Disciplined C (version 2.1h)

Yves L. Noyelle

École Supérieure d'Électricité, Plateau de Moulon, 91192 Gif/Yvette Cedex, France
E-Mail: Yves.Noyelle@supelec.fr

Abstract

Some proposals to render the C language a truly high level language are presented, as well as a program verifying that a given C program conforms to those proposals.

Introduction

Programming is, as every practitioner knows, a delicate art, where the main problem is not so much to obtain a "working" program (which is mandatory, of course), but to have designed it in such a way that it is not fragile, i.e. it can be modified/updated/debugged without breaking down. This means, for example, no (manual) duplication of code or "parallel" data (because, should a modification occur, the odds are high that the duplicate(s) will not be modified), or, more generally, that the programmer has dumped in the program the constraints set when he wrote it, so that they become apparent.

In order to attain these goals, programmers need tools. These tools should be as easy as possible to use, while not hampering creativity. "Easy to use" means that the tool is explainable (so its use is systematic), natural (it does not disturb common sense), and recognizes that human nature is fallible (so it protects "gently" the user against himself).

Among the tools that allow a programmer to express his ideas are of course the programming languages, and the compilers that go along, with their warning and error mechanisms.

One programming language much used these days is the (ANSI) C language, probably for the following reasons:

- it naturally supports modularity,
- being very close to the architecture of most current computers, it can express a wide range of applications; in fact, it is often called "the portable assembly language",
- it offers many representations for the integer type, which permits the best possible use of a given hardware,
- it offers a powerful macro mechanism, which can be used to prevent manual code duplication,
- the 'include' facility also works in the same direction, and allows some encapsulation of modules,
- the block concept makes it possible to minimize the scope of identifiers (which is always desirable, to prevent overflowing the reader's mind),
- it comes with a large set of predefined functions, allowing the user to easily implement exceptions, dynamic memory allocation, and so on; moreover, many system interfaces, such as X-Window's, are meant to be used via C,
- it is quite portable, along with its run time support; besides, the conditional compilation mechanism permits easy adaptation of programs to local conditions.

But the usual C compiler (or its fellow companion 'lint') lacks most of the devices that allow a program not to be too brittle (for example, to check that an array is of the same size as an enum used to name its elements); besides it is much too tolerant, even allowing things to work "by miracle" [MOD 91]. This is a pure disaster, because it gives programmers reasons to use obscure tricks, with the sole justification that "it works"! (on the local compiler, of course...).

C also is very tolerant, permitting programmers to fall into various well-concealed traps, such as: "`if (a = b)...`" (instead of "`if (a == b)...`"), "`a[i,j]`" (instead of "`a[i][j]`"), "`if (array1 == array2) ...`" (alas, it won't compare arrays !), or ... (long list).

But, on the positive side, this tolerance, well harnessed, can be turned into a great advantage; in other words, it is easy for C to be constrained into a disciplined language, with clean concepts, offering much syntatic and static semantic guidance to the programmer, and inciting him to dump in the code, rather naturally, the constraints he chose to obey while writing it, all of which without loosing any efficiency. The purpose of this paper is to propose such a "harness".

This harness is mainly designed to allow strong typing (especially between compilation units), well defined types, name equivalence for all types, reasonable automatic conversion rules, easy to use arrays (even dynamically allocated ones), limited use of casts, enhanced portability, and removal of most traps... It has been implemented as a C checker program.

The following gets into greater detail; it assumes a good knowledge of C, but, beware, may come as a shock to seasoned C programmers.

Areas where C can be disciplined

<u>Typing</u>

C offers a wealth of primitive types, plus very good means to create new types. But it lacks the boolean type, and more generally allows a complete mixture of all scalar types (including characters and pointers !), with little concern for information loss. Also, the type equivalence mechanism for arithmetic types is structural equivalence, which does not make it possible to enforce the distinction between "information" type and "representation" type (see later, "parallel types").

