N117
X3J11/90-062

A European Representation for ISO C

Keld Simonsen

University of Copenhagen, Denmark

Bjarne Stroustrup

AT&T Bell Laboratories, USA.

ABSTRACT

The proposed ANSI C standard uses the American national character set, ASCII, for its representation, but as other national character sets use some characters differently, the ANSI C standard proposal includes a specification for an alternate representation, the *trigraph* representation. This paper proposes an extension to the trigraph proposal based on keywords and two-letter combinations of special characters, that is much easier to read and write. We hope this proposal will stimulate a discussion about this and related problems leading to a widely accepted solution.

1. The problems

Writing programs in C can be quite difficult where English is not the native language. The problem is that C uses symbols from the American national character set as operators and punctuation characters. This implies that you cannot use all letters from a national alphabet in identifiers and that programs appear strangely transformed on your screen or printer because some C operators are printed as the corresponding letters.

In many countries in Europe, South America and Africa the national character set standard is a version of the ISO 646-1983 international standard for 7-bit character sets. ASCII (ANSI X3.4-1968) is the national version of ISO 646 in the USA. It is the impression of the authors that ASCII and the national standard competes bravely from country to country for being the national industry standard. Also, on IBM mainframes there are national EBCDIC implementations very close in character repertoire to the formal national standards. In some areas, such as Scandinavia, the national character sets are predominant both in the 7-bit and the EBCDIC world. In other countries, such as the Netherlands, you mostly see ASCII. There are some new international standards based on 8-bit characters (the ISO 8859/1/2 Latin character sets and the ISO 6937/2 Videotex standard) which includes both ASCII and the various national characters, but it will take years, if not decades, for these new official standards to become industry standards and for 7-bit equipment to completely lose its importance.

It is quite desirable (and only polite) to allow programmers to write (and read) programs using their national 7-bit character set representation. The recent proposed ANSI C standard allows a representation of C programs in a national character set that does not have all of the characters used to represent a C program in ASCII. This is achieved by defining alternative representations, trigraphs, for the "offending" C operators and punctuation characters. However, there is still no provision for identifiers with national letters and the trigraph representation is not very suitable for human eyes.

2. ASCII and Trigraph Representations

The ISO 646-1983 standard leaves 12 positions undefined and up to national standards organizations to decide. For those undefined positions used in C for national ISO chars, the proposed ANSI C standard specifies a trigraph representation, a three-character replacement beginning with two question marks (??). Three characters ('$', '@' and '`') are not used in the proposed ANSI C (but they are used in some dialects of C) so we will also refrain from using them.

The 12 "problem" characters in ASCII and their equivalent ANSI C trigraph representation are:

```
ASCII     #    $    @    [    \    ]    ^    `    {    |    }    ~
Trigraph ??=              ??(  ??/  ??)  ??'       ??<  ??!  ??>  ??-
```

ISO646 reserves the characters # and $ so that they cannot be used for letters of a national alphabet. This ensures that they are not a problem in the context of C. The other 10 characters are "available for national or application oriented use."

Let us see how these things looks on a common example. First the ordinary C version in plain ASCII:

```
main(argc,argv) char* argv[];
{
        if (argc<1 || *argv[1]=='\0') return;
        printf("Hello,%s\n",argv[1]);
}
```

And this is how it looks in the Danish ISO 646 character set:

```
main(argc,argv) char* argvÆÅ;
æ
        if (argc<1 øø *argvÆ1Å=='Ø0') return;
        printf("Hello,%sØn",argvÆ1Å);
å
```

Experience shows that at least some people can learn to read and write this. In our opinion this is not a skill anyone should be encouraged to enquire. The begin-end brackets and the or operator look ugly and the array subscripts simply drown. It looks just as weird in, say, the Finnish, French, German, or Spanish ISO 646. This is the reason for the proposed ANSI C trigraph solution. How does our example look using trigraphs?

```
main(argc,argv) char* argv??<??>;
??)
        if (argc<1 ??!??! *argv??<1??>=='??/0') return;
        printf("Hello,%s??/n",argv??<1??>);
??(
```

Where are the array subscripts? Which operator is used in the conditional statement? We believe that even though the problem of defining an ISO 646 representation is solved by introducing trigraphs, the original problem of being able to write C programs without losing one's native language has not been attacked at all. The resulting "C programs" are unreadable and unwritable.

We see no alternative to using trigraphs for representing {, }, \, etc., in strings and character constants. For example:

```
switch (tok) {
case '{':    ...
case '}':    ...
case '\\':   ...
}
```

becomes

```
switch (tok) {
```

```
case '??<':  ...
case '??>':  ...
case '??/??/':    ...
}
```

Trigraphs are not pretty, but with the notable exception of backslash they will not be common in this context.

3. Our Proposal

To solve the original problem one could provide a combination of new keywords and reasonably nice-looking one- or two-letter symbols.

