The Journal of

C Language Translation

Volume 3, Number 2

September, 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.

The entire contents are copyright © 1991, Rex Jaeschke. No portion of this publication may be reproduced, stored or transmitted in any form, including computer retrieval, without written permission from the publisher. All rights are reserved. The contents of any article containing a by-line express the opinion of the author and are not necessarily those of the publisher nor the author's employer.

Editorial: Address all correspondence to 2051 Swans Neck Way, Reston, Virginia 22091 USA. Telephone (703) 860-0091. Electronic mail address via 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.

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

Contents

10. Floating-Point Primitives

P.J. Plauger

Abstract

The C library has always included seminumerical functions for manipulating floating-point values. Early math functions were more robust and portable for being written in terms of these primitives. IEEE 754 floating-point arithmetic brings added complexities, however. It demands a different set of primitives for dealing with infinities, not-a-number codes, and other subtleties.

This article describes a set of primitives that I have found convenient for writing the math functions of the Standard C library for IEEE-based systems. It also discusses possible extensions to C that might ease the burden of writing safe floating-point code for less sophisticated programmers.

Introduction

Floating-point arithmetic can generate a number of exceptional conditions:

- Overflow occurs when the magnitude of the result is too large to represent.
- Underflow occurs when the magnitude of the result is too small to represent.
- Loss of significance occurs when the magnitude of a sum or difference is much smaller than that of either operand.
- A domain error (such as zero divide) occurs when you specify a combination of operands for which the operation is not defined.

Even the most naive programmers soon learn to worry about these exceptions. When they occur, different computers do different things. Some terminate program execution abruptly. Others continue, but make some attempt to signal the error. They may set an indicator (such as the notorious `errno`). They may produce a result that:

- unequivocally signals a problem (such as some special code),
- equivocally signals a problem (such as zero or `HUGE_VAL`), or

- is pure garbage (such as an oversize exponent that wraps around)

If your goal is to write code that is both robust and portable, this spectrum of possibilities is dismaying. You soon learn that the only safe way to deal with exceptions is to avoid them. Test before you compute. Make sure that all floating-point operations produce unexceptional results.

At the very least, localize the points at which exceptions can occur. That lets you isolate any special code you must introduce to handle the exceptions uniformly across diverse implementations. It lowers the cost of porting code.

One of the most demanding uses for floating-point code is in writing the functions that constitute the math library. In C, these are primarily the functions declared in `<math.h>`. (You can also include the functions that convert between text representation and floating-point values.) These functions must generate the most precise answers possible for all sensible inputs. They must avoid intermediate exceptions even for the most extreme argument values. And they often must be portable in the bargain, to recoup the investment of effort across as many markets as possible.

The C library has striven to meet these goals from the outset. Dennis Ritchie and his friends knew to avoid many of the common pitfalls in writing math functions. While C has had its notorious lapses in this area, it has probably fared better than most programming languages.

Evidence of this heightened awareness lies in the C library itself. There you will find a handful of seminumerical functions. These are clearly aimed at easing the burden of the cautious numerical programmer. They let you dismantle floating-point values in various ways. You can then work with pieces that are integers or floating-point values with a more restricted range. Finally, you put the pieces back together to develop the ultimate result.

I call these functions *seminumerical* because they need not execute floating-point instructions to get the job done. You can write them in C or assembly language as if they are operating on arrays of integers. You shift, mask, and merge to manipulate the components of a floating-point value separately. True, some floating-point processors have instructions that do part or all of the job. Even for these machines, however, you might still have occasion to avoid floating-point instructions. I discuss why below.

The three most important functions in the seminumerical group are:

1. `double frexp(double x, int *pex)` – which extracts the power-of-two exponent of `x` and stores it in `*pex`, then returns the residual fraction whose magnitude now lies in the interval [1/2, 1) (or is zero).

2. `double ldexp(double x, int ex)` – which multiplies `x` by 2^{ex} and returns the result (hence undoing the damage caused by `frexp`).

3. `double modf(double x, double *pin)` – which extracts the integer part of `x` and stores it in `*pin`, then returns the residual fraction whose magnitude now lies in the interval [0, 1) and whose sign is the same as `*pin` (and `x`).

These operations play a pivotal role in implementing nearly all of the standard math functions declared in `<math.h>`.

Unfortunately, these particular functions don't quite do the whole job. I have written several math libraries over the years, most of them in C. In each case, I ended up writing a different set of math primitives. The three functions shown above always turned out to be easily expressed in terms of the primitives I chose. But I could never quite do the job the other way around.

Exceptions

The problems lie primarily in the area of exception handling. Any of the three functions can be handed an exceptional argument, at least in principle. The function `ldexp` can generate an overflow or underflow. If the functions don't land on their feet when an exception occurs, you must ensure that they never see exceptions. (Remember what I said earlier about the best way to keep floating-point code portable.)

Because they are so widely used, these are the very functions you want to have help you in writing robust code. You don't want to have to call one function to test for exceptions, then another to unpack an operand appropriately. You don't want to have to test first whether repacking is safe, then call a function to do the actual repacking. That's *not* a good recipe for writing code that is both portable *and* efficient.

Consider the function `ldexp` as the simplest example. It is often the agent that repacks the components you have manipulated separately (and safely). You can thus make it the one and only place where overflow or underflow can occur. As a seminumerical function, `ldexp` can detect an impending exception without tickling the dragon's tail. You can steer well clear of any hardware traps when you write the function.

Unfortunately, you have only limited latitude in how you write `ldexp`. The C Standard dictates its outward behavior. The function can (and must) set `errno` on a range error. It can (and must) substitute `HUGE_VAL` or zero for an unrepresentable result when a range error occurs. But it has no nice way to tell the caller that it did so. Comparing the return value against `HUGE_VAL` and zero can be both time consuming and inconclusive.

Now consider what `frexp` should do when handed the value `HUGE_VAL`. On some machines, this happens to be just a very large representable value. It *can* be represented as a power-of-two exponent and a fraction. But should it? The C Standard doesn't really say. You probably want an unpacking primitive that is smarter, and more informative, than `frexp` is allowed to be.

That's the tip of the iceberg. The real danger to shipping lies in the complexities introduced with the IEEE 754 Standard for floating-point arithmetic. That standard introduces all sorts of codes for exceptions. Besides being a finite, representable value, a floating-point operand can be:

- a signalling NaN (for *not-a-number*) that should raise an immediate exception for any operation except a simple copy,
- a quiet NaN that should percolate through to the result wherever possible,
- Inf (for *infinity*), either positive or negative, or
- zero (either positive or negative).

The C Standard suffered a few last-minute edits designed to make it *tolerant* of IEEE 754 arithmetic. `HUGE_VAL` can be represented as Inf, for example. Domain errors can be represented in a variety of ways, probably including a NaN result. Those changes are necessary, but they are arguably not sufficient. An implementation of C that endeavors to support IEEE 754 arithmetic has little guidance from the C Standard.

Categorizing Operands

The Numerical C Extensions Group has, of course, addressed some of these issues. Much of the recent discussion that I have seen centers on the problem of comparing floating-point values in the presence of NaNs. Inf and zero, of either sign, have sensible orderings defined for the comparison operators. NaNs do not. Hence, in my opinion, the expression `x < y` can be neither true nor false if either operand is a NaN. That pretty much obliges the program to raise an immediate exception to handle a NaN when executing such an expression.

The basic idea I have seen most widely discussed is to introduce a slew of additional comparison operators to the C language. An operator that begins with a bang `!` tolerates NaNs. Thus, `x !< y` is true if `y` is greater than or equal to `x` or if either operand is a NaN. Such operators eliminate the need to raise exceptions, at the cost of complexifying C even further. People might even start accusing C expressions of being cryptic.

My experience to date is that such an approach is neither necessary nor sufficient:

- You need to do more with NaNs than simply copy them or compare them safely. You may need to distinguish quiet and signaling NaNs, for example. You always want to treat them quite differently from other operands.
- You often need to treat Inf quite differently from other operands.
- You may want to distinguish plus zero from minus zero in some contexts (although personally I have reservations about the utility of minus zero).

I have found more useful a different approach. I find frequent occasion to categorize a floating-point value before I muck with it. At the very least, I want to distinguish between:

- `NAN` for a quiet NaN,
- `INF` for Inf of either sign,
- 0 for zero of either sign, and
- `FINITE` for a finite, representable value of either sign.

Here is the simplest possible example. The function `fabs` looks to be trivial. In principle, one could write it as:

```
/* compute absolute value */

double fabs(double x)
        {
        return (x < 0.0 ? -x : x);
        }
```

In practice, this code is a sucker for NaNs. So I introduced the function `int _Dtest(double *px)` that categorizes `*px` seminumerically. The header `"xmath.h"` declares `_Dtest` and defines macros for the various integer category codes it returns, as indicated above. (In principle, the return value is an enumeration.) Now `fabs` can be written safely as:

```
#include "xmath.h"
/* compute absolute value */

double fabs(double x)
        {
        switch (_Dtest(&x))
                {       /* test for special codes */
        case NAN:
                errno = EDOM;
                return (x);
        case INF:
                errno = ERANGE;
                return (_Inf._D);
        case 0:
                return (0);
        default:        /* finite */
                return (x < 0.0 ? -x : x);
                }
        }
```

(`_Inf` is a union that is initialized as an array of `short` then accessed as the `double` code for infinity.) This version is not nearly as fast or elegant as the obvious version, but it works better.

Here is what _Dtest looks like. The various funny macro names that begin with _D define machine-dependent properties of the floating-point representation. They correct for changes in byte order among various IEEE 754 implementations. They also tolerate a few similar formats, such as the PDP-11/VAX-11 floating-point format:

```
/* _Dtest function -- IEEE 754 version */
#include "xmath.h"

/* categorize *px */

short _Dtest(double *px)
        {
        unsigned short *ps = (unsigned short *)px;
        short xchar = (ps[_D0] & _DMASK) >> _DOFF;

        if (xchar == _DMAX)     /* NaN or INF */
                return (ps[_D0] & _DFRAC || ps[_D1]
                        || ps[_D2] || ps[_D3] ? NAN : INF);
        else if (0 < xchar || ps[_D0] & _DFRAC
                || ps[_D1] || ps[_D2] || ps[_D3])
                return (FINITE);        /* finite */
        else
                return (0);     /* zero */
        }
```

If you want to handle signaling NaNs (which I chose not to do), here is the place to do so. You call _Dtest only when you intend to muck with a floating-point value. Hence, this function can raise a floating-point exception for you. It would then return NAN only for quiet NaNs.

I also found it convenient to introduce the macro DSIGN. It tests the sign bit of a floating-point value seminumerically. Thus, it can safely field NaN and Inf codes. It can also correctly distinguish plus and minus zero, a distinction otherwise difficult to make with C comparison operators.

Primitives Revisited

The other math primitives follow the model established by _Dtest. All are tolerant of the various IEEE 754 exception codes. All attempt to do something sensible with these various codes. All return a category code to guide the caller in its subsequent actions. Thus, it is necessary to call _Dtest only when none of the other common primitives are needed.

Here, for example, is a more robust substitute for frexp. The function _Dunscale unpacks an operand only if it is finite. Otherwise, it returns the appropriate category code:

```
#include "xmath.h"

/* separate *px to 1/2 <= |frac| < 1 and 2^*pex */

short _Dunscale(short *pex, double *px)
        {
        unsigned short *ps = (unsigned short *)px;
        short xchar = (ps[_D0] & _DMASK) >> _DOFF;

        if (xchar == _DMAX)
                {       /* NaN or INF */
                *pex = 0;
                return (ps[_D0] & _DFRAC || ps[_D1]
                        || ps[_D2] || ps[_D3] ? NAN : INF);
                }

        else if (0 < xchar || (xchar = _Dnorm(ps)) != 0)
                {       /* finite, reduce to [1/2, 1) */
                ps[_D0] = ps[_D0] & ~_DMASK | _DBIAS << _DOFF;
                *pex = xchar - _DBIAS;
                return (FINITE);
                }
        else
                {       /* zero */
                *pex = 0;
                return (0);
                }
        }
```

This function must also deal with another added complexity of IEEE 754 arithmetic. A value with very small magnitude can be *denormalized.* That provides a form of *gradual underflow* that has desirable properties in a few cases. It also mucks up some functions that would be otherwise fairly straightforward.

`_Dunscale` calls the function `_Dnorm` to deal with denormalized values. The latter function produces a normalized fraction, if possible. It also returns a corrected power-of-two exponent for a finite denormalized operand:

```
#include "xmath.h"

/* normalize double fraction */

short _Dnorm(unsigned short *ps)
        {
        short xchar;
        unsigned short sign = ps[_D0] & _DSIGN;
```

```
        xchar = 0;
        if ((ps[_D0] &= _DFRAC) != 0 || ps[_D1]
                || ps[_D2] || ps[_D3])
                {       /* nonzero, scale */
                for (; ps[_D0] == 0; xchar -= 16)
                        {       /* shift left by 16 */
                        ps[_D0] = ps[_D1], ps[_D1] = ps[_D2];
                        ps[_D2] = ps[_D3], ps[_D3] = 0;
                        }

                for (; ps[_D0] < 1<<_DOFF; --xchar)
                        {       /* shift left by 1 */
                        ps[_D0] = ps[_D0] << 1 | ps[_D1] >> 15;
                        ps[_D1] = ps[_D1] << 1 | ps[_D2] >> 15;
                        ps[_D2] = ps[_D2] << 1 | ps[_D3] >> 15;
                        ps[_D3] <<= 1;
                        }

                for (; 1<<_DOFF+1 <= ps[_D0]; ++xchar)
                        {       /* shift right by 1 */
                        ps[_D3] = ps[_D3] >> 1 | ps[_D2] << 15;
                        ps[_D2] = ps[_D2] >> 1 | ps[_D1] << 15;
                        ps[_D1] = ps[_D1] >> 1 | ps[_D0] << 15;
                        ps[_D0] >>= 1;
                        }
                ps[_D0] &= _DFRAC;
                }
        ps[_D0] |= sign;
        return (xchar);
        }
```

The analog of `ldexp` is even messier. `_Dscale` must test for all the usual exception codes in its argument `*px`. It must also generate infinities and denormalized values. The code that follows is safe against intermediate integer overflow so long as `short` has a smaller representation than `long`:

```
#include "xmath.h"

/* scale *px by 2^xexp with checking */

short _Dscale(double *px, short xexp)
        {
        long lexp;
        unsigned short *ps = (unsigned short *)px;
        short xchar = (ps[_D0] & _DMASK) >> _DOFF;
```

```
if (xchar == _DMAX)     /* NaN or INF */
        return (ps[_D0] & _DFRAC || ps[_D1]
                || ps[_D2] || ps[_D3] ? NAN : INF);
else if (0 < xchar)
        ;       /* finite */
else if ((xchar = _Dnorm(ps)) == 0)
        return (0);     /* zero */

lexp = (long)xexp + xchar;
if (_DMAX <= lexp)
    {   /* overflow, return +/-INF */
    *px = ps[_D0] & _DSIGN ? -_Inf._D : _Inf._D;
    return (INF);
    }
else if (0 < lexp)
    {   /* finite result, repack */
    ps[_D0] = ps[_D0] & ~_DMASK | (short)lexp << _DOFF;
    return (FINITE);
    }

else
    {   /* denormalized, scale */
    unsigned short sign = ps[_D0] & _DSIGN;

    ps[_D0] = 1 << _DOFF | ps[_D0] & _DFRAC;
    if (lexp < -(48+_DOFF+1))
            xexp = -1;  /* certain underflow */
    else
            {   /* might not underflow */
            for (xexp = lexp; xexp <= -16; xexp += 16)
                {       /* scale by words */
                ps[_D3] = ps[_D2], ps[_D2] = ps[_D1];
                ps[_D1] = ps[_D0], ps[_D0] = 0;
                }

            if ((xexp = -xexp) != 0)
                {       /* scale by bits */
                ps[_D3] = ps[_D3] >> xexp
                        | ps[_D2] << 16 - xexp;
                ps[_D2] = ps[_D2] >> xexp
                        | ps[_D1] << 16 - xexp;
                ps[_D1] = ps[_D1] >> xexp
                        | ps[_D0] << 16 - xexp;
                ps[_D0] >>= xexp;
                }
            }
```

```
                if (0 <= xexp && (ps[_D0] || ps[_D1]
                        || ps[_D2] || ps[_D3]))
                    {   /* denormalized */
                    ps[_D0] |= sign;
                    return (FINITE);
                    }
                else
                    {   /* underflow, return +/-0 */
                    ps[_D0] = sign, ps[_D1] = 0;
                    ps[_D2] = 0, ps[_D3] = 0;
                    return (0);
                    }
                }
        }
```

