The Journal of

C Language Translation

Volume 3, Number 3

December, 1991

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

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

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

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

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

Contents

18. Character Sets and C

P.J. Plauger

Abstract

One of the great strengths of C is its ability to manipulate characters flexibly and efficiently. A well-constructed C program can do so with no presumptions about the actual codes used to represent individual characters. Nevertheless, C has always imposed certain constraints on how it represents characters internally.

There has been a recent upsurge in standardization efforts for large "universal" character sets. That activity has instigated a certain amount of criticism of the constraints expressed in the C Standard.

This paper addresses that criticism. It explains the historic origins of each of the constraints imposed by C on character encoding. It justifies their continued presence. And it shows ways in which the proposed large character sets can coexist happily with C programs.

Introduction

Turf wars among standards organizations are inevitable. Where one standard meets another, you have an interface. A standard itself is an interface between producers and consumers—it defines the "treaty point" that both can agree on. Hence, each organization is in the habit of devising interfaces to resolve conflicts. It doesn't like to be told by another authority how to do so.

Programming language standards must coexist with character set standards. For one thing, you write a program source file in some character set. For another, the programs you run manipulate character sets. The C Standard is more ecumenical than many others in this regard:

- Unlike Ada, it doesn't mandate ASCII or any other particular character set for its definition.
- Unlike almost all other languages, it clearly separates source and execution character sets.

Nevertheless, the C Standard has struck sparks off other standards committees more than once over character sets. At issue is what a programming language can say about the character sets it manipulates. Languages are the province of SC22 within ISO. Character sets are the province of SC2. That

elevates many discussions to SC22, at least. That committee is the father of WG14 within ISO and a sort of step-father to X3J11 within ANSI. In some cases, the squabbles have attracted the attention of JTC1, the grandfather of international technical committees.

At the March 1991 meeting of WG14 in Tokyo, we reviewed the conflicts that have arisen between the C Standard and character set standards. That exercise was to help formulate a response to a gripe from the group developing the ISO 10646 standard. Since then, that gripe has become largely irrelevant. But at the September 1991 SC22 plenary in Vienna, a new set of issues arose. In some ways, these new issues are a more fundamental challenge to the way C does business.

I think it appropriate, therefore, to review what C has to say about character sets. Of particular interest is where others have challenged our right to impose constraints. In the coming months, I believe that as many people as possible should be educated on the underlying principles behind the issues at hand.

The constraints that follow appear in roughly chronological order. The first appeared earliest in C. The last was the latest concept to be clarified within X3J11.

Older Constraints

1) A text line is terminated internally with a new-line character `'\n'`, regardless of the external representation of line terminators.

The convention of terminating text lines with a new-line character is, of course, an early UNIX-ism. That operating system pioneered the notion that text within a computer should take a standard form. Handlers for different devices could and should map between this standard form and the peculiar requirements of the device.

UNIX actually uses the ASCII line feed character for `'\n'`. That character positions to the next line without returning the carriage to the start of the line. (The ASCII carriage return character `'\r'` is supposed to handle the latter operation.) New-line combines these two operations. EBCDIC has a specific new-line character but ASCII does not.

UNIX appeared, in the early 1970s, at a time of great confusion among terminal and printer manufacturers. To say that there was no standard text format is an understatement. Rather, there was active competition among different vendors to evolve the most flexible forms. UNIX found a useful common denominator with its new-line-terminated text lines. That went a long way toward making software tools work together on common files and in pipelines.

Those of us who moved C to other operating systems generally honored this convention. We often had to translate new-lines within the I/O library instead of out in device handlers. The effort was worthwhile. It let us continue to use a uniform notation that had proved useful under UNIX. It also greatly simplified moving code among systems. An early concern was moving code from UNIX.

But later, we found the uniform convention equally useful in making C code portable across diverse systems.

Somewhere along the way, however, the character-set curators had a change of heart. They decided that the ambiguous use of the ASCII line feed character should stop. People who made terminals and printers were put on notice. It was no longer considered acceptable to treat line feed as a synonym for new-line. X3J11 got a similar communique. We were told that our use of `'\n'` as a combined line feed and carriage return was no longer acceptable.

We replied that our use of `'\n'` isn't under their jurisdiction. What happens outside a C program is their domain. Implementors of C compilers have long recognized the need to match external conventions for terminating lines. Generating or reading an ASCII sequence of characters is just one such convention that we gladly honor. But what happens inside a C program is our business. And we can point to decades of usage to justify our particular internal convention for terminating lines.

I assume the complainants bought this argument. I don't recall any further communication on the subject.

2) Code value zero is reserved for the null character `'\0'`.

In the older six-bit codes, a common meaning for code value zero was the digit `'0'`. Another common meaning was the space `' '`. With only 64 codes, one couldn't afford to be generous. I don't know of any eight-bit codes, however, that use code value zero as other than the null character.

The C Standard introduced the formal notion of a "basic C character set." This set contains all the characters you need to write a C program. Throw in the null character and you have all the characters guaranteed to be representable in the execution character set. That null character has to be distinguishable from any of the others in the basic C character set.

It is no surprise that ASCII is closely related to the basic C character set. That was the character set used for the first few implementations of C. ASCII includes a few additional printing characters and a slew of control characters. None of these has code value zero. Not even EBCDIC has committed this particular sin against C.

Nobody has complained about this particular constraint simply because nobody's ox has been gored. If an important eight-bit character set exists that gives some serious meaning to code value zero, I don't know about it.

3) A binary file may have any number of null characters appended to it.

The very first version of `getchar` returned zero to signal end-of-file. Null characters seldom, if ever, occur in text files. It seemed like a reasonable thing to do. What people soon found, however, was that many programs could process more than text. They worked just as well for binary files—streams of arbitrary text.

It was unfortunate that `getchar` was *almost* transparent, but not quite. Before very long, the end-of-file code became −1 and `getchar` grew up. It was then easier to write utilities that process arbitrary text. The null character was just another character.

UNIX programmers take it for granted, in fact, that an arbitrary file is binary transparent. Whatever you write to a file you get back when you read it. No more, no less. True, an occasional exception exists. Certain "block physical" files exist only in multiples of 512 bytes. Write one byte to a magnetic tape, for example, and you could read back that byte followed by 511 null characters. But then, only specialized programs typically muck with such files.

On systems other than UNIX, this exceptional behavior is often the rule. Many systems expose the underlying block structure of files to every program. For a text file, the system usually superimposes some structure. It knows when you've read the last written line, even if rubbish follows it in the underlying file. The C runtime library can use this information to read back (almost) exactly the same text that was written to the file.

But for a binary file, it's harder to impose any sort of structure and still retain a desirable degree of transparency. Your typical C runtime library wants to muck with an arbitrary binary file—that's the very spirit of C. That means it's hard to hide the padding that files often contain. About all the runtime can do is ensure that the padding is null characters (bytes with code value zero), not arbitrary garbage.

The C Standard simply acknowledges the obvious and unavoidable. It weakens the UNIX I/O model inherited by C. A portable C program must assume that any binary file can acquire one or more null characters at the end. The wise programmer learns to skip such nulls. Even better, the wise programmer learns to mark the end of a binary file unambiguously for later readers.

X3J11 received a mild rebuke about this new wording. It seems that "NUL padding" is a concept in text transmission with a specific connotation. Synchronous channels can send ASCII NUL characters as place holders in lieu of actual characters. We had no business dictating how and when a NUL should appear outside a C program.

We sidestepped the problem, as I recall, by a slight change of wording. We rewrote any reference that might be construed as a reference to ASCII NUL. Instead, we talked about "null characters," a term with a language-specific meaning in the C Standard. Another small turf war fought to a draw.

4) `'0' + 9 == '9'` for any character set.

X3J11 saw fit to add one final pronouncement about conventional (typically eight-bit) character sets. It constrains the code values assigned to the ten decimal digits. The C Standard requires that the digits from `'0'` to `'9'` be assigned sequential code values in any character set.

You can well imagine why a typical programmer would *want* this constraint to apply. Converting between printable text and encoded integers is simpler

and faster if you can write code such as:

```
sum = sum * 10 + digit - '0';
```

You don't want to have to write the more cautious:

```
static const char digits[] = "0123456789";
int val = (const char *)strchr(digits, digit) - digits;

sum = sum * 10 + val;
```

At issue is whether the language *should* promise such a constraint. After all, many programmers write code that assumes:

- `'a'` through `'z'` and `'A'` through `'Z'` each form contiguous intervals (true in ASCII but not EBCDIC)
- `'A'-'a'` equals `'X'-'x'` – the cases are a fixed distance apart for all letter pairs (true in both ASCII and EBCDIC)

The C Standard does *not* promise that either of these constraints applies. The wise programmer knows to use functions such as `isalpha`, `toupper`, and `tolower` in a portable program.

Still, X3J11 felt that the digits were a special case. Speeding numeric conversions was deemed an important goal. And nobody could conjure up a character set where the constraint failed to hold. Thus, the committee endorsed an age-old shortcut.

As far as I know, no complaints have arisen on this subject. I'm sure that's once again due to a dearth of gored oxes. The constraint holds for both ASCII and EBCDIC.

Newer Constraints

X3J11 added support for large character sets such as Kanji, Chinese, Korean, and Arabic. A character set occurs in two forms:

- multibyte characters, each consisting of a sequence of one or more conventional (one-byte) codes
- wide characters, each consisting of a distinct code value of some integer type

The committee made a point of not mandating ASCII or any other conventional code set. We weren't about to mandate a specific encoding of, say, Kanji—a subject that most of us knew precious little about. Still, we had to impose some constraints on the large character sets that C must manipulate.

5) Code value zero is reserved for the null wide character `L'\0'`.

C has had considerable success with null-terminated text strings. It is a safe bet that wide-character strings will achieve comparable importance in programs that manipulate large character sets. The obvious thing to do is reserve code value zero for a null wide-character terminator as well.

The C Standard currently includes only the bare minimum support for large character sets. Five functions inspect these encodings and convert between the two forms. Yet almost every one of these five is influenced by the presumption that `L'\0'` terminates wide-character strings. The Japanese delegation to WG14 has proposed a much richer assortment of functions to manipulate wide characters. Naturally enough, they too rely on this presumption.

None of the wide-character encodings that I know of has trouble setting aside code value zero this way. That is hardly a surprise. Much of the experimentation that led to large-character support was performed in C. You can be sure that the experimenters were aware of the uses for null-terminated string. I expect no conflicts in this particular area.

6) `L'x' == 'x'` where `x` is any member of the basic C character set.

It seems only sensible to provide a simple mapping for all the members of the basic C character set. Each must be representable as a single-character multibyte sequence (at least in the initial shift state). Each must also have a representation as a wide character. What better code value for a wide character than its value as a single character?

Of course, reason and standards often maintain only a loose acquaintance. It helps that the principle Kanji wide-character encodings obey this rule. But it also turns out that the constraint is rather necessary at the moment.

When I implemented the Standard C library[1] last year, I faced a problem. I could see no clear way to scan the format strings for the print and scan functions, such as `printf` and `scanf`. The C Standard requires that a format string be a multibyte string. The string begins and ends in the initial shift state (if shift states matter). It can contain embedded conversion specifications, each beginning with a per cent character `'%'`.

Obviously, you must use `mbtowc` or `mbstowcs` to parse the format string. Either of those functions gives you the sequence of wide characters corresponding to the multibyte string. How do you know when you encounter a per cent? Just compare each wide character against `L'%'`. That part is easy.

Scan formats give special meaning to sequences of white space, however. That means you must be able to spot a white space character in its wide character form. Any character for which `isspace` returns nonzero in the current locale is a legitimate white space character. But, you cannot contrive a corresponding list of wide-character constants ahead of time.

The solution I arrived at is to test each format wide character `wc`, as in:

[1] *The Standard C Library*, Prentice Hall, 1992, by P.J. Plauger.

```
if (wc <= UCHAR_MAX && isspace(wc))
        /* must be white space */
```

For this code to be portable across all locales requires an even stronger statement than the one I made above. The constraint `L'x' == 'x'` must apply for *any* single-character multibyte sequence. That is a superset of the basic C character set. X3J11 may have failed to say this explicitly, but I believe that is what we meant. I also believe it is what we have to say.

Unfortunately, others disagree. I alluded earlier to a clash between the C Standard and the proposed ISO 10646 code set. Seems that ISO 10646 can represent all characters as four-byte sequences. The proposed way to represent, say, the letter `'a'` is with three spaces followed by the code for `'a'`. That is in conflict with the C Standard in two different ways:

- In any multibyte code, `'a'` should be represented as a single character, at least in the initial shift state.
- In any wide-character code, `L'a'` should have the same value as `'a'`.

Those proposing various codes for large character sets haven't been clear about how they are to be used, or so I believe. The distinction between multibyte codes and wide-character codes is better appreciated among C programmers. Even the possibility of various shift states in multibyte codes is not widely appreciated.

Hence, it is not always clear how X3J11 should respond to a claimed conflict. We could argue various ways:

- Three spaces followed by `'a'` is a fine way to represent `'a'` as a multibyte sequence not in the initial shift state. C simply requires another way to write `'a'` in the initial shift state (simply as `'a'`).
- A wide-character code is an internal matter to a C program. Try to write multibyte integers with one program and read them with another and you encounter all sorts of byte ordering problems. Thus, it is inappropriate to complain that `L'a'` doesn't fit this code scheme.

I depend heavily on this second argument in *The Standard C Library*. The functions that map between multibyte sequences and wide characters can handle only a limited degree of mapping. They must have license to define the internal codes for wide characters.

We discussed a response to this conflict with the proposed ISO 10646 code at the March 1991 meeting of WG14 in Tokyo. The response is now largely academic, however. In the interim, that proposal has been voted down within SC22. Taking its place is UNICODE, a large character set with the active backing of many in industry.

And that brings us to the last constraint.

7) The null character can appear only as the first and only character in a multibyte character sequence.

If you like null-terminated strings in C, you have to like this constraint. Basically, it promises that multibyte strings can also be treated simply as null-terminated strings for many purposes. You need not parse the string into separate multibyte sequences. The first null you encounter is the null character that terminates the entire string. Thus, you can use many existing functions, and techniques, for manipulating multibyte strings much of the time.

Unfortunately, UNICODE violates this rule—when used as a multibyte code, that is. The code for letter `'a'`, for example, is three null characters followed by the code for `'a'`. That makes it a wonderful candidate for a wide-character code. Sadly, that's not how its proponents seem to be promoting it. The issue came up before the SC22 plenary in Vienna in October 1991. Tom Plum and Keld Simonsen represented WG14 at that meeting. Both the C and POSIX folk urged that the current structure of UNICODE be reconsidered if it is to be used as a multibyte code.

I am told that the reception was not cordial. The prevailing attitude seems to be that C is just one language, and a rather old one at that. C programmers should learn new techniques for manipulating text strings. That's better than altering a universal character set for just one peculiar pattern of usage. I believe differently. From my perspective, the C community has much to teach the rest of the world about manipulating text. We certainly pioneered a new and very powerful approach to the subject. We deserve to be heard on a subject that we have handled so well.

Conclusion

None of the clashes between C and character set standards is a life or death issue. In the past, we have solved them by drawing clearer boundaries between jurisdictions. Our job for the present and immediate future is to draw additional boundaries. It is also to participate in the formation of standards that are still malleable. An important part of this participation is to share what we know about manipulating encoded text. We have done a lot with very few constraints.

We can all learn to live with embedded nulls. When we manipulate arbitrary binary data, we already do. But we also know how convenient it can be to use null terminators when it is safe to do so. It would be a shame if we were to lose a useful programming practice simply because we failed to educate others.

P.J. Plauger serves as secretary of X3J11, convenor of the ISO C working group WG14, and Technical Editor of The Journal of C Language Translation. His latest book, The Standard C Library, *is available from Prentice-Hall. He can be reached at* uunet!plauger!pjp.

∞

19. Parasol: A C-Based Systems Programming Language, Part I

Bob Jervis
17645 Via Sereno
Monte Sereno, CA 95030

Abstract

This paper discusses a new systems programming language called Parasol that is derived from C. New features are described including support for incremental compilation, object-oriented programming, and parallel programming. Each feature is compared with C, and a description is given as to why that feature was adopted.

Introduction

Parasol (Parallel Systems Object Language) is a new systems programming language designed to keep much of the original flavor of C while incorporating parallel and object-oriented features and some changed syntax. This article describes the syntax changes, why they were made, and the implications for clarity, compatibility, and compilation techniques.

I began the design of Parasol in 1989 with three goals in mind: add simple object-oriented extensions, make incremental compilation easier and preserve the efficiency of generated code. You might have seen information about this language under the name of OPAL, but as that name was already being used for another language, my language has been renamed Parasol. Earlier this year I reviewed the project and concluded that for Parasol to be successful, it must anticipate the needs of future hardware. As a result I added a fourth goal of including simple parallel programming constructs.

I have implemented a compiler targeted to generate 32-bit Intel 80386 code. Using this compiler I have created a network operating system, called ALYS, and over thirty application programs written in Parasol. Altogether over 60,000 lines of code have been written using Parasol. I am now finishing a new version of the compiler that incorporates the latest parallel additions to the language.

Upward-compatibility with existing C code was *never* a strong consideration. This was intended at first to be a research project, so I was free to make whatever changes were needed to simplify my work. Many of the changes I made were also motivated by the desire to combine some of the best features of

other languages that have emerged in the last twenty years. Even though preserving code was not important, I did take into consideration how programmers might feel about a change or feature. I do want Parasol to find an audience and, with luck, a user community.

One of my original thoughts was to use Parasol to learn how to do a powerful incremental C development environment. If Parasol drifted too far from C, then any lessons I learned would have no value when I transferred them back to C. For example, I toyed with the idea of doing a bytecode interpreter for Parasol. I ruled that out because I wanted an incremental environment that (like a C compiler) generated production quality code.

