The Journal of

C Language Translation

Volume 2, Number 3

December, 1990

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

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

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

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

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

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

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

Contents

19. Generalizing Type Qualifiers

P.J. Plauger

Abstract

The ANSI standard introduced type qualifiers into the C programming language. Signaled by the keywords `const` and `volatile`, the standard type qualifiers lie midway between types and storage classes. The common extensions `near` and `far` also behave as type qualifiers.

This article discusses why neither types nor storage classes are adequate vehicles for these added semantics. It also discusses ways to further generalize type qualifiers and use them to express a number of useful extensions to C. The concepts have been tested in a family of commercial C compilers over the past several years.

The Need for Constant Data

X3J11 knew from the outset that something like `const` had to be added to the C programming language. Programmers *need* a way to steer lookup tables and other reference data into ROM. Those of us selling compilers for embedded applications had been working around its absence for years. A common trick was to specify where to put data on a file by file basis.

The early Whitesmiths C compilers, for example, distinguished three logical contributions to the final code image:

- *Program text* consists of the contents of all the function bodies. It is execute-only and never alters itself.
- *Literal data* consists of any data objects generated by the compiler, such as string literals, switch tables, or floating point constants. It is read-only (yes, even for string literals).
- *Writable data* consists of any data objects declared in the program. It is assumed to be read/write, in the absence of any other information.

The code generator steered these three logical contributions to two physical segments in the executable image—the *code* segment and the *data* segment. That matched the restrictions of many assemblers, linkers, and operating systems in the late seventies. Two mappings were most common:

- A machine with separate I-space and D-space needed program text alone in the code segment. To access literal data from I-space was often difficult or impossible. So literal data and writable data shared the data segment. Write protection of literal data went out the window.
- A machine with ROM and RAM could pack program text and literal data together in the code segment. That let the literal data be write-protected. It also minimized the amount of RAM that had to be initialized on program startup. Writable data was alone in the data segment.

To steer certain data objects into the read-only code segment, you put them in a separate file. A compile-time switch specified how the contributions got steered. It was a messy system, but it was useful to many programmers.

X3J11 wanted to do better than this. We felt that compile-time switches were inelegant. They were also outside the charter of a language standard. We wanted some way to bring the information into the language proper. An early suggestion was to introduce a new storage class. Declare a data object with storage class `const` and it goes into ROM instead of RAM. That meets the immediate need, but it has a few holes.

Storage class also conveys linkage information. The rules for doing so are already complex, because the keywords `extern` and `static` are grossly overloaded. A programmer probably wants to specify both external constant data and internal constant data.

That argues that `const` should at least be a storage class qualifier, not just another overstressed storage class designator. Make it a qualifier and you can specify constant register, argument, and automatic storage as well. None of these creatures can be placed in ROM, but you can at least help the compiler catch obviously silly stores.

Once you decide you want to diagnose silly stores, however, you come face to face with another shortcoming. Take the address of a data object and you lose information about its constancy. C can't distinguish between a pointer to static storage and a pointer to automatic storage. There is no sensible way to require it to distinguish constant from nonconstant storage. No stores through pointers can be checked.

A new storage class won't do the trick, nor will a storage class qualifier. That suggests that `const` should be part of the type information. You hardly want to add just a single new type called `const`. Every data type wants to have a constant counterpart. So it makes sense to let the keyword `const` qualify any of the existing types. Naturally, a `const short` has the same representation as a `short`. It merely carries the added proviso that it is not to be modified by the executing program.

So far, `const` has much the flavor of `signed` or `unsigned`. Most popular computers today represent signed integers as twos-complement, so a change of signedness doesn't even alter the representation. Only the values associated with certain bit patterns change. It looks at first blush like `const` can be considered just another 'type part,' like all the older type keywords.

There is one important difference, of course. A pointer type has two opportunities to acquire `const`ness. You can talk about a pointer to a constant `char` or a constant pointer to `char`. In fact, you can talk about four different combinations of `const`ness. All make sense. That's what led X3J11 to accept the `const` qualifier following any `*` in a declaration.

This calls for a decision, by the way. It is completely arbitrary how you choose to interpret:

```
const char *p1;
```

You can say that `p1` is a constant pointer to `char` or a pointer to constant `char`. Then the alternate declaration:

```
char * const p2;
```

has the other meaning. As a matter of common sense, the committee chose to stick with the convention in C++, from which we borrowed `const`. The first declaration above declares `p1` to be a pointer to constant `char`. The second declares `p2` to be a constant pointer to `char`. (Read `* const` as *pointer which is constant.*)

Semantics of Type Qualifiers

Even though `const` has a funny way of decorating pointers, it still doesn't look all that special. The committee had to work up a mess of new semantics, such as:

- How constant data participates in type balancing across arithmetic operators.
- How `const` affects assignment compatibility.
- To what degree `const short` is a different type from `short`.

We faced much the same set of issues in adding `signed` and `long double`.

After a bit of experience, however, `const` started looking a bit more peculiar. Unlike any of the other type information, the presence of `const` makes sense only when you're trafficking in lvalues. `const` is a statement about how you can access a data object. It says nothing about the representation or the operations you can perform on values stored in the data object. In fact, we found it best just to say that `const`ness evaporates whenever you extract an rvalue. That saved us from generating considerable error-prone verbiage when we summarily doubled the number of possible types.

Until `const` came along, type information was used solely to determine representation and operations. How many bits does a data object of that type need? On what kind of storage boundary? What values do you attribute to

the various bit patterns? What operations does the language permit, on what values? This is the realm of data types.

Yet `const` addresses none of these questions. If you stretch a point, you can argue that it restricts the operations permitted by the assigning operators. That's not a strong case for calling `const` type information, however.

Storage class information traditionally deals with the addressability of data objects. Static data is directly addressable in memory. Automatic data is on the stack. Registers are someplace funny. A compiler that speaks assembly language even distinguishes between static names with external and internal linkage. The former derive by simple rules from the name itself. The latter must be private to a given translation unit.

But once the program obtains a stored value, it can forget what it took to locate the storage. Storage class distinctions evaporate when an lvalue becomes an rvalue. That sounds more like the arena where `const` wants to play. You can see why the committee started out thinking that we wanted a `const` storage class.

So this is the peculiarity of `const`. It wants to qualify type information, so that it can pop up in all the right places. But it behaves more like a storage class, since it only affects how you can access a data object with an lvalue. That's what I mean when I say that type qualifiers lie midway between types and storage classes.

The Need for Volatile Data

Adding `const` to C was a big help, but it wasn't the whole story. Another problem was commonplace in embedded programming. It stemmed from the widespread practice of controlling I/O devices with memory-mapped registers.

Many of us became converts to C because we could write low-level code in a more readable high-level language. Set a pointer to an absolute address and you can directly manipulate magic registers. The PDP-11 went crazy with memory-mapped I/O. Many other more recent architectures have imitated this winning strategy. In such an environment, you *can* rule the world from C.

The only problem was one of determinism. You have to know what kind of code a C compiler generates for various expressions, lest it outsmart you. A classic example is writing characters to an output buffer register. On a PDP-11, you might write something like:

```
#define XBUF (int *)0177566

        *XBUF = '\r';
        *XBUF = '\n';
```

The code stuffs a carriage return followed by a new-line into a UART transmitter buffer. At least it does so if the compiler is sufficiently stupid. Should it recognize that two successive stores occur to the same data object, it might

choose to optimize the first one away. Your program mysteriously fails to emit carriage returns.

A classic solution in the early days of C was to alter such a program to read:

```
int *xbuf = (int *)0177566;

        *xbuf = '\r';
        *xbuf = '\n';
```

If you find that the compiler is not brave enough to optimize out the first store, you're done. You leave that code alone and go on to the next problem. Perhaps years later you upgrade to a new compiler. Only then do you discover that you have a smarter optimizer to outsmart.

X3J11 wanted some way to end this arms race. We wanted to encourage aggressive optimization, yet still keep the world safe for people writing low-level I/O handlers in C. Our solution was to give programmers a way to mark those data objects that needed delicate handling. If a translator knew that other agents could tinker with a given data object, it could be careful not to optimize away accesses (or move them about rashly). For those data objects, the translator would generate code that reflects the overt intent of the C source. Just like in the old days.

That's where the type qualifier `volatile` comes in. You can write it in all the places where you can write `const`. You can even mix `const` and `volatile` type qualifiers in all possible combinations. Any lvalue that has a volatile type is handled with kid gloves. The optimizer knows to take a holiday.

Ironically, the great contribution of `volatile` is when you don't use it. C can now assume that any data object not marked `volatile` is the private property of the translator. It can be much more aggressive about optimizing accesses to such data objects. It can even advance or retard accesses to nonvolatile data objects past sequence points. It has much more latitude in generating the same code *as if* the sequence points were honored.

`volatile` is an even odder creature than `const`. Semantically, it carries no traditional type baggage at all. Even as a storage class qualifier, it is an odd duck. Dennis Ritchie has never had much good to say about `volatile`, and I can't blame him. It is a dirty little piece of pragmatism that doesn't even deliver completely on any of its promises. Nevertheless, I feel that it provides a needed service in the real world of C programming.

`near` and `far`

Another dirty piece of pragmatism didn't make it into the C standard. I refer, naturally, to the type qualifiers `near` and `far` bolted onto C by Intel 8086 compiler vendors. These came about because the IBM PC made this architecture obscenely popular. The pressure was overwhelming to produce great quantities of efficient C for an architecture that is inhospitable to the language.

Basically, C evolved on machines with flat address spaces. The language can tolerate separate spaces for functions and data. (That's the I-space and D-space model I described earlier.) But it doesn't like to have a chopped up data space. A pointer to data must be prepared to point anywhere. It forgets storage class, as I keep repeating.

The Intel 8086 runs fastest when it uses only 16-bit pointers. That, unfortunately, limits a program to 65,536 bytes of data. In today's world, such a limitation is simply unacceptable. Traffic in 32-bit pointers and you can use all the memory you can eat. (I won't go into the relative digestibility of the 640KB that DOS leaves for programs versus the 16MB or 1GB available in principle on newer machines.) The only problem is, programs get twice as big and run half as fast.

Pragmatism often dictates a hybrid solution. You use 16-bit pointers where you can, 32-bit pointers where you must. Sure, it makes for messier programming. And sure, it's nonportable code. But when your potential marketplace measures in the tens of millions of machines, you make compromises.

`near` and `far` are as peculiar in their own way as are `const` and `volatile`. Again, they say nothing about the representation of the type they qualify. They do affect the representation of the address of a `near` or `far` qualified data object. Hence, they affect the representation of any pointer you declare that points to a `near` or `far` qualified data object. But don't let that distract you. Once again, these type qualifiers affect only how you go about locating data objects.

Other Address Spaces

The `near` type qualifier has a chancy interpretation. You promise that a 16-bit offset will do, but you have no way of specifying the base that accompanies the offset. The politest thing you can say is that current implementations using `near` are linguistically fragile. It would be nice if there were a more precise, and more general, way of specifying when 16-bit offsets are acceptable.

Back before `near` and `far` became commonplace, I solved the more general problem. About five years ago, I added some peculiar machinery to the Whitesmiths family of C compilers. It supported an open-ended set of type qualifiers that could be tailored to the peculiar needs of different machines. I called them *address space modifiers*, because they usually affected where the program looked to find the qualified data type.

The syntax I used was hardly outstanding. An at sign (`@`) signaled that the identifier following was the name of a special address space. In the case of the Intel 8086 C compiler, four address spaces were obvious additions. If you declare

```
@cs char *pcs;
```

then `pcs` is a 16-bit pointer into the code segment, or `cs`. The compiler knew

to generate the `cs:` segment override prefix any time it dereferenced such a pointer. You got corresponding behavior with the address space modifiers `@ds`, `@ss`, and `@es` (for the data, stack, and extra segments, respectively). These were near pointers with no ambiguity. By contrast, the address space modifier `@far` warned of the need for an arbitrary segment plus offset address. It is equivalent to the `far` keyword that has become an institution.

If that were the end of it, the experiment would be just a minor historical footnote. The `near` and `far` keywords popularized by Borland, Microsoft, and others have clearly prevailed. Programmers have survived without the extra refinements that this family of compilers imposed on `near` pointers. Fortunately, the machinery has proved to have other uses as well.

Computers can have any number of address spaces. Even on the Intel 8086 architecture, at least one other space exists. Input and output occur through a set of ports that have nothing to do with memory. You write *in* and *out* instructions to access these ports. Memory-mapped I/O seldom gets used in this universe.

To perform port I/O from a C program can be a nuisance. One way is to call tiny assembly language functions and incur the call/return overhead. Another way is to write inline assembly code and risk both nonportabilities and suboptimal code. Address space modifiers offer a third and better alternative. Declare a data object in the space `@port` and the compiler knows to express accesses as *in* and *out* instructions. It also knows how to optimize arbitrary expressions containing references to ports, pointers to ports, and so forth.

For ports to work well, another extension is required. You often want to declare that a data object resides at some absolute address. You can sidestep the issue by assigning absolute values to pointers, as I showed earlier, but that is not as nice. So I added yet another bit of notation for defining absolute addresses:

```
int xcsr @0177566;
@port char rstat @0x40;
```

The first declaration defines `xcsr` as living at a particular place in memory. The second defines `rstat` as the `char`-sized port number 0x40. (I make no apologies for the overloaded use of `@` to signal various extensions to C. As the old saying goes, you can always hide it with a macro.)

Other people have used this machinery for even more peculiar extensions. They really pay off on bizarre little chips that barely support C to begin with. Performance is much more of an issue and standards conformance much less. Some of the extensions are:

- *Zero page* addressing, for those small architectures that favor accesses to the first few hundred bytes of memory.
- *Bit vector* addressing, for a chip that implements fast Boolean logic in a special array of bits.

You can even qualify the return type of a function to give it special properties. You might want the function to handle interrupts or traps directly. Or you might want it to be callable from some other language. In either case, you want the compiler to generate an alternate call/return sequence. You might even want to promise that a function has no side effects, to encourage a reluctant optimizer. All of these extensions have been expressed with address space modifiers.

Summary

The major point of this article is that type qualifiers are a semantic wart on the language. None of them has very clean semantics, not even `const`. It is even hard to craft a general statement about what they all do.

Nevertheless, they have redeeming social value. Type qualifiers crop up in a place where C rubs raw against the real world. They let the programmer make promises about stuff that looks like memory even when it has bizarre limitations. That's just the sort of thing you need in the blue collar world of C programming.

My experience is that generalized type qualifiers form a good hook for extending C. You can think of them as a standard way to introduce nonstandard features into C. Get the semantics right for carrying around any of the type qualifiers and you've got them right for all of them. That goes a long way toward adding extensions without doing utter violence to the C language.

P.J. Plauger serves as secretary of X3J11, convenor of the ISO C working group, and Technical Editor of The Journal of C Language Translation. He recently took over the editorial reins of The C Users Journal. He is currently a Visiting Professor at the University of New South Wales in Sydney, Australia. His latest book, Standard C, *written with Jim Brodie, is published by Microsoft Press. He can be reached at* uunet!plauger!pjp.

∞

20. European C Conformance Testing

Neil Martin and Dan Chacon
British Standards Institution

Abstract

In *Volume 1, number 2* of *The Journal*, I outlined the process that the British Standards Institution (BSI) had undertaken to set up a C compiler Validation service using the Plum Hall C Validation Suite. The service is also recognised by the European Commission as a *harmonised* service. Since that time many things have changed. There is now an ANSI standard for C and an identical ISO standard is expected to follow in the near future. This paper identifies the compilers that are now validated, describes the problems we have encountered, and how vendors and other interested parties can participate in the validation process and the benefits of validation.

World's First Validated C Compilers

In July of this year BSI Quality Assurance (BSI QA) announced the launch of its C compiler validation service. In order to make the process as fair as possible for all participants, we set a deadline for initial applications for validation of the 1 August 1990. All the applications received by this date were processed successfully, and in early September BSI QA made a press announcement under the title "World's First Validated C Compilers." The companies whose products passed the validation process were as follows:

- INMOS Ltd, with six compilers, as follows:
  - PC/MS-DOS serving both T800 and T425 Transputer compilers
  - SUN-3/UNIX host to both T800 and T425 Transputer targets
  - SUN-4 host/UNIX host to both T800 and T425 Transputer targets

  The INMOS host-target systems are interesting for a number of reasons, not least being that the normal host-target relationship is somewhat blurred. The compiler ran on the Transputer in the case of the PC, but the SUN-based products have both native and Transputer binaries supplied, thus the user may run the compiler on either the host or the target. In all cases, the results of execution are observed on the host. INMOS achieve

this unusual but highly effective design by using so-called *iservers*, which act as the interface between the compiler library and any machine that INMOS adopt as a development platform. Compiled applications running on the Transputer also use the same approach, which takes advantage of INMOS's iserver communications protocol.

