The Journal of

C Language Translation

Volume 1, Number 1

June, 1989

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

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

Editorial: Address all correspondence to 1810 Michael Faraday Drive, Suite 101, Reston, Virginia 22090 USA. Telephone (703) 860-0091. Electronic mail address via *uucp* is *uunet!aussie!jct*.

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

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

Contents

1. Pointers and Addresses – P.J. Plauger **1**

A look at the address spaces provided by several popular architectures, and a discussion about pointers and addresses in a standard-conforming implementation.

2. Numerical C Extensions Group Status – Rex Jaeschke **9**

A report on the first NCEG meeting and future directions of the group.

3. A Solution to Name Space Pollution – Sue Meloy **12**

H-P's solution to resolving name space issues in implementations that must support ANSI as well as other environments such as POSIX, X/Open, and "in-house" extensions.

4. ANSI/ISO Meeting Report – Jim Brodie **17**

A report on the status of the X3 Secretariat Vote, the recent ANSI and ISO meeting, and the future role of X3J11.

5. Adding Complex Arithmetic to C – Tom MacDonald **20**

A proposal to add a complex data type, an operator, and supporting library routines to the Standard C environment.

6. Pragmania – Rex Jaeschke **32**

A discussion of pragma-related issues and a look at the pragmas defined by WATCOM's V7 DOS compiler.

7. Standards Forum: Type Qualifiers – Jim Brodie **48**

A discussion of the history behind the `const` and `volatile` qualifiers and a look at the changes they force on handing and checking type information, and in code generation.

8. Parallel Programming: Linda Meets C, Part I – J. Leichter .. **55**

An introduction to the Linda programming model for developing explicitly parallel programs.

9. Miscellanea – Rex Jaeschke **61**

Subscriber questionnaire results, electronic polls, upcoming events, products and services, and a note on wide character constants.

10. Books and Publications **68**

Reviews of *Theory of Computation* by J. Glenn Brookshear, *Standard C: Programmer's Quick Reference Guide* by P.J. Plauger and Jim Brodie, and *ANSI C: A Lexical Guide* by Mark Williams Company.

1. Pointers and Addresses

P.J. Plauger

Abstract

One of the largest areas of variation among modern computers is the way they address memory. Since C supports a broad assortment of pointer data types, it is hard for the C implementer, or sophisticated programmer, to hide from the consequences of this variation. This paper describes what you can count on to be true about pointers and addresses in any conforming implementation of Standard C. It also warns you about variations that can occur among implementations.

Early History

Most of us are familiar with the general concept of "address space." An address space is the set of all possible values that can be used to specify where information is stored in a computer memory. The Intel 8080/8085 (and Zilog Z80) offer an extremely simple example. These chips use a 16-bit number to specify which of 65,536 possible 8-bit bytes they wish to access from computer memory. There is no way to distinguish the fetch of an instruction byte from an access to an arbitrary data byte. Hence, we say that this family has a 65,536-byte address space, and all sensible implementations of C represent all pointers as 16-bit (two-byte) values.

For a slightly more complex example, let's look at another old architecture. The PDP-11 series also develops a 16-bit address for any memory access. A memory management unit (MMU) typically maps each 16-bit program address to a 22-bit physical memory address, but that doesn't alter the programmer's view of the world. It is sufficiently difficult to manipulate the MMU from a program that PDP-11 programmers just think in terms of having a 65,536-byte address space occupying a small part of a memory that may be up to 4 megabytes. Only a few diehard RSX programmers usually master the window manipulations needed to access the larger memory.

Where the PDP-11 architecture does shine through to each running program is on the processor models numbered 44 and higher. These generally support separate address spaces for instructions and data. When the program wishes to access bytes of code from the executing program, such as a C function, it instructs the MMU to fetch from "I-space." When the program wishes to access bytes of manipulable data, such as a C data object, it instructs the MMU to

fetch from "D-space." Thus, it is easy to organize your PDP-11 program into upwards of 65,536 bytes of executable code and an equal quantity of manipulable data. In a C program compiled to take advantage of separate I- and D-spaces, the values stored in function pointers come from a different address space than the values stored in data object pointers. A function pointer can coincidentally have the same numeric value as a data object pointer and still point to a distinct entity. Moreover, you cannot inspect (or corrupt) your C functions as if they were data. Nor can you build executable code in a data object and contrive to execute it.

The Motorola 68000 family also supports separate code and data spaces, but this feature seems to be much less used than on the PDP-11. Perhaps this is because each address developed by the program is 32 bits. You seldom need to separate the spaces just to double the number of bytes you can access. The other advantages you get from separating the spaces evidently are not sufficiently compelling to convince most system designers to add the extra complexity in the hardware.

All of these machines that I have discussed so far are alike in one important respect. They all represent program addresses to byte resolution. Since C encourages you to treat an arbitrary data object alternatively as an array of characters, and since C programs often traffic heavily in character data, it's important to be able to manipulate data efficiently a byte at a time. Fortunately, most modern computers use byte resolution addressing.

Unfortunately, there are some important exceptions. Machines that naturally address a multi-byte word often must indulge in special representations for pointers to data objects smaller than a word. In some cases, pointers to character data objects might even occupy more bits than pointers to larger data objects. Standard C has endeavored to include such implementations, and not as second class citizens. Whether arbitrary programs port easily to such implementations, or whether they run well when ported unchanged, is a different matter. I suggest you write important programs with word-resolution machines in mind, but be prepared to tune and tailor if you ever have to port to such an implementation.

The Notorious 8086

IBM's decision to build its line of PC's around the Intel 8086 family has done more to sensitize people to address space issues than anything else that I can imagine. That architecture also develops a 16-bit program address for each memory access, but that's just the start of it. Each program address is accompanied by a reference to one of four (or more, on later models) "segment registers." The actual memory address is composed from the 16-bit program address and information in the segment register. In the earliest models (8086, 8088, and 80186), the segment register merely contains a 16-bit number that is shifted left four places and added to the program address to develop a 20-bit

physical address. That way, you can reach up to 1 megabyte of storage. Set aside about 384 kilobytes of high memory addresses for special purposes and you have the magic 640 kilobyte program size limitation for which DOS has grown infamous.

Later models of the 8086 family (80286, 80386) can enter a mode where the segment register value is used to designate a table entry that gives considerably more information about how to find the physical memory. The table entry provides access control information, for example, so you can write-protect chunks of code. It also holds a much larger number to add to the program address, so you can build systems that make use of up to 16 megabytes of storage.

The simplest way to use the 8086 is as a system that has four 65,536-byte address spaces. With this scheme, the segment registers form a sort of MMU that the program leaves largely untouched. That seems to be the original intent of the designers of the 8086 architecture. All pointers were to remain 16 bits, just like on the Intel 8085. Your program determined which address space, and hence which segment register, to use from context. It is no accident that Pascal can traffic pretty comfortably in four data spaces – one each for code, static data, automatic data (stack), and controlled data (heap). *The Intel 8086 is a Pascal engine.*

Unfortunately for those plans, the Intel 8086 has become a C engine. And C, given its PDP-11 heritage, can distinguish only two separate address spaces from context. All functions live in one space, all data objects live in another.

Sure, if you access a data object by name the compiler can tell whether it has static or dynamic lifetime. That way, it could know whether to reference the segment register for static data or the one for automatic data. And if it knew that a pointer was set by a call to `calloc`, it could know that the pointer points into the heap. (Pascal knows that *all* pointers point into the heap, at least in a strict implementation of standard Pascal.)

But once you take the address of a data object and store it in a pointer, all bets are off. Like so many things your children drag home, you don't know where it's been. That means that whatever you store in a data object pointer, it must be sufficient for you to access a static, automatic, or controlled data object. The easiest way to do this on an 8086 is to pack all flavors of data objects in one 65,536-byte address space. The only problem is, you can then run programs no bigger than those that fit on an old PDP-11. And our ambitions outgrew that architecture years ago.

A more powerful alternative is to make pointers big enough to store both address and segment register information. The architecture doesn't support storing (or making use of) a two-bit segment register designator. Besides, once you go beyond 16 bits, the next sensible stopping point is 32 bits. The architecture does support storing (and making use of) the 16-bit segment register contents alongside the 16-bit program address. So the natural thing to do is to compile C programs to represent pointers in 32 bits. Your program then spends considerable time loading and storing segment registers so that your measly four windows slide properly around within the larger address space.

Naturally, you pay a penalty when using 32-bit pointers instead of 16-bit pointers. Code size essentially doubles and execution time along with it. And, of course, the storage required for pointer data objects also doubles. A good bit of the extra headroom we won over the 8085 and PDP-11 is sacrificed to code space inefficiencies. One cultural side effect of these severe penalties is that people adopt heroic measures to avoid them.

You have a relatively easy out if all the functions in your program fit in one 65,536-byte window. You can then get away with 16-bit pointers for functions (which you probably don't use very often anyway), alongside the 32-bit pointers you need for data objects. C has never explicitly required all pointers to have the same representation. On many early implementations that happens to be the case, and a few programs doubtless took advantage of the coincidence. But Standard C has been clear on the subject for many years. Between the needs of the marketplace and the guidance of the evolving C standard, most programmers have long since cleaned up their act, and any code they hope to keep marketable.

Occasionally, you will find a program that has lots of functions, but only needs one 65,536-byte window for all of its data. In this case you go to 32-bit function pointers and 16-bit data pointers.

All these options drive compiler vendors gaga. Compilers for the PC are dripping with compile-time switches and in-line pragmas to help the programmer communicate his varied wants. And you can't easily mix code compiled with different pointer sizes. That means the compiler vendor is obliged to ship an assortment of libraries for linking with the various addressing models.

Now for the worst part. Some programmers are so desperate to avoid the inefficiencies that come with large pointers that they insist on writing hybrid programs. These traffic in pointers of mixed sizes. Now, you just plain can't do this in Standard C. Nevertheless, where there is a need there is a way. What the serious compiler vendors have done is extend C by adding pointer qualifiers such as NEAR and FAR. These provide direction to the compiler so that it can know what size to make certain nonstandard pointers. The semantics of these extensions are generally shaky. Mostly, they are offered on the caveat emptor basis that is customary for extensions that people want more than they need.

I believe that X3J11 was wise not to enfold hybrid pointers within Standard C. They aren't even listed as a common extension, though perhaps they should be. Having thought through the semantics rather thoroughly years ago (with the Whitesmiths Version 3 compilers), I can attest that you pay a high price in complexity for a small payoff in performance. That's not the stuff of standards.

Rules of the Game

Most of this history is doubtless familiar to you. I recite it here partly to fill in any gaps you may have, and partly to recreate the climate in which X3J11 made a number of key decisions about Standard C. To summarize the key decisions relating to pointers and addressing, I now state a number of principals, not all of which are obvious from a quick reading of the draft. Here goes:

1. **Standard C supports $2\frac{1}{2}$ address spaces.**

   You know about functions and you know about data objects from the earlier discussion. The extra "half" address space is for data objects whose address you never determine in your program. This includes, naturally, all data objects declared with storage class `register`. There is no valid way you can take the address of some other data object and use that address value to compute a pointer into one of these anonymous data objects. Hence, the compiler is at liberty to tuck these data objects in a separate address space. It can place anonymous data objects in machine registers or address them in other funny ways.

   All other data objects must be reachable by dereferencing an arbitrary data object pointer. This means that even if you know a shortcut for accessing certain data objects (it may have a 16-bit displacement off the code segment register, for example), you must also be able to store its address in a data object pointer (you write the FAR pointer composed of the 16-bit displacement and the value stored in the code segment register). Once you start trafficking in different pointer representations for the same data object, be wary of the rules for comparing pointers (described below).

2. **A pointer representation is not necessarily the same as an address space.**

   A perfectly valid implementation of C could have 8-bit pointers. The pointer value is the index into a table of data object descriptors, which contain full addresses and possibly other information on the data object. Far more commonplace, a pointer can have excess bits that do not participate in forming the actual storage address. Both situations can cause surprises when comparing pointers and when converting glibly between integers and pointers.

3. **A null pointer is not necessarily all bits zero.**

   This is unlikely, but permissible. Standard C requires that a pointer assigned an integer zero be a null pointer. It must point to no valid data object (or function) in your program. It must compare equal to integer zero. Neither of those requirements prevents an implementation from picking an arbitrary bit pattern (or patterns) to represent the null pointer. It does cause trouble with, say, static initializers. The implementation may have to replicate its funny idea of a null pointer many times to fill

out a partially initialized large array of pointers. But it can do so if it chooses.

An implementation can also represent 0.0 as other than all bits zero. The same caveats apply. I doubt that any real life implementer would go to the trouble to indulge in either of these liberties, but be warned. Either or both may just burn you some day.

4. **All function pointers have the same representation.**

   Standard C lets you cast any function pointer to any other type of function pointer without loss of information. You must cast it to a function pointer of type compatible with the function it points at before you dare use it in a function call, but you can copy it all over the place beforehand. This rule provides one small island of sanity in the world of pointers.

5. **Every data object pointer can have a different representation.**

   Actually, there are a few sets of pointers that must have the same representation among all members of the set, but the overriding rule is that variety reigns. Pointers to all character types have the same representation, which is the same as for pointer to `void`. (Remember that this is the only representation to which you can cast an arbitrary data object pointer with no loss of information.) Pointers to `const` whatever have the same representation as pointers to non-`const` whatever, where the two "whatever"s have compatible type. The same holds true for `volatile`, of course. And pointers to incomplete types have the same representation, perforce, as pointers to their completed data object types. Beyond these guarantees, don't play games with pointers in unions, or with pointers not coerced by a function prototype on a function call.