As time has gone by Parasol has drifted from C. As I worked more and more on the problems of object-oriented programming, and as I recognized the coming importance of parallel programming, it became clear to me that C was not the best base to work from. Parasol is now a language that is addressing a very different hardware environment than C was intended to use.

Like C, Parasol is still primarily a systems programming language. I can see that there is real potential for making Parasol a good scientific language as well, but that is a secondary concern. A programming language cannot be everything to everyone. While I have made some changes that improve the ease of writing scientific programs in Parasol, I must leave to someone else most of the problems of creating high quality math libraries or adding extensions such as complex arithmetic.

Incremental Compilation

Incremental compilation is an easy technique to describe. An incremental compiler stores intermediate results of a compile session to avoid work in future compiles. Traditional compilers use incremental techniques. Object files are a good example of intermediate information that helps avoid work. Only those sources that were actually changed (or that included headers that were changed) need to be recompiled. Almost all C compilers therefore use some incremental compilation techniques.

The term incremental compilation is usually reserved, however, to describe a compiler that keeps enough information around to be able to detect changes to a fine degree of granularity. Instant C[2] is an example of a compiler capable of detecting changes to individual functions and rebuilding only what is necessary.

I believe that incremental compilation techniques can be used in a very efficient programming language. To guide my work, I have used a goal of one second turn-around after changing the body of a function before the program is rebuilt. So far, I have been able to build a test compiler that rebuilds a sizable program in a few seconds.

I have been extremely surprised at how difficult it has proved to produce a stable incremental compiler. The information contained in a Parasol program,

[2] Instant C, from Rational Systems, is a C development system for DOS-based PCs.

like that in a C program, is quite extensive. What makes it so difficult is the important role that type plays. Since the type of an object can change independently of the code that uses the object, changes in a declaration must be propagated to all affected code. In an incremental compiler, you want to be sure that you minimize the amount of unaffected code that gets rebuilt as well. Devising a procedure that is correct, fast, and doesn't waste large amounts of memory is very difficult.

C was designed to be compiled module by module. Languages like Pascal or Algol that had no provision for modules were intrinsically limited. Parasol incorporates a number of changes that were intended to make incremental compilation easier.

The C Preprocessor and Parasol

I decided, almost from the beginning, that Parasol would *not* have a preprocessor. I felt that implementing one would have a significant impact on the complexity of the compiler. Having written a preprocessor that was integrated into a C compiler, I thought an incremental preprocessor would be extremely difficult to implement in full generality.

Used in a limited and well-structured way, the preprocessor is not a crushing problem. However, a compiler must support any valid use of the preprocessor, and valid uses of macros and even headers allow for all sorts of constructions that obscure the true structure of a C program.

I decided that if I ever wanted to do a full incremental C compiler, I could tackle the preprocessor after I understood all the other problems of incremental compilation.

Inline functions, enumerators, and compile-time constant objects in Parasol replace many of the "legitimate" uses of the preprocessor. There remain at least two reasonable uses that I have not replaced: short-hand macros for complicated static initializers (e.g., for large reference tables), and conditional compilation.

The use of macros in initializers can sometimes be replaced with runtime initialization using function calls, but that somehow seems inefficient. Below I will discuss the issue of initializers in more depth.

Conditional compilation within a function body can be expressed (though with some loss in flexibility) by the use of `if` statements with constant test expressions. Of course, it may be difficult to detect whether the test is intended to be a constant or not. If you accidentally code a test that is always true (like comparing a character variable with a large constant), the compiler should warn you about it. But, if you intend to conditionalize some code, the warning shouldn't be given. How do you tell the cases apart in general?

A significant use of conditional compilation is in removing unneeded members from structures or in configuring the types of variables depending on environmental considerations, and `if` statements can't help these cases.

In production shops, code becomes progressively more conditionalized as maintenance continues. Clearly, if Parasol is to be used in such environments,

conditional compilation must be reintroduced. Whatever syntax is used, there are certain semantic issues that must be addressed. The C preprocessor avoids certain problems by simply mandating that conditionalized code is processed from top to bottom. The order that you compile modules doesn't affect how each module is conditionalized.

In Parasol the constants used to conditionalize code are embedded in the same code that is being conditionalized. The order in which code is compiled may affect the results. Let us look at an example in C:

```
/* Header A */

#ifdef X
#define Z 0
#endif
#define Y

/* Header B */

#ifdef Y
#define Z 1
#endif
#define X
```

The value of Z is different depending on which header is included first. Parasol doesn't have headers, but public constants allow for an equivalent situation to be constructed. In C, you control the order of headers, but in Parasol the order of *units* is deliberately left unspecified.

Parasol Program Source Entry

One of the early decisions taken in the design of Parasol related to the issue of incremental compilation. I had to decide whether Parasol programs were to be written in the form of text files or whether some sort of structured program entry was to be used. If you want to do incremental compilation, a structured approach makes many operations easy. For example, detecting where changes have been made within a large program becomes much easier.

When you want to rearrange the order of declarations in a source file, add a new declaration, delete an old one, or change the name, type, or initializer of a declaration the compiler can in a structured environment demand that you specify exactly which operation you are performing. With text files, the compiler must guess what happened.

On the other hand, by using text files rather than a structured environment, there would be a better chance that a more traditional Parasol compiler could be constructed. Using text files also would allow programmers to use their favorite text editor to prepare a program.

There was also a bootstrapping problem. Since I wanted to have the Parasol

compiler written in Parasol, I needed some version of the language available as quickly as possible. This first compiler would have to be written in C. I decided that it would be easier if I built a conventional compiler structured like a C compiler (which I understood very well), but with a Parasol parser. In fact, the first Parasol compiler used a C runtime library with only slight modifications and linked using a conventional linker.

Units

Parasol programs are divided into *units*. Each unit is equivalent to a C module, with a single source file acting as the equivalent of a .c file. Parasol incorporates an `include` statement that is approximately equivalent to C's `#include` directive.

Parasol units differ from C modules in a few respects. In C, the programmer creates headers separate from the C source files. Typically, declarations of structures and manifest constants are placed in the headers while functions are placed in the main source files. In programs written using Standard C, prototypes for functions are placed in headers to provide compile-time checking of function calls. If a function returns an `int` in C, and the programmer doesn't care about argument type checking, a function doesn't need any declaration at all.

In Parasol, structure, constant, and function declarations are all placed in the unit source file. Any symbols declared to be public in one unit will be available to any unit that explicitly includes it.

In C, there is no fixed relation between a header file and the various modules of a program. It is possible to combine the function prototypes for several modules into a single header, or even to include explicit `extern` declarations in a module source file itself. C compilers almost always use a separate linker (which is used to link assembly language modules as well). Any `extern` reference can be resolved by a public symbol defined in any module.

The separate link phase means that `extern` references can be generated to non-existent (or misspelled) functions and objects. These linkers typically do not check that the types of the references and definitions match. Of course, C++ uses name mangling as a way to avoid this sort of error, but C compilers usually don't mangle names. As a result, link-time errors are a significant source of problems in C programs. The addition of prototypes in Standard C has gone a long way towards locating such problems.

Parasol simply does not have link-time errors. As a Parasol unit is compiled, the actual declarations of public symbols are used to check the validity of references. Separate function prototypes do not exist in Parasol—the information is derived directly from the function definition itself.

Other languages that use units, like Modula-2, distinguish between the interface part and the implementation part of a unit. This allows a Modula-2 compiler simply to scan the first half of a module source to locate the interface definitions. Parasol units, on the other hand, are not so divided. A programmer

simply writes the object, type, constant, and function definitions for a unit in whatever order seems logical. The Parasol compiler then scans the entire source to extract the interface information.

The unit structure of Parasol programs has a profound impact on how a Parasol compiler must be constructed. I had decided that since memory was cheap, I would compile a Parasol program entirely in core. As I designed how units were to work, the desire to compile everything in core became transformed into necessity. In pathological cases, it may be necessary to load the symbol table for every unit in a program in order to finish compiling a single unit, even if they all weren't explicitly included.

Let me explain this last point a little more carefully. While it is mostly true that public symbols are only available if they are included, there are ways that information can migrate from units that were not explicitly included. This state of affairs came about as the language and compiler evolved.

When I first implemented units in Parasol, the compiler didn't build everything in memory and had to load unit symbol table files as needed. This compiler only loaded symbol table information for units appearing in include statements. This led to some interesting problems as I began converting the Parasol compiler from C to Parasol.

The following (written using pseudo-C syntax so it will be easier to recognize what is going on) occurred in the compiler source:

```
/* Unit a */

typedef struct {
        char *name;
} symbol;

/* Unit b */

include a;

symbol *Function;

/* Unit c */

include b;
include a;          // needed in the early compiler

f()
{
        printf("%s\n", Function->name);
}
```

In unit `c`, the function `f` needs to refer to a member of the global variable `Function`. Since the `name` member is defined in unit `a`, the early Parasol com-

piler required that unit `a` be included to refer to it. While there might be some justification for this example, it seems rather awkward. It also undermines the encapsulation of `Function`. In order to use the variable `Function`, you need to know not only where it is defined, but some detail about how it is defined. If you were to change the location of the definition of the `symbol` type, you would have to change the `include` statement in unit `c`.

When inheritance was added to the compiler, the problem of encapsulation became even more acute. Suppose in the above example, the `name` member was not defined with `symbol` at all, but inherited from some base type defined elsewhere. Now, you would have to know intimate details of the type hierarchy. Such a situation would be very inconvenient and would make documenting interfaces fairly cumbersome. Changes to the type hierarchy might affect what units you have to include, even though the changes are not directly related to the code in unit `c` itself.

I changed the specification of Parasol so that when you refer to structures, you only need to include the unit defining the object you are using. The structure member definitions are extracted from whatever additional units are needed.

Another aspect of the unit structure had to be changed as I gained experience with converting the Parasol compiler. The early versions of Parasol prohibited units from mutually including each other. If you think of the units of a Parasol program as forming a graph, with the units acting as nodes and each include statement acting as an edge from the includer to the included unit, the original language specification required that the unit graph be acyclic. This meant that the compiler could systematically compile each unit completely, before proceeding to the next unit. An acyclic graph guaranteed that a compilation order could be found that built the program correctly.

This rule proved to be something of a limitation. For example, any compiler has a very complex internal structure in its symbol table. I tried to divide the code of the compiler so that one module dealt with symbol records, another with type records, and so on. The problem was that these various records pointed to each other in a tangled web of lists and back-pointers.

The only ways to write these definitions in Parasol interfered with a modular decomposition of the compiler. Two structures within the same module could point to one another because the compiler deferred resolving the first pointer encountered until the whole module was parsed and the second structure was found. Not so if the two structures were located in different units. One unit must be compiled to completion before the other. The first unit to be compiled could, of course, use a void pointer to store the link to the second structure, but that is a very poor solution to the problem.

By combining the declarations of the various symbol table records into a single unit, I was able to use typed pointers on all links. The modules that were responsible for each separate record still existed and they simply included the data structure unit. Unfortunately, this was not a very satisfactory solution from the perspective of object-oriented programming. One design issue that I

felt very strongly about was that methods should be defined within the body of the type to which they belonged. Making the Parasol symbol table object-oriented meant that the one data structure unit would have to be stuffed with nearly two dozen types, each with multiple methods. Since I generally dislike large source files, this was disappointing.

As I progressed in converting the compiler, I found still more problems. The Parasol parser I had written was a recursive descent parser in C that spanned several C modules. Just as in a C compiler, in Parasol there are several places where the parser uses recursion, for example, to describe array bounds expressions embedded in a declaration. The declaration itself could be embedded in a cast expression.

Since these parsing modules were large already, I didn't want to merge them together into a mega-unit. Instead, I introduced a couple of function pointers into some low-level units. At program start-up, these pointers were initialized to point to the parser entry points that I couldn't refer to directly. Again, this solution seemed very bad. Recursion is a valuable concept to describe certain kinds of problems (such as parsing) and it seemed that the unit structure was getting in the way of the natural way to write the program.

As a result of these experiences, I have further relaxed the specification of Parasol so that units can include each other in arbitrary ways. Cycles can occur in the include graph and the compiler simply must cope with them. Of course, there are cycles of references that are still disallowed. For example, the following is invalid whether the definitions are in one unit or two:

```
struct a {
        struct b b_member;
};

struct b {
        struct a a_member;
};
```

These are invalid in C as well—it is impossible to determine a size for the two structures.

The changes I have made in units as Parasol has evolved have been designed to reinforce the primary role of units, while reducing unwanted side-effects. Units exist lexically to encapsulate parts of a program. You should not be constrained in how your program executes because of unit boundaries. Your choice of data structures also should not significantly impact how you draw unit boundaries.

In many discussions of object-oriented programming, modules and units do not play a significant role. The type or class hierarchy is given center stage. While that may serve some needs, I believe the use of modules remains an effective tool in decomposing a large problem into manageable pieces and in minimizing the hidden interactions among those pieces.

Object-oriented languages like SmallTalk use the class as the encapsulation boundary. All classes then exist within a single flat name space. In Parasol, within a unit it is possible to define supporting types and static objects that are invisible outside the unit.

You can define, for example, a queue type that consists of a queue head and separate queue items as distinct structures. In Parasol, you can make the queue head structure be the visible type, and all queue operations are executed by calling methods in that structure. The queue item structure is known only within the queue unit itself.

Lexical Differences

Parasol and C accept virtually the same set of tokens. Parasol defines some new keywords, but otherwise there are few differences. Parasol does allow C++ style single-line comments.

Declarations

The most obvious difference between C and Parasol is in the syntax of declarations. Unlike C, where storage class and type information can be intermixed, in Parasol a declaration has a specific order to each of its elements. Parasol declaration syntax is similar in some ways to Pascal. Let's look at a simple example:

In C:

```
int  x;
```

In Parasol:

```
x: int;
```

In this example, `x` is declared to be an integer. In Parasol, the variable name always occurs first followed by a colon. That much is like Pascal, although, unlike Pascal, only one variable can be declared in each declaration.

A more complicated example will reveal the range of information in a Parasol declaration. Let us assume that you wish to declare a global named type called `funcptr`, of type "pointer to function returning pointer to `int`." Assume the function accepts one parameter of type `double`.

In C:

```
typedef int *(*funcptr)(double);
```

In Parasol:

```
funcptr: public type ref (double) ref int;
```

Let us examine the Parasol declaration more closely. The name being declared comes first and is set off from the rest of the declaration. Next is the keyword `public`. In Parasol an identifier is local to its scope unless explicitly declared to be public. The keyword `type` is a storage-class specifier that is equivalent to `typedef` in C. The remainder of the declaration is the type. The `ref` keyword declares a pointer type. This whole type is read as "pointer to function accepting one `double` argument and returning a pointer to an `int`."

Note that the `ref` keyword corresponds to the C asterisk (`*`) in declarations. This was changed because of ambiguities in cast expressions. The current Parasol compiler will accept an asterisk as a synonym for `ref` in declarations where the meaning is unambiguous. I have not tried implementing this syntax using a parser generator, but I suspect it isn't all that hard to get right.

You never need to use qualifying parentheses in Parasol type declarators because types are always read from left to right. This change profoundly simplified the parser for Parasol, compared to a parser for C. Function declarations must always declare their arguments, so empty parentheses mean a function that accepts no arguments. By not supporting old and new style function declarators and using simple left-to-right grammar rules, parsing a Parasol declaration is extremely straightforward.

It happens that searching for a declaration in Parasol source is fairly easy. If you never put space between the identifier and the colon, you can search for the identifier followed by a colon and find the declaration almost immediately.

Only one identifier can be defined in each declaration. This limitation turns out to be less of a burden than it might at first appear. A style of programming in C already is commonplace in which each variable or structure member is declared on a separate line anyway, with an accompanying comment describing the variable. However, as many C programmers declare local variables using the notation `int i, j, k;` I may add a similar capability to Parasol.

I limited declarations to a single identifier in part because of an early approach I was taking to incremental compilation. The concept was that in parsing a source file, each identifier would have a single contiguous stretch of source text that uniquely described it. In that way, a smart editor could track changes as they are made and report which identifiers were affected by an edit session. It turned out that I didn't, in fact, build such an editor. The cost of re-parsing a source file and comparing the resulting declarations turned out to be less than I initially anticipated. I may yet implement the editor tracking of source, because in very large sources the re-parsing could become slow.

Scopes

One of the principles I wanted to establish in Parasol was that the order of declarations in the source should never matter. I wanted this so that you could reorder the declarations in a file without having to rebuild anything. As a result, Parasol does not allow nor need forward declarations at all. Each identifier within a scope is declared exactly once.

The scope of a declaration always extends from the beginning of the scope itself, *not* from the point of the declaration. It is just as valid to declare all of your variables at the bottom of a block as at the top. The only issue to remember is that an initializer for an automatic variable is executed at the point of the declaration, even if the variable is referenced before that point in the block. This remains consistent with the notion in C that an automatic initializer is just short-hand for an assignment statement.

In order to keep processing of `goto` labels in line with other scope rules, I adopted the rule that `goto` labels have block scope just like any other local identifier. Since the compiler has to handle forward references to other identifiers, this didn't present much of a problem. One consequence of this new scope rule is that `goto`'s *cannot* jump into an inner block.

An interesting point here for implementors is that the new scope rules did more to change the large-scale structure of the Parasol compiler than all the changes to the declaration syntax. Syntax changes affected only the front-end parsing code and did not change how any of the later stages of processing were done. By comparison, the new scope rules not only forced changes in the cast expression syntax, they also affected how the compile process is divided into phases.

In a C compiler it is possible to do full type attribution as expressions are parsed. This is not possible in a Parasol compiler, since any identifier may be a forward reference to a later declaration in the block. A C compiler can be written to be a single pass compiler, emitting most code almost as each expression is parsed. It is sufficient to read a single function definition at a time to be able to generate extremely optimized C code.

In Parasol, code cannot be generated for a function until all the units of a program have been parsed. Parasol compilers will have to make at least two passes over the source. The first pass accumulates the public information for export between units and the second pass generates code. Some private declarations may be parsed and retained in the symbol tables for the second pass. By recording where the function bodies are located, it is possible to avoid rescanning the entire source during the second pass. By keeping the source in core, the extra scanning is reasonably fast.