The final primitive is the analog of `modf`. I found it useful to make `_Dint` somewhat more general. Some math functions have occasion to preserve one or more fraction bits while dropping the rest. Thus, negative values of the argument `xexp` specify how many bits to keep to the right of the binary point. Less-significant fraction bits are cleared. Note that this function returns the proper category code for the *fraction* that is discarded, *not* for the integer that is retained. As dizzying as that may appear, it proves to be the best behavior for the function:

```
#include "xmath.h"

/* test and drop (scaled) fraction bits */

short _Dint(double *px, short xexp)
        {
        unsigned short *ps = (unsigned short *)px;
        unsigned short frac = ps[_D0] & _DFRAC
                || ps[_D1] || ps[_D2] || ps[_D3];
        short xchar = (ps[_D0] & _DMASK) >> _DOFF;

        if (xchar == 0 && !frac)
                return (0);     /* zero */
        else if (xchar != _DMAX)
                ;       /* finite */
        else if (!frac)
                return (INF);
        else
                {       /* NaN */
                errno = EDOM;
                return (NAN);
                }
```

```
xchar = (_DBIAS+48+_DOFF+1) - xchar - xexp;
if (xchar <= 0)
        return (0);     /* no frac bits to drop */
else if ((48+_DOFF) < xchar)
        {       /* all frac bits */
        ps[_D0] = 0, ps[_D1] = 0;
        ps[_D2] = 0, ps[_D3] = 0;
        return (FINITE);
        }

else
        {       /* strip out frac bits */
        static const unsigned short mask[] = {
                0x0000, 0x0001, 0x0003, 0x0007,
                0x000f, 0x001f, 0x003f, 0x007f,
                0x00ff, 0x01ff, 0x03ff, 0x07ff,
                0x0fff, 0x1fff, 0x3fff, 0x7fff};
        static const size_t sub[] = {
                _D3, _D2, _D1, _D0};

        frac = mask[xchar & 0xf];
        xchar >>= 4;
        frac &= ps[sub[xchar]];
        ps[sub[xchar]] ^= frac;

        switch (xchar)
                {       /* cascade through! */
        case 3:
                frac |= ps[_D1], ps[_D1] = 0;
        case 2:
                frac |= ps[_D2], ps[_D2] = 0;
        case 1:
                frac |= ps[_D3], ps[_D3] = 0;
                }
        return (frac ? FINITE : 0);
        }
}
```

Conclusion

I have used these primitives to write all the math functions presented in *The Standard C Library* (Prentice Hall, 1992). That should serve as an existence proof. The functions are demonstrably portable and arguably efficient and easy to read. I have found the primitives to be useful in knocking together other math functions as well.

More interesting to me are the implications for extending the C language

proper. `_Dtest` seems to capture much of what you often need to know about an arbitrary floating-point value. I find it much more useful than, say, a comparison operator that tolerates NaNs. Perhaps what we need to institutionalize is a standard mapping from floating-point values to category codes.

I hesitate to suggest that C needs another operator. I hesitate almost as much to suggest that we further overload existing operators. With those caveats, I tentatively make two observations:

1. The logical operators `&`, `|`, `^`, and `~` are undefined for floating-point operands. Replacing a floating-point value with a bit mask that categorizes it *could* permit expressions that are both readable and optimizable.

2. The shift operators `>>` and `<<` are undefined for a floating-point left operand. Both have fairly obvious meanings in terms of the functions `ldexp` (or `_Dscale`).

Both observations are probably rash, but they might stimulate some creative thought.

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

∞

11. Reusable Incremental Scanning

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

Abstract

The design of a scanner for standard C that is independently reusable and incremental is presented. It is is claimed that similar designs are possible for other compiler components and other programming languages. This scanner uses reusable components for source line management and string management. The lexing algorithm itself is not the main issue—it can be conventional. Reusability is achieved by separation of function, simple and consistent interfaces, and provision of component-level testing. Incremental capability is achieved by saving lists of tokens corresponding to blocks of source lines and, when the source is unchanged, reusing the tokens instead of rescanning.

Introduction

This paper presents the design of a scanner that is reusable, and can be line-at-a-time incremental.

There are two reasons to do incremental scanning. First, assuming that changes are much smaller than the whole size of the file, incremental scanning is faster—the time to update the token representation of the source is proportional to the size of the change made to the source. Second, incremental scanning enables the consumer of tokens downstream to perform its processing incrementally and therefore faster as well.

The advent of very fast central processesors provides processing power that could make the performance gains of incremental computation irrelevant, although experience indicates that there is no limit on how many processor cycles we can use. Another perhaps more durable efficiency issue is minimization of file I/O. In a large memory environment, this scanner needs only infrequent access to files.

The reasons to promote reuse in general are obvious. It is not so obvious how to achieve it. Reuse depends on a delicate tradeoff between generality and specificity. Progress in reuse comes when one discovers a generalization (widely applicable component) which does not significantly degrade any one

use (efficiently applicable component). Components can be made reusable by paying attention to separation of function and to interface design. Reuse also depends on reducing the engineering effort to understand and incorporate an already implemented component into a larger whole. For this the designer can provide consistency between similar interfaces in both form and interpretation as well as amenities such as documentation and a demonstration.

This paper provides the signature and semantics for the interface of a reusable scanner. The demonstration of implementation adequacy assumes the existence of a driver program (or standalone test jig, in our parlance) which exercises all of the scanner functions. It turns out that there is synergy between making a component reusable and making the same component incremental. Both attributes require clear and narrow interfaces. Our approach is consistent in that it not only provides a reusable scanner interface but also assumes the availability of reusable components for handling strings and input text.

This scanner is usable in a compiler front end and also available for other uses such as pretty printers. The treatment here is for scanning standard C but the solution is generally applicable to popular programming languages with a few language-specific adjustments [2]. Ironically, K&R C is one exception to the general rule—with K&R, scanning is not easily separated from preprocessing.

The definition of a programming language usually includes a lexical grammar which specifies the transformation from character-stream source text into a token-stream needed for the next stage of processing. Even though there are tools for automating scanner construction based on a lexical grammar, most scanners are written by hand. It is easy to do and compiler writers perceive that generated scanners are less efficient [7, 8]. It is also true that most uses require some *ad hoc* tweaks to the scanner. Scanner generators may supply workarounds for the known needs to tweak, but cannot guarantee workarounds for future needs. The developer is therefore uncertain about whether a scanner generator will be up to the next task. For the work reported here, the next tweak was incremental processing. No generator provided the capability. This line of reasoning led us to use a lexical grammar for documentation while implementing the corresponding scanner by hand [2, §2.1.1.2, §A.1].

The paper is organized in the following sections:

- The scanner interfaces.
- The internals of the scanner.
- Example: a non-incremental consumer.
- Example: an incremental consumer.
- Performance.

The literature on incremental and parallel language processing overlaps because both kinds of processing use similar kinds of incremental structures. The

reader is referred to two comprehensive surveys of the literature for more information [3, 6]. One case study combines scanning and preprocessing, enabling either K&R-style or Standard C-style scanning to be chosen by the user [4]. Another prototype defines an editor interface that gives single character granularity for a simple language [1]. This paper presumes the reader has access to a description of Standard C [2, 5].

Much of the detail in the paper is taken from our implementation of an incremental scanner for C. Where we felt the presentation would be improved, or in retrospect we could have done a better job on the scanner, we have presented the view of how it should be, rather than how it is. This paper does not say much about parts of the scanner that are conventional since that information is available in any standard text.

Scanner Interfaces

The scanner interfaces are presented as a set of reusable abstractions. The form of, relation between, and naming conventions for these abstractions are part of the content of this paper. Each abstraction is organized about a principal data structure. The name of that data structure also names the abstraction and prefixes functions defined in the abstraction. These conventions are used throughout without further mention. (See `Token`, below, for example.)

We also use a convention which we call `struct`-wrapped handles to enhance the reusability of our abstractions. The use of `struct`-wrapped handles allows an implementation of an abstraction some freedom in laying out the handles without affecting clients. The actual encoding of any handle (for tokens or other things mentioned below) is private to the implementing component. In fact the representation of the handle is often a `struct`-wrapped index into an array managed inside the abstraction. This scheme allows position-independent pointers to various kinds of objects, hides the details of the implementation of our memory manager from its clients, enables C compilers to strongly type-check handles, and avoids inadvertent unsafe modifications of these handle values. The use of wrappers should not degrade performance.

Token Abstraction

The token abstraction is implemented (in C) in module `token.c` with public interface `token.h`.[1] The abstraction supplies primitive functions for stepping forward and backward in the source input stream one token at a time, and for examining various attributes of a given token. The user of this abstraction will want to jacket these primitives with a use-specific function, as is demonstrated in the examples later.

[1] In actual use, component source file names are prefixed with the processor name, as in `cc_token.c`.

The principal data type produced by the scanner is `Token`. All uses of type `Token` except for declarations, assignments, and parameter passing are private to the scanner. The values of type `Token` are referred to as token handles.

Used in a C translator, the token output of this abstraction is input to the preprocessor.

```
typedef struct {private layout} Token

                          – iterator functions
t = TokenFirst()          – start at head of current file
t = TokenNext(t)          – step to next token
t = TokenPrev(t)          – step back to previous token

                          – attribute functions
c = TokenClass(t)         – small integer classifying token t
s = TokenString(t)        – textual form of token t
s = TokenStringRaw(t)     – as above, preserving backslash and trigraphs
s = TokenWhite(t)         – white space preceding token t
s = TokenWhiteRaw(t)      – as above, preserving backslash and trigraphs
h = TokenLoc(t)           – source file location of token t
```

There is a significant implementation cost implied by the full generality of this interface. For us the cost has been acceptable. The cost can be reduced somewhat for simple uses. The iterator functions (above) imply the existence of forward and backward pointers. If one limits the arguments to the iterator functions to a window of the previous n unique argument values for some small n, the pointers do not have to be permanently recorded, therefore permitting the implementation to save the corresponding storage. For Standard C, it is sufficient to enable `TokenPrev(t)` only during the scanning of `#include` directives. The attribute functions, on the other hand, must always be callable with any valid token handle.

Function `TokenFirst()` returns the first token in the input stream. Given any token, its successor and predecessor are returned by `TokenNext()` and `TokenPrev()` respectively. There is a special token class represented by end-of-file (`eofTOKEN`). This token is returned by `TokenFirst()` if the input stream is empty, and by `TokenNext(t)` if it is called with the last token in the input stream. There is another token class for not-a-token (`ntTOKEN`). This token is returned when `TokenNext()` is called with the end-of-file token and by `TokenPrev(TokenFirst())`.

The small integer value representing the token class is implemented with an `enum` in the interface. The enum-valued function `TokenClass(t)` is used a lot and must therefore be especially efficient. There are about 40 distinct values corresponding to the preprocessor input symbols in the C grammar [2, §A.1]. For example there is a class for operator `++`, one class for all identifiers, one class for all preprocessing numbers, and so on. In addition, there are exactly

a dozen identifiers that are keywords immediately following `#`, but not otherwise reserved in the preprocessor. Finally, there is `defined` which is reserved following `#if` and `#elif`, and *new-line* which is treated as a token in a preprocessing directive and otherwise as white space. There are about 80 distinct post-preprocessing values including reserved words [2, §A.2].

One solution to all this detail is for the scanner to give a unique class to every different symbol including key and reserved identifiers, and then provide mappings to the codes appropriate to the situation. For example, `if` has a unique class. Its various mappings and the mapping functions are shown below.

Preprocessing keyword context
(immediately following a `#` *that starts a line)*
```
if (TokenClass(t) == ifTOKEN)...
```

Preprocessor expression context (following `#if`*,* `#elif`*)*
```
if (InPpExpr(TokenClass(t)) == identifierTOKEN)...
```

All other preprocessing contexts
```
if (InPpBody(TokenClass(t)) == identifierTOKEN)...
```

During parsing
```
if (InParse(TokenClass(t)) == ifTOKEN)...
```

Only the mapping `InPpExpr()` exposes the special preprocessor arithmetic operator `defined` to be the token class `definedTOKEN` during preprocessor expression context.

Another solution is to use different token-class values after preprocessing, constructing the tokens anew at the end of phase 4 [2, §2.1].

Functions `TokenString(t)` and `TokenWhite(t)` each return values of type `String`, which is presented next. The value of `TokenString(t)` is text of the token itself. The value of `TokenWhite(t)` is the text of the white space preceding the token[2]. The preceding white space is needed during macro definition and expansion in a preprocessor for Standard C. Any backslash-newline pairs and trigraphs that occurred in the source file do not appear in the values of these functions. The two other versions (suffixed with `Raw`) provide the exact textual form. Printing the `Raw` form of the strings for all the tokens in a file, in order, reproduces the source text.

Function `TokenLoc(t)` returns a value of type `Locator` which is a further abstraction providing access to token starting and ending column positions, file name and line, and other kinds of information associating token to source-text position. Function `TokenLoc(t)` is used in compilers principally to place diagnostics relative to the source text. What `TokenLoc(t)` locates is precisely `TokenStringRaw(t)` since that is what is in the source file. Further details about locators are not essential to this presentation.

[2]The term *white space* has a technical meaning [2, §3.1]. It is not affected by the interpretation of the token stream as preprocessing directives.

Token Increment Abstraction

The token increment abstraction is implemented (in C) in module `tokinc.c` with public interface `tokinc.h`. It supplies an interface similar to the token abstraction except that it operates on token increments.

A token increment is represented by type `TokInc`. Values of type `TokInc` are called tokinc handles or just tokincs. Objects of type `TokInc` can be declared, assigned, and passed as parameters.

A token increment is an abstraction for managing tokens. The scanner maintains an image of the source file as a list of logical source lines (also known as the tokinc sequence). Usually a real source line corresponds one-to-one with a logical source line in C. However, multiple real source lines turn into a single logical source line when real lines are glued together by a terminating backslash (\) character or by multi-line comments. A tokinc also records the creation-history of the tokens that are associated with it, using a time stamp.

Token increments enable incremental scanning. The granularity of the incremental update is the tokinc. When source changes, the scanner only needs to rebuild the tokincs that correspond to the changed lines in the source.

```
typedef struct {private layout} TokInc

                                – iterator functions
w = TokIncFirst()               – start at head of current file
w = TokIncNext(w)               – step to next increment
w = TokIncPrev(w)               – step back to previous increment
                                – list manipulation functions
TokIncLineDelete(j)             – line j has been deleted
TokIncLineInsert(j, a)          – line j inserted ahead of line a
                                – attribute functions
t = TokInc2Token(w)             – first token in tokinc w
w = TokIncOfToken(t)            – token increment associated with token t
k = TokIncMade(w)               – time of construction of tokinc w
```

The tokinc iterators behave in the same manner as the token iterators. When the start or end of the tokinc sequence is exceeded, a special handle value of `ntTOKINC` is returned by the iterators.

Functions `TokIncLineDelete()` and `TokIncLineInsert()` are the list manipulation functions to modify the tokinc sequence as the line structure of the source changes (e.g. old lines deleted, new lines inserted). Tokincs are hashed by line handles (see the `Line` abstraction below) to facilitate fast deletion and insertion operations.

