The Journal of

C Language Translation

Volume 3, Number 1

June, 1991

Publisher and Editor	Rex Jaeschke
Technical Editor	P.J. Plauger
Standards Editor	Jim Brodie
Numerical Editor	Tom MacDonald
Technical Consultant	David Prosser
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 the Internet 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

1. Translating Multibyte Characters

P.J. Plauger

Abstract

The Standard C library includes several pairs of functions that map between character-set representations. One pair maps a character string to a form that collates properly. A second pair maps multibyte characters to wide characters. A third does the reverse.

I have written an implementation of the Standard C library intended to be both exemplary and highly portable. It lets you alter the behavior of these functions when you change locales. It also lets you define your own locales as human-readable text files.

The various mapping functions are built around table-driven finite-state machines. You specify state tables in the locale file to define each mapping. The result is a set of mapping functions that is highly configurable, yet still offers modest performance.

Introduction

The C Standard lets you define an open-ended set of collation sequences. By calling `setlocale(LC_COLLATE,` *xxx*`)`, you can switch to the collation sequence defined by the locale *xxx*. (*xxx* points to a null-terminated string that names the locale.) The C Standard imposes no limit on the number of named locales that an implementation can or must support. Nor does it dictate how to specify a collation sequence. It simply describes functions in the Standard C library that enforce the collation sequence for the current locale.

The C Standard also defines limited support for large character sets. One way to represent these sets is as multibyte encodings—sequences of one or more eight-bit (or larger) codes. Another way is as wide-character encodings—fixed-size integers large enough to represent all codes in the set. You can alter these encodings, at least in principle, by calling `setlocale(LC_CTYPE,` *xxx*`)`. (What happens to multibyte and wide-character literals when you change code sets is a thorny issue.) Again, the C Standard doesn't describe the machinery for specifying code sets. The Standard C library simply includes functions that map between these representations.

The functions in question are:

- `strcoll` and `strxfrm`, declared in `<string.h>`, which map a character string to another character string, to define a collating sequence.

- `mbtowc` and `mbstowcs`, declared in `<stdlib.h>`, which map multibyte characters and strings to wide characters and wide-character strings, respectively.

- `wctomb` and `wcstombs`, declared in `<stdlib.h>`, which map a wide-character string to a multibyte string.

Collation sequences vary tremendously around the world. Try to look up a name in a Swedish telephone book and you will find that 'I' and 'J' sort interchangeably. Indeed, languages with accent marks try to outdo each other in devising esoteric collation rules. Even within a given language and culture you will find cause for variation. Compare an English dictionary, a telephone book, and the output from the MS-DOS sort utility. Each has a different notion of proper ordering rules.

The situation is only slightly better with multibyte encodings. Several encodings exist for Kanji, currently the most important of the large character sets (commercially). Larger vendors can afford to choose one set of codes and promote it. Smaller vendors are wise to write code that adapts easily to different codes.

The need to change these various mappings is very real. For widely-used mappings you may want functions tailored for size and performance. In general, however, you are better off trading space and speed for flexibility. You want some way to specify a collation sequence or a multibyte mapping that a program can digest on the fly.

Implementation

I have written a portable implementation[1] 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.

I described how I implement locales, at least in general terms, in my article in the previous issue of *The Journal.* (See "Implementing Locales," *Volume 2, number 4*, March 1991.) At the end of that article, I mentioned the problem of describing various mappings in a human-readable "locale file." I observed that a table-driven finite-state machine offers a reasonable compromise between flexibility and performance. You can specify the contents of state tables as part of a locale. The functions can execute these tables to perform a wide variety of mappings. That is why I chose to implement each of the three pairs of mapping functions as a table-driven finite-state machine.

[1]That library is now complete and will soon be published by Prentice-Hall in a book called *The Standard C Library*. The machine-readable source, over 9,000 lines of C code, is also available at a reasonable price. Refer to page 87 for more information.

It is not an easy matter to code the tables for a finite-state machine. The idea is not to bring this technology to the man in the street. Rather, I wanted people to be able to include predesigned mappings as chunks of text in a locale file. The more adventuresome may also choose to tinker. They do so at their peril.

In this article, I describe the workings of the three table-driven functions in this implementation of the Standard C library. I also show the notation you use to specify a state-table in the locale file. And I provide example locale files for several widely-used mappings.

Writing the Locale File

The three table-driven functions are necessarily different. Defining a collation involves mapping a string of characters to another string of characters. Translating multibyte characters to wide characters maps a string of characters to a string of wide characters. Translating wide characters to multibyte characters does the reverse. Other subtleties arise as a side effect of these basic differences.

Nevertheless, the functions have many similarities. For that reason, I was able to define a single table format to accommodate all three table-driven functions. (The table format also meets the needs of the `<ctype.h>` testing and mapping functions, but that is being somewhat precocious.) The behavior may vary in small ways, but the table entries have much the same meaning in all cases.

You can specify up to 16 state tables for each of the three entity names:

```
collate
mbtowc
wctomb
```

Each table has `UCHAR_MAX`+1 entries, one for each value representable by type `unsigned char`. Thus, the entry `collate[0, ’a’]` specifies how to process the character code `'a'` in state 0. A mapping starts out in state 0 with a wide-character accumulator set to a known state. The wide-character accumulator currently consists of two 8-bit bytes. It is set to zero for `collate` or `mbtowc` and to the current wide character for `wctomb`.

For each entry, you can specify:

- A mapped value to use in place of the character code,
- A successor state, and
- Four action flags.

The action flags determine what happens with a given character code in a given state. Using the notation of the locale file:

- `$F` folds the mapped value into the least-significant byte of the wide-character accumulator.
- `$R` rotates the wide-character accumulator one position to the left.
- `$O` produces output from the wide-character accumulator (`mbtowc`), from a nonzero mapped value (`collate` or `wctomb`), or from the least-significant byte of the wide-character accumulator (`collate` or `wctomb`).
- `$I` consumes the current input if it is nonzero.

Mapping proceeds until the function produces an output code of zero. Executing from a nonexistent state table or from a zero element causes the mapping to terminate with an error.

In the `"C"` locale, all three functions share a common state table zero. Tables 1–16 are nonexistent. You can specify this initial state in a locale file as:

```
mb_cur_max 1
collate[0, 0:$#] $@ $F $O $I $0
mbtowc [0, 0:$#] $@ $F $O $I $0
wctomb [0, 0:$#] $@ $F $O $I $0
```

The first line determines the value of the macro `MB_CUR_MAX`, defined in `<stdlib.h>`. The table specifications cause all three functions to perform a one-to-one mapping between input and output. The symbol `$#` has the value `UCHAR_MAX`, defined in `<limits.h>`. So the subscript `[0, 0:$#]` specifies all elements (`0:$#`) of state table zero (`0,`). Each element says:

- Fold the mapped value into the wide-character accumulator (`$F`)
- With the input code mapped to itself (`$@`)
- Write either the mapped value or the contents of the wide-character accumulator as the output (`$O`)
- Consume the input (`$I`)
- The successor state is state zero (`$0`).

Symbols `$0`–`$7` name the first eight successor states. The locale file provides no special symbols for successor states 8–15. As I indicated last month, you can write simple expressions, store values in the variables `A`–`Z` (using `SET`), and add comment lines (using `NOTE`). This hardly constitutes a major language, but it seems to suffice for specifying many locales.

Dictionary Order

Let's look at a few examples that are somewhat less trivial. The first shows the simplest version of a special collation function. Say you want to order pairs of strings by "dictionary order." Here, I'll take that to mean that only letters are important. Punctuation and spacing go away. Moreover, uppercase and lowercase letters are equivalent.

One way to define that collation is to discard everything but letters in the collation mapping. Uppercase letters map to their corresponding lowercase brethren. You get this behavior for the locale `SIMPLE_DICT` defined by the file:

```
LOCALE SIMPLE_DICT
NOTE simple dictionary collation sequence
collate[0, 0:$#   ]               $I $0
collate[0, 'a':'z'] $@         $0 $I $0
collate[0, 'A':'Z'] $@+'a'-'A' $0 $I $0
LOCALE end
```

Now let's add a refinement. Some languages require that collation proceed in two stages. If two strings compare equal in the first stage, they are checked by a different set of rules in the second stage. Accented letters, for example, may compare equal unless all letters *to the right* also compare equal. Then the difference matters.

The collate functions have a special property to permit such collations. When a table entry attempts to consume the terminating null on the input string, it doesn't just stick there (as `mbtowc` and `wctomb` do). Instead, it resets the input pointer to the start of the input string. A request for more input characters rescans the string. You can write state tables that make multiple rescans, not just one.

In the case of our dictionary sort, we can add such a second stage. If two items compare equal in the first stage, we'll compare them as raw text in the second stage:

```
LOCALE DICT
NOTE dictionary collation sequence
collate[0, 0      ] '.'        $0 $I $1
collate[0, 1:$#   ]               $I $0
collate[0, 'a':'z'] $@         $0 $I $0
collate[0, 'A':'Z'] $@+'a'-'A' $0 $I $0
collate[1, 0:$#   ] $@         $0 $I $1
LOCALE end
```

A null character now produces a dot to terminate characters generated in the first phase. The rescan occurs in state 1, which copies all characters unchanged. Visiting the terminating null the second time terminates the mapping.

You can be even more ambitious. Some name lists insist that 'McTavish' and 'MacTavish' sort equal. It takes a few more states to gobble 'Mac' and translate it to 'mc'—and to undo the damage you do to 'Mason' along the way. But it can be done. Try it.

Mapping Multibyte Characters

You have to specify multibyte and wide-character mappings in pairs. (I assume you want the transformations in both directions to agree.) They also tend to be messier than simple collation functions. I have been able to specify all the popular Kanji encodings I know about, however—at least to the extent that I understand them properly.

I assume, by the way, that the internal codes for wide characters are mine to choose. I know there is some controversy about this. Committee X3J11 has gotten flak in the past for:

- Commandeering code value zero for a terminating null character
- Using `'\n'` internally in place of carriage return and line feed externally
- Requiring that `'0'`+1 equal `'1'`
- Commandeering code value zero for a terminating null wide character
- Requiring that `L'a'` equal `'a'`

Our position on these issues is, I believe, quite defensible. Other groups can standardize what codes go in and out of a program. It is up to each C implementation to match relevant interchange standards. What goes on inside a C program, however, is the business of the C Standard and its implementors.

The machinery in this implementation of the Standard C library is pretty powerful. In many cases, you can contrive a mapping from a given set of multibyte codes to a given set of wide-character codes. But you can't always do so. I content myself, in these examples, to show that an adequate wide-character encoding exists. That's often hard enough.

One of the simplest of the popular Kanji encodings is the EUC code developed to internationalize UNIX software. I believe that DEC still uses the stripped-down version of EUC. In this scheme, a character code in the range 0xA1–0xFE signals that it is the first of a two-character code. The second character code must be in the range 0x80–0xff. Here is a locale that supports this encoding, with full checking:

```
LOCALE EUC
NOTE EUC codes with 0xA1-0xFE followed by 0x80-0xFF
SET A 0xa1
SET B 0xfe
SET S 0x80
SET T 0xff
SET X 0
mb_cur_max 2
mbtowc[0, 0:$#] $@ $F    $O $I $0
mbtowc[0, A:B ] $@ $F $R    $I $1
mbtowc[1, 0:$#]    X
mbtowc[1, S:T ] $@ $F    $O $I $2
mbtowc[2, 0:$#]  0 $F $R       $0
wctomb[0, 0:$#]       $R       $1
wctomb[1, 0:$#]    X
wctomb[1, 0   ]       $R $O $I $0
wctomb[1, A:B ] $@    $R $O    $2
wctomb[2, 0:$#]    X
wctomb[2, S:T ]          $O $I $0
LOCALE end
```

The convention here, as with the collate examples, is to first flood a state table with the most common entry value. I use `X` to signal an error state, the commonest flood value. Subsequent lines may overwrite intervals within the same state.

State 0 of `mbtowc` handles all single-character codes. State 1 puts all valid second characters. State 2 simply clears the high-order part of the wide-character accumulator before returning to state 0. The wide-character code is zero concatenated with a single character or the first character concatenated with the second.

State 0 of `wctomb` simply rotates the wide-character accumulator and enters state 1. You have to test the more-significant character to determine whether the wide character represents one character, two characters, or an error.

Shift JIS

Microsoft introduced one of the more popular encodings for the PC several years ago. Shift JIS is based on the older JIS standards for multibyte and wide-character encodings. It is somewhat simplified, however, and more restrictive. Basically, it follows the same philosophy as EUC. The first of a two-character code has values in the range 0x81–0x9F or 0xE0–0xFC. The second byte must lie in the range 0x40–0xFC. (IBM's DBCS encoding is the same, except that it excludes 0x7F as a second-byte code.) You will find its locale reminiscent of that for EUC:

```
LOCALE SHIFT_JIS
NOTE JIS codes with 0x81-0x9F or 0xE0-0xFC
NOTE     followed by 0x40-0xFC
SET A 0x81
SET B 0x9f
SET C 0xe0
SET D 0xfc
SET M 0x40
SET N 0xfc
SET X 0
mb_cur_max 2
mbtowc[0, 0:$#] $@ $F    $O $I $0
mbtowc[0, A:B ] $@ $F $R    $I $1
mbtowc[0, C:D ] $@ $F $R    $I $1
mbtowc[1, 0:$#]    X
mbtowc[1, M:N ] $@ $F    $O $I $2
mbtowc[2, 0:$#]  0 $F $R       $0
wctomb[0, 0:$#]       $R       $1
wctomb[1, 0:$#]    X
wctomb[1, 0   ]       $R $O $I $0
wctomb[1, A:B ] $@    $R $O    $2
wctomb[1, C:D ] $@    $R $O    $2
wctomb[2, 0:$#]    X
wctomb[2, M:N ]          $O $I $0
LOCALE end
```

The only significant difference is the need to specify two bands of values for the first of two characters.

Shift EUC

EUC has been extended to include many more Kanji characters. It defines two escape characters, 0x8E and 0x8F. Each can be followed by a character in the range 0x80–0xFF. My understanding is that the subrange 0xA1–0xFE signals that yet another character follows, as usual. For these escape sequences, I chose to set the wide-character code as follows:

- 0x8E followed by a single character gets a high-order code of 0x40. The low-order code is the single character code (0x80–0xA0 or 0xFF).
- 0x8E followed by two characters gets the usual code minus 0x6000.
- 0x8F gets the same code as 0x8E minus 0x80.

This scheme yields non-overlapping bands of code values. With a bit of head scratching, you can easily verify it. Its locale file is:

```
LOCALE SHIFT_EUC
NOTE SHIFT_EUC codes with 0xA1-0xFE followed by 0x80-0xFF
NOTE plus 0x8E and 0x8F shifts
```

```
SET E 0xa1
SET F 0xfe
SET S 0x80
SET T 0xff
SET W 0x60
SET X 0
SET Y 0x8e
SET Z 0x8f
SET C E-W
SET D F-W
SET M E-S
SET N F-S
SET A C-1
mb_cur_max 3
mbtowc[0, 0:$#] $@   $F    $O $I $0
mbtowc[0, Y   ]               $I $3
mbtowc[0, Z   ]               $I $5
mbtowc[0, E:F ] $@   $F $R    $I $1
mbtowc[1, 0:$#]      X
mbtowc[1, S:T ] $@   $F    $O $I $2
mbtowc[2, 0:$#]  0   $F $R       $0
mbtowc[3, 0:$#]      X
mbtowc[3, S:T ]  A   $F $R       $4
mbtowc[3, E:F ] $@-W $F $R    $I $4
mbtowc[4, 0:$#]      X
mbtowc[4, S:T ] $@   $F    $O $I $2
mbtowc[5, 0:$#]      X
mbtowc[5, S:T ]  A   $F $R       $6
mbtowc[5, E:F ] $@-W $F $R    $I $6
mbtowc[6, 0:$#]      X
mbtowc[6, S:T ] $@-S $F    $O $I $2
wctomb[0, 0:$#]         $R       $1
wctomb[0, S:T ]         $R       $5
wctomb[1, 0:$#]      X
wctomb[1, 0   ]         $R $O $I $0
wctomb[1, A   ]  Z      $R $O    $2
wctomb[1, C:D ]  Z         $O    $3
wctomb[2, 0:$#] $@+S       $O $I $0
wctomb[2, M:N ]      X
wctomb[2, S:T ]            $O $I $0
wctomb[3, C:D ] $@+W    $R $O    $4
wctomb[4, 0:$#] $@+S       $O $I $0
wctomb[4, S:T ]            $O $I $0
wctomb[5, 0:$#]      X
wctomb[5, 0   ]         $R $O $I $0
```

```
wctomb[5, A   ]  Y      $R $O    $2
wctomb[5, C:D ]  Y         $O    $3
wctomb[5, E:F ] $@      $R $O    $4
LOCALE end
```

Tracing this logic is a bit more daunting. I have executed it, on simple inputs at least, so I have reason to believe it is correct. You will find that it takes a bit of study to understand. All I will tell you is that states 2 and 4 of `wctomb` are almost identical. The former simply performs a bit more checking than the latter.

State-Dependent Encoding

I conclude with an even messier example. Another popular Kanji encoding contains "locking" shift states. In the examples so far, each multibyte character is a self-contained group of one to three characters. The older JIS encoding, however, contains escape sequences that influence any number of characters that follow. The shift state stays in effect until a subsequent escape sequence changes it.