Casts

In C, a cast has the type enclosed in parentheses in front of its operand. This means that when the compiler encounters any left parenthesis in an expression, it must examine the next token to decide what to do. If the next token is a type keyword, it is obvious that the parenthesis must begin a cast. When a simple identifier appears, the compiler must check the meaning of the identifier to see if it is a type name. In Parasol, the scope rules make it impossible to determine what identifiers are type names until the entire source is parsed.

Since you get a very different result depending on whether a parenthesis begins a cast or a subexpression, I had to change the cast syntax. In Parasol,

casts are written like function calls (as they are in C++). For example:

In C:

```
y = (int)x;          /* cast x to int type */
```

In Parasol:

```
y = int(x);          // cast x to int type
```

The Parasol compiler defers deciding whether this statement is a call or a cast until it can determine the type of `int` (in Parasol the standard types are *not* keywords, but are type names like user defined type names).

To cast to a pointer, use the following syntax:

```
y = ref char(x);
```

This code casts `x` to pointer to `char` type. Here the compiler can determine unambiguously that this is a cast (since `ref` is a keyword).

Visibility

For all scopes, each declared symbol has a visibility. Parasol defines three kinds of visibility: *public*, *visible*, and *private*. Public symbols can be read and written from any code that can reach the symbol. For example, a public function is reachable by any unit that includes it. A public structure member is reachable through any instance of the structure, using an arrow or dot operator.

Visible objects can be read, but not written outside the scope of definition. Thus a visible static variable in some unit may be changed by functions in the same unit, but not by functions outside the unit. Of course, you could return a pointer to the object and use the pointer to modify the object. The compiler will only check references that use the variable name itself.

A private object cannot be seen outside the scope of definition. For example, all automatic variables are private (for practical as well as syntactic reasons). Private declarations at unit scope are equivalent to C `static` functions and objects.

In Parasol, all objects are private (even structure members) unless they are explicitly declared otherwise. This is the opposite of C, where all structure members are public and all functions are global unless declared `static`. I felt that the default behavior of C was contrary to the principles of encapsulation. The public interface to a unit or object should be consciously determined, rather than simply arrived at by default.

Name Spaces

Unlike C, Parasol has no notion of separate name spaces. Parasol has no structure or union tags and no preprocessor macros. Structure members are not separated into their own name space. The only C name space that could be maintained is the `goto`-label name space. I decided that `goto` labels were not important enough to deserve a separate name space, so all identifiers must be unique within their scope of definition.

And unlike C, structure members are not in a separate name space; they are in a different scope. It is important to remember that with the object-oriented extensions, functions can appear within a structure definition. Those functions can use the structure member names as plain identifiers, as if they were simple variables. The next section will describe how the compiler determines which object's members are being referred to.

Name spaces arose in C because there were several distinct syntactic contexts that made it possible to distinguish which name space was needed, without reference to scope. In Parasol, the structure tag and macro name spaces have disappeared. The member name space in C originated in the days when you could use simple integers or character pointers on the left side of a dot or arrow operator. In those days, each member name had to be unique across all structures. The only exception was made for members that had the same type and offset.

When C converted to stricter rules of usage for member names and for the arrow and dot operators, the member name space was no longer needed. The context of the structure type appearing on the left hand side of the reference operator determined the structure of which the member had to be a part. Member names could not appear in any other way in an expression. The notion of a member name space was retained as much as anything to remind one that member names were a separate kind of identifier.

While it is conceivable that an algorithm could be written that exploits C's permissiveness about `goto` labels, the general response I've gotten when people have heard this new rule for Parasol is that it is perfectly fine. `goto`'s are such pariahs among C programmers anyway that restrictions on their use aren't terribly onerous.

Program Startup and Shutdown

C programs (in a hosted implementation) always begin execution at a function called `main`. If third-party libraries need special initialization, a function must be called at the beginning of `main` to accomplish this. The programmer must be aware of the need for initialization and must add the call. If several subsystems of the library are included, and each need initialization, the programmer must again be aware of which initializations must be done and in what order.

Many C implementations use assembly language code to map the operating system interface onto the interface for `main`. That assembly code often includes

tricks to properly initialize elements of the standard library.

The standard C library supplies the `atexit` function to provide the converse service of functions called to clean up libraries and programmer-supplied modules.

In Parasol I decided to provide a formal mechanism for scheduling initialization and cleanup functions without the need for explicit calls. Parasol defines two keywords `entry` and `cleanup` which act as storage-class modifiers on functions. These functions have no restrictions on their names and they do not have to be public.

Entry functions take no arguments and return no value. They are executed in a specific sequence determined by the pattern of includes in the various units of a program. The general rule is that the entry functions of a given unit are executed after the entry functions of all units it includes. This way, library units will execute their initialization code before the program units execute.

When building a program the programmer designates one of the units as the project unit, which serves conceptually as the top level module of the program. The top level project unit should contain an entry function that acts as the equivalent of `main` in C programs.

Cleanup functions are called from within the `exit` function, much like the `atexit` functions of C. Cleanup functions are called in the reverse order of the entry functions, so that program units can clean up before the library units they include. Since any entry function can lead to a call to `exit`, an early abort cannot safely execute all the cleanup functions. The rule here is that if a given entry function either directly or indirectly calls `exit`, all cleanup functions in included units are executed. Only those cleanup functions in the current unit that lexically precede the currently running entry function will be executed. This is an exception to the rule that lexical order of declarations does not matter. Practically speaking, this isn't much of an issue because most entry functions cannot lead to a call to `exit` and cleanup functions are not commonplace.

Cleanup functions are unlike `atexit` functions in that they accept one integer argument and return an integer value. The argument and return value are the exit code. This way, a cleanup function can choose, for example, to delete output files when `exit` was called with a non-zero exit code. Cleanup functions can also change the exit code, but such a practice should be followed with care since the exact order of execution is not completely controllable. Some cleanup functions will be called with the original exit code and others will not.

Cycles in the included units create a problem for scheduling of entry and cleanup functions. Currently entry and cleanup functions can appear in cycles, but the order of execution among the units in a cycle is unspecified. This is not a very satisfactory state of affairs. My experience has been that subtle changes in the layout of the program, adding a unit for example, can alter the execution order and that dependencies between entry functions, and can cause some pretty nasty bugs.

One Draconian solution would be to prohibit entry and cleanup functions

within a cycle. In fact, as long as only one unit in the cycle contains entry and cleanup functions there are no ordering problems. There is great value in placing initialization code in entry functions lexically associated with what they initialize. Since cycles are a necessary reality for large programs, it would be nice to be able to reliably include initialization code there.

I am currently exploring a design that would schedule entry and cleanup code based on the actual references in the functions themselves. That way, unless the initialization code involved some nasty cycle of calls, the compiler could resolve a satisfactory execution order. This becomes particularly important if I add the ability to use non-constant initializers on static storage duration objects. In effect, a non-constant initializer is a tiny entry function. Since a casual inspection of source code would not reveal which initializers are constant, it may be difficult for the programmer to know whether there is a potential sequencing problem.

Object-oriented Extensions

Parasol includes a set of simple object-oriented extensions. In Parasol, object-oriented extensions affect structures and unions. Structures in Parasol are equivalent to a class in other object-oriented languages. In this regard Parasol is similar to C++.

The following is a sample declaration of a structure in Parasol:

```
stream: public type {
        public:
        flags:          flagValues;
        bufferSize:     unsigned;
        buffer:         ref unsignedChar;
        ungetByte:      unsignedChar;
        curp:           ref unsignedChar;
        endp:           ref unsignedChar;
};
```

This is modified slightly from the stream type in the file I/O unit of the standard library. A Parasol stream object is equivalent to a C `FILE` object. In this case, `stream` is defined as a public type symbol (a C `typedef`). Note that the left brace is sufficient to declare the start of a structure. The several members above are all public (because of the first line inside the body of the structure). The `flagValues` type is another `typedef` used for specifying status bits. The `ref unsignedChar` types are pointers.

Using the visibility specifier at the top of the structure makes this declaration exactly equivalent to the following C declaration:

```
typedef struct {
        flagValues     flags;
        unsigned       bufferSize;
        unsigned char  *buffer;
        unsigned char  ungetByte;
        unsigned char  *curp;
        unsigned char  *endp;
} stream;
```

Structures and unions in Parasol are very similar to those found in C. In Parasol, in addition to normal structures there are also packed structures and structures that inherit from some other type. For all structures, as in C, the members are assigned to some contiguous set of locations. Padding of some number of bits or bytes may be inserted at various places to make access to members more efficient. Most modern computers take advantage of aligned data objects.

In the above example of the stream structure object, a C compiler may choose to force alignment for all objects so that padding would probably be inserted after the `flags` member (assuming it is just a byte) and the `ungetByte` member. Even more latitude is allowed in Parasol. A Parasol compiler is free to rearrange the physical order of the members in order to minimize the padding needed.

In Parasol, packed structures are used to specify externally defined data layouts. By adding the keyword `packed` before {, the compiler is instructed to maintain the order of the members as declared and also to supply the absolute minimum packing deemed reasonable. In the existing Parasol compiler, packed structures are never padded except for a few special cases involving bit fields. This permission to add padding allows compilers, for example, not to have to generate code to do shift and mask operations in orderto access floating-point members.

Inheritance

Parasol supports *single inheritance*. When you define a structure you can designate that the structure inherits from some other, non-function type. The newly derived structure is called a *subtype*. The type it immediately inherits from is called its *supertype*. The types inherited from either directly or indirectly are called *base types*. Since several subtypes can be derived from a common supertype, the collection of related types form a *tree*. The tree of related types is called a *type hierarchy*. The term *class* is sometimes used to mean a structure type, with *subclass* and *superclass* used as well.

Parasol allows you to use any type as a supertype. In practice if there is any type inherited it is usually another structure.

The default conversion rules of Parasol allow an rvalue to be converted from structure type to a scalar base type if needed in an expression. For example, a structure with a base type of `int` can appear as an operand of the `+` operator.

However, you cannot assign an integer expression to such a structure, because lvalues are not so converted. This capability is mostly used in the `string` object type of the standard library which inherits from pointer to character type.

Parasol does not support *multiple inheritance*. (This is the capability to define several direct supertypes for a single derived type.) Multiple inheritance certainly makes the compiler more complicated and also introduces some code overhead. I have yet to encounter a problem where the benefits of multiple inheritance outweigh the costs.

Inheritance has been touted as a way to re-use code, but my experience has indicated that code re-use is only a side-effect. The primary role that inheritance plays in the standard libraries and in many of the Parasol programs that I have written is as a mechanism for defining abstract data types. An abstract data type defines a set of functions on the type, but those functions have trivial implementations in that type. Usually, the implementation returns some error or simple default value. The power of abstract data types comes when subtypes are defined that inherit the interface of the base type. The collection of subtypes use the same interface but introduce their own specialized behavior.

For example, in the standard library the viewer type is an abstract type that defines generalized behavior for any interactive window in the display subsystem. Editors, menus, and data entry forms are all subtypes of the general viewer type. For another example, in the ALYS operating system device drivers are subtypes of several basic device types. Network interfaces, disk drives, and communications lines are each represented by an abstract data type. Each specific piece of hardware uses a driver that inherits one of the core driver types. Code that must use a viewer or a device driver need only know the interface for the base type. The details of internal implementation is left to the specific subtype involved.

An important point about inheritance is that the power of abstract data types can only be achieved through a capability called *polymorphism*. Polymorphism is the ability of an object to respond to generic interface functions with behavior that is specific to the object itself.

In Standard C, the equivalent functionality that polymorphic objects supply is frequently accomplished using `switch` statements. Where in Parasol you write a simple call to a polymorphic function, in C you might write a `switch` statement with each case being the body of one of the separate functions one might find in the Parasol program. Occasionally, function pointers are used in C code to provide similar functionality. Most of the time, though, function pointers in C are used for simple call-back situations like `qsort`.

Especially when the function bodies are copied into a `switch` statement, the C `switch` statement is usually faster than the indirect calls that Parasol uses. The timing difference is fairly small, however. I have not seen a program whose overall timing was noticeably affected by converting `switch` statements to polypmorphic calls. Why then does Parasol promote using polymorphism over the conventional approaches C programs have taken?

In a C program written with `switch`es, one finds more or less the same

`switch` statement in numerous places. The kinds of maintenance changes that most often occur usually involve adding a new case. In a C program, each `switch` statement must be examined and code must be added to some of them. Complicating matters is the fact that sometimes the `switch` statement is consolidated into a simple `if` statement when the cases are simple enough. Just finding all the places that are affected by adding a new case can be daunting.

In a Parasol program, each case is represented by a subtype. Adding a new case involves adding a single new type definition with all of the code for each of the places where C would have had a `switch` statement now written as a function. Of course, in the situation where a change would have involved a new `switch` statement in a C program, in the Parasol program one must add a new function to each affected type. Where the `switch` statement was added at a single place in the C program, the Parasol functions must be added in a number of different places.

In any procedural language there will be changes that are expressible as a single chunk of code and there are changes that must be written as distinct chunks scattered around the program. How you write your program and what features your language has will naturally influence exactly what changes are easy and what are hard.

I certainly do not have statistical studies to back up any claims I might make. So consider what I have to say as anecdotal evidence. My experience is that polymorphic objects tend to have more coherent behavior and changes are somewhat easier on balance than with conventional C approaches.

I will also say one more thing. Learning object-oriented programming after knowing C is not a simple process. One of the things I wanted to do as part of the project of designing Parasol was to learn object-oriented programming. When I started, my objects rarely used polymorphism. My type hierarchies were shallow and disconnected. Most of my structures still looked like C structures, with entirely public members and no methods.

As I have written more and more code, the type hierarchies have grown deeper. As I have modified programs I wrote early on, types that were originally separate have been gathered into ever more inclusive arrangements. Now, most types I introduce use polymorphism at least to some extent.

For anyone wishing to learn about object-oriented programming, I would say look first to polymorphism. Whatever language you use, find out how to make polymorphic objects and learn how to exploit the capability. My personal feeling about OOP is that without polymorphism there is nothing that we haven't been able to do just fine with C.

Pointers to Objects

In C, pointers that point to distinct non-`void` types can only be converted by an explicit cast. Parasol allows conversion of pointers in limited circumstances. When you are converting from a pointer to a subtype to a pointer to any of its base types you do not need a cast in Parasol.

By using pointers in Parasol you can treat an object as if it had the type of any of its base types. This is exactly how polymorphism is accomplished. When dynamically allocating objects belonging to a subtype, you would typically store the address into a pointer to some base type. In this way, code can be written that works for all types derived from that base type.

Methods

You can define functions (called *methods*) inside the body of a structure or union declaration. While they use the syntax of normal functions, there are some subtle differences. There are three kinds of methods supported by Parasol: *normal*, *dynamic*, and *factory* methods. Normal and dynamic methods operate on an existing instance of the structure including the method while factory methods are intended to create new instances of the structure.

For example:

```
inFile:   ref stream;    // declare a stream pointer
status:   int;
buf:      [80] char;

status = inFile read(&buf, sizeof buf);
```

In this code, the `inFile` object is being used. The `read` method is being called to read data into a buffer. Notice that no punctuation occurs between the `inFile` and `read` identifiers. I decided that using either a period or an arrow might mislead the reader. With no punctuation it is a method call, not a call through a member that happens to be a pointer to a function. I realize that this is a small point, but that was my reasoning.

I decided that I would not distinguish between a call through a pointer or through an object itself. In other words, the above statement could be written as follows:

```
status = (*inFile) read(&buf, sizeof buf);
```

I decided that automatically stripping one level of indirection would not cause any significant problems. Since pointers cannot have methods themselves, the only way a method call could be valid would be if the object being referenced was used instead. My experience of using the language is that blurring the distinction between objects and pointers to objects is not a major source of confusion.

Normal Methods

Unless you specify otherwise, a method will be defined to be normal. This means that any calls to that method are bound at compile-time. For static or automatic objects, compile-time binding is always possible since you know

exactly which type is being referenced. Normal methods are not polymorphic. If you call a normal method through a pointer you will execute the method defined for the exact type mentioned in the pointer. If the object being referenced has its own version of the same method it will not be used.

Because calls are bound at compile-time, normal methods can be redefined in a subtype without restriction.

Dynamic Methods

Dynamic methods are like normal methods, except that they are polymorphic. For example:

```
write: dynamic (buf: ref, len: int) int =
{
        ...
}
```

This defines a `write` function, with two arguments. The `dynamic` keyword indicates that this is a dynamic method. Dynamic methods are like virtual functions in C++. Note that the type of the `buf` parameter is a pointer to no specific type (i.e., a `void` pointer).

Dynamic methods work by using a tag pointer stored with each object. When an object with a type containing dynamic methods is declared or allocated, the tag pointer is stored into the object. The tag points to a dynamic function vector. For each type with dynamic methods, a function vector is created, with one function pointer to each dynamic method of the type.

When redefining a dynamic method in a subtype, you must use the same number and type of arguments and the same return type. The method's address is stored in the corresponding vector entry of the subtype vector as the original method's address was in the base type vector.

When calling a dynamic method, the tag value of the object is used to locate the dynamic function vector. In this way, the pointer type used to reference the object determines the interface of the method, but the actual type of the object determines which version of the method is actually called. Since all versions of the method have the same interface, the call itself is nearly as efficient as a normal function call.

Factory Methods

Factory methods are the third kind of function that can be declared in a structure type. They are somewhat different from other kinds of methods. Instead of using the method call described above, you call a factory method for the type itself. For example:

```
stream:   {
open:     factory (name: ref char,
                   options: int) ref stream =
          { ... }
};

inFile = stream open("somefile", AR_READ);
```

This statement calls the factory method `open`. Most factory functions return a pointer to their owning type. Factory methods mostly provide a convenient means of defining a function that creates objects of the desired type.