6. **Pointers and integers are incommensurate.**

   You can subtract two 8-bit pointers and get a 32-bit integer difference. You can subtract two 96-bit pointers and get a 16-bit integer difference. You are asking for trouble if you mix anything but integer zeros with pointers in comparisons, assignments, and initializations. Don't even think about playing games with unions. Forget anything you ever used to believe about relationships between pointer and integer representations.

7. **The result of the `sizeof` operator can always be represented as an `unsigned long`.**

   The actual integer type is, of course, `size_t`. But since that is merely a type definition, it must be the same as one of the unsigned integer types. The largest of the unsigned integer types is `unsigned long`. Q.E.D.

   But watch out when you subtract two pointers. True, the type of the result is `ptrdiff_t`, but the result can still overflow when represented as that type. On a machine with 16-bit pointers, you can sometimes declare 50,000-byte character arrays. The difference between pointers pointing at

the beginning and end of such an array cannot be properly represented as a 16-bit signed integer. You can trust that you can represent the result of `sizeof`, but you cannot always represent the difference between two arbitrary pointers.

8. **The type qualifiers `const` and `volatile` modify only lvalues.**

   (Brodie also discusses this matter in a separate paper in this issue.) Actually, the type qualifiers squat midway between the universe of types and that of storage classes. There are good reasons why Standard C introduces them as type qualifiers, but that can still be confusing. Just assume that `const int` becomes `int` when you replace an lvalue by its stored value, in an rvalue context.

9. **`const` (and `volatile`) data objects occupy the same address space as unqualified data objects.**

   I have given all the reasons why this must be true above. I state the (apparently) obvious here for emphasis. Compelling as it is to think you can tuck your read-only tables in with your read-only code, it ain't always safe. Make sure that you never have to develop an address to store in a pointer data object. Or if you do, make sure that the address will support proper read accesses to the data object. This is one of the unfortunate consequences of the weaker semantics for pointer to `const` that X3J11 settled on at the eleventh hour.

10. **Pointers to the same data object (or function) must compare equal.**

    This is true even if the two pointers are not bitwise identical. The 8086 family offers boundless opportunities for the kind of aliasing that can make pointer comparisons horrendously expensive. X3J11 decided, however, that the semantic price was too high to allow cheap comparisons to fail. An implementation must either not traffic in multiple representations for the same address, or it must do the extra work during comparisons to get the correct answer.

11. **Pointers to different data objects (or functions) must compare unequal.**

    Naturally.

Conclusion

It's a strange world out there, in the land of computer architectures. Standard C has not succeeded in embracing all popular architectures with its rather flat addressing model. But then, few members of X3J11 hoped to do so. I believe we did a good job of identifying the most sensible existing rules, and of codifying existing practice rather more clearly than in the past.

What we need to do now is gather more experience with the extensions that people have felt moved to add to C's addressing model. Perhaps we can still tame NEAR and FAR pointers and bring them into the fold. The recent meeting of the Numerical C Extensions Group demonstrated that aliasing issues are far from resolved, or dead. Newer chips such as digital signal processors are demanding accommodations from C so that it can serve as the base language for their special needs.

It will be fun to see what the conventional wisdom evolves to for addressing in C five years from now.

P.J. Plauger is a Chief Engineer for Intermetrics Inc. He also serves as secretary of X3J11, convener of the ISO C working group, and as Technical Editor of The Journal of C Language Translation. Dr. Plauger can be reached at uunet!aussie!pjp.

∞

2. Numerical C Extensions Group Status

Rex Jaeschke
NCEG Convener

Introduction

When I conjured up the idea for an ad hoc group to define numerical extensions to C earlier this year, I had no idea as to what the reaction would be. The evidence is now clear that this endeavor is seen as being very worthwhile. Not only have more than 90 people asked to be added to the contact database, but 30 of them attended the one-and-a-half day meeting at Cray Research on May 10–11.

The backgrounds of the attendees were diverse. The supercomputing industry was represented by Cray, Convex, Supercomputer Systems, and Thinking Machines. The IEEE community was well represented by Hough (from Sun), Cody (from Argonne Labs), and Thomas (from Apple.) Other organizations represented included Unisys, Microsoft, Digital Equipment Corporation, H-P, CDC, IBM, Solborne, Farance, University of Minnesota, Intermetrics, and Information and Graphics Systems. The digital signal processing industry was represented by Analog Devices. LLNL, Army BRL, and Polaroid Corporation represented the user community. Dennis Ritchie from AT&T also participated.

There was no real sentiment that we deliberately go against the direction established by ANSI C. In fact, quite the contrary. However, it was recognized that some of ANSI C's constraints may impede our activities and result in possible conflicts. The whole issue of `errno` and formatted I/O of NaNs and infinity are examples.

The Issues

The main purpose of the meeting was to identify and prioritize the principal technical issues. The group then voted on each topic, indicating high or medium (or no) priority. The high priority votes were weighted twice as much as the medium, and the following list of priorities resulted.

Main Numerical Issues	
Topic	*Priority*
aliasing	29
vectorization	27
complex	27
variably dim arrays	25
IEEE issues	24
exceptions/`errno`	24
`float`/`long double` library	23
parallelization	22
ANSI `<math.h>`	21
array syntax	19
extra math functions	17
aggregate initializers	15
inter-language issues	15
wide accumulators	10
math function precision	9
non-zero-based arrays	8
numerical representation	6
new data types	4
new operators	4
function overloading	4

Another topic, "Arrays as first class objects" had a high priority (21) but after considerable debate was dropped from the list. It was agreed that its addition would likely cause great confusion among existing C programmers.

Formation of Subgroups

The bulk of the agenda time was then given to the top ten topics, each getting 20–30 minutes. For each of these topics, attendees volunteered to be primary and alternate coordinators.

The intent is that the real technical work will go on between meetings and be coordinated by the leaders of each subgroup. Then, at the following meeting, each subgroup will present the results of its work and make formal proposals as appropriate. This way, the committee can focus on the final, distilled issues rather than everyone getting involved at all levels. It will also significantly reduce the amount of paper in the mailings.

If you wish to participate in any of these subgroups it is your responsibility to contact the leaders and identify yourself, your concerns, and how you can help. If your area of interest is not listed here let me know.

Mailings and Submissions

Most people interested in NCEG appear to have an e-mail address, which should make each subgroup's job much easier in coordinating various viewpoints and proposals. However, all formal distributions will be by paper mail. Since meetings are to be once every six months, there will be two mailings between meetings. The first will occur within 4–6 weeks after a meeting and will contain minutes, new papers, and other appropriate correspondence. The second will occur about 4–6 weeks prior to the following meeting. The cut-off date for formal submissions for the September meeting is August 11.

Forward all correspondence to me (either by mail or via *uunet!aussie!rex*) and I will assign it a document number. (Note that I do not have a troff formatter.) However, do that *only* if your paper is concerned with issues other than those being handled by the subgroups. For subgroup issues, forward papers to the subgroup coordinators so they can include it in their submissions to me. The intent is to avoid excessive duplication of points and to allow the short meeting time to be used more effectively. The more formal documents we have, the slower it will go.

Tom MacDonald at Cray Research has agreed to do the mailings, at least for the interim. Frank Farance of Farance, Inc., has volunteered to be the redactor of the group's working document. Thanks to Tom and Frank. (Thanks also to Randy Meyers from DEC, who acted as meeting secretary, and to Cray for hosting the meeting.)

Formal Affiliation

There was a consensus that we become affiliated with a recognized standards organization. The final proposal was that we become a working group within X3J11. If we follow that route, it will result in our publishing a Technical Report, a non-binding report on our findings and recommendations. Getting our extensions adopted as a standard is also possible, in the long term. At this stage, I plan to ask for agenda time at the next X3J11 meeting to discuss admitting us as a working group.

In the interest of economy, the next two meetings are scheduled in the same location and week as those of ANSI C's X3J11. These NCEG meeting dates are September 19–20 (Salt Lake City, Utah), and March 7–8, 1990 (New York City.)

∞

3. A Solution to Name Space Pollution

Sue Meloy
Hewlett-Packard
19447 Pruneridge Ave.
Cupertino, CA 95014

Abstract

This article describes some of the problems that the ANSI C name space guarantees cause for implementers. Hewlett-Packard required a solution that preserves backwards compatibility for old code, and allows conformance to superset standards such as POSIX and X/Open. H-P's solution for the various name space pollution problems is presented.

The Problem

ANSI C reserves certain names for the implementation: external names defined in the "Library" section of the Standard, all names beginning with `_[_A-Z]`, and all external names beginning with `_`. All other names are available to users for their own variable, function, and macro names.

UNIX has traditionally been extremely cavalier about polluting the user name space. Since our operating system, HP-UX, is UNIX-based, it has many of the same problems. The challenge was to find a solution that would conform to ANSI requirements, still allow users to access the extra symbols if they want to, require no source or makefile changes for backward-compatibility mode, and not require support for multiple versions of libraries.

Examples

The following examples demonstrate some of the common name space conflicts.

Header Files

The first problem concerns symbols contained in the standard headers that are not defined in ANSI C. There are several aspects to this problem. They are:

- Symbols defined by POSIX that are not defined by ANSI.
- Symbols defined by X/Open that are not defined by POSIX.
- Symbols defined by HP-UX that are not defined by any of the other standards.
- Internal symbols which conflict with ANSI name space guarantees.

For example, the internal name `_iob` in `<stdio.h>` was inherited from UNIX. By itself, this symbol is not in the user name space. Identifiers beginning with a single `_` are reserved for the implementation as external names. However, the `stdin`, `stdout`, and `stderr` macros reference this name, so it could conflict with a user *local* variable name. For example,

```
#include <stdio.h>

main()
{
        static float _iob[] = {1.1, 2.2, 3.3, 4.4, 5.5};

        fprintf(stdout, "Hello, world");
}
```

With the original version of `<stdio.h>`, the `stdout` reference in the example above will pass the address of the second element of the local `_iob` (2.2) as the first parameter to `fprintf`. This is likely to cause mysterious behavior.

Another problem is when names that are part of the ANSI user's name space are defined in a superset standard. For example, POSIX defines the `fdopen` function. This declaration must be present in `<stdio.h>` for POSIX, but must not interfere with a user name in a strictly conforming ANSI program. Similarly, X/Open defines `P_tmpdir` and `popen`, which are not defined in POSIX or ANSI, and HP-UX defines `ctermid`, which is not defined in X/Open, POSIX, or ANSI.

We solved this problem by checking various macros, which are defined depending on which standard is desired. These macros are:

- `_HPUX_SOURCE`: Default for backwards-compatibility mode; includes everything.
- `_XOPEN_SOURCE`: User-defined; includes ANSI, POSIX, and X/Open symbols.
- `_POSIX_SOURCE`: User-defined; includes ANSI and POSIX symbols.
- `__STDC__`: Default for ANSI mode; only ANSI symbols are defined.

Libraries

One example of library name space pollution involves the `fopen` function, which has traditionally been implemented by calling `open`. If the user runs the program shown in in the next example on a system with a polluted name space, it will not behave as required by the ANSI standard.

```
#include <stdio.h>

struct {
        FILE *fd;
        /* ... */
} file_info1;

main()
{
        open("file1");
        process();
        close();
}

open(f1)
char *f1;
{
        file_info1.fd = fopen(f1, "r");

        /* ... */
}
```

However, we still must allow users to access the system `open` if they want to, for backwards compatibility and conformance with other standards such as POSIX. The program shown next must continue to work.

```
#include <fcntl.h>

int file_info1;

main()
{
           file_info1 = open ("file1", O_RDONLY);
           process();
           close (file_info1);
}
```

The solution we decided to implement was to provide "secondary definitions" for all conflicting names in the standard library. A secondary definition[1] specifies an additional name that another symbol can be known by, but which will not cause a duplicate definition error if the user has already defined it.

The library routines refer to the primary definition (containing a leading underscore). When `_open` is linked in, such as by a reference from `fopen`, the secondary name `open` will not be automatically added to the linker symbol definition table. It will be added only if unresolved references to it currently exist. Since the library routines all refer to the primary name `_open`, there will only be unresolved references to `open` if the user referenced it without providing a definition.

Data symbols create another set of problems. For example, take `environ`. This name is a data symbol which is defined by POSIX, but not by ANSI C. The symbol `environ` is used by the `getenv` function, which *is* defined in ANSI C, however. POSIX users must be able to set and reference this variable, while ANSI C users must be able to use this name for their own purposes.

Problems really arise when a user declares this name at file scope without an `extern` storage class specifier or initializer: should the name be considered a common symbol that would link to the library definition, or should it be a definition of a user name? In ANSI mode, it must be a user definition. However, lots of existing code uses this coding style. In order to allow this code to work as it did before, the implementation must "read the user's mind" to determine which interpretation is desired. In the next example, does the user want to affect the `getenv` function, or not?

```
char **environ;
main()
{
        char *language;
        environ = malloc(2 * sizeof(char *));
        environ[0] = "mystuff";
        language = getenv("LANG");
        /* ... */
}
```

Secondary definitions cannot be attached to common data, so all library data we wished to make visible to users was changed to be explicitly initialized. The problem of "mind reading" for user-declared common data was solved by fiat: uninitialized file-level data declarations with no storage class specifier are presumed to be user names, in ANSI mode. A special bit is set in the symbol information so the linker will not resolve that reference to a secondary definition. A user who wishes to reference library data from ANSI mode must declare it `extern`.