New keywords:

```
or          |
cor         ||      (conditional or)
and         &
cand        &&      (conditional and)
xor         ^
compl       ~       (complement)
```

Introducing new keywords is always a way to break existing programs, but the keywords could be conditionally in effect for new programs. The new keywords requires a few more keystrokes than the ASCII characters, but not many, and some would consider them to improve readability of C! The keywords `and` and `cand` were added for consistency – it seemed silly to have an `or` but no `and`. The `&` and `&&` operators are still valid. The new keywords (except `cor` and `cand`) can be combined with `=` to make assignment operators.

Note that use of trigraphs will not by itself avoid the problem of keywords. Macros will be used to make trigraphs more palatable. For example:

```
#define begin ??<
#define end ??>
#define or ??!
#define cor ??!??!
...
```

Naturally, groups of programmers will agree on standard versions of such macros and naturally not everyone will agree on the same versions. In a few years we will therefore be faced with the problem of having several incompatible sets of "de facto keywords" and conventions. The alternative is to pick a minimal set now. This is what this proposal does.

New symbols:

```
(:          {
:)          }
!(          [
)           ]
??/         \
```

The use of `??/` as the escape character is probably the least elegant of these alternative representations, but we see no acceptable alternative given that trigraphs are necessary in some contexts anyway. One might argue that an escape need not to be too pretty anyway.

We decided to make the compound statement brackets digraphs, finding 'begin' and 'end' too long to write and too likely to be found "not in the spirit of C" by large numbers of programmers. The digraphs `(/ /)` might have been preferable to `(:` and `:)` but current usage precludes this. For

example

```
/*
  argument type commented out waiting for ANSI C  compiler:
*/

int printf(/* const char*, ... */);
int fprintf(/* FILE*, const char*, ... */);
```

Using (: will cause a minor problem for C++ parsers because C++ has the prefix scope resolution operator ::. For example:

```
if (::open("myfile.c",0)==0) ...
```

This problem can be solved either by a bit of lookahead in a C++ lexical analyzer or by requiring C++ programmers to use a space before the :: operator in this relatively rare case.

The grammar of C precludes using (and) for subscripting (as is done for many other languages). Using keywords to represent [ ] would be unacceptably awkward. Using digraphs would be little better but we could notfind an acceptable pair. For example (/ ... /) suffers from the problem mentioned above, (! ... !) collides with the common usage ``if (!p) ...'' and (* ... *) collides with the common usage ``if(*p)...''. Parentheses are typically unnecessary for subscripting and ! should be considered an infix subscript operator (as in BCPL). The binding strength of the binary ! operator (subscripting) should be just above the unary operators and it should be left associative. For example, a!b.c!2*d means ((a!(b.c))!2)*d.

Our proposal would allow the program look like this:

```
main(argc,argv) char* argv!();
(:
        if (argc<1 cor *argv!(1)=='??/0') return;
        printf("Hello,%s??/n",argv!(1));
:)
```

Using ! as an infix operator, this can be further cleaned up:

```
main(int argc, char* argv!)
(:
        if (argc<1 cor *argv!1=='??/0') return;
        printf("Hello,%s??/n",argv!1);
:)
```

One would still need parentheses for more complicated subscripts; for example, v!(i+j). The symmetrical nature of subscripting in C will actually be more obvious. An apparant ambiguity would exist between ! used for subscripts and ! used as the negation operator. Consider:

```
int v[!4];
```

This would have to be represented as

```
int v !(!4);
```

since

```
int v!!4;
```

would mean

```
int v[][4];
```

Programs written this way still have a distinct C flavor. Locally, they can even be more terse than C and they are always shorter than programs written using the trigraph notation.

4. Practical Details

As observed, the trigraph proposal solves only a small part of the problem they were introduced to compensate for, whereas this proposal is an almost complete solution.

It is debatable whether it should be legal to mix the constructs proposed here, such as `(:`, `or`, and `compl`, with traditional C constructs, such as `{`, `|`, and `~`.

Mechanical translation from the standard American/English notation for C to this proposed "European" C notation is trivial. The potential ambiguity of translating [] into ! is handled by using parentheses systematically; `[expr]` becomes `!(expr)`. The reverse translation requires understanding of operator precedence to handle subscripting. Furthermore, if national characters are allowed in identifiers such characters must be expanded into sequences of English characters. This is no major problem.

5. Conclusions

We have found a *usable* solution to the international representation problem of C for most languages using a latin alphabet. The solution is quite easy to implement in a C compiler. We tried it.

The `(:`, `:)`, and `!` constructs do not affect existing programs, and the new keywords are chosen so that it is likely that only a minimal number of existing programs will be affected. Using them only in programs identified as non-US ISO C should remove completely the possibility of hurting existing programs.

If adopted this proposal will open the way for allowing national characters to be used in identifiers. This proposal is submitted for consideration to the ANSI C committee and the ISO committees dealing with C. It is also considered for adoption into C++.

6. Acknowledgements

Brian Kernighan and David Prosser found errors in previous versions of this proposal and suggested improvements to the presentation.

7. References

K&R: C
ANSI C draft
Richards: BCPL
BS: C++