`self` and `super`

Inside methods, two built-in variable names are predefined. The `self` keyword is a pointer to the invoking object. For normal and dynamic methods, the object mentioned in the method call is assigned to `self` by being passed as a hidden argument. The `super` keyword is a pointer to the same object, but with type of the direct supertype of the enclosing type. `super` is valuable when recursively calling methods for the supertype, such as initializing an object when it is created.

`self` is also defined for factory methods, but means something a little different. In a factory method, `self` is an automatic variable that can be used like any other. (It was not passed to the function.) Typically, the first line of a factory method allocates the space for the object and assigns its address to `self`. `self` rarely appears in code explicitly. Any unqualified reference to a member of the enclosing type is interpreted as a reference through `self`.

Bob Jervis authored the Wizard C compiler and was the chief architect for Borland's popular Turbo C compiler. He is now self-employed and is developing software using Parasol. Bob can be reached at uunet!bjervis!rbj.

∞

20. Electronic Survey Number 10

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 about 100 people, with 28 of them responding. Since some vendors support more than one implementation, the totals in some categories may exceed the number of respondents. Also, some respondents did not answer all questions, or deemed them 'not applicable.' I have attempted to eliminate redundancy in the answers by grouping like responses. Some of the more interesting or different comments have been retained.

Compiler Response Files

Many implementations support a large set of command-line options. Some systems have severe limits on the maximum length of a command-line. As a result, some compilers support response files from which they can read arbitrary long command lines. Do you support such a notion and if so, what syntax do you use?

- 7 – Provide support via a response file
- 3 – Provide support via an environment variable
- 12 – No support or handled by operating system method
- Comments:

    1. The `@name` syntax is used. It can appear anywhere on the command line. `name` is first searched for in the environment. If it is there, the environment string is inserted into the command line at that point. If `name` is not an environment variable, it is searched for as a file, and the contents of the file are inserted into the command line at that point. The environment variable approach is nice because it is much faster than the file approach, and no cleanup is necessary.
    2. The right place to get rid of the limit on argument size is in the kernel. That's where we are going to do this.

3. Currently our system supports an environment variable that may contain more "permanent" options but any remaining options must be specified on the command line. Since length limitations usually occur in specifying multiple source/object files on a command line our implementation allows a file reference containing additional files using `@filename`.
4. The `-via` command line options causes commands to be read from that file as if they appeared on the command line. Also users can have local configuration files that may contain command line options.
5. Our response file uses syntax similar to that of the command line but only one control per line is permitted.
6. Our shell supports arbitrary length command lines. In addition, the shell language allows copying a file into the command line. In addition, every command line option is also available as a pragma (and these can be included in a header).
7. We support response files using a prefix `@` character. This is the convention used by other applications on our system (inherited no doubt from the DEC ancestry of the CP/M and MS-DOS systems).
8. `@file1` means back-substitute the contents of `file1` as if it were on the command line. Such files can be nested arbitrarily deep, limited only by memory availability.
9. There is no command line. Option settings are controlled via dialog boxes and may also be adjusted by using pragmas.

Support for C++-style Comments

Do you support the C++ style of line-oriented comments? If so, how do you handle the following fragment of code?

```
i = j;  // some comment \
i++;
```

According to C's phases of translation, backslash/new-lines are dealt with in phase 2 while comments are handled in phase 3 which means the line containing `i++;` *would be treated as part of the preceding line's comment. What would it take to accommodate this comment style in the phases of translation such that the two lines would be kept separate?*

- 9 – Support `//` comments with `i++;` as part of comment
- 2 – Support `//` comments with `i++;` as as separate statement
- 11 – Don't support `//` comments

- Comments:

    1. I believe this is a reasonable restriction. It is a silly construct (to have a backslash/new-line on the end of a `//` comment) so there is no point in making the compiler jump through hoops to handle it in some "intelligent" way.
    2. For both C and C++ compilers this is treated as two statements. In my opinion this is preferable to the way that ANSI defines it. Despite my opinion, in our next C compiler release the `i++` line will be treated as part of the preceding comment.
    3. We initially implemented rules that ended the comment on the same line in spite of the translation phases ordering. This was changed later on as a result of test suite checking for this.
    4. The C++ committee is so far begrudgingly accepting this so as not to "mess up" the phases of translation.
    5. To change it so that `//` comments are *not* continued by the `\` would be so ugly that I would not consider it as a solution at all.
    6. One problem that is frequently overlooked with regard to `//` is that Japanese programmers like to put wide characters into comments. Some of these turn out to have byte sequences in them that "look like" new-lines.

Subscripting Invalid Pointers

A common programming technique involves pretending arrays begin at subscript 1 instead of 0. This is achieved by writing something like the following:

```
int a[10], *p = &a[-1];

p[1] ≡ a[0]
```

Standard C declares this to be undefined behavior since `p` *does not point into a known object. Do you know of any systems where this will not work?*

- 18 – Works on the systems I know of
- 1 – Can have problems on IBM System 38 and AS/400
- Comments:

    1. The key is that pointer arithmetic silently wraps around (like integers) and that there is always a single, flat address space. If either of these is not true, the above code can fail, even though for most implementations, most of the time, it will work as intended.

2. I detect and flag the out-of-bounds reference at compile-time but the code will work anyway.
3. We haven't implemented this yet but may give a compile-time diagnostic and a runtime error if access to `a[i]` is out of bounds.
4. We went over all this in X3J11 and agreed that this constraint was necessary; the relaxation for `&a[10]` was a compromise that causes many implementations to have to resort to a bit of game playing, but we felt it was necessary and sufficient for simple array looping. Note also that `&a[-1]` can overlap the data address space by an arbitrarily large amount, depending on the type of an element of the array; this is not true for `&a[10]`, which requires only a single byte of address space extension.
5. This will work but the debugger will not let you look at `a[-1]` and if range-checking is enabled it will detect an error.
6. This particular example will work on Intel's 80x86 segmented architecture. Even if `a` is another pointer and is assigned a `malloc`ed pointer, and even if the pointer arithmetic wraps around, so that `p[0]` has a higher address than `p[1]`, the identity `p[1] == a[0]` holds. However, consider the following DOS-specific example:

   ```
   double  far * a, huge * p;
   a = malloc(10 * sizeof (double));
   p = &a[-1];
   p[1] ≡ a[0]?
   ```

   We need to use `double` type to have something large enough to allow for wrap-around to happen. In this case, `a` uses fast pointer arithmetic that doesn't compensate for wrap-around (that is, the arithmetic is not normalized), but `p` uses huge pointer arithmetic that does compensate. As a result, `&a[-1]` wraps, is assigned to `p` and normalizes to some high address, and `&p[1] != &a[0]`. This is a pretty contrived example, and relies on non-standard extensions.
7. `&a[-1]` may fail on an IBM AS/400 depending on where `a` is located in storage and how big each element of `a` is. The AS/400 will not allow an address to become negative and it does not wrap around to the high end like many other chips do. The System 38 behaves in the same manner.

`__STDC__` in Future Standard Revisions

Standard C defines `__STDC__` *to be 1 if you are compiling in a strict standard-conforming mode. However, it makes no reservation for future standard versions. And to complicate matters, some vendors are defining this macro to be*

0, 2, 3, etc., to indicate various levels of sub- or super-setting of Standard C. At the last WG14 (ISO C) meeting in Tokyo in May, I indicated I would be introducing a proposal to specify further values. My proposal is that future revisions of Standard C define `__STDC__` *as a 4-digit integer matching the year in which that revision was finally adopted. So assuming the revised ISO C standard debuts in 1993,* `__STDC__` *would be defined as 1993 for that version. Implementors would be at liberty to use values less than 1989 for their own purposes allowing us to also add 1989 for Level I conformance later if we so desired. Comments please.*

- 13 – Like the idea
- 5 – Don't like the idea
- 5 – Sounds interesting but not sure
- Comments:

  1. I believe the standards-making bodies should attempt to direct the way non-conforming compilers use `__STDC__` by specifying more information on its usage (in the Future Language Directions section) and making recommendations on quality of implementation issues in the Rationale or some other appropriate forum.
  2. I have no problem with branding successive standards with successive values of `__STDC__` but see no particular value in using the year as the value. There are obviously a few minor problems with this: What if we come out with two standards in one year; why do we choose to use the Western calendar for the year. It seems simpler and cleaner to just bump up the value by 1 with each successive standard. I admit to having absolutely zero sympathy for somebody who has already assigned a value of 2 to `__STDC__`.
  3. I think this is short sighted. If `__STDC__` is to pass information beyond yes/no, use a character string and define a few cases. Then I would not mind `"1993"`. Type `int` is not a very flexible way for the compiler to say what it does. On the other hand my suggestion requires adding stuff to preprocessor constant expressions so that conditional compilation can actually be done. Muddy waters!
  4. A fundamental problem is that no standard can constrain what a non-conforming implementation does.

     The original intention was that `__STDC__` be predefined as 2 in the next revision of the C standard; I see no need to deviate from that. Vendors who have invented their own use for `__STDC__` should not be allowed to constrain what the standard meaning is.
  5. Sounds okay but ... what about an implementation that incorporates all of ISO C except for the Japanese multibyte proposal? Are they

1989? 1992? What if a later version of C removes some part of 1989 ANSI C but an implementation keeps that part as an extension? 1989 (because they support all of 1989)? or 1995 (because they support all of it plus the extension)? Of course, NCEG will have its own set of values. Will `__NCEG__` be set according to some year or according to some value indicating what portions are supported (i.e., floating-point extensions but not variable-length arrays).

6. The only practical use I've seen for `__STDC__` is to be able to write code that works with old and new compilers. The question that is asked is really, "Are prototypes supported and is `const` around?" Since the ISO C addenda are mostly additional library functions, I'm not sure that an updated `__STDC__` is needed. Moreover, the only addendum that affects the language (new tokens) is such that having a compile-time test for availability defeats the entire intent! (I can't see how something like

```
#if __STDC__ > 1989
        main()
        <:
        :>
#else
        main()
        ??<
        ??>
#endif
```

   is at all likely!) Another drawback is that the simple test of today

```
#ifdef __STDC__
```

   cannot be used once there are rules about the values. Code would now have to protect itself further:

```
#if (__STDC__ - 0) == 1 || (__STDC__ - 0) >= 1989
```

   Ugh! I'd hold off on any upgrading of `__STDC__` until there is a clear need for this sort of distinction.

7. Your proposal would assure that no compiler could conform to both the current standard and the revision. Probably a safe assumption on general principles, but not something I'd like to have to get up at an ISO C meeting and defend.

   I think the whole idea of `__STDC__` was unsound from the beginning and I don't see any purpose being served by complicating it further.

Defining Function-Like Macros Using a Compiler Option

Just about every implementation provides a mechanism for defining an object-like macro on the compiler command-line. Does yours also allow you to define a function-like macro in this way? Since such a macro definition may contain almost arbitrary characters it is possible that some of them (e.g., < and >) might be misinterpreted by the operating system's command-line shell. Any comments on the value of having this capability?

- 6 – Support this
- 18 – Do not support this
- Comments:

  1. I can think up situations in which they might be useful, but since the facility is most often used to choose conditional compilation paths it does not seem to be necessary.
  2. No customer has ever asked for it in the 6 years we've been shipping.
  3. If somebody came up with a standard way of doing this we might implement it, but it seems fairly unimportant to me.
  4. Not yet, but maybe in the future.
  5. Our C/C++ compilers support this. To get the equivalent of

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

     we use either of the following:

     ```
     -Df(x)=((x)+1)
     -Df\(x\)=\(\(x\)+1\)
     ```

     Naturally it is possible that special characters (in addition to `(` and `)`) will be (mis)interpreted by the UNIX shell and the user must take care; but, of course, the same also applies to object-like macros.

     This ability seems to be nice but really only marginally useful.
  6. We accept macros of the form `-Dmacro(a,b,c)=`*body*. It is up to the user to ensure that no conflicts with the current operating shell occur and that the option winds up being parsed into a single `argv` element.
  7. The special character "problem" really isn't one [in UNIX] as far as I can tell; anything can be escaped or quoted to protect it from the shell. *[Ed: It certainly is a problem in a number of other popular systems.]*

8. The syntax we use is `-DM(a)=foo` and the limitation is that no blanks are allowed in the right-hand side.
9. Unlike object-like macros, which are good for entering configuration parameters, function-like macros are too powerful to be defined separately from the source code.
10. We permit `cc test.c "-Dg(x)=((x)*(x))"`. I can imagine that a program-generating program might make use of the facility.
11. Our implementation allows the user to specify an arbitrary "prefix," which is simply source text that is automatically prepended to each source file. The feature is intended to be used to specify `#define`, `#include`, and `#pragma` directives, although no restrictions are enforced.

Default Value for Object-Like Macros

Defining an object-like macro using something like `cc -dM` *or* `cc -DM` *causes the macro* `M` *to be defined with a value of 1 on many implementations. Does it do so on yours? If not, what value, if any, is assigned to* `M`*? If so, can the user override this somehow to get* `M` *to be defined with no value?*

- 19 – `-DM` defines the macro to 1 with `-DM=` assigning an empty string
- 1 – `-DM` defines the macro to 1 with `-DM=` not permitted
- 3 – `-DM` defines the macro as an empty string
- Comments:

  1. As a user I would prefer the implementations treat `-DM` as `#define M`. What use is there in providing an arbitrary value?
  2. The compiler is never invoked with a command line, so this question doesn't apply. Via an interactive dialog box, the user may specify an arbitrary "prefix," containing, for example, a standard (unabbreviated) `#define` directive.

∞

21. ANSI C Interpretations Report

Jim Brodie
Motorola, Inc.
Tempe, Arizona

Abstract

This article examines X3J11's responses to, and the issues surrounding, a variety of Requests for Interpretation that have been addressed by X3J11, the committee tasked with interpreting the American National Standard for the C language.

In this article we continue our look at the answers to the Requests for Interpretation that have been addressed by X3J11.

Accessing Objects

The first request for interpretation focuses on the rules concerning valid accesses to data objects.

The rules that govern access to objects are defined in §3.3, Expressions. They state:

> "An object shall have its stored value accessed only by an lvalue that has one of the following types:
>
> - the declared type of the object,
> - a qualified version of the declared type of the object,
> - a type that is the signed or unsigned type corresponding to the declared type of the object,
> - a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
> - an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a sub-aggregate or contained union), or
> - a character type."

The underlying theme of these rules is that a value stored in an object may only be accessed through an lvalue of an appropriate type. For example, given the declarations

```
double d = 3.0;
int i;
int *pi;
```

an attempt to access the value stored in the object associated with `d`, using

```
pi = (int *) &d;
i =  *pi;
```

results in undefined behavior (for which the Standard imposes no requirements).

These rules also allow translator optimization to assume that only declared objects with the *target type* of a pointer may be affected by a write through that pointer. For example, after the declarations

```
short s1;
float *f;
```

and the code

```
/* arbitrary code setting the value in f */
s1 = 14;
*f = 3.2
```

the optimizer can safely assume that the value stored in the object associated with `s1` still has the value 14.

The catch in all of this is that, with the exception of `char`-based access[3], the rules depend on "the declared type of the object."

The issue that the Request for Interpretation raises is "Do the rules apply when there is no declared type of an object?" This occurs when the object is created using `calloc`, `malloc`, or `realloc`. The committee's response was that these rules *do* apply for dynamically allocated objects as well as explicitly declared objects.

In its response the committee notes that Standard wording does not explicitly address this additional case. However, the committee felt that the footnote on the above list of ways you can access a value, which states

> "The intent of this list is to specify those circumstances in which an object may or may not be aliased."

indicated clearly that the intent was to include *all* cases.

In addition to the above citation, the Standard states a consistent position on limiting the way that values can be accessed. For example, in §3.3.2.3, Structure and Union Members, the Standard states:

[3]The `char`-based access was grandfathered in the Standard because of the common practice of viewing objects as a sequence of bytes.

> "... if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation defined."

This interpretation is an extrapolation of the wording in the Standard. However, it is consistent with the rest of the document and known existing practice.

Tags and Types

A request for interpretation asked, "Was it the committee's intent that the two types must have the same tag (or both lack tags) to be compatible?" The author of the request used the following example:

```
/* Translation Unit #1 */

struct foo { int i; } x;

/* Translation Unit #2 */

extern struct bar { int i; } x;
```

He then asks "Should these be considered compatible?"

The committee's response was that tags do *not* have to be the same for structure, union, or enumeration types to be compatible in separate translation units. In the example shown the two declarations are considered compatible.

To support this position the committee referenced §3.1.2.6, Compatible Type and Composite Type, which states:

> "... two structure, union, or enumeration types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values."

This section indicates, by omission, that the tags are *not* part of the type compatibility comparison.

Constant Expressions

The evaluation and exception handling of constant expressions was the topic of the next Request for Interpretation.

The request noted two potentially contradictory Standard references. The first, from §3.3, Expressions, states:

> "If an exception occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for the type), the behavior is undefined."

The second statement from §3.4, Constant Expressions, states:

> "Each constant expression shall evaluate to a constant that is in the range of representable values for its type"

The request then asks, in light of these statements, how each of the following should be handled:

```
INT_MAX + 2
INT_MAX + 2ul
(INT_MAX*4) / 4
enum { a = INT_MAX, b };
```

There are several parts to the answer for this Request for Interpretation.

The first is the observation that the statement, "Each constant expression shall evaluate to a constant that is in the range of representable values for its type" is a requirement placed upon a strictly conforming program and *not* a requirement upon the translator. It is the responsibility of the programmer to avoid writing a program that violates this constraint, rather than a statement to translator writers that they must make the constant value map into a specific range of values. This is supported by the somewhat obtuse definition of Constraints in §1.6, Definition of Terms, which states:

> "Constraints – syntactic and semantic restrictions by which the exposition of language elements is to be interpreted."

The next issue that must be considered when determining the rules for constant expressions is whether or not sub-expressions are considered liable to the same constraints as an entire expression.