[1] The concept of secondary symbols is taken from some non-UNIX linkers, such as for the SDS/Xerox Sigma computers of the late '60s and early '70s.

One final problem has to do with `matherr`. The previous examples dealt with symbols that the user references that may need to be defined in the library. `matherr` is a symbol that the *library* references that may need to be defined by the *user*. In the next example, ANSI requires that `sin` should *not* call the user's `matherr`. Backwards compatibility requires that it *should*.

```
#include <math.h>
int result;
double dres;

void matherr()
{
            result = 99;
}
main() {
            result = 5;
            dres = sin(HUGE_VAL);
            /* ... */
            if (x + y > 32) matherr();
}
```

The `matherr` problem has no good clean solution. The POSIX 1003.2 Draft 8 contains a mechanism for deciding which of two math libraries to search. `-lM` causes the "ANSI" math library to be searched, while `-lm` causes the "backwards-compatibility" math library to be searched. This does not fit in well with our goal of having only one version of the libraries, but we had little choice in the matter.

Most of the code in the two versions of the math library is identical, only the `matherr` symbol is named differently. The name difference can be handled by a simple macro definition when compiling to create the ANSI version.

Summary

The name space pollution problem has several different aspects, requiring multiple solutions. While the effort to solve these problems was not insignificant, we feel we have implemented a solution that meets the requirements of the various standards, provides for backwards compatibility, and is maintainable.

Sue Meloy is a software design engineer in the Hewlett-Packard California Languages Lab, and has been an active member of X3J11 for several years. She can be reached electronically at hplabs!hpda!sue or sue%hpda@hplabs.hp.com, or by telephone at (408) 447-5768.

∞

4. ANSI/ISO Meeting Report

Jim Brodie

Abstract

This article covers recent events in the standardization process for the C language. It discusses the major national and international issues which were discussed at the joint ANSI/ISO meeting held recently. It also covers the current status of the standardization process and the directions being taken by the ANSI standards committee.

This article will bring you up to date on the latest happenings on the C language standard front.

X3 Secretariat Vote

Prior to the last X3J11[2] meeting an X3 ballot was taken on the current draft of the C standard. The results were very encouraging. The vote was 34 in favor of the standard, 0 against, and 2 abstentions. The X3 ballot is usually the last technical hurdle before the acceptance of an American National Standard. However, because of an administrative problem, another X3 ballot may be required. (More on this in a moment.)

ANSI Meeting Report

X3J11 met jointly with WG14, the International Standards Organization (ISO) committee which is responsible for the C language, on April 10–11 in Seattle Washington. The major topics of the meeting ranged from addressing a lost public review comment to discussing the objections which the British and Danish standards organizations had to accepting the current X3J11 draft of the C standard as an ISO standard. In addition, the committee decided not to take on the standardization of the C++ language. (This will be left to another committee, if it is to happen.)

One of the first orders of business for the meeting was to deal with a "lost" second public review commentary letter. This letter was either lost in the mail or within the X3 Secretariat and unfortunately never got as far as X3J11. A copy, however, was received by ANSI (American National Standards Institute).

[2]X3J11 is the technical committee charged with the task of developing the American National Standard for the C language.

The ANSI rules for standards formation require that every letter received during the public review periods must be given a response. ANSI will not accept a document for consideration as a standard until this requirement has been satisfied.

During the meeting, an 8 member subcommittee spent about 4 hours reviewing and writing responses to the lost letter. The letter writer, Russel Hansberry, attended the subcommittee meeting. After the review by the subcommittee several issues were brought before the committee, at Hansberry's request. These issues included a request that the precedence of the bit-wise operators be changed to higher than the relational operators (or at least a warning be placed in the document that this may happen in the future), a somewhat nebulous request that interrupt handling from C be included in the standard, and a request that the ordering of declarations with the register storage class specifier be used as a guide for the actual allocating of physical registers. After some discussion, none of these changes was accepted by the committee.

At this point, each issue for which Mr. Hansberry does not accept the X3J11 response will be distributed to the X3 membership (along with the committee's response). A 20-day ballot will be held to to see if any X3 members would like to change their votes based on these issues alone. If no votes are changed (which is the expected outcome) the document will be sent up to ANSI for the final processing to become an American National Standard. **We are hopeful that we will have an approved American National Standard by July of this year.**

ISO Meeting Report

On the international front, X3J11 and WG14 spent considerable time discussing the concerns that the British Standards Institute had concerning the current draft of the standard.

It turned out that the primary difference is one of philosophy on how a standard should be written. The issue has to do with how "undefined behavior" is indicated in the document. An example of an undefined behavior is the program behavior when an arithmetic computation overflows. Although X3J11 enumerates some common undefined behavior situations in the draft, in general, it falls back on a blanket statement: if a behavior is not explicitly defined within the standard, it is undefined (and cannot be relied upon in a portable program). The British position is that every case of known undefined behavior should be explicitly stated in the document.

A potential compromise was worked out. The British will develop a commentary document (which is not a formal part of the standard, but which is included for further clarification of the standard). This document will list all of the cases of undefined behavior which the British feel should be explicitly identified. This document, after review by both WG14 and X3J11, will be presented for inclusion with the standard.

The committees were less successful in dealing with a proposal from the Danish standards committee. The Danes would like to see some changes to provide more readable alternatives to trigraphs. While there is considerable sympathy for those programmers who will need to use trigraphs (it is widely accepted that they are not very readable), all of the proposals put forth so far, including the one presented during the meeting, have been unacceptable to X3J11.

The Danes did not gain any support for their proposal from the other countries represented by WG14 attendees. At this point, no changes were made in the X3J11 draft standard document and the Danes have gone home hoping to muster additional support for their proposal from the other Scandinavian countries. If they are able to gain enough support, they could delay or block the acceptance of the American standard *as an ISO standard*. Stay tuned.

Future of X3J11

X3J11 is now in transition to becoming an interpretations body. Assuming that the American National Standard for C is approved without any further hitches, the work load of the committee should be reduced significantly. X3J11 will begin meeting only once every six months. During these meetings requests for interpretations and clarifications of the C standard will be addressed. A large majority of the current members of the committee indicated that they were planning to remain active during the interpretations phase. Users of the standard can use requests for interpretations as a forum for getting C standards questions answered.

At this point, there is no support for beginning another standards activity (e.g., no one will be working on an Extended C standard).

Unless superseded by a later standard, the current C standard will be in force until 1999. In about 5 years, the committee will begin the formal process of reaffirming the current C standard or developing the next C standard (if this is necessary). It must achieve one of these goals within 10 years.

The next scheduled X3J11 meeting is in Salt Lake City, Utah on September 21–22, with DECUS hosting. Correspondence to X3J11 should be addressed to Tom Plum, Vice Chair, at Plum Hall, 1 Spruce Avenue, Cardiff, NJ 08232.

Jim Brodie is the convener and Chairman of the ANSI C standards committee, X3J11. He is also President of Brodie and Associates, a consulting company based in Phoenix, Arizona. He has coauthored books with P.J. Plauger and Tom Plum and is the Standards Editor for The Journal of C Language Translation. Jim can be reached at (602) 961-0032 or uunet!aussie!jimb.

∞

5. Adding Complex Arithmetic to C

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

Abstract

The addition of a complex type to C requires examination of all aspects of the language. A new keyword is proposed to permit declarations of this new arithmetic type. Type conversion of complex values to and from other arithmetic types is examined. Several methods for creating complex constants are proposed, including an `i` suffix and an operator. The new operator can be used to create a complex value out of two floating-point values, one representing the real part and the other representing the imaginary part, thus permitting complex constants to exist. Finally, a number of complex library functions are defined.

Cray Research has investigated the impact of adding a complex type to C. The following is a discussion based upon that research.

One of the reasons C is not used for numerical applications is the absence of a complex type. The motivation for adding complex arithmetic to C is to make the language more appealing to numerical and scientific programmers. For the most part, providing a complex extension is fairly straightforward. Although the following discussion describes the entire extension, it is focused on controversial decisions that need to be made.

Adding a New Keyword

A new keyword is needed to indicate a complex type. The obvious choice is `complex`, but that introduces a compatibility problem. C programs already exist that contain a `typedef` defined something like the following:

```
typedef struct { double real, imag; } complex;
```

Adding a new keyword that is spelled `complex` breaks existing standard conforming code.

The ANSI Standard gives implementers a set of names that they can use to extend C. These names must begin with two underscores or an underscore

and an uppercase letter, leading to the following two obvious possibilities: `__complex` and `_Complex`. However, neither of these names is especially appealing because neither conforms to the style of other keywords.

Two possible solutions to this problem are: a compile time option and a `typedef` name. The compile time option approach permits the `complex` spelling but inhibits access to the new type unless the option is specified. The compiler classifies the token `complex` as a keyword if the option is specified, otherwise it remains an ordinary identifier. The `typedef` name approach assumes the presence of a new header, `<complex.h>`, that contains a declaration similar to the following:

```
typedef _Complex complex;
```

along with function prototypes for several new library functions discussed later. This approach assumes that anyone wanting to use this new type is willing to include the `<complex.h>` header in their C source file. It also means that the implementation does not have to specify the actual spelling of the new keyword. Since compile time options that change the behavior of lexical analysis are error-prone and confusing, the `typedef` name approach is appealing. This allows existing portable programs to continue working because the new `<complex.h>` header will not be present.

A New Arithmetic Type

The new `complex` type is an arithmetic type but not a floating type. However, both the real and imaginary parts of a `complex` type are floating-point numbers.

The next implementation decision is to define the underlying type of the real and imaginary parts. Ideally, there would be three complex types: `float complex`, `double complex`, and `long double complex`. However, this requires extensive library and code generation support that is hard to justify when experimenting with a new type. The decision was made initially to implement only a `double` version as the underlying type of the real and imaginary parts. Type `double` is the default floating type, in that floating constants default to type `double`, and the default promotion of a `float` argument is to `double` (where applicable). `complex` implies `double complex`. However, the design should permit the logical evolution to multiple `complex` types. The following are several declarations that use the new type:

```
#include <complex.h>

complex cx;
complex acx[10];
complex cxsum(complex *cxp);
```

There must be some way of creating `complex` constants, to permit `complex` initializers. One approach is to expand upon the existing suffix notation used currently to create `float`, `long double`, `unsigned` and `unsigned long` constants. That is, a constant such as:

```
2.1i
```

has `complex` type. The imaginary part has the value 2.1 and the real part has the value zero (0). This permits an expression such as:

```
3.4 + 2.1i
```

to create a `complex` value with 3.4 for the real part and 2.1 for the imaginary part. An example of a `complex` initializer using this approach is:

```
#include <complex.h>

complex cx = 3.4 + 2.1i;
```

There is also a need to create a complex number out of two floating-point values. The `i` suffix approach does not provide any way to create a `complex` value out of two arbitrary expressions. Other possibilities are a new operator, a new keyword, or a library function. These possibilities are examined further.

The Complex Operator

The complex operator creates a complex number out of two floating-point values. Several different approaches are possible. For example:

```
cmplx(real, imag)       /* new keyword */
<real, imag>            /* grouping */
real %% imag            /* infix */
```

New keywords are, again, controversial because of the potential to break existing code. However, both the grouping and infix approach can be made to work fairly easily.

```
c1 = c2 + <2.1, 3.4>;
c1 = c2 + 2.1 %% 3.4;
```

A problem with the infix approach, however, is that it forces the use of parentheses in certain situations. For example, in:

```
c1 = c2 + <2.1 + x,  3.4>;
c1 = c2 + (2.1 + x) %% 3.4;
```

assuming that `%%` has precedence just below the unary operators. However, the grouping approach forces the use of parentheses in macro calls. For example:

```
MAC(<x, y>)     /* two args */
MAC((<x, y>))   /* one arg */
```

Another implication of the grouping approach is that an operand that uses the comma operator can only appear inside a parenthesized expression. (It may also cause parsing problems.)

```
c1 = c2 + <(x++, 2.1), 3.4>;
```

The grouping approach, however, does not introduce any operator precedence issues because the `<>` pair groups identically to the `()` and `[]` pairs.

This operator can also be used to create a `complex` constant expression which can be used in an initializer.

```
complex c1 = <2.1, 3.4>;
```

Since one of the goals is to permit future extensions to multiple complex types, the behavior when the types of the real or imaginary parts is not `double` needs to be discussed. The following rules are defined:

- First, all integral operands are converted to `double`.
- Then neither operand can have type `long double` or `complex`.
- Then, at least one operand must have type `double`.
- Then, if only one operand has type `double` the other operand is converted to `double`.
- Then, the real part is assigned the value of the first operand.
- Then, the imaginary part is assigned the value of the second operand.

These rules permit the semantics of the complex operator to be enhanced in the future to include `float complex`, and `long double complex`.

As with most operators, the order of evaluation of the operands is unspecified.