The tokinc abstraction provides two attribute functions. `TokInc2Token()` returns the leading token of its associated logical source line. Thus the following invariant is always maintained:

```
TokenFirst() == TokInc2Token(TokIncFirst())
```

The function `TokIncOfToken()` is used in the incremental mechanisms. Given a token, it will return the containing token increment. More details on token increments are given in the next section.

Time stamps, represented by a type `TimeStamp`, are discussed in general in a later section. Function `TokIncMade()` returns the time stamp of the specified token increment. In this case the time stamp indicates the age of the tokens in the tokinc as well. Individual time stamps for each token could have been used but since a new list of tokens is created each time a change to a logical source line occurs, a single time stamp assigned to a tokinc is sufficient.

Line Abstraction

The line abstraction is implemented (in C) in module `line.c` with public interface `line.h`. It supplies an interface similar to the previously described abstractions except that it operates on source lines.

A line is represented by a type `Line`. Values of type `Line` are called line handles. Items of type `Line` can be declared, assigned, and passed as parameters.

The line abstraction provides access to the text of a source file. It is the main input abstraction for the scanner. The actual origin of the text is hidden behind this interface, i.e. it could be from disk or from an editor. There must be a mechanism for selecting the file; the specifics are not important for this paper.

The following public interface to source text is sufficient for scanning.

```
typedef struct {private layout} Line

                                  – iterator functions
j = LineFirst()                   – start at head of current file
j = LineNext(j)                   – step to next line
j = LinePrev(j)                   – step back to previous line

                                  – attribute functions
c = Line2Cstring(j)               – text of j, null-terminated
h = LineLoc(j)                    – source file location of line j
k = LineMade(j)                   – time of construction of line j
LineCleanSet(j)                   – set line to status clean
b = LineCleanGet(j)               – TRUE means line is clean
```

The line iterators behave in the same manner as all the other iterators. When the start or end of the line sequence is exceeded, a special handle value `ntLINE` is returned by the iterators.

Function `Line2Cstring(j)` provides a read-only null-terminated C string for line `j`. Function `LineLoc()` returns a value of type `Locator` (discussed earlier under tokens).

The function `LineMade(j)` is useful for incremental scanning. It returns a value of type `TimeStamp`. Alternatively, one may use the pair `LineCleanSet(j)` and `LineCleanGet(j)` if the line abstraction supplies a clean/dirty bit instead of a time stamp.

The typical use of source information is to start at the beginning and step through the whole input, one line at a time. On occasion it is necessary to look ahead in the input before processing the current line. For compilers, it is acceptable to restrict all the functions to some small window containing just the last few accessed lines. This is a reasonable compromise with the full generality of the functions.

In Fortran, for example, the lookahead line must be examined for a continuation mark before the current line can be scanned. In C, the scanner cannot process `#include` directives in one pass—there are three possibilities, the third of which applies only when the other two fail. This backup situation is confined to a logical source line, but that may cross many physical source lines because of comments and backslash suppressing line breaks. Because our incremental scanner works on a granularity of physical source lines, resetting to try the third possibility can therefore require stepping back many lines[3].

String Abstraction

The string abstraction is implemented (in C) in module `string.c` with public interface `string.h`.

The type `String` is an abstraction of C `char*` data. It is closely related to the standard library functions [2, §A4.11]. Variables of type `String` can be declared, assigned, and passed as parameters. Values of type `String` are called string handles.

The string abstraction is used by the scanner to manage the storage of all text. For example, the text-attribute functions for the token abstraction return values of type `String`.

The string abstraction has a collection of functions and holds a collection of strings. Text `c` is registered in the abstraction via one of the function calls `Cstring2String(c)` or `Chars2String(c,n)`. The first form expects a null-terminated C string. The second form expects at least `n` non-null characters. The first function is just an abbreviated form of the second. That is Cstring2String(c) is equivalent to Chars2String(c, strlen(c)) so long as `c` is null-terminated.

The value `String2Cstring(TokenString(t))` is a null-terminated `char *` text of the token `t`. Similarly, `String2Cstring(TokenWhite(t))` provides a (perhaps empty) null-terminated sequence of comments, blanks, tabs, and *newlines* immediately preceding the token.

The principal data type is `String`.

[3]Because of the 509 character limit for logical source line, there is a limit to the number of lines of backup, but it is too large to be interesting.

```
typedef struct {private layout} String

s = Cstring2String(c)      – tabulate null-terminated string
s = Chars2String(c, n)     – tabulate n characters
c = String2Cstring(s)      – null-terminated char* representation
n = StringEql(s1, s2)      – like strcmp() == 0
n = StringLen(s)           – like strlen()
```

The characteristics of the package are that `StringEql(s1, s2)` is fast and that all variable-length storage is automatically managed, thus relieving the compiler writer (or other users) of that task. Outside the string package, variables can be declared and values can be assigned and passed around with low overhead. Values of type `String` are only created inside the string package. Because source program input is repetitive (use an identifier once, use it a dozen times...) and only one unique copy of each string need be tabulated, the size of the internal tables grows more slowly than the total size of source input.

The actual package which motivates this writeup has other functions, for context switching, catenation, hashing, testing, and so on. Further details are omitted here.

Scanner Internals

This section presents the technology behind the scanner interfaces. The presentation includes a discussion of time stamps, how lexical analysis is implemented, and a description of how coherency is maintained between the tokinc sequence and the line structure of the source.

Figure 1 illustrates the relationship between the main abstractions described in the previous section. The incremental nature of the scanner is revealed as well.

The input to the scanner is the sequence of lines. The scanner builds an image of the input as a sequence of logical lines (type `TokInc`). Parallel to the tokinc sequence is the sequence of tokens. Each tokinc provides direct access to the token that starts its logical line.

The consumer of the tokens can traverse the length of the token stream to obtain the token-representation of the source. The scanner can incrementally update portions of the token sequence by rebuilding only the tokincs and tokens associated with the modified lines.

The token increments and associated tokens can be saved, even written to a file. The token-producing interface could be driven directly from the filed information without reference to the source. This allows scanning to be separated in time from the use of the tokens, effectively turning the token increments into an intermediate language. An important consequence shows up later in `#include` directive processing, where rescanning can be avoided by reusing the token increments left over from a previous inclusion of the same file.

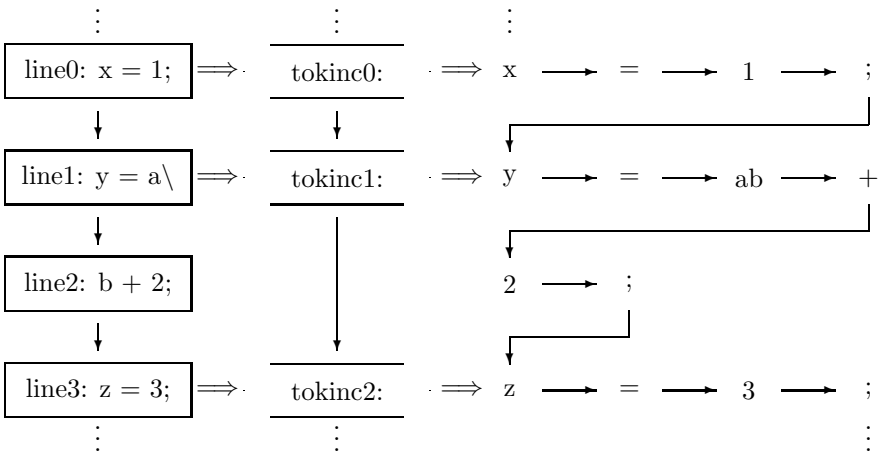

Figure 1: Relationship between Lines, Tokincs and Tokens

Clean, Dirty, Time Stamps

In incremental computation, the dependencies between the components form the basis for reprocessing. If something older depends on data that is newer, the older must be replaced with a recomputed newer value. It must therefore be possible to compare times of creation for all of the atomic increments (e.g., types `TokInc`, `Token`, and `Line`). Dirty bits or time stamps provide the essential information—they are associated with increments of various kinds.

A dirty bit for an increment is set to TRUE by the producer whenever the value is changed. The dirty bit is set to FALSE by the consumer when the value of the increment has been read and assimilated. Write-access to the dirty bit must be given to both the producer and consumer.[4]

A time stamp may be associated with increments. The order of time stamps is the same as the order of time-of-creation for the increments. Typically a time stamp is an `int` value and the time stamp generator just keeps adding one for each new value. With a 32-bit `int`, one need not fear of running out of values. A stale increment will have an older (smaller) time stamp than something it depends upon. The clean/dirty information is derived by comparing time stamps. With this solution only the producer needs write-access to the increment. Furthermore, several consumers can share an increment from a single producer. In C, for example, lines in header files are shared by all the modules that include them.

It is necessary that time stamps be correlated across all of the various incremental constructs. For example, the time stamps of lines must be comparable

[4]This simple device breaks down when there is more than one consumer—each consumer must have a way of separately setting FALSE.

with the time stamps of tokens. Whenever a token is stale with respect to the line it comes from, the old token must be discarded and a new token created in its place with time stamp value at least as young as the new line from which it is taken.

If incremental constructs are cascaded (for example, across editing, scanning, preprocessing, parsing, etc.), it may be convenient for an increment to have two kinds of time stamps, one comparable to its input and one comparable to its output. The two kinds of time stamps need not be comparable with each other. This permits each abstraction to own its own time-stamp generator without any need to coordinate with those in other abstractions.

Lexical Analysis

Scanners have three principal responsibilities: lexically separating the input, associating the pieces with the token classes needed by the next stage of processing, and diagnosing a few kinds of input errors. Scanners may also be responsible for detecting and responding to the context in which the tokens are found. For example, in C the text of a string for a header name follows different rules than the text of string literals. Other languages have more complex context dependencies.

Having provided interfaces for the input and output of the scanner, we now show how to isolate lexemes and make tokens. There are many ways to do it. Most readers will have already done it more than one way. We divide the tokens into groups detectable by their first character and switch inside the scanner to cases, one for each group. For C, we use a dozen unique switch values. They are represented by an enumeration local to the scanner:

```
typedef enum {
    STARTNOTHING,          /* most control characters  */
    STARTSHARP,            /* # and ##                 */
    STARTSINGLE,           /* , ( @ ...                */
    STARTMULTIPLE,         /* + = * ...                */
    STARTID,               /* ids, wide char & string  */
    STARTPPNUM,            /* 0 1 ... dot ellipsis     */
    STARTCHAR,             /* '                        */
    STARTSTRING,           /* " (string or header name)*/
    STARTCOMMENT,          /* /* / /=                   */
    STARTHEADER,           /* <stdio.h> < <= << <<=    */
    STARTWHITE,            /* blank tab ...            */
    STARTENDOFLINE         /* null (from Line2Cstring) */
} LexemeGroup;
```

In each switch case the details of constructing the token are filled in. Within the case for `STARTMULTIPLE`, for example, there is a nested switch on the leading character, and within these cases there are switches on the next character, and

so on. The following code fragment illustrates the structure for lexing the four C symbols `!`, `=`, `!=`, and `==`. The function `CharGet()` is a local function extracting characters from the current line of source input.

```
...
case STARTMULTIPLE:
    switch (c) {            /* c is leading character   */
    case '!':
        c = CharGet();      /* discard `!'              */
        switch (c) {
        case '=':           /* `!='                     */
            c = CharGet();  /* discard the `='           */
            class = NEQ;    /* token class for `!='     */
            break;
        default:            /* `!'                      */
            class = NOT;    /* token class for `!'      */
            break;
        }
        break;
    case '=':
        c = CharGet();      /* discard `='              */
        switch (c) {
        case '=':           /* `=='                     */
            c = CharGet();  /* discard the second `='   */
            class = EQL;    /* token class for `=='     */
            break;
        default:            /* `='                      */
            class = ASG;    /* token class for `='      */
            break;
        }
        break;
    case ...                /* other multiples...       */
    }
    break;
```

The case on `STARTID` handles wide string and character literals as well as ordinary identifiers and reserved words. The case on `STARTPPNUM` handles the dot operator and ellipsis as well as *preprocessing-number*.

The collecting of the type `String` values for the tokens as well as the processing of backslash at the end of a line and trigraphs is hidden inside function `CharGet()`. The suppressed new-line characters never appear outside `CharGet()`. Trigraphs are translated into their equivalents. In the infrequent case that the `Raw` version of the text differs from the normal value, special construction is required. This is also implemented inside `CharGet()`. The line abstraction produces null-terminated C strings for lines, so the new-line character has to be rematerialized for some white space values.

Once the token class has been determined, and all the other information recorded (such as locating line and column), the token is assembled. There is no implication that the data is all packed into a token—the data can be held in internal tables. The `Token` abstraction provides a scanner-private routine for assembling a token given its class, text, raw-text, whitespace, raw-whitespace, and locator.

This design leads to a module that is big but easy to build and maintain. The design also gives convenient hooks for the language-specific context-dependent situations. Note also that the lexical analyzer is isolated from the rest of the scanner module since its only external interfaces are `CharGet()` (with calls to `Line` routines) for input and the `Token` construction routine for output.

Token Increment Construction

In a traditional scanner, the assembled token is returned and used immediately. As a consequence, the scanner holds a partially processed source line between calls.

An incremental scanner will save each token by appending it to a local queue. This queue may be of arbitrary length, which allows the scanner to process text and produce tokens until a natural stopping point is reached (usually the end of a line). This queue-centered design simplifies (at little cost to efficiency) the between-call state-saving of the scanner since there are no partially processed lines of text hanging around. More significantly, the queue-centered design also enables incremental scanning. Each queue-full corresponds to the tokens generated by scanning a logical line (tokinc). The token iterators manipulate these queues to traverse the token stream.

By the nature of the constructing mechanism, each token increment is aligned with the line structure of the source input text so that if the source text is unchanged then so also is the token increment—a clean line leaves a clean token increment. If the source text is changed, then the scanner will create a new queue-full of tokens and a new tokinc, which is inserted in the appropriate location in the tokinc sequence to mirror the new line structure of the source.

The first time the source is scanned, source lines can be processed sequentially from front-to-back. Token increments are produced in the order the lines are scanned, and the resulting tokens are associated with tokincs. After a source file has been scanned, a correspondence between source lines and the token increments is established (as in Figure 1). For C the correspondence is many-one—more than one source line may be needed to make a complete token increment (because of backslash and multi-line comments).

During edit, source lines may remain unmodified, be inserted, or be deleted. All other kinds of edit activity can be expressed as a combination of these. After some edit activity, the previous correspondence between saved increments and source lines is broken—there will be token increments that no longer have a source (deleted source), and source lines that have no token increment (inserted

source). The difference can be expressed as a sequence of deletes and inserts. The sequence is not unique. Two sufficient sequences are the real-time sequence as recorded by the editor and the front-to-back sequence produced by a file differencing algorithm. Either sequence can be used to drive the functions:

```
TokIncLineDelete(deleted);
TokIncLineInsert(inserted, before);
```

When a line is deleted, the scanner must free the corresponding token increment, perhaps construct another to replace it, and reestablish the sequence between the remaining token increments. All of this requires that the scanner find the token increment corresponding to the deleted source line. An efficient solution is to provide a hash table taking the source line handle `deleted` into the corresponding tokinc handle. The hash table is maintained by the token increment abstraction. The inverse operation, finding the source lines from the token increment, requires that the token increment contain the handle of at least one of the source lines it represents.

If the deleted line is part of a multi-line increment, the neighboring lines must be included in the rescanning. For C, if the deleted line is the last line in a multi-line increment, rescanning may swallow the following token increment also. When all of this is done, the correspondence is reestablished.

Suppose, on the other hand, that a line has been inserted—it is new and therefore will be marked dirty. The hashing mechanism cannot look up a new line. The hash is used instead to find the increment containing the information from the source line now following the inserted source line. If the source line following the inserted source line has been newly inserted as well, then a temporarily disjoint tokinc sequence is created. This sequence grows until a source line that corresponds to a tokinc in the main sequence is provided, at which point the entire disjoint sequence is inserted before this tokinc in the main sequence. The previously stated considerations about rescanning and swallowing the following increment and reestablishing the correspondence apply. In addition, inserting a line may cause a multi-line increment to be broken into two separate increments.