The multibyte functions in the Standard C library are prepared to handle locking shifts. What causes problems with JIS is the nature of the escape sequences. The two most important are the sequences `ESC+(+B`, which shifts to two-character mode, and `ESC+$+B`, which shifts back to one-character mode. (I ignore others for now.) In two-character mode, both character codes must lie in the range 0x21–0x7E. Here is what you have to specify in the locale file:

```
LOCALE JIS
NOTE JIS codes with ESC+(+B and ESC+$+B
SET A 0x21
SET B 0x7e
SET X 0
SET Z 033
mb_cur_max 5
mbtowc[0, 0:$#] $@ $F    $O $I $0
mbtowc[0, 0   ] $@ $F    $O $I $1
mbtowc[0, Z   ]             $I $1
mbtowc[1, 0:$#]    X
mbtowc[1, '$' ]             $I $2
mbtowc[1, '(' ]             $I $3
mbtowc[2, 0:$#]    X
mbtowc[2, 'B' ]  0 $F $R    $I $0
mbtowc[3, 0:$#]    X
mbtowc[3, 'B' ]             $I $4
mbtowc[4, 0:$#]    X
```

```
mbtowc[4, Z   ]              $I $1
mbtowc[4, A:B ] $@ $F $R     $I $5
mbtowc[5, 0:$#]    X
mbtowc[5, A:B ] $@ $F     $0 $I $4
wctomb[0, 0:$#]       $R        $1
wctomb[1, 0:$#]    X
wctomb[1, 0   ]       $R $0 $I $0
wctomb[1, A:B ]  Z        $0    $2
wctomb[2, 0:$#] '('       $0    $3
wctomb[3, 0:$#] 'B'       $0    $4
wctomb[4, 0:$#]    X
wctomb[4, 0   ]  Z        $0    $7
wctomb[4, A:B ] $@    $R $0    $5
wctomb[5, 0:$#]    X
wctomb[5, A:B ]           $0 $I $6
wctomb[6, 0:$#]       $R        $4
wctomb[7, 0:$#] '$'       $0    $7+$1
wctomb[8, 0:$#] 'B'       $0    $1
LOCALE end
```

It takes almost 8K bytes of code tables to implement this encoding. Over 2K of those bytes simply reads and writes 'B's. You can argue that this is hardly an efficient way to do the job. I agree. If you intend to process great quantities of JIS-encoded text, you should certainly write a specialized function. It will run much faster and occupy much less space.

What this example demonstrates is that you can avoid writing such a function, if you so choose. Table-driven finite-state machines can handle a broad assortment of encodings. They do simple jobs well enough that you may never need to supplant them. They do the harder jobs well enough to show you what's worth recoding when efficiency becomes paramount.

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 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.

∞

2. A Standard C Compiler for the Transputer[2]

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

Abstract

The Inmos transputer is a family of processors featuring on-chip point-to-point communication (link) engines and hardware scheduling support, designed to be the building block for parallel Multi-Instruction Multi-Data (MIMD) computers. Both 16- and 32-bit versions exist. The 32-bit version is available both with and without an on-chip FPU. All transputers feature 2K or 4K of on-chip, fast memory. Designed for parallel processing, the transputer features a hardware scheduler and hardware process support, yet has no MMU for memory protection or virtual memory. This article mentions some of the peculiarities of the transputer, and discusses some of the problems and decisions involved in the implementation of the standard-conforming PACT Parallel C Compiler for that chip.

The Transputer

The Inmos transputer family consists of three series of processors: the 16-bit T2 series, the 32-bit T4 series, and the 32-bit T8 series with on-chip FPU. Each series offers the basic combination of a RISC core unit, some amount of on-chip 1-cycle memory, and the links for intercommunications. Only the T8 series features an on-chip FPU, which runs in parallel with the core unit. The T4 and T8 series are pin-compatible and thus easily interchangeable.

Because it requires little interface logic to hook up some ROM, and has on-chip RAM as well as links to communicate with the outside world, the 16-bit version makes for an interesting controller. Its 64K address space is often large enough to hold both code and data. The 32-bit versions with their intercommunications, parallel processing support, and optional on-chip FPU constitute powerful building blocks for parallel computers.

Many transputer applications can easily be designed to run on a variable-sized set of transputers, resulting in a scalable system. When things get too

[2]Transputer is a trademark of Inmos.

slow you simply add a couple of transputers and redistribute your processes over the new network. By checking for correct behavior and redirecting tasks elsewhere in case of failures, fault-tolerant systems can be created.

The transputer has a limited instruction set and no addressing modes. Not all instructions execute in a single cycle, and the instructions are executed from microcode. Whether it can be called RISC depends on your definition. If care is taken not to rely on particular hardware characteristics, object code can be executed on all members of the transputer family or a subset thereof (e.g., 32-bit processors only).

The transputer instruction set supports creation and manipulation of processes. Processes can execute at two levels of priority. A hardware scheduler schedules the processes in the low-priority queue in a round-robin fashion. High-priority processes can interrupt low-priority processes any time and are *not* time-sliced.

Synchronized interprocess communication is provided by channels and channel input/output instructions. The interprocessor communication links appear to the software just like ordinary channels—all communication details are handled in hardware by the link engines, using DMA.

Two timers are provided, one for each priority level. The low-priority timer is advanced every 64 μsec (15,625 times per second). The high priority timer is advanced every μsec (1 million times per second). The timers can be used to timeout certain (e.g., communication) operations. Processes waiting for the timer to reach a certain value are placed in a timer queue, again under complete hardware (and microcode) control.

Memory Management

The transputer has no hardware memory-management unit (MMU). It has no memory protection or virtual memory—just a big, flat address space. This is a bit of a problem if we're going to have a bunch of processes running, each growing and shrinking stacks and allocating heap space as necessary. We've got a real problem if these processes can be created dynamically and we can't determine the amount of stack needed for each of them in advance.

Because of the dynamic nature of this system, we have no other option than to allocate everything, including stacks, from a heap and allow stacks to grow and shrink as necessary. When a new process is started, we allocate a chunk of memory to serve as a stack. Each function prologue checks to make sure the amount of stack space that function needs, is available. If necessary, the stack is expanded by allocating a new block of memory and copying the current stack frame, including the function parameters. This new block is released again at the end of the function.

When a function decides the stack needs to be expanded, it allocates a little more than necessary. By doing so it reduces the potential need for functions called from this function to have to expand the stack again. This granularity, as well as the size of the initial stack given a new process, can be set with compiler pragmas. It is also possible to disable stack checking and expansion. This can be useful for production code once maximum stack sizes are known and fixed, and for small leaf functions. For example, most functions in the standard library are small enough to be compiled without stack checking.

A smart linker can use a call graph to determine whether to check for stack overflow, and if so, where to check. This way, stack checking can be completely eliminated. An initial stack of exactly the right size is allocated if the process is non-recursive and doesn't make any indirect calls.

Signed Pointers

On the transputer, pointers are signed integers. This allows the faster signed integer instructions to be used with them. This means that the transputer's address range extends from `INT_MIN` to `INT_MAX` (0x8000–0x7FFF on the T2 series and 0x80000000–0x7FFFFFFF on the others). The transputer's internal memory is mapped starting at address `INT_MIN`. A number of words at the very start are reserved for various processor control functions. This poses the problem of what to do with `NULL`. Address 0 is right in the middle of the address range, and on the 16-bit T2 series it's quite possible that real memory is mapped at this address.

Note, however, that the 32-bit T4 and T8 series transputers are not likely to have memory at address 0. If address space is to be mapped contiguously to real memory, address 0 would point into real memory only if more than 2GB were installed. Although not impossible, it is very unlikely for a node in a parallel machine to contain this much memory. If memory is scattered over the address range, it is easy enough to avoid mapping to address 0.

The transputer itself uses `INT_MIN` as its nil pointer to terminate the various queues it maintains. However, the problem with using anything other than 0 for `NULL`, despite X3J11's efforts to permit this, is that there is software out there that assumes that all the world's a VAX (for example), and a `NULL` pointer is represented by a zero-bits pattern. (The problems of arrays of pointers allocated with `calloc` or initialized with `memset` come to mind.)

Therefore, we decided to define `NULL` as 0 and take care to never allocate anything at that address. This involves a straightforward check in `malloc` and related functions. This is hardly a concern for the 32-bit T4 and T8 series, which are unlikely to have memory at address 0 anyway. It is deemed acceptable on the 16-bit T2 series.

Expression Evaluation

The transputer has but a few registers. The most important ones are:

Iptr – The instruction pointer (program counter)

Wptr – A pointer to the current workspace (stack frame)

Areg, Breg, Creg – Core expression-evaluation stack

FAreg, FBreg, FCreg – FP expression-evaluation stack (T8 series only)

The expression-evaluation stack consisting of Areg, Breg, and Creg is used to evaluate expressions and pass arguments to functions. By loading something in the Areg, the old values of Areg and Breg are moved to Breg and Creg respectively, and the old value of Creg is lost. Similarly, popping the Areg moves the old values of Breg and Creg to Areg and Breg, respectively, and loads Creg with an indeterminate value. Transputer instructions get their operands (up to three for some instructions) from the stack and place their results back there.

The T8 series has a similar evaluation stack for floating-point operands. Each stack register can hold either a single- or double-precision value, and is tagged as such. This operand tag is used by the FPU to determine whether to operate in single- or double-precision.

One of the advantages of this small register set is that it results in a fast context switch, which is very important for a processor running many processes in parallel. (Hundreds of processes is normal.) Low-priority processes can only be descheduled at particular points, at which time the evaluation stack is empty. So the typical context switch has no registers to save except the Iptr and Wptr. (A high priority process can interrupt a low-priority process at any time. It does save and restore the evaluation stack(s).)

Efficient code generation for this evaluation stack requires finding a sub-expression-evaluation ordering that allows subexpression results to be kept on the stack as much as possible. The PACT Parallel C Compiler uses DAGs (Directed Acyclic Graphs) for the intermediate representation of a basic block. In addition to allowing a number of interesting optimizations, DAGs also map quite easily to the transputer's stack architecture. When generating transputer code for the DAG, the compiler walks the DAG twice. The first pass attributes each DAG node with stack requirements: the number of core and FP stack registers needed to evaluate the node, and the number of stack registers left on the core and FP stacks after the evaluation. On the second pass, actual transputer assembly code is generated. The stack requirements information is used to determine when to spill values to memory.

It turns out that many basic blocks can effectively be evaluated on the evaluation stack without requiring any spills at all. Function calls are big spoilers, however. They require everything on the stack to be saved and later restored. We've found the evaluation stack to be relatively easy to generate good code for, and to give satisfactory performance.

The first few words of function arguments are passed on the evaluation stack. Functions return their results on the evaluation stack, ready for further use. Inline assembly is provided by a pseudo-function with the following prototype:

```
        void __asm (const char *s, ...);
```

The first argument is emitted verbatim to the assembler file. The other arguments, if any, are evaluated as if they were function arguments. This makes it very easy to use C expressions as the arguments for transputer instructions. If a transputer instruction returns a value, `__asm` can be cast to a function returning an object of the appropriate type and the result can be used as any other value returned by a function. This way, function-like macros can be defined that expand to inline transputer assembly, without the user being aware of it. For instance, the `clock` function has been implemented as follows:

```
        #define clock() (((clock_t (*) (const char *)) __asm) \
                ("ldtimer"))
```

Parallel Processing

The transputer is meant to be used in parallel computers. As such, it has hardware support for process creation and control, as well as for interprocess communication. Interprocessor communication is simply a special case of interprocess communication, largely transparent to the programmer. A typical transputer program consists of a large number of processes, dynamically created and terminated, distributed over a number of processors.

Although parallel processing support could be provided through a library containing functions to start processes, engage in communication, etc., we felt it was absolutely crucial to provide the proper level of abstraction necessary to write complex programs consisting of many communicating processes. Therefore, we decided to extend the C language with constructs to spawn child processes and to engage in communication. These constructs compile directly to transputer instructions and suffer no function call overhead. These extensions are described elsewhere[3].

The most important new construct is the one that starts child processes. This construct, in its most complex form, looks like this:

```
        par ( exp-1 ; exp-2 ; exp-3 ) {
                statement-1
                statement-2
                ...
                statement-n
        }
```

The parenthesized expressions following the `par` keyword are collectively called the *replicator*, and look just like the controlling expressions in the C `for` construct. Each expression in a replicator is called a *replicator expression*. All

[3]For a description of these parallel extensions refer to *A Parallel Extension to ANSI C* by the same author in *The Journal Volume 2, number 4* page 267.

variables that occur in the replicator expressions are called replicator variables. The construct above is called a *replicated* `par`. Each statement in the replicated `par` body is started as a child process. The processes 1–n are collectively called a *process group*. The replicator serves to start every process in the process group once for every iteration through the replicator. For every replication, the process group gets a unique copy of all replicator variables. For example, to start 5 processes calculating and displaying x^2 and 5 more calculating and displaying x^3 for $x = 1$ to 5, the following could be used:

```
par (i = 1; i <= 5; i++) {
        printf("%d**2 == %d\n", i, i*i);
        printf("%d**3 == %d\n", i, i*i*i);
}
```

In this example the process group consists of two processes each doing a simple calculation and printing the result. The replicator variable `i` is copied to each process group and is shared between the two processes in each process group.

For each process group started in a replicated `par` a block of memory, allocated from the heap, contains the stacks for all processes in the group. This same block of memory contains the instantiation of the replicator variables for this process group, as well as a pointer back to the parent process' workspace. Inside processes the normal C scope rules apply. That is, each process has access to any and all variables that are defined in enclosing scopes. Access to variables in parent processes involves traversing the dynamic chain. In the following example, variable `ch` resides in the parent workspace and can only be accessed by traversing the chain:

```
ch = getchar();
par (i = 0; i < 10; i++) {
        process_char (ch, i);
}
```

Another language extension is the addition of the channel type. This allows channels to be declared as simply as other variables, and allows for simple and efficient interprocess communication. Using and setting an operand with channel type translates directly into channel input and output transputer instructions.

Miscellaneous Issues

The transputer has no interrupts (except for a high-priority process interrupting a low-priority process) or exceptions. All error signalling is done with one single error flag. This flag is set after (signed) integer overflow and a few special test instructions (e.g., check if array index is within bounds). The error flag can be used to trap programming errors in a number of ways. The first is to simply check the flag after certain operations to see if the flag got set. If so, do whatever is necessary to fix things. It is also possible to stop the process when the error flag turns out to be set, allowing for somewhat graceful system degradation as processes encounter errors. Meanwhile, the transputer keeps executing its other processes. Finally, it is possible to configure the transputer to halt when the error flag gets set. This is mostly useful for debugging purposes or in certain fault-tolerant systems.

The compiler has to support the various ways in which the error flag may be used. It may have to insert checks after certain transputer instructions automatically to detect errors as soon as possible and raise a software signal. If the error flag is not used at all it is possible to replace some checked instructions (instructions that set the error flag on overflow and similar conditions) by faster, unchecked instructions. The PACT Parallel C Compiler has five error flag modes:

1. Don't pay any attention to error flag.
2. Be sure not to set the error flag erroneously.
3. Check error flag after certain operations and call exception handler.
4. Check error flag after certain operations and stop process.
5. Halt processor as soon as error flag is set.

The T8 series transputers have a similar flag to detect FP errors. This FP error flag must be explicitly tested, and may be used in the same way as the core error flag.

Processes running at low-priority can be descheduled at certain instructions only. The instructions that cause descheduling are the unconditional jump and the loop instructions, plus a few special parallel processing support instructions. These instructions normally occur at least once in each loop, guaranteeing that no process can loop forever and consume all processor cycles. Sometimes, however, it may be necessary or advantageous to make sure a particular piece of code will not be descheduled. The two ways to do this are to run it at high-priority, or to make sure the code does not contain any descheduling instructions. The PACT Parallel C Compiler supports a compilation mode where every unconditional jump is replaced by a conditional jump preceded by an instruction to ensure the conditional jump is always taken. This is slightly slower, of course, but allows code to be made slightly more atomic (high-priority processes can always interrupt the low-priority process) without resorting to running it at high priority.

Conclusion

The transputer is somewhat of a strange beast. Its unique parallel processing capabilities provide a powerful parallel programming platform but also resulted in such peculiarities as signed pointers, no memory protection, an evaluation stack instead of registers, etc.

The stack checking and expansion system, although expensive, is absolutely crucial for parallel program development. Especially on the transputer, where stack overflows are not in any way trapped or signalled. They just overwrite some other process' data.

The expression evaluation stacks are relatively easy to generate good code for and allow for minimal context switch delays.

It has turned out to be possible to provide a standard-conforming C compiler that also gives access to the transputer's unique parallel processing capabilities in an intuitive manner. The language extensions maintain the spirit of C quite well, and the resulting Parallel C language provides a powerful programming tool.

The current generation of transputers basically stems from 1986 and, as such, doesn't perform too well compared with todays RISC processors. The next generation of transputers, the T9000 series, is expected to become available later this year or early next. The T9000 will be a superscalar architecture performing at some 200 MIPS and 25 MFLOPS peak (115 MIPS, 18 MFLOPS sustained). It will also be object code compatible with the current generation of transputers. It will have some form of memory protection, 16KB of on-chip cache, and faster and better interprocessor communication. This should make the T9000 a very interesting processor indeed.

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.*

∞

3. NCEG Progress Report

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

Abstract

The Numerical C Extensions Group (NCEG) was formed at a meeting in May 1989. Since then there have been four more meetings, the most recent of which was held at Norwood, Mass., in March 1991. A major goal of this meeting was to identify subcommittees that could produce documents for public comment by the end of the January, 1992 meeting. We also discussed issues which resulted from our liaison with X3J11, ISO WG14, X3J16, X3H5, IFIP/WG 2.5, and X3T2. In this article I report on the outcome of that meeting and on the status of NCEG issues in general.