- Jensen and Partners Ltd; TopSpeed C under MS-DOS

  TopSpeed C is a very modern sophisticated PC compiler. It comes with a fully integrated development environment, and has the usual wealth of options and features expected of a DOS compiler. This compiler has already made a big name for itself in Europe partly because of Jensen and Partners policy that if you are not satisfied with the product you can return it and get your money back.

- Knowledge Software Ltd, with two compilers:

  – MCC hosted on PC/MS-DOS to C standard abstract machine
  – MCC hosted on SUN-4 to C standard abstract machine

  MCC stands for 'Model C Checker.' This is actually one of the tools with which BSI QA measured the standard coverage of the test suite. The MCC is actually an interpreter-based system. The fact that it is interpreted allows run time checking of source code. The MCC, in addition to the diagnosing of normal constraint errors, flags all undefined and implementation-defined behaviour listed in the standard. The target is described as C standard abstract machine because the interpreted environment is the minimum described in the C standard.

Active Participation

One aspect of formal validation, that many C compiler vendors may not be aware of, is the technical review board. This is the group of people with a variety of interests that effectively control what goes in and out of the test suite used for formal validation. In addition, they participate in any appeal process. If prior to a formal validation a vendor wishes to challenge the validity of a test then the first point of call is the technical review board. In addition to challenges, the technical review board is responsible for reviewing any standard interpretations and advising on their impact to the test method. BSI QA would be pleased to hear from any vendors wishing to participate in this process. Please send email to: *cvs-euro@bsiqa.co.uk.uucp*.

USA Validation

In the USA, validation is a slightly different affair. The whole process is administrated by NIST, the US National Institute of Standards and Technology

located near Gaithersburg, Maryland. Validation in the USA is, in general, a mandatory requirement for vendors bidding on Federal contracts. The stated aim of this process is as follows:

- Encourage more effective utilisation and management of programmers by insuring portability of programming skills.
- Reduce the cost of program development by the effective use of high level languages.
- Reduce overall cost of software by making it easier to maintain and transfer programs among different computer systems.
- Protect the existing software assets of the Federal Government.

One other major difference between the U.S. and Europe is that in the U.S. validation is not against a National (ANSI) or International (ISO) standard, but against a document known as a FIPS[1], which is an acronym for Federal Information Processing Standard. The FIPS publications do normally refer to an ANSI or ISO standard. However, NIST reserves the right to make additional requirements over and above the referenced standard. In the case of the C programming language, NIST are planning to make some additional requirements which just may come as a shock to vendors that have not seen the draft FIPS for C and are planning to be formally validated. Theses additional requirements are as follows:

> A facility must be available in the processor for the user to optionally specify monitoring of the source program at compile time. The monitoring will be for all obsolete language elements included in the processor, or all C language elements that are not in conformance with the standard, or both. Any syntax used in the source program that does not conform to that in the FIPS will be diagnosed and identified to the user through a message on the source program listing. Any syntax for an obsolete language element included in the processor and used in the source program will also be diagnosed and identified through a message on the source program listing. In addition the message will identify:
>
> - The statement or declaration that directly contains the nonconforming or obsolete syntax.
> - The source program line and an indication of the beginning of the location within the line of the statement or declaration which contains the nonconforming or obsolete code.
> - The syntax as 'nonconforming nonstandard' if the nonconforming syntax is a nonstandard extension included in the processor, and monitoring for all C language elements that are not in conformance with the standard is selected.

[1] For a status report on the FIPS C Standard, see page 248.

The introduction of additional requirements by a FIPS can easily be justified, as these publications cover more than just a standard reference—they also give guidance on interpretation of a FIPS and on validation. However, the introduction of additional requirements without (I believe) consultation with X3J11 seems to be misguided. In addition to specifying the Federal requirements for a C processor, the C FIPS also gives contacts within NIST for both interpretations of the FIPS and validation to the FIPS.

This brings us to the subject of NIST's solicitation for a C validation suite. In the past, NIST has always managed to obtain the use of a test suite to measure conformance to FIPS at very low cost to itself. In attempting to repeat this exercise for C, they issued a solicitation (No. 52SBNBOC6042) to obtain a test suite for C. There was no need to read between the lines of the solicitation to realise that the primary purpose of the solicitation was to obtain a test suite and training in its use for NIST staff at no cost to themselves. While it is clear that NIST has little money to spend on validation, it is not obvious that prioritising cheapness over technical merit, or any other feature for that matter, is beneficial to either NIST or the US C community in general.

Marketing of Standard Conformance

The difficulty of turning standard conformance into a marketable commodity has been a problem for most producers of software products. In order to help rectify this problem, BSI has registered the name CERTware as a world wide trademark. The idea behind this mark is to give the consumer an easily identifiable logo that they can associate with a quality product. This mark will be available for use by all vendors who have successfully passed BSI conformance testing. The mark has already been launched in the UK along with the C compiler validation service and has received substantial press coverage.

Neil Martin and Dan Chacon both work in the C and POSIX validation division of BSI Quality Assurance in Milton Keynes, United Kingdom. They may be reached via electronic mail at neil@bsiqa.uucp *and* dan@bsiqa.uucp, *respectively.*

∞

21. Parser-Independent Compilers

W.M. McKeeman, Shota Aki, and Scot Aurenz
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract

The parse of a text is a sequence of grammar rules applied to source input. The text is brought into the parser as shift actions and the rules are applied as reduce actions. The resulting shift/reduce sequence has some useful properties as an intermediate language for compilers. It is independent of the parsing technology used to produce it. It can be stored in a file. It can be incrementally updated. It can be used to build other intermediate forms such as syntax trees. This paper discusses some techniques for building and optimizing the shift/reduce sequence. One such technique has been applied by the authors to an incremental ANSI C compiler.

Background

The parser is both the best understood and the most central facility of a compiler. The driving loop of the compiler is in the parser—it calls the scanner and drives the generator. For the purposes of this paper, a top-down parser is a set of mutually recursive routines. A bottom-up parser is a table-driven stack automaton. Both examine the input text and report its structure. There are several relatively standard and low-cost ways of building parsers that are error free and efficient [1, 4].

In the process of attempting to reuse a particular C front end, the authors found that the output of the existing parser would not serve the intended new purpose. One alternative was to modify the parser to make a different form of output that would work. Another was to build a new parser. Neither alternative provided reuse—in the end there would be two artifacts to maintain. When we in fact built a new front end, we built output from the new parser that would have been sufficient for both previous uses. It seems as if this result is more generally reusable, and therefore is documented here.

Technical Basis

The use of a context-free grammar (CFG) to describe the phrase structure of programming languages is nearly universal [1]. Nonterminal symbols represent the main structures of the language. Terminal symbols represent the punctuation (operators, reserved words, etc.) and words (identifiers, constants, strings, etc.) of the language. The implementation representation of a terminal symbol is called a token. The effect of the scanner is to produce a sequence of tokens.

For C the output of the scanner is preprocessor tokens which are then input to the preprocessor which produces tokens for the parser. Providing preprocessing is irrelevant for the purposes of this paper. Think of the preprocessor as part of the scanner.

The parser examines the token stream and discovers and reports the phrase structure. The two major contending technologies for implementing parsers are bottom-up and top-down (BU and TD).

A BU parser such as *yacc* is typically a shift/reduce automaton. There is a parse stack which is initially empty. Each shift action takes one terminal symbol from the input and pushes it on the parse stack. Each reduce action applies a rule from the CFG to the top of the parse stack. The right-hand-side of the rule consists of a sequence of n terminal and nonterminal symbols; the top n symbols of the parse stack must match the right-hand-side. The reduce action pops all n symbols and pushes the left-hand-side of the CFG rule. Parsing terminates when the input is exhausted and the parse stack contains exactly one symbol—the so-called goal symbol of the CFG.

A TD parser is a set of recursive routines, each named for a nonterminal symbol and responsible for choosing and applying the CFG rules defining that nonterminal. The process begins when the routine for the goal symbol of the CFG is called. The process 'descends' through further routine calls and eventually returns to the original routine, which by returning, signifies that the parse is complete. There is no explicit parse stack, but the same sequence of shift and reduce actions implicitly takes place in the call stack.

The effect of the scanner and parser together is completely described by an interleaved sequence of shift and reduce actions. Suppose the rest of the compiler, represented by a module called the generator, receives only the shift and reduce actions via entries:

`Shift(t)` – report shifting token t to the generator
`Reduce(r)` – report applying rule r to the generator

The token is reported as soon as it is accepted and the rule is reported as soon as it is applied. This interface can be satisfied by either a BU or a TD parser.

The generator then has enough information to implement any kind of translation desired without referring to private data in the parser. In particular, the parse tree itself can be constructed, which contains all of the information about the original program (a left-to-right sweep of the leaves of the tree) as

well as complete information on the application of the CFG. This paper proposes the restriction of the post-parser interface to just the two routines, `Shift` and `Reduce`, together with access functions for the abstractions that deal with the rules and tokens passed by these two routines to the generator. It is our experience that any differences in compiler performance caused by following the structure recommended here are slight.

Tradition

The traditional BU and TD parsers each differ from the proposed solution in the manner in which they provide storage for the generator. The parse stack, an internal artifact of the BU parser, is a convenient structure to elaborate and exploit to save intermediate generator information [2]. The call stack of the recursive routines in a TD parser provides local variables, which are the corresponding place to save intermediate information. If either traditional storage technique is used, the parsers are incompatible. The alternative is to provide a general state saving mechanism in the generator, replacing the traditional use by the generator of the BU parse stack or TD call stack.

The description of the technique of parser-independent compilation requires some detailed discussion of functions provided across the compiler interfaces. There are many ways such interfaces can be defined, and many names by which the functions can be called. The interface presented here is picked to make the presentation easy to read. There is no implication that either the names or the specific choice of functions is optimal.

It is sometimes necessary to go beyond strictly grammatical means of constructing a parse. The variety of such ad-hoc solutions (backtracking for resolving ambiguity, feedback from declarations to the scanner for `typedef`, etc.) is beyond the scope of this paper. One can observe that the more regular the parser, the easier it is to make ad-hoc modifications.

The Scan/Parse Interface

Each compiler has a scanner that delivers up the source text of the program as a sequence of tokens. As a practical matter, the scanner needs three entries. Suppose they have names as follows:

```
Scan()                – steps ahead in the input
t = CurrentToken()    – provides the current token
t = LookAheadToken()  – provides the lookahead token
```

It happens that each call of `Scan` is always followed by a call of `CurrentToken`. (Otherwise, why bother to step ahead?) There is no reason not to implement an entry into the scanner combining `Scan` and `CurrentToken`, but doing so does not simplify this presentation.

Each token carries some required information: a lexical code (a small integer identifying which terminal symbol it represents), a textual representation (a character string), and perhaps also some other less often used information, such as the line and column in the input where the token begins. Access to the information is provided by routines acting on the value t of function `CurrentToken`:

```
c = LexCodeOf(t)   – small integer identifying token t
v = TextOf(t)      – string representation of token t
f = FileOf(t)      – name of source file containing token t
n = LineOf(t)      – source file line for start of token t
n = ColOf(t)       – source file column for start of token t
```

and so on. The value t itself is a unique representation for the token upon which no operations are allowed except assignment and those supplied by the scanner. The frequency of use of function `LexCodeOf` is high, indicating that its implementation should be particularly efficient. Both `LexCodeOf` and `CurrentToken` may in fact be macros and/or use hidden local variables to improve performance.

The parser calls through these entries to the scanner. Excepting nonstandard situations (such as caused by C's `typedef`), the parsing decisions require only `LexCodeOf(t)` for each token `t`. A TD parser has numerous calls to `Scan` and `CurrentToken` scattered over a number of recursive procedures. A BU parser needs just one call to `Scan` and also just one call to `CurrentToken` to implement the read-state processing of the automaton it implements. In both cases the parser may be unable to make some decisions without looking ahead. As before, a TD parser may have many scattered calls to `LookAheadToken` where a BU parser has exactly one call to `LookAheadToken` to implement the reduce-state processing of the automaton. The important point is that the scan/parse interface is the same for both TD and BU parsers.

The Parse/Generate Interface

The principal action of a parser is the application of a CFG rule to reduce the input. A sequence of such actions constitutes the canonical parse.

A BU parser naturally calls `Reduce(r)` when each rule r is applied. The proposed TD parsers must do exactly the same. It is not difficult to arrange for the TD parser to cause the same sequence of `Reduce` calls that the BU parser causes. The resulting recursive routines are more regular since all non-parsing detail is removed from them. The details of the rules may be built into the generator or may be available through a grammar abstraction. For example, the generator may have access to functions in addition to those accessing tokens, to simplify the process of interpreting the rules. For example:

```
c = RuleCodeOf(r)   – small integer identifying rule r
s = LhsOf(r)        – left side of rule r
n = LengthOf(r)     – number of symbols on right side of rule r
s = RhsOf(r, i)     – ith symbol on right side
```

The generator uses the rule and token information to build the intermediate representation of the program. The intermediate representation is typically some form of prefix notation, linear pseudo-code, or abstract syntax tree.

The traditional path for the token is from the scanner through the parser to the generator. For traditional TD compilers, a call to `Shift` immediately precedes each call to `Scan` because that is the moment of acceptance.

```
Shift(CurrentToken());  – send token along
Scan();                 – discard token
```

For traditional BU compilers the tokens are already in the (private) parse stack. Rather than send the token to the generator by calling `Shift`, the information may be kept in the parse stack and delivered up to the generator on demand (a pull by the generator from the parser, instead of push by the parser into the generator). Some generators contain private knowledge of the layout of the parse stack data structure. Others use a procedural interface to get at the information, keyed on the match of the top of the parse stack and the right-hand-side of the rule. For example:

```
t = ParseStack(2)
```

might retrieve the token positioned two below the top of the parse stack, and so on. This technique is *not* to be used with the parser structure proposed here. To match the activity of the TD parser, the proposed BU parser must also call `Shift` (so it too does a push into the generator).

To summarize, the traditional BU parser calls `Reduce`. The traditional TD parser calls `Shift`. They both use ad-hoc methods for communicating additional intermediate information to the generator (parse stack versus local variables in the call stack). The proposed BU and TD parsers must limit their interactions with the generator to calling only the two routines `Shift` and `Reduce`.

There are two convenient places for a parser to add the calls to `Shift`. Each call to `Scan` can be preceded by a call to `Shift`, as noted above. Or the scanner itself can call `Shift` immediately upon entry to `Scan`, just before updating the value of `CurrentToken`. It is obvious that the effect is the same. When it is difficult to modify the parser it may be necessary to have the scanner call `Shift`.

A Parse Tree Generator

The parse tree is too voluminous for practical use as an intermediate language. However, the following example shows how a very general generator is constructed using the two-routine interface described above.

Suppose that the only effect of `Shift(t)` is to stack the token t on a local stack in the generator. And suppose the only effect of `Reduce(r)` is to build an n-ary node (where n is the length of rule r), pop n things off the generator stack, place them in the the node, and push the node on the generator stack. At the end of parsing, the generator stack will have a single entry—the root of the parse tree itself.

The important point is that the parse tree is built without reference to any information saved in the parser. This shows the sufficiency of the two-function interface.

Filtering

Some tokens and some rules have no semantic significance. That is, they result in no action in the rest of the compiler. While it can be said that tokens carrying semantic information, such as identifiers and constants, and rules corresponding to semantic actions, such as arithmetic and branching, are surely significant, there is no corresponding concept of 'surely insignificant.' Only the language implementor knows for sure.

Without loss of generality one can say that the compiler filters the sequence of tokens and rules, discarding insignificant items. The filter may be placed on either the sending or receiving end, much as the call to `Shift` is placed before or within the call to `Scan`. At the receiving end, the generator may provide filtering by ignoring `Shift` and `Reduce` when insignificant information arrives. This is in fact how things end up if nothing special is done.

Another way to filter is for the implementor of the compiler to tabulate the significant tokens and rules so that `Shift` and `Reduce` omit sending the insignificant items to the generator. It is slightly more efficient to eliminate them at the source rather than ignore them later at the destination. The augmented interface to the generator becomes:

```
if (SignificantToken(t)) Shift(t)
if (SignificantRule(r))  Reduce(r)
```

For TD parsers, the test on r above is often computable at the time the parser itself is compiled. If filtering is on the sending end, the receiving generator needs to compensate by not looking for the missing information.