To remedy this, Disciplined C sets the following:

- 'char' (<u>without signedness specifier</u>) is a specific character type, whose constants are the character constants of C; this type is a closed type, not an arithmetic type, which means that the (in)famous idiom:

```
int c;  /* notice the 'int' ! */
while (( c = getchar() ) !=  EOF) {...}
```

is to be replaced by:

```
char c;
while (c = getc(stdin), !feof(stdin)) {...}
```

A 'char' constant must contain just one character,

- integral types are: (un)signed char, ((un)signed) short, ((un)signed) int, ((un)signed) long; combinations of 'signed' and 'unsigned' varieties that could lead to information loss or "unexpected" results ([K&R 88] p. 198) are pointed out. Besides, an attempt to compare differences of 'unsigned' via relational operators is flagged, because the underlying C compiler will generate an unsigned branch in such a case, fact which may not be obvious to all programmers. Consider a graphic application where point coordinates are represented as unsigned; then `xPoint2 - xPoint1` is still an unsigned (from the underlying C compiler point of view), which means that `xPoint2 - xPoint1 > -1` will always be false ! It is strongly recommended to '#define' a "byte/ubyte" synonym for 'signed/unsigned char',
- floating types: no change,
- a boolean type is introduced, that is to be the type of the (first) argument(s) of 'if', 'while', '?:', '||', '&&' or '!' operators, and the type of the second argument of a 'for' operator. It has to be defined by the following:

  ```
  typedef unsigned int bool;
  ```

  Relational and equality operators, as well as '||', '&&' and '!', yield a result of type 'bool'. '&', '|', '^' operators yield a result that can be used generally as 'bool', but , since its value is not guaranteed to be in the [0, 1] interval, a check is performed to warn in case of possible strange result (e.g.: `if ((feof (stdin) & fclose(stdin)) == TRUE)`).

  The constants 'TRUE' and 'FALSE' (which may carry any other name suitable to the programmer, such as VRAI and FAUX) are defined via constant boolean expressions, for example:

  ```
  # define TRUE  (0==0)
  # define FALSE (0!=0)
  ```

  Besides eliminating bad programming practices, such as (excerpt from 'strcpy'):

  `while (*s++ = *t++);`
  (instead of `while ((*s++ = *t++) != '\0');`)

  a side effect of the introduction of this type is that the standard error:

  `if (y = 0)...` (instead of `if (y == 0)...`)

  is flagged by a "Boolean expected" warning (warning also for:

  `if (setOfBits & mask == Msk1)...` , instead of
  `if ((setOfBits & mask) == Msk1)...`)

  Explicit comparison to zero is less cryptic, and entails no loss of efficiency, the compiler testing against 0 anyway,
- 'enum' types are closed, i. e. they are not mixable between them, nor with arithmetic types (in fact, the 'char' type seen previously is considered as an enum type). However, some amount of mixing is allowed: a (signed) int can be added to or subtracted from an enum, yielding an enum of the same kind. This is because the notion of distance between two enum or char is often useful (for example, '9' - '0'), so subtracting two enum (of the same kind) is allowed, and yields an 'int'.

  Enum constants can be initialized by signed 'int'. The constants of a given enum must have differing values, except if the /*~SameValue*/ d-pragma is used ("d-pragmas" will be described later).

  The only operations allowed on enums, besides comparison, assignment and addition/subtraction of distance, are '&', '|', '^', '~', '>>', '<<'; so enums can be used as sets of bits, but not (in the absence of a /*~TypeCombination*/ d-pragma telling the contrary) as arithmetic quantities,
- a major innovation of Disciplined C is the notion of "parallel type", that allows a distinction between information type and representation type. The following:

  ```
  typedef  int  Tindex, Tval;
  typedef  Tindex  Trow, Tcol;
  ```

  creates four <u>distinct</u> types, but which all accept the same operations and the same constants as the "representation" type ('int' here). Tindex, Tval, Trow and Tcol are examples of "information" types, because they convey an idea of the semantics of the corresponding objects. For example, they may be put to use in a checkers playing program: Tval will name 'int's that represent values of checkers, Trow and Tcol, 'int's that represent row and column indexes, Tindex, generic type for indexes.

  Tindex, Tval, Trow and Tcol are called parallel types; in fact, a type T is said to be <u>parallel</u> to a type xxx iff it is defined by a "typedef xxx T", with xxx parsing to a naked 'baseType' (no qualifier nor modifier; see grammar in Appendix A). In other words, T must be a strict synonym of baseType.

  This "parallel" relation is transitive but antisymmetrical. Other typedefs do not introduce parallel types; they just name qualified/modified variations of the baseType.

  The <u>representation type</u> of a parallel type is the possibly qualified or modified nativeType associated, using traditional C rules, to its type identifier.

  Cascaded synonymous typedefs create a hierarchy of parallel types, hierarchy used to set a compatibility rule, and to find the result type of an operator. Let us define that a type T1 is "higher" than a parallel type T2 if T1 is T2 or any ancestor of T2 (including the representation type), that is, T1 has been used in the chain of typedefs needed to define T2 from its representation type. For example, Tindex is higher than Trow, but not than Tval, and Tval is not higher than Tindex or Trow. If a parallel type meets a non parallel type compatible with its representation type, the wider representation type is the higher type (unsigned varieties are considered wider than their signed counterpart; numeric constants are supposed to have the narrowest representation type possible, in the same variety if they are signed or U-suffixed).