Introduction

The fifth meeting of the Numerical C Extensions Group (NCEG) occurred March 4–5, 1991. It was hosted by Analog Devices at their facility in Norwood, Mass. The primary purpose of the meeting was to continue refining the proposals of the active subcommittees. The subcommittees presenting at this meeting were:

Subcommittee	*Primary Contact*	*Secondary Contact*
Aliasing	Bill Homer	Tom MacDonald
Array Syntax	Frank Farance	Tom MacDonald
Complex	Tom MacDonald	Bob Allison
Extended Integers (new)	Randy Meyers	
Floating-point Extensions	Jim Thomas	Dave Hough
Variable-length Arrays	Dennis Ritchie	Tom MacDonald

Considerable agenda time was devoted to liaison activity, especially for the Language Compatible Arithmetic Standard (LCAS) being produced by X3T2, and the C language binding to the parallel execution model being developed by X3H5 (formerly the Parallel Computing Forum). Formal presentations were made by representatives from X3T2 and X3H5.

The primary goal of committee NCEG is to produce a professional, high quality Technical Report (TR) that defines extensions to C such that C will be more attractive for numeric and scientific programming. Recently X3 voted to accept the project proposal for NCEG and approved the creation of the technical committee X3J11.1, a working group within X3J11. Since NCEG is now an official ANSI working group, our procedures will have to be a little more formal. For example, only paid-up X3J11 and X3J11.1 members will be allowed to cast formal votes. In the past we have striven for consensus to resolve issues. Producing a TR for public review requires that we make decisions about issues that are currently unresolved. Most likely this will happen by a two-thirds majority vote. We have also been recognized by the ISO C committee WG14, so NCEG has agreed to provide that committee with copies of relevant documents.

Liaison Activity

Committee X3H5

NCEG provided several hours of agenda time to Bob Gottlieb (HP-Apollo's representative to X3H5) so that he could present their proposed C binding. This talk was intended to both educate NCEG about X3H5's current status and to solicit input from NCEG and X3J11 about how to make the proposed language binding better. This prompted NCEG to reactivate its parallel subcommittee in order to collate responses to X3H5.

The X3H5 parallel-programming model is too extensive to cover in detail here. The following is a very brief and simplified overview of their model.

The model is based on a shared-memory architecture rather than on a distributed-memory. A parallel program begins executing with a single *process* (just like a serial program) called the base process. When the base process encounters a *parallel construct*, a *team* of processes is formed. Inside a parallel construct all code is executed by every process in the team. Typically there is at least one *work-sharing construct* inside a parallel construct that provides the opportunity to distribute the work among the processes in the team. All processes are *blocked* from leaving the work-sharing construct until all processes have finished their share of the work. When all have finished, they continue executing the code following the end of the work sharing construct. This pattern continues until the end of the parallel construct is encountered. Again, processes are blocked until all have finished executing their code inside the parallel construct. Then the base process continues executing in a serial fashion until another parallel construct is encountered. Nested parallelism occurs when a process executing inside a parallel construct encounters another parallel process. At that point the process becomes a base process for the nested parallel construct.

The set of data objects accessible to a process is called the *data environment* for that process. An object that can only be accessed by a single process is *private* to that process. An object that can be accessed by two or more processes is *shared* between the processes. *Synchronization* is provided to control access to shared objects and coordinate execution among processes.

The language binding specifies new syntax (where required) and semantics for all of the concepts in the execution model. The C binding is very new and many issues are still being addressed. The following is a brief overview of that binding:

New keywords are proposed. They are:

`parallel` – Identifies the beginning of a parallel construct. The C punctuators { and } delineate the range of a parallel construct.

`pfor` – Identifies an *iterative* work-sharing construct. This is intended to be similar to a C `for` loop except that each iteration is a unit of work that can execute in parallel.

`pstatements` – Identifies a non-iterative work-sharing construct. Each statement inside the `pstatements` is a unit of work. The following fragment identifies three separate units of work:

```
pstatements { /* work1, work2 and work3
                 can execute in parallel */
   work1();
   {  /* work2 start */
      inner1();  /* inner1 must complete */
      inner2();  /* before inner2 begins */
   }  /* work2 end */
   work3();
}
```

`pwait` – Implements the non-iterative synchronization between units of work. The operands of `pwait` are statement labels. For example:

```
pstatements { a:f1();  b:f2(); pwait(a,b); f3(); }
```

`critical` – Implements the *critical section* notion of the model. A critical section is a common concept within the field of parallel programming. However, I could not find any definition for it in the model document.

`stkextern` – This appears to be implementing some notion of sharing stack data across compilation units by name. I am convinced that this idea needs closer examination. The C language binding specification states:

> "Any place that `extern` is used in the C standard should be interpreted [as] referring to `stkextern` also."

Certainly this is intended to be confined to the declaration of data objects and not the declaration of functions. The C standard requires a single definition without the `extern` keyword to exist in some compilation unit. It is not clear if there is a similar mechanism used to actually define the `stkextern` object.

`lock`, `event`, `counter` – These are type-specifiers. Objects declared to be of these types are not permitted to be initialized.

There are also a number of new library routines:

```
int init_lock ( lock );
void lock ( lock_name, LOCK_WAIT );
int lock ( lock_name, LOCK_NOWAIT );
void unlock ( lock_name );
int is_lock ( lock_name );

int init_event ( event );
void post_event ( event );
void clear_event ( event );
void wait_event ( event );
int is_posted ( event );

int init_ctr ( counter, iv, inc );
void post_ctr ( counter, value );
void wait_ctr ( counter, value );
long value_of_ctr ( counter );
```

where *iv* is the initial value of the arithmetic sequence, *inc* is the increment of the arithmetic sequence, and *value* is one value of the arithmetic sequence.

My initial reaction to this proposal is they do *not* need so many new keywords. For instance, the new type-specifiers could be `typedef` names defined in a new header (e.g., `<parallel.h>`) and called `lock_t`, `event_t`, and `counter_t`, respectively. (This is similar to the object type `fpos_t` defined in the header `<stdio.h>`.) Otherwise, these names will only unnecessarily usurp the user's name space.

It also seems that other keywords, such as `pwait` and `critical`, might be disguised as macros. There seems to be a problem with having two functions named `lock`, one returns `void` and the other returns an `int`. And I'm not convinced that `stkextern` is necessary at all, however, I must admit to not understanding all the issues being addressed by this new keyword either.

Again I should stress that the proposed C binding is very new and this is just an initial glance. Their execution model is intriguing. A good language binding will provide a nice parallel environment for shared memory machines. I look forward to following their progress.

Committee X3T2

NCEG was fortunate to have Martha Jaffe attend our meeting and talk to us about X3T2's efforts to create a "Language Compatible Arithmetic Standard" (LCAS) across architectures and languages. Some of the highlights of LCAS follow.

LCAS is incompatible with Standard C because LCAS requires `sqrt` to raise a visible exception if its argument is a negative number. Standard C defines the (somewhat controversial) `errno` behavior which explicitly states that no visible exceptions are raised when `sqrt` executes. The `sqrt` function is the only function defined in the `<math.h>` header that is addressed by the LCAS.

The LCAS is intended to be a software specification. This is in contrast with the IEEE floating-point standard which is primarily a hardware standard (although it can be implemented in software). However, there are a few areas in the LCAS specification that dictate hardware support for an efficient implementation. For instance, the LCAS requires notification of operations that produce integer overflow. *[Ed. In practice, this requirement is more of a problem for LCAS and C than the similar one for* `sqrt`*. A separate LCAS-oriented math library would suffice for* `sqrt`*, while notification on integer overflow is a direct language change and is difficult to implement efficiently on many popular architectures.]* Another issue is that the floating-point accuracy requirements rule out architectures that do not use a guard bit (e.g., Cray Y-MP). Only the signed integral types `signed int` and `long int` can conform to the LCAS. All of the other integral types do not conform. All three floating types `float`, `double` and `long double` can conform.

Committee X3J16

Committee X3J16 (C++) has inquired about any possible conflicts with the proposed `<complex.h>` header as a header by this name also exists in C++ implementations. C++ headers are processed differently than C headers due to issues with name mangling and overloaded functions. For this reason C++ headers often reside in a different directory than C headers. Therefore, there appears to be no obstacle to having `<complex.h>` in both standards some day. It should be noted that the proposed NCEG approach to complex arithmetic is different from the C++ overloaded operator and function approach. *[Ed. This divergence may be problematic to those vendors that provide a combined C and C++ programming environment.]*

Variable Length Arrays

There are two competing proposals for adding Variable Length Arrays (VLA) to C. One proposal is described in an article *Variable-Size Arrays in C* by Dennis Ritchie in the September, 1990 issue of *The Journal*, but was not presented at this meeting. (Ritchie's proposal is *not* discussed in this article.) The other proposal (developed by Steve Collins at Cray Research) was presented at this meeting. Cray's proposal is similar to the GNU C implementation and is described in an article *Variable Length Arrays* in the December, 1989 issue of *The Journal*. The following is a list of issues raised with the Cray Research, Inc., (CRI) proposal at the March meeting.

VLAs as Structure Members

The first discussion revolved around allowing structure and union members to have VLA types.

Question: Should there be some way to declare a VLA member?

Yes: 11 No: 1 Undecided: 4

There seems to be considerable sentiment among NCEG members for adding this feature. Two possibilities were briefly discussed:

```
Within Structure           Outside Structure

                           int n = 5;
struct tag {               struct tag {
   int n;                     int m;
   double a[n];               double a[n];
} x = { 5 };               } x;
```

The committee was much less decisive on which of these approaches was preferred.

Question: Which method would you prefer?

Within Structures: 6 Outside Structures: 3 Undecided: 7

A comparison of both approaches is needed in order to determine which method is best.

The `sizeof` Operator

The `sizeof` operator does not produce a compile-time constant when its operand has a variable length array type. This seems to be generally acceptable.

Compatible VLA Types

The following are all compatible types:

```
double a[m][n];
double b[x][y];
double c[6][n];
```

implying that the following pointer to a two-dimensional array of `double`s

```
double (*p)[m][n];
```

is assignment compatible with the addresses of all three arrays:

```
p = &a; p = &b; p = &c; /* all acceptable */
```

Of course there is a requirement that the execution-time values of `x` and `m` must both be `6`, and `n` and `y` must have the same value. No objections were raised about these semantics.

Local `static` VLAs

Although there is no issue with prohibiting local `static` VLAs such as:

```
int n = 12;

{
   static a[n];   /* error */
}
```

there was sentiment for allowing local `static` pointers to VLAs as follows:

```
int x[12];
int n = 12;

{
   static (*p)[n] = &x; /* OK */
}
```

with `n` being captured each time the block is entered but with the value of `p` being initialized to the address of `x` once only, at program startup. The proposal will be changed to reflect this new behavior.

Lexical Ordering

The lexical ordering issue is by far the most controversial. The following example is used to explain the issue. Which `n` does the VLA declaration of parameter `a` bind to?

```
int n = 5;

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

According to the Cray Research proposal, the VLA declaration binds to the formal parameter that is lexically after parameter `a`. This is controversial because it forces two passes over the parameter declarations. The first pass identifies all parameters that are VLA declarations and assigns them a *universal* type that allows the VLA expression to be parsed. The second pass binds the identifiers found in VLA expressions to their definitions. This permits an identifier in a VLA expression to be defined lexically after that expression.

The semantics are defined this way for the benefit of the programmer. The proposal maintains that programmers should not be burdened with worrying about the order in which parameters are specified. There are no other cases where the parameter order is important. The lexical ordering issue only exists for parameters because of the new prototype syntax introduced into Standard C. If programmers are burdened, they may write them with old style declarations such as:

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

where the lexical ordering is not a problem. Another opinion is that programmers will learn to write their programs in the lexically correct way.

Enumeration Constants vs. Formal Parameters

The lexical ordering issue raises sub-issues such as the following:

```
enum { n = 5 };

void f(int a[n][n], int n) {/* well defined in Std C */
    /* ... */
}
```

This example is well defined in Standard C because the enumeration constant `n` is used in the declaration of parameter `a`. Therefore, the rule is, "If an identifier is a constant it binds in the first pass over the parameter list." This means that in the following example:

```
enum { r=3, s=4, t=5 };

void f(int n, float x[][n+t], int t); {/* enum used */
    /* ... */
}
```

the parameter `x` uses the enumeration constant `t` (bound in the first pass) and not the lexically following parameter.

VLAs in Function Prototypes

With the current proposal, all of the following prototype declarations are compatible.

```
extern int n;
void f(int a[n]);   /* ‘n’ is in scope */
void f(int a[*]);   /* ‘*’ means VLA   */
void f(int a[x]);   /* no ‘x’ in scope */
extern int x;
```

The issue here surrounds the use of `x` before it is declared. Should this be an error? Some members of the committee felt that the `[*]` syntax or a visible identifier should be required; otherwise, it is an error. The implication of this change is that two passes will be required over function prototype declarations (i.e., not a definition) with VLA parameters to determine if the identifiers in the VLA size expression are visible.

```
void f (int a[n][n], int n);   /* OK */
void g (int b[n][n]);          /* error */
```

The current proposal only requires two passes over a function prototype definition (i.e., with a function body) and not a simple prototype declaration. Identifiers that are part of the size of a VLA expression in function prototype declarations are assigned the universal type, just as they are in function prototype definitions. However, no second pass is made over the parameters in the prototype declaration because they go out of scope at the end of the prototype. This new requirement forces two passes over a simple function prototype declaration just to issue these new error messages. In general the `[*]` seems to be the most widely accepted and least controversial syntax.

Question: Which notation is preferred to denote a VLA parameter (in a prototype)?

Arbitrary `[x]` *(where* `x` *can include undeclared names): 0*
`[*]`*: 11   Undecided: 3*

Type Definitions

The committee agreed that the size of a VLA type does not change if a variable used to define the type is modified.

```
void f(int n) {
   typedef int A[n]; /* A has size n, n evaluated now */
   n++;              /* Does not affect type of A */
   {
      A a;
      int b[n];      /* b and a have different sizes */
   }
}
```

Initialization

There was no disagreement with disallowing initialization of VLA objects.

```
extern int n;

main () {
   int a[n] = { 1, 2 }; /* error - can't init VLA */
   int (*p)[n] = &a;    /* OK - p is a scalar */
}
```

Bypassing VLA Declarations

Another issue involves entering a block and bypassing the VLA declaration. The current proposal states that this is undefined behavior if the VLA object is referenced. These semantics are generally agreed upon.

```
void f( int n ) {
   int j = 4;
   goto L3;     /* error - bypassing VLA declaration */
   {
      double a[n];
      a[j] = 3.14;
   L3:
      a[j-1] = 0.1;
      goto L4;  /* OK - stays within scope of VLA */
      a[2*j] = 1.4;
   L4:
      a[j+1] = 1.4;
   }
      goto L4;  /* error - bypassing VLA declaration */
}
```

Another possible way of bypassing a VLA declaration is with a `switch` statement. Of course `setjmp` and `longjmp` provide their own unique problems with leaving dynamically allocated memory around. If a `longjmp` causes abnormal termination of a block with an allocated VLA, memory may be lost.

Floating-Point Extensions

New Relational Operators

The new relational operators, `!<>=`, `<>`, `<>=`, `!<=`, `!<`, `!>=`, `!>`, and `!<>` are in the current proposal principally to support NaN-cognizant comparisons. (The NaN concept is part of the IEEE Floating-point standard and stands for "Not a Number.") Applying comparison operators to floating-point values requires thinking about the NaN case. Prior to IEEE arithmetic, two numbers always compared greater than, equal to, or less than each other. The invention of NaNs introduces the concept of unordered comparisons. The following table shows their effect both in the number of operators and comparison types. (The first six operators are from Standard C. The rest are NCEG inventions. In the table, T represents True and F is False.)

Relational and Equality Operators				
	Less Than	*Equal To*	*Greater Than*	*Unordered*
`<`	T	F	F	F
`<=`	T	T	F	F
`>`	F	F	T	F
`>=`	F	T	T	F
`!=`	T	F	T	T
`==`	F	T	F	F
`!<`	F	T	T	T
`!<=`	F	F	T	T
`!>`	T	T	F	T
`!>=`	T	F	F	T
`<>`	T	F	T	F
`!<>`	F	T	F	T
`<>=`	T	T	T	F
`!<>=`	F	F	F	T

Since a NaN does not even compare equal to itself, it is possible to use something like the following:

```
a != a || b != b || a <= b
```

instead of the operator `!>`. However, the new operators are provided because such conditional expressions are too awkward.

Widest-Scanned Semantics

There was a good deal of discussion about the meaning of *widest scanned*. The idea behind this is to capture as much precision as the widest type present. Therefore, if the type of one operand increases in range or precision then all operands should increase, including floating-point constants. For example, assume widest-scanned semantics are in effect for the following example:

```
float f1;
float f2;
float f3;

f1 = f1 + f2 + f3;   /* all arithmetic done as ‘float’ */
```

Now if the declaration of `f3` changes to:

```
long double f3;
```

and widest-scanned mode is in effect, then all of the operations are evaluated in `long double` arithmetic including `f1 + f2`. The following operators terminate the widest-scanned semantics: function call, cast, relational, equality, logical AND, logical OR, assignment, conditional, `sizeof`, and comma. A question was raised about the semantics of embedded assignments. In the following example:

```
f1 = f1 + f2++ + f3;
```

it is not clear how `f2++` is evaluated since assignment operators terminate widest-scanned semantics. One possible interpretation is that `f2` is updated with a `float` value but the value of the entire expression `f2++` is evaluated as a `long double`.

Floating-point Environment

There was some sentiment for allowing statements such as:

```
y = x * 1;
```

to simply be optimized to:

```
y = x;
```

even if `x` might be a signaling NaN. The issue is that the multiply operation causes the signal to be raised while the simple memory reference need not. The motivation for this is most programs do not need strict signaling NaN semantics.

Another optimization issue involves examples such as:

```
void f ( ) {
   void g(void);
   double x = 0.1;
   double y = 0.1;
   double z;

   z = x + y;
   g();
   z = x + y;
}
```

The two `x + y` expressions cannot be considered as common subexpressions by the compiler because the call to function `g` might change the rounding modes and produce a different answer, or might check for an inexact computation. The proposed solution to this problem is to allow a `#pragma fenv_access off` or `#pragma fenv_access on` to be placed prior to the definition of `f`. This flag tells the compiler that `g` (or some function invocation within `g`) may examine or change the floating-point environment (`on`), or must not examine or change the floating-point environment (`off`). For instance, if `g` changes the rounding mode then both `x + y` expressions may produce different results including the raising of exception flags. The directive `#pragma fenv_access off` does not forbid `g` from temporarily changing the environment but it must change it back before returning control to `f`. The proposed default of `#pragma fenv_access off` is counter to the strict letter of the combined Standard C and IEEE FP requirements.

New Math Functions

Although their definitions are not complete the following library functions are expected to be added:

`acosh`	`aerf`	`atanh`	`erfc`	`hypot`	`max`
`acot`	`annuity`	`compound`	`erf`	`lgamma`	`min`
`acsc`	`asec`	`cot`	`expm1`	`log1p`	`sec`
`aerfc`	`asinh`	`csc`	`gamma`	`log2`	`solve`

There is some question about the behavior of `max` and `min` if NaNs are present since they have no notion of order. One proposal is to just ignore them. If all of the operands are NaNs then both `max` and `min` return a NaN. Another way to view this is to say that a NaN is minimal for the `max` function and maximal for the `min` function.

NaN Names

The names for NaNs are still open to debate, but the draft document now uses `NAN` and `nan` as prefixes for quiet NaNs, and `NANS` and `nans` as prefixes for signaling NaNs, uniformly.

Array Syntax

The Array Syntax subcommittee is still unable to converge on a single approach. There does seem to be a focus, though, on only two approaches. The first approach is based largely on the C* language as developed by Thinking Machines Corporation. A major enhancement provided by C* is a way of declaring parallel data. For example:

```
shape [512][512]Ashape;
```

uses the new keyword `shape` to define a parallel shape named `Ashape`. The declaration of a parallel shape is similar to an array except that the dimensions are specified to the left of the name. Parallel variables can then be declared with that shape.