An Abstract Syntax Tree Generator

Using the filtered sequences, one can build an abstract syntax tree generator analogous to the parse tree generator described above. Because only significant tokens get stacked, the size of the n-ary node is reduced to the number of significant tokens and non-terminals in the rule. Nodes are built when rules are reported—and therefore fewer nodes are built for the abstract syntax tree because insignificant rules are not reported. At the end of parsing, the generator's parse stack has a single entry—the root of the abstract syntax tree itself. If the compiler requires a different, or more elaborate, intermediate language, all of the building activity can be isolated in the generator. This frees the generator from conforming to structures of the parser, and leaves the parser function and implementation unchanged.

Syntax Error Behavior

In addition to correctly parsing correct programs, the parser must respond constructively to syntax errors. There are two issues: how useful is the diagnostic message and what happens after the error is detected? The proposed solution makes detection, diagnosis and continuation private to the parser. The parser is responsible for terminating the compilation, or alternatively guaranteeing that the reported `Shift` and `Reduce` values are consistent with *some* correct program. The point is that the rest of the compiler is spared the extra engineering required to deal with invalid input from the scanner/parser.

Recovery from syntax errors is simpler with BU parsers because the entire state of the parse is available for manipulation by the error routine. In TD parsers, the state is wrapped up in the call stack. Typically TD parsers written in C resort to the `setjmp/longjmp` functions as a relatively clumsy way to control the contents of the call stack. Recovery can also be better with BU parsers because there is a well-developed technology for gathering right-context (the so-called forward move algorithm [3]).

Scanning and parsing diagnostics are inherently limited to the model of "an X was seen in the context of trying to do Y, and only one of $Z1$, $Z2$, or $Z3$ is acceptable." Diagnostics that go beyond this formula are guessing what might have been intended. Such guesses are often helpful, but also often misleading. We prefer to stick to the known facts. In addition to stating what happened, the diagnostic should provide the location of the offending X. During scanning, the current file, line, and column are apparent to the scanner. During parsing, the token abstraction carries the necessary information so that the parser can report the location to the diagnostic mechanisms (recall functions `LineOf(t)`, etc.).

If a parsing error occurs at a token that resulted from macro expansion, the reasonable location to report is the outer macro invocation. The compiler can, in addition, list the stack of active macros, although it takes some preplanning beyond just the information available in tokens.

Later in the compilation process, the diagnostics report inconsistencies between two sources of information (for example, declaration and use). The tokens in the shift/reduce sequence provide the basic signposts upon which to establish the locations, although it can happen that an otherwise insignificant, and therefore filtered, token is significant to the diagnostic process. The filter must then pass it.

Conclusion

There are many reasons behind choosing a parsing technique. The point of this note is not to make the choice, but rather to remove one set of reasons often cited for making the choice. The proposed solution rules out any criterion based on the rest of the compiler since the rest of the compiler is independent of the choice. The proposed solution is also of comparable efficiency to traditional solutions. In any case, the authors believe parsing cost is small compared to the rest of compilation.

References

[1] Aho, Sethi, and Ullman, "Compilers – Principles, Techniques and Tools," Addison-Wesley (1985).

[2] McKeeman, Horning, and Wortman, "A Compiler Generator," Prentice-Hall (1970).

[3] Pennello and DeRemer, "A forward-move algorithm for LR error recovery," Fifth Annual ACM Symposium on Principles of Programming Languages (POPL), pp 241–254.

[4] Nicklaus Wirth, "Algorithms + Data Structures = Programs," Prentice-Hall (1976).

William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. He can be reached at mckeeman@tle.dec.com.

Shota Aki is a Principal Software Engineer at Digital. He was a principal developer of the VAX APL interpreter. His current interests are in the areas of CASE, Analysis and Design Methods, Compilers and Tools. He can be reached at aki@tle.dec.com.

Scot Aurenz is a Principal Software Engineer at Digital and is currently working on the Language Sensitive Editor project. He can be reached at aurenz@tle.dec.com.

∞

22. Electronic Survey Number 6

Compiled by **Rex Jaeschke**

Introduction

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

The following questions were posed to 90 different people, with 21 of them responding. Since some vendors support more than one implementation, the totals in some categories may exceed the number of respondents. Also, some respondents did not answer all questions, or deemed them 'not applicable.' I have attempted to eliminate redundancy in the answers by grouping like responses. Some of the more interesting or different comments have been retained.

Trigraph Recognition

Given that trigraph processing can slow down compilation do you (plan to) have a compiler option to disable trigraph recognition? (Perhaps you provide a separate utility to convert to/from trigraphs.)

- 5 – Can disable recognition
- 11 – Cannot disable recognition
- Comments:

    1. Not at the moment. The trigraph processing doesn't seem to be a particularly expensive part of our compilation time.
    2. Turning off trigraph processing based on an option was considered, but since we don't look for trigraphs inside comments, and comments represent a significant portion of the source we are scanning, why complicate the testing procedures?
    3. A separate utility is inappropriate. A compiler should be as strictly-conforming as it can be by default, and trigraphs are, like it or not, part of the standard. Allowing the user optionally to turn off trigraphs is fine, but making him manually run a separate pass to enable them is not. Actually, I'm not convinced that trigraph processing is expensive.

4. The incremental speedup masks the extra level of function invocation in my C compiler so I have no intention of doing anything special for trigraphs. It is my opinion that a decently written front-end will be entirely speed-limited by a decently written back-end anyway.
5. In implementing trigraphs I found that I indeed had to take a second pass in my preprocessor (which previously required only one pass). However, writing the trigraph scan in assembly yielding no discernable difference in speed. This coupled with the massive confusion associated by having too many options for a compiler led me to decide against allowing the user to disable the trigraph scan.
6. You have to get rid of *both* trigraphs and arbitrary backslash/new-line splicing before things can be made faster.
7. We found that trigraphs were dangerous and that they broke existing code containing `printf`s with '??' in the string.
8. We have an option to disable trigraphs but it's not for speed, since we get only a 1–3% speedup. We do it because trigraphs are an abomination!
9. The requisite trigraph processing should not slow down compilation appreciably since '?' source characters are quite rare, most of a source file should simply have an additional CMPB and BEQ (not taking the branch) per source character. The average processing per source character generally far exceeds that small added amount.
10. A special option is needed to enable trigraphs, since I don't expect sensible people to use them.
11. We do not have any plans to provide options that turn off trigraph processing. However, it is certainly interesting what trigraph processing can do to non-C language input.
12. We provide a separate filter program to support recognition of trigraphs. IBM PC users have the full ANSI C character set available and so we decided not to burden PC users with the slow-down needed to support trigraphs and a few of the more strict scanning rules of ANSI C (like allowing backslash/new-line combinations to appear within a token other than a string literal).
13. We convert trigraphs on the first pass through the source and read the converted text on the subsequent pass. The trigraph recognition is awkward, but since trigraphs are converted immediately, they are not a major time expense.

Is `main` Special?

Does your compiler recognize `main` *as a 'special' function? (Perhaps you do extra checking or generate different code.) Have you ever found a need to call* `main` *recursively?*

- 3 – `main` is special
- 10 – `main` is not special
- Comments:

    1. I've never called `main` recursively but I've seen programs that have done so.
    2. When `main` is recognized, the compiler emits a special global which will cause the linker to pull in the program startup and rundown code. In addition, this global is different depending upon whether or not arguments are given to `main`, so that if no arguments are present, the code to handle command-line processing is not linked into the program.
    3. `main` is special for us in that our debugger needs additional information to handle it.
    4. In many ROMable applications the main entry routine need not save or restore any registers. However, having a special feature to remove the few instructions that could be saved here results in very little savings since the treatment is usually limited to one function in the entire program.
    5. Our MS-DOS version does because it has to emit certain segment-ordering information. Our UNIX versions don't.
    6. `main` itself is not special. However, from the loader's perspective, the library startup code is since it is designated as the primary entry point of the code. We have a `#pragma main` *name* which designates the function *name* to be the primary entry point. The function `_crt0` (main startup code function in our *libc*) uses this pragma. Users can designate any other function as the main entry point if they so desire.

Extended Integer Types

At the most recent Numerical C Extensions Group meeting a subgroup was formed to consider extended integer types. Prior art in this area comes from one 64-bit machine where the type `long long int` *has been added. What is your opinion on* `long long` *as a solution? Do you have an interest (or prior art) in a more general solution such as* `int16`, `int32`, `int64`, *etc?*

- Comments:

    1. The `long long int` type causes a number of difficulties but can be implemented easily and usefully. I think a more general extension such as allowing subranges and specific size specifications would make more sense though.

2. `long long` shows how stubborn the C community is in rejecting innovation from other languages. Our interest is in a more general solution. We have an extension to C syntax for integer range types, which lets the user say things like `typedef |-32768..32767| int16;`. Not only is this a cleaner syntax than `short`, `long`, `long long`, etc., but it also is more portable and offers greater opportunities for error checking.
3. I have a huge dislike of the more general `int16`, etc., solution. This is just not portable to all architectures and encodes too much information in the type. `long long` seems a bit kludgy to me. Is there really a need for more than four different integer sizes, even on a 64-bit architecture?
4. We have no prior art in this area; however we are very keen that some solution to the large file size problem should become 'standard' soon. We prefer that the large file size problem should be solved be changing C to allow a larger type (rather than by providing new `lseek`, etc., functions or by making `long int` 64 bits wide). `long long int` is adequate and likely to become accepted more quickly than the FORTRAN type syntax.
5. Our machine has 64 bits per word. If `long long` were added it would more likely be a 128-bit integer for us. We may even have a use for it! I'm not sure `int16`, etc., is a good idea, but I have no better ideas.
6. I think `long long int` is in the 'spirit of C.' I know some people desperately want bit-count-level control, but I've never needed it. I dislike `int32`, etc., partly because I've never needed them and partly because I don't see how many of them there should be or what a compiler should do if it can't provide the size requested. Also, they're ugly and look more like PL/I than C.
7. We have received several requests from our users for a 64-bit integer even thought ours is not a 64-bit machine. We plan to implement such a type in the future.
8. I prefer a header with `int`*NN* defined for every reasonable *NN* in terms of the base types supplied by the compiler. I think programmers should avoid cute optimizations on word length to save storage because the cost is broken programs when something gets bigger or longer than they ever imagined. Just use `int` unless there is overriding reason.
9. `long long` makes the most sense. To have specificational types such as `int32`, etc., would mean a major change in the language specification and rightly belongs in a new language.
10. `long long` is a workable concession to nonportable programs. But it makes the clear statement that there was no good choice. Of

course adding more keywords is not any better. Using more and more such combinations will only make things much worse. Some form of extensible specification would be better. For the meantime, I would think that having `long` be your wider-than-32-bit integer and `int` be 32 bits is a reasonable alternative.

11. `long long` seems OK to me. When 128 comes around we'll rethink, but that will be 20 years.
12. No real opinions here although I can't help commenting that people with 36-bit machines might want `int18` and `int36` as well.
13. We have no pressing need for extended types.
14. I cannot support the use of reserved identifiers as additional keywords. `long long` has the advantage that it is a conforming extension. However, I see no real need to access all possible integer sizes.
15. We already support `long long`.
16. I do not believe that `long long int` guarantees a 64-bit integral type. The best that could be guaranteed is at least 64 bits. Currently, there are four different precisions possible for integral types. This allows 8-bit `char`, 16-bit `short`, 32-bit `int`, and 64-bit `long`. I am against this proposal.
17. My preference is to introduce the concept of integer ranges as found in Pascal. Extremely large integer types can be implemented as `typedef`s using large integer ranges. I think that Pascal style ranges afford better compile-time analysis than the `int32` and `int64` variants found in FORTRAN.
18. In our implementation, `short`, `int`, and `long` in essence all use 64 bits. Therefore, `long long int` should not be used to designate a 64 bit object.

Inter-Language Issues

Which other languages does/must your C code interface with? What are the main problems in establishing correct and efficient inter-language communication? How have you solved them?

- Comments:

1. The worst problem is memory management—interfacing to modern languages that need garbage collection.
2. Pascal, FORTRAN, and C++. The onus of establishing the communication is mostly on the C compiler/programmer. Our primary compiler has new keywords to allow the programmer to specify the

calling convention he wishes. The programmer still has to make sure that the names will agree.

3. FORTRAN and Ada. We let the user figure it out, of course.
4. BASIC, FORTRAN, Pascal, and assembly. We provide

   `#pragma linkage` *language module-list_opt*

   to allow the programmer to indicate a given module should be called with a particular linkage.
5. FORTRAN and assembly. The main problems are with byte/word and by-value/by-reference differences. Case sensitivity between C and FORTRAN is another issue. Our binary format contains case sensitive names and the FORTRAN compiler generates, by default, uppercase names Our C compiler passes byte addresses while FORTRAN passes word addresses. To resolve this we have the pragma

   `#pragma XXX language(` *language*`,` *name* `)`

   where XXX is our company's initials and *language* is either `C` or `FORTRAN`. This directive can be used on declarations and definitions.
6. I've never thought it entirely fair to discuss this as a C issue (though, of course, everybody does); it's as much an issue of the other languages with which communication is desired. It comes up so much with C simply because C's low-level power and popularity make it a natural choice for writing subroutine libraries, device drivers, and 'glue' logic, all of which have a much better than average chance of needing to be called from (or make calls to) heterogeneous languages.

   The only reasonable, general solution is an architecture-level procedure call standard. DEC designed an excellent one for the VAX. (It is frequently, and unfairly, badmouthed though because its generality was institutionalized in the VAX CALLS and CALLG instructions which are more powerful but allegedly less efficient than many people would like them to be.)
7. Pascal and, to a lesser degree, FORTRAN. We have an established Procedure Calling Convention that all programming languages follow. As long as data types in parameters are compatible, there is not much of a problem.
8. FORTRAN, Ada, Pascal, C++, and other C compilers too. Its a mess!
9. Assembly. I use a pragma to specify the registers in which to pass arguments.
10. C++ and FORTRAN. C++ takes care of its own C interfaces, while C does the work needed to map to FORTRAN's data types. Multidimensional arrays passed between FORTRAN and C do require some thought though on the part of the programmer.

11. FORTRAN, Pascal, and C++. We have pragmas to change calling convention and to specify an object-module name of a function as opposed to the source name. We also have pragmas to exploit FORTRAN COMMON blocks.
12. FORTRAN and Pascal. With FORTRAN, there is a big problem on the user education side concerning the way in which FORTRAN views parameters. The other major problem is concerned with runtime library initialization: in effect, one language has to be 'in charge' in a program, and the I/O statements of the other languages cannot be used.
13. Ada. The main problem here is in passing an exception handler context. We have not come up with a satisfactory solution to this. As for Ada calling C, this is not a particularly hard thing to do.
14. F77 and Occam. The function calling convention is different, so special keywords (such as `__fortran` and `__occam`) will be needed. And the linker will need to be smarter.
15. FORTRAN, Pascal, and Ada (however, for Ada the main entry point must be in Ada). The biggest problems are between C and FORTRAN. Array storage (column/row major), call-by-reference/value, TRUE/FALSE, character pointers versus character descriptor, variable dimensioned arrays, and complex arithmetic.

    When applied to functions, our `fortran` keyword implicitly takes the address of any argument that does not have a pointer type. `fortran.h` contains several helpful macros that convert a FORTRAN LOGICAL into a 0 or 1, convert a C integer into a FORTRAN LOGICAL, convert a C character pointer and integral length into a FORTRAN character descriptor, convert a FORTRAN character descriptor into a C character pointer an a length (with unsigned type). We have extended our C implementation to include variable length arrays and complex types.

Intrinsic Library Functions

Have you or are you planning on implementing the FORTRAN 'Intrinsic Function' idea by making certain C functions behave more like operators than function calls? If so how does the user select this feature or is it the default?

- 14 – Yes
- 4 – Transparent
- 9 – User selectable
- 3 – No

- Comments:

  1. It is under user control via command-line arguments and pragmas.
  2. The implementation is controlled by macros in `math.h` and is triggered by macro definitions on the command-line.
  3. All are available to the C programmer by declaring a full prototype for the intrinsic and using the following pragma:

     ```
     #pragma XXX intrinsic ( name )
     ```

     where XXX is the company initials and *name* is the name of an intrinsic function known by the compiler. Inside `math.h` many of the math functions (such as `sin` and `cos`) have this pragma declared for them. No intrinsic is selected unless this directive is seen.
  4. I assume you mean 'function inlining.' `pow` immediately comes to mind and (architecture permitting) `sqrt`, `sin`, etc. The method suggested in ANSI X3.159-1989, §4.1.6, page 100, footnote 96 suggests a reasonable approach; in `math.h`, have something like:

     ```
     extern double pow(double, double);
     #define pow(x, y) _BUILTIN_pow(x, y)
     ```

     and then have the code generator special-case calls to `_BUILTIN_pow`. This behavior would be the default, but could of course be disabled by the user by calling `(pow)(x, y)` or with `#undef`.
  5. We are looking into inlining some popular functions for performance reasons, e.g., `strcpy`.
  6. Within the standard, I feel free to do as I like here. The worst problems are for the debugger which has to wend in and around lots of special code. In any case, I won't do much of any of this in my compiler since code speed is not the issue in an incremental environment.
  7. Standard C already allows intrinsic functions in that a hosted implementation can know the semantics of a standard library function once its header is included. If you mean the type-conformable macro-like FORTRAN functions, they would be a bad idea in C. (Function-like macros are already available.) A much better approach in C is to extend the language to allow function overloading. This covers a part of the need. The other difference between operators and functions has to do with computational exceptions. This should be handled in some fashion for existing functions first. Once that is done, it should be available for the new overloaded functions.
  8. The compiler recognizes lots of library functions since we [the implementors] 'own' them.