  Then the compatibility rule is the following: an operator (except shift operators, whose operands are deconnected) can only combine operands such that the type of one operand is higher than all other operand types; for operators other than relational and equality operators, the result type is this higher type. For assignments, the higher type has to be the type of the left operand. For functions, see below ('Casts') the /*~ResultType*/ d-pragma.

Since these rules do not facilitate mixing of types, the /*~TypeCombination*/ d-pragma tells the allowed combinations of types (e.g. Tohm * Tamp -> Tvolt). Besides, there is a notion of 'coefficient', for multiplication, division, modulo, whereby if the coefficient is of representation type, then the result type is the type of the other operand. Also, a constant of representation type meeting a parallel type behaves as if it was of that type.

The special rules that apply to enums also apply to their descendants.

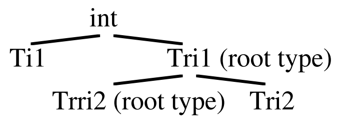
The /*~RootType*/ d-pragma isolates the so qualified type from its ancestors (renders invisible the subtree headed by that type); also, the constants of a such "root" type are compatible with all its (visible) descendants.

Let us consider the following example:

```
typedef int Ti1, Tri1 /*~RootType*/;
typedef Tri1 Trri2 /*~RootType*/, Tri2;
Ti1 i1; Tri1 ri1; Tri2 ri2; Trri2 rri2;

i1 = 0;              /* OK */
ri1 = 0;             /* Wrong */
ri1 = (Tri1)0;       /* OK */
ri2 = 0;             /* Wrong */
ri2 += (Tri1)1;      /* OK (generic cst) */
ri2 = ri1;           /* Wrong */
ri1 = ri2;           /* OK */
rri2 = (Tri1)0;      /* Wrong */
rri2 = (Trri2)0;     /* OK */
ri1 = (Trri2)0;      /* Wrong */
ri1 = (Tri2)0;       /* OK */
```

(the created hierarchy is :



)

To ease up things, 'int' constants can be added/ subtracted uncasted to/from (numeric) root type constants.

As can be seen, the /*~RootType*/ d-pragma allows a (leaf) type to be closed (not to accept constants of any of its ascendants),

In short, the parallel type facility enables name equivalence instead of structural equivalence for any type in C, and so fosters programmers to give differing names to the types of their different object classes.

<u>Arrays</u>

Arrays are a common data structure, easily understood by most programmers because of their mathematical background.

But in C, arrays, especially dynamically defined arrays (by way of 'malloc'), are very difficult to use without coming across the notion of pointers, an awful prospect for many naive users. So, with the help of a set of macros (predefined in a header file called "dynarray.h"), Disciplined C makes it possible to use any array (static or dynamic) without ever having to use pointers (cf. Appendix B).

On the other hand, the array concept being natural, a pointer can always be used as an array name.

There is no automatic conversion from an array to a pointer to its first element (except for string literals, and for function parameter passing: simulation of call "by reference"), and one has to explicitly use the construct "&array[i]" to get a pointer on the i$^{\text{th}}$ element. One consequence is that the construction `if (array1 == array2) ...` is not accepted, so programmers will not believe that arrays can be compared.

Another feature of Disciplined C is that, at each use of an array, the type of the index expression (any integral, enum or bool type) is checked against the type of the bound; this last type can be specified via the /*~IndexType <type>*/ d-pragma (which overrides the type of the bound-giving expression, if present). This d-pragma may also be used for pointers, to cater to the case where they are used as dynamic array names.

The value of a constant index expression is checked to be positive and less than the bound (except if indexing pointer).

<u>Casts</u>

To encourage programmers to choose the right types from the outset, and to enhance program portability with respect to, for example, alignment problems, the use of casts is severely monitored:

- any arithmetic/enum/bool type can be cast to any other arithmetic/enum/bool type, but an overflow check is performed on constant expressions,
- pointers: they can only be cast to other pointers; a non 'void *' pointer cannot be cast to a pointer on higher alignment-requiring type, or to a pointer on type whose internal layout is machine dependent; only the constant 0 may be cast to a pointer. Conversion from a 'void *' pointer (except NULL) to any other pointer has to be documented either by a cast or by using the /*~VoidToOther*/ d-pragma, this for example to prevent the following construct:

  ```
  void *pv; struct{...} *pst; float *pfl;
   ... ; pst= pv = pfl; ...
  ```

  from going unnoticed. A d-pragma, /*~ResultType*/, eases the situation for generic "modifier" functions, such as 'memcpy' or 'realloc', by specifying that the result type of a given call to such a function is the type of the current actual parameter corresponding to the /*~ResultType*/ qualified formal parameter, e.g.:

  ```
  void *realloc(void *p /*~ResultType*/,
                size_t n);  /* declaration */

  realloc(ptr, exp)   /* type of this
           'realloc' call = type of 'ptr' */,
  ```

  The type of the /*~ResultType*/ formal parameter must be the same that (or a descendant of) the return type of the function.