```
double:Ashape p1, p2, p3;
```

This declares three parallel variables `p1`, `p2`, and `p3` each having the shape `Ashape`. Parallel variables can be used in ways similar to arrays through the use of left-indexing:

```
[2][2]p1 = [3][3]p2;
```

but also provide access to the whole parallel variable. This makes parallel variables first-class objects because an operation is performed on every element of the parallel variable.

The new keyword `with` specifies the current shape for parallel data execution within the `with` body. For example:

```
with (Ashape) {
   p1 = p2 + p3;
}
```

The next statement causes every element of parallel variable `p3` to be added to corresponding elements of parallel variable `p2` and the result is assigned to `p1`.

The array syntax proposal presented by Cray Research has similarities to that in Fortran-90. However, there is no access to whole arrays without the `[;]` syntax. The following example parallels that of C* above:

```
typedef double A[512][512];
A p1, p2, p3;

{
   p1[;][;] = p2[;][;] + p3[;][;];
}
```

The Cray Research proposal does not add any new declaration syntax or statements but does force the usage of the `[;]` syntax to gain access to whole arrays. This is a fundamental difference between the two proposals. A vote was taken to determine the interest in declaring parallel data.

Question: Should a distributed memory architecture model be part of the array syntax proposal?

Yes: 13 No: 1 Undecided: 3

Question: For an explicitly declared parallel object, should parallel accesses be implicit (e.g., `name`*, not* `name[;]`*)?*

Yes: 9 No: 5 Undecided: 4

There was also some support expressed for documenting both approaches to array syntax in the TR.

Aliasing

At the initial NCEG meeting, aliasing was cited as the highest priority issue. The solution being explored to the aliasing problem in C introduces the concept of a *restricted pointer*. For the most part the restricted pointer solution to the C aliasing problem seems to be fairly well received. However, there are some severe reservations being expressed by a few individuals who feel that it is inappropriate to solve the aliasing problem by enhancing C's typing mechanism.

The motivation and a more complete description of restricted pointers is presented in the article *Restricted Pointers* in the December, 1990 issue of *The Journal*. Essentially, a new keyword `restrict` is introduced that can be used to qualify a pointer declaration.

```
double *restrict p;   /* p is a restricted pointer */
```

The concept is that the compiler may treat a restricted pointer, for the purpose of alias analysis within its scope, as if it points into a unique entity. This is intended to be in the same sense as an array being a unique entity, or a call to `malloc` returning a pointer to a unique entity. This unique entity is called an *associated array.* The compiler is at liberty to assume that a file-scope restricted pointer points into a unique associated array (as if it were allocated by `malloc`) upon entrance to `main`. If the restricted pointer has block scope then for each execution of the block the compiler may assume it points into a unique associated array that is allocated upon entrance to the block. The assumption that a restricted pointer points into an associated array is only used by the compiler to determine that two lvalues reference disjoint objects. Of course an implementation is at liberty to ignore all assumptions implied by the use of the `restrict` keyword.

Restricted Formal Parameters

The formal definition allows the compiler to assume that a restricted pointer points into an associated array that was allocated upon entrance to the block. This means the compiler can assume that a formal parameter that is a restricted pointer cannot point into the same array as a file scope unrestricted pointer when the function begins execution. This is a change from the last NCEG meeting. This is a direct result of the formal definition and has the benefit of allowing more optimizations. There were some objections raised to this implicit disjoint nature between a formal parameter that is a restricted pointer and a file scope unrestricted pointer. In defense of the formal definition, it is easy to understand, it is consistent, and it is quite concise.

New Syntax

The current proposal introduces new syntax for formal parameters that are restricted pointers:

```
void f( int x[restrict][100] )
```

declares parameter `x` to have type "restricted pointer to an array of 100 `int`s." This is equivalent to:

```
void f( int (*restrict x)[100] )
```

which contains the cryptic "`(*restrict x)[100]`" pointer declaration. There were no objections to this new syntax. All in all there seems to be some convergence on restricted pointer semantics.

Extended Integer Range

There is a new subcommittee studying extended integer ranges. Some vendors provide a `long long` integral type. NCEG has formed a new subcommittee to study its effect on C. The motivation for a `long long` type is to provide a 64-bit integral type. An argument can be made that most implementations can use the following mapping:

`short`	16 bits
`int`	32 bits
`long`	64 bits

but there is concern that there are too many existing applications that assume type `long` is 32 bits. (I still remember when someone asked X3J11 to standardize the size of a word at 16 bits.) Some of the areas of the C standard that are affected are:

- New `<limits.h>` names: `LONGLONG_MIN`, `LONGLONG_MAX`, `ULONGLONG_MAX`.
- Suffixed constants with type `long long` such as `3LL`.
- Suffixed constants with type `unsigned long long` such as `4LLU`.
- Larger, optionally suffixed constants such as `9223372036854775807U`.
- Usual arithmetic conversions.
- Controlling expression of `switch` having `long long` type.
- Preprocessor `#if` expressions (currently evaluated as `long` or `unsigned long`).
- Library issues concerning `*printf`, `*scanf`, `strtoll`, `strtoull`, `llabs`, `lldiv_t`, and `lldiv`.

Complex Arithmetic

The complex proposal is rapidly approaching closure and there are no serious issues left to be resolved. The committee accepted three new macros for creating complex numbers: `CMPLXF`, `CMPLX`, and `CMPLXL` that create complex values with types `float complex`, `double complex`, and `long double complex`, respectively.

The committee also accepted a change to the usual arithmetic conversions that allows a scalar floating operand of the operators `/`, `*`, `+`, `-`, `==`, and `!=` to remain a scalar type even if the other operand has a complex type. The motivation for this change is that promoting a scalar operand to a complex type introduces a zero imaginary part that sometimes leads to a different result than is expected. For example:

$$\begin{array}{ll} & (3.0,\ +\infty)\ \times\ (2.0) \\ \texttt{=>} & (3.0,\ +\infty)\ \times\ (2.0,\ 0.0) \\ \texttt{=>} & (3.0 \times 2.0\ -\ +\infty \times 0.0,\ +\infty \times 2.0\ +\ 3.0 \times 0.0) \\ \texttt{=>} & (6.0\ -\ NaN,\ +\infty) \\ \texttt{=>} & (NaN,\ +\infty) \end{array}$$

which is not the same as the expected result:

$$(6.0,\ +\infty)$$

Issues needing further investigation include:

- Providing a `j` suffix that is identical to the `i` suffix so that complex constants appeal to both engineers and mathematicians.

- Explicitly state that complex library functions do not interact with the troublesome `errno` macro.

- There is no definition of the result of many complex library functions when an argument contains a real or imaginary part that is infinity, NaN, and sometimes a signed zero.

Miscellaneous

I have recently setup an E-mail distribution list at *nceg@cray.com*. (This is separate from that used by X3J11, which is currently at *x3j11@sri-nic.arpa*.) If you would like to be added to this facility then send me your E-mail address.

There is considerable sentiment expressed within NCEG for some form of lightweight function overloading to alleviate the name space explosion occurring in the math and complex libraries. However, no proposals have been submitted yet.

Due to its immediate acceptance by NCEG, the initializer enhancements proposed by David Prosser (based on work by Ken Thompson) are easily overlooked. Essentially, this provides a capability for initializing individual members of structures and unions, and individual elements of arrays. For example:

```
struct {
   int m1;
   float m2;
   char m3;
} x = { .m2=3 };

double a[5] = { [4]=3.14, [2]=0.1 };
```

The initializer for `x` only explicitly initializes member `m2`. Similarly the initializer for `a` only explicitly initializes elements 2 and 4. The others are all zero.

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.

∞

4. C/C++ Compatibility

Paul Kohlmiller
Control Data Corporation

Abstract

This paper gives a brief overview of some compatibility issues between C and C++. The perspective is from the X3J16 committee charged with producing an ANSI standard for C++.

From the very beginning the X3J16 committee stated that ANSI C++ should be "as close as possible to Standard C but no closer." (You have probably all heard this quote by now).

I am not going to try to answer the compatibility issues here, but I want to make you aware of the relevant discussions that go on in X3J16. Some of these issues come up at meetings of the ANSI C Compatibility subgroup, of which I am a member. Others, as you will see, involve other X3J16 subgroups as well. Whenever a discussion is brought before the whole committee there is never a lack of opinions.

Modes of Compatibility

C++ is *not* a superset of Standard C. The fact that C++ can be turned into strict C code only says that C is a good assembly language. (For that matter, Lisp can be turned into C code as well.) So, when does C++ compatibility with C *really* matter? There are three scenarios to consider:

1. Mixed C/C++ code

   In this case, C and C++ code coexist in the same application. A favorite target in this case is `setjmp/longjmp`. Consider the following. A C routine does a `setjmp` and then calls a C++ routine that goes down several layers of function calls and then calls a C routine that does a `longjmp` back. What happens to all of the destructors at the end of the C++ routines?

   Another interesting question is which translator gets to compile `main`. The semantics are not quite identical between C and C++. I know of one C environment where executing `main` initializes the C I/O routines. I also know of a C++ environment (on a separate machine) where `main` causes some variables to be initialized. In many implementations `main` is actually called from a runtime library routine. Can the C++ startup routine call a function `main` that is written in C?

2. Standard C code run through a C++ translator

   Can Standard C code be run through a C++ translator? Well, for around $200 you can buy a C++ compiler that also handles Standard C on a PC. Why would you want a separate C compiler? Translators like this have switches to go from C to C++ mode, but shouldn't strictly-conforming Standard C code work in C++ mode? The one obvious exception is old-style function declarations and definitions which the C standard labels obsolescent. After eliminating these obsolescent features, the Standard C code looks like pretty good C++ code.

   Of course, a lot of people are already doing their C code in a C++ environment. It works, as long as you avoid a few of the swampier parts of C. X3J16 hopes to define those parts of the swamp that contain alligators.

3. C++ code calling the C library routines

   If the first two cases sounded obvious then this one surely does also. Can a C++ program call the C library routines? There is a separate subgroup at X3J16 looking at library issues. They will eventually come up with a section that is at least the moral equivalent of §4, Library, of X3.159-1989. One outstanding question in this regard is "Should the ANSI C++ document simply reference this section from Standard C or should it copy and carefully edit it?" The `setjmp/longjmp` issue is still relevant but in a new way. You don't want implementors to copy the entire contents of *libc* minus two routines. You also don't want to force C++ compilers to detect calls to routines with suspicious names. That leaves one alternative, to specify what happens to class destructors when `setjmp/longjmp` is used. That specification might be very simple—undefined behavior.

Why Bother with Compatibility at All?

The C++ community could certainly take the stance that C is a separate language and compatibility is no more important between C++ and C than it is between C++ and Fortran. There is at least one good reason for rejecting this hard line. C++ is new—maybe not to you—but to a lot of people like managers and the people who act like them. If we want to move programmers to C++ then a little lubrication along the migration path is a good idea.

Even if we take the hard-line approach, the C compatibility subgroup is still charged with producing a document (for the X3 folks) that carefully outlines those areas in which C++ is not strictly upward compatible from Standard C. This kind of document is always necessary for languages that are upgrading their standards (like Fortran 90 from Fortran 77). But in this case we have a bit more latitude, since C++ is a separate language. (There is an interesting footnote to this. It seems that COBOL dropped a feature when it was changing the standard. Some people were so strongly affected that they threatened to take legal action against ANSI for allowing this to happen.)

National Character Sets

A funny thing happened last March. X3J11 voted very strongly *against* the principle established by a proposed set of digraphs and keywords that could be used instead of the hated trigraphs. (In some circles this is called the Danish proposal, given that it was first proposed by Denmark at the ISO C level.) One week later, X3J16 voted almost as strongly *in favor* of the proposal. This issue is surprisingly passionate with charges of national chauvinism leveled against the insensitive Yankees and, "Are you still using vacuum tubes?" charges against the Europeans who do not have terminals that can handle the Latin 1 character set. The problem is a serious one. My last name lost one or two umlauts between Bavaria and Pennsylvania. If I was living in Europe I would not take this lightly. The arguments against this proposal are many. I'll describe just one small part of them. Take the following Standard C code sequence:

```
#define str(x) #x

printf("%s\n",str(<>));
```

The problem is that `(<` is the proposed digraph that substitutes for curly brace {. There are at least two ways to solve this problem without destroying the Danish proposal altogether. One solution is to force C programmers to put a space between the parenthesis and the angle bracket unless you really want that kind of token replacement. This kind of solution was what was swiftly rejected at X3J11. A second solution is to note that the macro invocation `str(<>)` will be expanded at preprocessor time. The digraph `(<` will be left alone if these new digraphs are *not* implemented for the preprocessor. In Standard C terms, the preprocessor expands macro definitions in translation phase 4 time while the digraphs could be handled at the same time as escape sequences (phase 5) or treated as new tokens only when preprocessing tokens are itnverted to tokens (which would leave it to phase 7). Note that in a string, braces and the backslash would still require trigraphs.

Another example of code that would be broken is:

```
#define apply(op, l, r) ((l) op (r))
```

CPP Despised!

While I'm at it, I should point out that several members of X3J16 look down with great disdain at one of the hallmarks of C compilers. I speak of course about the preprocessor. Already people are looking forward to ANSI C++ 99 when the preprocessor can be legislated out of existence. The above example with the stringizing operator can be cited as an example of what some people say is just bad programming. *[Ed. However, stringized expressions are necessary to provide the* `assert` *macro and similar constructions; these are an important part of good programming.]* Simple object-like macro constants can be defined just as easily with the `const` keyword (in C++, not in C). Function-like macros can be done with inline functions. (In C++ `inline` is a keyword not an optimization policy.) If you like incremental compilers then you dislike what `#define`s can do to you anyway.

One major reason for using the preprocessor today is to do file inclusion via the `#include` directive. But some environments are already moving away from a simple file-copy model of `#include`. As I understand it, Borland's latest C/C++ compiler provides a header precompile facility. At compile-time, you are actually processing some kind of symbol table rather than a text header file. Some implementations do not grab a file when including the standard C header files but simply make parts available when the `#include` directive for the appropriate header file is found. This does not mean that some kind of resource inclusion is not needed, but it is becoming less necessary to have that inclusion take place in a preprocessor. In the future, a compiler might only need to bring in those resources during the syntactic or semantic analysis phase.

This still leaves conditional compilation directives and pragmas. These things will also hurt the incremental compiler because any change can force a full recompilation of the translation unit. I think X3J16 is not in the mood to consider pragmas as being very important because there aren't any portable pragmas. At the moment, I don't see how one can get the benefit of conditional compilation without a preprocessor.

What's Next?

I have only scratched the surface of the C/C++ compatibility issue. The C Compatibility subgroup is going through the C++ *Annotated Reference Manual* (ARM) and the working draft of the C++ standard looking for every item that is a change from Standard C. Tom Plum (X3J11 Vice-Chair and X3J16 member) estimates that there are about 200 such items in chapters 2–6 alone of the ARM. Most of these are simply upward-compatible features in C++. In a future article, I'll discuss some of the changes that represent true incompatibilities between C and C++.

Paul Kohlmiller is a consultant at Control Data Corporation. He is a member of both X3J11 and X3J16 and can be reached at paul@svl.cdc.com.

∞

5. ANSI C Interpretations Report

Jim Brodie
Motorola, Inc.
Tempe, Arizona

Abstract

This is the fourth article in an ongoing series on the Requests for Interpretation being handled by X3J11. This month I will look at some of the Interpretation requests handled at the March 1991 meeting, held at Norwood, Mass. Questions related to preprocessing, tokenizing, and external linkage are discussed. In addition, X3J11's reply is presented for the long-standing question, "Do functions return values by copying?"

Preprocessing

The early actions of translation (tokenizing and preprocessing) have always been areas of considerable confusion among C programmers. This has been particularly true when people move between environments, because existing practice has traditionally varied greatly. The C Standard addresses this area by significantly increasing the discussion of the preprocessing activities. Specific operators, such as the token-pasting operator (`##`), were added to establish a universal way to accomplish actions which had been previously implemented using significantly different mechanisms. The macro replacement and rescanning processes were also formally defined.

Beyond the preprocessing activities, the Standard attempts to disambiguate the order in which the early tokenizing and preprocessing activities take place. This is done by specifying the *phases of translation*.

There are eight phases to the translation process. In ANSI X3.159-1989 §2.1.1.2, Translation Phases, these phases are defined as follows:

"The precedence among the syntax rules of translation is specified by the following phases.

1. Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character.
3. The source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or comment. Each comment is replaced by one white-space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.
4. Preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.
5. Each source character set member and escape sequence in character constants and string literals is converted to a member of the execution character set.
6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.
7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated.
8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment."

Several interpretation requests explore the new Phases of Translation and extended preprocessing portions of the standard.

The writer of one request for interpretation asks:

> "While processing a token within phase 4 it is sometimes necessary to get the following tokens from the input (e.g., reading the arguments to a function-like macro). But when getting these tokens it is not clear how many phases operate on them.
>
> Do the following tokens only get processed by phases 1–3 or by phases 1–4?

> When an identifier declared as a function-like macro is encountered, how hard should an implementation try to locate the opening/closing parentheses?"

The first example given in the request addresses the issue of finding the initial parenthesis of a function-like macro invocation. Given the following macro definitions:

```
#define lp (
#define fm(a) a
```

to what does the following text expand?

