL CFUN

        STOP
        END
```

```
#include <stdio.h>

struct {
        int m1;          /* member names need */
        double a[100];   /* not match the     */
        int n[20][10];   /* COMMON names      */
} GLOB;
```

```
void CFUN() {
        int i, j;

        printf("m1 = %d\n", GLOB.m1);
        for (i = 0; i < 100; i += 25)
                printf("a[%d]=%4.1f ", i, GLOB.a[i]);
        putchar ('\n'); putchar ('\n');

        printf("n:\n   ");
        for (j = 0; j < 10; j += 2)
                printf("%4d", j);
        printf("\n   --------------------\n");

        for (i = 0; i < 20; i += 4) {
                printf("%2d: ", i);
                for (j = 0; j < 10; j += 2)
                        printf("%3d ", GLOB.n[i][j]);
                putchar('\n');
        }
}
```

The output produced by compiling and linking these modules together is:

```
m1 = 7
a[0]= 1.0 a[25]=26.0 a[50]=51.0 a[75]=76.0

n:
       0   2   4   6   8
   --------------------
 0:   0   2   4   6   8
 4:   4   6   8  10  12
 8:   8  10  12  14  16
12:  12  14  16  18  20
16:  16  18  20  22  24
```

The external linkage of global data that is described in the ANSI C standard is often called the 'ref/def' model. Essentially this describes a model such that, for each external name, there is one definition (an entry point name that is exported) and possibly multiple references to the definition (a name that is imported). The SCC implementation is compatible with this model, but also extends it by allowing multiple compatible definitions.

Interlanguage Function Invocation

The interlanguage conventions for invoking a subroutine or function are again the CFT77 calling conventions. When C calls a Fortran subprogram:

- The name of the subprogram must be an interlanguage name (<= 31 characters, upper-case)
- All actual arguments must be addresses (arrays are considered to be addresses)
- Language-dependent data must be converted to the correct format
- Function return values must be language-independent scalar data types

The following examples demonstrate these points:

```
double a[100];

main() {
        double x = 2.3;
        extern double F4(); /* interlanguage name F4 */
        int i;

        x = F4(&x, a);  /* must pass address of ‘x’    */
                        /* interlanguage type returned */
        for (i = 0; i < 100; i += 25)
                printf("a[%d]=%4.1f ", i, a[i]);
        printf("\nx = %4.1f\n", x);
}

        FUNCTION F4(X, A)

        REAL A(100)

        DO 10 J=1,100
  10       A(J) = J + X
        F4 = 4.5      ! returns interlanguage type

        RETURN
        END
```

The output produced by this example is:

```
a[0]= 3.3 a[25]=28.3 a[50]=53.3 a[75]=78.3
x =  4.5
```

When Fortran calls a C function:

- the name of the C function must be an interlanguage name (<= 31 characters, upper-case)
- All formal parameters must be declared to be pointers
- Language-dependent data must be converted to the correct format by the C function
- Function return values must be language-independent scalar data types

The following example demonstrates these points:

```
        PROGRAM JOE

        X = 2.3
        I = 7
        CALL CFUN(X, I)
        PRINT *, 'X = ', X, '  I = ', I

        STOP
        END

void CFUN(double *px, int *pi) {

        *px += 7.12;
        *pi += 30;
        return;
}
```

The output produced by this example is:

```
X = 9.42  I = 37
```

fortran.h

The SCC implementation defines a header, `fortran.h` that is used for interlanguage communication. This header defines one type and several macros. The type is `_fcd` which corresponds to the type of a CFT77 character descriptor. This type can be passed to a Fortran function that expects a `CHARACTER` argument.

The macros are:

`_cptofcd` – merges a C character pointer and a length into a Fortran character descriptor

`_fcdtocp` – extracts a C character pointer from a Fortran character descriptor

`_fcdlen` – extracts the byte length from a Fortran character descriptor `_btol` – converts a C integer into a Fortran `LOGICAL`

`_ltob` – converts a Fortran `LOGICAL` to a 1 or 0

Prototypes for the contained functions are:

```
_fcd _cptofcd(char *ccp, unsigned len);
char *_fcdtocp(_fcd fcd);
unsigned _fcdlen(fcd);
long _btol(long boolean);
long _ltob(long *logical);
```

The following example uses these interface functions:

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

main() {
        long x;
        long *logical = &x;
        long bool;

        BOB(_cptofcd("abcdef", 6), logical);
        bool = _ltob(logical);
        printf(" bool is %d\n", bool);
}

void CFUN(_fcd fcd) {
        char *cp = _fcdtocp(fcd);
        unsigned len = _fcdlen(fcd);

        cp[len-1] = 0;  /* truncate last character */
        printf(" len = %u  cp is %s\n", len, cp);
}
        SUBROUTINE BOB(CHR, LOGIC)

        CHARACTER * (*) CHR
        LOGICAL LOGIC

        PRINT *, 'IN BOB: ', CHR
        CHR = 'XYZABC'
        LOGIC = .TRUE.
        CALL CFUN(CHR)
        RETURN

        END
```

The output produced by this example is:

```
IN BOB: abcdef
len = 6  cp is XYZAB
bool is 1
```

Thus far, no new syntax has been described for either CFT77 or SCC to support interlanguage communication. The machinery described here is sufficient to perform all of the interlanguage communication available on Cray Research computer systems. However, there is one new keyword, `fortran`, that makes interlanguage communication easier. The semantics of this keyword are: the name of the called subprogram is converted to upper-case (if necessary), and all parameters that are not addresses are implicitly converted to addresses. Conversion includes placing expressions that are not lvalues into compiler-generated temporaries, and then passing that address.