9. Certain functions are known by our compiler. Most are specific to our target environment, but some (like `memcpy` and `strcpy`) are standard C functions. These are known *even* if the programmer forgets to `#include` their parent header or puts in his own external declaration for these functions.
10. We do this with `abs`, `labs`, and `fabs`. Also, most of the `is`*, `str`*, and `mem`* functions.
11. The transcendental math functions are recognized and converted into calls to special functions that accept their arguments in registers, and have a fast entry and exit sequence.
12. We support a collection of intrinsic functions, but they all have reserved names beginning with two underbars. For example, there is an `__abs__` function intrinsic to the compiler. The `stdlib.h` header defines an `abs` macro that maps to `__abs__`. This fits the ANSI rules for overloading of standard library functions and a simple `#undef` will selectively disable intrinsics that are not desired for whatever reason. Also, the rules for expanding function-like macros allow one to still take the address of the real library function even when the intrinsic is available for direct calls.

∞

23. Case study: Building an ANSI CPP

John H. Parks
Compass, Inc.
550 Edgewater Drive
Wakefield, MA 01880

Abstract

Designing a production quality ANSI C preprocessor was an unexpectedly complex task. Since K&R (1st Edition) devoted but two pages to its initial description, one might anticipate that building one would be a relatively simple matter. It was not, and the ANSI extensions were just part of the story.

In this paper, I define our rather lofty goals and discuss our approach to lexical processing and the processing of macros, `#include` directives, and conditional inclusion. I also discuss the extensions we considered and actually added.

Introduction

We began the project by attempting to define an *ideal* C preprocessor. After considerable study of the ANSI documents and the preprocessors in the field, we concluded that an ideal preprocessor should exhibit the following characteristics:

- Well defined functionality
- Fast
- Minimal capacity constraints
- Integrated but separable
- Sensible error handling
- Flexible/portable

One of the largest and most hazardous areas of diversity among C compilers is in the preprocessor. Implementation-specific features abound, and efforts to document preprocessor functionality have been notoriously insufficient. Quite frequently, the only reliable way to determine a preprocessor's treatment of a

certain issue has been through testing. This is undesirable. An ideal preprocessor should accept a well defined (if primitive) language and produce well defined output.

An ideal preprocessor should also be as fast as possible and have few capacity constraints. "Too many formals," "Actuals too long," "Expansion size exceeded," and other such diagnostics are a common source of irritation for C programmers developing large software systems, and at times prove to be a constraint on development. They needn't be. Preprocessing is not inherently limited by these things, and a well designed preprocessor should be both fast and capacious.

An ideal preprocessor should also be able to generate an explicit output file. This does not imply that the preprocessor should generate an output file as a means of passing its output to the compiler proper. This common technique has severe performance consequences for the compiler since it requires additional I/O and a second lexical parse of the input. Instead, it simply means that the preprocessor should provide the ability to generate such a file when requested.

The ability to generate explicit output files can be extremely useful in debugging and development, especially when macros and header files are used extensively. It can also allow the preprocessor to be used as a general purpose tool. Any text file within the broad lexical requirements of the preprocessor can be effectively preprocessed.

Another characteristic of an ideal preprocessor is that it handle errors in a sensible and systematic way. The preprocessor should never crash, no matter what the input. Error messages should be descriptive and highlight the offending sections of input. And the preprocessor should be error correcting wherever possible. After generating the appropriate diagnostics, the preprocessor should "assume" a missing parenthesis or comma, complete an incomplete header name, and so forth. This will yield more complete diagnostic information and more useful output to the user.

Finally, as with any development tool, an ideal preprocessor should be flexible and portable. It should provide switches for commonly found preprocessor features and allow selective warning message suppression. In sum, it should be usable on as many systems as possible.

General Description

The Compass C preprocessor is a token based, ANSI-conforming preprocessor. Because performance is important, it is integrated into the C compiler and passes tokens directly to the parser, eliminating the additional I/O and lexical analysis necessary if an output file were first generated. Macro processing is optimized, employing sophisticated algorithms and data structures for macro definition and expansion to avoid copying and rescanning of text. File inclusion is also optimized, with special treatment given to guarded header files.

High capacity is a requirement, so the preprocessor uses no fixed-size data

structures. All tables, stacks, and lists are dynamically expandable. Capacity limits are set by process memory limits, not by program constants. Though integrated into the compiler, the preprocessor can produce an explicit output file for debugging or development purposes. It can also be built with a separate driver to produce a general purpose translation tool with well defined input and output.

The preprocessor has an extensive and flexible set of error handling facilities, and is fully error correcting. Certain diagnostic messages may be selectively suppressed. Moreover, the preprocessor continues processing the input no matter what the error, making reasonable assumptions about the source writer's intent when an error is diagnosed.

Finally, because portability and flexibility are design requirements, the preprocessor contains a full set of switches to allow the user to turn on or off the more novel and controversial ANSI features, to define the include file search path algorithms, to describe the structure and form of the explicit output file, to enable non-ANSI but popular features found in other preprocessors, and so forth. The preprocessor also allows the user to define certain "implementation defined" and "undefined" issues from the ANSI Standard.

Lexical Processing

Translation Phases

To clarify the nature of syntactic analysis, the ANSI Standard defines a conceptual model for the translation of source text into tokens. This model is specified as a sequence of eight distinct translation phases, and defines the order of precedence among the processing rules. The first four phases are primarily concerned with the actions of the preprocessor. They may be summarized as follows:

1. Map the physical source file to the source character set and interpret trigraphs.

2. Delete each sequence of backslash followed by new-line, splicing physical source lines into logical lines.

3. Recognize preprocessing tokens, white space, and new-lines; convert each comment into a single space character.

4. Execute preprocessing directives, expand macros, and process `#include` directives by recursively processing the named file through these four phases.

The final four phases are concerned with translating the stream of preprocessing tokens emitted by the preprocessor into actual tokens and then eventually into a program image.

Trigraphs

The ISO-646 Invariant Code Set, an internationally agreed upon character set, is not rich enough to express the C language in full. Nine essential characters are absent. The ANSI committee introduced trigraph sequences as alternate spellings for the nine missing characters to allow the implementation of C in all ISO-646 conforming character sets.

As it is defined as a separate pass over the source text, trigraph processing can be costly. There are two obvious implementation choices: create a filter to screen trigraphs before source text is fed to the lexer, or incorporate trigraph processing directly into the lexer/preprocessor. The first choice is undoubtedly expensive. It requires that every character of the source input be read at least one more time, and complicates the task of pointing to the offending section of code when errors are found. The second choice has neither of these problems. We chose the second.

Sophisticated Lexer

We combine the first three translation phases (and the three scans of the stream input that they imply) into one sophisticated lexical analysis phase. This phase inputs source text and outputs a stream of preprocessing tokens, each containing source position information and a bit describing whether or not white space appeared before it in the source text.

This is accomplished with a standard finite-state machine, generated from a regular expression grammar like that given in the ANSI Standard, but with the following modifications:

1. Trigraphs are included as alternate representations for the affected characters. For example:

```
TOKEN_LBRACKET      ←—   '[' | '??('
```

2. Fixed multi-character tokens allow arbitrary sequences of backslash followed by new-line. This includes, of course, the trigraph representation of backslash. For example:

```
BACKSLASH           ←—   '\' | '??/'
BNL                 ←—   BACKSLASH NEWLINE
TOKEN_EQUAL_EQUAL   ←—   '=' BNL* '='
```

3. Variable length, multi-character token classes are described by two different tokens, one for tokens that contain embedded backslash/new-line sequences and one for tokens that do not. So, for instance, there is a `TOKEN_IDENTIFIER` and a `TOKEN_IDENTIFIER_BNL`, a `TOKEN_STRING` and a `TOKEN_STRING_BNL`, and so forth.

When an identifier or numerical constant contains a backslash/new-line sequence (possibly of the trigraph backslash variety), the token's text is filtered before being entered into the text table. Though this requires additional work, it should be a relatively rare occurrence. In the normal case, where backslash/new-lines do not appear, no additional work is necessary. The text of string literals and character constants is always filtered for trigraphs and sometimes for backslash/new-lines as well. With trigraph processing enabled, this is unavoidable.

Disabling Trigraph Processing

Several preprocessors allow the user to completely disable trigraph translation, presumably to improve performance and avoid the syntactic changes that it entails. We do not. We allow the user to suppress trigraph processing only within string literals and character constants. We view this as a compatibility feature allowing the user to preprocess "old C." Since trigraphs may not legitimately appear in "old C" anywhere but within these tokens, and since recognizing them elsewhere costs us almost nothing, there seems to be no point in providing the ability to disable translation completely.

Comment Processing

Comments do pose something of a problem for the generated lexer. The regular expression grammar needed to recognize C comments is non-trivial even without the added complexity of trigraphs and backslash/new-lines because the closing delimiter must be excluded from the body of the comment. Allowing arbitrary sequences of backslash/new-lines to appear between the `*` and `/` of the closing delimiter complicates the grammar immensely, rendering it practically intractable.

We solve this problem by recognizing only the opening comment delimiter in the lexer. The preprocessor then recognizes the closing delimiter itself. This is relatively simple since it is looking for a pattern to terminate the comment rather than trying to find the longest match to a pattern that matches everything except the terminating sequence. This solution has the advantage of allowing the preprocessor to write the text of comments to an explicit output file very easily at the user's request. The preprocessor simply writes out the comment text while scanning for the terminating sequence. Were the full comment recognized by the lexer, its text (of unbounded, and often very large size) would need to be buffered by the lexer and then rescanned to write it to an output file.

Macro Processing

Macro processing involves macro definition and macro replacement. Macro definition is computationally simple. There are interesting optimization issues to

resolve involving how macro definitions should be stored and how redefinitions should be examined, but the work is not complex. Macro replacement is another matter. It may be quite complex, confusing, and counter intuitive. Macros may nest within other macros. Macro calls may be built "on the fly" using formal parameter substitution. Macro replacement, or token pasting, and some limited forms of recursion are allowed.

Designing a fast macro processor that scans each token as few times as possible, that does minimal list shuffling, and that is bound by minimal capacity constraints is a non-trivial task.

ANSI Macro Replacement

Though macro replacement may be quite complex, its description is straightforward. It is logically a three step process:

1. Prescan (function like macros only) - The actual parameters are macro expanded. This may involve expanding the macro containing the actual parameter, and may involve many levels of macro nesting. When this phase is complete, each actual parameter will have been transformed into a replacement list containing no valid calls to other macros.

2. Substitution (function like macros only) - The replacement lists formed above are substituted into the macro body in place of their associated formal parameters. If the formal parameter is to be stringized or token pasted, the unexpanded replacement list is used.

3. Postscan - The macro body is scanned for stringizing operations to perform, token pasting operations to perform, and embedded macro calls to replace. As in the prescan phase, this may involve many levels of macro nesting, but it may not involve recursively expanding the macro corresponding to this body.

The ANSI Specification elaborates on the avoidance of recursive macro expansion by stating that once a macro name is considered for expansion and rejected because of recursion, it is marked to prevent it from expanding if it later occurs in a context in which it would normally expand. The following example shows how such a "later context" can occur:

```
#define f(x) x
f(f)(1)
```

When fully preprocessed, this should produce `f(1)`. It cannot expand further because `f` is "marked" to prevent expansion.

Expansion Algorithm

A straightforward implementation of the above would first produce and store the replacement list for every actual parameter that needs expansion (Pres-

can). It would then scan the macro body and substitute these replacement lists where appropriate (Substitution). Finally, it would rescan the macro body to perform stringizing and token pasting operations and to expand macros that it encounters (Postscan).

This sketch describes an algorithm that requires two complete scans of the macro body, storage for the expanded forms of actual parameters, and storage for the fully expanded macro body. Recognizing these costs, we chose another approach.

Our approach expands macros in a single left-to-right pass over the macro definition. It builds no intermediate token lists except for unexpanded actual parameters, and it outputs each token as it is determined. The state of the expansion is preserved while control is returned to the parser so that expansion can resume the next time it is called. The costs of this solution are algorithmic and data structure complexity. It must accomplish in one pass what is specified in two, and must represent an essentially recursive expansion process in static data structures.

Expansion Data Structures

Three central data structures are used in macro processing: the token table (which contains a record for each token used in macro processing), the macro table (which contains a record for each presently defined macro), and the expansion stack (which contains a frame for each macro or actual parameter presently expanding).

The expansion algorithm builds a new stack frame for each macro it begins expanding. The frame contains pointers into the macro table and the list of unexpanded actual parameters for the macro call. As it expands the macro, it moves these pointers though the appropriate token lists to maintain its state. When a valid macro call is encountered within an expansion, the algorithm constructs another frame, pushes it onto the expansion stack, and continues processing with that frame. One frame situated above another on the expansion stack corresponds to an embedded macro expanding within another.

When a formal parameter that is not stringized or token pasted is encountered within a macro body, a similar thing occurs. A stack frame is built to macro expand the actual parameter. This frame is not pushed onto the stack, however, because the context of the expansion is slightly different. Instead, it is linked to the current frame with a pointer. One frame pointing to another in this way corresponds to a macro expanding actual parameter being substituted for a formal parameter.

Macro Definition Issues

When the preprocessor encounters a `#define` directive, it must first determine whether or not it is a pernicious macro redefinition and then enter its text into

the macro table. Macro definitions are stored as token lists, and so interesting opportunities for definition time token transformations present themselves.

We move token pasting operators in front of their left operands (transforming *token* `##` *token* into `##` *token token*) to remove the macro expansion burden of looking past each token to determine if it is token pasted. We also transform stringizing operations(`#` followed by a formal parameter) into single stringizing tokens. This has the nice consequence of distinguishing stringizing operators from ordinary `#` tokens. These are simple transformations triggered by specific tokens; they seem worthwhile to do while building the definition.

We do not mark formal parameters as such at definition time, however. Quite often in real code a substantial percentage of the macros defined in header files are never called within a particular compilation. Since finding the occurrences of formals within a macro body involves some type of lookup for each identifier, it makes sense to do this processing only for those macros that are actually called. We decided, then, to mark formals as such during the first expansion of each macro. Macros that are never called are then cheaper, and macros called more than once have no added burden.

Include Processing

The `#include` preprocessing directive allows the user to include header files in the preprocessor input. It has three variations: the system file variation, the user file variation, and the computed header name variation. In the latter, the argument tokens are macro expanded and combined to form one of the two other forms.

According to ANSI, including a file should produce the same results as copying the contents of that file into the original, with the following three exceptions: comments may not begin in one file and end in another, and backslash and new-line may not be the final two characters in any file (tokens cannot span file boundaries), and related conditional inclusion directives must all reside within one file.

Implementation

Because header files may nest one within another, we chose an obvious, stack based algorithm for include processing. There were just two sticky issues, header name determination and include search strategy. The lexical definitions for header names conflict with the lexical specifications for the C language proper. It is thus impossible to use a common grammar for both. This adds complexity to the lexical analysis process.

The other sticky issue was include file search strategy. The ANSI document does not elaborate on how or where the preprocessor should search for header files. This is left for the implementation to define. Assumptions about operating systems and file storage are also carefully avoided.

Given that extant preprocessors vary significantly from one another in this area, we felt that the best solution was to allow the user to define search strategies for both system and user header files from the command line. The user may define a set of places to search and an order in which to search them. This also involves defining whether or not the search should begin in the location of the original source file or the location of the (include) file presently being processed.

Guarded Header File Optimization

It is common in large software systems for a source file to include the same header file more than once. Because of this, many header files are *guarded* to prevent the full processing of these files more than once. This is traditionally done as follows:

```
#ifndef FILE_GUARD
#define FILE_GUARD

contents of header file

#endif  /* FILE_GUARD */
```

Notice that if this file is included a second time, the preprocessor will not process macro definitions or pass tokens on to the parser or explicit output file. It will, however, read in and process the file, at a non-trivial cost.