The delete/insert notifications can be applied immediately, or queued for later application either all at once or in small batches. Likewise, an editor can delay making the notifications, sending them a batch at a time. This trick allows an editor to locally optimize the queue contents, canceling insert/delete pairs. What is required is that the actions are applied, in order, prior to using the tokens.

If the source lines have no independently assigned time stamps, time stamps can be "invented" by the scanner as the insert notifications arrive. The correspondence between line handles and time stamps is then maintained in a separate data structure in the scanner rather than by the editor.

If the editor provides time stamps or clean/dirty bits for lines, the incremental problem reduces to creating a valid insert/delete sequence from the old

token increments, the new source text, and the change information. The source lines in the editor may be examined in file order. The change information for each source line handle is examined, and perhaps compared with the corresponding record of it in the token increment. If the line is dirty, it must be rescanned—effectively treated as a delete/insert pair.

A disagreement between the new sequence of line handles and the old record of line handles in the token increments means either a line was deleted, or was inserted, or both. Since there is a fast way to go from the source line handle to the corresponding token increment, the correspondence can be checked in that direction. If the offending source line handle is found in a token increment, one may deduce that all lines corresponding to the token increment sequence from the point of disagreement to the found line have been deleted. This results in a sequence of deletes until the handle agreement is reestablished and the correspondence is lengthened one step—therefore insuring that progress is made.

If the source line handle is not found in any tokinc, this source line has been inserted. The line is scanned and its token increment inserted into the token increment sequence, which again insures that progress is made.

The use of change information gives a delete/insert sequence (nearly) identical to the sequence one would get from file differencing.

Example 1: Non-incremental Consumer

The use of tokens can be as varied as the number of users. So long as the reusable primitives supply enough information, any special interface can be built by the user in terms of the primitives.

A sample non-incremental consumer is described below. This consumer uses the scanner interface in a non-incremental fashion, i.e., tokens from the scanner are processed from start to finish without making any use of its history.

Function `Init()` must be called once to initialize access to this module. It sets a private variable called `NextToken` to the first token in the source input stream. Function `Scan()` may then be called to process tokens in input stream order. Function `LookAhead()` may also be called to examine one token ahead in the input stream without affecting the input stream order used by `Scan()`.

The private variable `CurrToken` is used to track the most recent token returned by `Scan()`. Its use will be described in the next section.

```
#include "token.h"

static Token NextToken;
static Token CurrToken;

void
Init(void) {
    NextToken = TokenFirst();
}

Token
Scan(void) {
    CurrToken = NextToken;
    NextToken = TokenNext(NextToken);
    return CurrToken;
}

Token
LookAhead(void) {
    return NextToken;
}
```

A non-incremental consumer may use these routines to display all the tokens from the source input. Function `Scan()` is used to get each token. The consumer terminates processing when `eofTOKEN` is returned by `Scan()`. Function `LookAhead()` may be used to control formatting of the current token based on what follows it. This example is contrived but it does illustrate how the scanner interface may be reused in a simple situation.

One final point, even though the consumer is non-incremental, the scanner is managing its token-representation of the source incrementally. Thus, the consumer gains some efficiency in performance since the scanner does not have to work as hard.

Example 2: Incremental Consumer

The previous example is extended in this section to describe an incremental consumer. This consumer uses the creation-history of the tokens from the source input stream. That is, if a section of the input stream has not changed since the last time it was processed, then the consumer may skip over it.

The interface and associated semantics used by the non-incremental consumer may be reused by the incremental consumer. That is, `Scan()` and `LookAhead()` are used as previously described to process each token in the input stream the very first time. They are also used to process sections of the input stream that have changed.

Two new routines are introduced. Boolean function `CleanInput()` returns `TRUE` to indicate that the consumer may skip over the current section of the

input stream (since it has not changed) and reuse the corresponding output rather than producing it over again. A return value of `FALSE` indicates a change has occurred and that new output will needed. Function `SkipInput()` is the incremental analogue of `Scan()`; it returns the first token that immediately follows the skippable section of the input stream. It should be called only if `CleanInput()` returns TRUE.

The unit of skipping is the token increment. Recall that a token increment describes a logical source line and therefore a contiguous sequence of lines within the input stream. More than one token increment may be needed to form an increment for the consumer. The routines `CleanInput()` and `SkipInput()` allow the consumer to process and skip-process over the input stream in multiples of token increments.

The routine `TokIncOfToken(t)` may be used to track the sequence of token increments associated with the tokens returned by `Scan()` and `SkipInput()`.

```
#include "tokinc.h"

... reuse Non-incremental Consumer code ...

int
CleanInput(int skipSize, TimeStamp maxTS, TokInc ess) {
    TokInc ti;

    ti = TokIncOfToken(CurrToken); /* get container      */
    for (;;) {
        if (ti == ntTOKINC) break; /* off end of sequence*/
        if (TokIncMade(ti)>maxTS) break; /* out of date  */
        if (--skipSize == 0) break;/* end of subsequence */
        ti = TokIncNext(ti);       /* OK, move ahead     */
    }
    return (skipSize==0) && (ti==ess);   /* reached end? */
}
```

```
Token
SkipInput(TokInc ess) {
    TokInc ti;

    ti = TokIncNext(ess);          /* skip past this     */
    if (ti == ntTOKINC) {          /* off end, find eof  */
        CurrToken = TokInc2Token(ess);
        for(;;) {
            if (TokenClass(CurrToken) == eofTOKEN) break;
            CurrToken = TokenNext(CurrToken);
        }
    } else {                       /* normal tokinc      */
        CurrToken = TokInc2Token(ti); /* first token     */
        NextToken = TokenNext(CurrToken);/*prep lookahead*/
    }
    return CurrToken;              /* start next tokinc  */
}
```

When `CleanInput()` returns FALSE, the scanner is called to process the section of the input stream that has changed. When new output is produced, the consumer must record the following attributes of the section of the input stream that created it:

- `skipSize` specifies the length of the token increment subsequence processed to produce the output (1 or greater).
- `maxTS` specifies the maximum time stamp from the token increment subsequence.
- `sss` specifies the token increment that starts the subsequence. Function `CleanInput()` uses `CurrToken` to infer `sss`.
- `ess` specifies the token increment that terminates the subsequence.

When the same section of the input stream is revisited by the consumer the next time around, the attributes are passed to `CleanInput()` to determine if it can be skipped. The consumer must remember the token that starts the section so that the next time around it will know when to call `CleanInput()`. The code for `CleanInput()` detects the insertion, modification, and deletion of logical lines to the section of the input stream described by the attributes.

The consumer is responsible for skipping over and managing its output data structures.

Performance

The reusable interface without the incremental capability is competitive in speed with traditional scanners. Used incrementally, the scanner is much faster.

The cost for this efficiency is space to manage all of the data structures in memory. There are three main consumers of memory in the scanner: the `String`, `TokInc`, and `Token` abstractions. The amount of memory consumed is proportional to the size of the source being scanned.

We have developed a scanner for Standard C along the lines outlined in this paper. The scanner has not seen hard use and is therefore not a very good source of performance data. On the other hand, it is the only source of data we have. With that disclaimer in mind, we state that our incremental scanning is about 14 times faster than our first-time scanning. The cost is roughly the ratio of checking clean/dirty information (via a call to the editor) to the cost of checking plus rescanning dirty text. The memory use, taken from a single data point (scanning the implementation of the lexical analyzer itself) is given below.

source size	50,688 bytes
string table	6,545 bytes
token increments	45,942 bytes
token table	200,606 bytes

One can make tradeoffs in space versus speed by adjusting the granularity (size) of the increments. The finer the increment, the more memory is consumed. The coarser the increment, the less incremental the scanner. However, coarser increments can provide larger skips for the consumer of the scanner.

Conclusions

It is feasible to separate the scanner out of a compiler so that it can be used for other purposes. A scanner can be made incremental at approximately the granularity of source lines. The combination of these two capabilities provides a useful and flexible component for an organization dealing with processing of programming languages.

From the viewpoint of the traditional consumer of scanner output, and aside from efficiency, it does not matter whether the scanner saves and reuses tokens or recreates them from the source text. The scanner is therefore at liberty to implement any incremental mechanism consistent with its interface.

The speedup provided by incremental scanning is real, but not likely to be significant relative to the other costs such as preprocessing, parsing, symbol analysis, and so on. On the other hand, having done line-granularity incr-Standardscanning, we have also gathered the time-stamp information enabling incremental processing for following passes. We expect the speedup ratio for a whole system to be better than 14. Our preliminary C preprocessor and parser data support this expectation. Details will follow in a later paper.

References

[1] John F. Beetem and Anne F. Beetem, *Incremental Scanning and Parsing with Galaxy*, *IEEE Transactions on Software Engineering*, Vol 17. No 7 (July 1991).

[2] *American National Standard for Information Systems—Programming Language C*, X3.159-1989, American National Standards Institute, New York, N.Y.

[3] Neal M. Gafter, *Parallel Incremental Compilation*, PhD Thesis, Department of Computer Science, University of Rochester, June 1990 [available as Technical Report 349].

[4] John H. Parks, *Case Study: Building an ANSI CPP*, *Journal of C Language Translation*, Vol 2, No 3 (December 1990).

[5] P. J. Plauger and Jim Brodie, *Standard C*, Microsoft Press (1989).

[6] D. B. Skillicorn and D. T. Barnard, *Compiling in Parallel*, Department of Computing and Information Science Technical Report 90-289, Queen's University, Kingston, Ontario, Canada (October 1990).

[7] Duane Szafron and Randy Ng, *LexAGen: An Interactive Incremental Scanner Generator*, *Software—Practice and Experience*, Vol 20, No 5, pp 459–483 (May 1990).

[8] Bill Waite, *The Cost of Lexical Analysis*, *Software—Practice and Experience*, Vol 16, No 3, pp 473–488 (March 1986).

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.

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

∞

12. Electronic Survey Number 9

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.

Passing Narrow Arguments

Given the following program fragment, does your translator pass all narrow types as such, widen them all, or widen only some of them?

```
void x(char, short, float);

char c;
short s;
float f;

x(c, s, f);
```

- 11 – All kept narrow.
- 5 – All widened.
- 6 – `char` and `short` widened but not `float`.
- 1 – User selectable.
- Comments:

    1. Our machine is 64-bit word addressed, so all types are de facto widened so they can be stored without shifting and masking. However, we have the unfortunate habit of storing a `char` left-justified

in a 64-bit word instead of right-justified, so there is a gratuitous incompatibility between prototyped and nonprototyped functions.

2. Passes all types as narrow. *[Ed. While the argument itself is not widened, more bytes might be passed than are occupied by the argument.]* This can be more efficient. For example, you can push 32 bits of memory starting at the location of `c` for the first argument. You might get 3 bytes of trash but you don't care, since the receiving function doesn't care. Pushing 32 bits from memory (1 instruction) might be faster than loading a byte and pushing the entire register it's contained in (2 instructions).
3. The narrow form is passed with padding. Keeping the stack longword aligned generally yields better performance on processors such as the 68020, 030, and 040.
4. Only `float` is kept narrow. The real savings for `float` is the elimination of the two format conversions, not stack space.
5. This is a machine-specific option for the integer types. However, `float`s are never widened.

Returning Narrow-Typed Values

If a function returns a value having type `char`, `short`, *or* `float`, *do you pass it back in its narrow or wide form? (This is an interesting issue in that Standard C does NOT state the type of the result produced by the function-call operator.) What would your reaction be if X3J11 made an interpretation requiring the opposite behavior to what you have?*

- 14 – All kept narrow.
- 6 – All widened.
- 4 – `char` and `short` widened but not `float`.
- Comments:

    1. Because our return value lives in a 64-bit register, all returned types are widened that far. Contrary to our handling of `char`s as arguments, a returned `char` is right-justified in the word. An X3J11 mandate that types be left in their small form would probably be a no-op, since we could satisfy that requirement via the as-if rule.
    2. The return value is in the narrow form, except for some UNIX machines where to be cc-compatible we must widen it. X3J11 can't make a "do it this way, not that way" interpretation at this point; there's too much water under the bridge. cc definitely does it one way and lots of DOS compilers definitely do it the other way.

3. If an interpretation required a change, that would then be in conflict with the 88open binary standards. I suppose we could ignore such a change, and maybe implement a switch if the validation suite tested it. (Such an option would likely never be used otherwise!)
4. I don't think there is any support for such an interpretation in the standard, nor is it supported by prior implementations. (As far as I know there are lots that do it the way we do it.) As for my reaction, I don't suppose there would be a lot I could do about it, but I'd probably try.
5. It has little impact on us but we wouldn't like a change anyway.
6. I would be very surprised if ANSI required widening.
7. I believe keeping them narrow is the correct behavior. To distinguish `float` and `double` functions at declaration time and not at invocation time seems absurdly inconsistent.
8. For prototyped floating-point functions I return the declared type without conversion. Most programmers I know these days either do their floating point in one of `float` or `double` format but not both and when they do it using `float` they generally want *everything* to be in `float` without any invisible conversions done by the compiler. Besides, the original reasoning behind the `float` conversion was that `float` operations would be faster than `double` operations, not really something that it makes sense to build into a compiler spec when it destroys the symmetry of the language, and in any case is no longer a valid point.

   If X3J11 made an interpretation contrary to my own implementation, I might consider changing the `float` aspect.
9. Nothing is widened upon return. I would object to changing this, as it would make our compiler incompatible with other C/C++ compilers and Fortran compilers. Also, it would slow things down, and people using `float` want speed.
10. Mainly for compatibility, a narrow object is returned only when safe. In all cases, nothing narrower than `int` is returned. For `float` returns, only if the function is defined/declared with a prototype that includes at least one `float` argument will the return be single-precision. This turns out to be a reasonable compromise: function calls that benefit from a single-precision return almost always have at least one `float`-typed argument.
11. The type should be exactly that specified by the declaration of the function.
12. I wouldn't see a problem with an opposite ruling under the assumption that one could dummy up `sizeof(f())` but otherwise the as-if

rules would make it difficult to tell in what manner it was implemented. If the changed ruling required a change to the code generated (moving a widening operation from function call to function return), then I would have strong concerns about this causing an incompatible break in object file formats (particularly in case of `float` to `double` conversions).

13. I would be very unhappy if widening of `float`s were required. The integer types don't seem very important since it is cheap to widen them anyway.
14. We would oppose any interpretation by X3J11 to mandate widening.

Fixed versus Variable Argument List

If a function expecting a variable number of arguments is called without having a prototype containing an ellipses in scope, will the call still work? That is, do you take advantage of a different/more efficient call mechanism in the case of (assumed or actual) fixed length argument lists?

- 19 – Will still work.
- 3 – Will not work.
- Comments:

  1. With optimization switched on parameters are passed in registers. We demand prototypes for this feature.
  2. If the programmer specifies no special options to the compiler the call will still work, even if prototyped incorrectly. However, if the programmer specifies special optimization options to the compiler, such as registered-args, and does not properly prototype a var-args function, the program will not work.
  3. We emit a warning for the user, but still allow this to work.
  4. We are considering changing to a scheme where the presence of a prototype won't affect the way variable arguments are passed.
  5. No, we don't take advantage of any such mechanism. Even if we did, we would try to find a way to make it work for the case you described.

Packed-Decimal Arithmetic Support

Do you provide any support for packed-decimal arithmetic? If not, have more than a few of your customers ever requested it? What is your reaction to adding packed-decimal support to the C language rather than via a library?