The following example is identical to that in the Section 'Interlanguage Function Invocation,' except that the `fortran` keyword is used.

```
double a[100];

main() {
        double x 2.3;
        fortran double f4();   /* new keyword */
        int i;

        x = f4(x, a);   /* name converted to F4  */
                        /* &x is implicitly done */
        for (i = 0; i < 100; i += 25)
                printf("a[%d]=%4.1f, ", i, a[i]);
        printf("\nx = %4.1f\n", x);
}

        FUNCTION F4(X, A)

        REAL A(100)

        DO 10 J=1,100
  10       A(J) = J + X
        F4 = 4.5      ! returns interlanguage type
        RETURN

        END
```

Finally, the next example shows how to pass an array of character data to a Fortran function. Both the C and Fortran-77 standard require the characters in character arrays to be stored contiguously. This allows single-dimensioned arrays of characters to be viewed as multidimensioned arrays. The following C function converts the address of a single dimensioned array of 9 characters into a Fortran character descriptor with a length of 3. Since the Fortran function `JOE` declares its first parameter to be an *array of* `N` *assumed-size characters*, the correct value for `N` is 3. Next, it converts the address of the first element of a two-dimensional array of characters into a character descriptor with a length of 3. This time `JOE` is called with `N` having the value 2.

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

extern JOE();

main() {
        int i;
        char ac[3][3] = {"uvw", "xyz", ""};
        char *cp = "abcdefghi";   /* 9 chars */
        int j = 3;
        _fcd afcd;   /* fortran character descriptor */

        afcd = _cptofcd(cp, j);   /* CHARACTER *3 */

        printf(" j = %d  cp = %s *ac = %s\n", j, cp, *ac);

        i = JOE(afcd, &j);        /* CHARACTER *3,  3 */

        afcd = _cptofcd(*ac, j);  /* CHARACTER *3 */

        i = JOE(afcd, &i);        /* CHARACTER *3,  2 */

        printf(" i = %d\n", i);
}

        FUNCTION JOE(ASTR, N)

        CHARACTER * (*) ASTR (N) ! array of assumed size

        PRINT *, 'N = ', N       ! N is 3
        DO 10 I=1,N
 10        PRINT *, '<', ASTR(I), '>'
        JOE = N-1
        RETURN

        END
```

The output produced by this interlanguage program is:

```
j = 3  cp = abcdefghi *ac = uvwxyz
N = 3
<abc>
<def>
<ghi>
N = 2
<uvw>
<xyz>
i = 1
```

Conclusions

Extensive support exists for interlanguage communication by using the CFT77 conventions as the interlanguage conventions. The emphasis is on identifying what is common between the languages and providing an environment that is similar for both languages. The process of standardizing interlanguage communication has to focus on how much can be accomplished just by recognizing commonality, and how much requires new syntax and explicit interfaces. A significant part of the problem can be solved by providing an execution environment that stores function arguments in the same order, has a common prologue and epilogue sequence, performs external name linkage based on common identical names, and has a straightforward data mapping. More can be done with a header such as `fortran.h`. Additional `typedef` names could be provided that map onto the Fortran-77 `INTEGER`, `LOGICAL`, `REAL`, and `DOUBLE PRECISION` types.

The Cray Research method of specifying that the CFT77 conventions are the interlanguage conventions has worked quite well, but there are a few problems. There is no C analog to blank common. Sometimes there is a need for a CFT77 subprogram to call a lower-case C function. This forces the programmer to write an interface function and encounter additional overhead. Finally, there is the unfortunate difference in the values used for CFT77's `LOGICAL` and SCC's true and false. More work needs to be done in these areas, but overall the mechanisms exist to perform interlanguage communication and they are both easy to use and understand.

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 C standard. He specializes in the areas of floating-point, vector, array, and parallel processing with C language and can be reached at (612) 683-5818, tam@cray.com, *or* uunet!cray!tam.

∞

35. Iterators

Thomas J. Pennello
MetaWare Inc.
2161 Delaware Avenue
Santa Cruz, CA 95060-5706

Abstract

The iterator is perhaps the most significant—and overlooked—language construct invented in the 1970s. It first appeared in the CLU language authored by Barbara Liskov of MIT.

Here I describe iterators as designed in MetaWare's High C language. Initial motivations and examples of use are followed by a precise description of iterator syntax and semantics. Finally I summarize some advantages and disadvantages.

I argue that iterators contribute to program maintainability and readability, and add significantly to the C language's expressive power.

Introduction

This paper describes a powerful language construct called an *iterator*.

An iterator is a new kind of function that supplies values for the variable(s) of an iterator-driven `for` loop. The `for` loop body is executed each time the iterator produces value(s) for the `for` variable(s) via a new predefined function `yield`, as discussed below.

The major benefit of iterators and iterator-driven `for` loops is that the algorithm to determine the values of the `for` variables for each loop iteration is defined separately in an iterator rather than being exposed in the Standard C `for` loop construct itself. The iterator may be invoked in many such `for` loops.

Iterators originated in CLU, a programming language from MIT that was designed to promote the use of abstractions in program construction. Iterators and iterator-driven `for` loops are not part of Standard C. They have been made available in MetaWare's High C because they help programmers produce readable, easily modified code. For example, in Standard C you might write:

```
int i;
for (i = 1; i <= 10; i++)
   printf("%d squared is %d\n",i,i*i);
```

This loop could be written using an iterator, as follows:

```
void Upto(int Lo, int Hi) -> (int) {
   int i = Lo;
   while (i <= Hi) yield(i++);
   }

for i <- Upto(1,10) do
   printf("%d squared is %d\n",i,i*i);
```

The sequencing from one number to another (1 through 10 in this example) is programmed once in the `Upto` iterator. The syntax `for i <- Upto(1,10)` "starts" the iteration. Each time the iterator `Upto` calls the predefined function `yield` with a value, that value is substituted for `i` in the body of the `for` loop, and the body is executed. When execution of the body is finished, control is returned to a point immediately after the `yield` and `Upto` continues its `while` statement.

Each call to `yield` causes the `for` loop body to be executed once. When `Upto` is finished yielding, it returns just like a regular C function, and the invocation of `Upto` (and therefore the `for` loop itself) is complete. Control then passes to the statement after the `for` loop.

The syntax `-> (int)` in the header of the iterator definition is the only thing that distinguishes it from a normal function. After the `->` appears a prototype-form parameter list specifying the type(s) of the results yielded by the iterator (more on multiple yields later). Here `Upto` yields an `int`—one for each execution of the `for` loop body. Within the definition of an iterator, the predefined function `yield` is defined. Outside an iterator you may use the name `yield` for any other purpose.

Some Uses for Iterators

Iterators are quite general, but they are probably most useful for iterating over each element in some set. The set is often the entire contents of some data structure, but can be restricted to those elements in a data structure that meet certain conditions. The set might not be stored in a data structure at all; for example, an iterator could be used to provide the prime numbers up to 100 by computing and yielding them.

The following sections present some programming situations where you might want to consider using iterators instead of regular `for` loops. They also show a few of the programming problems that can be solved with iterators.

List Processing

Let us look at a loop that processes each element in a list:

```
for e <- each_element(list) do {
   ... // Process element e.
   }
```

In the loop, `e` is the single `for` variable. Each time through the loop, `e` is given a value yielded by the iterator `each_element`. (In this case, we would expect `e` to have a different value each time through the loop, but this is not always so.) `e` is really a parameter to the loop body. It is not visible outside the loop body and is instantiated anew each time through the loop. Assuming the following declaration of an element:

```
typedef struct element {
   ...  // Some data.
   struct element *next;
   } *element;
```

The iterator `each_element` might look like this:

```
void each_element(element e) -> (element) {
   while (e != 0) {
      yield(e);
      e = e->next;
      }
   }
```

Each time the predefined function `yield` is called, control passes to the `for` loop that invoked `each_element`. The `for` variable element takes on the current value of `e` for that pass through the loop body. At the bottom of the `for` loop, control returns to `each_element` at the statement following the call to `yield`.

Changing the Implementation of a Data Type

Suppose in some program you make heavy use of a sorted linked list. If you want to speed up your code, you might replace the list with a binary tree. That would mean not only replacing the code that implements the list (the insert, delete, and sort functions, for example) but also finding all the places in the program where the list gets traversed. You would need to replace all the `for` loops that are some variation on:

```
for (lptr = head; lptr != NULL; lptr = lptr->next) {
   ...  // Use list element.
   }
```

You may traverse the list several places in your program, and you would have to change them all. On the other hand, if you use an iterator, you would still have to replace the code that implements the list, but you would not have to touch the code that traverses the list. Your loops would look something like this (both before and after you changed over to a binary tree):

```
for lptr <- traverse_list(head) do {
   ...  // Use list element.
   }
```

Traversing a Tree

A `for` loop is a natural way to conceptualize getting each item in a data structure. You might write pseudo-code for some loop that processes every node in a tree, such as:

```
for each node in tree {
   ... // Process node.
   }
```

However, when you actually write such a loop in C, probably you would not use a `for` loop. Trees are naturally traversed recursively, and C's `for` loop construct cannot naturally express a recursive computation. However, a High C iterator-driven `for` loop can express recursive computations as easily as any other kind; the resulting code looks almost identical to the pseudo-code:

```
for n <- each_node(tree) do {
   ... // Process n.
   }
```

The iterator `each_node` can be recursive. Thus, a recursive algorithm can easily be used to generate values for variables.

Recursion and Code Clarity

In the list-traversal examples in *Some Uses for Iterators* above, you may complain that all we have done is make nice, straightforward `for` loops complicated. But consider a different problem: iterate through the nodes of a binary tree, where the tree is implemented using a data structure that for each node N has a pointer to the left and right subtrees of N.

The obvious way to obtain the nodes of such a tree is by a simple recursive tree walk. However, this is not possible using Standard C `for` loops. Imagine having to translate the following pseudo-code into C:

```
Process the tree:
    Do some stuff to the tree
    For each node in the tree
        Print the node
    Do some other stuff to the tree.
```

Because you cannot use a `for` loop to get the nodes of the tree, you would probably package the node-printing process into a routine and pass that routine to be executed by a tree-walking routine:

```
typedef struct node {
   ... // Some data.
   struct node *Left, *Right;
   // Left and right subtrees.
   } *Node;

void Print_node(Node N) {
   printf("Node is ");
   ... // Code to print a Node.
   }

void Each_node(Node N, void Doit_toit(Node N)) {
   if (N == 0) return;
   Each_node(N->Left, Doit_toit);
   Doit_toit(N);
   Each_node(N->Right, Doit_toit);
   }

Process_the_tree(Node Root) {
   ... // Do some stuff to the tree.
   Each_node(Root, Print_node);
   ... // Do some other stuff to the tree.
   }
```

Rearranging the computation this way allows the recursive tree walk to occur, but there is a cost. The simple `for` loop in the pseudo-code clearly expresses that a computation is being done on each element of the tree data structure. This is no longer as apparent in the body of `Process_the_tree`. To find out what is being done you now have to look outside `Process_the_tree`, in `Print_node`. In this simple example, the choice of good names assists a great deal in promoting understanding of the code. But when dealing with real problems, the code is more complex and the names are usually not perfectly explanatory.

Here is a solution to the same problem, using iterators:

```
void Each_node(Node N) -> (Node) {
   if (N == 0) return;
   // Walk the subtrees.
   for L <- Each_node(N->Left)  do yield(L);
   yield(N);
   for R <- Each_node(N->Right) do yield(R);
   }

Process_the_tree(Node Root) {
   ... // Do some stuff to the tree.
   for Node <- Each_node(Root) do {
      printf("Node is ");
      ... // Code to print a Node.
      }
   ... // Do some other stuff to the tree.
   }
```

Notice that the code for `Process_the_tree` mirrors the pseudo-code extremely closely, yet unlike the Standard C version the algorithm that determines the successive values of `Node` in the `for` loop body is recursive. This is a major increase in expressive power.[6]

The body of `Each_node` is not very elegant (or efficient) as written. The recursive calls do nothing but re-yield a result already yielded at a deeper level of recursion. Using nested functions[7] and the fact that calls to `yield` can occur within functions nested within iterators, we can produce an elegant version:

```
void Each_node(Node N) -> (Node) {
   void P(Node N) {
      if (N == 0) return;
      // Walk the subtrees.
      P(N->Left);
      yield(N);
      P(N->Right);
      }
   P(N);
   }
```

Work done by the programmer in the Standard C version—packaging the body of the pseudo-code `for` loop as a function—is done instead by the High C compiler in the iterator version. This is appropriate. After all, compilers were invented to promote more easily understandable and modifiable languages. Why make a programmer do what the compiler can do?

[6]We are not claiming an increase in computational power. C is already Turing-machine equivalent. We are claiming that algorithms can be expressed in a more natural fashion, making the code easier to write, understand, and modify.

[7]The nesting of functions is another High C extension. It is documented in *Volume 2, number 3*, page 240 of *The Journal*.

Replacing Macros with Iterators

Another place an iterator can be gainfully used is when the iteration algorithm, although not recursive, is complex. Often, in this circumstance, a macro is designed to make up for the shortcomings of the C `for` loop. Consider this example from the High C 2.x series of optimizing compilers. The problem is to sequence through the objects that overlap a given object `obj` in memory. Prior to the addition of iterators, a macro was required to simulate the iteration:

```
#define for_each_overlapping_object(o,obj,p){\
   struct obj_entry *_op = &objtab[obj];\
   obj_class_type _class = _op->class;\
   long _len = _op->len;\
   long _disp = _op->disp;\
   ushort _word = _op->un.word;\
   bool is_deref = (ea_DEREF & _op->flags) != 0;\
   bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags) != 0;\
   register struct obj_entry *p; int o;\
   for(o=1,p=objtab+1;o<=last_object;o++,p++){\
      if (is_deref && \
          ((p->flags&ea_DEREF) ||\
              (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&\
               p->xlen >= _op->xlen &&\
                   _op->disp < p->xlen)||\
          is_adr  && (p->flags&ea_DEREF)!=0 &&\
             p->xlen<= _op->xlen|| \
             p->class == _class && \
          (p->un.word == _word ||\
          (p->flags&_op->flags&ea_TEMP)) &&\
          (p->disp<=_disp && p->disp+p->len >_disp ||\
           _disp+_len > p->disp && _disp < p->disp)){
```

It was invoked as follows:

```
object_index obj;
...
for_each_overlapping_object(o,obj,p)
   ... do things with o and p ...
   }}}  // Required to match {'s in macro.
```

The `#define` defined the variables `o` and `p` whose names were arguments to the macro.

With this approach an enormous amount of code is reproduced each time the macro is invoked. Every time a change is made to the macro, every module where it appears must be recompiled. An iterator requires much less maintenance. When a change is made, only the iterator itself and not its "client" `for` loops need be recompiled. Another advantage is that the iterator can declare the types of its parameter and yielded results, providing improved type checking at the iterator invocation.

```
void Each_overlapping_object(object_index obj)
        -> (int o, struct obj_entry *p) {
   struct obj_entry *_op = &objtab[obj];
   obj_class_type _class = _op->class;
   long _len = _op->len;
   long _disp = _op->disp;
   ushort _word = _op->un.word;
   bool is_deref = (ea_DEREF & _op->flags) != 0;
   bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags) != 0;
   register struct obj_entry *p; int o;

   for(o=1,p=objtab+1;o<=last_object;o++,p++){
      if (is_deref &&
          ((p->flags&ea_DEREF) ||
               (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&
               p->xlen >= _op->xlen &&
                   _op->disp < p->xlen)||
          is_adr  && (p->flags&ea_DEREF)!=0 &&
             p->xlen<= _op->xlen||
             p->class == _class &&
          (p->un.word == _word ||
          (p->flags&_op->flags&ea_TEMP)) &&
          (p->disp<=_disp && p->disp+p->len >_disp ||
           _disp+_len > p->disp && _disp < p->disp)) {
            yield(o,p);
         }
      }
   }
```

It is now invoked as follows:

```
object_index obj;
...
for o,p <- Each_overlapping_object(obj) do
   ... do things with o and p ...
```

Note that the iterator yields two results for each execution of the `for` loop body: the `int o` and the `struct obj_entry *p`.

Now we move away from examples and to a more precise specification of iterators and iterator-driven `for` loops.

Syntax and Constraints

To invoke an iterator `I` for *n* for variables, write:

```
for N1,N2,...,Nn <- I(E1,E2,...,Em) do
   ...   // Body of for loop, using Ni.
```

Iterators yield one or more values. The declarative syntax for both the iterator input arguments and yield types follows that of the input arguments to normal functions. An iterator `I` is declared as follows

```
void I(Formal_parm_list1) -> (Formal_parm_list2) {
   ...   // Body of iterator, with calls to yield.
   }
```

Arguments

The iterator's *Formal_parm_list1* has exactly the same constraints and semantics as the formal parameter list of other functions, except that we additionally require that this list use the ANSI prototype syntax. Old style C function parameter declarations are not permitted. When the iterator is invoked in a `for` loop, the expressions passed to it must satisfy the same constraints as those of expressions passed to a function, given the same parameter list. The parameters are passed in the same ways.

Formal_parm_list2 must also use the ANSI prototype syntax. Here, however, the names of the parameters may be omitted, just as they may be omitted when a function is declared but not defined.

Formal_parm_list2 is called the *yield list.* The `for` variables of a `for` loop that invokes the iterator take on values of the types specified in the yield list, respectively. The iterator supplies the values through a call to the function `yield` which is defined in the body of each iterator:

```
yield(E1, E2, ... En);
```

where the following constraint must be satisfied:

> If `void yield(`*Formal_parm_list2*`);` is a valid function declaration, then `yield(`E_1`,` E_2`,` ..., E_n`);` must be a valid call to that function.

Alternatively, the syntax of High C's named parameter association[8] can be employed to yield the values, just as in a function call. *Formal_parm_list2* must contain the names of the parameters. However, even if that notation is not used, it is useful to name the "yield parameters" for the sake of documentation.

Like other functions, iterators can use the `...` notation to specify that they may take a variable number of arguments. The yield formal-parameter list may

[8]Named parameter association is another High C extension. It is documented in *Volume 2, number 3*, page 239 of *The Journal.*

also use `...`, requiring the use of the `va_arg` macros within the `for` loop body to access the remainder of the expressions yielded.

Semantics

Each time an iterator executes a yield statement, the body of the `for` loop is executed with its i^{th} `for` variable assuming the value of the i^{th} expression yielded, for all $1 <= i <= m$. When the iterator returns (just as a function may return), the `for` loop is terminated. A `break` or `goto` from the `for` loop body also terminates the iteration.

The semantics of iterators and `for` loops that invoke them can be specified precisely in terms of an implementation using nested functions, as follows. The compiler bundles up the body of each iterator-driven `for` loop and turns it into a function. Each so-bundled function is passed as an extra parameter to the iterator. The iterator is called once per `for` loop and receives as a parameter the function that used to be the body of that `for` loop. Each call to `yield` within the iterator is translated to a call to the function that is its extra parameter.

More formally, the meaning of:

```
void I(Formal_parm_list1) -> (Formal_parm_list2) {
   ...   // Calls to yield in here.
   }
```

and:

```
for N1,N2,...,Nn <- I(E1,E2,...,Em) do
   ...   // for loop body here.
```

is precisely the same as the meaning of:

```
void I(void yield(Formal_parm_list2)!, Formal_parm_list1) {
   ...   // Calls to yield in here.
   }
```

and:

```
{// Would-be for loop:
void For_loop_body(Formal_parm_list3) {
   ...   // for loop body here.
   }
I(For_loop_body,E1,E2,...Em);
}
```

where *Formal_parm_list3* is *Formal_parm_list2* but with the names N_1, N_2, ..., N_n replacing the parameter names (if given) in *Formal_parm_list2*.

Note that the calls to `yield` in the iterator `I` become calls to function `I`'s function parameter named `yield`—the syntax is the same in both cases. Also, the `for` loop body is made into a function and passed as an extra first parameter to the function `I`. Calling function `I` is what starts the loop. `I`'s return terminates the loop, barring a `goto` out of the loop body.

It is the function `I` per se that does the iterating. That may occur via a contained loop, for example, or by `I` calling itself recursively as it traverses some data structure. Each time it wants to yield something to the `for` loop body, it does so by calling its yield parameter and passing the value(s) of the `for` variable(s) to the next iteration of the loop.

The `!` declaration guarantees we can pass a nested (non-global) function to the iterator. Because the `for` loop is guaranteed to lie within a function, the `for` loop body, when transformed into a function, is guaranteed to be a nested function. For example:

```
void Primes(int Lo,int Hi) -> (int ThePrime) {
      // We named the yield result.
      // Yield the primes in the interval Lo..Hi.
   int I,J;
   extern double sqrt(double);
   for (I = Lo; I <= Hi; I++) {
      // Ask if we can divide I evenly:
      for (J = 2; J <= sqrt(I); J++)
         if ((I/J)*J == I) goto Composite;
      yield(I);                   // I is prime.
   Composite: ;
      }
   }
...
for I <- Primes(1,100) do printf("%d is prime.\n",I);
```

This example has the same semantics as the following:

```
void Primes(void yield(int ThePrime)!,
            int Lo, int Hi) {
   // Yield the primes in the interval Lo..Hi.
   int I,J;
   extern double sqrt(double);
   for (I = Lo; I <= Hi; I++) {
      // Ask if we can divide I evenly:
      for (J = 2; J <= sqrt(I); J++)
         if ((I/J)*J == I) goto Composite;
      yield(I);                 // I is prime.
   Composite: ;
      }
   }
```

```
...
   {
   void For_loop_body(int I) {
      printf("%d is prime.\n",I);
      }
   Primes(For_loop_body,1,100);
   }
```

Predefined Function `yield`

Because the types yielded by an iterator are described by *Formal_parm_list*, there is no restriction on the types except that they must be types that can be passed to a normal function. Thus, an iterator can yield integers, structures, functions, and even iterators. (An iterator that computes the strongly connected components of a graph and an example of its use are provided with our High C distributions. This iterator's yield is itself an iterator that yields one set of strongly connected components. No one has yet accepted our challenge to rewrite this example in Standard C.)

Advantages and Disadvantages

Reusable Code

The major benefit of iterators is that the algorithm to determine the values of the `for` variables for each loop iteration is defined separately in the iterator. Unlike the Standard C `for` loop, the iterator can do an arbitrary amount of computation and can automatically maintain its environment across passes through the body of the loop, because it suspends rather than returns when it provides the loop body with values. Once the iterator is written, it can be invoked by as many loops as desired without repeating the code that provides the values used within the loop (as compared with using many copies of standard `for` loops, which involve repeating the code for each loop).

Information Hiding

The term information hiding comes from the literature in structured programming. It refers to the placement of information only where it is needed keeping the information from where it is not needed. For example, a module implementing a tree data type may wish to hide all the details of traversing trees within functions defined in that module. In Standard C, however, traversing the tree generally means exposing the tree data structure in any `for` loop sequencing through nodes of the tree. Instead, iterators can be used to localize the sequencing techniques and their required data-structure access to within the tree module itself. The `for` loop need only invoke an appropriate iterator. It need not be concerned with the representation of trees.

Information hiding promotes better and more easily maintained programs, and reduces recompilation of modified programs. Iterators promote information hiding.

Execution Speed

The major drawback of iterators is that they are slow, compared to standard `for` loops, because they involve the overhead of function calls. Speed is lost by placing the computation of the iteration in a separate function. A loop that executes n times involves $n + 1$ function calls related to loop overhead. The speed loss occurs not only due to the function linkage but also because the body of the `for` loop becomes a nested function. Any variables in the body's parent accessed from the body cannot reside in high-speed machine registers.[9] For example:

```
int Sum = 0;
for I <- Upto(1,10) do
   Sum += I;
printf("Sum of 1 to 10 is %d\n", Sum);
```

is implemented as:

```
int Sum = 0;
void Body(int I) {Sum += I;}
Upto(Body,1,10);
printf("Sum of 1 to 10 is %d\n", Sum);
```

Here, `Sum` cannot reside in a register because it is accessed from the nested function `Body`. Thus all accesses to `Sum` will be slowed. Therefore, if extreme speed is required, do not use iterators, especially for simple `for` loops where the iteration algorithm is simple and there is no need for hiding it away in one place in an iterator.

In earlier releases of our compiler, any function that includes an iterator `for` loop containing an `exit` will have none of its local variables allocated to registers, because of the way `exit` was implemented. However, we have avoided this limitation in the latest release.

Conclusion

Iterators allow you to express some computations more naturally by virtue of recursion. They can make your code easier to maintain because there are fewer places where code must be modified if there is a design change. Recursion is possible with iterators because the function that drives the `for` loop does not

[9] This is due to the current state of optimizer technology. An improvement may come some day.

have to return. If you use `for` loops to access the elements of a data structure, iterators allow you to change the implementation of the data type without changing the implementation of the `for` loops. With an iterator, each `for` loop is driven by an independently specified computation that can maintain its environment across invocations of the loop. Without an iterator, the syntax accessing the structure is inside each `for` loop. Therefore each `for` loop must be modified, because the loops contain code specific to the data structure.

Thomas Pennello, MetaWare's Vice President and Chief Technical Officer, did his graduate work under Frank DeRemer at UCSC. Together, they developed fully-automatic error recovery, and efficient LALR look-ahead set computation. Thomas Pennello received his Ph.D. degree in transformational grammars from UCSC. He may be reached at tom@metaware.com*.*

∞

36. Miscellanea

compiled by **Rex Jaeschke**

Implicit Function Declarations

I sent the following question to David Prosser, redactor of the C Standard, for his unofficial comments. *[Ed. This is* **not** *to be construed as an official interpretation.]*

> §3.3.2.2, Function Calls, on page 41, lines 28–32 reads: "If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration
>
> ```
>         extern int identifier();
> ```
>
> appeared." This is very clear to me. However, what I'm looking at are cases like
>
> ```
>         (f)()
> ```
>
> where `f` is not declared. Now `f` and `(f)` are both primary expressions and, therefore, also postfix expressions. As such they can be used with the `()` function call operator. The quote above says "If the expression that precedes the ..." Which expression in my example precedes the call? Well both the expression `f` and the expression `(f)` do. The difference though, is that the quote also says, "If the expression ... consists solely of an identifier ..." Well, lexically (I'm guessing) the expression immediately preceding the call is `(f)` *not* `f`. And since `(f)` contains more than just an identifier, does the implicit typing apply? Taking the words literally, I think not. Note, however, we don't say "that *immediately* precedes". But then again, I don't think we meant "that precedes it somewhere in the token stream" either. That's why I took it to imply "immediate."

David's reply was:

> This issue was debated and the "can't be parenthesized" side won. The winning side argued that it was too difficult to recognize when

an undeclared identifier that wasn't immediately followed by a left parenthesis was still going to be an undeclared function to call. And forcing all implementations to get this right was too high a price to pay for little, if any gain.

I think I have correctly recalled the arguments. I, on the other side, argued that a lazy-evaluating parser could delay complaining about unknown identifiers until the entire context was clear. Thus, even situations such as

```
( exp1, exp2, f)( arg-list );
```

could be handled when `f` is undeclared. As was common, the implementors won the day. However, their argument does have a valid point in that when functions are called, the name isn't parenthesized, isn't the last element of a comma list, and isn't the second or third operand of a `?:`. (That is, unless the function has already been declared.)

I also involved Derek Jones in the discussion since he is the project editor for the ISO C Normative Addendum. During this exchange he raised the following related point.

Consider the following example:

```
void g()
{
        if (f != 0)  /* f undeclared here */
                f(); /* implicit declaration */
}
```

Is this a valid construct? The Standard does say the implicit declaration is *as if* it were really there, at the beginning of the block *before* any statements.

David's response was:

This is more interesting. It is just a variation on the BSI comment #16 that I looked over when I was putting together the final editorial changes to the draft. I had been convinced that I needed some wording change in the draft, but when reviewing the changes with the officers of X3J11, we found that we'd be making substantive changes by nailing down the declaration. No matter where you choose to put the created declaration, it is possible to arrange for a use of the name that would have otherwise been valid if the declaration actually were present.

Thus, the argument goes, the only guarantee by the current wording is that the created declaration is in scope *by the time of* *the function call expression, but need not be in scope earlier.* You say, "But in the above example, the only valid place for a declaration is between the { and the `if` statement, and thus that use of `f` must now be valid." This is not unreasonable, but we know that certain identifiers can be declared anywhere within a block (tags, structures, unions, enumerations) due to `sizeof` and casts. It is not at all unreasonable that the created declaration occurs anywhere within the block and, of course, carries its visibility forward until the closing }.

Even if my argument is tenuous, I don't believe anyone ever intended or expected the created declaration have any effect on previous expressions (or even, possibly, within the same expression).

Strict Type Checking

In §3.4, Constant Expressions, the following constraint is specified: "Each constant expression shall evaluate to a constant that is in the range of representable values for its type."

Based on what appears to be loose checking in this regard by my 'ANSI-conforming' compilers, I have contrived the following test case.