```
fm lp "abc" )
```

Let's look at what happens as the translator processes this text. First it identifies the name `fm`. The translator must resolve the question "Is this the beginning of an invocation of the function-like macro `fm`?" The standard addresses the rules for answering this question. In §3.8.3, Macro Replacement, after describing the definition of a function-like macro it states:

> "Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro)."

In this case, the next token is `lp`. If left unexpanded (`lp` is the name of the object-like macro) it fails to meet the above criteria. Therefore, `fm` would not be the start of the function-like macro invocation. If `lp` is first processed as a macro, the result is different because the `lp` macro expands to the opening parenthesis which would then meet the above requirement.

The committee's decision on this question is that the next token must be a `(` at the point when the potential function-like macro name is being processed. A parenthesis which is the result of a macro expanding tokens later in the token sequence is not sufficient to meet the requirement.

To answer the request for interpretation directly, this means that the tokens following get processed only by translation phases 1–3 prior to being checked for being a `(`. The macro expansions called out in phase 4 are *not* performed. Therefore, the text

```
fm lp "abc" )
```

expands to

```
fm ( "abc" )
```

This request for interpretation provided another example, which questions whether a macro expansion can be used to provide the closing `)` for a function-like macro invocation. For example, given the macro definitions

```
#define i(x) 3
#define a i(yz
#define b )
```

to what does the following text expand?

```
a b )
```

Does it expand to `3` or `3)`?

Let's look at this expansion process. First the name `a` is encountered. Since it is an object-like macro name, it is expanded. This results in

```
i(yz b )
```

The key question is whether the `b` is expanded to be a `)` before determining what constitutes the argument list. In §3.8.3, Macro Replacement, the standard identifies the end of the macro invocation as

> "The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens."

Using the same principle applied above, the committee decided that the text requires an explicit `)` token, not one generated from a macro expansion of one of the tokens from the following sequence of tokens. In this example, this means that the expansion continues

```
i(yz ) )
```

where `yz )` is the argument list to the function-like macro `i`. The final expansion results in the answer `3`.

Although not referenced in the formal reply to this request for interpretation, the committee's position is further supported by the discussion in the standard, §3.8.3.1, Argument Substitution:

> "After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. ... Before being substituted, each argument's preprocessing tokens are completely macro replaced ..."

These words indicate that identification of the argument list is done *prior* to any argument substitutions.

These interpretations are consistent with the committee's stated goal of defining an overt preprocessing activity. As a general rule, the committee has consistently voted in favor of a straightforward preprocessing activity rather than one where the 'obvious' semantics are changed by macro expansions.

The next request for interpretation addresses the new preprocessing token-pasting operator (`##`) by posing the following puzzle. Given the following set of macro definitions:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
```

To what does the following macro invocation expand?

```
join(x, y)
```

The requestor asked whether this expanded to either `xy` or `x ## y`.

From the macro definition of `join` the first step of the expansion results in:

```
in_between(x hash_hash y)
```

Next, we need to expand the `in_between` macro invocation. As part of this macro expansion, we expand the `hash_hash` macro (from the token sequence making up the `in_between` argument).

This is the point of interest. The `hash_hash` macro calls for concatenating the two `#`s which precede and follow the `##` operator. Does this result in the preprocessing token-pasting operator? In the reply, the committee writes:

> "... expanding `hash_hash` produces a new token, consisting of two adjacent `#`s, but this new token is *not* the token-pasting operator."

In other words, a preprocessing operator *cannot* be introduced by macro substitution. Again, the bias for an overt preprocessing activity was supported.

The Relaxed Ref/Def Linkage Model

Another area which has prompted a number of questions centers around how the acceptance of certain constructs affects the conformance or non-conformance of translators. One request asked:

> "Is a compiler which allows the Relaxed Ref/Def linkage model to be considered a conforming compiler? That is, consider a compiler which compiles the following code with no errors or warnings

```
/* FILE 1 */

#include <stdio.h>
void foo(void);
int age;
void main ()
{
     age = 24;
     printf("my age is %d.\n", age);
     foo();
     printf("my age is %d.\n", age);
}
```

```
/* FILE 2 */

#include <stdio.h>
int age;
void foo()
{
     age = 25;
     printf("your age is %d.\n", age);
}
```

and which produces the following output:

```
my age is 24.
your age is 25.
my age is 25.
```

Can this be called an ANSI-compliant compiler?"

The issue is that `age` has been defined in both FILE 1 and FILE 2. This violates the restriction stated in the Semantics portion of §3.7, External Definitions:

> "If an identifier declared with external linkage is used in an expression (other than as part of the operand of a `sizeof` operator), somewhere in the entire program there shall be exactly one external definition for the identifier;..."

An important point to remember is that this restriction is part of the Semantics portion of that section.

We need to reference three other portions of the standard to determine whether the fact that the translator accepted this program without complaint forces the translator to be considered non-conforming. The first is §1.7, Compliance, where a conforming implementation is defined:

> "The two forms of conforming implementation are hosted and freestanding. A conforming hosted implementation shall accept any strictly conforming program. A conforming freestanding implementation shall accept any strictly conforming program in which the use of the features specified in the library section (§4) is confined to the contents of the standard headers `<float.h>`, `<limits.h>`, `<stdarg.h>` and `<stddef.h>`. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program."

Earlier in the Compliance section (§1.7) a definition of strictly conforming is provided:

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

Finally, in §1.6, Definitions of Terms, it is stated:

> "If a 'shall' or 'shall not' requirement that appears outside of a constraint is violated, the behavior is undefined."

The violation of the restriction on multiple definitions means that the program relies on undefined behavior. Since the program relies on undefined behavior, it is not a strictly conforming program. Since a translator is only required to accept strictly conforming programs, the translator is free to accept, reject, or do strange things with this program while still meeting the requirements of being a conforming implementation. In other words, yes, the translator *can* accept this program while still claiming to be ANSI-compliant.

This request demonstrates one of the key reasons for including undefined behavior within the C standard. Many people view undefined behavior as an undesirable property, where X3J11 simply put insufficient restrictions on how a translator should respond to error conditions. While it is true that much undefined behavior is allowed because it was 'too hard' for the translator to detect and handle an error, undefined behavior has also been consistently used to define a clear arena where language extensions or increased features are allowed. To modify an old saying, "One person's error is another person's feature."

It is important that translators be given this realm of freedom. It allows flexibility so that existing practices which are no longer directly supported by the C Standard but still in use can be supported by a translator without forcing the translator to be non-conforming. It also provides an area where experimentation and new language features can be tried. This process of experimentation is important to the long-term health and growth of the C language.

Library Function Issues

Several requests for interpretation explored the handling of boundary or error conditions when dealing with the library functions. One such request asks

> "What is the resultant output from `printf("#.4o", 345)`? Is it `0531` or is it `00531`?"

The cited reference is §4.9.6.1, The `fprintf` Function, which includes the statement:

> "For `o` conversion, it increases the precision to force the first digit of the result to be zero."

The question is essentially whether the precision is increased by at least one, even if the increased precision is not required to guarantee the leading 0?

The committee's position is that the increase is only provided if required to support a leading 0. In this case *no* increase in precision is performed and the answer should be `0531`. In other words, an increase of 0 positions of precision is appropriate when the number can be represented with a leading 0 in the specified precision.

Functions Return By Reference or By Value?

In *Volume 2, number 3* (December 1990) I discussed, at length, the interpretation request, "Do functions return values by copying[4]?" After long consideration X3J11 finally came to a decision. The committee decided that **the function return must be done as if a copy was being performed**. Optimizations are, of course, allowed, so long as the observed behavior is not different from what would be obtained by the copy action.

The committee's answer is based upon the description in §3.6.6.4, The `return` Statement. Of particular importance is the statement:

> "If a `return` statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression."

The committee interpretation response notes that

> "If any storage used to hold 'the value' overlaps storage used for any other purpose, then 'the value' would not be well-defined. Therefore, no overlap is allowed."

[4]For a discussion of how this interpretation came about see Paul Eggert's article, page 54.

This decision has a direct impact on how translator developers implement structure return actions. Before structure return optimizations such as passing an address of a regular object rather than pushing a value onto the stack can be used, there must be appropriate checks to ensure that no overlap can occur.

Until these additional checks are made and in cases where the translator is unable to determine whether an overlap occurs, less efficient code will probably be generated. This is a tradeoff between 'the right answer' and 'the fast answer.'

Jim Brodie is the convenor and Chair of the ANSI C standards committee, X3J11. He is a Software Engineering Process Manager for the Semiconductor Products Sector of Motorola, Inc., in Tempe, 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) 897-4390 or brodie@ssdt-tempe.sps.mot.com.

∞

6. C as an Intermediate Language: A Case Study

Paul Eggert
Twin Sun, Inc.
360 N Sepulveda Bl.
El Segundo, CA 90245, USA

Abstract

An increasingly common way to implement special-purpose and experimental languages is to write a translator that generates C code. The resulting system can be portable, relatively efficient, and easier to build and support than a native code generator. We have recently written a translator from a C variant called "C-cured" to Standard C itself. Because the languages were so close and the target so standardized, the problems that we encountered are likely to occur in any project that uses C as an intermediate language. These problems include making the C level invisible, working around deficiencies in Standard C and its implementations, tracking C types, interfacing to external programs and data, cross-compiling, and managing memory. Although such problems are common to many such projects, they are rarely discussed openly. Our solutions are discussed, and a basic checklist of design considerations is proposed.

Introduction

In the old days, compilers generated machine code. But the increasing need for specialized languages for application generators and for other special purpose environments, coupled with the continued proliferation of CPU architectures, has made it harder to justify writing a traditional compiler even where good performance is required. Another option is compiling to a special-purpose machine-independent intermediate language, but this can still require nontrivial effort to port to a new machine. Many implementers are turning to a different approach: instead of generating machine code, generate code using a portable, low level programming language, then compile the resulting code on each target architecture. The most common language used is C, because of its efficiency and increasing popularity. Let us distinguish these new compilers by calling them *translators*.

At first glance, it may seem that generating C code essentially solves most problems of code generation, letting the implementor concentrate on simpler issues like lexing, parsing, and symbol-table management. To some extent this first impression is true. The translator writer need not worry about questions like whether `clrl d0` or `moveq #0,d0` is more efficient. At some cost in performance, such low level issues are no longer the translator writer's concern. This makes the translator smaller, simpler, and easier to build, support, and port than a full fledged compiler. In effect, such a translator is reusing the C compiler's code generator, with substantial software-engineering savings.

However, several problems of traditional code generation resurface in altered form in translators. And some new issues arise, partly because C is more complicated than most architectures, and partly because when the source language is C-like there is a natural desire to make the translation resemble the source.

We have written a translator to Standard C for a C variant called "C-cured." This variant language makes it relatively easy for a compiler to check for common programming errors like subscript violations and dereferencing null pointers. It is intended for sensitive applications where high software reliability is required [6]. C-cured is intended to complement the use of formal methods: whereas formal methods address the issue of high-level bugs (e.g., a program raising a robot arm that it should have lowered), C-cured addresses the issue of low-level bugs (e.g., a program dumping core).

Much of the C-cured translator is devoted to issues like linear programming and sophisticated type checking that is beyond the scope of this paper. However, the software engineering issues involved are similar to those of other translators that generate Standard C. In fact, since the source language C-cured is so similar to Standard C, the relatively few problems with translating it may well be common to most translators to Standard C. Furthermore, since C-cured is a syntactic extension of C, translators for other languages that have C as a sublanguage (e.g., embedded SQL) may well have similar design issues. Therefore, although this study concentrates on problems in translating into Standard C, it will also discuss problems in translating among Standard C variants.

Standard C is a natural choice for a target language because it is standardized, it will soon be ubiquitous, and it permits low-level manipulations that a translator might need. In the future, C++ may be a suitable target, but C++ is not yet standardized and C++ technology is less mature. Unless the source language is class- or object-oriented, C++ is little better than C as a translator target. Because some crucial areas like exception handling were not fully standardized, we chose Standard C over C++ for our first project.

Design Goal

No matter what the source language is, any translator to C should *keep the C implementation invisible.* Although the user may know that the system is C-based, its internal details should not be exposed to view.

For example, compile-time error messages should be generated by the translator to C, not by the C compiler itself. C compiler messages will probably confuse the programmer because the underlying C code may not resemble the source code. Therefore, the output of the translator should be accepted by the C compiler without complaint.

For another example, when the programmer is debugging a program, the C level should be invisible. Views of the source code and commands to the debugger should be expressed in the source language, not in C. If the source differs enough from C, this will probably require changes to the programmer's debugger. The translator writer will still need access to the C debugger to guard against errors in translation, but the translator user should normally be insulated from the C level.

Syntactic Issues

C-cured's context-free grammar is an extension of C's. C programs usually violate the stricter type rules of C-cured. But when given an unmodified C program, the translator should generate type-violation messages that are more useful than "syntax error." Unfortunately, C's grammar is not easy to extend in certain ways. For example, it is tempting to invent keywords to represent the extended concepts, but this would invalidate existing programs that happen to use those words as identifiers. Therefore, C-cured introduces no new keywords except for `new`, which it borrows from C++.

It is hard to add new type constructors to C because its type syntax is already so confusing. Because C-cured requires much more precise types, incremental changes to the C type syntax would be hopelessly confusing. Instead, we use a new syntax for C types and C-cured extensions. We support the old syntax for compatibility purposes. The following table shows two types in both syntaxes.

C vs. C-cured Type Constructors

C	*C-cured*	*Explanation*
`char (*)[10]`	`&[10]char`	pointer to array of 10 `char`
`char * [10]`	`[10]&char`	array of 10 pointers to `char`

This new syntax lets C-cured add new type constructors in a disciplined way. For example, `+[10;]char` stands for "nonnull pointer to a null-terminated array of 10 characters." It would have been difficult to add this notion to the already-confusing C type syntax without straining it beyond recognition. It is quite a strain to shoehorn the new type syntax into the grammar while keeping the language context free, because both old and new syntaxes had to be supported. Future versions of C should support a simpler type syntax. We suggest C-cured's new syntax as a starting point.

In contrast, adding new type qualifiers is straightforward. C has just the type qualifiers `const` and `volatile`. In C-cured, `const` is used so often that it can be abbreviated as '`<`'. A new complementary qualifier '`>`', for unreadable storage, is easy to add to the syntax. The relative ease arises partly because type qualifiers cannot be confused with normal identifiers, unlike typedef names. The following table shows two examples.

C vs. C-cured Type Qualifiers		
C	*C-cured*	*Explanation*
`const int *`	`&<int`	pointer to unwritable integer
`int * const`	`<&int`	unwritable pointer to integer

Bugs in the C Standard

One design decision that we do not regret is that of translating to Standard C instead of to traditional C. The new standard defines a better language and has fewer loopholes than the traditional definition. We recommend that future implementers study the C standard carefully. However, because translators often generate code that no human would write, they tend to explore unused corners of standards and implementations, where bugs are more likely to lurk. We found two bugs, one minor, and one major.

The minor bug, discovered by Mike Coleman, was that the December 1988 ANSI C draft [1] permitted multiple definitions of unused identifiers with external linkage. For example, consider the source file

```
int V = 0;
int V = 1;
```

where `V` is mentioned in no other source file. Ironically, we found this bug when considering what checks to put into the C-cured translator itself, because in this area C-cured is identical to C. We caught this bug before the final standard was approved, and X3J11 fixed it in the final standard by inserting the phrase "there shall be no more than one [external definition of an identifier]" at the end of §3.7.

