Comments on the ANSI draft standard X3.???-1988 Programming Language C (dated 13-May-88). Issued for THIRD public review by the X3J11 committee

Submitted by the BSI C PANEL for Comment by X3J11.
19 August 1988

Respond to: Cornelia Boldyreff
C Panel Convenor
Department of Computer Science
Brunel University
Uxbridge, UB8 3PH
UK

NOTA BENE:

These comments are submitted to X3J11 with the intention of achieving resolution of issues as early as possible so that the resulting ANSI draft is acceptable at ISO level.

Section 1.7

The text on page 4 of the standard doesn't make clear if an implementation must provide at least one strictly conforming implementation, as well as conforming and non-conforming implementations. We would recommend that they should. We suggest that a couple of new sentences are added after line 25, thus:

> A conforming implementation shall provide an implementation that warns about all non strictly conforming features of a program. This provides programmers with the assurance that a program is maximally portable.

MAJOR POINT: Section 3.1.2.5

One of the changes made in the third draft is not mentioned as one of the 37 substantial points for review, but could be considered a more radical change than any of them. Alternatively, it could be considered as a serious ambiguity in the standard. This is that the type 'void *' has been altered almost to be another type with the same properties as 'char *', but has still many aspects of its old form.

The principal ambiguity is caused by the first paragraph on page 24 (section 3.1.2.5):

> "A pointer to void shall have the same representation and alignment requirements as a pointer to a character type. Other ...."

It is not clear whether this implies that 'char *' and 'void *' values may be used as the other one, without being put through an explicit cast (even though they are clearly different types). This affects argument passing, unions and several other constructions. For example, is the following program strictly conforming or not?

```
union {char *fred; void *joe;} bill;
char alf;
bill.fred = &alf;
printf("%p %p0, &alf, bill.joe);
```

Another effect of this change will be to encourage a particular extension, permitting programs that are explicitly forbidden by ANSI C, and this will seriously reduce program portability. The aspect that we are

referring to is that the following program is not conforming (by 3.1.2.5, 3.3.3.4 and 3.3.6):

```
typedef int fred[20];
void *joe = malloc(sizeof(fred));
void *bill = joe-sizeof(int);
int alf = 123;
for (i = 0; i < 20; ++i) memcpy(bill += sizeof(int), &alf, sizeof(int));
```

The use of this sort of construction considerably simplifies writing functions like qsort, and is very likely to be allowed by some compilers and used heavily by many programmers. It is undesirable to encourage 'almost universal' non-standard extensions, where they can be avoided.

We suggest:

1) that the sentence mentioned above (in the first paragraph on page 24, in section 3.1.2.5) is removed from the standard. This would restore the state of 'char *' and 'void *' possibly having different representations.

2) that the consequences of the changes to 'void *' in the third draft be considered further, for possible inconsistencies. It might be more consistent to define sizeof(void) to be 1, despite the fact that void is an incomplete type; this would permit the addition of integral values to 'void *' values, in the same way as for other pointers.

Section 3.3.5

The text on page 45 at line 25 should read:

> If either operand is negative, whether the result of the / operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation defined, as is the sign of the result of the % operator.

MAJOR POINT: Section 3.2.1.1

The problem is that section 3.2.1.1 states that objects of type char are widened to int ***wherever an int or unsigned int may be used***. This includes sizeof() context (as sizeof (1) is legal) and hence sizeof(char-variable) must be promoted to sizeof(int). done

Lest it is thought that this does not matter, consider:

```
char ch;
fread(&ch, sizeof(ch), 1, file).
```

It must be clearly stated in the draft that sizeof(char-variable) is always sizeof(char) and not sizeof(int).