Since there already is some pressure to add an *aggregate constructor* to C, the following method should also be considered as a way to create a complex number out of two floating-point numbers:

```
double x = 4.5, y = 6.7;
complex cx;

cx = (complex) {2.1, 3.4};      /* real 2.1, imag 3.4 */
cx = (complex) {x, y};          /* real 4.5, imag 6.7 */
```

Complex Type Conversions

Conversions to and from `complex` values are easily defined in terms of the existing arithmetic conversion rules. Essentially, when a `complex` value is converted to a floating or integral type the imaginary part is discarded and the real part is converted as if converting a value of type `double`. Similarly, when an integral or floating value is converted to `complex`, the real part is assigned the value obtained after converting it to type `double` and the imaginary part is assigned the value zero (0).

```
#include <complex.h>

complex cx = <2.1, 3.4>;
double d;
int i;

d = cx;         /* d == 2.1 */
i = cx;         /* i == 2 */
cx = 5.6;       /* cx == <5.6, 0.0> */
cx = i;         /* cx == <2.0, 0.0> */
```

These conversion rules permit `complex` values to be assignment compatible with other arithmetic types.

The *usual arithmetic conversions* must also be enhanced to cover `complex` operands. The following conversion rule permits `complex` operands to be mixed with other arithmetic operands:

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

All of the following expressions have type `complex`:

```
#include <complex.h>

complex cx;
double d;
int i;

d + cx
cx - i
<2.1, 3.4>  *  d
```

Complex Expressions

The following arithmetic operators can have complex operands:

unary: `+` `-` `++` `--`
binary: `*` `/` `+` `-`

Since the `!` operator is a logical operator whose result has type `int`, it seems innocuous to also permit the other logical operators `&&` and `||` operators to have `complex` operands. In all cases the `complex` operand is compared to zero (0).

```
#include <complex.h>

complex cx1 = <2.1, 3.4>;
complex cx2 = 0;

/* ... */

!cx1                    /* FALSE */
!cx2                    /* TRUE */
cx1 && cx2              /* FALSE */
cx1 || cx2              /* TRUE */

if (cx1 && !cx2)        /* TRUE */

while (cx1)             /* loops until cx1 == 0 */
```

The first operand of the conditional `?:` operator can also have a `complex` type since it behaves like the logical operators. (Of course, both the second and third operands of the conditional operator are also permitted to have type `complex`.)

As with all arithmetic types, casts to type `complex` are also permitted.

The relational operators `<`, `<=`, `>`, and `>=` cannot have `complex` operands. However, the equality operators `==` and `!=` are well defined for `complex` operands with the result still having the value one (1) or zero (0) and type `int`.

Discussions at the first NCEG meeting indicate that there is no support for conforming to the FORTRAN requirement that

```
sizeof(complex) == 2 * sizeof(double)
```

Furthermore, the committee felt that the order of the real and imaginary parts should not be specified.

The Complex Library Header <complex.h>

There are some differences between the behavior of the functions defined here and those defined under the <math.h> header. Nothing is defined that corresponds to the HUGE_VAL macro, and in general, the treatment of error conditions is left unspecified. This means that no defined relationship exists between the following functions and the expression errno. There is no requirement that these functions execute as if they were a single operation without generating visible exceptions!

The header <complex.h> declares several mathematical functions and one type. The type is:

```
complex
```

which represents an object with a real and imaginary part. Both parts have type double.

These functions take double or complex arguments and return double or complex values.

The csin function

Synopsis

```
#include <complex.h>

complex csin(complex x);
```

Description

The csin function computes the sine of the complex number x (measured in radians).

Returns

The csin function returns the sine of x.

The `ccos` function

Synopsis

```
#include <complex.h>

complex ccos(complex x);
```

Description

The `ccos` function computes the cosine of the complex number `x` (measured in radians).

Returns

The `ccos` function returns the cosine of `x`.

The `cexp` function

Synopsis

```
#include <complex.h>

complex cexp(complex x);
```

Description

The `cexp` function computes the exponential function of the complex number `x`.

Returns

The `cexp` function returns the exponential of `x`.

The `clog` function

Synopsis

```
#include <complex.h>

complex clog(complex x);
```

Description

The `clog` function computes the natural logarithm of the complex number `x`.

Returns

The `clog` function returns the natural logarithm of `x`.

The `cpow` function

Synopsis

```
#include <complex.h>

complex cpow(complex x, complex y);
```

Description

The `cpow` function computes the complex number `x` raised to the complex power `y`.

Returns

The `cpow` function returns the value of `x` raised to the power `y`.

The `csqrt` function

Synopsis

```
#include <complex.h>

complex csqrt(complex x);
```

Description

The `csqrt` function computes the square root of the complex number `x`. The sign of the imaginary part of the root is the same as the sign of the imaginary part of `x`.

Returns

The `csqrt` function returns the square root of `x`.

The `cabs` function

Synopsis

```
#include <complex.h>

double cabs(complex x);
```

Description

The `cabs` function computes the absolute value of a complex number `x`.

Returns

The `cabs` function returns the absolute value of `x`.

The `cimag` function

Synopsis

```
#include <complex.h>

double cimag(complex x);
```

Description

The `cimag` function computes the imaginary part of the complex number `x`.

Returns

The `cimag` function returns the imaginary part of `x`.

The `cmplx` function

Synopsis

```
#include <complex.h>

complex cmplx(double x, double y);
```

Description

The `cmplx` function computes the complex number that has a real part represented by `x` and an imaginary part represented by `y`.

Returns

The `cmplx` function returns the complex number with a real part of `x` and an imaginary part of `y`.

Note that the `cmplx` function cannot be implemented as a macro unless a complex operator exists.

The `conj` function

Synopsis

```
#include <complex.h>

complex conj(complex x);
```

Description

The `conj` function computes the conjugate of the complex number `x` by negating the imaginary part of `x`.

Returns

The `conj` function returns the conjugate of `x`.

The `creal` function

Synopsis

```
#include <complex.h>

double creal(complex x);
```

Description

The `creal` function computes the real part of the complex number `x`.

Returns

The `creal` function returns the real part of `x`.

I/O of Complex Values

One area that is conspicuously absent in this discussion is I/O. Since the functions `creal` and `cimag` allow access to the real and imaginary parts, they can be printed in any format. Similarly, two floating-point values can be read in and converted to a `complex` value by using the complex operator or the `cmplx` function. Sufficient machinery already exists to handle the general problem of converting `complex` values to decimal representation and back again.

Conclusion

A `complex` extension seems to fit into C without affecting any existing portable code. However, there is no substitute for an actual implementation discovering hidden problems. Several issues raised by this paper were presented at the first Numerical C Extension Group (NCEG) meeting in May. My hope was that that committee should assist in resolving these issues, allowing an actual implementation to benefit from that insight. The outcome of this proposal will be presented in a future issue.

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

∞

6. Pragmania

Rex Jaeschke

Introduction

Welcome to the first installment of this column. I had little trouble coming up with an appropriate column title since I had already installed several compilers that understood numerous pragma directives. It seemed that, before long, everyone would be jumping on a (different) pragma wagon.

At a first, and maybe even a second or third, glance, it seems most unusual to have something in a language standard whose syntax and semantics are implementation-defined. However, once an implementer lays back and thinks of the possibilities, some pretty abstract ideas can arise. Many of these are being implemented. I refer to pragmas as the "Pandora's Box of ANSI C." (If you remember your Greek mythology, when Pandora opened her box all evils were released. Only hope remained. You can decide for yourself whether pragmas represent the evil or the hope. I think they are a mixed blessing.)

Now I don't mind the concept of the pragma directive. In fact, I strongly endorse it. However, I do have one problem and that has to do with the following sentence in the Draft Standard.

> "Any pragma that is not **recognized** by the implementation is ignored."

Just what does "recognized" mean? Specifically, can an implementer diagnose a misspelled pragma really intended for it? For example, if the following pragma is defined for a given implementation,

```
#pragma ABCD stuff
```

can the following directives be diagnosed by that implementation or are they "not recognized?"

```
#pragma ABDC stuff
#pragma ABCD stuxx
```

Actually, there are two problems one can encounter with pragmas. The first involves a misspelled use of a pragma directive actually intended for the translator that ignores it. The second involves porting code containing pragmas

where the target translator does not ignore the pragmas. Rather, it gives them semantics other than originally intended. Perhaps the first type of problem will be more common; but both, no doubt, will occur.

Just what is a reasonable solution to this issue? Without any standard pragmas it is difficult if not impossible to generalize an approach. Certainly, we could specify just how many preprocessing tokens should match before you accept a directive as actually being "intended for me but misspelled," but that would involve some sort of policing of leading token name space and it would require every pragma to have at least that many tokens.

It seems to me that most implementers will have to provide some mechanism to report "quality of implementation" messages. The pragma issue could be handled that way. Specifically, I would like to see a translator write a warning (or, possibly, informational) message to `stderr` for *every* pragma that it accepted *or* rejected. That is the only way the programmer can reliably detect pragma misspellings or misinterpreted "foreign" pragmas. As someone who much more closely fits the profile of a user than an implementer, I urge developers to seriously consider this proposal. Without it, debugging a pragma directive containing transposed characters could be expensive, particularly if the pragma was supposed to subtly alter the way code was being generated.

Public Commentary

During a recent electronic exchange with Richard Stallman, Chief GNUisance at the Free Software Foundation, he asked if I would be interested in his opinions on `#pragma`. I was, so here they are.

> "I consider it nearly useless, because it can't appear in macros. Almost any extension that is useful is useful in macros, so I design it in terms of something other than `#pragma`. The only pragma I have implemented is `#pragma once`, which is used in a header file to say that duplicate `#include`s should be suppressed.
>
> Also, the semi-standardness of pragma is of no help. Since you can't tell what `#pragma foo` will mean in some other compiler, it is in practice no more standard than anything else you might invent."

Reserving Pragma Name Spaces

I recall that way back when the `#pragma` directive was added to the proposed C Standard, that many ANSI C members (at least privately) said it would be a good idea for each vendor to have a unique prefix for its own pragmas. Of course, vendors then went and implemented pragmas *without* following their own advice. Whether this is simply a lack of thought, discipline, or perhaps even an ego trip by some (since their pragmas were obviously so innovative that everyone else should adopt them as spelled) remains to be disclosed.

In any event, of the many implementations that I have come across that are providing pragmas, I have seen only one that is using some sort of unique company or product naming scheme. That is the Eco-C88 compiler for DOS, from Ecosoft. All their pragmas begin with the token `eco`. Now, Hewlett-Packard of Cupertino, California, is boldly taking the plunge. By the powers vested in them by the State of California (and this publication), HP hereby reserves the name space of all pragmas beginning as follows:

```
#pragma HP *
#pragma _HP *
```

for use by their compilers. Any vendor found violating this name space will be sentenced to writing all their source using trigraphs.

If you have adopted some rational pragma naming conventions, please let me know so I may similarly "reserve" that name space for you in this publication.

WATCOM's V7 DOS Compiler

WATCOM has been implementing C language translation tools for a number of years, primarily on IBM mainframes. However, a couple of years ago they entered the already crowded MS-DOS marketplace with version 6 of their compiler. The most interesting aspect of this product, from this column's point of view, is the extent to which WATCOM has embraced the notion of pragmas.

The following information is reprinted with permission from WATCOM Systems, Inc. This material is extracted from their manual *Optimizing Compiler and Tools User's Guide* 2nd Edition, and is copyright ©1989 by WATCOM Publications Limited. Except for minor editorial changes, the material presented here is taken verbatim from the above-mentioned manual. The original text has been considerably shortened by omitting many DOS-specific examples of code generation and other information that is neither central to the purpose of this column nor necessary to understand the extract. Great care has been taken so that the extracted material is not printed out of context.

Introduction

There are essentially four classes of pragmas:

1. pragmas that specify options
2. pragmas that specify default libraries
3. pragmas that describe the way structures are stored in memory
4. pragmas that provide auxiliary information used for code generation

Using Pragmas to Specify Options

Currently, there are two options that can be specified with pragmas. They are, `unreferenced` and `check_stack`.

`unreferenced` controls the way WATCOM C handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause WATCOM C to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause WATCOM C to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if `unreferenced` was specified.

`check_stack` controls the way stack overflows are handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When `check_stack` is on, WATCOM C will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

Using Pragmas to Specify Default Libraries

Default libraries are specified in object module comment records. These special records are recognized by the WATCOM Linker and the library names extracted. When unresolved references remain after processing all object files specified in a linker `FILE` directive, these default libraries are searched after all libraries specified in a linker `LIBRARY` directive have been searched.

By default, that is if no library pragma is specified, the WATCOM C compiler generates, in the object file containing `main`, default libraries corresponding to the memory model used to compile the file. For example, if you have compiled the source file containing `main` for the medium memory model, references to the libraries `clibm` and `mathm` will be placed in the object file.

If you wish to add your own default libraries to this list, you do so with a `library` pragma. Consider the following example.

```
#pragma library (mylib);
```

The name `mylib` will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as `\`, `:`, or `,` (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("c:\watcomc\lib\graphics.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "c:\watcomc\lib\graphics.lib");
```

Pack Pragmas

The `pack` pragma can be used to control the way in which structures are stored in memory. By default, WATCOM C aligns all structures and their fields on a byte boundary. The following form of the `pack` pragma can be used to change the alignment of structures and their fields in memory.