- 2 – Support requested/desired.
- 25 – No support.
- 1 – Support in Library.
- 0 – Support in Language.
- Comments:

1. No. So far, Standard C has avoided specifying the methods of storing and computing numbers in favor of specifying the desired results wherever possible. This is a good thing.
2. I'd rather see it as a C++ class rather than part of a language that isn't easily extensible.
3. Support for packed-decimal and zoned-decimal arithmetic has shown up several times on the DEC Users Society (DECUS) wish list of enhancements to the C compilers from Digital. The requests have most often asked for library support rather than native language support. Personally I think that any such extensions should only be considered via library mechanisms.
4. I think that would be contrary to the spirit of C.
5. Definitely no!
6. If provided, it should be in a library; definitely *not* in the language.
7. I would not support such an extension.
8. I haven't thought about it enough to consider many ramifications, but I'd prefer it in the language.
9. I do not plan to add any packed-decimal support. Frankly, there are hundreds of numeric and other formats out there and trying to implement them all as hardwired constructs within the compiler will never work, nor satisfy enough people. This is more a job for object-oriented languages rather than something one should hack into C.
10. Our C and C++ compilers do not support packed-decimal arithmetic. We have received some small number of requests for it. We are considering adding support for it in our C++ compiler product, perhaps as a class library implemented using direct language support for a packed-decimal type. If packed-decimal support is added to the C language, I think it should be specified as an optional extension to the language; i.e., I don't think it should become an official part of Standard C standard unless perhaps it is supported only in the library.
11. We have had no requests for it although we do have COBOL customers.

12. I'd prefer it via a library, but I can see how it would be efficient to have a BCD type and transparent overloading of the arithmetic operators, so I can't say I'd oppose a method of implementing it that was transparent to portable programs.
13. The topic comes up from time to time but so far we have had no requests for it. If we were going to do it, we would add it as an "almost conforming" extension (i.e., break no conforming programs unless they made other use of whatever we chose as a keyword.) Without major inlining, using a library would seem painfully slow—we would oppose any attempt to standardize it that way.

Integration of Preprocessor

Is your preprocessor integrated into your translator? Why did you chose your particular approach?

- 16 – Integrated.
- 10 – Separate.
- Comments:

1. It is much faster to pass a token than a second character stream. We still offer the option of spitting out preprocessed text when that is needed.
2. Integrating it dispenses with an I/O step as well as doing low-level parsing and symbol lookup only once. Also, you can share parsing and evaluation for `#if`. Furthermore, errors are detected and reported in direct sequence rather than hopping around depending on what pass you're in, and a simple preprocessor error doesn't prevent reporting of post-processing errors.
3. It is a separate product which is exploited in many other products.
4. The integrated format allows us to pass detailed source position information on each token to support our X-windows based optimized code debugger.
5. Our non-ANSI preprocessor is integrated. The compile-time benefits of diminished file I/O seem obvious. You can also share expression-parsing code and some scanning code. For our ANSI preprocessor we use a modified external cpp.
6. My preprocessor is not integrated into the translator. It makes little sense to do so especially on an Amiga which has the ability to preload executables at the user's discretion and with near zero overhead running such executables from the front-end.

Apart from that, having looked at two other commercial compilers available for the Amiga I have found no advantage in speed to integrating the preprocessor. Integration only creates headaches and hard-to-find bugs.

7. Yes, our preprocessor is integrated. This was chosen for possibly improved performance, and it seemed a more natural approach with the token-based preprocessing (as opposed to character-based preprocessing) required for ANSI C. It can also easily be made into a separated preprocessor if we wanted to.
8. The preprocessor is not integrated. We did this because it was the easiest to do and the anticipated compilation speed disaster never showed up (even after trigraphs). We still have some cpp problems re ANSI which makes us wish we had taken a stab at integration but we need to do other things as well.
9. There is one problem however. The C preprocessor gets called by other language processors (C++, Fortran, etc.). Do you keep another preprocessor lying around for these guys? (Dual maintenance headaches.) Or do you have these other guys call the new integrated front-end with the -P option? (Slower and some unexpected results).
10. Integrated. Measurements determined that the fastest separate preprocessor took approximately 5–10% of the source-to-object translation time. Surprisingly (to me at least), integrating the preprocessing in as the tokenization for the compiler *reduced* the above source-to-object time by an additional 5% or so! Thus, an inverted adage: the result is less than the difference of its parts. Also, the token-oriented behavior required by the C standard makes a separate (presumably character-oriented) preprocessor's job harder.
11. It is not integrated so we have the flexibility to call a different cpp if necessary.
12. Our preprocessor is a completely separate pass and handles preprocessing directives and parses a source into binary encoded C tokens. We localized all of text processing in a single pass because we wanted to support ASCII, EBCDIC, and other encodings with few modifications and to make the size of each pass small.

Details of Backend

Do you generate intermediate code for a language-independent optimizer and/or code generator or do you have a C-specific backend? Any comments on the tradeoffs involved either way?

- 2 – C-specific.
- 22 – (Mostly) language-independent.

- 1 – Emit assembler.
- Comments:

    1. We generate intermediate code for a backend that was supposed to be language-independent. However, the people who designed the backend only understood Fortran, so our implementation is full of kludges to get around that. In general, if you can get someone who understands all the languages involved to build your backend, a common backend will cut way down on maintenance and porting time.
    2. The back end must know some things peculiar to C and some things peculiar to the other languages. For example, there are IL ops that are just for C and IL ops just for for other languages.
    3. We generate intermediate code for a back-end common to all supported languages. This allows us to quickly implement new optimizations for all languages. The overall cost is considerably less, but the common back end becomes quite complex, and at times one language interferes with the other.
    4. A base intermediate language is used by several frontends, but each language has its own additions to that base. Same for backends (all from shared base but customized for the language). We were not willing to give up customization for each language, but needed to share as much of the technology as possible.
    5. We generate an IL that is shared with Pascal, Fortran, Ada, and Modula2 compilers. From the company's point of view, the benefits of a single optimizer and code generator are obvious—less expense and all languages benefit from the concentrated performance expertise in the optimizer and code generator. Once you re-target the code generator, all languages port to the architecture. This also forces the IL to be general and rich enough so that we support debugging of optimized code.
    6. We generate intermediate code and have a machine-independent optimizer pass that is optional that works on the intermediate code. Currently, we have three code generators (8088, 80386, 68000) and two front ends (C, C++) so we think the concept has proved itself.
    7. Sooner or later, a serious product needs language-specific backends and frontends. I figure you can want three things to be true: fast compilation speeds, excellent generated code, and commonality. But you can actually only achieve two of the three and even that will be difficult.
    8. A language-independent optimizer is used for Fortran, C, and Pascal. A lot of things are done for Fortran only. Inspection of the code shows that Fortran optimization is more aggressive. Also, history shows

that optimizations like loop unrolling are first done for Fortran and then later an attempt is made to make it work for C (like find out how to determine when a C loop is like a Fortran DO loop.) Pascal, C, and Fortran programs can be compiled to the common intermediate code and then that code can be optimized across all of the programs (meaning interprocedural optimizations, inlining, etc.). Some work done on a Fortran-only optimizer shows that better generated code is possible but the compilation rate takes a hit.

9. There are few parts of the optimizations where the choice of input language would make any difference. One is in the handling of aliasing. Perhaps we will make ways for the parser to give language-specific hints about aliases to the back end. This surely can't be as hard as writing another back end for each language.

∞

13. C/C++ Compatibility

Paul Kohlmiller
Control Data Corporation

Abstract

Most ANSI C programs can be compiled with a C++ compiler. This paper identifies those cases where this is not true. Each case has a compilable code segment that should illustrate the problem better than a text description. Also, each case is documented with section numbers from the ANSI C Standard and the Annotated C++ Reference Manual.

Last issue I gave a quick overview of the compatibility problems between C and C++. I identified three scenarios where compatibility issues arise. In this paper I will go into some detail on the first scenario, namely C code that compiles and executes correctly with an ANSI C implementation but does something different in C++. For each such problem I will specify the sections of the ANSI C Standard (X3.159-1989) and the Annotated C++ Reference Manual (aka the ARM, aka the ANSI C++ Base Document) that disagree. I also will give a purely personal opinion about the likelihood that the final ANSI C++ standard will eliminate the incompatible behavior.

For most of the compatibility problems listed below I have supplied some source code. This code will compile without error with an ANSI C implementation (unless otherwise noted). If `main` is defined then it will also execute correctly. The code will produce a fatal compile-time diagnostic or potentially produce different run-time results when a C++ implementation is used (I used AT&T's cfront version 2.0).

1. Everyone tries this sort of program as a first attempt with a new language.

```
main()
{
        printf("Hello, World\n");
}
```

   This will work on every C compiler I know of although the missing `#include <stdio.h>` must be considered bad form[5]. The natural temptation is to try this on a C++ compiler. It will fail because `printf` was used without a declaration. At this point the programmer will mumble

[5]See **Fixed versus Variable Argument List** on page 124 for a discussion of this issue.

something about C++ being over restrictive and may try to solve the problem without including `stdio.h`. So the next attempt might be:

```
extern printf();

main()
{
        printf("Hello, World\n");
}
```

This will still fail to compile because C++ will not accept the "old-style" declarations—an empty argument list implies a `void` argument list. The programmer will then notice what seems to be an easy solution—tell the compiler that `printf` is not a C++ function but, rather, a C function:

```
extern "C" printf();

main()
{
        printf("Hello, World\n");
}
```

Unfortunately this still won't work because C++ will still check the number of arguments. Not only that but now your program won't compile back on your C compiler. The correct code is:

```
#include <stdio.h>

main()
{
        printf("Hello, World\n");
}
```

This works because the C++ implementation has its own set of headers that contain the correct function declarations. In general, programs that try to declare library functions themselves instead of using the standard headers will probably be in trouble.

Of course, the same kinds of trouble can occur with user-written functions. The user must use function prototypes for all declarations. In the above example the correct code (if not using the header) would be:

```
extern "C" printf(const char *, ...);

main()
{
        printf("Hello, World\n");
}
```

Relevant standards references: ANSI C §3.5.4.3, Function Declarators, and ARM §8.2.5, Functions.

There is no chance that ANSI C++ will resolve this. The problem cases noted above all represent code that ANSI C already labels as 'Obsolescent.'

2. The following code may appear to be innocent:

```
i = sizeof('x');
```

This code compiles and executes correctly under both C and C++, but gives different answers for `i`. The problem is that in C, `'x'` has type `int` while C++ says its type is `char`.

Relevant standards references: ANSI C §3.1.3.4, Character Constants, and ARM §2.5.2, Character Constants.

There is no chance that ANSI C++ will resolve this. C++ code often performs I/O without format effectors as in:

```
cout << 'x'
```

If ANSI C++ changed to match ANSI C then this line of code would have to emit an integer value and not the character `'x'`.

3. The following code looks suspicious almost immediately:

```
char aa[3] = "abc";
```

Doesn't the string `"abc"` require 4 characters, one for the zero-byte terminator? Not in ANSI C. In this case, ANSI C will store the three characters and let the user worry about the fact that a zero-byte terminator is missing. C++ still requires the space for the zero-byte terminator so this line of code results in a compile-time diagnostic.

Relevant standards references: ANSI C §3.5.7, Initialization, and ARM §8.4.2, Character Arrays.

There is a chance that ANSI C++ will resolve this. The restriction appears to be unnecessary except to protect the programmer.

4. Now let's look at the first of several problems that occur when structures are involved. The following code compiles just fine with ANSI C:

```
struct X {
        enum {k = 1} foo;
        int k;
};
```

   However, C++ will not accept this code because `k` is used as a constant and as a variable within the same structure declaration. There are other variations on this theme in the ARM.

   Relevant standards references: ANSI C §3.1.2.1, Scopes of Identifiers §3.1.2.3 and Name Spaces of Identifiers, and ARM §9.9, Local Type Names.

   It is very unlikely that ANSI C++ will resolve this. Allowing the above code to work eventually would cause the C++ compiler to accept some rather unbelievable constructs. See the discussion on page 190 of the ARM.

5. Bizarre structure definitions:

```
int foo(struct Z{int y, yy;} X)
{
        return X.y;
}
```

   C will allow the user to define a type in the function prototype scope. A good implementation will warn the user that a type has been created and then immediately goes "out of scope." Going out of scope is somewhat benign in C because a structurally similar type (in a different source file) could be passed to this function and it would work. In C++, a type that immediately goes out of scope means that there is no chance for a program to call this function. Therefore, C++ will generate a diagnostic.

   Relevant standards references: ANSI C §3.5.4, Declarators, and ARM §8.2.5, Functions.

   There is no chance that ANSI C++ will resolve this. The function would be uncallable in C++.

6. In the above example we saw that function arguments cannot contain their own definitions. The same is true for function return types.

```
struct X {int y, yy;} foo()
{
        struct X Z;

        Z.y = 1;
        return Z;
}
```

This is valid in ANSI C but invalid in C++. The reason is similar to that in the previous example. The definition of the structure is out of scope anywhere outside the function itself so it cannot be called. The error that I get from my copy of C++ is that `X` is undefined at the line that is trying to define it.

Relevant standards references: ANSI C §3.5, Declarations, and ARM §8.2.5, Functions.

There is no chance that ANSI C++ will resolve this. The function would be uncallable in C++.

7. A major problem for code written in C and ported to C++ is that the name space rules are different. For example:

```
struct X {int x, y;} rec;
typedef int X;
```

This is valid in ANSI C but C++ complains because structure tag names occupy the same name space as 'ordinary' names (which includes `typedef` names). In this example, C++ thinks `X` is being redefined.

Relevant standards references: ANSI C §3.1.2.3, Name Spaces of Identifiers, and ARM §7.1.3, The typedef specifier.

There is no chance that ANSI C++ will resolve this. Remember that C++ needs to differentiate types in order to perform type-safe linking between functions. If tag names and `typedef` names are allowed to be spelled the same, then disambiguating the two types becomes harder due to the current way in which these names are mangled.

8. Is the following code valid?:

```
int gg = 1;
int gg; /* ? */
```

It is in C. The first line is a definition of `gg` because it has an initializer and it requires storage. The second line is merely a declaration of `gg` so there is no conflict. C++ says that any declaration is also a definition unless it has a storage specifier of `extern` and it has no initializers.

Relevant standards references: ANSI C §3.5, Declarations, and ARM §7.1.1, Storage Class Specifiers.

There is no chance that ANSI C++ will resolve this. C++ treats builtin types (such as `int`) the same way that it treats user-defined types. Consider what would happen with a user defined type:

```
myint gg = 1;
myint gg;
```

The first declaration of `gg` calls a constructor. What can the second declaration do? It must also call a constructor but now the first declaration is lost—it is not even possible to call its destructor. (This problem occurs again in item 14 below.)

9. In C++ a structure defines a scope. In C a structure has the same scope as the block that contains it. For example:

```
struct x {
        enum E1 {i = 1, j = 2, k = 3} fld1;
        int fld2;
} y;

enum E1 name1;

int name2 = k;          /* error */

main()
{
        name1 = 300;    /* error */
}
```

The first error arises in C++ because the `enum E1` definition is no longer in scope. Note that the definition of `name1` is OK, however, because the tag name is still available. The second error arises since in C++, it is not permitted to assign an `int` to an `enum`. This is deemed a type mismatch.

Relevant standards references: ANSI C §3.5.2.2, Enumeration Specifiers, and ARM §7.2, Enumeration Declarations.

There is no chance that ANSI C++ will resolve this. The notion that a structure (or class) establishes a scope is a part of the C++ fabric.

10. The following keywords are reserved in C++ but not in ANSI C: `asm`, `catch`, `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `public`, `protected`, `template`, `this`, `throw`, `try`, and `virtual`. Older versions of C++ also may use `overload` as a keyword. Certainly, a C program that uses any of these keywords as an identifier will not work with C++.

Relevant standards references: ANSI C §3.1.1, Keywords, and ARM §2.4, Keywords.

There is no chance that ANSI C++ will resolve this, except that `asm` may be dropped. The keyword `overload` is not defined in the ARM. Note that `catch`, `template`, `throw`, `and` `try` are new keywords. Some new keywords may also be adopted during future X3J16/WG21 deliberations.