```
#include <limits.h>

char c[INT_MAX + 1];    /* error */

int i = INT_MAX + 1;    /* error */

enum etag {e1 = INT_MAX, e2};   /* error */

void f()
{
        void g(int);
        static long l = INT_MAX + 1;    /* error */

        g(l);

        switch (l)
case INT_MAX + 1: ;             /* error */
}
```

The initialization of `l` is perhaps the most interesting since the type of the initializer is `int` yet the value won't fit in an `int`. This is more likely detected on implementations in which `int` and `long` map to the same size.

Signed Address Space

[Ed: The following information comes courtesy of Conor O'Neill, INMOS Ltd., designers of the Inmos parallel processing transputer.]

Transputers use a signed address space, going from the most negative integer address, through zero, to the most positive address. In many cases this can simplify address calculations. (You don't need to worry about overflow.) Hence, an all-bits-zero pointer points smack into the middle of the address space.

There are three approaches one can take regarding defining a null pointer:

1. Use a different bit-pattern for `NULL`. Transputers normally use `MOSTNEG INT` as a `NULL` pointer, because it is the address of a transputer link, so data and code cannot reside there.
2. Use all-bits-zero to indicate `NULL`. Ensure that nothing is ever allocated which crosses (or approaches very closely from below) the 'zero' address.
3. Ignore the problem—use all-bits-zero anyway.

We use approach 3 on our 32-bit processors. So far as we know, no one has attached 2 Gigabytes to a transputer, so the problem of making sure that nothing is allocated there does not arise. (Note, transputers do *not* use virtual memory.) No doubt it would be possible to tailor both the loader and `malloc` so that no memory was ever allocated at address zero. This would then satisfy approach 2.

16-bit processors are a different matter. Many of our customers use more than 32K of memory. Therefore we've taken approach 1, but used a specific bit-pattern (`MOSTNEG INT`, or 0x8000) which can never correspond to user-accessible memory.

Generate `limits.h` and `float.h`

[Ed: The following announcement was extracted from a posting to the conference comp.compilers. I have tested the source with several compilers. It produces a lot of useful information. For example: size, representation, and alignment of the basic types, character packing order, and integral and floating type properties. If nothing else, it serves as a good exerciser and sanity check. It broke one of my compilers.]

Enquire.c (which used to be called *config.c*) is a program that determines many properties of the C compiler and machine that it is run on. As an option it produces the Standard C headers `float.h` and `limits.h`.

It is a good test-case for compilers, since it exercises them with many limiting values, such as the ability to handle the minimum and maximum floating-point numbers.

Version 4.3 of enquire.c has been submitted to comp.sources.misc, and will appear as part of the gcc distribution (where it is used to generate `float.h`). It is also available by anonymous ftp from mcsun.eu.net and hp4nl.nluug.nl as misc/enquire43.c, and by mail from

> info-server@hp4nl.nluug.nl

by sending the mail message

> request: misc
> topic: enquire43.c

ISO C Standard Status Report

ISO C is completed and it is technically equivalent to ANSI C.

ANSI and ISO now use the same format for standards but did not do so when X3J11 started its task. Therefore, the ANSI standard had to be significantly reformatted to make an equivalent ISO standard document. Unfortunately, the line number references were not permitted to remain. This non-trivial conversion has been done by the ANSI C redactor, David Prosser of AT&T. (Many thanks, David.) The document is now wending its way through the ISO maze for final processing.

New work is continuing in WG14, the ISO C committee. Several national groups are heading up projects which will eventually result in one or more normative addenda. Derek Jones, of the UK delegation, was unanimously nominated as Project Editor for these addenda.

Specifically, the primary work items are:

- UK members are adding clarification wording to many parts of the standard but are *not* making substantive changes.
- The Japanese are working full steam on extra multibyte library support. (At the Copenhagen meeting in October 1990, it was agreed this work would be contained in a new header and would *not* impact the existing standard headers.)
- The Danish continue to work on their alternate trigraph proposal. (Bjarne Stroustrup issued a revised version late in 1990 which proposed a `!` postfix punctuator.)

It is expected that in a couple of years one or more of these addenda will be completed and accepted by member nations in some form. At that time they will be added to the ISO C standard and **will have the full weight of a standard**. At that point, ISO C becomes a proper superset of ANSI C. Whether these addenda are retrofitted back into the ANSI standard or not remains to be seen.

In this regard, I foresee the standard possibly breaking into a mandatory core part with optional modules, just like some other language standards. Why? Well, the Federal Information Processing Standard (FIPS) is based on the current ANSI C standard. FIPS reflects the needs of government agencies in the US. If a new ANSI standard adopts all of the ISO addenda, it's possible FIPS could not use that new ANSI C standard since alternate trigraphs or multibyte support [for example] might not be considered *mandatory* requirements for US government business. So does FIPS then make its own standard? No, I think it better they require ANSI C Base Level and not the other options.

It does seem reasonable, however, that at some time in the future the ANSI standard will take on the ISO format to make it easier to reconcile the two standards. At this stage David Prosser of AT&T is the redactor of both standards. I don't know how that will work once the addenda come together since they are not being produced by the redactor.

At the September, 1990 X3J11 meeting, the officers proposed that X3J11 consider moving **all new C standard work to the ISO arena including, eventually, interpretations**. This proposal is under consideration. As you might think, it is a controversial one. Due to ANSI requirements, the interpretations *must* be done by X3J11 for at least the foreseeable future but new work will almost certainly be restricted to the ISO level. Since it is acknowledged by both X3J11 and WG14 that the bulk of the technical expertise lies within X3J11, the two committees have agreed to meet jointly once a year (every 2 meetings under our current schedule). As such, a joint meeting will be held in Milan, Italy on December 11–13, 1991.

I was the US Head of Delegation at the Copenhagen meeting of WG14, held last November. Before debate opened on the latest alternate trigraph proposal from Denmark, I interjected a suggestion. Instead of continually shooting holes in new and revised proposals, we should establish some guidelines for what approaches might and might not be acceptable in terms of the existing language, its history, and 'the Spirit of C.' This suggestion was then debated at length. The primary categories for change are summarized in the electronic poll on page 280. (I posed this question in order to get input to help make future decisions and I intend to raise this issue at the March 1991 meeting of X3J11.)