The major bug is that the standard does not clearly state whether functions must return values by copying. Returning by reference can be more efficient than returning by copying, but the results can differ in the presence of aliasing. Using unions, one can write an assignment `S = f(X)` in which `f(X)` obtains the returned value from an object whose storage overlaps that of `S`. Here is an example, where `S` is `u.a` and `f(X)` is `deref(&u.b.s)`.

```
struct s { int i, j; } t;
struct s deref(struct s *q) { return *q; }

union {
        struct s a;
        struct { int i; struct s s; } b;
} u;
void p() { u.a = deref(&u.b.s) };
```

This is quite unlikely to appear in programs written by humans, but was uncovered during the design of our code generator when considering the code it might generate for unions. We filed a formal request for interpretation on this question. After lengthy debate [4], X3J11 decided in March 1991 that functions must return values as if they were copied[5].

Bugs in the Underlying Implementation

One problem with a translator-oriented approach is faults in the underlying system. We have encountered a few bugs in GCC [12] and a few more bugs in Sun software. Using the translator ourselves has proved to be of great use in this area.

One major remaining trouble spot is checking for memory exhaustion at run-time. This cannot be done by the compiler at present and, unfortunately, needed abilities like robust recovery from run-time stack overflow are not yet widely available. This is particularly a problem with C-cured, where software reliability is paramount, but we expect it also to be a problem in any translator that makes more than casual use of the run-time stack. Exhausting the heap area is also related to garbage collection issues discussed below.

Implementation-Specific Code Generation

Ideally, the code generated by the translator should be portable to any Standard C implementation. In practice, this goal *cannot* be achieved. Partly, this is because the translator must take advantage of some implementation-dependent information. For example, to check for arithmetic overflow, the translator must know the value of `ULONG_MAX` for the target implementation. Otherwise it cannot tell whether the constant `4294967296` will overflow. The problem of knowing the target architecture is particularly acute when considering C's sometimes odd rules for integer arithmetic. For example, in C the expression

```
    -1 > (unsigned)0
```

yields 1. In contrast, C-cured uses a subset of the mathematical integers and largely ignores C's distinction between signed and unsigned integers of various sizes.

Even ignoring the problem of checking for overflow at compile-time, code generation depends upon knowledge of the target architecture. For example, in C on any 32-bit machine with IEEE floating point, comparison need not be exact. The sample code

[5]For further information on this interpretation see Brodie's report on page 49.

```
float f = 2147483648;
int i   = 2147483647;

if (i == f && (double)i != (double)f)
        printf("weird");
```

can output `weird` because the first comparison is permitted to discard information about `i` before comparing it to `f`. In contrast, C-cured performs comparisons exactly, so that the C-cured comparison `(i == f)` must be translated on such a machine to something like the C comparison `((double)i == f)`. A similar analysis applies when comparing a value stored in a C `int` object to a value stored in a C `unsigned` object, to avoid the problem of `-1 > (unsigned)0` mentioned above.

The target compiler may have extensions that the translator can take advantage of. For example, C-cured supports local arrays with size not known at compile-time. This notion is not supported by Standard C, but it is a common extension [10, 11], and it appears in GCC. Another C-cured feature that maps directly into GCC is constructors. In general, the translator must fall back to less efficient Standard C to support these constructs, but it should take advantage of them if it knows the target compiler supports them efficiently.

With this in mind, it should be clear that the translator suffers the same cross-compilation issues as a traditional compiler. A program translated for one target architecture may not compile correctly on another, or may compile but run incorrectly.

Interfacing to C Data Structures

A common desire when using a special-purpose language is to link to code written in C. Usually the main problem here is not naming or calling foreign functions. Instead it is passing data of a common format back and forth. Choosing a language close to C mitigates this problem, but does not always eliminate it. For example, C-cured uses the same run-time representation as C, so there is no problem with run-time conversion of values when passing data back and forth. However, there is a compile-time problem with the description of the data. A C compiler cannot parse the C-cured type syntax. Thus it cannot incorporate C-cured headers. Conversely, the C-cured translator rejects a C-cured program that attempts to use C headers, because the C types appear hopelessly loose, and their use in the C-cured program violate many of the stricter C-cured type rules. We have worked around this problem internally by writing a translator from C-cured headers to C headers.

This solution is not entirely satisfactory because of preprocessor directives. Normally, the C-cured translator first passes the input text through the standard C preprocessor, then translates the resulting text from C-cured to C. But this cannot be done for headers, because it loses preprocessor definitions needed by modules that include the headers. For example, if a C-cured header `id.h` contains the code

```
#define ID_MAX 10000

|0:ID_MAX| mainid;
```

the C preprocessor generates

```
|0:10000| mainid;
```

and the C-cured translator might then generate

```
unsigned mainid;
```

But one cannot put just this last line into the translated C header `id.h`, because a C program might contain

```
#include <id.h>

char tab[ID_MAX+1];
```

and thus need the preprocessor symbol `ID_MAX`. We currently address this problem by a rigid layout for C-cured headers that permits copying the preprocessor information bodily to the C headers. But this solution is not entirely satisfactory.

Preprocessor Issues

Writing a translator to C naturally raises another issue: should this translator operate before the C preprocessor or afterwards? In general, Standard C does not require that the C preprocessor operate as a separate pass, but most implementations have options for running the C preprocessor separately.

C-cured uses an unmodified C preprocessor, and the C-cured translator operates between the C preprocessor and the C compiler proper. However, this setup is suitable only for languages that have the same lexical tokens as C. One should beware using the C preprocessor for a widely different notation (e.g., as a preprocessor for makefiles). Even the slightest deviation should be cause for concern. For example, C-cured originally used a more Pascal-like notation `|0..N|` for the range of integers from zero through N, but this had to be changed to `|0:N|` because a quirk in the Standard C preprocessor tokenization rules means that `0..N` is a single (preprocessor number) token, and the preprocessor cannot expand the `N` when `N` is a macro.

Because C-cured is so close to C, it is not unreasonable for programmers to use existing C debuggers with little or no modification. To communicate with the C debugger the translator must put the tedious but necessary line-number information into its output file.

Source languages whose names and scope rules differ markedly from C's may require more extensive interfaces and changes to the debugger, but with C-cured these problems are rare. For example, C-cured has the same scope rules as C and can therefore output the same names that it input. Although it generates names for temporary variables that are visible to the debugger, they do not distract the user because the user never sees them. Also, C-cured uses the same expression sublanguage as C, so there is little need to change the syntax of debugger commands that have C expressions as operands, and we use a standard C debugger for C-cured programs. A language with a different expression syntax would need changes to the debugger.

Managing Memory

Many modern languages require garbage collection in some form. C has no builtin support for garbage collection, so this must be added by the translator writer. Translated code must put all root pointers in global storage known to the collector so that the collector will correctly mark all reachable storage. This must be done even in the presence of signal handlers and interrupts. One can put local pointer variables of the source language into a large C array that is managed as a stack. A better method, adopted by the Unisys Swift translator [7], is to take advantage of C's builtin stack by translating source local variables into C local variables, storing both a type descriptor and a parent-frame pointer in each C stack frame. This maintains efficiency while letting the garbage collector easily traverse stack frames in an implementation-independent way.

Recently a new "conservative" method for garbage collection in the C run-time environment has been proposed. It was first implemented by D. McIlroy of Bell Labs, and first described by Boehm and Weiser [3]. Instead of maintaining run-time type descriptors, it traverses the entire run-time stack and assumes that all properly aligned bit patterns that could be pointers are in fact pointers. Although conservative collection is not suited for every application [13], and it does not work well when a pointer to an object component can outlast the pointer to the containing object, we will use it in the first C-cured implementations because it is so easy to integrate. We consider memory management to be one of the biggest potential trouble spots in any translator to Standard C.

A Basic Checklist of Design Considerations

If you find yourself in a project that will write a translator that emits C, ask yourself the following questions:

- *How close to C do you want to make your source language?* The closer the language is to C, the less design freedom you will have. On the other hand, it will be more likely to appeal to existing C programmers, and you will be able to use existing software like the C preprocessor if your source language is sufficiently close.

- *How will you map generated code back to source?* Without this mapping, programmers will not be able to use their debuggers. Without debuggers, programmers will be far less likely to use your system.

- *How will you map deficiencies of the underlying standard or implementation back to the source?* Something must give when C lacks features needed for smooth translation of the source language. For example, if the source requires garbage collection, the translated code must be either less portable, less efficient, or more complicated, and will probably have at least two of these three properties.

- *How will new programs interface to existing software?* If the source language is close to C, the answers to this should be straightforward, and simple descriptions of layout and naming conventions will often suffice. If not, careful thought must be given to the needs of developers who must link modules in the source language to modules written in C.

Related Work

Translators to C have been with us since the days of yacc [2]. General-purpose language implementation started adopting this approach in the early 1980s (e.g., AT&T's C++ translator [8] and Unisys's Swift language [7]). Since then, this approach is becoming the method of choice for new experimental implementations that are meant to execute efficiently (e.g., Kyoto Common Lisp, Eiffel, SRC Modula-3, and Scheme-to-C). A related, flexible approach can be found in application generator technology (e.g., AT&T's MetaTool [5]).

Conclusion

Translating to Standard C, and choosing a C-like source language, can ease implementation of an experimental language. Both can also bring new problems. The underlying C implementation must be kept invisible, and the translator must avoid triggering error messages in it. Extending C's syntax can be an adventure in itself, especially if changes are made to the already baroque syntax of declarations. One must beware of bugs not only in the underlying system, but in the C standard itself. The software engineering issues of cross-compilation are not much less severe than with traditional compilers. Careful thought must also be given to interfacing source language data structures to C data structures. When the source language is sufficiently close to C, the C preprocessor may also be used, but this can complicate data structure interfacing. Finally, memory management and garbage collection can be among the trickiest issues when the source language's memory model differs from C's.

Despite all these problems, we have been quite happy with our approach. Porting to a new architecture is simple. Because of the success of the UNIX operating system, new hardware architectures are often coupled with excellent C compilers, and the generated code is far superior to what we could have developed with our own resources. We also expect that many of the problems we have identified will be addressed by future C implementations as translating to Standard C becomes more popular.

References

[1] American National Standards Institute, *Draft Proposed American National Standard for Information Systems—Programming Language C*, X3J11/88-158, December 7, 1988.

[2] American National Standards Institute, *American National Standard for Information Systems—Programming Language C*, X3.159-1989, approved December 14, 1989.

[3] H-J. Boehm and M. Weiser, *Garbage collection in an uncooperative environment. Software—Practice and Experience* **18**, 9 (September 1988), 807–820.

[4] Jim Brodie, *ANSI C Interpretations Report. The Journal of C Language Translation* **2**, 3 (December 1990), 207–215.

[5] J. Craig Cleaveland, *Building application generators*, *IEEE software* **5**, 4 (July 1988), 25-33.

[6] Paul Eggert, *Toward special-purpose program verification.* ACM International Workshop on Formal Methods in Software Development, Napa, California (9–11 May 1990).

[7] Paul Eggert and D. Val Schorre, *Swift Design Notes.* Unpublished memorandum, System Development Corporation, Santa Monica, CA (1981).

[8] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Menlo Park, CA, 1990.

[9] Stephen C. Johnson, *Yacc: Yet Another Compiler-Compiler. Bell Laboratories Computing Science Technical Report 32* (31 July 1978).

[10] Tom MacDonald, *Variable Length Arrays. The Journal of C Language Translation* **1**, 3 (December 1989), 215–233.

[11] Dennis M. Ritchie, *Variable-Size Arrays in C. The Journal of C Language Translation* **2**, 2 (September 1990), 81–86.

[12] Richard Stallman, *Using and Porting GNU CC*, version 1.39.1, Free Software Foundation, Cambridge, Massachusetts (18 January 1991).

[13] E. P. Wentworth, *Pitfalls of conservative garbage collection*, *Software—Practice and Experience* **20**, 7 (July 1990), 719–727.

Ed: A preliminary version of this paper appeared in the Proceedings of the 16th jus Unix Symposium *(Osaka, 15–16 November 1990), pp. 137–149.*

Paul Eggert is Director of R&D for Twin Sun, Inc., a company that specializes in software engineering for portability and reliability. His technical interests include compile-time error detection, formal methods, software management, and software visualization. He can be reached at eggert@twinsun.com.

∞

7. Electronic Survey Number 8

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 100 different people, with 22 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.

`putenv`

Standard C defines `getenv` *but not* `putenv`. *Does your* `getenv` *do anything useful? If so, do you also supply* `putenv`*? Where is* `putenv` *declared?*

- 2 – `getenv` doesn't do anything useful.
- 16 – `getenv` works as per the UNIX definition.
- 8 – Have the standard UNIX `putenv`.
- 9 – Don't have `putenv`.
- Comments:
    1. I don't do my own compiler, but a missing `putenv` is certainly an indication of the quality of the implementation.
    2. The lack of `putenv` is an obvious flaw in the standard.
    3. `putenv` is declared in `<stdlib.h>`. Of course, when the compiler is standards conforming, the declaration is not seen in the header.
    4. We provide `putenv` but it is not declared in any standard header.

Alternates to Trigraphs

In order to have have more readable and writeable versions of trigraphs, Denmark has proposed alternate spellings for the 9 affected tokens that cannot be directly represented in the ISO-646 character set. Their latest proposal is to add the following alternate spellings for the tokens indicated:

`(: :)`	`[ ]`	`bitxor`	`^`
`(< >)`	`{ }`	`bitcompl`	`~`
`and`	`&&`	`bitand=`	`&=`
`or`	`||`	`bitor=`	`|=`
`bitand`	`&`	`bitxor=`	`^=`
`bitor`	`|`		

Thus far, this proposal has been favorably looked upon by X3J16 (ANSI C++) but not so by X3J11. If Standard C (and C++) were to require such support, currently conforming Standard C programs will be broken. Your comments please on the specific technical proposal and on the problem in general.

- 3 – It's OK.
- 7 – It's OK, but ...
- 10 – Don't like it.
- Comments:

  1. I don't object to most of these new tokens and keywords. However, I do object to using `and` or `or` as keywords, since they are too short and probably too common.
  2. This is a great idea. Trigraphs were a botched kludge!

     The tokens spelled as identifiers could be implemented now by some sort of standard header, ostensibly one which redefines the trigraphs. The bracketing tokens, however, do seem to break possibly existing features of some code. I can see this happening in the case

     ```
     #define postfix_dyad(a,b,x) ((a) x (b))

     postfix_dyad(s,t,>)
     ```

     which, under the current standard, after preprocessing, expands to

     ```
     (s) > (t)
     ```

     but, using the proposed translations, in translation phase 1 the code would instead become

```
#define postfix_dyad(a,b,x) ((a) x (b))

postfix_dyad(s,t,]
```

And that is intractable.

This example is one of such utility that it makes me believe that the character sequences `>)` and `(<` should retain their current untranslated nature.

3. Specifically, I would use `and`, `or`, and `xor` for the bit-wise operators, and `andif` and `orif` for the logical ones. Generally, so long as they can guarantee this won't impact ASCII and Latin-1 I don't much care.
4. I have discussed the favorable reception of this in X3J16 with our representative. I have deep concerns regarding the intention to make these keywords and tokens. The attitude of having switches to cover the C programs doesn't address the situation of the shop with a very large body of existing C code that they want to incrementally convert to C++. This is a very large burden to place on those shops, and I will do all I can at the X3 level to prevent this from being allowed as part of the C++ standard.
5. Trigraphs were a mistake, and the Danish alternative is no better. There are better ways to solve the problem than by tweaking the standard language. For example, it's not especially difficult to configure GNU Emacs to display non-native characters as Danish digraphs, without actually changing the contents of the file.

   I programmed in APL for years on a plain ASCII terminal. I used a front-end that translated ASCII trigraphs into APL characters. There was never any need to change the standard APL language to make those trigraphs universal.
6. I think it's ugly. I can't see having all this as a requirement. As an option to the user it's OK. Of course I understand options are awkward, there being no standard way to specify them.
7. First, the deficient keyboards and terminals that prompted this proposal have been obsoleted by the ISO 8859 character set. Why add complexity to an international language standard because of obsolete gear? However, even assuming one had to add brain damage to C to work around this local disaster, this proposal is a bad technical solution to the problem. A good technical solution would have the following properties:

   (a) Existing conforming programs run without change.

   (b) Programs using the new syntax can be transported easily to implementations that support only the old syntax, e.g. using macros.

(c) The solution is easy to explain.

(d) The solution is compatible with new notations, e.g., C++.

The proposed solution fails on all four counts. It obviously fails on 1. It fails on 2 because no simple editor script or C macro set can transform programs from the new syntax to the old. It fails on 3 because one must explain to programmers that

```
        char b[100], p = bitand a;
```

is legitimate because `bitand` is shorthand for `&`, even though *no* bitwise arithmetic is being performed here. It is also counter-intuitive that `&` is shorter than `&&`, but its counterpart `bitand` is longer than `and`.

Finally, the proposal fails on 4 because it will make C++ hard to read: e.g., destructors will use a strange syntax:

```
        class String {
        public:
                bitcompl String();  // a destructor
                ...
        };
```

and in general, English symbols like `bitand` will have to used in contexts where they are wildly inappropriate.

Although I'm sympathetic to the Danes' plight, their proposal is seriously deficient and needs further thought.

8. We strongly oppose this idea since it would cause large problems with our lexical analyzer. It also raises the issue of reducing the users' name space.