11. C++ lacks certain support for incomplete types. For example, the following will not work:

```
char a[];
void *b = a;
char a[10];
```

ANSI C will accept this use of an incomplete type.

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

There is no chance that ANSI C++ will resolve this. In this example, `char a[];` is a definition of an array without a size—that is an error. The type completion at `char a[10];` is a redefinition so that is also an error.

12. C++ is none too supportive about C's use of `void *` in many instances. It will flag assignment expressions that use pointer to `void` as the right hand side of an assignment statement. For example:

```
char a[10];
void *b = a;

void foo()
{
        char *c = b;
}
```

ANSI C will accept this usage of pointer to `void` being assigned to a pointer of another type. C++ will not.

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

There is no chance that ANSI C++ will resolve this. C++ is a more strongly typed language than C and this is an example of that strong typing. The example could be made to work with explicit casts.

13. A `const` item at file scope has external linkage in ANSI C. That means that:

```
const int j;
```

    is valid in ANSI C because `j` might be defined in another translation unit. In C++ such items have internal linkage. As a result, C++ will detect this as an error. Without an `extern` specifier this code is a definition in C++ and it doesn't make sense to define a constant to have an undefined value. In general C++ programs use `const` instead of `#define` to declare constants.

    Relevant standards references: ANSI C §3.1.2.2, Linkages of Identifiers, and ARM §3.3, Program and Linkage.

    There is no chance that ANSI C++ will resolve this. The C++ community appears to be heading toward a consensus that the C preprocessor is not needed. The internal linkage of `const` is considered an aid toward that goal. If a `const` item has an internal linkage and therefore has to be defined when it is declared, then the compiler can make all of the optimizing decisions it could have made if a `#define` was used.

    If a program is changed to use `const` instead of `#define` to define a constant, the object must also be made `static` so multiple definition errors won't result when it is compiled with a C compiler.

14. The `goto` statement. What value is printed for `i`?

```
#include <stdio.h>

main()
{
        int i = 2;

        goto label1;
        {
                int i = 3;

label1:         printf("i is %d\n", i);
        }
}
```

    The value printed is undefined—a conforming ANSI C implementation might print 3 or any other number. C++ disallows jumping past initializers and not just because of the undefined results. In C++ a declaration invokes a constructor that at some level may do something like call `malloc`. When the block containing the constructor is exited, a destructor

is invoked automatically. This destructor may do something like call `free`. This means that `free` can be called without there being a corresponding `malloc`.

Relevant standards references: ANSI C §3.6.6.1, The `goto` Statement, and ARM §6.7, Declaration Statement.

There is no chance that ANSI C++ will resolve this. The reasoning above does not apply to this example because there is no destructor to worry about. However, the problem is that C++ wants to treat builtin types the same way it treats user-defined types.

15. Trigraphs. The ARM defines only 6 trigraphs but the ANSI C standard defines 9. Those missing are `??<` for {, `??>` for }, and `??-` for `~`. It turns out that this was simply an error that crept into the final printed copy of the ARM. There is no intended difference here.

    Relevant standards references: ANSI C §2.2.1.1, Trigraph Sequences, and ARM §16.2, Trigraph Sequences.

    This oversight is corrected in the working draft of the C++ standard.

16. There are preprocessor problems caused by the fact that some C++ implementations use an ANSI C preprocessor while others use an older "Classic C" preprocessor. Most ANSI C preprocessors work on tokens instead of character strings. This can be a problem for some C++ programs but won't affect ANSI C programs because C++ has some unique tokens. The problem for C programs is illustrated below.

```
#include <stdio.h>

main()
{
        printf("%d\n", __STDC__);
}
```

    An ANSI C implementation will cause this program to print the value 1. What happens on C++ is anyone's guess. Most likely it will fail to compile with `__STDC__` undefined. The problem is significant for code that uses `__STDC__` to include or exclude code.

    Relevant standards references: ANSI C §3.8.8, Predefined Macro Names, and ARM §16.10, Predefined Names.

    It is doubtful this will be changed by ANSI C++. There is some sentiment for not forcing an ANSI C++ implementation to use an ANSI C preprocessor.

17. Ideally, the preprocessor will handle C++-style comments (`//`) correctly.

```
#include <stdio.h>

main()
{
        int a = 4, b = 2;

        printf("%d\n", a//*comment*/b);
}
```

    If the `//` is taken as the beginning of a comment then this program will have a compilation error. Many C compilers will have difficulty with this code because allowing C++-style comments is a common extension.

    Relevant standards references: ANSI C §3.1.9, Comments, and ARM §2.2, Comments.

    There is no chance that ANSI C++ will resolve this. More likely, the C++-style comments will be presented to the committee that looks at ANSI C revisions.

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.

∞

14. Adding a Bit Type to C

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

Abstract

There is interest in adding a bit type to C because it allows values to be stored in a packed format. Although a macro solution is available for a limited bit implementation, it is neither general nor easily extensible. An exploratory proposal is presented that extends bit-fields to allow declarations of both arrays of bit-fields and pointers to bit-fields. Finally, an extension is discussed that allows bit-fields to be declared with a range of values exceeding that of the `long` and `unsigned long` types. This extension is presented as an alternative approach to some current extensions like `long long int` that increase the integer range.

Introduction

There have been several proposals made to NCEG to add more support for bit types to the C standard. Most of these proposals are part of some data-parallel extension like array syntax. For example, the C* language is a data-parallel language that runs on the Connection Machine and supports the type `bool`, an unsigned single-bit data type. Committee X3J3 entertained the notion of adding a bit type to the emerging Fortran-90 standard, but dropped it later on as a way of simplifying the language, and to eliminate some portability concerns. Here at Cray Research we have received some inquiries from customers about ways to improve bit manipulation in our supported languages. Adding a consistently defined and generally usable bit type to C would be welcomed by many programmers because it allows certain algorithms to use less memory while retaining familiar programming techniques. There are several levels of support that can be provided for a new bit type. Some provide minimal support but have no effect on the current language definition. Others provide extensive support but also raise questions about the size of the language and the implementation cost.

Motivation

The primary motivation for wanting additional support for bit types is to be able to represent many results while consuming less space. There is some support for bit manipulation already in the language through the use of bit-fields and bitwise operators. Unfortunately this support is not general enough. Neither arrays of bit-fields nor taking the address of a bit-field is allowed by the C Standard. The bitwise operators are limited to operating on the number of bits present in the standard integral types. The C Standard permits declarations of an array of structures with a bit-field member as follows:

```
struct {
   int bf:1;
} abf[100];
```

but this does not conserve memory because structures are padded out to a byte or word boundary. It is possible to declare a sequence of bit-fields but variations in the different implementations make portability and efficiency difficult to achieve. Bit-fields, as they exist in Standard C today, are not general enough for many of the applications that could benefit from them.

The current set of general integral types are:

```
signed char       unsigned char
signed short      unsigned short
signed int        unsigned int
signed long       unsigned long
```

Except that they waste space, these can be used quite effectively in place of a bit type. A new bit type would just be another integral type that happens to also conserve memory.

One use for a bit type is to define a logical vector that indicates a relationship between two other arrays. The following example shows how a character array can be used to represent a set of logical values.

```
int i;
unsigned char flags[100];
double x[100], y[100];

for (i = 0; i < 100; i++)
   flags[i] = x[i] < y[i];
```

Each element of `flags` indicates a relationship between corresponding elements of `x` and `y`. Since the proposed bit type is often associated with an array-syntax proposal, the previous example looks something like the following when a bit type is used:

```
int i;
bit flags[100];
double x[100], y[100];

flags[:] = x[:] < y[:];
```

This set of logical values can be stored in tightly packed storage, thus consuming approximately one-eighth the memory of most character-array implementations. The logical values can be used later to control references to corresponding array elements.

Another potential use of a bit type is to represent the location of a shape in a particular geometry. For example, a two-dimensional bit array could be defined such that `0` represents the background and `1` represents a shape. An $n \times m$ rectangle that rests on the horizontal axis with its left side on the vertical axis is created with the following simple function:

```
bit screen[ROWS][COLS];

void horiz_rect(int n, int m) {
   int i, j;

   for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
         screen[i][j] = 1;
}
```

Functions could be defined to create other geometric shapes such as circles, squares, and curves, all represented in a tightly packed bit map. Since bit maps are often used to represent large arrays, a compact representation is desirable. For instance, a 1024×1024 screen is represented by over 10^6 bits.

Macro Solution

It is possible to define a macro package that references individual bits inside an integer array. The idea is to declare an array of elements, where each element has type `unsigned long`. Masks and bitwise operators are used to reference the individual bits of each element. This means the bit index used to reference an individual bit contains two pieces of information: an index into the array, and a bit offset into the array element. First, the bit-offset portion of the index is shifted off before indexing into the array. Next, the bit-offset portion is masked off and used to shift a 1 into the corresponding bit position to mark the bit location within the array element. A bit is set by ORing the shifted 1 into the array element, cleared by ANDing the complement of the shifted 1 into the element, and tested by simply ANDing with the shifted 1. The next example shows an implementation that works on the Cray-YMP system.

```
#include <stdio.h>

#define bpw   64UL              /* bits per word */
#define b_off 6UL /*bitsize of bitoffset into word*/
#define b_mask (~0UL>>bpw-b_off) /* bitoffset mask*/

#define N     1024UL    /* numb of elems in array */
#define nbits (N*bpw)  /* numb of individual bits */

#define ELEM(A,I) (A[(I)>>b_off])  /* ref element */
#define BIT(I) (1UL<<((I)&b_mask)) /*mark bit loc */
#define SET(A,I) (ELEM(A,I) |= BIT(I)) /* set bit */
#define CLR(A,I) (ELEM(A,I) &= ~BIT(I)) /*clr bit */
#define TST(A,I) (ELEM(A,I) & BIT(I)) /* test bit */

unsigned long bits[N];

main () {
   unsigned long i;        /* used as a bit index */

   for (i = 0; i < nbits; i += 2)
      SET(bits, i);        /* set every other bit */

   for (i = 0; i < nbits; i += 2) {
      if (TST(bits, i) == 0)
         printf("ERROR - bits[%lu] is clear\n", i);
      CLR(bits, i);             /* clear the bit */
   }

   for (i = 0; i < nbits; i++)
      if (TST(bits, i))
         printf("ERROR - bits[%lu] is set\n", i);

   printf("All Done!\n");
}
```

It is easy to modify this macro package to work on most 32-bit implementations by simply changing the values of `bpw` to 32 and `b_off` to 5.

Although this macro package simulates a simple vector of bits nicely, it is not directly extensible to more complicated problems. For instance, there is no way to take the address of a particular bit. Both the underlying array address and the bit index into the array are required to represent the address. This makes more complicated, operations such as manipulating a subsection of a simulated bit array or passing bit subsections as arguments to functions. There is no direct analogy to multi-dimensional arrays, such as those required for geometric shapes identified above. The multidimensional macro packages are more complicated

and a unique package is required for each array rank that is needed. Finally, the macro package would have to be changed to support simulated arrays of 2-bit elements, or 3-bit elements, or arbitrary n-bit elements.

Multi-Bit Elements

Arrays of n-bit elements are useful when the application permits more than two values for each element. For instance, consider the geometric shape example again. The application might be defined to allow geometric shapes of different colors. If the goal is to allow patterns such as a red triangle, green circle, blue square, yellow triangle, with a black background and a white border, to reside on the same screen, then a natural representation is to declare a two dimensional array for the screen, and have each element be a 3-bit integer. This allows each element to represent eight different colors depending upon its value. Cycling through the colors is accomplished by incrementing the elements of a particular shape. Thus, integer arithmetic on bit elements is an important operation. Again, the primary motivation for the bit-type is to use less memory, but it is equally important to define all integer operations for bit operands in order to have a more general and useful feature.

An Exploratory Proposal

Before a general bit type can be added to C many key issues need to be addressed. Implementors may want to implement a subset of what is defined here, or they may decide the implementation costs outweigh any benefits gained from adding a bit-type. The intent is to make it easier to decide what is really needed and what is too complicated. A complete bit-type feature does permeate the entire language. And since I have not explored the impact on every corner of the language, further work may be needed to reach linguistic closure.

The essence of this proposal is to allow bit-field declarations outside of structure declarations. This part of the proposal allows any bit-field declaration that currently is permitted inside structure and union declarations to also be declared outside of structure and union declarations. Later on additional extensions will be considered. The following declaration:

```
unsigned ub4:4;
```

declares variable `ub4` to have 4 bits of precision and a range of values `[0,15]`. However, the amount of storage that `ub4` occupies can be larger than 4 bits—there might be padding. That is, `ub4` behaves very much like a standard bit-field. Similarly, the following:

```
signed sb4:4;
```

has four bits of precision and a range of values [-8,7], assuming a twos-comp representation. Again, there is a direct analogy to standard bit-fields. Since Standard C states that it is implementation defined whether the high order bit of a bit-field declared with type `int` is a sign bit or not, it seems reasonable to grant implementors the same latitude here. The *integral promotions*, as defined in the C Standard, state that bit-fields can be used in expressions wherever an `int` or `unsigned int` can be used, and that they are converted to type `int` if it can represent all the possible values, otherwise bit fields are converted to type `unsigned int`. Again, it seems reasonable to to apply the same rule to this proposed extension to bit-fields.

One issue is defining the best syntax for declaring arrays of bit fields. Since arrays of bit-fields do not exist in Standard C, new syntax has to be invented. I have found in the past that issues surrounding new syntax stir unbridled emotions in many people. This tends to cloud the exploration of new concepts. There are bigger issues to dwell on at this stage of the proposal. For now the following use of a `typedef` will suffice:

```
typedef unsigned UB4:4

UB4 a4[N];  /* array of bit-fields */
UB4 *pa4;   /* pointer to bit-field */
UB4 f();    /* function returning bit-field */
```

The need for bit constants that begin with `0B` or `0b` such as:

```
0B110011001100
```

also needs examination if binary is going to be at the same level as octal, decimal, and hexadecimal. *[Ed: Such constants would be useful outside of this propoasl as well.]* Similarly there might be a desire to support bit strings such as:

```
B"111000110010"
```

to place bit arrays at the same level as character strings and wide-character strings.

Pointers to Bit-Fields

As soon as arrays of bit-fields are introduced, then pointers to bit fields must also exist, because expressions designating arrays are converted to pointers to their first elements for most operations. The representation of a pointer to a bit-field may not be straightforward for some implementations. Since the Cray-YMP has 64-bit words but word addresses require only 32-bits, there is plenty of room to store the 6 bits needed for the bit-offset into the Cray word pointer.

Many 32-bit byte-addressable implementations do not have an extra three bits to spare in their pointer representation, to represent the bit-offset into the byte. An initial thought might be to limit arrays of bit-fields to 29 bits of byte address, thereby leaving 3 bits for the bit-offset into the byte. However, this can cause problems with dynamic memory allocators such as `malloc` that do not know the type of the object being allocated. A virtual-address implementation can easily consume more than 29 bits of address space. For these implementations I see no other solution than to use at least an additional contiguous byte to represent the bit-offset. Some implementations might have to use an entire additional word. This is unfortunate because most of the bits of the additional word are not used. However, the savings in space that is possible with arrays of packed bit-fields still creates an overall savings for all but the smallest arrays. Additional issues with pointers to bit fields are examined later.

Bit-Field Alignment

Another issue with tightly packed bit-fields arises when a single element crosses a storage boundary. On some implementations this is a byte boundary, and on other implementations it is a word boundary. Consider the following example:

```
typedef unsigned UB3:3;

UB3 a3[100];
```

Assume a 32-bit byte-addressable architecture. If every element is tightly packed then some elements will cross both byte and word storage boundaries. This may cause problems for efficient code generation. It is for this very reason that the C Standard allows an implementation to pad out to the next addressable storage unit when a bit-field does not fit into the remaining space of the current addressable unit. A similar technique can be used for arrays of bit-fields. A brief side issue must be explored to explain this idea.