```
#pragma pack ( n );
```

where n is 1, 2, or 4 and specifies the method of alignment.

If n is 1, all subsequent structures and their fields are aligned on a byte boundary. This gives the most compact use of storage.

If n is 2, all subsequent structures and their fields are aligned on a word boundary. The size of the structure is rounded up to a multiple of 2.

If n is 4, all subsequent structures and their fields are aligned on a double word boundary. The size of the structure is rounded up to a multiple of 4.

If no value is specified in the `pack` pragma, a default value of 1 is used. (Note that the default value can be changed with the `zp` WATCOM C compiler command line option.)

Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

Specifying a Symbol's Attributes

Auxiliary pragmas are used to describe a symbol's attributes. All symbols have default attributes as defined by WATCOM C. Initially, these default attributes are assigned to all symbols. When an auxiliary pragma refers to a particular symbol, you are changing its default attributes. Alternatively, if `default` is specified for the symbol name, the attributes specified by the auxiliary pragma becomes the default for all symbols. For example,

```
#pragma aux MyRtn ...;
```

sets the attributes for the symbol `MyRtn` while

```
#pragma aux default ...;
```

sets the default attributes for all symbols.

Alias Names

An alias name can also be specified with the symbol that the auxiliary pragma refers to. When specified, the symbol referred to by the auxiliary pragma assumes the attributes of the alias name in addition to the attributes specified in the auxiliary pragma. Consider the following example.

```
#pragma aux MS_C "_*"                          \
        parm caller []                         \
        value struct float struct routine [ax]\
        modify [ax bx cx dx es];
#pragma aux (MS_C) Rtn1;
#pragma aux (MS_C) Rtn2;
#pragma aux (MS_C) Rtn3;
```

The routines `Rtn1`, `Rtn2` and `Rtn3` assume the same attributes as the alias name `MS_C`. The alias name `MS_C` defines the calling convention used by Microsoft C. Whenever calls are made to `Rtn1`, `Rtn2`, and `Rtn3` the Microsoft C calling convention will be used.

Note that if the attributes of `MS_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `Rtn1`, `Rtn2` and `Rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

Special Symbols

Three symbols are treated in a special way by the WATCOM C compiler. If you use one of the keywords `cdecl`, `fortran`, or `pascal` in a function declaration, the calling conventions used for that function will be those defined by the corresponding auxiliary pragma.

For example, if we define a pragma for the symbol `cdecl` then any function that is declared with that keyword will be called using the method described by the pragma.

```
#pragma aux cdecl ...;
```

The actual definitions for these pragmas are included in the header file `stddef.h` in the WATCOM C package. The `cdecl` pragma that is provided by WATCOM C defines the calling conventions used by WATCOM Express C [WATCOM's interactive development environment].

Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

`#pragma aux` *symbol_name object_name* `;`

where

symbol_name is any valid C identifier

object_name is any character string enclosed in double quotes.

When specifying *object_name*, the `*` character has a special meaning; the asterisk is replaced by *symbol_name*.

In the following example, the name `MyRtn` will be replaced by `MyRtn_` in the object file.

```
#pragma aux MyRtn "*_";
```

This is the default for all function names.

In the following example, the name `MyVar` will be replaced by `_MyVar` in the object file.

```
#pragma aux MyVar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore (_).

Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#pragma aux function_name far;
   or
#pragma aux function_name near;
   or
#pragma aux function_name = {constant} ;
```

where *constant* is a valid C integer constant.

In the following example, WATCOM C will generate a far call for all calls to the function `MyRtn`.

```
#pragma aux MyRtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, WATCOM C will generate a near call for all calls to the function `MyRtn`.

```
#pragma aux MyRtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following example, WATCOM C will generate the sequence of bytes following the `=` character in the auxiliary pragma whenever a call to `Mode4` is encountered. `Mode4` is called an in-line function.

```
void Mode4( void );
#pragma aux Mode4 =                                    \
        0xB4 0x00 /* mov AH,0  "set mode" function */ \
        0xB0 0x04 /* mov AL,4   mode 4             */ \
        0xCD 0x10 /* int 10H    BIOS video call    */ \
        modify [ AH AL ];
```

The above example demonstrates how to generate BIOS or DOS function calls in-line without writing an assembly language function and calling it from your C program. The C prototype for the function `Mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by WATCOM C.

Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that WATCOM C is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other C compilers or functions written in other programming languages such as FORTRAN.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#pragma aux function_name parm [caller or routine]
            [reverse] {reg_set} ;
```

where *reg_set* is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#pragma aux function_name parm {reg_set} ;
```

where *reg_set* specifies the registers that are to be used for argument passing.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, WATCOM C is free to choose registers in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, WATCOM C will only select certain combinations of registers for arguments of a particular type.

[The details of how register passing is implemented are specific to the Intel chips and have been omitted.]

The following example shows the specification of two register sets.

```
#pragma aux MyRtn parm [AX BX CX DX] [SI DI];
```

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; remaining arguments go on the stack.

Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you want a routine, say `MyCopy`, that copies data from the default data segment (pointed to by segment register DS) to far memory (memory outside of the default data segment). The first argument is the source and is of type "near pointer." The second argument is the destination and is of type "far pointer." The third argument is the length to copy and is of type `int`. If we want the first argument to be passed in the register SI, the second argument to be passed in the register pair ES:DI and the third argument to be passed in register CX, the following auxiliary pragma can be used.

```
void MyCopy( char near *, char far *, int );
#pragma aux MyCopy parm [SI] [ES DI] [CX];
```

Note that you must be aware of the size of the arguments to ensure the proper mapping.

Passing Arguments to In-Line Functions

For functions whose code is generated by WATCOM C and whose argument list is described by an auxiliary pragma, WATCOM C has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, WATCOM C assumes that each register set corresponds to an argument. Consider the following in-line function called `ScrollActivePgUp`.

```
void ScrollActivePgUp(char,char,char,char,char,char);
#pragma aux ScrollActivePgUp =                        \
        0xB4 0x06 /* mov AH,6  select "scroll up" */ \
        0xCD 0x10 /* int 10H   BIOS video call    */ \
        parm [CH] [CL] [DH] [DL] [AL] [BH]            \
        modify [AH];
```

When passing arguments, WATCOM C will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `ScrollActivePgUp` was called with an argument whose type was `int`, it would first be converted to `char` before assigning it to register CH. Similarly, if an in-line function required its argument in the register pair DX:AX and the argument was of type `int`, the argument would be converted to `long int` before assigning it to the register pair DX:AX.

In general, WATCOM C assigns the following types to register sets.

- A register set consisting of a single 8-bit register (1 byte) is assigned a type of `unsigned char`
- A register set consisting of a single 16-bit register (2 bytes) is assigned a type of `unsigned int`
- A register set consisting of two 16-bit registers (4 bytes) is assigned a type of `unsigned long int`
- A register set consisting of four 16-bit registers (8 bytes) is assigned a type of `double`

Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were passed on the stack.

`#pragma aux` *function_name* `parm caller` or `routine ;`

`caller` specifies that the caller will pop the arguments from the stack; `routine` specifies that the called routine will pop the arguments from the stack. If `caller` or `routine` is omitted, `routine` is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order. That is, the rightmost argument is processed first and the leftmost argument is processed last.

`#pragma aux` *function_name* `parm reverse;`

Normally, arguments are processed from left to right.

Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other C compilers or functions written in other programming languages such as FORTRAN.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#pragma aux function_name value [reg_set]
            [struct_return] ;

struct_return ::= struct [float] [struct]
            [routine or caller] [reg_set]
```

Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
#pragma aux function_name value reg_set ;
```

Depending on the type of the return value, only certain registers are allowed in *reg_set*.

[The specific register allocation scheme has been omitted.]

Returning Structures

Typically, structures are not returned in registers. Instead, a pointer to the structure is returned in a register. By default, the caller allocates space on the stack for the structure return value and sets register SI to point to it. The following form of the auxiliary pragma can be used to specify the register which points to a structure return value.

```
#pragma aux function_name value struct
        (caller or routine) reg_set ;
```

`caller` specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set, by the caller, before the function is called.

`routine` specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value.

[The specific register details have been omitted.]

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2, or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register SI to it.

```
#pragma aux function_name value struct struct;
```

The following form of the auxiliary pragma can be used to specify that function return values whose type is "single" or "double" are not to be returned

in registers. Instead, the caller will allocate space on the stack for the floating-point return value and point register SI to it.

```
#pragma aux function_name value struct float;
```

A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#pragma aux function_name aborts;
```

For such functions, WATCOM C generates a JMP instruction instead of a CALL instruction.

Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller. This pragma causes the code generator to omit the updating of in-memory copies of variables across function calls.

```
#pragma aux function_name modify nomemory;
```

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux function_name parm nomemory modify nomemory;
```

You must specify both `parm nomemory` and `modify nomemory`.

Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a routine will use without saving.

```
#pragma aux function_name modify [exact] reg_set ;
```

Specifying a register set informs WATCOM C that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function. Hence, if necessary, code will be generated to save and restore the contents of that register.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called routine. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called routine does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called routine.

`#pragma aux` *function_name* `modify exact` *reg_set* `;`

The above form of the auxiliary pragma tells WATCOM C not to assume that the registers used to pass arguments will be modified by the called routine. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87 [floating-point processor]. The discussion in this chapter assumes that the "7" option is used to compile functions. The following areas are affected by the use of the "7" option:

- passing floating-point arguments to functions
- returning floating-point values from functions
- which 80x87 floating-point registers are allowed to be modified by the called routine

Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed in 80x87 floating point registers. Only four floating-point registers are used to pass floating-point arguments. When these floating-point registers are exhausted (more than four floating-point arguments are passed), the remaining floating-point arguments are passed on the stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "7" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type `int`.

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#pragma aux function_name parm {reg_set} ;
```

reg_set can contain 80x86 registers and/or the string `"8087"`.

If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string `"8087"` appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers. If the string `"8087"` was not specified in the register set, all floating-point arguments would be passed on the stack.

Using the 8087 to Return Function Values

The following form of the auxiliary pragma can be used to describe the way a function return value is returned.

```
#pragma aux function_name value reg_set ;
```

If the register set contains the string `"8087"`, floating-point values will be returned in 80x87 floating-point register ST(0). If `"8087"` is not specified in the register set, floating-point values will be returned on the stack. The caller will allocate space on the stack for the return value and point register SI to it. No 80x86 registers will be used to return floating-point values when a function is compiled with the "7" option.

Preserving 80x87 floating-point Registers Across Calls

Four of the eight 80x87 floating-point registers are used for a function's local variables. These four floating-point registers are known as the 80x87 cache. When a function is called, the registers in the 80x87 cache must be preserved. The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

```
#pragma aux function_name modify reg_set ;
```

reg_set is a register set containing the string `"8087"`.

This instructs WATCOM C to save any local variables that are located in the 80x87 cache before calling the specified routine.

Submit Details of Your Pragmas

In future installments I'll be looking at interesting and useful pragmas from various MS-DOS compilers, DEC's VAX C V3, and the new Cray Standard C compiler, among others.

If you would like your favorite pragmas discussed, either send me details via e-mail or mail a copy of the user manual. Perhaps you even have a proposal for some new pragmas you would like to see implemented. For example, anyone out there interested in something like `#pragma lint ...`?

∞

7. Standards Forum: Type Qualifiers

Jim Brodie
Brodie and Associates

Abstract

This article looks at the type qualifiers, `const` and `volatile`. It provides a brief background on why they were included into the language. It then discusses the issues which an implementer must deal with when introducing them into a conforming translator.

Introduction

As you begin or continue the process of upgrading your translator to support the C language as defined by the upcoming ANSI C standard, you will be confronted with some important changes that may have a wide-ranging impact. One of these changes is the introduction of the type qualifiers `const` and `volatile`. In this article we will explore the `const` and `volatile` type qualifiers and the impact that their addition has on a translator implementation.

Background

The `const` type qualifier was originally borrowed from C++, although similar concepts and facilities are available in other languages, such as Pascal. The `const` qualifier allows the programmer to state that the value of a data object should be unchanging. With this information, the translator is expected to help detect and diagnose (presumably accidental) attempts to change the value. In addition, the translator can use this information to help it make various optimization decisions (e.g., to avoid unnecessary loads).

The `volatile` type qualifier is a creation of the committee. `volatile` allows programmers to inform the translator that most optimizations are not appropriate when generating code that accesses a particular data object.

The `volatile` qualifier is used when a programmer needs to indicate that a memory location has special properties. The standard says:

> "An object that has `volatile`-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine."

The abstract machine is a hypothetical environment where every operation is performed exactly as described in the standard. (Issues of optimization are irrelevant.)

`volatile`-qualified types can be used with memory-mapped I/O locations such as exist in a microprocessor-based system. For example, in most Motorola MC680x and MC680x0 processor based systems, communications with devices is performed by mapping certain memory addresses to physical device control registers rather than to real memory. A program communicates with devices by performing what appears to the processor to be normal loads and stores of memory. These loads and stores, however, cause special actions such as resetting device states or instituting data transfers.

The protocols for handling these devices often include what, to a program working with normal memory, may seem to be unnecessary operations.