9. I understand the desire to have alternate spellings for the `[` and `]` tokens. They are critical to convenient language use. However, I believe that Tom Plum's proposal of a standardized header with standard macros for the other tokens is a more reasonable approach. One doesn't, for example, have to wait for implementors to provide conforming implementations to begin to use the agreed names. If we can just find some reasonable alternate spelling for `[` and `]`, I would be hard-pressed not to support such a combined (standard header plus two new tokens) scheme. The best reasonable alternate spellings would be ones that do not change the behavior of strictly conforming C code. (One possibility might be `<:` and `:>`, for example.)

10. I don't see why several of these alternate spellings can't just be supplied as macros. The `and`, `or`, `bitand`, `bitor`, `bitxor`, and `bitcompl` can be added by anyone who likes without any special effort on the

committee's part. Personally, I find the `bitFOO=` token form more than a little disgusting. The alternate spellings for `[]` and {} seem okay to me though.

Whenever I see these proposals it makes me wonder why we are stepping back in time to support character sets that are obsolete? This is an issue for much much more than just the people saddled with old, C-hostile equipment. If C implementations were required to support these forms, those forms would be with us forever. Every implementation would have to add these strange forms. Programmers would have to be aware, when they learn the language, that C uses multiple spellings for this collection of tokens.

This latest proposal adds more keywords than any other single proposal ever put forward for C. Frankly, this capability is not worth six keywords.

Local macro substitutes have the advantage of not imposing on other environments. The people with inadequate equipment have the problem and they need the fix. The majority of C programmers who don't have the problem shouldn't have to conform to a dwindling minority who will ultimately become non-existent.

11. Unlike heads, two trigraphs are *not* better than one. Other than that and the obvious problem of introducing many new keywords it is easy enough to implement. I would suggest allowing the `bitand`*op* operators to be two tokens for the sake of automatic paragraphers and such.

12. I think the concept is fine. I would prefer, however, `bitnot` to `bitcompl` as more in the spirit of C. I would object if a space were `not` allowed between the identifier part and the `=` though.

13. I don't like the use of macros. I think that the alternative spellings are OK.

14. I voted in favor of this proposal on X3J16. C++ is already introducing new keywords, so the impact of the proposal on C++ is less than on C.

    If the proposal goes into C I would expect many vendors to have to support a *transition mode* in which these keywords were not recognized as such into the indefinite future.

    Since we do not yet have a standard-conforming compiler I'd expect adding new keywords would be possible at the time we introduce the standard-conforming compiler without creating many (extra) headaches.

15. If they *really* want this, I could live with it, but it would have to be on a command line option (or by including some special header).

16. While we are sympathetic to the problem being solved, we think there are several problems with the Danish solution being proposed:

it breaks existing programs and it includes changes that are not necessary (e.g., the `and` names are not needed). We would look more favorably on a solution using macros.

17. We're opposed to this proposal for C (at least). Trigraphs are admittedly ugly but we've seen nothing better so far.
18. I agree that the proposed syntax is more readable than trigraphs. However, as described, the proposal does have problems. I think a macro solution makes more sense.
19. My guess is that if we support these alternate spellings some (or many) of our customers will complain about it because of the name space conflict.
20. Most of the alternatives are much more readable than the corresponding trigraphs. However, I can see a number of drawbacks: It breaks existing code and in C++ the template-argument-list is delimited by `<>` which is ambiguous in some cases with the relational operators `<` and `>`.

    The spellings for the tokens `&&` through `^=` tokens are a bad idea because they break existing code. The problem can be solved using macros.

Validation

Given that NIST has chosen a validation suite different from that used by BSI how important to you is mutual recognition of validation certificates in the U.S. and Europe? (Don't care, minor issue, or major problem.) What resources (staff, etc.) do you perceive you will need to dedicate to conformance testing? (For example, 1 person half-time, full-time, etc.)

- 10 – I care about mutual recognition.
- 6 – I don't care about mutual recognition.
- Comments:

  1. This issue only affects users if there is a difference in the quality of the translators certified by the validation suites. The users will want to know which is the most protective and look for assurance that the translator they want meets that. As such, I am opposed to having mutual recognition. This is the only way a user can be certain the ISO and FIPS-compliant compiler has met the most thorough validation tests.
  2. We test with both suites [Perennial and Plum-Hall]. We may not seek certification from both if reciprocity is required.

3. We feel its important to be validated by both but are very disappointed that two different validation suites have been chosen. We feel that validation for both NIST and BSI will require 1 person approximately 8 hours per week.
4. I presume that we would get both if mutual recognition doesn't happen. We have used the test suites for both and have no outstanding bugs that would cause problems. (Although there may well be a few rulings put in the works on the correctness of the tests once we start the validation process.) Thus, it would be nice if each recognized the validity of the other, but I doubt that it's a big problem.
5. Multiple conformance certificates presents a small problem to us. We regard conformance validation as an internal QA function as well as an external acceptance function. As a result, we use validation suites as part of our standard test suites and don't actually measure the costs of validation separate from our QA costs.
6. It's a minor issue. Our validation testing is part of our normal product-release testing so I can't really give a very good estimate of how much time we spend on it—less that 1 person half-time I would think.
7. I will test against every suite anyway since it's a one-time cost.
8. I think that it is a good idea to have lots of test suites. That way there is some competition. Any vendor worth their salt will use all available tests suites. The expensive bit is having to pay out for the official validations. I think that there ought to be some procedure for mutual recognition.
9. The divergence in conformance standards is important to us in direct proportion to how often customers are asking for conformance to both standards. At present this is a minor issue but it has potential for being a major issue. Our perception is that conformance testing will take one person part-time.
10. We have one full-time person doing compiler testing including conformance testing.
11. We intend to run and pass both validation suites. Therefore, the manpower requirements are not significant if there is not mutual recognition. The only issue will be the actual cost of certification and so we will get certification for both only if that is required for our customers.
12. Here, in Japan, validation is not required officially. At this stage, I don't care about any differences between U.S., Europe, and Japan. If any official requirement for the validation comes out there *must* be a mutual convention that accepts a compiler validated by other organizations.

13. Compiler verification is an important issue and marketing aspect for us. Therefore, we are willing to spend at least 1 person full-time during the validation test period.

AT&T's cfront

Have you determined whether AT&T's C++ cfront *works with your C implementation? If so, any comments on the effort?*

- 4 – Works fine.
- 3 – Works but we had problems.
- 6 – Don't know/care.
- Comments:
  1. It only partially works but will be working fully shortly. The problems are in cfront.
  2. The only interesting problem is significant name lengths, and our C compiler imposes no limits other than those enforced by memory allocation, etc. Each new cfront doesn't take very long to port.
  3. We have a C++ compiler integrated with our C compiler so cfront compatibility is not an issue for us.
  4. My experiences with cfront in the dark corners have been uniformly dismal.
  5. We had to fix a couple of minor C compiler bugs. The whole effort including fixing bugs and adjusting headers took less than two weeks.
  6. The porting effort took a significant fraction of a man-year. The process uncovered bugs in both our compiler and in cfront.

Implementation Language

What language is your translator written in?

- 16 – All in C.
- 3 – Mostly in C.
- 1 – Some in C++.
- 1 – Some in Pascal.
- 1 – Some in Assembler.

∞

8. Emitting C Source

Paul Long
3365 Arbor Drive
West Linn, OR 97068-1115

Abstract

This paper discusses problems associated with emitting well-structured Standard C source. Alternative solutions are presented, culminating in the one chosen by the author. A set of C functions is described that implements this solution, followed by limitations, enhancements, and possible uses.

Introduction

Emitting code in a high-level, free-format language such as C presents problems that do not occur when emitting machine or assembly code. Machine code, by its nature, is not particularly human-readable—nor is it intended to be. There is no impetus to make it so. Assembly code is inherently well-structured because it has short instructions that start in fixed positions and traditionally occupy one line each. In contrast, emitted high-level code often contains lines of arbitrary length. Its style dictates indentation that reflects a structural aspect of the code. As an example, a heavily-qualified structure member reference occurring at a deeply-nested point in the code may extend well past the end of a typical 80- or 132-character line.

When C is produced as an intermediate language, the programmer may need to refer to the emitted code from time to time. When used as a target language, however, the code produced takes on a life of its own. It irrevocably diverges from the original form in many cases. Consider, for example, a program that is converted from Fortran to C. In both cases, it is desirable to emit well-structured, human-readable high-level code with two lexical qualities in particular—a reasonable limit to line width and proper indentation.

Line-Width Control

For a source translator, such as Pascal-to-C, if there were a one-to-one lexeme translation from one language to the other(e.g., `:=` to `=`) the widths of the emitted lines would be about the same as the original lines. However, an across-the-board, one-to-one translation is rarely possible. Even if the semantics of the original program were carried over to the emitted program, the form would likely be distorted. One lexeme may expand into many, and many lexemes may contract into a few, one, or none. Practically speaking, contraction is not a problem, but a line that was originally 75 characters wide, for example, may expand into one that is 300 characters wide, making it impossible to display or print in an elegant manner.

Other types of source-code-emitting programs may not even have an original source-code representation. There are no reasonable line widths, written by programmers, to exploit. Such an origin can at least indirectly limit the line widths of the emitted code.

Hasty Solutions

Since one cannot just truncate all lines wider than the maximum line width, the next solution would be to start a new line every time the maximum width is reached. C is free-form and allows statements to span several lines, so this solution looks promising. However, a line break will frequently occur in the middle of a lexeme, which is intolerable. Line-spanning lexemes would be replaced with invalid or unintended lexemes. For example, if a line was broken in the middle of the C decrement operator, `--`, the result might be a subtraction operator followed by unary minus operator—not at all the same thing.

One solution made possible by Standard C [6] is to terminate the continued line with a backslash wherever the line break occurs and continue the lexeme in the first column of the next line. However, this would not be very readable. We are not accustomed to seeing lexemes split in this way. Besides, the continued lines would visually detract from our indentation.

Smart Emitter

The trick then is for the emitter, the function that handles the output of code, to recognize distinct lexemes. If a line break is about to occur in the middle of a lexeme, break the line right before it. The code generator, upstream to the emitter, could separate lexemes with a character that does not occur in the emitted language's character set. That would sufficiently distinguish lexemes from each other. However, the emitter would then have to treat each separator character the same, typically by replacing it with white space, such as a single space. Some programmers like a lot of white space, but most would agree that this would result in a tedious programming style.

But what if the emitter could recognize lexemes on its own? The code generator could implement whatever programming style it liked because it would not have to separate lexemes with a special character that was eventually (always) translated into more white space. Although more complex to implement than using a separator character, this is the approach the author took to control line width.

The code generator just has to feed discreet, logical lines of code to the emitter. They must be discreet because the emitter is not aware of the emitted language's syntax. Therefore, it cannot recognize a sequence of lexemes, such as an assignment statement, that might typically occupy a line by itself. The emitter is being told that no other lexemes are to occupy the same line (or lines, if broken) as these. The lines must be logical because the code generator sends the emitter a sequence of lexemes that it considers 'a line's worth.' The emitter may then have to break it up into two or more actual lines if it is too wide.

Proper Indentation

The most obvious way to achieve the second quality, proper indentation, is just to carry the original indentation (if there is any) over to the emitted code. But this information is typically lost as white space in the early, lexical analysis phase of a well-partitioned translator. Even if carried over somehow, the structure of the original program is often modified so much that handling the complications would be a nightmare. (For example, translating a case statement into a set of nested ifs or creating a multi-threaded version of a single-threaded program.)

Depending on the compiler's translation context (specifically, the current indent level of its input source), the code generator could pass sufficient white space to the emitter at the beginning of every line to achieve the desired indentation. However, the emitter would have to decide on its own how far to indent the continuation of too-wide lines that had to be broken, because the code generator does not know about these. Regardless of whether it could be done, this type of coupling between the code generator and emitter should be avoided as poor design because the same function is performed in two places [5].

Indentation Within the Emitter

The author chose to consolidate indentation in the emitter. The code generator only has to tell the emitter when the indentation level is increased or decreased—something it should have no trouble doing—and the emitter decides how much to actually indent each line. It is the code generator's responsibility to make certain that for every indent there is a corresponding exdent[6]. For example, after a left-brace punctuator, the code generator tells the emitter to increase the indent level (by one) and subsequently, before the corresponding right-brace, to decrease the indent level.

The code generator can also temporarily suspend the emitter's automatic indentation by instructing it to left-justify the next line. This is useful for

[6]Throughout this paper, a negative indent will be referred to as an *exdent*.

emitting labels that are the targets of `goto` statements, however harmful—one has carte blanche when emitting intermediate code.

Calling Protocol

The emitter is accessed through four functions: `set_emit`, `clear_emit`, `femit`, and `emit`. `set_emit` is called once to initialize the emitter. `clear_emit` flushes the emitter's internal output buffer. The `femit` and `emit` functions do the actual emitting. The latter two accept arguments very similar to the `fprintf` and `printf` functions defined by Standard C.

set_emit

```
EMIT_FILE *set_emit(FILE *stream, const char *string,
        unsigned num_colms, unsigned max_width,
        void (*errorf)(E_Code *),
        void (*new_linef)(EMIT_FILE *));
```

This function allocates a buffer to hold an `EMIT_FILE` control block. It returns a pointer to the buffer unless there was an error. Otherwise, it returns `NULL`. This pointer thereafter identifies the output stream for the emitter in much the same way as a `FILE` pointer does for the functions in `stdio.h`. However, the pointers are not interchangeable.

`stream` is the Standard C library stream to which the output is sent. If `NULL` is used instead, the output is sent to `stdout`.

`string` is a pointer to the character string to output for each level of indentation. `num_colms` is the number of columns the string represents. It is particularly needed when the indent string contains a horizontal-tab character, as in Example 4. If `NULL` is given as the string argument, the default indent string (three spaces) is used, and `num_colms` is ignored.

`max_width` is the maximum width of the emitted line. If zero is given, this width defaults to 80.

The arguments `errorf` and `new_linef` are pointers to functions that allow the emitter to be expanded without modification. If they are `NULL`, default functions are used that have no effect. If specified, the function pointed to by `errorf` is called with an error code when an error occurs. Errors occur when the indent level goes negative (more exdents than indents) or an invalid Standard C lexeme is encountered. (It is still emitted upon return, however.) If `new_linef` is not `NULL`, the function to which it points is called before each new line is output. This gives an opportunity, for example, to intermix source-language statements as comments or `#line` preprocessor directives with the emitted C source. The function is passed a pointer to the corresponding `EMIT_FILE` control block so that it has access to the emitter's current indent level, line state (not described herein), and output stream, among other things.

`clear_emit`

```
void clear_emit(EMIT_FILE *buffer);
```

Besides flushing the emitter's output buffer, `clear_emit` releases the control-block buffer that was allocated by `set_emit`. (The output buffer is contained in the control-block buffer.) Its sole argument is a pointer to the buffer. If `NULL` is given instead, it will use the buffer that was last allocated by `set_emit`.

`femit` and `emit`

```
void femit(EMIT_FILE *buffer, const char *format, ...);
void emit(const char *format, ...);
```

The difference between `femit` and `emit` is that `femit` has an extra argument (`buffer`) that points to an `EMIT_FILE` control-block buffer, in the same way that `fprintf` has a file pointer while `printf` does not. The `emit` function uses the control block last allocated by `set_emit`.

The ability to identify a specific control block with `femit` allows several output streams to be in use at the same time, e.g., to generate a header file at the same time as a source file. Using `emit` means that one does not have to worry about such details if there is no need for multiple output streams.

Although both use one output stream, compare Examples 1 and 2. C source files that contain calls to the emitter functions must first include `emit.h` as shown.

```
#include <ctype.h>
#include "emit.h"

main()
{
        int c;

        set_emit(NULL, "", 0, 1, NULL, NULL);

        while ((c = getc(stdin)) != EOF)
                emit("%c", isprint(c) ? c : ' ');

        clear_emit(NULL);
}
```

Example 1. Print one lexeme per line with no indentation

```
#include <ctype.h>
#include "emit.h"

static unsigned line_number = 0;

static void on_error(E_Code ec)
{
        fprintf(stderr, "\n** %s error on line %u **\n",
            ec == EC_LEV ? "level" : "lexical",
            line_number);
}

static void on_new_line(EMIT_FILE *efp)
{
        if (efp->line == EL_FIRST)
              fprintf(efp->out_fp, "Lexeme Listing\n\n");
        ++line_number;
}

main()
{
        int c;
        EMIT_FILE *efp;

        efp = set_emit(stdout, "", 0, 1,
            on_error, on_new_line);

        while ((c = getc(stdin)) != EOF)
                femit(efp, "%c", isprint(c) ? c : ' ');

        clear_emit(efp);
}
```

Example 2. More elaborate version of Example 1

The remaining arguments to `femit` and `emit` are just like those for the `printf` family of functions—a format string followed by zero or more arguments. In fact, since a member of that family, `vsprintf`, does the formatting for the emitter, one can use whatever `printf` conversion features are present in the library that is linked with the emitter.

Embedded Control Characters

Once a line has been formatted, four embedded control characters (horizontal tab, backspace, new-line, and carriage return) have special meaning to the emitter. The horizontal-tab character, `'\t'`, increases the indent level by one.