Noticing this cost, we designed the preprocessor to specifically recognize guarded header files. When it finds one, it saves the guard for future reference. Then, if the file is included again, the preprocessor can process the guard without having to input the file. If the guard test fails, the file is dismissed without having to be input. The savings can be quite considerable.

Conditional Inclusion

The preprocessor's conditional inclusion facility allows the user to include or exclude source code at compile time, an important ability when developing portable applications. ANSI has defined six preprocessing directives for this: `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`. They may nest one within another giving the user substantial flexibility.

Constant expressions in this context are somewhat interesting. All operands are either `long` or `unsigned long` and identifiers that are not macros are interpreted as `0L`. The grammar includes the new ANSI `defined` operator, but excludes the `sizeof` operator and casts.

Our implementation utilizes a stack based algorithm to control directive nesting and a recursive descent parser/evaluator to process constant expressions. Unfortunately, the grammar differs enough from the C constant expres-

sion grammar to make reuse of that evaluator difficult, and so the preprocessor has its own.

Extra ANSI Functionality Issues

When designing the Compass C preprocessor, one of our stated goals was that the preprocessor should be flexible enough to replace most popular C preprocessors compatibly. This design goal involved a study of existing preprocessors and the creation of a considerable number of user switches to enable non-ANSI features. Some features were as simple as allowing the `$` character to appear in identifiers, or escaping the `"` and `\` characters within the `__FILE__` macro expansion, or allowing extraneous tokens to appear after `#else` and `#endif` directives. Others were more involved. As described above, the specification of the `#include` search path algorithm was particularly interesting. The following issues were also sticky.

Token Concatenation

Designing a preprocessor to produce an explicit output file introduces a new measure of correctness. It seems reasonable to assert that a correct preprocessor should be consistent with itself to the extent that when it produces explicit output and that output is then fed back into the compiler, the result should be identical to that produced when no explicit output is generated. This seems like an appropriate objective, and it would not be unreasonable for a user to expect this, but the objective can be extremely difficult to obtain. Consider the following example:

```
#define f(x) hello
f(1)there
```

When preprocessed, this produces `hellothere`. ANSI specifies that this result is actually two tokens with no intervening white space. But if an output file were generated and then fed back into the preprocessor, just one token would result. This effect is not limited to identifiers, but can occur whenever macro expansion places concatenatable tokens adjacent to one another in preprocessor output.

One common solution is to insert a space character after every macro expansion written to preprocessor output. This would solve the problem in the above test case. This solution is incomplete, however, as evidenced by the following two examples:

```
#define f +
+f

#define g(a) a+
g(+)
```

Both examples will produce a single `++` token if explicit output is generated and then fed back into the compiler, even with the solution described above.

A better (yet more costly) solution to this problem would be for the preprocessor to determine whether every two immediately adjacent tokens are concatenatable, and then to insert a blank space between them if they are. An "adjacency matrix" could easily be defined, and the lookups should not be prohibitively expensive. Feeling that this feature will not be frequently enabled (it is only useful if an explicit output file is generated), we chose the second solution. Partial solutions seem hardly worth the price.

Macro Parameters in Strings and Character Constants

Replacing formal parameters within string literals is fairly common practice. It is semantically ugly, however. It is also strictly non-ANSI, and in fact was the motivation for the explicit stringizing operator (`#`). One major problem with existing practice in this area is that replacing formal parameters within strings is not really stringizing. `"` and `\` characters are not escaped during the replacement, and so the result may not be a single lexically valid string. Feeling that this practice was too common to ignore, we provide support for old style stringizing and charizing at the user's request. We also provide escaping within such constants when necessary.

It is worth adding that the old style stringizing switch has the important benefit of allowing the preprocessor to be used effectively with pre-ANSI compilers that do not provide adjacent string concatenation. Without such concatenation in the compiler proper, the following common macro cannot be effectively expressed in ANSI preprocessor syntax:

```
#define old_string(x) "string is: x"
```

Comment Token Pasting

In some preprocessor implementations, most notably the UNIX CPP, comments can be used within macro definitions to paste tokens together. Comments are not replaced by single space characters (as they are in ANSI-conforming implementations) and preprocessor output is written to an output file before being read in by the compiler proper. The effect of this can be seen in the following example:

```
#define paste(x,y) x/**/y
paste(field_,1)
```

When preprocessed by the preprocessor described above, this yields a single identifier token `field_1`. This facility is useful, but was deemed semantically ugly by the ANSI committee, and so it was replaced by the explicit token pasting (`##`) operator.

Unlike old style stringizing, old style token pasting is fairly easy to systematically detect and edit in C code. We felt, then, that providing support for this feature was less important than for stringizing. Instead, we provide the user a command line option to detect and report instances of this usage.

Preprocessor Output File

The ANSI Standard says nothing about generating an explicit preprocessor output file. Nonetheless, the common view that the preprocessor is a general purpose development tool demands that one give careful consideration to what an output file should look like. Should it contain comments? Should white space be preserved? Should it be compressed? Should `#line` directives be generated? ANSI style or UNIX style? Constrained by our goal of maximum flexibility, we implemented all of the above. When used as a development tool, all seem useful.

Conclusion

Building a production quality ANSI C preprocessor was a more complex task than we had anticipated.

Many extant preprocessors restrict functionality to gain performance and simplicity. An ANSI conforming preprocessor, however, cannot. It must process the language as specified. It is common, for instance, for preprocessors to restrict directives to begin in column 1 to simplify directive recognition. An ANSI preprocessor, though, must recognize directives in any column (preceded by new-line and horizontal white space). Simplifying assumptions are not allowed.

Our design was complicated by our requirement that the preprocessor be backward compatible with others in the field. The preprocessor had to handle both old style and new style features where differences arose. Two stringizing strategies were necessary, for example, two token pasting strategies, and a number of file inclusion algorithms.

We also required that the preprocessor be fast. Since relexing after preprocessing is a serious performance penalty, this requirement seemed most compatible with a single pass, token based design. Creating an explicit output file, however, was somewhat difficult in this setting. The integrity of tokens can easily be lost, if care is not taken to prevent it. White space information can also be lost rendering the output difficult to read. Getting this right was a non-trivial task.

The requirements we set down at the outset of the project pushed us towards an interesting and complex design. Optimizations abound, and the tool is

flexible throughout. It operates as both a general purpose development tool and an integral part of our suite of C compilers.

John Parks is a software engineer at Compass Inc. He developed their ANSI C preprocessor with Rich Peterson and is currently working on the Compass C compilers. He may be reached electronically at parks@compass.com *or by phone at (617) 245-9540.*

∞

[Ed. Inspired by John's discussion of trigraphs, backslash/new-lines, and phases of translation, I constructed the following test program:

```
#include <stdio.h>

#define M(arg) printf(">%s<\n", #arg)

main()
{
M(a/* comment 1 */b);

M(a/??/
\
??/
* comment 2 *??/
\
??/
/b);
}
```

An ANSI-conforming translator should produce the following output:

```
>a b<
>a b<
```

since the arguments in both macro calls are identical after translation phases 1 and 2. Of the 9 compilers claiming ANSI conformance that I tested this on, five failed miserably.

24. ANSI C Interpretations Report

Jim Brodie

Abstract

In this article, I will continue the review of C Standard interpretation requests being addressed by X3J11, the standards committee that developed the American National Standard for C. In particular, I will explore the issues surrounding one of the most difficult and interesting requests.

Function Return Optimization

In the March 1990 issue of *The Journal* (*Volume 1, number 4*), I started writing about the interpretations activities of X3J11. In that article, I noted that a small backlog of interpretations had accumulated for consideration by X3J11. Since that time, there have been several X3J11 meetings. One of these original interpretation requests still remains unanswered. In this article I will explore the issues surrounding this unresolved interpretation request.

This difficult interpretation is one that, at first glance, looks like a fairly simple request. Paul Eggert, in the first formal interpretation request addressed to X3J11, asked the question, "Do functions return values by copying?" After reviewing the interpretation request, it turns out that the real question has to do with the validity of a particular function return optimization that uses a *by reference* rather than a *by value* return approach. In almost every case, the programmer cannot tell how the value is returned. However, the writer has found a case where it makes a difference.

To understand this request, consider the following program, which is a simplification of the one presented in the interpretation request:

```
struct s { int i, j, k, l;};

/* The following function simply returns the structure
to which its argument points.  */

struct s ret_struct(struct s *q)
{
        return *q;
}
```

```
/* Define a union which has two struct s structures that
partially overlap */

union {
        struct { struct s s;} s1;
        struct { int t; struct s s;} s2;
} u;

main()
{
/* initialize the s1 structure */

        u.s1.s.i = 1;
        u.s1.s.j = 2;
        u.s1.s.k = 3;
        u.s1.s.l = 4;

/* The heart of the matter: Does the translator have
to get the following assignment correct? */

        u.s2.s = ret_struct(&u.s1.s);
        printf("%d, %d, %d, %d", u.s2.s.i, u.s2.s.j,
                u.s2.s.k, u.s2.s.l);
}
```

Briefly, the example performs the following actions: The program initializes the members of the `u.s1.s` structure and passes that structure's address to the `ret_struct` function. The `ret_struct` function returns the value of the structure pointed at by its argument. The returned structure value is assigned to the `u.s2.s` structure. A call to `printf` allows us to examine the values stored in the structure members.

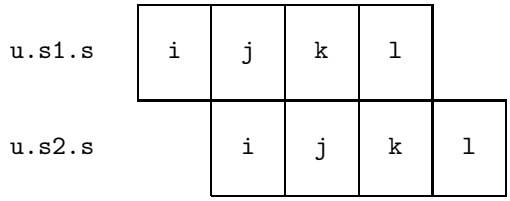
In situations such as the one shown above, some compilers (notably the GNU compiler) handle the `ret_struct` structure return by returning the address of the structure. This avoids copying the entire structure onto the stack. Depending on the size of the structure, this can result in a significant execution-time savings. The assignment following the function call is then done as a straightforward assignment from `u.s1.s` to `u.s2.s`.

The structure return optimization used in this case uses a return *by reference* approach rather than returning the structure value by copying the value onto the stack. (This is why the requestor asks, "Do functions return values by copying?") This optimization leads to code that can fail in such cases. Let's start by looking at why this optimization causes problems.

The first thing to note here is that the structures `u.s1.s` and `u.s2.s` partially overlap. There would be no issue if there was either no overlap or exact overlap. Unions are the only portable way that partial overlaps can arise, although, with the use of casts and pointers we can create almost any situation

imaginable.

The members of structure `u.s1.s` overlap with the members of `u.s2.s` as shown in the following diagram:



The `u.s2.s` members are offset by one due to the addition of the `int`-sized element `t` into the `u.s2` structure definition. Because `u.s1` and `u.s2` are part of a union, the members `u.s1.s.j` and `u.s2.s.i` share the same physical memory. This also holds for `u.s1.s.k` and `u.s2.s.j` as well as `u.s1.s.l` and `u.s2.s.k`.

The partial overlap means that the act of writing members of one structure may interfere with the reading of the members from the other. The function call optimization has the effect of treating the assignment portion of the statement:

```
u.s2.s = ret_struct(&u.s1.s);
```

as if it were the assignment:

```
u.s2.s = u.s1.s
```

The problem that can arise in the assignment revolves around the fact that the `u.s1.s` structure may be (and most probably is) copied a piece at a time into the `u.s2.s` structure. A structure assignment algorithm may do the structure assignment by copying each element, starting with the first element and proceeding, element by element to the last element. Let's see what happens when this approach is used when the source and target partially overlap.

Assume that the members `i`, `j`, `k`, and `l` of `u.s1.s` are initialized with 1, 2, 3, and 4, respectively. The copy starts by moving the value 1 from `u.s1.s.i` to `u.s2.s.i`. Because of the overlap caused by the union, this also has the effect of setting the value of the `u.s1.s.j` to 1. So when the second step of the structure assignment occurs, the copying of `u.s1.s.j` to `u.s2.s.j`, the newly set value of 1, rather than the original value of 2, will be copied. When the structure assignment is complete, all of the members of `u.s2.s` will have the value 1. This is probably not what was intended by the author of the program. (By the way, if you think that this problem could be solved by copying in the other direction, consider the case where the operands of the assignment expression are reversed. In that case the reverse order copying would produce similar results.)

In the partially overlapping case, the resulting behavior of the direct structure assignment expression is undefined according to the Semantics subsection

of ANSI C §3.3.16.1, Simple Assignment:

> If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

The translator is *not* required to handle the partial overlap case correctly because it would result in significant overhead (e.g., copying the members to a temporary structure before starting the writes to the target structure) for structure assignments that have a potential for overlapping members. Unless the potential for overlap is explicitly tracked—and this can become quite complex—this would mean that every structure assignment would be slowed down. This general performance penalty on all structure assignments was deemed to be too costly to justify support for this special case. Therefore the Standard specifies that it is the programmer's responsibility to ensure that partial overlaps do not occur between the source and target.

Despite the benefit and special case clarification that comes from the above assignment expression restriction, an interesting new problem arises out of its wording. This problem is the notion that a value is "accessed from another object" as it is being stored into a data object. This is contrary to the traditional position held by many, that an rvalue value is something independent of the object from which it is derived. This position holds that an rvalue is something you create as the result of the evaluation of an expression (or subexpression) and "hold up in the air" until you are ready to use it.

The wording in the quoted assignment restriction allows, if not requires, an alternate view where the data object origin of the rvalue can also be of importance, at least within the bounds of the assignment expression. In some not completely clear way, the rvalue and the data object it is derived from are linked. The Standard's wording implies that the creation of the rvalue from a subexpression can be interleaved with the side effect of storing the value into another object. Another way of saying this is that the creation of an rvalue is *not* an atomic action.

The question now becomes one of how complicated an lvalue expression trail is supported by the restriction. Does there have to be a direct one-to-one correspondence between the rvalue and the source data object, or is the fact that the partially overlapping data object was used in some way while producing the rvalue sufficient to bring the assignment restriction into play? At the March 1990 meeting of X3J11, Bob Jervis, one of the most experienced X3J11 members, summarized the possible positions. The meeting minutes report:

> Jervis eventually summarized two extreme positions. ... The strict interpretation of the assignment caveat is that only direct assignment between members of the same union is dangerous. The looser interpretation is that determining the value to assign to any union

> member must not involve evaluating another member of the same union.
>
> Jervis asserted that the Standard can support either interpretation. He also argued that no intermediate position is defensible.

At this point, it is not clear whether X3J11 will adopt a strict or loose interpretation.

In some ways X3J11 is like the U.S. Supreme Court, which, in general, tries to avoid establishing a fundamental new position of law. In this case, X3J11 wants to answer the interpretation request, but it would prefer to do so, if possible, by referring to already clear requirements of the Standard rather than by making "new law."

It is clear that the direct assignment is not guaranteed to work correctly. However, does the introduction of the intermediary function call bring other restrictions and requirements of the Standard to bear that will force the assignment to work correctly, even in the partially overlapping case? In other words, does the statement:

```
u.s2.s = ret_struct(&u.s1.s);
```

in the above example always have to work correctly? What restrictions are placed on the translator as it generates code for this statement (and the corresponding function that is called)?

It is clear from the Standard that function arguments are copied. The Standard says, in §3.3.2.2, Function Calls:

> An argument may be an expression of any object type. In preparing for the call to a function the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.

However, the Standard is less clear about returned values. In §3.6.6.4, The return Statement, the Standard states only that:

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

The method that is used to return the value is not specifically stated. This seems to leave open the door to a return *by reference* as well as *by value*.

Do other parts of the Standard set up restrictions that must be satisfied to guarantee that the partially overlapping case must be handled without failure in this case? During the X3J11 discussions, the guarantees (and implementor freedoms) provided by sequence points were raised. Sequence points, for those unfamiliar with them, are defined in the Standard in §2.1.2.3, Program Execution, in the following way:

> Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified point in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

Sequence points help establish boundary points and limitations on the optimization of expressions and their side effects.

Sequence points in the statement:

```
u.s2.s = ret_struct(&u.s1.s);
```

exist prior to the calling of the `ret_struct` function (all of the arguments and the function designator expression must be fully evaluated) and at the end of the statement.

Does the sequence point prior to the function call ensure that the reading of the structure argument will be complete prior to the writing of the target? Several members of the committee felt that this was the case. However, after some discussion, the general opinion of X3J11 seems to be that the sequence point prior to the function call causes only a partial ordering on the subexpressions in the assignment statement. It guarantees the evaluation of the argument expression prior to the function call, but nothing more. This sequence point does *not* control the ordering of the other subexpressions in the assignment statement. Note also that the sequence point prior to the function call only ensures that the evaluation of the pointer to the structure is completed. It does *not* really address the issue of the value of the pointed-to structure.