The header `<limits.h>` defines expressions that represent the minimum and maximum values for the integral types. It is a common misconception that an examination of these values gives information about the amount of storage required to store variables declared with corresponding types. In *Volume 1, number 4* of *The Journal*, I wrote an article, *Cray C: C in a 64-bit World* that describes several aspects of the Cray Research C implementation that are unique. One of these involves the range and size of the `short` type. Essentially, the Cray Research implementation exploits the fact that the C Standard allows arithmetic types to occupy more storage than is needed for the precision of the type. The Cray architecture has two different sizes of scalar registers (64-bits and 32-bits), but one memory word size (64-bits). Since all loads and stores involving `short` objects go through the 32-bit registers, the amount of memory used to store `short` objects is greater than number of bits of precision. Specifically, both of the following expressions evaluate to true on a Cray-YMP:

```
sizeof(short) == 8 && SHRT_MAX == (1 << 31) - 1
sizeof(short) == sizeof(long) && SHRT_MAX < LONG_MAX
```

This technique of allowing the actual storage to have more bits than the precision can be used to prevent bit-field elements from crossing a storage allocation boundary. For instance the following declaration:

```
typedef unsigned UB3:3;

UB3 a3[100];
```

could be implemented such that each bit-field element of `a3` occupies 4 bits of memory (or even 8 bits) but only yields 3 bits of precision. Since 4 bits pack tightly into 8-bit, 32-bit, and 64-bit boundaries, no elements cross an addressable boundary. As a more complete illustration of this concept the following mappings for bit fields would make sense for the Cray Research implementation:

Declaration	*Precision*	*Storage*
`int i:1`	1	1
`int i:2`	2	2
`int i:3`	3	4
`int i:4`	4	4
`int i:5`	5	8
`int i:6`	6	8
`int i:7`	7	8
`int i:8`	8	8
`int i:9`	9	16
`int i:15`	15	16
`int i:16`	16	16
`int i:17`	17	32
`int i:31`	31	32
`int i:32`	32	32
`int i:33`	33	64
`int i:63`	63	64
`int i:64`	64	64

Of course 32-bit architectures may not want to support bit-field sizes greater than 32. Allowing this padding means that structures with consecutive 3-bit bit-fields may not lay out in memory the same way as an array of 3-bit bit-fields. However, the C Standard does not guarantee that several consecutive structure members with identical types lay out in memory the same as an array of that type.

More on Pointers to Bit-Fields

The relationship that pointers to bit-fields will have with the generic pointer, `void *`, needs to be investigated. Currently, Standard C requires generic pointers to have the same representation as a character pointer. Since Standard C identifies objects with character types as objects that have the least strict alignment requirement, and that can have their address taken, this one-to-one correspondence is consistent. The introduction of a generalized bit-type feature forces examination of the problems introduced by changing the representation of `void *` to the representation of a bit pointer. That is, define the least strict alignment requirement to be bit alignment. Such a change does introduce problems with library functions such as `memset`, because a pointer argument is expected that, at a maximum, points to a `char`-aligned object. If a function argument can be passed that points into the middle of a `char`, then the implementation and specification of `memset` is more difficult. For this reason it does not seem like a good idea to change the current specification of `void *`. There are several obvious choices for new library functions like `bitset` and `bitcmp` that accept a bit pointer. Something similar to the following can be used to define a generic bit-pointer type, `bit_t`:

```
typedef unsigned UB1:1;
typedef UB1 *bit_t;
```

The maximum size of a bit array might not be limited by the amount of addressable memory, but rather by the value of `ULONG_MAX`. This is because the number of bits available might be greater than the value of `ULONG_MAX`. This issue can also affect the range of elements that can be accurately represented by the type `prtdiff_t` when two bit pointers are subtracted. That is, it might be possible to have more bit-field elements in an array than can be represented by the `ptrdiff_t` type. *[Ed:* `ptrdiff_t` *already is not guaranteed to cover the difference between two arbitrary character pointers.]*

A new keyword like `bitsizeof` is needed to adequately handle dynamic memory allocation of arrays of bit-fields, and to portably compute the number of elements in an array. One issue concerns the type `size_t` that represents the type of the value computed by the `sizeof` operator. If `size_t` is also used for the new `bitsizeof` operator, then there is a limit to how many elements can be present in an array of bit-fields. This raises a slightly more complicated issue with trying to use `malloc` to dynamically allocate arrays of bit-fields. Currently `malloc` returns a `void *` pointer that might not be compatible with the bit pointer being assigned to. Similarly, `malloc` expects an argument with type `size_t` that specifies the number of characters in the array. One possibility is to define a macro that converts bit sizes to byte sizes for this purpose. The idea is to round up to the nearest byte length. For example:

```
#define BITS_TO_BYTES(X)  ((X) + 7U >> 3)
```

works if the value of `CHAR_BIT` is 8 (the macro `CHAR_BIT` is defined in the `limits.h` header). Another possibility is to define assignment compatibility such that pointers to objects that are not bit pointers can be assigned to bit pointers, but not the other way around. This is somewhat like the rules for *pointer to* `const int` and *pointer to* `int`. It is permissible to assign in one direction but not the other.

```
typedef unsigned UB1:1;
UB1 *pb;
const int *cp;
int *p;

cp = p; pb = p;  /* OK */
p = cp; p = pb;  /* ERROR */
```

If this rule is introduced then the aliasing implications need closer examination. It is always possible to define a new library function for dynamically allocating bit arrays, but that should be a last resort.

Optimizations

There are additional optimization possibilities with arrays of bit fields. For example, performing a bitwise-AND operation on two arrays of bit-fields can often times be done on multiple elements at a time with a single hardware instruction. Assume a 32-bit, two's complement, byte-addressable architecture with 32-bit registers R1, R2, and R3. Given the following example:

```
typedef unsigned UB1:1;
UB1 A[32], B[32], C[32];
unsigned long i;

for (i = 0; i < 32; i++)
   A[i] = B[i] & C[i];
```

then, assuming `A`, `B`, and `C` are properly aligned, it is possible to generate code similar to the following sequence:

1) Load R1 with 32 contiguous bits from `B`

2) Load R2 with 32 contiguous bits from `C`

3) R3 = R1 & R2

4) Store R3 into `A`

This effectively treats the registers as vectors of 32 elements. Since there are

only 32 elements in each array the entire loop is computed in a single iteration. *Strip mining* techniques can be used on loops with longer trip counts to get performance gains. Another example that can be optimized is the following:

```
typedef unsigned UB4:4;
UB4 A[8], B[8];
unsigned long i;

for (i = 0; i < 8; i++)
   A[i] = B[i] << 1;
```

In this case the 32-bit registers can be used as vectors of 8 4-bit elements each.

1) Load R1 with eight contiguous elements from `B`

2) R1 contains some bit pattern
   `R1 = XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX`

3) Load R2 with the following bit pattern:
   `R2 = 0111 0111 0111 0111 0111 0111 0111 0111`

4) R3 = R1 & R2:
   `R3 = 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX`

5) Shift R3 left one bit:
   `R3 = XXX0 XXX0 XXX0 XXX0 XXX0 XXX0 XXX0 XXX0`

6) Store R3 into `A`

A final optimization example is:

```
typedef unsigned UB3:3
UB3 A[8];
unsigned long i;

for (i = 0; i < 8; i++)
   --A[i];
```

Assuming the 3-bit elements are padded out to 4 bits, the 32-bit registers can be used as 8 element vectors as follows:

1) Load R1 with eight contiguous elements from `B`
   `R1 = ?XXX ?XXX ?XXX ?XXX ?XXX ?XXX ?XXX ?XXX`

2) Load R2 with the following bit pattern:
   `R2 = 0111 0111 0111 0111 0111 0111 0111 0111`

3) R3 = R1 & R2 ; clear pad
   `R3 = 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX 0XXX`

4) R3 = R3 + R2 ; decrement

5) Store R3 into `A`

Depending upon the available instruction set, optimizations can be performed for bitwise operations, addition, subtraction, and comparisons by using the 32-bit register as a vector of elements. Better performance results are possible with 64-bit registers.

Extended Integer Range

There are some implementations that support a `long long` type. The NCEG committee recently created a new subcommittee to examine the impact of extended integer ranges. The primary motivation for adding the new `long long` type is to support at least 64-bit integral types. One obvious question that has come up about this proposal is the long-term effect on the language. Is the next step a `long long long` type, with constants such as `3LLL` and `3ULLL` or `3LULL`?

Another possibility is to extend my bit-field proposal to include bit lengths that exceed the range of types `long` and `unsigned long`. These bit-fields are called *extended bit fields.* Each implementation has to define a maximum bit length and corresponding types such as:

```
#define BIT_MAX 64

typedef unsigned maxuint_t:BIT_MAX
typedef signed maxint_t:BIT_MAX
```

that place upper limits on the maximum precision supported for extended bit-fields. This idea seems upward compatible but there are many linguistic hurdles that have to be overcome before it becomes workable.

Declaring constants with an extended range is no longer straightforward. It is certainly consistent to define constants that cannot be represented as type `unsigned long` as having the type of the nearest extended bit-field type that can represent the value. However, it is sometimes necessary to represent smaller constants in an extended precision also. For instance, there should be some way

of defining a mask that is `BIT_MAX` bits long. Again syntax needs to be invented that is acceptable. One possible syntax is to use a new suffix, say `M`, that can be used to specify integer constants with an extended bit-field type. For example:

```
3M     /* maxint_t by default */
3UM    /* maxuint_t by default */
3M.37  /* 37 bits of precision */
```

specifies the integer 3 is an extended bit-field. A mask of 42 ones is created with the following expression:

```
~0UM.42
```

by using this approach. No doubt a more elegant syntax could be invented.

The usual arithmetic conversions have to be changed to produce a common type when one of the operands is an extended bit-field. The most straightforward approach is to state that the result of the type is the type of the operand with the greatest precision. If both operands have the same precision and one of the operands is unsigned then the result is also unsigned.

Passing extended bit-fields as arguments to functions has the biggest impact on C. Although user defined functions can use function prototypes to convert an argument to an extended bit-field with greater precision, variable number of argument functions such as `printf` do not support the concept of an extended bit-field. New print-specifier syntax is needed along the lines of:

```
%~43d
```

to print a 43-bit extended bit-field in decimal format. A `%b` print-specifier that prints binary format is also needed for completeness anyway. The overall impact on the formatted I/O functions is substantial. There is a similar impact on the conversion functions `strtol` and `strtoul`. New conversion functions are probably needed that return values with types `maxint_t` and `maxuint_t` respectively. These new functions must also understand the new syntax for extended range constants. Other library issues are defining functions that are analogous to the `abs`, `labs`, `div`, and `ldiv` functions for the `maxint_t` type.

Conclusions

The total impact of adding a bit-type to C depends upon the extent to which it is defined. There are several levels at which a bit-type can be implemented. The first level is to define a macro package that is quite limited but still usable for many needs. This requires no additional language features. The second level is to define a new `bit` keyword that allows declarations of a single-bit integer (probably unsigned). Arrays of bits and pointers to bits are also permitted. The introduction of a bit pointer, however, is a non-trivial addition to the language,

because it introduces an alignment that is less strict than character alignment. Additional linguistic features such as `bitsizeof` are needed to support these less strict alignments. The third level is to permit bit-fields to be declared outside of structures and unions. Once a bit pointer exists, this level is a straightforward and general extension. The final level is to implement extended bit-fields as an alternative to extensions like the `long long` type. Although this level provides a path to general extensions of integer ranges, it also introduces the greatest impact on the language, especially in the area of argument passing to standard library functions.

Extensions always have an effect on programming languages. More research is needed to determine which level provides the greatest benefit when factors such as language size, linguistic complexity, and extensibility of the language are considered.

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

∞

15. Pragmania

Rex Jaeschke

Borland C/C++ V2.0

Registering Startup and Termination Functions

Standard C provides the `atexit` library function to allow specified functions to be called at normal program termination. Borland has augmented this mechanism via two pragmas which have the following syntax:

```
#pragma startup function-name [ priority ]
#pragma exit    function-name [ priority ]
```

Functions registered using `startup` are called before `main` gets control. Functions registered using `exit` are called after any functions registered using `atexit`. As with `atexit`, all functions registered via these pragmas must have a `void` argument list and `void` return type.

The optional *priority* allows the programmer to control the order in which registered functions are executed. The valid range of priorities is 0–255 with 0 being the highest. Priorities 0–63 are reserved for use by library routines. The default priority is 100. At startup, functions are called in decreasing order of priority while at shutdown, they are called in increasing order.

As far as I could ascertain by experiment, for a set of functions having the same priority, they are invoked in the reverse order of their registration, provided they were all registered in the same translation unit. For registrations at the same priority in different translation units, the execution order seems to be affected by the order in which the object modules are linked.

A feature that might be useful, but which is not provided, would be the ability to override the default priority via a command-line option.

Miscellaneous Pragmas

By default, the compiler warns if any formal parameters in a function definition go unreferenced in that definition. Such warnings can, however, be suppressed using the following pragma:

```
#pragma argsused
```

This directive must be used outside of a function definition. It suppresses the warning for the next function definition only.

Borland C/C++ programs may contain embedded assembly code. When assembler instructions are encountered, the compiler restarts itself in a different mode. To avoid the need for this restart, a pragma can be used to indicate this need. Obviously, this pragma is best placed at the start of the translation unit. Its syntax is:

```
#pragma inline
```

Various command-line compiler options can be embedded in source via an options pragma. It has the general syntax

```
#pragma option [ options ... ]
```

In certain DOS-specific functions (called `huge` functions), it is desirable to ensure that the values of all registers go unchanged when such functions are called. This is achieved via the following pragma:

```
#pragma saveregs
```

which should be placed immediately before the corresponding function definition. It applies to that function only.

A number of warning messages can be enabled and disabled via command-line options. This can also be achieved via the `warn` pragma. The directives

```
#pragma warn +aaa
#pragma warn -bbb
#pragma warn .ccc
```

enable warning *aaa*, disable warning *bbb*, and restore warning *ccc* to its compile-time default value, respectively. *aaa*, *bbb*, and *ccc* are three-character abbreviations. For example, 'cln' corresponds to the warning 'Constant is long.'

∞

16. C Standards Update

Jim Brodie
Motorola, Inc.
Tempe, Arizona

Abstract

An effort is underway to coordinate the development, review, and adoption of future changes to the ISO and ANSI C language standards. To help facilitate this, X3J11 is faced with the decision on whether to recommend that the United States withdraw its current standard and adopt the ISO standard in its place. In this article, we will look at the issues surrounding this decision.

X3J11 recently held a letter ballot to decide whether to recommend to X3 (X3J11's parent standards body) to simultaneously withdraw the current American National Standard for C, X3/159-1989, and replace it with the recently adopted International Standards Organization (ISO) standard for C, document ISO 9899:1990. The result of this letter ballot was 27 votes in favor and 3 opposed (with 4 more Yes ballots arriving after the voting deadline had past). This result was a clear 2/3 majority vote supporting the change to the ISO standard document. However, according to X3 rules, the 3 negative ballots must be addressed and a committee position taken on how to respond to them. Assuming that a consensus forms around the appropriate responses to the negative ballots, the next step would be a formal X3 ballot on this proposal.

Why Make the Change?

The obvious question, given all the years of struggle to produce an American National Standards Institute (ANSI) standard for C, is why would X3J11 make this request in the first place?

First, it is important to understand that the ANSI and ISO standards are technically identical. The documents, however, were prepared using different document-preparation style conventions. The ANSI C standard was formatted according to the then current ANSI document style rules. The ISO document was formatted, not surprisingly, according to the international standard style rules.

Foreseeing this situation, the document editor for the ANSI C standard had made several attempts to get permission to use the international style rules for the ANSI C standard. However, at that time ANSI was unwilling to accept the

document in that format because it did not meet their document standards. (If there is one organization that is bound and determined to follow their own standards, it's ANSI!) It is interesting to note that ANSI has since adopted (or at least allows) the ISO style conventions in its documents.