A C program that is interacting with a memory mapped device may include fragments such as:

```
volmem = 0;
volmem = 0;
```

or

```
for (i = 1; i < num; i++)
        {
        volmem = 1;
        a[i] = volmem2;
        }
```

In most situations a programmer would want the translator to detect opportunities for speeding up the code by eliminating "extraneous" stores and loads of memory. There are several opportunities to perform these optimization in the above code fragments. Unfortunately, if these optimizations are performed when accessing a memory mapped I/O device, the program will no longer work.

Memory mapped I/O is not the only case where data object values may change in ways unknown to the implementation. Similar problems can also arise when memory locations are used to communicate between two executing processes.

These situations, and others like them, make dangerous certain optimizations, such as using a previously fetched value which still resides in a register. Perhaps another process has changed the value since the last access.

Translators that address markets where these special situations arise have been caught between a rock and a hard place. Their customers always want very efficient, highly optimized code, except when they don't (because it will break the program). Prior to `volatile`, there was no standard way of distinguishing these two cases in a C program.

It is interesting to note that the real importance of `volatile` is that implementers can perform more aggressive optimizations when it is *not* present.

If the object's type is not `volatile`-qualified, an implementer can assume that they have complete control and know all of the ways a data object will be modified.

Although the primary reason for using the `volatile` type qualifier is to control optimizations, the full inclusion of the `const` and `volatile` type qualifiers will require that you make a variety of changes. You must, of course, accept the new keywords `const` and `volatile` in declarations and casts, and modify your optimizer. You must also change the type information that you save, how you perform certain type checking operations, and possibly how you perform code generation. Let's look at some of these required changes.

Type Information Changes

One of the first changes an implementation will need to make is to save, along with the current type information, whether the `const` and `volatile` type qualifiers have been specified. This can be done in as little as a pair of one-bit flags. Notice, however, that the type qualifiers are not a global property of a complex type. Rather, they are associated with base type and/or the *pointer to* decorations. There is a significant difference between:

```
int *const cpi = &a;        /* const ptr to int */
const int *pci;             /* ptr to a const int */
const int *const cpci = &x; /* const ptr to const int */
```

In the first case (`cpi`) the pointer is unchanging, in the second (`pci`) the `int` object pointed to is unchanging, and in the third (`cpci`) both are unchanging. The requirement to save `volatile` and `const` type qualifier information with each *pointer to* decoration for a type may break various type information packing schemes where the *array of*, *pointer to*, and *function returning* attributes of a type are packed as a series of small bit-fields in a word. (This approach has its roots in the Ritchie and PCC compilers out of Bell Labs.)

Only the pointer type requires the `volatile` and `const` attributes. With arrays, the qualifier is applied to the elements of the array, not the array type. `const` and `volatile` have undefined effect on a function type.

Another issue that you need to deal with is how the type qualifiers interact when type information is merged to create a new composite type. This occurs when a second or later declaration for an external data object is encountered in a translation unit. The type information from the multiple declarations is used to determine the composite type.

Type merging may also occur when dealing with the second and third operands of the conditional operator (`?:`). A translator must allow the second and third arguments to be pointers which point to differently qualified types (assuming that they are otherwise compatible). The information from the two types is used to determine the resulting composite pointer type of the

expression.

In both of these cases the type qualifiers are additive. If a type qualifier appears in either of the types being merged, then the resulting composite type includes the qualifier. For example, if the second argument for the conditional operator has the type *pointer to* `const int` and the third argument has the type *pointer to* `volatile int`, the type of the conditional operator result will be *pointer to* `const volatile int`.

Types which include the `const` or `volatile` type qualifiers are distinct from the types without these qualifiers. However, the standard requires that the qualified and unqualified versions of a type have the same representation and alignment requirements. This implies interchangeability of data objects with qualified and unqualified types in situations such as arguments to functions, return values from functions, and members of unions.

Type qualifiers deal only with the way data objects are accessed. Because of this, type qualifiers do not have any effect on the type of an rvalue result of an expression. The type information associated with rvalues never carries along the type qualifiers. For example, after the declarations:

```
const int j = 3;
volatile int k = 4;
```

the resulting type of the expression

```
j + k
```

is simply `int` (not `const volatile int`).

There are several other areas where the type qualifiers are essentially ignored. A translator must allow the comparison of pointers which point to differently qualified types (assuming that they are otherwise compatible). A translator must also allow the subtraction of pointers which point to differently qualified types, assuming that they both point to elements within the same array.

After looking at all of these places where an implementation needs to ignore the type qualifiers, let's look at a few places where the type qualifiers are taken into account.

Type Checking Changes

Several changes must be made to the way an implementation checks types when dealing with lvalues in expressions.

An implementation has always had to check that an lvalue on the left hand side of an assignment (or an operand to an increment or decrement operator) does not have an array type. Now an additional check must be made to ensure that the lvalue does not have a `const`-qualified type. If a structure assignment is being performed, none of the members of the structure addressed by the

lvalue may have a `const` qualified type. You must, of course, test for `const` members in structures within structures. An implementation needs to produce a diagnostic if these restrictions are violated.

The assignment compatibility rules are also modified with respect to the type qualifiers. A pointer with type *pointer to type T* may be assigned to a pointer with type *pointer to qualified version of type T*. However the reverse is not allowed. A pointer with type *pointer to qualified version of type T* may not be assigned to a pointer with type *pointer to type T*. For example, you can assign a pointer to `char` to a pointer to `const char` but you cannot assign a pointer to `const char` to a pointer to `char`. Some other examples are:

```
const char * pcc;
char * pc;
volatile char *pvc;

pcc = pc;       /* OK */
pc = pcc;       /* INVALID, produce a diagnostic */
pc = pvc;       /* INVALID, produce a diagnostic */
pcc = pvc;      /* INVALID, produce a diagnostic */
```

The implementation is not required to diagnose indirect attempts to modify a data object with a `const`-qualified type. For example:

```
const int ci = 5;
int *pi;

pi = (int *)&ci;
*pi = 7;        /* INVALID, no diagnostic required */
```

Code Generation Changes

As discussed previously, the most dramatic impact of type qualifiers is in the area of code generation and optimizations.

When a data object is defined with a `const`-qualified type, it may be placed into read-only memory (ROM). Particularly in embedded system, this can be a very useful option to have. The `const` qualifer also helps the optimizer make less pessimistic assumptions about function calls. For example, common subexpressions which contain `const` global objects are not killed by subroutine calls; and passing the address of objects with no qualifiers to functions whose formal parameters are declared *pointer to* `const` type means the optimizer can assume the function does not kill those objects via the formal parameters.

The biggest impact of the type qualifiers is the change that the `volatile` type qualifier has on code generation. The standard puts severe limitations on what optimizations can be performed in code that references data objects with `volatile`-qualified types. The standard says:

> "At sequence points, `volatile` objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred."

This rule prevents many common optimizations (e.g., code motion or common sub-expression elimination across sequence points). It may require changes both in an implementation's initial code generation algorithms and in its peephole or global optimizer (if they are included).

Sequence points, by the way, occur at the end of the first operand in the logical AND (`&&`), logical OR (`||`), conditional (`?:`), and comma (`,`) operators. The call to a function, after all of the arguments and the function designator have been evaluated, is also a sequence point. A sequence point also occurs at the end of a "full expression," which is an expression that is not a sub-expression. Full expressions include the expression in an expression statement, the selection expressions in the `if` and `switch` statements, the controlling expression in the `while` and `do` statements, the three expressions in the control portion of the `for` statement, and the optional expression in the `return` statement.

While there are restrictions on any movement of expression evaluation involving access of a data object with a `volatile`-qualified type, there still remains some limited opportunities for optimization within the interval between two sequence points. For instance, you can still use the rearrangement of commutative operators in expressions such as:

```
7 + vol + 22
```

to obtain (in the absence of concerns about introducing overflow):

```
7 + 22 + vol
```

and with constant folding, to `29 + vol`

The revised expression saves a run-time addition. You could also convert:

```
1 * vol
```

to simply:

```
vol
```

When X3J11 initially considered the addition of the `volatile` keyword, many members assumed that it would mean that there would be some fixed rules for the number of accesses which could be made to `volatile` objects. For example, there must be exactly one access per reference in an expression. After some study it was realized that this was simply not practical. There are too many limitations which arise in real world hardware. For example, on word-

oriented hardware (which always accesses memory multiple bytes at a time) it may not be possible to avoid accessing a `volatile` location immediately adjacent to another data object, even when it is not being referenced in the current expression. Depending on the size and current alignment of a data object, it may not be possible to access an entire data object in a single load operation.

An implementation is given a lot of leeway when deciding how and when `volatile` data objects will be accessed. It is, however, required to detail in its documentation what constitutes an access to a data object with `volatile`-qualified types. This information can then be used by the programmer to select the correct type for his programming needs. Note, however, that you cannot write a program that reliably accesses volatile data objects across multiple implementations.

When doing optimizations within an interval bounded by two sequence points, an implementation needs to ensure that the optimized code satisfies the rules for the number and types of accesses (loads and stores) described in the documentation. How much leeway they have depends on what will be acceptable to their customer base.

Conclusion

The addition of the new type qualifiers `const` and `volatile` is not a trivial exercise. There are numerous changes in several places in almost any translator. However, the addition of this facility gives the programmer a way to supply useful information to the translator. The `const` and `volatile` type qualifiers are important new tools which will allow C translators to produce very high quality code without compromising the programmer's ability to effectively perform low level tasks from within C.

Jim Brodie is the convener and Chairman of the ANSI C standards committee, X3J11. Jim is also President of Brodie and Associates, a consulting company based in Phoenix, Arizona. His latest book Standard C: A Quick Reference Guide was coauthored with P.J. Plauger and was published by Microsoft Press. Jim is the Standards Editor for The Journal of C Language Translation. He can be reached at (602) 961-0032 or uunet!aussie!jimb.

[Editor's note: We would like to use this column to discuss issues related to, and problem areas in, the C standard. If you have any issues or problems which you would like to see addressed, contact Jim at shown above.]

∞

8. Parallel Programming: Linda Meets C, Part I

Jerrold Leichter
Yale University
and
Digital Equipment Corporation

Abstract

Linda is a programming model for developing explicitly parallel programs. It is radically different from such familiar models as message passing and remote procedure call. Linda is not a complete programming language; it is designed to be injected into existing languages like C, thus making them parallel.

Introduction

There is broad agreement that the future of computing will involve the use of parallel, sometimes physically distributed, machines. How are those machines to be programmed?

Various approaches have been suggested. Some advocate leaving the problem to the compiler, whether an automatically parallelizing compiler for a traditional language like FORTRAN, or one for a super-high-level functional language. Others advocate making the raw communications hardware visible to the application programmer, whether it is a message-passing network or a shared memory. Still others build higher-level models, whether new ones like monitors or extensions of old ones like remote procedure calls.

Linda falls into this last group. It is a programming model for parallel computation. It differs from other models in significant ways, ways we believe make it more flexible and effective. Linda's programming model is at a higher level than many, and as a result is implementable across a broad range of machines, from shared-memory multiprocessors to distributed "machines" consisting of nodes on a network. But it is low enough in level so that these implementations can be efficient. In this way, it is philosophically close to C: It is intended to provide the programmer with power without saddling him with unnecessary and expensive machinery.

Linda is not a programming language; it is a set of objects, and operations on those objects, which can be "injected" into an existing sequential language to transform it into a parallel programming language—that is, a language in which process creation and coordination are supported in the same way that sequential

operations like looping are supported in traditional languages. Because of the similar approaches they take to the programming problem, an injection of Linda into C is natural. Although some work on versions of Linda in other sequential languages has been done, most of our efforts at Yale have centered on C.

In this and subsequent articles, we'll discuss the Linda programming model, the issues that arise in trying to fit it into C, and the languages that have resulted. We'll also discuss some of the implementation issues that arise.

The Linda Programming Model

The Linda programming model was first proposed by David Gelernter in his doctoral dissertation[3] about five years ago. He coined the name *generative communication* to describe it. Gelernter's original work envisioned a complete programming language. Over the years, the essence of generative communication has been distilled out of the original proposal, and has evolved into the language-independent model we will describe.

Fundamental Objects

Linda is based on two fundamental objects: *tuples* and *tuple spaces*.

Tuples are collections of *fields*. Fields have fixed types associated with them; the types are drawn from the underlying language. A field can be a *formal* or an *actual*. A formal field is a place-holder—it has a type, but no value. An actual field carries a value drawn from the set of possible values allowed for that type by the underlying language.

If we take C as a concrete example, then:

$$\langle \texttt{1}_{\texttt{int}}, \texttt{1.5}_{\texttt{float}}, \texttt{2}_{\texttt{int}} \rangle$$

is a tuple with three fields: Two integers, `1` and `2`, and a floating point value, `1.5`. It is essential to distinguish it from the tuple:

$$\langle \texttt{1}_{\texttt{int}}, \texttt{1.5}_{\texttt{float}}, \texttt{2}_{\texttt{float}} \rangle$$

which differs in the type of its third field.

The previous two tuples contain only actual fields. The tuple:

$$\left\langle \texttt{1}_{\texttt{int}}, \square_{\texttt{float}} \right\rangle$$

contains an integer actual, and a float formal.

The definition of the Linda language places no *a priori* restrictions on what types are allowed. *Any* type from the underlying language is allowed. In the

[3] "An Integrated Microcomputer Network for Experiments in Distributed Computing", State University of New York at Stony Brook, 1982. See also, his paper "Generative Communication in Linda", TOPLAS, Volume 7, number 1, January 1985.

case of C, arrays, structs, and pointers, among others, might be reasonable. We'll see later that, in practice, we want to be somewhat more restrictive.