During the discussion of sequence points, it was noted that the Standard gives considerable freedom to the translator to rearrange the order of operations to make them as fast as possible, so long as the sequence point guarantees are not violated. Expression side effects and data accesses can be advanced or delayed between two sequence points as the translator sees fit. In §3.3, Expressions, the Standard states:

> Except as indicated by the syntax or otherwise specified later (for the function-call operator `()`, `&&`, `||`, `?:`, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

Further, if the code directly or indirectly modifies the same locations multiple times, then the result falls into the undefined behavior category. This position is supported by the the statement in §3.3, Expressions, which establishes the following restriction on the programmer:

> Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an ex-

> pression. Furthermore, the prior value shall be accessed only to determine the value to be stored.

In this example the sequence points that bound the expression are the one at the end of the previous statement:

```
u.s1.s.l = 4;
```

and the one at the end of the complete expression statement including the function call and assignment:

```
u.s2.s = ret_struct(&u.s1.s);
```

The boundaries are essentially the two semi-colons. If you only consider this assignment statement directly, this flexibility would seen to weigh in favor of allowing the function return optimization. The argument is that since this occurs between two sequence points, the reading of the `u.s1.s` structure value can be delayed until, and be interleaved with, the writing of the new value for the `u.s2.s` structure.

The required two-thirds majority of the committee does not accept this argument. Despite many hours of discussions, X3J11 has not yet been able to take a position on this interpretation request. In fact, the last X3J11 vote showed essentially a three-way split between those who though the function return optimization was allowed, those who thought the function return optimization was not allowed, and those who did not know what the answer should be.

One important additional point has been made in the discussions of this interpretation request. The answer to this question affects more that just functions that return structures. These arguments apply just as well to the return of any multiple-byte data object. For instance, in a 16-bit environment a 4-byte `long` might be copied a piece at a time (just as the structure was in this example). If you have partially overlapping `long` objects, `long1` and `long2`, then an expression such as:

```
long1 = ret_long(&long2);
```

would follow the same rules, and optimizations would be limited in the same way as for structures.

Some Additional Thoughts

I would like to add a couple of personal thoughts, beyond those discussed within X3J11.

One difficulty with the view that focuses on the assignment statement and implementor flexibility promised by sequence points is that it essentially ignores the semantics defined within the called function, `ret_struct`. By the operator

precedence rules, the function call must be completed prior to the assignment operation. In particular, the function's `return` statement must have been completed prior to the assignment operation. As noted above, in the description of the `return` statement, the `return` statement evaluates its argument prior to returning from the function, to produce a value.

Despite the committee's focus on the impact of sequence points in this case, sequence points may be the wrong place to be looking for the answer. Sequence points address the timing of modifications of the program state. They do *not* deal directly with the issue of when the production of a value (as a result of an expression evaluation) is complete. The guarantees in this area are addressed in §2.1.2.3, Program Execution, of the Standard, which states:

> The semantic descriptions in this Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

Later in that section is the paragraph:

> In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced ...

It seems, at least to me, that the function `return` *by reference* is a way of delaying the evaluation of the `return` value expression. Since this value is used in the assignment statement once the the function call is complete, it seems that the freedom given in the statement "An actual implementation need not evaluate part of an expression if it can deduce that its value is not used" does not apply.

These points argue that the return by reference approach is invalid. Of course, one person's opinion does not make an interpretation response. I will continue to report on this on-going saga as X3J11 continues its deliberations.

The Impact of X3J11's Indecision

This unresolved interpretation request leaves programmers with a clear direction to pursue, if they want to ensure portable, reliable code. They should *avoid* writing expressions where the resulting value of an object depends on the reliable, well-defined behavior of an expression that includes references to other partially overlapping objects. In that way, they are ensured of portable code no matter what the committee decides.

For implementors, the game is a little more difficult. If you support the partially overlapping case, you will probably generate code that is less efficient than your competitors who follow the GNU lead and perform the function return *by reference* optimization. On the other hand, no matter what X3J11 decides, if you support the partially overlapping case, you will not need to

change anything in this area because of the future decisions of the committee. (If X3J11 rules that the partially overlapping case results in undefined behavior, you will simply have a product that does something nice in this undefined case. If they decide that the partially overlapping case must be supported, you will already supply the appropriate support.)

At this point, implementors have to evaluate the issues, determine what they feel the likely final position will be, and then weigh this against the competitive cost of less than optimal code versus the cost of removing this optimization from a future release of their product.

An ISO C Update

By the time this article is published, there will possibly be an approved ISO (International Standards Organization) Standard for the C programming language. It will be technically identical with the American National Standard. However, the formatting rules for International standards prevents the identical text from being used.

The ISO Standard will guide the development of C programs and translators throughout the world. The on-going international work on Normative Addenda will, once it is developed and approved, modify this Standard. It is interesting to note that the single largest collection of interpretation requests has come from Derek Jones of England. Derek is heavily involved in the development of the (ISO) Normative Addenda. As a leader in this effort, he has brought forward many of the issues the UK has with the ANSI Standard so that he can ensure a clear understanding of the X3J11 view. This is important, since the stated intent of the UK's Normative Addendum has always been editorial clarification rather than making substantive changes to the Standard.

Ed: If any readers have thoughts on how this issue might be resolved they are encouraged to contact Jim (or any other X3J11 member) as indicated below.

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

∞

25. Pragmania

Rex Jaeschke

DEC's PDP-11 C V1.0

The following information applies to version 1.0 of PDP-11 C, a compiler hosted on VAX or PDP-11 systems running under various DEC propriety operating systems. The compiler generates code for PDP-11 systems only.

The preprocessing tokens between `#pragma` and its terminating new-line are subject to macro replacement provided they are not enclosed in single or double quotes. Unrecognized pragmas cause an informational message to be produced.

Character Set Specification

The `charset` pragma can be used to specify the character set for each of the source, message, list, and execution files. It is used as follows:

`#pragma charset` [*category*] [*charset_name*]

where *category* may be one of the following:

```
source      message          list          execution
```

and *charset_name* may be any one of:

```
ascii       dutch            german        portuguese
british     finnish          iso_latin_1   spanish
danish      french           italian       swedish
dec_mcs     french_canadian  norwegian     swiss
```

By default, all four categories are `iso_latin_1` and each category may be specified independent of the others. If the category is omitted, all four categories are assumed.

A header (including nested headers) assumes the same character set as its including file. The effect of any character set specified within a header is, however, implicitly disabled at the end of that header.

String literals and character constants are translated to the execution character set.

`charset` pragmas allow you to use older, 7-bit input/display devices that assign certain character positions to represent certain national-specific glyphs.

For instance, the ASCII decimal code 92 normally represents the backslash ('\'). However, the 7-bit French National Replacement Character set (NRC) replaces this position with the French c-cedilla ('ç'). Consider the case where program development is performed using an ISO Latin-1 8-bit input/display device that does not require character set translation, but where the program is to display text on a 7-bit French NRC terminal at run time. PDP-11 C supports this as follows:

```
#include <stdio.h>
#pragma charset execution french

int main (void)
{
        printf ("Ici, on parle Français.\n");
}
```

If the pragma were omitted and this program run on a 7-bit French NRC terminal, the message would display as follows:

```
Ici, on parle Frangais.
```

The 'ç' in 'Français' is displayed as 'g' because the high order bit of the 8-bit 'ç' (ASCII decimal code 231) is stripped, resulting in 'g' (ASCII decimal code 103). The result is far from gratifying to a Frenchman. The pragma directs the compiler to translate the characters in the string literal to correspond to the French NRC, and the result appears on the French NRC terminal as follows:

```
Ici, on parle Français.
```

The Swiss NRC replaces the '_' character with 'é'. This is contrary to ISO standards for NRCs, but they do it anyway. And because the Swiss approach is not sanctioned by an ISO standard, it is not covered by the trigraphs invented by ANSI C. Thus, if you specify your source character set as Swiss, there is no way to represent the underscore character in your program—a new trigraph would be needed. However, it is presumed that most program development will be done using 8-bit DEC MCS or ISO Latin-1 input/display devices and that this pragma will be used primarily for run-time and listing display devices. (DEC MCS is the character set used by Digital's VT200 series of terminals and later. It pre-dates the establishment of ISO Latin-1 and is almost identical. Most C programs need not consider the minor differences between DEC MCS and ISO Latin-1. While PDP-11 C can translate between DEC MCS and ISO Latin-1, it is usually not necessary.)

The ability to specify different character sets for source files or compile-time messages may be desirable for some customers. Except for the case of '_' when using the Swiss character set for source files, trigraphs allow all characters used in C to be represented. Note that using the Swiss character set for run-time

display introduces no complications (unless you need to display the '_' glyph—but then, a Swiss NRC display device is not capable of generating that glyph!). The problem occurs only when you try to use the Swiss NRC for source files.

Code and Data Placement

In the PDP-11 (and VAX/VMS) environment, DEC's language translators partition code and data in object modules into units called *program sections* (or more simply, psects). The default psect placement can be overridden using a family of pragmas having the following form:

`#pragma psect` *psect_type* [*psect_name* [, *attributes, ...*]]

This pragma allows PDP-11 C code to be linked with object modules written in other languages and whose compilers have different psect defaults.

Object Module Title Control

The module identification information of an object module can be controlled with a pragma of the following form:

`#pragma module` *title* [[,] *version*]

The title and version must be strings of no more than 6 characters and may, of course, result from macro expansions.

This pragma allows project revision information to be included in the object module. (Object module version numbers are include in linker maps.)

Listing Title Control

The title and subtitle on list file pages can be established using the following set of pragmas:

```
#pragma list on
#pragma list off
#pragma list title      "..."
#pragma list subtitle   "..."
```

The list on/off switch is cumulative. That is, it is a counter that starts at zero and is incremented for each `list on` and decremented for each `list off`. A listing is produced so long as the counter is positive.

Inter-Language Communication

A problem common to many implementations is that of linking together object modules produced from different (and somewhat incompatible) transla-

tors. Unlike the VAX, the PDP-11 has no standard calling convention. As a result, different compiler and utility design groups chose different register and stack layouts for argument passing. To help resolve these conflicts, PDP-11 C provides the linkage pragma whose syntax is as follows:

```
#pragma linkage linkage_type [ function1 [, function2 ] ... ]
```

where *linkage_type* specifies a language such as `c`, `pascal`, or `fortran`.

The functions whose names are listed are assigned the corresponding linkage. If the function list is omitted, the linkage type specified becomes the default for all the functions that follow. Linkage pragmas remain in force until overridden by another linkage pragma.

Disabling Standard Mode

Compiling in 'standard mode' is useful as far as extra checking is concerned. However, sometimes it is useful to be able to temporarily switch off this mode. This can be achieved using the following pragmas:

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

If these directives do not occur in pairs, an informational message is produced.

TopSpeed C V1.04

[Ed: The following information is taken from JPI's documentation which is © 1989–90, by Jensen & Partners International. It is reproduced here with their kind permission.

TopSpeed C is a member of a series of MS-DOS based languages from Jensen and Partners International (JPI) that conform to international standards. Of these, Ada is the only one that has a formal definition for a pragma syntax. As such, JPI chose the official Ada standard syntax. This has the advantage that all TopSpeed languages share the following pragma syntax:

```
#pragma pragma_name( parm1 => val1 [,  parm2 => val2, ] ... )
```

For example:

```
#pragma call( c_conv => off, near_call => off )
```

turns standard C calling conventions off and near calls off. The next example:

```
#pragma call( reg_param => (ax,bx,cx,dx) )
```

tells TopSpeed C to use the indicated registers for parameter passing to functions.

The set of pragmas are broken into the following groups:

TopSpeed Pragma Groups	
Category	*Purpose*
`call`	Specify function call convention
`check`	Allow run-time checking for various conditions
`data`	Control data storage location
`link`	Specify object file list for linking
`name`	Control public name spelling
`option`	Override default (or explicit) compiler options
`save`	Save and restore current pragma state
`warn`	Control numerous warning and information messages

Some Highlights

While there are many many pragmas defined by TopSpeed C, only a few of them will be discussed here in detail.

call – Specify whether arguments are passed by register or stack. For register, the registers used and which ones should be preserved. For stack, the order pushed and who cleans up the stack.

check – Run-time checking for dereference of `NULL`, invalid array index access, and stack overflow.

data – Can establish stack and heap size and define shared globals in a multi-thread environment.

name – Can add a user-specified global name prefix and specify if external names are to be case-significant.

save – You can save an entire pragma state for later restoration. Since a stack mechanism is used, pragma states may be nested.

warn – Numerous very useful *lint*-like warnings can be produced, for example:

- No expression in `return` statement.
- No return from non-`void` function.
- Function called and not declared.
- No prototype in scope of call.
- Code has no effect.
- Assignment in test expression (e.g., `if (x = y)`).
- Returned address of an automatic variable.

- Variable declared but never used.
- Variable assigned but never used.
- Possible use of variable before assignment.
- Parameter never used.
- Unknown pragma.

The Complete Grammar

Pragma tokens are case-sensitive like any other token in C.

pragma_directive ::= `#pragma` *directive_list*

directive_list ::=
directive
directive_list *directive*

directive ::=
call_pragma
check_pragma
data_pragma
link_pragma
name_pragma
option_pragma
save_pragma
warn_pragma

call_pragma ::= `call (` *call_param_list* `)`

call_param_list ::=
call_param
call_param_list `,` *call_param*

call_param ::=
`c_conv` `=>` *on_off*
`ds_entry` `=>` *seg_name*
`inline` `=>` *on_off*
`interrupt` `=>` *on_off*
`io_priv` `=>` *on_off*
`near_call` `=>` *on_off*
`reg_param` `=>` *reg_list*
`reg_return` `=>` *reg_list*
`reg_saved` `=>` *reg_list*
`same_ds` `=>` *on_off*
`seg_name` `=>` *seg_name*
`windows` `=>` *on_off*

on_off ::=
```
off
on
```

seg_name ::=
```
none
null
```
any other C identifier

reg_list ::= `(` *reg_id_list* `)`

reg_id_list ::=
reg_id
reg_id_list `,` *reg_id*

reg_id ::=
```
ax | bx | cx | dx |
si | di | ds | es |
st0 | st1 | st2 | st3 |
st4 | st5 | st6
```

check_pragma ::= `check (` *chk_param_list* `)`

chk_param_list ::=
```
index   => on_off
nil_ptr => on_off
stack   => on_off
```

data_pragma ::= `data (` *data_param_list* `)`

data_param_list ::=
data_param
data_param_list `,` *data_param*

data_param ::=
```
far_ext      => on_off
heap_size    => number
near_ptr     => on_off
seg_name     => seg_name
ss_in_dgroup => on_off
stack_size   => number
```

link_pragma ::= `link (` *object_file_list* `)`

object_file_list ::=
identifier
object_file_list `,` *identifier*

name_pragma ::= `name (` *name_param_list* `)`

name_param_list ::=
name_param
name_param_list `,` *name_param*

name_param ::=
`prefix =>` *string*
`upper_case =>` *on_off*

string ::= any string literal

option_pragma ::= `option (` *opt_param_list* `)`

opt_param_lst ::=
option_param
opt_param_lst `,` *option_param*

option_param ::=
`ansi =>` *on_off*
`lang_ext =>` *on_off*

save_pragma ::=
`restore`
`save`

warn_pragma ::= `warn (` *warn_param_list* `)`

warn_param_list ::=
warn_param
warn_param_list `,` *warn_param*

warn_param ::=
`wait =>` *on_off_err*
`watr =>` *on_off_err*
`wcld =>` *on_off_err*
`wclt =>` *on_off_err*
`wcne =>` *on_off_err*
`wcor =>` *on_off_err*
`wdne =>` *on_off_err*
`wdnu =>` *on_off_err*
`wetb =>` *on_off_err*
`wfnd =>` *on_off_err*
`wftn =>` *on_off_err*
`wnid =>` *on_off_err*
`wnre =>` *on_off_err*
`wnrv =>` *on_off_err*
`wntf =>` *on_off_err*

```
                    wpcv => on_off_err
                    wpic => on_off_err
                    wpin => on_off_err
                    wpnd => on_off_err
                    wpnu => on_off_err
                    wprg => on_off_err
                    wral => on_off_err
                    wrfp => on_off_err
                    wsto => on_off_err
                    wtxt => on_off_err
                    wubd => on_off_err
                    wvnu => on_off_err

on_off_err ::=
                    err
                    off
                    on
```

∞

26. Restricted Pointers

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

Abstract

The initial meeting of the NCEG committee identified and assigned priorities to several key issues that needed to be addressed. C aliasing was identified as being the highest priority issue because of its effect on optimization. One approach being explored is a new kind of pointer called a *restricted pointer*. A restricted pointer gives the compiler the liberty to assume that the pointer behaves "as if it were an array" for aliasing purposes.