One of the unfortunate results of these differing style rules is that additional sections were introduced in the front material of the ISO document. This means that the numbering of document sections is different for the ISO and ANSI standards. This complicates discussions between the U.S. and ISO communities, particularly when discussing issues such as interpretations where specific section references are frequently used.

The common standard document issue, however, is broader than simply trying to work together with documents that have differing section numbers.

From the creation of X3J11, one of the stated goals of the committee has been to have identical ISO and ANSI standards. There have been repeated votes and actions by X3J11 that have supported this position. In fact, the ANSI standard was delayed between one and two years as attempts were made to incorporate support for the international community. The committee and, we believe, the C community in general still support this position.

In an effort to ensure the on-going cooperation and coordination between the ANSI work and ISO work, several X3J11 members have been actively involved in the ISO standards committee that is addressing the C language. (The committee is known as SC22 Working Group 14 or WG14 for short). P.J. Plauger, the X3J11 Secretary, is serving as the Convenor of WG14. Rex Jaeschke is the International Representative from X3J11 to WG14 and Tom Plum, Vice Chair of X3J11, has regularly attended the WG14 meetings. X3J11 itself has served as the U.S. Technical Advisory Group (TAG) that reviews all technical issues raised by WG14 and votes to establish the U.S. position on technical questions.

This cooperative work paid its initial dividends when the ISO C language standard, which is technically equivalent to the ANSI standard, was adopted. This cooperative work is currently focused on the Normative Addendum that will extend the current standard. (In previous articles we have discussed some of the issues being addressed by the Normative Addendum. See *C Standard Update* in *The Journal Volume 1, number 4* and *Extended Multibyte Support* in *Volume 2, number 2.*)

Every effort is being made to ensure that, now that we have close synchronization of the ISO and ANSI C standards, we don't lose this coordination when the Normative Addendum is adopted by WG14.

If the ISO and ANSI efforts are not appropriately synchronized we could end up in a position where the ISO standard is changed by the Normative Addendum, but the ANSI standard does not include these changes. If we then go through a process to revise the ANSI standard to be consistent with the ISO standard, any public review comments which cause changes in the content of the Normative Addendum would force the ANSI standard to remain inconsistent with the ISO standard. We could get stuck in an almost endless cycle of changes in an attempt to match the "other" standard.

The best way to proceed is to establish a close synchronization between the review and adoption of the Normative Addendum for inclusion in the ANSI and ISO standards.

To coordinate this work fully and to ensure that appropriate public reviews and comment period are held in the United States prior to the ISO acceptance of the Normative Addendum, a special kind of coordinating project must be established within X3. This project is called an "I" project, (which stands for International Development—concurrent National/International approval). There is one catch. Since the current ISO effort is an update to the existing ISO standard, the U.S. can only form an "I" project to consider the simultaneous adoption of the updates if the existing ANSI and ISO standards are completely identical, not "merely" technically equivalent.

This then leads us to the proposal to adopt the ISO standard document as the ANSI standard. The common standard allows us to have a synchronization project which in turn is the best way to ensure that the ISO and ANSI standards don't diverge.

The Concerns

Despite these reasons for forming an "I" project, and the corresponding acceptance of the change in standards documents from X3/159-1989 to ISO 9899:1990, there were still reservations by several members of X3J11 about moving to the new document.

The major points raised in the negative ballots on this proposal were:

1. Other standards currently reference X3/159-1989 and changing the standard will mean that all of these references will need to be updated.

2. There was a concern that the decision relinquishes too much control. The concern is that by taking this step the committee will be pressured to accept the future ISO changes to the current standard (in particular the Normative Addendum) even if they are not in the best interest of the U.S. C community.

3. There were concerns over who will take over control of the interpretations process for the standard document. Will the ISO committee assume full control over the interpretations? The underlying question is whether the ISO committee is in a position to make high-quality interpretations of the document.

4. There are concerns for the on-going changes in the standard. There is a sense that stability is needed more than further improvements.

As of the writing of this article, X3J11 had not completed its process of deciding how to respond to these issues. However, the following points address some of these issues.

The adoption of a new standard document has the same impact that any new revision of the standard would have. The old references remain valid; they would simply reference the previous C standard document. At this point, since the documents are technically equivalent, this should not be a major issue. If the Normative Addendum is accepted as part of the ANSI and ISO standards then the related standards bodies will need to decide whether they wish to reference the old standard or the new standard. As new editions of these related standards are published they will need to update the references appropriately.

It is true that there is considerable pressure to adopt international standards. There are signed international treaties that strongly encourage us to move in the direction of common international standards. (Common standards reduce the effective trade barriers between countries.) Despite this, the U.S. is not forced to adopt the ISO standard or any future changes, such as the Normative Addendum. We can accept all, part of, or none of the future changes WG14 makes to the C standard.

However, as a very active member of the ISO community, if the United States, and X3J11 in particular, feels that the ISO C standard is moving in directions counter to the best interests of the C language then we have a responsibility to make our concerns understood. In fact, this is exactly what has happened in the area of the proposed trigraph alternatives proposed by the Danish delegation to WG14. The U.S. has steadfastly opposed changes which were felt to be technically flawed. This may have, at times, made us look like "ugly Americans," but the committee genuinely felt that significant problems would arise if the original proposals were adopted. After several rounds of proposals and discussions, the U.S. outlined what it felt should be the guidelines for future proposals in this arena. The latest Danish proposal is consistent with these guidelines and will probably be supported and accepted by all member nations, including the U.S. The bottom line is that we are all better served by actively participating and ensuring that the Normative Addendum and the changes it recommends are acceptable to us rather than stepping back and rejecting whatever is produced because it does not meet the needs of the U.S. community.

The area of interpretations remains one of the least clear areas. It is in all of our best interests to have common ISO and ANSI responses to requests for interpretations. This avoids a divergence of the standards based on differing opinions of what the standards mean. The rest of the international community, on its own, probably does not have the resources or the history with the variety of technical issues which frequently surround many request for interpretation to do all of the interpretations. Some preliminary discussions between X3J11 and WG14 suggest that one way to meet all of the requirements would be to have X3J11 draft the responses for the interpretation requests and then have WG14 review and provide the approval of the interpretations for the ISO standard. If there are differences between the X3J11 recommendation and the response that WG14 can support, these would be worked out prior to the adoption of either an ANSI or ISO response. This proposal has not been adopted, but it

does seem to have considerable support, and it does make a lot of sense.

Even if this arrangement is worked out, the current understanding is that X3J11 remains responsible for interpretations of the ANSI standard. It remains a possibility, even if it is not a desired situation, to have the ANSI interpretation different from the ISO interpretation. Just as is the case with the on-going work on the Normative Addendum, it is up to X3J11 and its representatives on the ISO committee to ensure that WG14 clearly understands the rationale behind the U.S. positions prior to making any decisions on interpretations.

The question of stability is an area where there is no clear-cut answer. There is clearly a real advantage in having a stable environment so that implementors and programmers have a non-moving target to shoot at. The other side to this issue, however, is that we are just gaining the experience is several areas, most notably the support for programming in a multibyte character environment, to establish widely useful standards.

The changes that are being proposed in the Normative Addendum are almost entirely "bolt on" additions to the existing standard. They should not change the way the typical U.S. programmer writes code for the U.S. marketplace. On the other hand, if programmers are trying to serve the international community, then they would probably have to be concerned with the ISO standard, independent of what the ANSI standard requires. In addition, a common ANSI and ISO standard will allow U.S. compiler, interpreter, and other language-related product suppliers who are trying to sell in both the United States and around the world to offer a single product.

In the end, there are many issues that must be considered when trying to determine the most effective route to a C standard that serves the needs and interests of both the international and U.S. communities. These range from trying to ensure that the basic procedures are in place to ensure coordination and synchronization to issues of whether a common standard with the rest of the world is the best answer for a particular group. At this point the strong leaning is towards finding a way to make sure that the ISO and ANSI standards remain in sync as changes are made.

Other Standards News

There have been many requests over the past year for some formal publication of all of the interpretation requests and their responses. In response to this need, X3J11 requested that a project be authorized to produce a Technical Bulletin containing this information. The Project Proposal for a bulletin was approved in May of this year by a ballot of 35 in favor, 0 against, and one abstention.

There will be certain challenges in putting together this bulletin, particularly if the new ANSI standard document is adopted. For example, this will mean that all of the references to sections in the current standard will have to be supplemented with references in the new document.

X3J11 has not set a deadline for production of this bulletin, however. Given

the fact that the next meeting is not scheduled until December in Milan, Italy, it is probable that the publication date will be some time in 1992.

[Ed: The result of the X3J11 letter ballot regarding the responses to the No votes, was 23 in favor and one opposed. The wheels will now be set in motion for X3 to vote on this issue and, assuming they vote in the affirmative, for the ISO document to be adopted as the ANSI standard.]

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

∞

17. Miscellanea

compiled by **Rex Jaeschke**

Precompiled Headers

Early in 1991, Borland International released V2.0 of their popular C++ compiler which also incorporated an ANSI C compiler. One of the new features added was precompiled headers. The idea is that if the compiler could save a symbol table for a header, the header's source need not be reprocessed on subsequent `#include`s of that header.

The primary motivation for this feature appears to be to handle one very large header, `windows.h`. This header provides support for programming for Microsoft's Windows software. As supplied by Borland, this header contains approximately 120,000 bytes and spans nearly 3,500 source lines. As such, it takes far longer to process than do all the other headers combined. And if this header is processed multiple times in the same compilation, the time savings may be significant.

By default, precompiled headers are all stored in one big file. Then, when a program is compiled that contains `#include` directives for an already precompiled header, the compiler gets the corresponding symbol table information from the symbol-table file instead of reprocessing the header source. A number of compiler options and pragmas are provided to support this capability. There are also some restrictions placed on code so that the compiler can, in fact, take advantage of header precompilation.

Compiler Options

A number of command-line options are provided to handle precompilation. They are:

`-H` Instructs the compiler to generate and use precompiled headers.

`-H-` Turns off generation and use of precompiled headers. (This is the default.)

`-Hu` Instructs the compiler to use existing, but not generate new, precompiled headers.

`-H=`*filename* Specifies the name of the file to hold precompiled headers. (By default, the compiler uses TCDEF.SYM which is located in the same directory as the compiler executable.) This option subsumes the option `-H`.

Pragmas

Two pragmas are provided: `hdrfile` and `hdrstop`. The first has the following syntax:

```
#pragma hdrfile "filename.SYM"
```

It allows a symbol-table file to be explicitly named so the default can be overridden. This might come in handy where each project finds it useful to have their own symbol table independent of other projects on the same system.

The other pragma can be used to select some precompiled headers and some non-precompiled headers. Its syntax is:

```
#pragma hdrstop
```

Given a source fragment such as the following:

```
#include <stdio.h>
#include <stdlib.h>
#pragma hdrstop
#include <assert.h>
```

the compiler sees it should use precompiled versions of the first two headers but not those following the pragma. Perhaps the user is alternately defining and not defining `NDEBUG` during compilations and therefore needs to reprocess the source of `assert.h` each time to get the intended effect.

Restrictions

According to Borland's documentation, each file that intends to use precompiled headers must adhere to the following rules:

- It must have the same set of headers included *and* in the same order.
- It must have the same macros defined and to identical values.
- It must be written in the same source language (i.e., either one of C or C++, but not both).
- It must have actual header files with identical time stamps. (A header with the same name but with a different time stamp will be assumed to have changed and therefore need reprocessing from source.)
- In the case of several compiler options, all subsequent compiles must have identical settings for these options. These include the handling of spelling of external names, length of significance of identifiers, object alignment (particularly relevant for structure member offsets), and the signedness of plain `char`s.

To be accepted for precompilation in the first place, a header is not permitted to contain code except inside macros.

Some Comments

It is not clear to me that the actual savings are worth the effort. Borland claims that "Directly loading the symbol table from disk is over ten times faster than parsing the text of the header files." This may be true but if your total compilation time drops from 30 seconds to 5–10, that is not necessarily significant unless you are recompiling on a very regular basis. And with very large source files, the header processing phase will likely pale against the time spent on other compilation tasks.

Since `windows.h` is 5–10 times larger than the other headers provided, whatever savings are possible will come mostly when processing this header. However, to do serious programming for Windows you need an Intel 80386 processor which will probably have a clock speed of at least 16 MHz, if not 20. As such, the machine is quite fast anyway. However, if you were doing Windows programming on a smaller/slower machine or even non-Windows programming on an 80286 or 8088 system, savings on other headers may become more appreciable.

As for me, I come from the old school of designing and planning, when you had plenty of time to do so between once-a-day batch compilations. As such, I don't see the need to spend too much time worrying about compilation time. I'm much more concerned about execution speed. However, integrated and interactive development environments such as that provided with Borland C/C++ certainly encourage "try it and see" development with frequent recompilation.

Calendar of Events

- September 24–27, 1991 **Numerical C Extensions Group (NCEG) Meeting** – Location: At an Apple facility in Cupertino, California (Silicon Valley area). Note that this will *not* be a joint meeting with X3J11. As such, NCEG will meet more than the usual two days. For more information about NCEG, contact the Convenor Rex Jaeschke at (703) 860-0091 or *rex@aussie.com*, or Tom MacDonald at (612) 683-5818 or *tam@cray.com*.

- November 10–15, 1991 **Joint ISO C SC22/WG21 and X3J16 C++ Meeting** – Location: Dallas, Texas. For more information, contact the X3J16 Vice-Chair William M. (Mike) Miller, P.O. Box 366, Sudbury, MA 01776-0003, (508) 443-7433 or *wmmiller@hplabs.HP.com*.

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

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

- December 3–6, 1991 **X3H5: Parallel Model and Language Bindings** – Location: Livermore, California. For information, contact the Vice-Chair Walter G. Rudd at (503) 737-5553, Fax: (503) 737-3014, or *rudd@cs.orst.edu*.

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

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

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

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

- March 2–4, 1992 **X3H5: Parallel Model and Language Bindings** – Location: New Jersey.

- 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.

- May 13–15, 1992 **Joint ISO C SC22/WG14 and X3J11 Meeting** – Location: Salt Lake City, Utah.

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

News, Products, and Services

- **Clarification:** The previous issue of *The Journal* stated on page 87, "Sun has announced V1.1 of their Sun C compiler which is now standard-compliant."

  In actual fact, Sun C version 1.1 contains two compilers: cc – K&R C with Sun extensions, and acc – ANSI C.

  Sun ANSI C fully implements the ANSI C language definition as described in the ANSI X3.159-1989 document as well as the ISO 9899 document. There are, however, some incompatabilities with the portions of the standard which define the operating environment because SunOS 4.x isn't compliant with ANSI C (e.g., headers, libraries). These differences are documented in Chapter 9 of the Sun C Programmers Guide.

- **WATCOM** is shipping a Windows version of their C/386 compiler. Contact (519) 886-3700.

- **ACE** of Amsterdam have announced an MC68040 code generator for their language compilers inclusing C. Fax: (31) 20 750389.

- **Softools, Inc.** of Ellicott City, Maryland is shipping a family of cross-compilers for the HD64180, Z80, Z280, and 8075 systems. (301) 750-3733.

- **QTC** has announced two **numerical libraries** for DOS- and Mac-based C compilers. Contact them at (503) 626-3081.

- sml2c is a Standard ML to C compiler. It compiles an extended version of SML to portable C. The extensions include first-class continuations, asynchronous signal handling, separate compilation and a facility to freeze and restart programs. It is based on the Standard ML of New Jersey (SML/NJ) implementation (version 0.67) and shares its front-end and most of the runtime system.

  sml2c has been developed at the School of Computer Science, Carnegie Mellon University as a part of research into portable implementations of properly tail-recursive languages with first-class continuations. The design of sml2c has been described in the technical report *No Assembly Required: Compiling Standard ML to C* David Tarditi, Anurag Acharya, and Peter Lee, School of Computer Science, Carngie Mellon University. CMU-CS-90-187. For information contact *david.tarditi@cs.cmu.edu*.

∞