Section 3.5.4.3

The text of the draft on page 66, line 18, would be much improved if some simple examples of a function prototype being used in conjunction with typedef and structures were added. For example, we suggest:

```
typedef double (*pfunc)(double A,double B);
typedef double (*pfunc2)(double,double);
pfunc a[3];
pfunc2 a2[3];
        /* 2 arrays of 3 function pointers returning double
        with 2 double arguments*/
struct sfunc{
        pfunc f1;
        pfunc2 f2;
        int (*f3)(char A,long B);
        int (*f4)(char,long);
};
        /* sfunc is a structure with f1,f2 being pointer to functions
        returning double with 2 double arguments, and  f3,f4
        being pointers to functions returning int with arguments of
        char and long. */
```

It is unclear whether the standard requires the type of definitions for pfunc and f3, or those for pfunc2 and f4. It may allow both. In particular this facility will enhance the use of pointers to functions which currently aren't typechecked. The use of the typedef facility makes them easier to use later on.

Section 3.5.6

There is an error on page 68, line 31. It doesn't make sense to allow the field t to be specified twice. Since we get the problem of which occurance is used by the compiler when t is referenced? We would expect the compiler to tell me of such name conflicts in a structure definition. We suggest line 31 should read: wrong

```
const :5;
```

Section 3.5.7

The footnote on page 69 isn't clear enough. Does this mean that any automatic variable can be assigned to variables only known at compile time? Or is it acceptable to only know them at execution time? We suggest appending a cross-reference to 3.6.2. in the footnote.

In addition the text on page 69 doesn't clarify when automatic variables won't be initialised, (nor does 3.6.2, p73-74.) What could be stated is that the only way a block can be entered other than at the top (and hence automatic initialisation is carried out), is when an explicit label is present, or when a nonsense value for an offset is added to a pointer (eg. an offset is added to a pointer to a function!). In other words, standard loop and conditional constructs can be safely used with initialising automatic variables. The following text should be added at line 25, page 69:

> If a compound statement does not contain any explicit label statements then automatic initialisation can be assumed to work unless nonsensical offsets are added to pointers. Implementations are not required to diagnose such nonsensical offsets.

It is unclear whether case labels can be covered by this.

The text on page 70, lines 5-7 requires clarification regarding the use of recursion. An example to clarify might help. Is this construct allowed, using 'struct as' as below?

```
struct bs{
        struct as A,B;
}b= {a,a};
```

Section 3.6.2

The text on page 74, lines 4-5 causes problems:

For example, what should happen to the following example of automatic initialisation?

```
struct as{
        struct as *pSelf;
        int Data;
}a={&a,5};/*a trivial circular list*/
```

This raises the problem as to when an object comes into scope and whether its initialisation can use objects only known at runtime.

We believe the final sentence should read:

> The initialisers of objects that have automatic storage duration are evaluated using run-time objects in scope and the values are stored in the objects in the order their declarators appear in the translation unit.

and the following sentence should be added to the standard:

> An object comes into scope from the moment its identifier is defined.

Section 3.8.3

This section only covers redefinition of object-like with object-like and function-like with function-like. The standard should clearly state that the name space of object-like macros and function-like macros is identical; thus, no other cases, are relevant. This section should also state what action is taken when the conditions for redefinition are **not** met; for example, as in the case, below:

done

```
#define a junk
#define a() precious
#define b junk
#define b rubbish
```

We would not find it acceptable for this to implicitly **undefined**.

Section 3.8.3.4

The standard should clearly state under which circumstances, function-like macros are marked as not subject to further expansion. We think the problem arises because when one puts (C_fn_name)() in one's C program source, C_fn_name, is the name of a C function; and this trick stops it being treated as a macro-defined function. I.e. (getchar)() is a real C function call; while getchar() may be a call of the getchar macro-like function.

Below is an example of the confusion caused by the wording in the current draft:

```
#define a() x
#define b ()
#define c a b
```

Does this expand to

```
a ()
or
x
```

The reasoning is as follows:
Case 1: a ()
When **c** is expanded, **a** is not expanded and **b** expands to (). On the rescan, **a** has been marked as non-expandable, and does not get expanded.

Case 2: x On the rescan, **a** () expands to x.

We think the problem arises because when one puts (C_fn_name)() in one's C program source, C_fn_name is the name of a C function; and this trick stops it being treated as a macro-defined function. I.e. (getchar)() is a real C function call; while getchar() may be a call of the getchar macro-like function.