The array analogy is being explored because optimizations involving arrays are understood, and programmers can view this new kind of pointer in terms of an existing concept. Several new areas must be explored though, because pointers can be used in ways that arrays cannot. The greatest benefits of restricted pointers are achieved when they are used for function parameter declarations, or when they are the targets of calls to dynamic memory allocation functions.

Introduction

In the September 1989 issue of *The Journal* (*Volume 1, number 2*), I wrote an article, *Aliasing Issues in C*, in which I discussed the optimization issues associated with the unconstrained aliasing present in C. I alluded to a new kind of pointer that would help optimizers. This article is a follow on to that one. It proposes a new kind of pointer that ameliorates C aliasing.

The following two terms are used throughout the rest of this article:

- *reference* – access or modify an object
- *lvalue* – an expression that references an object

The presence of unconstrained pointers in C introduces aliasing that inhibits many useful optimizations. Two of these optimizations are automatic vectorization and automatic parallelization. On a single processor CRAY Y-MP, a typical vector loop runs ten to twenty times faster than the equivalent scalar

loop. Enhancing the optimization potential of C is the motivation behind this proposal for a new kind of pointer.

Automatic vectorization requires the optimizer to determine that it is safe to reference many array elements simultaneously. Automatic parallelization requires the optimizer to determine that different statements are independent and can execute in parallel. Loop-level parallelization applies these optimizations to entire loops.

In order to perform these optimizations correctly, a translator must know that, within a loop, an object modified through one lvalue is not the same object referenced through a different lvalue. When an object is referenced through a pointer, the optimizer does not always know which underlying object is referenced. The remainder of this section is an excerpt taken from my *Aliasing Issues in C* article. It is a good example of the problems encountered when trying to apply loop-level parallelism to C programs.

The following example demonstrates that pointers can introduce hidden aliases that are not detectable at compile time.

```
01 #include <stdio.h>
02
03 int a[6] = {0, 1, 2, 3, 4, 5};
04 int b[6] = {9, 8, 7, 6, 5, 4};
05 int c[6];
06
07 void blackbox(int *p1, int *p2, int *p3, int n);
08
09 main() {
10      int i;
11
12      blackbox(c, b, a, 6);           /* no aliases */
13      for (i = 0; i < 6; i++)
14              printf(" c[%d] = %d ", i, c[i]);
15      putchar('\n');
16
17      blackbox(&a[1], &a[1], a, 5);   /* aliases */
18      for (i = 0; i < 6; i++)
19              printf(" a[%d] = %d ", i, a[i]);
20      putchar('\n');
21 }
22
23 void blackbox(int *p1, int *p2, int *p3, int n) {
24      int i;
25
26      for (i = 0; i < n; ++i)
27              *p1++ = *p2++ + *p3++;
28 }
```

The following output is produced when the program is executed in scalar fashion.

```
c[0] = 9 c[1] = 9 c[2] = 9 c[3] = 9 c[4] = 9 c[5] = 9
a[0] = 0 a[1] = 1 a[2] = 3 a[3] = 6 a[4] = 10 a[5] = 15
```

The function `blackbox`, whose definition starts on line 23, appears to add the corresponding elements of two arrays together, storing the results in a third array. This is exactly what happens when `blackbox` is called without any aliases at line 12. The resulting array `c` contains the sum of `a` and `b`. This makes the loop inside `blackbox` appear to be a candidate for parallelization. However, when `blackbox` is called with aliases at line 17, something different happens. Each element of the resulting array `a` contains partial sums of the values in the preceding elements. This time the loop must be executed as a scalar loop to obtain the correct results. Since, `blackbox` is not declared `static`, it can be called from a separately compiled module. Therefore, the compiler must make the worst case assumption that this loop might contain aliases.

Just Like an Array

Unlike pointers, array references identify the underlying object being referenced. The optimizer can assume that an lvalue involving array `a`, for instance `a[i]`, can never reference the array `b`. No aliasing is possible between two different arrays (assuming, of course, no unions are present). If the `blackbox` function referenced arrays directly, then no hidden aliasing exists, and the optimizer can determine that it is safe to parallelize the `for` loop.

```
int a[6] = {0, 1, 2, 3, 4, 5};
int b[6] = {9, 8, 7, 6, 5, 4};
int c[6];

void blackbox(int n) {
        int i;

        for (i = 0; i < n; ++i)
                /* no hidden aliasing */
                c[i] = b[i] + a[i];
}
```

Unlike arrays, it is not possible in ANSI C to indicate that two pointers are never aliases for each other.

```
int *a;
int *b;
int *c;

void blackbox(int n) {
        int i;

        for (i = 0; i < n; ++i)
           c[i] = b[i] + a[i];  /* potential aliasing */
}
```

If there were some way to indicate that pointers `a`, `b`, and `c` are never aliases for each other, and behave just like arrays, then optimizations similar to those permitted in the first case would also be permitted in the second.

The proposed semantics for a restricted pointer is that it behaves *just like an array* when being viewed by the optimizer. That is, if an optimization can be applied to an array reference, the same optimization can be applied to a restricted pointer reference. The proposed syntax for restricted pointers involves a new keyword, `restrict`, and syntax similar to that of the type qualifiers `const` and `volatile`.

```
double * restrict p;
```

The semantics of this new type of pointer involves only assertions about aliases. The primary motivation is to allow the compiler to perform more optimizations. If the compiler is not optimizing, then the semantics of `restrict` can be ignored (just like `volatile`).

The semantics of restricted pointers needs to capture the concept that aliasing assertions, that are correct for arrays are also correct for restricted pointers. The array characterization is used because it allows the compiler and the programmer to view the program in the same way. The `blackbox` function can now be written in the following way:

```
void blackbox(int * restrict p1, int * restrict p2,
    int * restrict p3, int n) {
        int i;

        for (i = 0; i < n; ++i)
                /* no hidden aliasing */
                *p1++ = *p2++ + *p3++;
}
```

The compiler can now assume that the restricted pointers `p1`, `p2`, and `p3` behave like distinct arrays for optimization purposes. This means the following call:

```
blackbox(&a[1], &a[1], a, 5);   /* aliases */
```

is no longer well defined behavior because the pointers passed to `blackbox` do not point to distinct arrays. Of course, a compiler could warn about potential aliasing problems if such a call were in the presence of a prototype.

Array Semantics

Many optimizations require the optimizer to know how far lvalues can move relative to other lvalues. Considerable latitude is given to the optimizer in reordering the following array references because they are lvalues that reference disjoint objects.

```
#include <stdlib.h>

int a[10];

void f1(int i) {
        static int b[10];
        int c[10];
        int *p1;

        p1 = malloc(10 * sizeof(int));
        a[i] = i;
        b[i] = i;
        c[i] = i;
        p1[i] = i;
}
```

It is perfectly acceptable for the optimizer to reorder the assignments as follows:

```
#include <stdlib.h>

int a[10];

void f1(int i) {
        static int b[10];
        int c[10];
        int *p1;

        c[i] = i;
        b[i] = i;
        p1 = malloc(10 * sizeof(int));
        p1[i] = i;  /* cannot be moved before ‘malloc’ */
        a[i] = i;
}
```

These reorderings just affect the order in which assignments are performed on independent objects. This is acceptable because the semantics of these arrays is such that the optimizer knows where their storage resides, when the storage is allocated, and just how far loads and stores to these objects can be moved. For instance, the assignment through pointer `p1` cannot be moved before the `malloc` call that allocates the array. They are all arrays and, therefore, independent objects. Since optimizations like automatic vectorization must reorder array references, the optimizer must know how far they can be moved. The following example replaces all array declarations with restricted pointers.

```
#include <stdlib.h>

int * restrict a;

void f2(int i) {
        static int * restrict b;
        int * restrict c;
        int * restrict p1;

        p1 = malloc(10 * sizeof(int));
        a[i] = i;
        b[i] = i;
        c[i] = i;
        p1[i] = i;
}
```

Since a restricted pointer is "just like an array," these assignments can be reordered by the optimizer in similar ways.

C is a block scoped language, with blocks delineated by { and } tokens. Two blocks are considered to be siblings if they have the same parent block. The next example shows some different behavior for different arrays defined in sibling blocks.

```
void f3(int i) {        /* parent block    */

        {                   /* sibling block 1 */
        static int m[10];
        int n[10];          /* overlaps with y */

        m[i] = i;
        n[i] = i;
        }
```

```
        {                   /* sibling block 2 */
        static int x[10];
        int y[10];          /* overlaps with n */

        x[i] = i;
        y[i] = i;
        }
}
```

The `auto` arrays `n` and `y` are allowed to share the same physical memory location. That is, it is quite common for compilers to allow `auto` variables to overlay the same physical memory as `auto` variables in sibling blocks. This means the optimizer must not move assignments to array `y` before any references to array `n`. In a way, they are aliases with each other. This is referred to as the *auto array model*. However, arrays `m` and `x` are `static` arrays and do not share the same physical memory locations. The assignment to `x` can be moved before the assignment to `m`. This is referred to as the *static array model*. The following is an acceptable reordering of assignments by the optimizer (though one could not write C code this way).

```
void f3(int i) {     /* parent block    */

        x[i] = i;           /* OK to move here */
        {                   /* sibling block 1 */
        static int m[10];
        int n[10];          /* overlaps with y */

        m[i] = i;
        n[i] = i;
        }

        {                   /* sibling block 2 */
        static int x[10];
        int y[10];          /* overlaps with n */

        y[i] = i;           /* cannot move     */
        }
}
```

This raises the question about which model to follow. In the following example, should the two restricted pointers `p` and `q`, declared inside the sibling blocks, follow the auto array model or the static array model?

```
void f4(int i) {

        {
                int * restrict p;
        }

        {
                int * restrict q;
        }
}
```

There is a third model that can be used to describe the behavior of restricted pointers declared in local blocks. A restricted pointer behaves as if it points into memory obtained from a call to `malloc`. §4.10.3 (Memory Management Functions) of the ANSI standard guarantees that each call to `malloc` "shall yield a pointer to an object disjoint from any other object." This allows the optimizer to assume that lvalues derived from restricted pointers are never aliases with lvalues derived from arrays. This is called the *malloc array model*.

```
int a[10];

void f5(int i) {

        {
        int * restrict p /* = malloc(N) */;
        /* cannot be moved above implied malloc call */
        a[i] = p[i];
        }

        {
        int * restrict q /* = malloc(N) */;
        /* cannot be moved above implied malloc call */
        a[i+1] = q[i];
        }
}
```

The optimizer can assume that `p` and `q` point into disjoint arrays that were allocated by `malloc`. Therefore, neither points into array `a`. The `malloc` calls are hidden inside comments, to show the implied behavior the optimizer is relying upon. The simplicity of the "as if the array came from a `malloc` call" makes this model easy to understand and straightforward to define.

Another effect of using this model is that the aliasing assertions of restricted pointers are confined to the scope that contains the declaration. They are only in effect when the declaration is visible. There are no aliasing assertions if the restricted pointer is not visible. This is part of the behavior guaranteed by the implied `malloc` call. Therefore, neither of the two assignments present in

function `f5` can be moved outside the block that contains the declaration of the restricted pointer.

It is important to understand that the implied `malloc` calls are not part of an alternate execution model for programs containing restricted pointers. Rather, they represent aliasing assumptions that only the optimizer can use for purposes of statically analyzing the program. It is the programmer's responsibility to ensure that these assumptions are correct. The following example shows both well defined behavior and undefined behavior.

```
void f6(int i) {

        i = 1;  /* defined behavior */

        {
        int * restrict p = &i;

        *p = 3; /* defined behavior */
        }

        {
        int * restrict q = &i;

        i = 3;  /* i and *q are not disjoint */
        *q = 2; /* undefined behavior */
        }
}
```

The assignments to object `i` occur both through `i` and through two restricted pointers that point to `i`. The undefined behavior exists when the assignment through `i` occurs in a context where either of the two restricted pointers is visible. The behavior is undefined because the optimizer is at liberty to reorder the two assignments. If neither of the two restricted pointer is visible, assignments to `i` are well defined.

Pointer and Array Differences

The two primary purposes of restricted pointers are to declare that formal parameters point at disjoint objects and to assert that space acquired from `malloc` calls is only accessed through a single restricted pointer. The following example demonstrates both of these uses.

```
int * restrict a;

void f7(int * restrict b, int * restrict c) {

        int i;
        int * restrict p;

        a = malloc(100 * sizeof(int));
        p = malloc(100 * sizeof(int));

        /* parallelizable loop */
        for (i = 0; i<100; i++) {
                a[i] = b[i] + c[i];   /* no aliasing */
                p[i] = b[i] * 2;      /* no aliasing */
        }
}
```

For these two purposes, the aliasing assertions work very well. This alone makes the restricted pointer proposal appealing because these are the areas in the language where restricted pointer semantics are needed the most. However, the "just like an array" paradigm needs to be explored in areas of the language where pointers can be used in ways that arrays cannot. This is necessary to achieve closure on the language for restricted pointers.

One of the primary purposes just identified for restricted pointers is to indicate that formal parameters are not aliases with each other, as in:

```
void blackbox(int * restrict p1, int * restrict p2,
    int * restrict p3, int n);
```

However, C does not permit a formal parameter to be declared as an array. What does it mean to say that formal parameters behave just like arrays? Is it meaningful to say that the implied `malloc` call occurs inside function prototype scope? The question becomes, does the implied `malloc` call occur before the function is called or after the function is entered? The answer matters when one parameter is declared to be a restricted pointer and another is declared to be an unrestricted pointer.

```
void f8(int *p, int * restrict q) {
        /* is the behavior as if q=malloc(N)? */

        int i;

        for (i=0; i<10; i++)
                p[i] = q[i];
}
```

```
int *a, *b;

main() {
        /* is the behavior as if b=malloc(N)? */
        f8(a, b);
}
```

If the answer is that the implied `malloc` call is associated with `b`, the actual argument, and occurs sometime before the call to `f8`, then the optimizer must assume that `p` and `q` are potential aliases. Remember that the unrestricted pointer `p` can point anywhere, including into the same space as `q`. However, if the implied `malloc` call is associated with the formal parameter `q`, then the optimizer can assume that `p` and `q` are not aliases at the time the function begins executing. Ultimately the choice depends upon how much latitude should be given to the optimizer.

Since restricted pointers are intended to help optimizers it seems reasonable to allow more optimizations, and to assume the implied `malloc` call is associated with the formal parameter. This is consistent with the view that declarations of restricted pointers are associated with an implied `malloc` call. Since the declaration of a formal parameter occurs within the function definition, it is consistent to view the implied `malloc` call as being associated with the formal parameter. Most likely, all or none of the formal parameters declared to be pointers will be declared to be restricted pointers. Therefore, the answer is important for completeness only.

Another difference between a pointer and an array implicitly converted to a pointer is subscripting with negative indices. An array reference such as `a[-1]` is always undefined behavior. However, a pointer reference such as `p[-1]` can be well defined behavior. For this reason, a restricted pointer only needs to behave as if it points into an array allocated by `malloc`, but does not have to behave as if it points to the first element. The following:

```
int * restrict p = (int *)malloc(N * sizeof(int)) + n;
```

is an equally acceptable implied `malloc` call for the optimizer to make. This allows restricted pointers to have negative indices and still behave just like arrays.

Pointer subtraction between two arrays is not well defined behavior. The result of `a - b` where both `a` and `b` are arrays is not guaranteed to be meaningful, because pointer subtraction is only defined when both pointers point into the same array. Should the difference between two restricted pointers also be undefined? This certainly seems reasonable because both pointers are supposed to behave as if they point into arrays obtained from separate calls to `malloc`.