Subsequent lines will automatically be indented (in the same way) one more level. The backspace character, `'\b'`, decreases the indent level by one so that subsequent lines will be indented one less level. The carriage-return character, `'\r'`, temporarily overrides the current indent level causing only the next line to be left-justified. (Subsequent lines will go back to being indented.) The new-line character, `'\n'`, terminates the current line and causes subsequent lexemes to be output on the next line, with indentation based on the previous use of horizontal-tab and backspace characters. No other control characters are allowed. Example 3 shows the use of all emitter control characters. Listing 1 shows what the code in Example 3 would generate.

```
labelNo = 5;
set_emit(NULL, "     ", 5, 80, NULL, NULL);
emit("\t\t\t");                     /* Not normally done */
emit("if (id > MAXID)\n{\n\t");     /* 2 lines, 1 call */
emit("goto lbl%u;\n", labelNo);     /* Var part of lexeme*/
emit("\b}\rlbl%u:\n", labelNo++);   /* Left justify label*/
emit("resetId(&id);");              /* Not justified */
emit("    /* A comment */\n");      /* 1 line, 2 calls */
emit("\b\b\b");                     /* Not normally done */
```

Example 3. The emitter control characters in use

```
               if (id > MAXID)
               {
                    goto lbl5;
               }
lbl5:
               resetId(&id);    /* A comment */
```

Listing 1. Output from Example 3

Automatic Line Breaks

Besides these explicit format commands to the emitter, there is an automatic operation that is solely under the control of the emitter. This is the line-break operation. If a C lexeme will not fit on the current line, it is automatically placed on the next line, indented one extra level. (This seems customary, although double indentation is occasionally used. It is a matter of style.) If necessary, this continuation line may also be continued on more lines, but not with further indentation. A new line, indicated by a new-line or carriage-return character, terminates the extra, line-continuation indentation.

Example 4 shows a situation where a line would have been broken in the middle of a logical AND operator. As Listing 2 shows, the emitter backs up to the beginning of the operator, breaks the line there, and continues the logical line, indented, on the next physical line. Note that the subsequent line goes back to the pre-continuation-line indent level.

```
set_emit(NULL, "\t", 4, 20, NULL, NULL);
emit("\tif (id > MAXID && isLast)\n{\n\t");
```

Example 4. Automatic line break just waiting to happen

```
if (id > MAXID
    && isLast)
{
```

Listing 2. Broken line

Although unusual, if a lexeme plus the current indentation is longer than the maximum line width, it is still output, even though the line is wider than the maximum width. There is no other choice. This would probably happen a lot with Examples 1 and 2, where the maximum line width is set to 1.

The Lexical Analyzer

Although it is not the purpose of this paper to describe the implementation, it might be useful to mention a few things about the lexical analyzer used by the emitter.

The lexical analyzer is implemented as a deterministic finite automaton (DFA) [1] that recognizes spaces, the four emitter control characters, and preprocessor tokens. Preprocessor tokens are the minimum lexical elements in Standard C's translation phases 3–6 that can be separated by white space [6] and, therefore, the emitter's line breaks.

The author took liberties with Standard C regarding comments. Instead of looking for C lexemes in comments or just skipping them entirely, the DFA shifts to just recognize spaces, the four emitter control characters, and non-space character sequences. It then shifts back to recognizing C lexemes after the comment. This simple form of word-wrapping inside comments, like that used in word processing, results in a much more natural division of comments into several lines when it is needed.

Using It

The emitter is straightforward to use. However, it is difficult to come up with a meaningful, brief example program that uses it, as an example. A real source-code generating program, such as a translator, might make an occasional call to `emit` with multiple-statement sections of target code and then lots of calls with little bitty pieces—a relational operator here, a bracket there, part of an identifier now, the rest later. Since the kind of programs that could use this emitter tend to be fairly complex, even a relatively simple meaningful program would overwhelm the parts relating to the emitter.

As with language development in general, be aware of the difference between the source and execution character sets. To emit the following statement:

```
printf("Hello\n");
```

one might, at first, expect to use a call to the emitter such as:

```
emit("printf("Hello\n");\n");
```

However, one quickly realizes that this will not achieve the desired goal. The proper call is, of course:

```
emit("printf(\"Hello\\n\");\n");
```

Limitations

Nested comments are not allowed but they are rarely needed in emitted code anyway.

Header names used in `#include` directives are not recognized as distinct (*h-char* and *q-char*) lexemes. Although this causes no problem with most quoted names (they are treated just like strings), there is theoretically a problem with an angle-bracketed name. It is seen as `<` and `>` typically enclosing identifiers and a period. However, since `#include` directives tend to hug the left-hand margin, this is rarely a practical concern. They have little to fear from line breaks.

If a `#define` directive is wider than the maximum line width, the emitter breaks it into multiple lines as usual. This is a problem because such breaks must be marked with a back-slash character, `\`. Since macro definition directives (and preprocessor directives in general) are more predictable than the rest of emitted C, this is usually not a problem. The code generator must generate back-slash-separated lines itself and not depend on the emitter to do so.

Currently, of the character set defined for Standard C source files, the vertical tab and form-feed characters are invalid as input to the emitter.

Standard C trigraph sequences are not recognized. This decision was made based on the added complexity they would have imposed on the DFA versus their benefit to most users of the emitter.

Possible Enhancements

The first enhancements the author would provide are:

1. The ability to disable and re-enable the emitter's lexeme enforcement in order to break the rules occasionally and emit something that is invalid in the emitted language;
2. Allowing the form-feed character as input into the emitter.
3. To be more consistent with the standard library's `printf` family of functions, perhaps printed-character counts should be returned and other emitting functions implemented, e.g., the analogue to `puts`.

Though the author chose not to, partly for portability reasons, the emitter's lexical analyzer could be generated with lex [4]. This would make the lexical analyzer easier to maintain than the existing finite state, hand-coded machine [3]. The next big step for the emitter would be to format (break and indent) lines based on the syntax of the language, without any help from the code generator. To do this would require parsing its input, possibly with a parser generated by yacc [2]. However, this would be going too far. The size and complexity of the resulting emitter could approach that of the rest of the main program.

Possible Uses

The emitter described here could be used in the back end of a source-code reformator, e.g., a 'pretty printer,' that does not translate between languages. It just rearranges source code according to some prescribed style.

It could also be used to emit code in an application generator. For example, once a user has interactively defined the layout and behavior of a set of screens with such a program, the source code for an application is automatically generated, ready to compile.

The most obvious uses, however, are in a source-language translator where Standard C is the target language, e.g., a Pascal-to-C translator, and in a preprocessor where Standard C is an intermediate language.

Notes

Although this facility has always been written in C, it first supported TAL, Tandem Computer's system language. It was developed simultaneously on a PC running XENIX and a Tandem VLX running its proprietary operating system using their respective C compilers. It was then ported to a DEC 3500 running VMS and DEC's VAX C compiler. After that, the shift from supporting TAL to C occurred, along with an opening up of the implementation of the DFA to eliminate some redundant processing by remembering DFA state between emitter calls [3]. More recently, it was ported to a PC running MS-DOS and Borland's Turbo C compiler.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Addison-Wesley*, Reading, Massachusetts, 1986.

[2] Stephen C. Johnson, *Yacc: Yet Another Compiler Compiler. Bell Laboratories Computing Science Technical Report 32* (31 July 1978).

[3] Douglas W. Jones. *How (Not) to Code a Finite State Machine*, SIGPLAN Notices, vol. 23, no. 8, August 1988.

[4] M.E. Lesk and E. Schmidt. *Lex – A Lexical Analyzer Generator*, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[5] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*, 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[6] American National Standards Institute, *American National Standard for Information Systems—Programming Language C*, X3.159-1989, approved December 14, 1989.

Paul Long is an independent computer consultant with a background in software-development tools, telephony, database applications, and image processing. The source to the emitter described in this paper is available from the author for a small handling charge. He can be reached at (503) 697-7965 and on CompuServe at 72607,1506 *or via the Internet at* 72607.1506@CompuServe.com.

∞

9. Miscellanea

compiled by **Rex Jaeschke**

Constructing Header Name Tokens

In §3.8.2, Source File Inclusion, page 89 lines 9–17, the Standard reads:

> "A preprocessing directive of the form
>
> `# include` *pp-tokens new-line*
>
> (that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms. The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined."

I have been studying this and related sections of the standard to try and determine whether this directive format can be used at all in a strictly-conforming program. As such, I wrote a number of small test cases and ran them through 5 compilers claiming to be either standard-conformant or 'very close to it.' I'll refer to the compilers as C1–5.

Test 1:

```
#define M1 <stdio.h>
#include M1
```

All five compilers found the header `stdio.h` and processed it. Let's look closer at what is going on here. In §3.1.7, Header Names, page 33 lines 33–34, the standard reads:

> "Constraints : Header name preprocessing tokens shall only appear within a `#include` preprocessing directive."

From this, one can deduce that a header name token *only* exists in the context of an `#include` directive. Specifically, in the definition of macro `M1`,

`<stdio.h>` is *not* recognized as a header name. Instead, it is seen as the token sequence

```
{<} {stdio} {.} {h} {>}
```

When `M1` is called this set of tokens is combined (in an implementation-defined manner) to form a token that is expected to look like a header name. Since implementation-defined behavior must be documented I went in search of a description of how the 5 compilers constructed this new token. I found almost nothing in their documentation, and certainly not enough to tell me would happen in the rest of my test cases.

Test 2:

```
#define M2 <  stdio /*...*/ .  h  >
#include M2
```

Compilers C1 and C2 ignored all white space and formed the header name `<stdio.h>` and found the correct header. C3 and C5 reduced each lot of contiguous white space to a single space finishing up with `< stdio . h >` (which did not map to an existing header). C4 took a similar approach although it stripped off leading and trailing white space giving `<stdio . h>`. Three of the compilers seem to differentiate between the absence and presence of white space between tokens when the final header name is constructed.

Can all the white space legitimately be ignored? Well the standard simply says "The method by which a sequence of preprocessing tokens ... is combined into a single header name preprocessing token is implementation-defined." There is no mention about any white space that may be separating these tokens.

Can any of the white space legitimately be retained? You can argue "Yes" since it is implementation-defined as to how the header name is constructed. You could also argue "Yes" since nothing is stated about white space so this aspect is unspecified—anything can happen.

Test 3:

```
#define A stdio
#define B h
#define C <A . B>
#include C
```

C1 produced `<A.B>`. This is an error since iterative macro expansion should occur on the call to `C`. C2 had no problem. As before, C3, C4, and C5 kept some white space but C3 and C4 replaced the macros and C5 didn't.

Test 4:

```
#define A stdio
#define B h
#define D <A.B>
#include D
```

While Test 3 contains white space, Test 4 does not, so one might expect more predictable results. C1 and C5 produced `<A.B>` failing to expand the macros. (They should have expanded the call to `A` and `B`.) C2, C3, and C4 all expanded both `A` and `B` and the header was processed.

Test 5:

```
#define XX std ## io
#define Y h
#define Z <X ## X.Y>
#include Z
```

Based on the results from Test 4 the results of this test were predictable. C1 and C5 again failed to expand macro calls and produced `<XX.Y>`. On the other hand C2, C3, and C4 processed `<stdio.h>`.

Test 6:

```
#define M3(arg) #arg
#include M3(a.h)
```

In this much more straightforward case, all compilers formed the header name `"a.h"`, as expected.

Admittedly most of these examples are academic. However, the primary reason for this format of the `#include` directive existing in the first place is to allow the programmer to construct a header-name token from pieces. Yet such a token can never exist except when hard-coded in an `#include` directive or as the result of some magic incantations which are labelled as implementation-defined. From this I deduce that one *cannot* portably rely on the outcome of *any* usage of this directive format. Even the simplest case (Test 1) might not work since the way in which the tokens are put together is implementation-defined (although it hard to image what other reasonable token could be formed here). And in the situation where the macro is defined on the compilation command-line, the command-line processor may treat the `<` and `>` characters as command-line redirection characters instead. The bottom line then is that the construct

```
        #ifdef X
                #include <a1.h>
        #else
                #include <a2.h>
        #endif
```

is strictly-conforming, but

```
        #ifdef X
                #define M <a1.h>
        #else
                #define M <a2.h>
        #endif

        #include M
```

is not. On the other hand, if headers with names of the form `"..."` were used instead of `<...>`, either approach is strictly-conforming since tokens of this form are treated as string literals from the beginning until finally being 'converted' into a *q-char* sequence.

Calendar of Events

- July 8–10, Tutorial and workshop on **High Performance Compilers** – Location: Portland Marriot Hotel, Portland, Oregon. For information regarding course content contact the instructor Michael Wolfe at (503) 690-1153 or *mwolfe@cse.ogi.edu.*

- 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 16–17, 1991 **Hot Chips Symposium III** – Location: Stanford University, Stanford, CA. This IEEE-sponsored conference will consist of presentations on high-performance chip and chip-set products, and related topics. It is directed particularly at new and exciting products. The emphasis is on real products, not academic designs. For further information contact Martin Freeman at (408) 991-3591 or *mfreeman@sierra.stanford.edu.*

- 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 24–27, 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. 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. For more information, contact the Vice-Chair William M. (Mike) Miller, P.O. Box 366, Sudbury, MA 01776-0003, (508) 443-7433 or *wmmiller@hplabs.HP.com*.

- November 14–16, 1991 **Supercomputing Debugging Workshop '91** – Location: Albuquerque, New Mexico. This workshop will be held in conjunction with Supercomputing '91. For information, contact one of the following: Jeffrey S. Brown, (505) 665-4655 or *jxyb@lanl.gov*; Peter Rigsbee, *par@cray.com*; or Ben Young *bby@craycos.com*.

- December 1–5, 1991 **Third IEEE Symposium on Parallel and Distributed Processing** – Location: Dallas, Texas. For more information contact Vijaya Ramachandran, (512) 471-9548 or *spdp@cs.utexas.edu*; or Greg Pfister, (512)823-1589 or *pfister@austin.iinus1.ibm.com*.

- December 11–13, 1991 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Milan, Italy. WG14: Contact the US International Rep. Rex Jaeschke at (703) 860-0091, or *rex@aussie.com*, or the convenor P.J. Plauger at *pjp@plauger.com* for information. X3J11: Address correspondence or enquiries to the vice chair, Tom Plum, at (609) 927-3770 or *uunet!plumhall!plum*.

- January 6–10, 1992 **Numerical C Extensions Group (NCEG) Meeting** – Location: In the Dallas, Texas area, hosted by Convex. Note that this will *not* be a joint meeting with X3J11.

- January 7–10, 1992 **Workshop on Parallel Programming Tools** – Location: Kauai, Hawaii. This event is the Hawaii International Conference on System Sciences – 25 (HICSS-25). For information, contact Dr. Hesham El-Rewini at (402) 554-2852 or *rewini@unocss.unomaha.edu*.

- January 19–22, 1992 **Principles of Programming Languages** – Location: Albuquerque, New Mexico. This is the 19th Annual ACM SIGPLAN-SIGACT symposium. For information, contact Andrew Appel at (609) 258-4627 or *appel@princeton.edu*.

- 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

- **Associated Computer Experts** bv (ACE) of the Netherlands has integrated their EXPERT C (and other language) compilers into the Motorola VMEexec Real-Time environment. *uunet!doeke@ace.nl*

- Oakley Publishing, publisher of the *Programmers' Journal*, has acquired the *C Gazette*. Bobbi Sinyard (503) 747-0800.

- The third edition of *C: A Reference Manual* by Harbison and Steele is now available from Prentice Hall. Call (201) 767-5937 for a 15-day free trial.

- **paracom, inc**, vendor of Inmos transputer systems and software has been renamed to **parsytec**. (708) 293-9525.

- To order a copy of the ANSI C standard (ANSI X3.159-1989) in Canada, contact:

  Foreign Standards Sales Section
  Standards Council of Canada
  350 Park St., Suite 1200
  Ottawa, Ontario
  K1P 6NT
  (613) 238-3222

  The cost is about Can$93.

- The ISO draft C Bindings for **GKS/C** and **GKS-3D/C** were registered as a DIS in July 1990. Both bindings were approved by all P members. Only 4 members had some comments, almost none of which seemed controversial. As a result, the final standard is expected soon. For information on these bindings contact Miente Bakker in the Netherlands at *miente@cwi.nl*.

- **ANSI has moved**. The new address is:

  American National Standards Institute
  11 West 42nd Street
  New York, NY 10036
  (212) 642-4900
  Fax: (212) 398-0023
  Sales Fax: (212) 302-1286

- **Digital Equipment Corp.** has announced V1.1 of the **PDP-11** C compiler.

- **Sun** has announced V1.1 of their Sun C compiler which is now standard-compliant.

- **Zortech** has announced a native mode C and C++ compiler for the Apple Macintosh. For further information call: North America, (617) 937-0696 or Fax (617) 937-0793; England, (0)81 316-7777 or Fax (0)81 316-4138.

- **P.J. Plauger** has produced a portable implementation of the entire library of Standard C (ANSI X3.159-1989 and ISO/IEC 9899:1990). The math functions have at most 2 bits of error. An open-ended set of locales can be supported as text files. A Kanji locale is provided, as well as a "simulated widechar" locale for Westerners. The entire source code, special test programs, and full explanation are to be published as a textbook by Prentice-Hall. (Anticipated release date: Summer 1991.)

  Plauger has granted full rights to each reader to incorporate functions from this library royalty-free into bound-binary (i.e., executable) programs, provided only that Plauger's copyright notice is embedded somewhere in each copy. The reader may keyboard the functions by hand, or obtain a diskette from the **C Users' Group** at (913-841-1631).

  Compiler vendors who wish to distribute source code or linkable libraries containing some or all of the functions from Plauger's library can license these rights from **Plum Hall** at (609) 927-3770 or *plum@plumhall.com*).

- The **Japanese agency JMI** has signed a letter of intent to use **Plum Hall's suite** for official validation in Japan. For further information on this suite, contact Plum Hall at (609) 927-3770 or *plum@plumhall.com*).

∞