Tuples, in turn, live in *tuple space*, which is simply a collection of tuples. It may contain any number of copies of the same tuple. In mathematical terms, it is a *bag*, not a set. Tuple space is the fundamental medium of communication in Linda. Linda processes communicate through tuple space: All Linda communication is a three-party operation—Sender interacts with tuple space, tuple space interacts with Receiver—rather than a two-party operation as in traditional models.

Tuple space is a *global*, *shared* object—all Linda processes that are part of the same Linda program have access to the same (logical) tuple space. In fact, it is access to this shared object that *defines* what processes constitute a single Linda program.

The Operators

The `out` operator inserts a tuple into tuple space. For example, if `f` is a `float` variable with value `1.5`, and `i` is an `int` variable with value `2`, then:

```
out(1,f,i)
```

would insert into tuple space the tuple we saw earlier, $\langle 1_{\texttt{int}}, 1.5_{\texttt{float}}, 2_{\texttt{int}} \rangle$.

The `in` operator extracts tuples from tuple space. It finds tuples that *match* its arguments, in a sense we will describe shortly. However, equal tuples match, so the tuple of the previous paragraph could be extracted by the operation:

```
in(1,1.5,2)
```

Formal fields are denoted by a question mark prefix. What should follow a question mark is language-dependent. The intent is that it should be "anything that can go to the left of an equal sign."

When a formal is used in an `in`, any actual in a tuple in tuple space will match. The operation

```
in(1,?f,1)
```

might extract the same tuple as our earlier operation, $\langle 1_{\texttt{int}}, 1.5_{\texttt{float}}, 2_{\texttt{int}} \rangle$. In addition to removing the tuple, it would *bind* (assign) `1.5` to `f`. Note that the type of `f` is significant—for this match to be possible, `f` must have type `float`. If `f` has any other type, even `double`, the match will not succeed.

After an `in` operation, the tuple matched is removed from tuple space. The `rd` operator is similar to `in`, but leaves the matched tuple in tuple space unchanged. It is used for its side effects—bindings and synchronization.

The "?" prefix may be used with `out` as well. The tuple

$$\left\langle 1_{\texttt{int}}, \square_{\texttt{float}} \right\rangle$$

could be produced by:

```
out(1,?f)
```

The `eval` operation is a variation on `out`. Like `out`, it inserts a tuple into tuple space. However, the tuple is inserted in *unevaluated* form. Suppose `g` is a function producing a `float`. Then the operation:

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

produces an *active tuple*:

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

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

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

This tuple can be matched in the usual way; for example, it might be matched by

```
in(?f)
```

where `f` is a `float` variable.

Tuple Matching

The `in` and `rd` operators are defined in terms of *matching*. Call the tuple defined by the fields in an `in` or `rd` a *template*. A template $\mathcal{M}$ matches a tuple $\mathcal{T}$ in tuple space if:

- $\mathcal{M}$ and $\mathcal{T}$ have the same number of fields;
- Corresponding fields have the same types;
- Each pair of corresponding fields $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ match as follows:
  - If both $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ are actuals, they match if and only if their respective values are equal, where equality is defined by the base language for objects of this type;
  - If $F_{\mathcal{M}}$ is a formal and $F_{\mathcal{T}}$ is an actual, they match; the value of $F_{\mathcal{T}}$ may eventually be bound. No binding takes place unless *all* the fields match, however.
  - If $F_{\mathcal{M}}$ is an actual and $F_{\mathcal{T}}$ is a formal, they match unconditionally. The value of $F_{\mathcal{M}}$ is discarded.
  - If both $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ are formals, they never match.

If no matching tuple can be found in tuple space, `in` and `rd` block, and the process waits for a tuple to appear. If there is more than one matching tuple, `in` and `rd` choose one non-deterministically.

Examples

Some simple examples illustrate the uses of the Linda constructs. While we haven't yet discussed how to embed the Linda operations in C, in these examples we will stick to constructs whose meaning should be clear.

Simple message passing can be accomplished quite easily. Suppose that the sending process has executed:

```
out("Node 1",3.14159);
```

After the receiver process (`Node 1`) completes

```
in("Node 1",?val);
```

the value of variable `val` will be `3.14159`, and the two operations will have effected the transfer of a floating-point value from the sender to `Node 1`.

Many parallel computations can be organized as a master and a group of workers. Suppose the master requires the results of one hundred computations, and these computations may be performed in parallel. The outline of the master process is then:

```
for (i = 0; i < 100; i++)
        out("Do this",i);
for (i = 0; i < 100; i++)
{       in("Result",?j,?result);
        result_vec[j] = result);
}
```

The corresponding workers have the form:

```
while(1)
{       in("Do this",?job);
        result = do_the_job(job);
        out("Result",job,result);
}
```

The traditional fork/join constructs for creating and waiting for a parallel process are easily implemented. We implement fork as an `eval` of a tuple consisting of the code to be forked, and join as an `in` of the final value of that tuple. The value to which the forked expression evaluates is available to the code which does the `in`. It might be a status value if one is desired, or it may be some meaningless value which can be discarded

Finally, the unlikely-seeming ability to place formals in tuples is rarely needed, but is present to allow for possibilities like the following. Imagine that we have a number of equivalent servers—for concreteness, say printer servers. We are willing to start a communication with any of them, but having started with one we must continue with it. Suppose `my_id` contains some kind of unique identifier in each process. Servers execute

```
in("Request",my_id,?client,?params);
```

to wait for work. Clients execute

```
out("Request",?server,my_id,params);
```

to contact *any* server. The server that receives a tuple replies with a tuple containing its own identity:

```
out("Response",client,my_id);
```

which the client retrieves using

```
in("Response",my_id,?server);
```

From then on, the client may reach its particular server by executing `out`'s of the form:

```
out("Request",server,my_id,params);
```

Summary of Current Status

Two different embeddings of Linda in C have been implemented at Yale. We will talk about them in subsequent articles.

Well over a dozen papers about Linda have appeared in the literature, but except for the Supercomputing '88 paper, which deals with the VAX network system, most earlier articles about Linda are out of date. If you want current reports or users' manuals, drop us a line.

Linda has been used for a pretty wide variety of purposes, ranging from numerical codes to simulation, theoretical parallel algorithm design, biological applications, and on-line intelligent monitoring.

There is a Linda user's mailing list on the Internet. To be added to the list, send a subscription request to any of the following addresses:

linda-users-request@cs.yale.edu
linda-users-request@yalecs.bitnet
[ucbvax!]decvax!yale!linda-users-request

Jerrold Leichter is completing his doctoral research, which includes an implementation of Linda for shared-memory and networked VAXes, at the Yale University Department of Computer Science. He is also a long-time employee of Digital Equipment Corporation, whose Graduate Engineering Education Program supported him during some of his work. He may be reached electronically as leichter-jerry@cs.yale.edu.

∞

9. Miscellanea

compiled by **Rex Jaeschke**

A Note on `wchar_t` Support

When the multibyte stuff was proposed in ANSI C, quite a few members bought in to the idea on the understanding that it "really wouldn't cost them much" if they didn't care to sell to that marketplace. Some vendors are now finding out what that "small" cost is, thanks in part to Plauger's article in the sample issue. At least one of these vendors had (incorrectly) assumed they could simply ignore the `L` preceding a wide character constant, treating it as a "normal" character constant. The standard states:

> "A wide character constant has type `wchar_t` ... which is an integral type ..."

which means `wchar_t` can be a `char` type. If an implementation chooses to equate `wchar_t` and `char` then:

```
sizeof(wchar_t) == sizeof(L'x') == sizeof(char)
```

Therefore, the standard now allows an integer constant to have type `char`. (Previously, all integer constants had type `int` or greater.) This means an implementation cannot just ignore the `L` prefix on a character constant because `'x'` has type `int` while `L'x'` has type `char`. (If you map `wchar_t` to `int`, then string literals of the form `L"abc"` become arrays of `int`.)

Apparently, the Plum-Hall test suite has a check of the form

```
if (sizeof(wchar_t) != sizeof(L'x'))
        complain();
```

Another reason the leading `L` cannot be ignored has to do with the stringize preprocessor operator `#`. For example,

```
#define M(a) #a

char *pc = M(L'a');
```

The string produced by expanding the macro will be `"'a'"` if the `L` is ignored when it should be `"L'a'"`.

Note too, that you can use the token pasting operator `##` to construct a wide character constant as well as a wide character string literal. That is, the `L` prefix can be pasted onto a string during preprocessing.

Questionnaire Results

In December, 1988, the first promotional literature for this publication was mailed to about 1,300 people world-wide. Enclosed was a questionnaire containing a list of more than 60 topics against which respondents were asked to identify their interests. 90 people responded and the results are summarized below. While the primary targets of the mailing were known implementers of C language translation tools and libraries, several hundred knowledgeable applications programmers were also included.

The interest in the items listed follows, with topics shown in descending order of frequency.

Questionnaire Results

Count	*Topic*
61	Optimization
59	Standards conformance and test suites
55	Inter-language calling conventions
54	In-line functions
54	Classes (ala C++)
54	Aliasing issues
53	Standards activities
53	C++ and other prior art
52	Concurrency support (parallel processing)
51	Quality of implementation issues
50	Prototypes
45	Floating-Point issues in general
43	Preprocessor issues in general
42	Headers, macros and library functions
41	Arrays as first class objects
40	Environment control (OS, machine, etc.)
39	RISC architectures
39	Pragmas
39	Namespace pollution lists
39	International issues
39	Anonymous unions (as in Pascal variant records)
39	Address space pointers – near/far pointers

Questionnaire Results (cont)	
Count	*Topic*
38	Vector support
38	Floating-point IEEE issues
37	`void *` experiences
37	Semantics for `register` in prototypes
36	Strong `enum` typing
36	Semantics for shared file access
36	Includability of a header
35	`volatile`
35	yacc and ANSI C
35	Cardinal/unsigned arithmetic issues
35	Benign `typedef`s and tags
35	Automatic aggregate initializers
34	`const`
33	`typeof()` operator
33	`case` ranges
33	`alignof()` operator
33	Line-oriented comments (as in C++)
32	constructors
32	External names – spelling and length
32	Common shared regions
31	`ASM` – in-line assembler
31	Screen I/O control (as in curses)
31	Initializer repeat counts
31	Default parameter values in functions
30	Teaching ANSI C – terminology and issues
30	Preprocessor I/O (`#in`, `#out`)
29	`long double`
29	`errno`
29	Word architectures
29	Representation control (e.g., Integer*4)
29	Embedded environments
28	Zero-sized aggregates
27	Complex arithmetic
27	Charize (stringize for characters)
26	Named parameter association
26	Functions to inspect/report on heap state
26	Embedded SQL
25	`#exit` [message]
22	Permit statements to precede declarations
21	Flushing input streams – discarding `scanf` input
19	Large device support issues
16	Echo of `stdin` to `stdout`
14	Trigraphs

- 47 respondents use an object-oriented language.
- 85 have been following the activities of the ANSI C committee.
- 66 use some flavor of UNIX, 39 use MS-DOS, 22 use VAX/VMS, 14 use a Macintosh, and 20 ran on other hosted systems.
- 6 worked in free-standing environments.
- 45 were actually implementing C compilers, compiler development tools or tools that generated C source. 16 developed support libraries and tools.

In addition to the topics listed above, more than 50 extra topics were identified by the respondents, with quite a few new topics being suggested multiple times.

As a result, the editorial staff now has a better direction in which to focus their efforts. Of course, if you have other suggestions please let us know. A number of the "hot" topics will certainly be on the agenda at future meetings of the Numerical C Extensions Group (NCEG) announced in the sample issue.

One particular editorial decision that has been made is that *The Journal* will **not** be a C++ publication. Certainly, C++ has had, and may continue to have, an influence on C. We may reflect that in an occasional article. However, there is a sufficient number of publications providing editorial space for C++, and there are plenty of other C-related issues for us to discuss. Also, there are those among us who would suggest that C and C++ are quite separate languages.

Electronic Polls

Occasionally, I'll be conducting polls via electronic mail and publishing the results in the issue following. Examples of topics planned for the first poll are:

- What pragmas do you support?
- What predefined macros do you have?
- What structure member packing options do you provide, if any?
- If you support case ranges, what syntax do you use?

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

Calendar of Events

- July 10–14, **Writing Fast Compilers** – Provides broad coverage of compiler technology at an elementary level. Topics include language description, scanning, macro definition and expansion, parsing, symbol tables, generators, code emitters, object modules, linkers, loaders, and debuggers. Also, the theory of LALR parser generation and use.

  Participants need no previous experience in compiler writing, but should know C. The course leaders are Dr. William McKeeman and Gary Pollice. Wang Institute of Boston University, 72 Tyng Road, Tyngsboro, MA 01879. (508) 649-9731. Cost is $1,475.

- August 7–11, Stanford University, **Compiler Construction and Programming Language Translation** – State of the art methods for constructing compilers for modern high-level languages. Students will work with compiler writing tools on a project. No compiler experience required.

  Seminar leaders are Dr. Susan L. Graham, Dr. John L. Hennessy, and Dr. Jeffery D. Ullman. The text will be *Compiler Design: Principles, Techniques, and Tools* by Aho, Sethi and Ullman. Cost is $1,175. (916) 873-0575. (See the August 14–18 entry too.)