The committee position is that the intent of the "representable" requirement (from, "Each constant expression shall evaluate to a constant that is in the range of representable values for its type.") applies to each sub-expression of the complete expression. Essentially, this position is based upon the fact that each sub-expression is, itself, an expression. If this were not the position taken, then the sub-expressions would have to be calculated using types with a larger range of representable values and this is clearly not required by the Standard.

With this background, we can now answer the questions about the first three expressions.

> `INT_MAX + 2` is a constraint violation (the result is larger than can be held in an `int`, which is the type used for the computation and the result).

> `INT_MAX + 2ul` is OK (the result can be represented as an `unsigned long`, which is the type used for the computation and the result). `(INT_MAX * 4) / 4` is a constraint violation (the intermediate value is too large to be represented in an `int`).

The answer to the final example requires a little more information. In §3.5.2.2, Enumeration Specifiers, the following constraint is established:

> "The expression that defines the value of an enumeration constant shall be an integral constant expression that has a value representable as an `int`."

In the Semantics portion of that section, the following rule is specified for how the value of an enumeration constant is determined:

> "... Each subsequent enumerator with no `=` defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant."

These two items make it clear that in

```
enum { a = INT_MAX, b };
```

that the expression `b` would have the value `INT_MAX + 1`. However, since this is outside the range of values "representable as an `int`" this is a constraint error.

Allowable Domain Errors

The next Request for Interpretation asks:

"If `sin(DBL_MAX)` results in `errno` being set to `EDOM`, is this [a] violation of the standard?" This question is based on the statement in §4.5.1, Treatment of Error Conditions, which states:

> "For all functions, a domain error occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function."

The issue centers around the limitations on what an implementation can declare as domain errors for a function.

X3J11's position was that this section does `not` allow an implementation to set `errno` to `EDOM` for `sin(DBL_MAX)`. Since the mathematical function is defined for this value the error would not be "consistent with the mathematical definition of the function." The footnote attached to the above reference in the Standard gives a better feeling for what kind of flexibility is intended by these words. The footnote says:

> "In an implementation that supports infinities, this allows infinity as an argument to be a domain error if the mathematical domain of the function does not include infinity"

It is interesting to note that while the standard prevents a conforming translator from generating code that will set `errno` to `EDOM` in this case, it does not impose any restrictions on the accuracy of the resulting value. In this arena, the accuracy of the results is left to the market forces and "quality of implementation."

Questions about `strtod`

Another Request for Interpretation probes the requirements placed upon `strtod` when dealing with unusual specifications of the value 0. The request asks:

> "Assuming that 99999 is larger than `DBL_MAX_10_EXP`, what is the result of `strtod("0.0e99999", &ptr)`—is it 0.0, `HUGE_VAL`, or undefined."

The request cites several references and arguments for each possible interpretation:

In §3.1.3.1, Floating Constants, the Standard states:

> "The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled."

The request notes that in this case 0.0e99999 means 0.0 times 10 to the power 99999, which is still 0.

The second reference is §4.10.1.4, The `strtod` Function, which states:

> "If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`."

The Request for Interpretation argues that

> "Since the exponent is larger than `DBL_MAX_10_EXP`, the value is outside the range of representable values (overflow). Therefore, return `HUGE_VAL`."

The third reference is §2.2.4.2.2, Characteristics of Floating Types, which describes the floating-point types. The number 0.0e99999, as written, is not part of that model (it cannot be represented since the exponent is larger than e^{max}). Based upon this and §3.2.1.4, Floating Types, which states:

> "... if the value being converted is outside the range of values that can be represented, the behavior is undefined."

The Request for Interpretation argues:

> "Therefore, since this number, as written, has no representation, the behavior is undefined."

After evaluating these arguments, the committee position was that the result of calling `strtod("0.0e99999", &ptr)` is exactly representable. The value does lie within the range of representable values and can be represented exactly as 0. Despite the way the value is specified in the C program source, the mathematical value is clear and therefore the arguments based on a value "outside the range of values that can be represented" do not apply in this case.

This means that implementations have to perform a special case test for the value 0.0 to ensure that it creates the correct internal floating-point representation, independent of the size of the exponent part of the constant.

The next request also examines the `strtod` function. It asks what first appears to be a simple question:

> "What is the result of `strtod("100ergs", &ptr)`? Is it 100.0 or is it 0.0?

If you look at §4.10.1.4, The `strtod` Function, it states:

> "The `strtod` function converts the initial portion of the string pointed to by `nptr` [the first argument to `strtod`] to `double` representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling a floating-point constant and a final string of one or more unrecognized characters, ..."
> "... The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white space character, that is of the expected form."

In this case, the longest initial sub-sequence of the expected form is `"100"`, therefore the value returned should be 100.

The confusion arose when the author of the request considered §4.9.6.2, The `fscanf` Function, which describes the conversion specifiers:

> "... `e`, `f`, `g` – Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function."

Examples later in that section of the Standard show that `"100ergs"` fails to match the `%f` specifier in an `fscanf` function call. The examples show that

the `100e` is consumed by the `fscanf` function, but since this is an invalid form for a floating-point constant, the function fails.

The essential question is, "If the input string `"100ergs"` fails in `fscanf`, should it also fail in `strtod`?" If the answer is yes, then the `strtod` function should return 0 instead of 100. This is based on the statement in §4.10.1.4, The `strtod` Function, which establishes:

> "If no conversion could be performed, zero is returned."

In reply to this Request for Interpretation the committee writes:

> "`strtod` and `fscanf` are different—`fscanf` expects a sequence that matches what works for `strtod`. `fscanf` can easily get into corners that it can't back out of, as it is constrained not to "leave unread" more than one byte. On the other hand, `strtod` must do a reasonable job with any input string.
>
> Thus, `fscanf(fp, "%f", &f)` when handed `"100ergs"` on its input stream will fail to convert, lose `"100e"`, and leave `"rgs"` unread (§4.9.6.2). In contrast, `strtod("100ergs", &ptr)` returns 100.0 and sets the pointer to point to the `e` character (§4.10.1.4)."

The key difference can be further explained in terms of the rules that are used to establish what item is to be converted to a floating-point value. As noted above, the `strtod` function identifies the item to convert using the rule:

> "... The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white space character, that is of the expected form."

On the other hand, the `fscanf` function uses the following rule to select its input item:

> "An input item is defined as the longest matching sequence of input characters"

In this case, the trailing `e` matches a potential floating-point number format and is therefore included as part of the input item. It is only after the input item is selected that the `fscanf` matching process is used to determine if the input item is a valid format for a floating-point constant or only a prefix for such a constant.

Jim Brodie is Chair of the ANSI C standards committee, X3J11, and a Chief Software Engineer for Motorola, Inc. He has coauthored books with P.J. Plauger and Tom Plum and is the Standards Editor for The Journal of C Language Translation. Jim can be reached at (602) 897-4390 or brodie@ssdt-tempe.sps.mot.com.

∞

22. C♭: A Low-level Subset of C

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

Abstract

A subset of Standard C is presented together with a set of transformations for taking Standard C into the subset. The intended purpose of the subset is to enable a software vendor to distribute a C program in source form without disclosing proprietary ideas that would normally be obvious to someone reading the source. Subsidiary uses are an enhancement to ANDF, the definition of some C semantics, a low-level target for language processors, and a C compiler test generator.

Introduction

The distribution of software in source form is attractive from the viewpoint of portability (standard library headers are target dependent) and user-chosen application tailoring (implemented with `#if`). On the other hand availability of source code normally reveals more proprietary information about the software than the vendor would like. One solution is to mangle the source for human readers without changing the meaning of the program. This paper presents some largely untried ideas for creative mangling.

The idea of mangling source has been around a long time. The Stanford Computer Center compressed Algol source to save space and card stock in the metal file cabinet which held the run-time library. Randy Meyers tells me that Digital distributed some of the PDP-11 operating system without comments in MACRO assembler source. Jim Gimpel mentioned some distribution of obscured Fortran. The C Shroud[4] product is an existing example of a program to obscure C source [3, 4].

This paper introduces a very small but complete subset of standard C [1]. The subset, a kind of opposite to C++, is an elaboration of the C Shroud technology. The monogram for C++ could have been C♯ because it is a little bit higher than C. I have therefore chosen the name C♭ for C with its structure flattened. It is my intent that the name C♭ be generic in the sense that it can be

[4] C Shroud is a trademark of Gimpel Software.

used for any language designed in the spirit of C♭. C♭ can be compiled by any standard-compliant compiler or alternatively by a (simpler) C♭ compiler. The use of C♭ is as a target intermediate language. It can also be used to generate a new kind of test for compilers.

The intent of flattening a C program down to C♭ is to obscure the engineering decisions that went into the program without degrading the eventual executable. The C♭ form of the source can then be safely used as a distribution medium. C♭ is principally a post-preprocessor source and target language. A special hack is added to allow the inclusion of standard library headers to be deferred, allowing the `#include` to be passed into C♭. All other preprocessing is completed prior to flattening. C♭ therefore does not provide the capability for the user to make build-time choices by setting `-D` flags for the compiler.

ANDF, standing for 'Architecture Neutral Distribution Format' is a compiler intermediate language with some special features to parameterize it relative to different target machines [2]. One of the requirements for ANDF was security of proprietary engineering information. What was achieved is the protection naturally inherent in tree-structured compiler intermediate code (which is to say, not much). To remedy this defect in ANDF, C♭ can be used as a preprocessor for ANDF input to enhance the protection and make reverse engineering more difficult. One should still not expect too much. While removal of comments and renaming of variables is not mechanically reversible, much of the structure of the program could be largely recovered from flow-graph analysis, which is an inherent part of optimizing compilation.

A set of transformations from Standard C into C♭ is the principal contribution of this paper. A well-engineered C program can be processed into C♭ leaving an ugly but still standard compliant C program. Because the transformations introduce a large number of explicit `goto`s, the eventual compiler of C♭ must have excellent flow analysis and optimization. There is the chance that a compiler would be able to compile some program but not its flattened form. That possibility reduces the utility of the idea proposed here and also motivates fixing any errant compilers.

The unit of translation is the *module*. In its flattest form, the translated module contains no comments and no pretty white space. One static variable without initialization replaces all the uninitialized statics. Each function contains the declaration of one local variable followed by a sequence of so-called *flat fragments*. The flat fragments are position-independent, and therefore can be sorted into an arbitrary order. The enabling technology of the transformations is a traditional front-end for C including scanner, preprocessor, parser, and symbol table. The flattener itself would be about as complicated as a straightforward C compiler. I have not implemented it. There probably exist correct C programs that cannot be flattened, requiring some adjustments to user source.

The order of presentation below of obscuring transformations is not the algorithmic order of application.

An Example

A hand-prepared *before/after*, C/C♭ example follows. As a courtesy to the reader, white space has not been entirely suppressed. The *after* form shows the result of a few of the obscurations. They are pretty horrible for the human reader. C compilers, on the other hand, face this kind of stuff all the time. The introduced typedef names below may force the programmer to change some program names to avoid complaints from the mangler.

```
/* before scrambling */
# include <stddef.h>
int
main(int argc, char **argv) {
    int i, j;
    struct {
        char a;
        wchar_t b;
    } c;
    i = c.b = argc--;
    {long j = 0;
        while(--i) j++;
    }
    exit(0);
}


/* after scrambling */
typedef char T1;
typedef signed char T2;
typedef unsigned char T3;
typedef short T4;
typedef unsigned short T5;
typedef int T6;
typedef unsigned int T7;
typedef long T8;
typedef unsigned long T9;
typedef float TA;
typedef double TB;
typedef long double TC;
# include <stddef.h>
int
main(T6 L8F021812, T1 **LEDE011B8) {
T3 L[
3*sizeof(T6)
+sizeof(wchar_t)
+sizeof(T8)
```

```
];
T7 L1001D4D1;
T3 *L19098AEB,*LDD3A3EF1,*LABA0B310;
goto LFDF1FE34;

L003A3F11:*(T6*)L= *(T6*)L-1;goto L1EC10566;
L00AF3F1B:*(T8*)LABA0B310=0;goto L003A3F11;
L1EC10566:if(*(T6*)L)goto L29887E12;else goto L320F832A;
L29887E12:*(T8*)LABA0B310= *(T8*)LABA0B310+1;goto L003A3F11;
L320F832A:exit(0);goto L6A6A1210;
L445AA12E:LDD3A3EF1=L19098AEB+sizeof(T6);goto L554A1101;
L54AEB1BB:L8F021812=L8F021812-1;goto L91123AE1;
L554A1101:*(wchar_t*)LDD3A3EF1=L8F021812;goto L7680D810;
L652DDE10:L19098AEB=L+L1001D4D1;goto L445AA12E;
L6A6A1210:return;
L7680D810:*(T6*)L= *(T7*)LDD3A3EF1;goto L54AEB1BB;
L91123AE1:LABA0B310=LDD3A3EF1+sizeof(T7);goto L00AF3F1B;
LFDF1FE34:L1001D4D1=2*sizeof(T6);goto L652DDE10;
}
```

The Header Hack

At the cost of losing the portability parameterization of standard library headers, those headers can be expanded and their contents obscured along with the rest of the program. This is the simplest way to handle headers. If preserving the `#include` directives into the C♭ target is deemed important, one can imagine a two stage elaboration to the flattening process.

The insertions from the standard library headers are in fact macro definitions and declarations of various kinds. These header contents are inserted as usual into the C source file during the C♭ translation process, using a vanilla version of the header. Any name defined in the inserted material is marked *untouchable* (not be be mangled and not to be expanded). In the final target image, the inserted material is deleted and the original `#include` directive restored in its place. This hack will not work for headers in general because not all headers are structured as simply as those in the standard library.

Boring Names and Expression

One of the tasks of the flattener is generating names that convey no information to replace the names carefully crafted by the programmer. I propose names of the form `LXXXXXXXX` where the `X`s are random hexadecimal digits. They are reasonably simple to generate using `rand()` and `sprintf()` with format `%8X`. In C♭ generated names are used temporarily for variables but in the final output

appear mostly as labels. The fixed-width format is used because it is convenient in the final scrambling sort of executable statements.

Variable Renaming

Because of scoping and name spaces, one name can be used for more than one thing in a C module. The symbol table is the mechanism that separates the uses based on context. Suppose that every use of every name local to the module (e.g. no `extern`s) and not defined in a library header is consistently given a unique `LXXXXXXXX` name. The resulting C program still has the same behavior as before the substitution.

Automatic variable declarations in function scope or deeper may or may not have initializers. If they do, the initializers can be dropped and replaced with one or more explicit assignments just following the declarations. Now any declaration in inner function scopes can be physically moved to the outermost function scope without introducing any name conflicts or changing the meaning of the program, or (excepting performance) changing its behavior. Other declarations in local contexts are treated similarly. The declaration transformations allows the simpler rule:

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

dropping all local declarations, to be used except for *function-definition* itself.

Local Variable Clotting

The next set of transformations eliminates most declaration and expression information. It is described now because it fits with the previous material but in fact must be carried out later because declarations and expressions subject to these transformations are generated in some later processing steps.

Each function scope contains zero or more variable declarations. Delete all the variable declarations (except for formal parameters to the function, structures with bit-fields, and anything carrying the qualifier `volatile`). Replace the declarations with a single declaration:

`T3 L[`*n*`];`

where *n* is the total accumulated size of the locals as expressed as a sum of `sizeof(`*type*`)` primaries. Now replace every reference to a local variable with an expression of the form

`*(`*type* `*)(L +` *frame-offset* `+` *local-offset*`)`

where `L` becomes a base address for the local frame, the first offset is a (perhaps null) sum of `sizeof` operands giving the start of the variable in that frame, and the second offset is another sum of `sizeof` operands or an `offsetof` operand

giving the offset within the variable (for a structure member). The cast restores the type information lost when the declaration was deleted. Avoiding the generation of explicit constant values keeps the transformed code target independent. It is reasonable to assume that `L`, being the first variable in a frame, is aligned on an efficient memory boundary, as are all subsequent local variables. The offsets take this into account by rounding up to the size of the addressed data item as necessary.

Variables with `volatile` can be included in clotting if the whole array `L` is given the attribute `volatile`. This may, however, cause optimizers real trouble.

Structure selection '`.`' has disappeared except for bit-fields. All subscripts `a[e]` are turned into their equivalent form `*(a+(e))`. All structure pointer indirections `e->n` are turned into their equivalent `*(*(e) +` *local-offset*`)`.

Expression Flattening

Turn all compound assignments into a pair of simple assignments (i.e. `++ -- *= /= %= += -= <<= >>= &= ^= |=`):

–before–	*–after–*
`++a`	`(a+=1)`
`--a`	`(a-=1)`
`a++`	`(p=&a, t=*p, *p+=1, t)`
`a--`	`(p=&a, t=*p, *p-=1, t)`
`a` *op*`= e`	`(p=&a, *p=*p` *op* `e)`

and transform expression statements not in the form:

```
;
p;
a = p;
a = q op r;
```

Here `a` is an identifier or dereferenced identifier; `p` is an identifier, constant or function invocation; and `q` and `r` are identifiers. Transform these statements into a sequence of expression statements in the above forms by introducing temporary variables. All parameters for function invocations are therefore generated names, to which the actual values have just previously been assigned.

The consequence of these transformations is to replace all local names with anonymous references to the local frame (except for formal parameters which just get boring names), and to replace all expression statements with the equivalent of compiler triples. The transformations never cause side-effect changes because each side-effect causing construct is moved out of its containing expression into an assignment to a temporary variable.

Because expression flattening introduces new variables and variable clotting introduces new expressions, these two processes are best carried out together. The example starting this paper, in fact, left generated local variables unclotted, which is a viable option.

Static Variable Clotting

Uninitialized static variables are collected analogously to variables in a frame. The clot is

```
static T3 S[n];
```

Initialized static variables are renamed `LXXXXXXXX` but otherwise left alone.

After these transformations only the external names and names from library headers have their original mnemonic form.