Another prospect raised in WG14 (by Denmark) is more direct support for extended national characters in identifiers and the idea of a compile-time locale. (Note that the poll question on page 278 arose from that agenda item.)

I am looking forward to the next WG14 meeting to be held in May, in Tokyo, Japan. Members of the Japanese delegation are major contributors to WG14 and will, no doubt, bring us up to date on their latest multibyte extension proposal.

NCEG Status Report

The Numerical C Extensions Group (NCEG) is very much alive and well. In September 1990, the ANSI group SPARC approved it as a working group within X3J11 (tentatively called X3J11.1). There was 1 NO vote: that we should produce a standard rather than a report. The parent body X3 must also vote before the process is complete.

In November 1990, WG14 adopted NCEG as a rapporteur group within WG14. NCEG now has the same role in both committees (WG14 and X3J11)—to research numerical extensions and publish a Technical Report (tentatively sometime in 1992). So NCEG issues now have impact in the ISO arena as well. Most of the NCEG proposals are new inventions, although quite a few are based loosely on prior art. Note that a Technical Report is *not* binding like a standard.

The September 1991 meeting will not be in conjunction with an X3J11 meeting. As such, NCEG will meet for more than the usual two days in order to start towards closure on the Technical Report.

Calendar of Events

- May 13–15, **ISO C SC22/WG14 Meeting** – Location: Tokyo, Japan. Contact the US International Rep. Rex Jaeschke at (703) 860-0091, or *rex@aussie.com*, or the convenor P.J. Plauger at *uunet!plauger!pjp* for information.
- June 13–14, 1991 **First ISO C++ Meeting** – Location: Lund, Sweeden.
- June 17–21, 1991 **ANSI C++ X3J16 Meeting** – Location: Lund, Sweeden.
- June 24–28, 1991 **ACM SIGPLAN '91 Conference on Programming Language Design and Implementation** – Location: Toronto, Ontario. The conference seeks original papers relevant to practical issues concerning the design, development, implementation, and use of programming languages.
- August 12–16, **International Conference on Parallel Processing** – Location: Pheasant Run resort in St. Charles, Illinois (near Chicago). Submit software-oriented paper abstracts to Herbert D. Schwetman at *hds@mcc.com* or by fax at (512) 338-3600 or call him at (512) 338-3428.
- August 26–28, 1991 **PLILP 91: Third International Symposium on Programming Language Implementation and Logic Programming** – Location: Passau, Germany. The aim of the symposium is to explore new declarative concepts, methods, and techniques relevant for

implementation of all kinds of programming languages, whether algorithmic or declarative. Contact *plilp@forwiss.unipassau.de* for further information.

- September 23–27, 1991 **Numerical C Extensions Group (NCEG) Meeting** – Location: At an Apple facility in Cupertino, California (Silicon Valley area). Note that this will *not* be a joint meeting with X3J11. As such, NCEG will meet more than the usual two days. The actual number will be determined at the March 1991 meeting. For more information about NCEG, contact the convenor Rex Jaeschke at (703) 860-0091 or *rex@aussie.com*, or Tom MacDonald at (612) 683-5818 or *tam@cray.com*.
- November, 1991 **ANSI C++ X3J16 Meeting** – Location: Toronto, Ontario.
- December 11–13, 1991 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Milan, Italy.
- May 11–12, 1992 **Numerical C Extensions Group (NCEG) Meeting** – Location: Salt Lake City, Utah.
- May 13–15, 1992 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Salt Lake City, Utah.

News, Products, and Services

- Prentice Hall is publishing a **UNIX System V Release 4** manual called *ANSI C Transition Guide*. 1990, 64 pages, ISBN 0-13-933698-2.
- **paracom, inc** is now shipping a Standard C compiler for their development systems built around the **transputer**. The compiler is heavily extended to support the parallel nature of the hardware and is available as a cross-compiler from Sun-UNIX. (708) 293-9500 or fax (708) 884-9065.
- **Intel** has announced their 32-bit C development kit for Intel 386/486 systems running DOS. The kit includes the compiler, Microsoft-compatible libraries and librarian, source-level debugger, linker, utilities, and Make. It also includes a DOS extender.
- In an interesting development **Cygnus Support** is now providing commercial support for **GNU** software from the **Free Software Foundation**. Currently supported platforms include Sun-3 and SPARC, Silicon Graphics IRIS-4D, and VAX Ultrix and BSD. For details call them in California at (415) 322-3811 or fax (415) 322-3270.
- Based on some comments in a net conference recently, I tracked down a couple of interesting papers. They are: *A Study of a C Function Inliner*

by Jack W. Davidson and Anne M. Holler; *Software–Practice and Experience*, Vol 18(8), 775-790 (August 1988); and *Subprogram Inlining: A Study of its effects on Program Execution Time*, by the same authors. The second is published by the Department of Computer Science at University of Virginia where the authors work. A distribution kit for the INLINER software is available for a handling fee provided you have a *pcc* source licence. For more information contact Prof. Davidson at *jwd@virginia.edu*.

- **XDB Systems Inc.** of College Park, MD has announced an **embedded SQL** C precompiler for DOS and OS/2. (301) 779-6030.

- **LPI** has upgraded its ANSI C compiler NEW C for Sun SPARC. Support is provided for their CodeWatch symbolic debugger. The workstation price is $1,695 while upgrades cost $425. (508) 626-0006 or fax (508) 626-2221.

- **Bullseye Software** of Seattle is shipping version 2 of its C source analysis coverage program C-Cover. This MS-DOS-based package identifies the control structures not used in your source and gives objective metrics to use in testing. (206) 524-3575.

- *[Ed: from comp.compilers]* – The University of Colorado has released a new version of the Eli compiler construction system for Sun3, Sun4, and VAX computers. Eli integrates off-the-shelf tools and libraries with specialized language processors to generate complete compilers quickly and reliably.

  Costs are: installation tape (DC600A cartridge $75, 9-track $100), optional hard copy documentation (about 600 pages: $30)

  The University requires payment in advance via a check or money order drawn on a US bank and made payable to the University of Colorado. Be sure to specify cartridge or 9-track tape, and optional documentation if required. Mail your order to:

  Software Engineering Group
  Dept. of Electrical and Computer Engineering
  University of Colorado
  Boulder, CO 80309-0425

- Benjamin Cummings has published a C edition of *Crafting A Compiler*, entitled *Crafting A Compiler With C*, by Charles Fischer and Richard LeBlanc. ISBN 0-8053-2166-7.

∞