- August 8–12, **International Conference on Parallel Processing**. Held at Pheasant Run resort in St. Charles, Illinois. The day before and after the conference proper there will be tutorials offered. Contact Dr. Peter M. Kogge, MS 0302, IBM Corporation, Route 17C, Oswego, NY 13827, (607) 751-2291 or owego@IBM.COM.

  Topics of interest from the advanced program include: software tools, sorting and searching, compilers, algorithmic potpourri, languages, graphs and trees, compilers – data dependences, and program transformations. A tutorial called "Parallel Languages and Parallelizing Compilers" is also being offered.

- August 14–18, Stanford University, **Code Optimization and Code Generation** – A follow-on option from the previous week, run by the same leaders. Cost is also $1,175, or $2,100 for both weeks.

- September 19–20, Salt Lake City, **Numerical C Extensions Group (NCEG) meeting** – The second meeting will be held to consider proposals by the various subgroups formed at the Minneapolis meeting in May. It will precede the X3J11 ANSI C meeting being held at the same location later that week. Contact Rex Jaeschke at (703) 860-0091 or uunet!aussie!rex. The third meeting is scheduled for March 7–8, 1990, in New York City.

- September 21–22, Salt Lake City, **ANSI C X3J11 meeting** – Sponsored by DECUS. This one-and-a-half day meeting will handle questions

from the public, interpretations and other general business. Address correspondence or enquiries to the vice chair, Tom Plum, at (609) 927-3770 or uunet!plumhall!plum. The following meeting is scheduled for March 5–6, 1990, in New York City.

- October 10–12, **Frontiers of Massively Parallel Computation** – To be held at George Mason University, Fairfax, Virginia. Cosponsored by IEEE. Conference chair is James Fischer at NASA, (301) 286-9412.

News, Products and Services

- A new **C language validation suite** has been announced. It is a joint effort by HCR Corporation of Toronto, Canada and ACE Associated Computer Experts bv of Amsterdam, The Netherlands. They will jointly market the *C Test super-suite* starting in the third quarter of 1989. For more information contact Marco Roodzant at ACE, (31) 20 6646416 or *marco@ace.nl*, or Heather Grief at HCR, (416) 922 1937. Validation suites are also available from Plum Hall, (609) 927-3770 or uunet!plumhall!plum, and Perennial, uunet!sun!practic!peren!beh or (408) 727-2255.

- In the market for a generic version of **lint** in source form? Gimpel Software of Collegeville, PA is selling a generic version of their popular DOS product PC-Lint. OEM enquiries are welcome. Call Anna at (215) 584-4261.

- If you implement tools in the MS-DOS environment you might want to take a look at **LALR** a parser generator from LALR Research of Knoxville, TN. LALR V3.0 is a complete LALR(1) parser generator. The $99 package includes grammars for Ada, BASIC, Turbo Pascal, and Turbo C. Parser skeleton and error recovery source is provided, as is source for a lexical scanner and syntax checker. 60-day money back guarantee. Hard to beat at that price. (615) 691-5257.

- **METAMORPHOSIS** is a generic utility that aids transformation of syntactically reducible character-oriented files to any other form while preserving the synonymy thereof. For example, it can translate Fortran to Ada, Jovial to Ada, and any language to C. It also functions as a custom compiler, assembler, macro processor, query language processor, and report generator. Requires MS-DOS. J. H. Shannon Associates, Inc., PO Box 597, Chapel Hill, NC 27514. (919) 929-6863.

- Need to **translate Pascal to C**? Intermetrics, Inc., is now selling source code to the seasoned product from Whitesmiths, Ltd. It supports ISO Pascal (Level 1) plus numerous extensions, and comes with a Pascal runtime library written all in C. Intermetrics, (617) 661-1840.

- **ANSI NEWS:** Graphics folks may be interested in the Programmer's Hierarchical Interactive Graphics System (**PHIGS**), dpANS X3.144-198x, which specifies a language-independent nucleus of a graphics system. A C language binding was available for public comment through January 9 of this year.

  Embedded SQL also has a proposed standard. It's called X3.168-198x and is dependent upon the SQL Standard X3.135-1986. The X3H2 Chair is Donald Deutsch, at (301) 340-4580.

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

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

- IPT of Palo Alto, CA is shipping **lint-PLUS**, a static analysis tool for C on VAX/VMS. Program trace and debugging options are also available. (415) 494-7500.

- Need **IEEE floating-point libraries** in source form? US Software of Portland, Oregon is shipping FPAC/DPAC for numerous Intel, Zilog, and Motorola processors and microcontroller chips. (800) 356-7097 or (503) 641-8446.

- GeoMaker is shipping **SymTab** and **PTree** for DOS systems. SymTab provides symbol table management capabilities, while PTree is a parse tree manager. $49 and $69, respectively. (415) 680-1964 in Concord, California.

∞

10. Books and Publications

We receive a continuous stream of C-specific and other books and publications. Those that make it through the filtering process will either be of interest to C implementers or to their customers. Many product documentation sets include "Recommended Reading" sections where further information can be obtained about C, or ANSI C in particular. As such, some of the books we review will make suitable additions to these lists.

Theory of Computation
Formal Languages, Automata, and Complexity
J. Glenn Brookshear
1989 Benjamin/Cummings
$37.95, 322 pages
ISBN: 0-8053-0143-7

reviewed by *Rex Jaeschke*

"I designed this book to serve as a text for a one-semester introductory course in the theory of computation ... My goal in writing this book was to present the foundations of theoretical computer science in a format accessible to undergraduate computer science students. I wanted students to appreciate the theoretical ideas as the foundation on which real problems are solved, rather than viewing them as unusable abstractions."

The main topics covered are: regular, context-free, and general phrase structure languages along with their associated automata; computability in the context of Turing machines, partial recursive functions, and simple programming languages; and complexity theory with an introduction to some of the open classification problems relating to the classes *P* and *NP*.

The introduction provides a review course in set theory, the grammatical basis of language translation, and some historical background.

The five main chapters cover: Finite automata and regular languages; Pushdown automata and context-free languages; Computability; Complexity; and Turing machines and phrase-structure languages. The five appendices include discussions on the construction of LR(1) parse tables, some important unsolvable problems, and the string comparison problem.

Each chapter is concluded with a set of review problems. However, no answers are provided.

Numerous transition diagrams, tables and figures complement the well-written text which reads quite easily. It's appropriate for a refresher course or for those just breaking into the implementation world.

ANSI C: A Lexical Guide

Prentice-Hall
$35.00, 565 pages
ISBN: 0-13-037814-3

reviewed by *Rex Jaeschke*

According to the introduction, "*ANSI C: A Lexical Guide* describes the American National Standards Institute (ANSI) standard for the C programming language. It discusses in clear English every library function, every macro, and every technical term that appears within the Standard. All entries are fully cross-referenced internally to the Standard and to the second edition of *The C Programming Language*; many are illustrated with full C programs."

"In this book, Mark Williams Company presents a reading of the ANSI C Standard, based both on our participation on the committee that wrote it, and on our experience as writers of C compilers and operating systems. This book contains *all* the information you need to write strictly conforming C programs that can be compiled and run on *every* computer for which a conforming implementation of C exists."

Certainly, the use of the terms *all* and *every* indicate some marketing license since the front cover boldly states "Based on Draft-Proposed ANSI C" and that as of this writing (early May 1989) there is still not yet an official standard. Also, page 3 reads "Both [ANSI and ISO] felt that the greatest benefit would be wrought if the two standards synchronized, and in final form, were identical. This goal has been achieved." [Ed: Not yet it hasn't, but I'm hoping it will be as I've just accepted the role of X3J11's International Representative to ISO.]

Actually, the book was completed in March 1988 and there have been two X3J11 meetings and one public review period since. Fortunately, the only significant changes that have resulted since the book was published are the removal of the `noalias` keyword, the change from `CLK_TCKS` to `CLOCKS_PER_SEC`, and some tightening up of wording.

The book is a collective effort by employees of Mark Williams Company. (They remain nameless since there are no credits or acknowledgements listed.) This company has been developing and selling compilers and operating systems to the mini- and micro-computer marketplace since 1976.

The book is aimed at programmers who want a detailed guide to the ANSI C language, preprocessor, and run-time library arranged in alphabetical order, in one place. From that point of view, the book is quite successful. It contains some 580 entries each organized as: name, syntax (if applicable), description, example, and cross-references. While most entries take up a third to a half a page, many are quite small. Some take up several pages.

I found it strange that a book organized essentially as an alphabetically ordered glossary had a cross-reference index as large as 26 double-column pages.

Certainly, the index is extensive and, along with the cross-references accompanying each entry, lets you navigate the book from pretty much any point.

When I look at a new book about "modern" C, the first thing I look for is whether it uses modern C terminology. Things like "storage duration," "object- and function-like macros," "file position indicator," and "modifiable lvalue." In this respect the book excels—it appears to have all the terms introduced or made popular by X3J11.

The book does indeed cover every operator, punctuator, keyword, function, macro, and type in ANSI C. It also provides a credible overview of the locale and multibyte additions. And, unless the draft Standard is significantly rearranged, all references to specific section numbers should remain intact for the final version of the standard.

As a text for the average C programmer (its intended audience), *ANSI C: A Lexical Guide* is very good. I have no reservation in recommending that you add it to your customer "Recommended Reading" lists. It does contain, however, many rough edges. These involve lack of precision and care with detailed and complicated corners of the language, rather than outright technical errors or misleading statements. As such, these problem areas (many of which are quite subtle) are unlikely to cause most readers any problem.

A few examples of the problem spots should demonstrate my point.

- `define` is continually called a keyword—it is not one.
- "When parentheses precede an identifier and enclose a typename alone, then they function as a cast operator. Here, the type of the identifier is changed, or cast, to the type enclosed within parentheses." The term *identifier* is often used instead of *expression*. Also, the type of an identifier cannot be changed.
- "If a pointer points to an array, then the result of addition will point to another member of the same array ..." confuses pointers to arrays and pointers to elements of an array.
- The discussion of the dot operator claims that the left operand must name a structure or union when, in fact, it can be any expression that designates a structure or union.
- Quite a few entries have disorganized, weak, or incomplete discussions. These include: the `[]` operator, `const`, function prototypes, and sequence points. The discussion of lvalues, on the other hand, is quite good.

The only out-and-out error I found was the implication that only one macro in `float.h`, `FLT_ROUNDS`, was a compile-time constant. There is only one, but it is `FLT_RADIX`.

Before the next printing, the book could stand a thorough review by someone outside the company who is intimately familiar with the ANSI C Standard.

With some tightening of terminology the book's rating would rise from "very good" to "great".

Standard C: Programmer's Quick Reference Guide
P.J. Plauger and Jim Brodie
1989 Microsoft Press
$7.95, 207 pages
ISBN: 1-55615-158-6

reviewed by *Rex Jaeschke*

According to the introduction, "This quick reference guide for the Standard C programming language provides all the information you need to read and write programs in Standard C. It describes all aspects of Standard C that are the same on all implementations that conform to the standard for C."

The book is divided into three parts: language, library, and appendices. The language part is further divided into the topics Characters, Preprocessing, Syntax, Types, Declarations, Functions, and Expressions. The library part has an introduction and one section for each of the 15 standard headers. Appendix A contains eight pages of information about the "hot spots" involved in porting code and Appendix B lists all standard names alphabetically, giving their parent headers and attributes (where applicable). For example, `fpos_t` resides in `stdio.h` and is an "assignable type definition;" `stderr` is in `stdio.h` and is a "pointer to `FILE` rvalue macro;" `putc` is in `stdio.h` and is a "function or unsafe macro." Future reserved words are also listed.

Syntax diagrams replace the commonly used formal grammar specification, making it much easier for the lay reader to build valid constructs.

Both authors are intimately involved with the ANSI and ISO C standards efforts, and have been from the beginning. In fact, Brodie convened the ANSI C committee while Plauger is the ISO C convener. This involvement shows, as the text reflects all the definitions and terminology introduced in these standards. The text is crisp, correct, and with no fat—all typical Plauger traits. In fact, Plauger was directly responsible for much of the new terminology added to the draft.

The sections are generally quite small "bite sized" chunks and contain references to other sections as appropriate. The 11 pages of cross-reference index are adequate.

The book reflects the most recent draft of the ANSI C standard, produced at the September 1988 meeting. It also includes the last minute modification to `time.h`, `CLOCKS_PER_SEC`. Since there is every expectation that this version of the draft will become the final ANSI Standard without change, the book should not need any technical revision when the standard is finally approved.

While many C books get "most of it right," *Standard C* does that and more. In particular, it excels in the following complex areas:

- Phases of translation
- Preprocessor macro handling
- Incomplete, compatible, and composite types
- Linkage
- Type conversion
- Grouping
- Side effects and order of evaluation

In conclusion, this small book is not only a lay person's guide to the ANSI C Standard, it's a very useful and coherent distillation of the formal standard that would suit C language implementers, particularly those who have not been active within the X3J11 committee.

And to those of you struggling to write a Standard C language manual for your compiler, you might want to consider distributing this book instead.

I've placed this book in the #1 slot on the list of C language reference texts I recommend to my many C seminar students. A must buy for introductory and advanced C programmers alike.

∞