Transforming the Function Body

A series of transformations is applied to the executable statements in the function body, outside in. In each case the outer construct is matched to a template and replaced with a simpler, and perhaps more verbose, equivalent. The final objective is to turn the entire function body into a non-nested list of fragments, each having one of the following six forms where *e* is an expression:

```
LXXXXXXXX: goto LXXXXXXXX;
LXXXXXXXX: return;
LXXXXXXXX: return e;
LXXXXXXXX: e; goto LXXXXXXX;
LXXXXXXXX: if (e) goto LXXXXXXX; else goto LXXXXXXX;
LXXXXXXXX: switch (e) { transfer vector } goto LXXXXXXXX;
```

The *transfer vector* is a list labeled statements of the form `case` *e*`:  goto LXXXXXXXX;` followed by one more `goto`. See the transformations for `switch` below for more detail.

These constructs are called flat fragments. Part of the process is generating label names. Since there is no value in obscuring the meaning of this paper from the reader, the more readable names `L00`, `L01`, `L02` ... will be used in the examples instead of random values for `LXXXXXXXX`. The flat fragments are position-independent within the function body, and therefore may be sorted into a meaningless order before exposing them to examination. Since the labels are randomly generated, the labels become a convenient key for the sort.

The first transformation is applied to the *statement-list* in the function body. The statement list is transformed as follows:

–before–	*–after–*
{*local declarations*;	{*local declarations*; `goto L00;`
S0;	`L00:`*S0*`; goto L01;`
S1;	`L01:`*S1*`; goto L02;`
...	
S3;	`L03:`*S3*`; goto L04;`
	`L04:return;`
}	}

This transformation may generate unreachable code; never mind—it will be taken care of later. The generated `return` may be inconsistent with the function itself. If the original program conforms to the standard, an inconsistent `return` cannot be reached, so never mind. The resulting fragments are position-independent but not necessarily flat because each of the statements *S1, S2,... S3* above may be complex. Thus there are some transformations left to do. Each statement can be processed independently of its neighbors, which means one at a time, or even in parallel.

Transforming Statements

The total set of statements that must be transformed is described in the C Standard [1]. Some of the transformations are purely textual; some require context analysis (such as associating `break` with its correct span).

Transforming `break` Statements

The `break` statement requires special preprocessing before the other transformations are carried out. Suppose *S5* is an *iteration-statement* or `switch`

```
L05:  S5; goto L06;
```

and *S5* contains one or more occurrences of `break` not nested within a deeper iteration or switch. Then the text `goto L06` is substituted for each such `break`.

Transforming Empty Statements

An empty expression statement is shortened:

–before–	*–after–*
`L07:  ; goto L08;`	`L07:goto L08;`

Transforming `goto` Statements

A `goto` statement is shortened:

–before–	*–after–*
`L09:  goto L0A; goto L0B;`	`L09:goto L0A;`

Transforming `return` Statements

A simple `return` without a return value drops the following `goto` to get into flat form.

–before–	*–after–*
`L0C: return; goto L0D;`	`L0C:return;`

If there is an expression, it may be separated out into an assignment that can be flattened using the expression algorithm detailed above.

–before–
```
L0E: return e; goto L0F;
```

–after–
```
L0E:L10 = e; goto L11;
L11:return L10;
```

Transforming Labeled Statements

A labeled statement is flattened as follows:

–before–
```
L12:L13:S6; goto L14;
```

–after–
```
L12:goto L13;
L13:S6; goto L14;
```

Transforming Compound Statements

A compound statement has no declarations because of the initial flattening of the declarations. The transformation removes the curly braces.

–before–
```
L15:{S7; S8;...S9;} goto L16;
```

–after–
```
L15:S7; goto L17;
L17:S8; goto L18;
...
L19:S9; goto L16;
```

Transforming `switch` Statements

Replace the switch expression with a generated variable and precede the switch with a flat assignment of the expression to the generated variable. Then transform the switch by inserting a new switch statement body just before the original switch statement body. The new body is a transfer vector in the form of a *compound-statement* listing all the original `case` labels (including `default`), each followed by a `goto` to a new generated label, and one final unlabeled `goto` to the label following the original switch body. The original switch body itself gets a new generated label and each `case` label in it is replaced by the corresponding generated label in the transfer vector. This restores the program to flat form, although the generated assignment and substituted original switch body now need to be flattened. As before, this may generate some unreachable code. Never mind.

–before–

```
L20:switch(e) S10; goto L21;
```

–after–

```
L20:L22 = e; goto L23;
L23:switch(L22) {
    case C1:  goto L24;
    case C2:  goto L25;
    ...
} goto L21;
L27:substituted S10; goto L21;
```

Alternatively, a switch statement can be mangled into a sequence of if statements, but this places an unreasonable load on the optimizer of the eventual compiler.

Transforming Iteration Statements

An *iteration-statement* has one of four forms: two for `for` and one each for `do` and `while`. For all of them the `continue` statement must be processed as part of the flattening transformation. The details of removing `continue` are slightly different for the four cases.

The `while` statement is treated much as in C Shroud and also the standard.

–before–

```
L28:while(e)S11;goto L29;
```

–after–

```
L28:if(e)goto L2A;else goto L29;
L2A:S11;goto L28;
```

Any `continue` statements in *S11* (and not in a more deeply nested *iteration-statement*) are replaced with `goto L28` either before or after the above transformation.

The `do` statement is similar:

–before–

```
L2B:do S12;while(e);goto L2C;
```

–after–

```
L2B:S12; goto L2D;
L2D:if(e) goto L2B; else goto L2C;
```

The `continue` statements in *S12* turn into `goto L2D`.

The `for` statement with a non-null limit expression *e2* translates as follows. Either *e1* or *e3* or both can be null.

–before–

```
L2E:for(e1;e2;e3)S13; goto L2F;
```

–after–

```
L2E:e1; goto L30;
L30:if(e2) goto L31; else goto L2F;
L31:S13; goto L32;
L32:e3; goto L30;
```

The `continue` statements in *S13* (not in more deeply nested loops) turn into `goto L32`.

The `for` statement with a null limit expression translates as follows. As above, either *e1* or *e3* can be null.

–before–
```
L33:for(e1;;e3)S14;goto L34;
```

–after–
```
L33:e1; goto L35;
L35:S14; goto L36;
L36:e3; goto L35;
```

The `continue` statements in *S14* (and not in nested loops) turn into `goto L36`.

Transforming `if` and `else`

There are two forms of the `if` statement, with and without `else`. The control expression may be separated out to make them available to the expression statement flattener.

–before–
```
L37:if(e) S15;
else S16; goto L38;
```

–after–
```
L37:L39 = e; goto L3A;
L3A:if(L39) goto L3B; else goto L3C;
L3B:S15; goto L38;
L3C:S16; goto L38;
```

–before–
```
L3D:if(e) S17; goto L3E;
```

–after–
```
L3D:L3F=e; goto L40;
L40:if(L3F) goto L41; else goto L3E;
L41:S17; goto L3E;
```

Dead Code

Programs in C may have unavoidable dead code. For example, the *statement* following a switch clause cannot be reached because the switch is an unconditional transfer. After the flattening transformations, this dead code, and other dead code introduced by the transformations, shows up as labels that are never referenced. The flat fragments starting with unreachable labels may be deleted without changing the effect of the program (and perhaps avoiding compiler-generated or lint-generated complaints). Mechanical means to recover the original text may be defeated by such removal—something must be invented to replace the removed stuff.

Similarly, arbitrary dead code can be inserted behind unreachable labels to further obscure meaning.

Order of Flattening

Flattening takes place in a specific order because of the interdependence of the steps.

1. Rename all but external names with `LXXXXXXXX` names.

2. One function at a time:
   (a) Replace all initializations with assignments.
   (b) Move all declarations to the outermost function scope.
   (c) Label the statements in the outermost statement list.
   (d) Flatten one statement at a time, applying transformations until the statement is a sequence of flat fragments.
   (e) Remove dead code.
   (f) Clot local variables and flatten expressions.
   (g) Sort the flat fragments by leading label.
3. Clot uninitialized static variables.

Testing Compilers

Testing requires generating a test, running the test, and evaluating the result against a criterion for correctness. The first and third steps are often labor intensive, so much so that providing them is a reasonable commercial activity. For compilers in particular, because there is no one correct output (object file), the test criterion is correct behavior of the compiled program. Any substantial testing of compilers, except for correctness and quality of error diagnosis, is necessarily indirect.

Suppose some arbitrary C program compiles and runs, producing some evidence of its behavior (trace, output, dump). The C♭ transformation may be applied to the C program, the program compiled and run again. Now it is required that the two outputs are identical. Thus the problem of generating a test is reduced to selecting any existing C program. The problem of evaluating the test is reduced to a file comparison. When the flattener has optional transformations, more than one test can be derived from each C program.

The assumption underlying this test strategy is that flattening causes the compiler to take wildly different paths, and that comparing the results of those paths allows the compiler to be tested against itself. It is also true that flattened code would never be written by a programmer, so it tests otherwise unexercised functionality in the compiler.

In fact each use of C♭ for making a distributable program falls into the above pattern. Surely the engineer is going to test the transformed program against the original. When a difference does appear it is not clear how one would go about locating the problem.

Summary

Nearly every Standard C program can be transformed into an equivalent program in C♭. C♭ is a reasonable low-level intermediate language, obscures engi-

neering design information, can be compiled by standard compliant compilers, and provides a new kind of compiler testing.

References

[1] American National Standard for Information Systems—*Programming Language C*, X3.159-1989, American National Standards Institute, New York, NY (1989).

[2] *ANDF, Application Portability, and Open Systems*, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, June 1991.

[3] Jim Gimpel, *The C Shroud*, Gimpel Software, 3207 Hogarth Lane, Collegeville, PA 19426.

[4] Rex Jaeschke, *Miscellanea/Encrypting C Source for Distribution*, *Journal of C Language Translation*, Vol 2, No. 1 (June 1990).

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

∞

23. European C Conformance Testing

Neil Martin
British Standards Institution

In June of this year we held our second Test Method Control Executive (TCME) Meeting. The TCME is the committee tasked with controlling the content of the test suite used in formal validation. For the European Testing Services (CTS-C) this is the Plum Hall C Validation Suite. Participation in the TCME is open to all, but in practice it is largely users of the CVS, certification bodies, plus one or two others that participate. Decisions are made on a consensus basis.

Last year we held our meeting in London, but this year we wished to get input from the large number of CVS users in the U.S. The second TCME meeting was therefore held in the historical setting of the Colonial Inn in Concord, Mass. (Note some considered this to be a dangerous activity for the British as there was once a tendency for Americans to shoot at us there.)

The meeting itself concentrated mainly on technical issues although, not surprisingly, the issue of mutual recognition of certificates between the CTS-C service and the forthcoming NIST service was raised. Mutual recognition of certificates is now a big issue in Europe, Japan, and the USA, for the simple reason that the absence of agreement creates trade barriers. This is true not just in specific software related areas such as compiler and POSIX certification, but across the board.

Within Europe the whole issue of trade barriers is high on the political agenda with 1992 approaching rapidly. In fact, within Europe an organisation known as the European Organisation for Testing and Certification (EOTC) has been tasked with harmonising cross-border certificates of conformity. Although most of the activity of EOTC relates to testing and certification, the actual goal behind it all is the free trade of products. This relates as much to products originating outside the EEC as within it.

The current status of EEC vs. USA C compiler certificates is that the two parties agreed at a meeting in May, at NIST, to work towards mutual recognition of certificates. This means that NIST and the CTS-C service need to come up with a satisfactory means of demonstrating that conformance testing with different test suites can give comparable results. Current timetables are for a plan by August with implementation by October. The Japanese have now chosen to follow the CTS-C approach and also use the Plum Hall test suite for their testing service.

The rest of the meeting was dedicated to practical issues regarding the test suite and the content of tests. Some of the issues raised sufficient doubt with those present that a formal interpretation request will be made to X3J11. In

general, vendors wanted as many test cases as possible both of a conformance and of a quality nature. The compromise situation is that any new tests will have a status of "pending" for one year to allow for debate and objection before being included in the formal test process. It is hoped that this will give sufficient time for vendors to implement any necessary changes.

One of the more interesting subjects discussed related to hardware architectures that are accurate to only 6 digits (decimal). In some embedded systems 10 digits of accuracy cannot therefore be reasonably obtained (or possibly required). The test suite currently judges two floating-point numbers to be "sufficiently equal" if their `DBL_DIG-1` decimal digits of representation are equal. (This is referred to as *one decimal digit of fuzz*.) Two questions therefore arise:

1. Is one decimal digit of fuzz based on the compiler vendors' value of `DBL_DIG` satisfactory?

2. Can a compiler based on an architecture that is only capable of working to six significant digits be standard conforming?

Neil Martin is a Principal Software Engineer at BSI Quality Assurance in Milton Keynes, United Kingdom. He may be reached electronically via neil@bsiqa.uucp.

∞

24. Massively Parallel C: Architectures and Data Distribution

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

Abstract

The usability of massively parallel systems to solve real problems is growing. One popular memory system design for massively parallel systems distributes the memory with the processors. Mapping data structures, such as arrays, onto distributed memories requires the individual elements to be distributed, such that each processor owns part of the data structures. Careful data distribution can permit a processor to spend most of its time referencing array elements that are located in its fast local memory. Since data distribution syntax does not exist in Standard C, linguistic enhancements are needed to support the mapping of data structures onto distributed memory systems. One possible approach is presented that defines syntax for declaring arrays with regular data distributions across the distributed memory.

Introduction

Amdahl's Law tells us that program speedup is asymptotically limited to the amount of time spent executing the sequential region. For example, if an algorithm that requires 1000 time steps when executed sequentially permits 800 time steps to execute in parallel, the speed up approaches 5 times *not* 800 times. However, the expense of 800 processors was required to achieve this speedup. Explaining Amdahl's Law this way is an oversimplification of the behavior of real parallel programs. One problem with this explanation is that the efficiency is relative to the performance of the program running on a single processor. The single processor implementation may itself be inefficient relative to the peak performance of the processor. In particular the parallel region, which we assume can distribute work across all the processors, itself has overhead associated with it just to get additional processors synchronized and executing. Once all the processors are executing, there is memory contention, network communication costs, and synchronization overhead. The effect of all this overhead is that the maximum speedup is often less than the number of processors, and a

point of diminishing returns is reached beyond which adding more processors actually decreases performance. A reasonable conclusion from this is that a parallel program should be designed to run on as many processors as possible, but the implementation bottlenecks may force a change in algorithm or data layout to achieve the best efficiency.

Much of the current parallelism focus is on hardware architecture, especially for massively parallel systems. The belief is that the classic von Neumann architectures are not capable of solving the problems of tomorrow, and that massively parallel systems offer peak performance capability not available by any other technology. Another belief is that RISC processors, improvements in memory technology, and better communication networks have sufficiently reduced the hardware impediments to usable parallel processing.

The National Science Foundation has identified Grand Challenge Problems and U.S. taxpayers will fund attempts to conquer them. Grand Challenge Problems attempt to solve compute intensive and heretofore unsolved problems like long-range climate modeling, semiconductor and superconductor modeling, quantum chromodynamics, and mapping the human genome. Any research organization, wishing to receive funding for these problems must be willing to include a massively parallel architecture in their approach.

Since my background is primarily in programming languages, my interest lies in the language extensions and execution environments needed to support these new architectures. Even if hardware technology has advanced sufficiently, it is not at all clear that software is in the same position. What are the issues that confront implementors defining a parallel programming model? Before language extensions are explored, an understanding of massively parallel architectures and the current limitations of existing languages is necessary. Massively parallel systems are often harder to program, run many applications slower than sequential machines, and permit execution time errors that are not readily reproducible.

When considering programming languages for massively parallel systems, one can take the extreme position that additional compiler technology can extract parallelism from "dusty deck" programs and programmers can continue to use existing sequential languages. Another extreme position is that new functional languages will eliminate dependence problems and allow parallelism to be expressed without side-effects (caused by assignments) getting in the way. Both positions have merit but also leave scientists and engineers with either inadequate compilers or languages that are not based on standards. It is my belief that near-term practical solutions must build on existing language standards even if linguistic purity is compromised along the way. Most modern architectures were built knowing which programming languages are primarily used by customers. Architectural features were added to enhance the performance of these languages. Massively parallel systems are designed around available hardware technology that offers very high peak performance, and the programming languages are expected to catch up. Since language standards are a treaty between the implementors and the programmers, it is essential to understand

the needs of both. This is the first in a series of articles that outlines the issues of extending C for massively parallel architectures.

Architectures

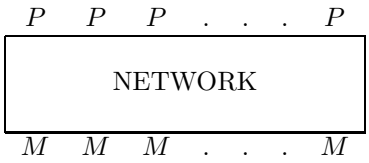
Access to the parallel capability of many existing parallel machines is through automatic parallelization by the compilers, directives inserted by the programmer, and array syntax. Many supercomputers have vector registers that permit a form of Single Instruction Multiple Data (SIMD) parallelism that is often referred to as vectorization. Array processors are another form of SIMD architecture. More recently, arrays of Processing Elements (PEs) have been connected by networks to create another kind of SIMD architecture. The term SIMD is used to describe parallelism where a single operation is applied to many data elements. A single instruction stream is combined with multiple data streams to produce a result. SIMD offers a simpler synchronization mechanism because the operations are kept in lock step by the instruction issue mechanism. Even with multiple PEs there is no need for user-specified synchronization because every PE is executing the same instruction at the same time. SIMD architectures map nicely onto data parallel operations, like those defined by a Fortran-90 style of array syntax, because multiple array elements are operated on in parallel.

Another kind of parallel architecture is called Multiple Instructions Multiple Data (MIMD). These architectures use a network to combine several general purpose processors that are capable of executing their own separate instructions on their own data. In a MIMD machine, each PE must be capable of storing its own instruction stream (or at least that part that it will execute) and contain enough intelligence to decode and execute the instructions. MIMD architectures require synchronization primitives to control communication and coordination between the PEs. MIMD is more general but also more complicated. Sometimes SIMD is compared to a puppet and MIMD to a robot.