Although functions can return pointers, they cannot return arrays. What then are the semantics of the type *function returning restricted pointer*? Since function call expressions do not return lvalues, no aliasing assertions are made by restricting the return type of a function.

```
/* no aliasing assertions */
int * restrict f9(void) {
        static int a[10];

        return a;
}
```

If the keyword `restrict` were left out of the return type, the function would have the same semantics. Similarly, no aliasing assertion is made by restricting a pointer to a function, since such a pointer does not point at an object.

```
/* no aliasing assertions */
int (* restrict pf)(void);
```

Comparison with `noalias`

The X3J11 committee attempted to solve the aliasing problem in C by introducing a new type qualifier, `noalias`. That effort failed because of technical problems with the proposed semantics of `noalias`. The restricted pointer proposal is different in many ways. Only pointers can be declared to be restricted. In the `noalias` proposal, all objects were permitted to be `noalias`-qualified. It is the declaration of the restricted pointer that makes the aliasing assertions, while it was the `noalias`-qualified lvalue that made the aliasing assertions. A restricted pointer can be an alias with an unrestricted pointer, whereas a pointer to a `noalias`-qualified type was guaranteed to be alias free. The proposed semantics of `noalias` defined an alternate execution path in which virtual objects were created and later synchronized with the original object. No alternate execution path is defined for restricted pointers. The proposed semantics for restricted pointers merely permit the optimizer to statically analyze restricted pointers. Finally, a block scope restricted pointer only makes assertions on the containing scope. In the `noalias` proposal, a block scope `noalias`-qualified object made assertions that affected the entire containing function.

Conclusions

There is no substitute for an actual implementation. The true worth of the restricted pointer proposal will be determined when a compiler is available that can be tested. Several things must happen before restricted pointers are considered a success. First, there must be improved execution times of applications modified to use restricted pointers. Second, programmers must be willing to use restricted pointers. And, finally, the semantics must be understandable and sensible. It must be easy to understand when it is safe to use a restricted pointer and when it is appropriate to use the traditional unrestricted pointer.

A restricted pointer is only useful for optimization purposes. At the point a restricted pointer is declared, the optimizer is allowed to assume that it points

into an associated array allocated by an implied call to `malloc`. This means the optimizer can assume that the associated array is an object disjoint from all other objects. Furthermore, the aliasing assertions are only meaningful while the declaration of the restricted pointer is visible.

I would like to thank my colleagues Bill Homer and Steve Collins for their input into this article.

Tom MacDonald is the Numerical Editor of The Journal of C Language Translation. He is Cray Research Inc's representative to X3J11 and a major contributor to the floating-point enhancements made by the ANSI standard. He specializes in the areas of floating-point, vector, array, and parallel processing with C language and can be reached at (612) 683-5818, tam@cray.com, *or* uunet!cray!tam.

∞

27. Miscellanea

compiled by **Rex Jaeschke**

Extensions

MetaWare's High C

High C contains a number of interesting extensions, most of which were influenced by Pascal and Ada.

The compiler version used for this article was R2.3c running under MS-DOS on an Intel 80386. It runs in 32-bit native mode where the types `int`, `long`, and all pointer flavors are represented in 32 bits.

Case Ranges

Non-overlapping case ranges are permitted with `switch` as follows:

```
#include <stdio.h>

main()
{
        int c;

        while ((c = getchar()) != EOF) {
                switch (c) {

                case 'a'..'m':
                        printf("a-m: %c\n", c);
                        break;

                case '0'..'9':
                        printf("0-9: %c\n", c);
                        break;

                default:
                        printf("default:\n");
                        break;
                }
        }
}
```

It is interesting to note that while `1..3` represents a range of three cases, under ANSI C rules this is one token; a *pp-number*. Therefore, to allow this extension, High C disallows `..` in a *pp-number* except when compiling in ANSI mode.

This extension has also been provided by other implementors such as Whitesmiths.

Structure Member Alignment

Like some other compilers, High C provides a compiler option and/or a pragma to enable/disable structure member packing. The options they provide are: byte-aligned or 32-bit word-aligned. High C allows packing to be specified using the keywords `_packed` and `_unpacked`, as follows:

```
#include <stdio.h>

_packed struct t1 {
        char c;
        int i;
} s1;

_unpacked struct t2 {
        char c;
        int i;
} s2;

main()
{
        printf("sizeof(s1) = %u\n", sizeof(s1));
        printf("sizeof(s2) = %u\n", sizeof(s2));
}

sizeof(s1) = 5
sizeof(s2) = 8
```

Named Parameter Association

Arguments in a function call can be arranged in any order provided they contain the name of the formal parameter to which they correspond. For this to work, *every* argument in a prototype *must* contain an identifier. As such, the technique cannot be used with functions containing variable argument lists.

The actual argument list may have one of two possible formats: either every argument is explicitly associated with a parameter using the syntax

formal-arg `=>` *actual-arg*

or the leading arguments have no such association and are interpreted according to their position and trailing arguments have explicit association. For example:

```
#include <stdio.h>

main()
{
        void f(char c, int i, double D, char *pc);

        int j = 10;
        double b = 1.2;
        char *p = "abcd";

        f('x', D =>b, pc => p + 1, i => j * 5);
}

void f(char c1, int i1, double D1, char *pc1)
{
        printf("c1 = %c, i1 = %d, D1 = %f, pc1 = %s\n",
                c1, i1, D1, pc1);
}

c1 = x, i1 = 50, D1 = 1.200000, pc1 = bcd
```

In this case, function `f` is called with 4 arguments. The first has no explicit association and is therefore interpreted as corresponding to the first formal argument (`char c`) in the prototype. The trailing arguments are associated with the remaining formal argument names using the `=>` notation. As you would expect, the case of the formal argument identifiers is significant. However, the names used in the function definition do not have to match those in the prototype.

Nested Functions

High C permits functions to be nested, as follows:

```
#include <stdio.h>

main()
{
        void f(void);

        f();
}
```

```
void f(void)
{
        void g1(void);
        void g2(void);
        static int i = 10;

        puts("Inside f");

        void g1(void)
        {
                printf("Inside g1: i = %d\n", i);
        }

        g2();

        void g2(void)
        {
                printf("Inside g2: i = %d\n", i);
                i = 20;
        }

        g1();
}

Inside f
Inside g2: i = 10
Inside g1: i = 20
```

The definition of function `f` contains the definitions of two other functions: `g1` and `g2`. That is, these two functions are nested within `f`. As such, they can access any local identifiers declared within their parent(s) except for those having storage class `register`. The definition of a nested function need not precede its use, but if it doesn't, it should at least be declared prior to use (as is the case with `g2` here).

Note that High C also permits declarations and statements to be interspersed within a block.

Nested functions require extra support if their address is to be taken and used meaningfully. Since a nested function 'belongs' to another function (and can access its local variables), it has a current context which includes information about its parent. This context information is called an *environment*. An environment along with the corresponding function's address is called a *full function value*. So a pointer to a function is not able to store the context of a nested function. Instead, a pointer to a nested function and a context pointer are needed. High C refers to such a pointer pair collectively by the term *full function value variable*, which can be declared as follows:

```
void pnf(void)!;
```

Here, `pnf` is declared as a variable capable of containing the full function value of a nested function having no arguments and no return value. In short, `pnf` is a pointer to a nested function that has these attributes. `pnf` can be assigned the value of a nested function simply by assigning it the function's name. The name of a nested function is *not* converted to a pointer to that function. In fact, you *cannot* take the address of a nested function at all. `pnf` can also be assigned the value of a non-nested function by assigning it the dereferenced function's name. For example:

```
pnf = f1;
```

causes `pnf` to contain the value of the nested function `f1`, while:

```
pnf = *h;
```

causes `pnf` to contain the value of the non-nested function `h`. Since non-nested functions do not have environments, a dummy one is created for them in contexts where one is needed (as in the second case above) but it is thereafter ignored.

[As a side-issue, the most recent proposal from Bjarne Stroustrup to add a readable alternative to trigraphs to ANSI C++ (and presumably to ISO C) includes adding `!` *as a postfix punctuator in declarations involving arrays. This, of course, would lead to a conflict with High C, as he has been made aware.]*

The following example demonstrates the use of nested functions and full function value variables.

```
#include <stdio.h>

main()
{
        void f1(void)
        {
                puts("\nIn function f1");
        }

        void pnf(void)! = f1;

        printf("sizeof(pnf) = %u\n", sizeof(pnf));
```

```
        void f2(void p(void)!)
        {
                puts("\nIn function f2");
                p();    /* call f1 */
        }

        f2(f1); /* pass nested function's full value */
        f2(pnf);

        void g(void p(void)!);

        g(f1);  /* pass nested func to non-nested func */

        void h(void);

        pnf = *h; /* get 'value' of global function */
        pnf();
}

void g(void p(void)!)
{
        puts("\nIn function g");
        p();    /* call f1 */
}

void h(void)
{
        puts("\nIn function h");
}

sizeof(pnf) = 8

In function f2

In function f1

In function f2

In function f1

In function g

In function f1

In function h
```

Note that while all function pointers in High C are 4 bytes long, the sizeof a full function value is 8 bytes.

Full function values can be passed to functions. Function `f2` expects such an argument and is declared as follows:

```
void f2(void p(void)!);
```

This allows that function to be called using an expression of that type, as follows:

```
/* use nested function's name */
        f2(f1);

/* use variable having nested function's full value */
        f2(pnf);
```

In fact, full function values can also be returned from functions using the expected notation (even though it looks rather unusual). For example:

```
int g(void)(void)!;
```

Function `g` takes no arguments and returns a full function value of a function taking no arguments and returning an `int`.

The last part of the example passed a nested function's full function value to a global function which then indirectly calls that nested function. So while a nested function is in some sense private to its parent(s), it can still be called indirectly from outside its parent(s) but only if they have at least one activation record active.

The operations permitted on full function values are intuitive and parallel (for the most part) those permitted with function pointers. (You can even cast one full function value to another.) However, the two mechanisms are mutually exclusive. In summary, you can find the full function value of a nested or non-nested function. You can take the address of a non-nested function but not that of a nested function.

Another aspect, not demonstrated here, is that any label declared in a function is visible to all its subordinate nested functions. (Any labels of the same spelling in nested functions temporarily hide the outer label.) According to the High C documentation, "Jumping to an ancestor's label ... is a disciplined from of C's `setjmp/longjmp` and comes from Pascal."

Extended Number Syntax

Floating-point and integer constants may include underscore ('_') characters among the digits. For example:

```
#include <stdio.h>

main()
{
        printf("%d, %f\n", 1_234_567, 9_876.123_456);
}

1234567, 9876.123456
```

Underscores may be used anywhere in the digit sequence, *except* at the beginning so identifiers such as `_123` continue to be recognized correctly.

The `pragma` Keyword

High C supported the notion of pragmas before ANSI C invented the `#pragma` directive. The syntax used was to have a `pragma` keyword instead. While both the keyword and preprocessor directive are supported, the keyword form is permitted inside macro definitions, thus solving one of the biggest objections to `pragma` directives.

`enum` Representation

According to ANSI C (§3.5.2.2 page 62, lines 40–41), "Each enumerated type shall be compatible with an integer type; the choice of type is implementation-defined." While most implementations make all enumerated types the same size (that of an `int`) the standard permits different types to have different representations so you can save on storage. 'Implementation-defined' simply means you must document how you do it.

To take advantage of this, it would seem reasonable to have a compiler option or pragma to chose either 'smallest possible representation' or '`int` representation.' Instead, High C gives semantics to `short` and `long` when applied to enumeration types, as follows:

```
#include <stdio.h>

short enum t1 {red1, blue1, green1 = 5};
short enum t2 {red2, blue2, green2 = 500};
short enum t3 {red3, blue3, green3 = 50000};
short enum t4 {red4 = -1, blue4, green4 = 50000};
short enum t5 {red5, blue5, green5 = 5000000};
```

```
main()
{
        printf("sizeof(enum t1) = %u\n",sizeof(enum t1));
        printf("sizeof(enum t2) = %u\n",sizeof(enum t2));
        printf("sizeof(enum t3) = %u\n",sizeof(enum t3));
        printf("sizeof(enum t4) = %u\n",sizeof(enum t4));
        printf("sizeof(enum t5) = %u\n",sizeof(enum t5));
}
```

```
sizeof(enum t1) = 1
sizeof(enum t2) = 2
sizeof(enum t3) = 2
sizeof(enum t4) = 4
sizeof(enum t5) = 4
```

The `short` modifier directs the compiler to store the enumerated type object in as small an integer object as possible. In the case of `t1` all values fit into a `char`. Similarly, `t2` will fit into a `short`, `t3` requires an `unsigned short`, and `t4` and `t5` require an `int`.

Calendar of Events

- March 4–5, 1991 **Numerical C Extensions Group (NCEG) Meeting** – Location: Norwood, Mass. Analog Devices is the host. The fifth meeting will be held to consider proposals by the various subgroups. It will now **precede** the X3J11 ANSI C meeting being held at the same location (see below) and will run for two full days. For more information about NCEG, contact the convenor Rex Jaeschke at (703) 860-0091 or *rex@aussie.com*, or Tom MacDonald at (612) 683-5818 or *tam@cray.com*. Thinking Machines is hosting a reception on the Tuesday night at their Cambridge offices where their (massively parallel) Connection Machine will be demonstrated.

- March 6–8, 1991 **ANSI C X3J11 Meeting** – Location: Norwood, Mass. Analog Devices is the host. This three 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*. Note that this meeting now follows NCEG and is scheduled for three days instead of two.

- March 11–15, 1991 **ANSI C++ X3J16 Meeting** – Location: Nashua, New Hampshire.

- May 13–15, **ISO C SC22/WG14 Meeting** – Location: Tokyo, Japan. Contact the US International Rep. Rex Jaeschke at (703) 860-0091 or *rex@aussie.com* or the convenor P.J. Plauger at *uunet!plauger!pjp* for information.

- June 13–14, 1991 **First ISO C++ Meeting** – Location: Lund, Sweeden.
- June 17–21, 1991 **ANSI C++ X3J16 Meeting** – Location: Lund, Sweeden.
- June 24–28, 1991 **ACM SIGPLAN '91 Conference on Programming Language Design and Implementation** – Location: Toronto, Canada. The conference seeks original papers relevant to practical issues concerning the design, development, implementation, and use of programming languages.
- August 12–16, **International Conference on Parallel Processing** – Location: Pheasant Run resort in St. Charles, Illinois (near Chicago). Submit software-oriented paper abstracts to Herbert D. Schwetman at *hds@mcc.com* or by fax at (512) 338-3600 or call him at (512) 338-3428.
- August 26–28, 1991 **PLILP 91: Third International Symposium on Programming Language Implementation and Logic Programming** – Location: Passau, Germany. The aim of the symposium is to explore new declarative concepts, methods and techniques relevant for implementation of all kinds of programming languages, whether algorithmic or declarative. Contact *plilp@forwiss.unipassau.de* for further information.
- September 23–25, 1991 **Numerical C Extensions Group (NCEG) Meeting** – Location: probably in the Washington D.C. area.
- November, 1991 **ANSI C++ X3J16 Meeting** – Location: Toronto, Canada.
- December 11–13, **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Tentatively in Milan, Italy.

News, Products, and Services

- **NIST** (the National Institute of Science and Technology) **has selected a C validation suite for US Government acceptance testing. The winner was Perennial.** (See *Volume 1, number 2* of *The Journal* for information on this and other validation suites.)

Perennial
4699 Old Ironsides Drive
Suite 210
Santa Clara, CA 95054
USA
(408) 727 2255

The processing of a **FIPS C** (Federal Information Processing Standard) is well under way. The public comment period on the proposed FIPS closed on August 9. Written comments received are part of the public record and may be viewed and copied in the Central Reference and Records Inspection Facility, room 6628, Herbert C. Hoover Building, 14th Street NW, Washington DC, 20230.

The FIPS C standard is effective six months after the date of publication of the final document. At that time, a 1 year transition period begins to allow industry to produce C processors conforming to the FIPS standard. Interpretation requests of the FIPS standard are to be directed to NIST for processing. For further information, contact Ms. Kathryn Miles on (301) 975-3156 or L. Arnold Johnson on (301) 975-3247.

Since BSI (the British Standards Institute) and several other European National standards bodies chose the Plum-Hall suite, it is expected there will be some sort of mutual recognition of each others' suites.

- A new release, version 1.5, of the SRC **Modula-3** compiler and runtime are now available. This is the third public release of SRC Modula-3. The system was developed at the DEC Systems Research Center. It is being distributed in source form (mostly Modula-3) and is available for public ftp. You must have a C compiler to build and install the system. Contact Eric Muller at *muller@src.dec.com* for more details.

- The third edition of the very popular book **C: A Reference Manual** by Harbison and Steele is now available from Prentice Hall ISBN 0-13-110933-2. This edition has been revised to include the final ANSI C standard and exercises.

- **Lattice, Inc.** has announced it has significantly scaled back its C compiler activities and in future will provide maintenance releases only. Sales and technical support will continue. (708) 916-1600.

- **LPI** now offers colleges and universities a 50% discount on its compilers and related development tools and 25% on additional sets of documentation. Updates to educational users are also discounted at 50%.

- **MetaWare, Inc.** has announced the availability of two new products: Globally optimizing High C compiler for Extended DOS 386/486 V2.3 and a new 32-bit source-level debugger V1.0. Steve Noonan (408) 429-6382.

∞