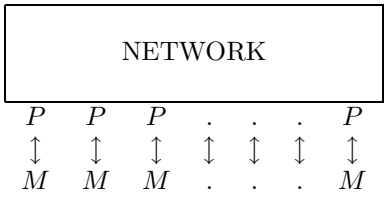
There is another kind of architecture called Very Long Instruction Word (VLIW). Execution of instructions occur in lock step, but each PE can execute a different instruction. An instruction word carries many instructions. Because there is one instruction for each PE, it is not necessary for each PE to be capable of storing an instruction stream. This places VLIW somewhere in between SIMD and MIMD.

Memory Systems

There are two common ways of arranging multiple processing elements P with respect to memory modules M. The *dancehall* configuration, shown below, is used for designs with shared memory architectures.



This design permits every processor P to have equal access to every memory module M. All of memory takes the same amount of time to reference no matter which PE makes the reference. Another configuration is the *boudoir* configuration, shown below, used with distributed memory architectures.



In this configuration, each processor P has direct access to to its own local memory module M, but must send a request across the network in order to reference a remote processor's memory. The amount of time it takes to reference a remote memory location depends upon how far away it is. Implementations may provide a *memory latency tolerance* feature that decreases the marginal cost of referencing a block of remote memory locations. The most distinguishing difference between shared memory designs and distributed memory designs is the cost function associated with memory references. Fast access shared memory designs are complicated and expensive to build but provide a more general programming model. Distributed memory machines force the programmer to maximize the locality of data references in order to achieve high performance. In order to maximizing locality, programmers must specify the data layout of major data structures across the PEs. Specification of exact data layout is an additional programming burden that programmers of massively parallel machines must endure. However, distributed memory machines are simpler and cheaper to build, and provide good peak performance.

Again, one can take an extreme position that the compiler should optimize for data locality, freeing the programmer from this burden. Since this kind of compiler technology is not currently available, short term solutions are needed. Traditionally, C programs have executed on sequential processors with monolithic memories. Memory systems contained one or more monoliths and all data objects were expected to fit entirely within one monolith, or possibly several contiguous monoliths. The desire for higher peak performances is changing these traditions.

Interconnection Networks

Networks connect all the PEs together. Interconnection Networks are not new, having been around at least as long as telephone systems. The network allows PEs to communicate with each other. Networks provide data sharing mechanisms, synchronizations, and useful primitives such as global broadcasts, compare and swaps, and fetch and adds. The network is an integral part of the computational model. The primary factors to consider when evaluating a network are:

I/O Latency The time it takes for data to travel from one PE to another.

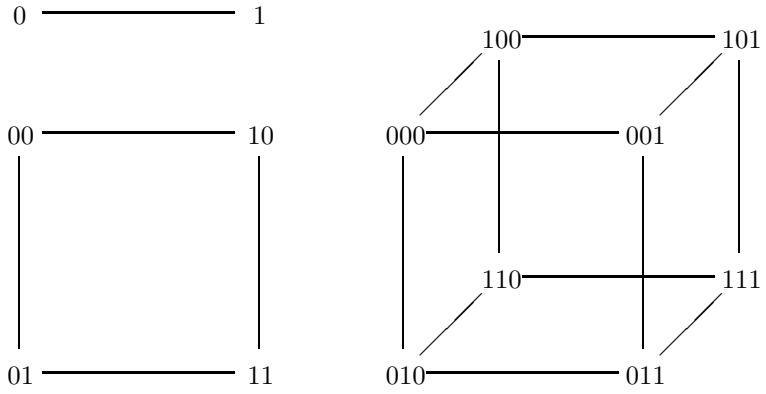
I/O Bandwidth The amount of traffic the network can handle.

Network Diameter the maximum of the shortest distances between all pairs of processors.

Bisection width the number of wires that must be cut to separate the network into two sub-networks of equal size.

The I/O constraints indicate how the network performs and how easily the network can become flooded. It is important to get enough data in as short a time as possible to keep the PEs busy. A small network diameter minimizes the amount of time data spends traveling across the network. A network with a small bisection width is cheaper to build but may also have a large diameter. There are many trade-offs to evaluate when deciding which network is best.

An example of one popular network is the *hypercube*. The hypercube is a versatile and efficient network that is a popular choice for multipurpose parallel machines. A 1-D hypercube has two PEs and one connection, a 2-D hypercube has four PEs and four connections, and a 3-D hypercube has eight PEs and twelve connections. They can be represented pictorially as follows:



In general an *n-dimensional hypercube* has 2^n PEs and $n \times 2^{n-1}$ edges. A convenient PE numbering scheme is a cyclic Gray code because consecutive numbers differ by a single bit. This allows all nearest neighbor PEs to also differ by one bit. Traversing the network is accomplished by noting which bits differ, and traveling in that direction. For example, one algorithm for traversing from source PE[010] to destination PE[001] is accomplished by noting that the first bit is 0 in both the source PE and the destination PE. This means no traversal is necessary because both nodes already reside on the same plane of the first dimension. Next, the second bit is examined, revealing a 1 in the source PE but a 0 in the destination PE. A traversal is necessary when the bits differ. The first traversal is from PE[010] to PE[000]. The third bit also differs between the source and destination PEs. After the second traversal from PE[000] to PE[001], the destination has been reached. Two traversals (commonly called *hops*) were needed to reach the destination PE. The number of hops to reach the destination PE is equal to the number of bit positions that differ between the source and the destination. Three hops are required to travel from PE[010] to PE[101] because all three bit positions differ.

The advantage of a hypercube is that it can efficiently simulate other networks of the same size because of its small diameter ($log_2 N$). Algorithms that are efficient for those networks are automatically efficient on a hypercube. The disadvantage of the hypercube is that the number of edges to each PE grows exponentially with the size of the network. The cost of the network becomes prohibitive as the size of the network increases because the bisection width becomes large ($N/2$). To overcome this scaling problem, vendors choose hypercube derivatives such as butterfly networks or tori. Understanding the network is important to figuring out how to minimize the communication costs of a parallel algorithm.

Data Distribution

A distributed memory architecture adds another layer to the layout of memory. Now the address of a data object consists of two pieces of information, the PE number and the address on the PE. In some ways this is similar to segmented architectures that require a segment number and an offset into the segment. However, massively parallel systems must provide ways to distribute the elements of an array across all the PEs that are working on a particular solution. Many parallel solutions involve divide and conquer algorithms where distributing the data is as important as distributing the work to achieve high performance. Effective data distribution maximizes memory references that are local to the PEs and minimizes global memory references across the network. An effective data distribution method for one algorithm may not be effective for another. Effective distribution methods also vary from one massively parallel system to another because different interconnection networks have different optimal inter-processor communication schemes. Consider the following example:

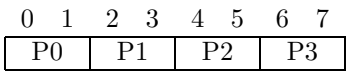
```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

The two outer loops have N^2 independent iterations that can execute in parallel because each iteration is computing a different element of matrix `c`. These loops are called the `i` and `j` loops because they are the loop control variables. The method used to distribute matrices `a`, `b`, and `c` can have a significant effect on performance. Since the inner `k` loop references row `i` of matrix `a` and column `j` of matrix `b` every time an element of matrix `c` is computed, it is reasonable to keep row `i` of `a` on the same PE as column `j` of `b` to help maximize locality. The memory latency tolerance feature can be used to fetch remote rows and columns while each PE computes `c[i][j]` values with local elements. However, different implementations may require different layouts to maximize locality. Programmers need a convenient mechanism for controlling the data layout of major data structures like matrices.

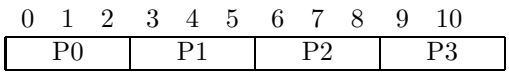
Traditional programming languages do not allow programmers to define the distribution of data across a number of PEs. Much current research is devoted to exploring ways to distribute the elements of arrays by extending Fortran-77 and Fortran-90. These extensions have some applicability to Standard C. The distribution methods provide ways for specifying the distribution of different dimensions and different elements. Consider the problem of trying to specify that an array of N elements should be distributed across P processors such that each processor contains a block of contiguous elements, and each block has the same number of elements. One approach might be to introduce a new keyword, `block`, that can be used in an array declaration as follows:

```
double a[N ; block];
```

The `;` token terminates the size expression of the array declaration and the `block` keyword specifies that each PE is allocated a block of contiguous elements. The number of elements in each block is called the *block-size* and is defined to be N/P where P is the number of PEs the data is spread across. Consider the case where $P = 4$ and $N = 8$. The data is distributed as shown in the following figure.



In this example the block-size is 2 with the first two elements residing on P0, elements 2 and 3 residing on P1, elements 4 and 5 residing on P2, and the last two elements residing on P3. Now consider the case where $P = 4$ and $N = 11$.



Note that in this example, N is not evenly divisible by P. In this case the ceiling of N/P is used to determine that the block-size is 3. Note that the last PE does not contain as many elements as the others.

Permitting arbitrary block-sizes gives greater flexibility in data distributions. For example:

```
double a[N ; block(2)];
```

declares the block-size to be 2. If $N = 32$ and $P = 4$ then blocks of 2 elements each cycle across the PEs in a round-robin fashion as follows:

```
Element:          11111111112222222223
                  01234567890123456789012345678901
P:                00112233001122330011223300112233
```

There are 4 cycles through the PEs. A completely cyclic distribution is possible by specifying a block-size of 1 as follows:

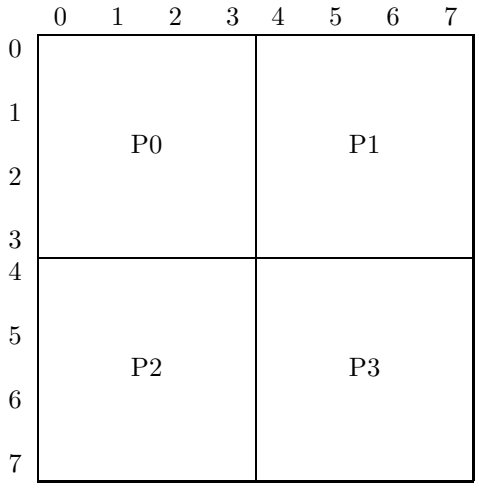
```
double a[32 ; block(1)];
```

```
Element:          11111111112222222223
                  01234567890123456789012345678901
P:                01230123012301230123012301230123
```

Dimensional distributions are also defined for multidimensional arrays. Again assume that $P = 4$ for the following example:

```
double x[8 ; block][8 ; block];
```

The distribution is pictorially represented as:



This can be conceptually viewed as assigning two PEs to each dimension and 16 elements to each PE. Other reasonable two dimensional distributions are:

```
double x[8 ; block(1)][8 ; block];
```

	0	1	2	3	4	5	6	7
0	P0				P1			
1	P2				P3			
2	P0				P1			
3	P2				P3			
4	P0				P1			
5	P2				P3			
6	P0				P1			
7	P2				P3			

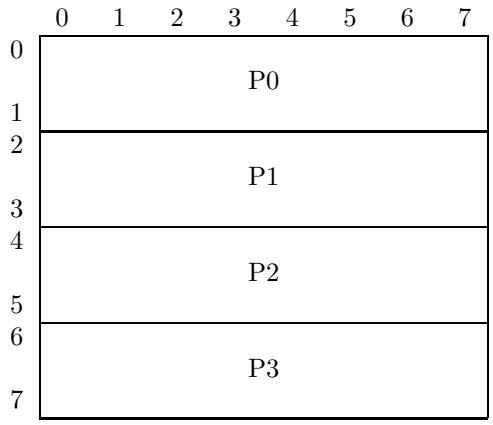
```
double x[8 ; block(2)][8 ; block(4)];
```

	0	1	2	3	4	5	6	7
0	P0				P1			
1								
2	P2				P3			
3								
4	P0				P1			
5								
6	P2				P3			
7								

The last example declares block-sizes of 2 and 4 for the first and second dimensions, respectively. This essentially creates tiles of 2×4 elements that are cycled through all the PEs.

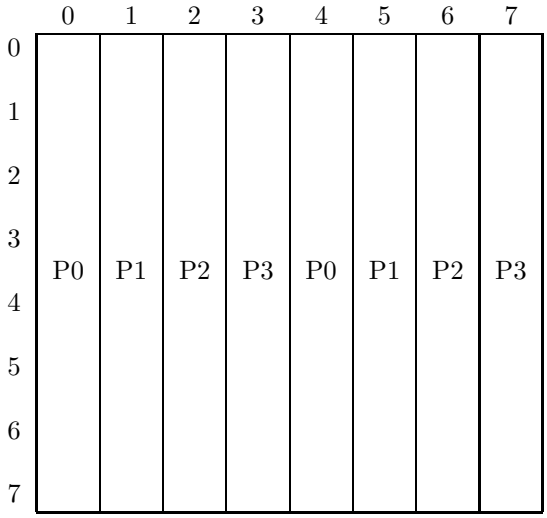
It is also possible to define the block-size to be the same as the number of elements in that dimension. This means that all elements of that dimension should be resident on a PE, and the PEs should be assigned to other dimensions. The following example places a block of consecutive rows on each PE.

```
double x[8 ; block][8 ; block(8)];
```



As a syntactic convenience the token : refers to a block-size that is the size of the dimension. The following example places a column on a PE and cycles through all the PEs.

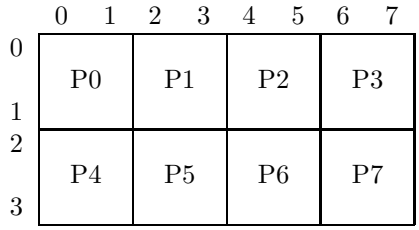
```
double x[8 ; : ][8 ; block(1)];
```



Another data distribution consideration is the mapping of the PEs onto the array dimensions. By convention, all of the 2-D examples above used $P = 4$ and mapped 2 PEs onto each dimension (excluding the : dimensional distributions). However, it might be desirable to weight the PE map such that one dimension gets more PEs than another. For instance, if $P = 8$ then one dimension can be assigned 4 PEs and the other 2 PEs. The following syntax could be used for this purpose:

```
double y[4 ; 1:block][8 ; 2:block];
```

The `1:` and `2:` specifiers are *weights* that indicate how the PEs should be mapped onto the dimensions. The example above indicates that half as many PEs should be assigned to dimension 1 than are assigned to dimension 2. The distribution is pictorially represented as:



This distribution allows 4 elements to reside on every PE. A dimension declared without a weight has a default weight of 1. If no weights are specified in the declaration then an equal number of PEs are assigned to each dimension.

The following declarations show two arrays such that corresponding rows of array `a` reside on the same PE as corresponding columns of array `b`.

```
double a[N ; block][N ; : ];
double b[N ; : ][N ; block];
```

The data distributions described above are called *regular* distributions because each PE is allocated the same number of blocks, and each block has the same number of elements. A variety of regular distributions is available with these syntactic extensions. For many problems these distributions are adequate to maximize locality. Declarations of *irregular* distributions involve the specification of an additional level of indirection. Irregular distributions allow the user to specify that certain PEs have more blocks than other PEs, or that some blocks have more elements than others. They are useful for sparse array applications but are not considered further in this article.

Array References

A reference to an array element involves calculating where the element resides, and the offset into that PE. Assume that $P = 4$ in the following example:

```
double a[32 ; block];

for (i = 0; i < 32; i++)
   a[i]++;
```

For this example the block-size is 8 because four equal sized blocks of 8 elements each can reside on each PE. This means that `a[0]` resides on P0 with offset 0 from the base address of `a`, and `a[9]` resides on P1 with offset 0 from the base address of `a`. Every PE has its own base address of `a` and this base

address must be at the same location on every PE. This makes the calculations associated with array references more complicated. Traditionally, the array reference `a[i]` is computed by first adding `i` to the base address of `a` and then referencing that memory location. With data distribution the calculation is more involved. The index expression `i` contains two pieces of information, a PE number and an offset in that dimension. Both pieces of information must be extracted. First `i` must be divided by the block-size, `i/8`, to find the PE on which the element resides. The offset is obtained by computing `i%8` (because 8 is the block-size). It is apparent that having block-sizes and dimension sizes that are powers of two reduces the divide to a right shift instruction and the modulus to a bitwise-AND with a mask. The offset can now be added to the base address of `a` on the identified PE to reference that memory location.

The following 2-D example is more complicated. Assume that $P = 4$.

```
double z[16 ; block(4)][8 ; block(2)];
```

Equal number of PEs are assigned to each dimension because no weights are specified. The number of PEs assigned to dimension 1 is 2 and the block-size is specified to be 4. The number of PEs assigned to dimension 2 is also 2 and the block-size is specified to be 2. Given this information it is possible to compute the PE number and offset for any `z[i][j]` expression.

The following program computes the PE number and an offset that can be added to the base of any two-dimensional array that is distributed in a regular way. The required input for each dimension is the the dimension size (extent), number of PEs assigned to the dimension, and block-size for the dimension, Since this program processes two dimensional arrays, six numbers must be specified on the command line. It is possible to see one-dimensional distributions by entering `1 1 1` as the last three or first three numbers. The results for array `z` in the example above are shown after the program is discussed. The `main` function reads in the extents, block-size, and number of PEs, one-dimension at a time. Next it calls functions to compute PE number and offset for every element of the array.

```
#include <stdio.h>
#include <stdlib.h>
static const int rank = 2;

main(int argc, char **argv) {
   int PE_num, PE_off, i, j, offset[2];
   int PEs[2], bk_sz[2], extents[2], indices[2];
   if (argc != 7) {
      printf("Usage: %s extent[1] bk_sz[1] PE[1]"
             " extent[2] bk_sz[2] PE[2]\n", *argv);
      exit(1);
   }
```

```
   for (i = rank-1; i >= 0; i--) {
      extents[i] = atoi(*++argv);
      bk_sz[i] = atoi(*++argv);
      PEs[i] = atoi(*++argv);
      offset[i] = (extents[i]+PEs[i]-1) / PEs[i];
   }

   for (j = 0; j < extents[0]; j++)
      printf("%7d", j);
   putchar('\n');

   for (i = 0; i < extents[1]; i++) {
      printf("%2d", i);
      for (j = 0; j < extents[0]; j++) {
         indices[1] = i;  indices[0] = j;
         PE_num = compute_PE(indices, PEs, bk_sz);
         PE_off = compute_off(indices, PEs, bk_sz, offset);
         printf(" (%1d,%2d)", PE_num, PE_off);
      }
      putchar('\n');
   }
}
```

The `compute_PE` function extracts the PE field from each index and computes the PE number that the element resides on. This is done by dividing by the block-size and 'mod'-ing with the number of PEs assigned to that dimension. Finally, the contribution of this index to the PE number must be scaled appropriately. The scaling is done by setting `scale` to 1 initially and accumulating the partial products of the number of PEs assigned to each dimension.

```
compute_PE(int *ix, int *NP, int *bk_sz) {
   int P = 0, scale = 1, i;

   for (i = 0; i < rank; i++) {
      P += (ix[i] / bk_sz[i] % NP[i]) * scale;
      scale *= NP[i];
   }
   return P;
}
```

The function `compute_off` computes the offset into the PE by performing the following operations for each dimension. First, extract the block-size field from the index with a modulus operation. Second, eliminate the block-size and PE fields by dividing them out, leaving the cycle field. Third, scale the cycle field by the block-size add it to the block-size, producing the contribution of this index to the local offset. Finally, scale this dimension's contribution to the

local offset appropriately and add it to the accumulated offset.

```
compute_off(int *ix, int *NP, int *bk_sz, int *offset) {
   int local_off = 0, scale = 1;
   int bk, cyc, i;

   for (i = 0; i < rank; i++) {
      bk = ix[i] % bk_sz[i];
      cyc = ix[i] / (bk_sz[i] * NP[i]);
      local_off += (bk + cyc * bk_sz[i]) * scale;
      scale *= offset[i];
   }
   return local_off;
}
```

The output for array `z` in the example above is obtained by running the program as follows:

```
a.out 16 4 2 8 2 2

      0      1      2      3      4      5      6      7
 0 (0, 0) (0, 1) (1, 0) (1, 1) (0, 2) (0, 3) (1, 2) (1, 3)
 1 (0, 4) (0, 5) (1, 4) (1, 5) (0, 6) (0, 7) (1, 6) (1, 7)
 2 (0, 8) (0, 9) (1, 8) (1, 9) (0,10) (0,11) (1,10) (1,11)
 3 (0,12) (0,13) (1,12) (1,13) (0,14) (0,15) (1,14) (1,15)
 4 (2, 0) (2, 1) (3, 0) (3, 1) (2, 2) (2, 3) (3, 2) (3, 3)
 5 (2, 4) (2, 5) (3, 4) (3, 5) (2, 6) (2, 7) (3, 6) (3, 7)
 6 (2, 8) (2, 9) (3, 8) (3, 9) (2,10) (2,11) (3,10) (3,11)
 7 (2,12) (2,13) (3,12) (3,13) (2,14) (2,15) (3,14) (3,15)
 8 (0,16) (0,17) (1,16) (1,17) (0,18) (0,19) (1,18) (1,19)
 9 (0,20) (0,21) (1,20) (1,21) (0,22) (0,23) (1,22) (1,23)
10 (0,24) (0,25) (1,24) (1,25) (0,26) (0,27) (1,26) (1,27)
11 (0,28) (0,29) (1,28) (1,29) (0,30) (0,31) (1,30) (1,31)
12 (2,16) (2,17) (3,16) (3,17) (2,18) (2,19) (3,18) (3,19)
13 (2,20) (2,21) (3,20) (3,21) (2,22) (2,23) (3,22) (3,23)
14 (2,24) (2,25) (3,24) (3,25) (2,26) (2,27) (3,26) (3,27)
15 (2,28) (2,29) (3,28) (3,29) (2,30) (2,31) (3,30) (3,31)
```

The address of every element is printed as a pair (PE, offset) which can be used to reference the memory location assigned to that element. It is possible to see the 4×2 tiling of elements on each PE.

Conclusions

It is highly probable that sometime in the next decade massively parallel systems will be commonplace tools to most scientists and engineers. The architectural simplicity of PEs with local memories connected by a network is a trade-off that is being accepted over a simpler programming model. The programming model proposed here imposes the burden of declaring data layout specifications on programmers wishing to use data structures that span the entire network. The data distribution program demonstrates how complicated it is to calculate the PE number and the offset required to address elements of an arbitrary regular distribution. The expense associated with irregular distributions is even higher. The complications associated with address computation exist because a pointer value represents both a PE number and a location. Pointer arithmetic requires knowledge of the block-size and the number of PEs over which the data is distributed. Since array subscripting is defined in terms of pointer arithmetic and pointer dereferencing, implementations might want to require dimension sizes, block-sizes, and the number of PEs over which the data is distributed to all be powers of two. Powers of two permit strength reduction optimizations to shifting and masking. Long term solutions may be found to ameliorate this restrictive requirement but short term needs for performance may force these power-of-two sizes for regularly distributed arrays. These trade-offs are being explored in hopes of reaching the peak performances capable of solving tomorrow's problems with a usable programming model.

Recommended Reading

George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc. (1989)

F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kauffman, (1991)

G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D Language Specification*, Technical Report TR90-141, Dept. of Computer Science, Rice University, (April 1991)

Barbara M. Chapman, P. Mehrotra, and H. Zima, *Vienna Fortran – A Fortran Language Extension for Distributed Memory Multiprocessors*, Department of Statistics and Computer Science, University of Vienna. (September 1991)

Tom MacDonald is the Numerical Editor of The Journal of C Language Translation. He can be reached at (612) 683-5818 or tam@cray.com.

∞

25. C/C++ Compatibility

Paul Kohlmiller
Control Data Corporation

Abstract

This paper identifies more cases of ANSI C programs that cannot be compiled with a C++ compiler. As with the last installment, each case is documented with section numbers from the ANSI C Standard and the Annotated C++ Reference Manual.

18. In C++ it is invalid for a function to promise to return a value and then not deliver.

```
int f(int x) {
        x++;
}
```

C++ does not allow this for two reasons. First, it very likely is an error. Second, making such code invalid allows the compiler to make some assumptions. For example, the compiler might be able to assume that a return value is actually an address returned in a register. The compiler can generate code that assumes that the register contains an address and not some garbage left from some previous code. Relevant standards references: ANSI C §3.6.6.4, The `return` statement, and ARM §6.6.3, The `return` statement.

ANSI C++ will retain the difference. ANSI C allows this but says the behavior is undefined if you use the returned "value." Writing code in this way is to be discouraged.

19. C is happy to ignore some things that C++ considers to be in error. For example, it does not make much sense to define a structure template and give it a storage class specifier.

```
extern struct tag1 {
        int j,k;
};
```

In C++, `static` and `extern` can only be applied to names of objects and functions.

Relevant standards references: ANSI C §3.5.1, Storage-Class Specifiers, and ARM §7.1.1, Storage Class Specifiers.

ANSI C++ will keep this distinction because such code adds nothing to the language. Also, a C++ class definition could include some member functions that can be declared to be `static`. Imagine the confusion if a class is declared `static` and one or more member functions are also declared `static`.

20. When a C++ program defines a structure it is really defining a type for the block that contains the definition. This means that a structure name hides any other use of the name in the enclosing scope of the definition. Consider the following program:

```
#include <stdio.h>
main() {
        int x[89];
        struct x { int a,b,c; };

        printf("The size of x is %lu\n",
                (unsigned long)sizeof(x));

                /* size of the array in C    */
                /* size of the struct in C++ */
        }
```

The program compiles without error for both ANSI C and for C++. However, the answer that is printed out in each case is quite different. The structure definition for `x` hides the definition of the array `x` in C++ but not in C.

Relevant standard sections: ANSI C §3.1.2.3, Names Spaces of Identifiers, and ARM §9.1, Class Names.

ANSI C++ will certainly not change this. The use of structure tag names as type names is essential to C++. This allows user-defined type names to look like basic type names.

21. ANSI C and C++ will generally turn a "function returning type" to a "pointer to function returning type." Each language makes an exception for the `&` operator. C++ goes on to make an exception for the function call operator `()` too. This is not really an exception since the function call operator obviously results in a function call and the function returns something of the appropriate type. ANSI C makes an exception when the function is the operand of the `sizeof` operator. The result is that trying to find `sizeof` a function gives a diagnostic. C++ simply says that the operand of the `sizeof` operator may not be a function. Therefore this

code won't work with either ANSI C or C++ but you may get different diagnostics.

```
int g();
int f() {return sizeof(g);}
```

Relevant standards sections: ANSI C §3.2.2.1, Lvalues and Function Designators, and ARM §4.6, Pointer Conversions, and ARM §5.3.2, `sizeof`.

22. ANSI C allows for recursive calls to `main`. In §2.1.2.2.3 it describes how a `return` from the initial call to `main` is supposed to behave just like a call to `exit`. C++ explicitly disallows user calls to `main`.

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

int i = 0;

main() {
        i++;
        printf("Invocation number %d\n",i);
        if (i > 2) {
                printf("Leaving main\n");
                return 0;
        }
        main();
        printf("Returned from main\n");
        return 0;
}
```

C++ does not allow recursive calls to `main` because exiting from `main` invokes the destructors for all initialized static objects. Also it allows an implementation to implement `main` as something other than a function.

Relevant standards sections: ANSI C §2.1.2.2, Hosted Environment, and ARM §3.4, Start and Termination.

23. In item 12 of my September article we saw that `void *` expressions on the right-hand side of assignments were likely to cause problems. Here is a particular case.

```
main() {
        int *ptr;
        #define NULL (void *)0;
        ptr = NULL;
}
```

This will work for ANSI C where `(void *)0` is a typical definition for `NULL`. However, C++ will complain about this.

Relevant standards references: ANSI C §3.2.2.3, Pointers, and ARM §4.6.

Certainly, a lot of people want to be able to write code like `ptr = NULL`. To allow for this some C++ implementations define `NULL` to be 0 in their C++ header files. This leads to header files with code like this:

```
#ifdef __cplusplus
#define NULL 0
#else
#define NULL (void *) 0;
#endif
```

There is some sentiment to allow code like `ptr = (void *)0;` However, this opens up uses of `void *` on the right hand side of an assignment. Of course, a compiler could special case zero cast to `void *` but then you have type checking being done based on values not types. I doubt that ANSI C++ will change this situation.

Acknowledgements

Other lists of incompatibilities between C and C++ exist. One such appears in the back of Lippman's book (*The C++ Primer*). Another is in the *C++ Annotated Reference Manual* by Ellis and Stroustrup. See also *C++: As Close to C as possible but no closer*, *C++ Report* March 1989, by Koenig and Stroustrup.

My list of incompatibilities comes partly from the above references and partly from the work of the ANSI C Compatibility subgroup within the X3J16 Committee. I am particularly indebted to Josee Lajoie, Thomas Plum, Gavin Koch, and Paul Jackson for their input.

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

∞

26. Miscellanea

compiled by **Rex Jaeschke**

Problems in `stdio.h`

Each time I come across a new implementation I like to peruse the standard headers. And over the years I've developed a small set of tests to check their contents. In a recent investigation of 12 implementations that claim to either be standard-conforming or very close to it, I found that half had a problem in the way they declared the `vprintf` function family.

According to Standard C, the prototypes for these functions are as follows:

```
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
```

In the past, many implementors created their prototypes *exactly* the same as they existed in the Standard. However, as I pointed out in a paper to X3J11 some three or four years ago, **all dummy argument identifier names used in the Standard are non-conforming!** (The essence of my paper was distilled into a few examples and narrative and added to the Rationale.) Essentially, these identifiers must either be omitted, spelled with a leading underscore and a capital letter, or with two leading underscores. Of course, this rule applies to all standard headers, not just `stdio.h`.

Standard headers may be included in any order and must not be interdependent. Specifically, `stdio.h` must compile whether or not `stdarg.h` is included. This requires that the type name `va_list` be defined prior to the three `vprintf` prototypes being processed. A common "solution" is to have something like the following piece of conditional code in both `stdio.h` and `stdarg.h`:

```
#ifndef _VA_LIST
#define _VA_LIST
typedef ... va_list;
#endif
```

Unfortunately, this is **not** permitted as the name `va_list` simply is not defined for `stdio.h`. That is, the following code is standard-conforming:

```
#include <stdio.h>

void f()
{
        int va_list[20];
}
```

Another common "solution" is to have `stdio.h` directly include `stdarg.h` itself. This too is invalid since not only does it cause the name `va_list` to be defined subordinate to `stdio.h`, it also defines the names `va_arg`, `va_end`, and `va_start`, none of which is allowed to be defined when `stdio.h` is included.

The trick then is that you need something equivalent to `va_list` defined in `stdio.h` but it must be spelled in the implementor's namespace. For example, you could use something like:

```
typedef ... __va_list;

int vfprintf(FILE *, const char *, __va_list);
int vprintf(const char *, __va_list);
int vsprintf(char *, const char *, __va_list);
```

Rather than create this secret typedef name, you could simply hard-code the actual type directly in the prototypes, as follows:

```
int vfprintf(FILE *, const char *, char *);
int vprintf(const char *, char *);
int vsprintf(char *, const char *, char *);
```

In the Standard, the synopsis for each of these functions indicates that in order to call them, the user must include both of the headers `stdarg.h` and `stdio.h`. However, from the implementor's viewpoint, each header must stand alone.

Apparently, X/Open is trying to get `va_list` defined in `stdio.h`.

Calendar of Events

- January 19–22, 1992 **Principles of Programming Languages** – Location: Albuquerque, New Mexico. This is the 19th Annual ACM SIGPLAN-SIGACT symposium. For information, contact Andrew Appel at (609) 258-4627 or *appel@princeton.edu*.
- March 2–4, 1992 **X3H5: Parallel Model and Language Bindings** – Location: New Jersey. For information, contact the Vice-Chair Walter G. Rudd at (503) 737-5553, Fax: (503) 737-3014, or *rudd@cs.orst.edu*.
- March 8–13, 1992 **ISO C SC22/WG21 C++ Meeting** – Location: London, England.

- April 29–May 2, 1992 **Workshop on Parallel Compilation** – Location: Kingston, Ontario Canada. Researchers are invited to submit an extended abstract of about 2,000 words by January 13th 1992. Invitations will be made by February 15th, 1992, and final versions of papers will be required by March 15th, 1992. Workshop Chair: David Skillicorn (613) 545 6050, Fax: (613) 545 6453, or *skill@qucis.queensu.ca*.

- May 11–12, 1992 **Numerical C Extensions Group (NCEG) Meeting** – Location: Salt Lake City, Utah, hosted by DECUS. 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*.

- May 13–15, 1992 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Salt Lake City, Utah. WG14: Contact the US International Rep. Rex Jaeschke at (703) 860-0091, or *rex@aussie.com*, or the Convenor P.J. Plauger at *pjp@plauger.com* for information. X3J11: Address correspondence or enquiries to the Vice Chair, Tom Plum, at (609) 927-3770 or *uunet!plumhall!plum*.

- May 24–26, 1992 **X3H5: Parallel Model and Language Bindings** – Location: Ithaca, New York.

News, Products, and Services

- Gimpel Software is shipping V5.0 of their tools **PC-Lint** and **FlexeLint**. New features include strong type checking of typedefs and control-flow based analysis of variable initialization. (215) 584-4261.

- The **Numerical Algorithms Group, Inc.** has released the world's first Fortran 90 compiler. What makes this interesting to the C community is that the compiler, f90, is written in portable K&R C, and produces K&R C as its output. At this time, f90 is available primarily on UNIX-based workstations. NAG also sells a **scientific C library** containing support for complex arithmetic, Fourier transformations, interpolation, linear algebra, basic statistics, and regression analysis. For further information, contact NAG at 1400 Opus Place, suite 200, Downers Grove, IL 60515-5702. Phone: (708) 971-2337, fax: (708) 971-2706.

- If you are at all interested in **internationalization** of software, you might wish to pick up a copy of *Digital Guide to Developing International Software* from Digital Equipment Corporation's publishing group Digital Press. While parts of the text talk about DEC's proprietary systems there is sufficient general information to make it of interest in non-DEC environments.

- **Watcom** of Canada is shipping V8.5 of their PC-based 16- and 32-bit C compilers. They will also supply their 32-bit compiler to IBM for their forthcoming OS/2 2.0 operating system. For more information, phone: (800) 265-4555, fax: (519) 747-4971.

- **SAS Institute** has announced a Release 5.00 of the SAS/C compiler for IBM mainframes running MVS and CMS. Contact them at SAS Campus Drive, Cary, NC 27513, phone: (919) 677-8000.

- **Quantitative Technology Corporation** has announced the availability of a C library for **scientific and engineering** applications. Both Mac and PC versions are being shipped. Contact QTC at 8700 SW Creekside Place, Beaverton, OR 97005. Phone: 503) 626-3081, fax: (503) 641-6012.

- **Zortech**, implementor's of C and C++ compilers for PCs, are shipping V3.0. Included in that release is support for numerous IEEE library routines proposed by the **Numerical C Users Group**. The company has also been acquired by **Symantec**. Contact them at 4-C Gill Street, Woburn, MA 01801. Phone: (617) 937-0696, fax: (617) 937-0793.

- **MetaWare** is shipping a Windows development kit for their C compiler family. (408) 429-6382, fax: (408) 429-9273.

- The C Standard is available in Australia as document AS 3955-1991. It has been reproduced from ISO/IEC 9899:1990. Its title is *Programming languages – C* and its number is ISBN 0 7262 6970 0. The Rationale is not included. Cost is about A$122 (including handling and postage).

Standards Australia
80 Arthur Street
North Sydney, NSW 2060
Telex 26514
Phone (02) 746 4600
Fax (02) 959 3896

∞