  For generic "creator" functions, such as 'malloc', see below the /*~Generic*/ d-pragma ,
- arrays: they can't be cast to anything (meaningless),
- parallel types: they may be cast freely into each other (but the /*~CastTo*/ d-pragma is rather to be used in that case).

Casts not conforming to these rules, or to a lower qualified pointer type, elicit a warning, which can be avoided by using the /∗~OddCast∗/ or the /∗~PortableQM∗/ d-pragma.

Since the number of casts in a program can be put to use in a qualimetry tool, the needless use of them (or of /∗~OddCast∗/, /∗~PortableQM∗/ d-pragmas) is also flagged.

Functions

Non-void functions should normally return named types (problem-related names), and so make use of the parallel types facility. Yet some functions are 'utilities' functions, that is there is no meaningful name for their returned type (e.g. 'strcmp'). The /∗~Utility∗/ d-pragma signals this.

Also, the parallel type mechanism uses inheritance, and sometimes subtyping is needed. So the /∗~Generic∗/ d-pragma tells that the result of a such qualified function is compatible with any visible (cf /∗~RootType∗/) descendant of its return type. For a 'void ∗' returning function, this d-pragma tells that its result is compatible with any pointer.

Functions formal pointer parameters that could be qualified 'const' are signaled. Besides, the /∗~MayModify*/ d-pragma tells that, although its formal parameters are marked const, a function may modify its environment through them, either via casted parameter or through pointer embedded in struct/unions.

The /∗~ResultPtr*/ d-pragma tells that a returned pointer is the same as the so-qualified parameter, this to be able to propagate the 'const' checking (e.g. in 'strchr').

The /∗~SizeOfMemBlk*/ d-pragma is intended to be used with memory allocating functions, such as 'malloc', to allow verification that the argument of the possible sizeof used is of the same type as the receiving pointer pointed type.

Others d-pragmas related to functions are /∗~ResultType∗/ (cf 'Casts'), /∗~PseudoVoid∗/ and /∗~NeverReturns∗/ (see later, 'Miscelleanous'); function parameters may also be marked /∗~NotUsed∗/ as well as /∗~Utility∗/ (do not accept an actual parameter whose type is a parallel type).

As an extension, several formal parameters can be qualified by /∗~ResultType∗/; then the type of one of the corresponding actual parameters must be higher than all other corresponding actual parameters' types; that type is the result type of the function call.

Compilation units

One of the plagues of C is the lack of type-checking between formal and actual parameters for external functions, or more generally, between definition and uses of external objets (we will, for the following discussion, call "global" a block-level-0 object/function visible in only one compilation unit, and "external" an object/function visible from several compilation units).

Another problem is that the encapsulation of C modules is usually very bad, because one of the tenets of encapsulation is not respected: "anything that is not explicitly made visible must be hidden". In C, the rule is: "anything (at block level 0) not explicitly hidden is visible"; this is because the 'static' keyword has to be explicitly used, instead of being the default option, and so is "often" (the word is weak) forgotten.

Something also confusing to many programmers is the difference between declarations and definitions, and the uniqueness of definitions. This is because, for C, some declarations are also definitions ("tentative definitions"); also, a number of compilers/linkers allow several definitions for the same external object.

Disciplined C solves these problems in the following way: any object/function/type identifier has to be declared (just once) before use; any object/function defined at block level 0 as not 'static' has to be declared in a header file; there must be one "header" file for each "body" file defining external objects/functions.

A header file is constrained to contain only (besides type and macro definitions) external declarations, which must make use of the 'extern' keyword. A definition must not make use of this keyword.

This provides the following benefits:

- programmers are warned if they have forgotten the 'static' keyword,
- for really external objects/functions, since they must be declared in (common) header file(s), type checking between compilation units is secured,
- an external object definition (and possible initialization) is easier to find: it can only be in the corresponding "body" file),
- the difference between declaration and definition is made clearer.

To cater for module composition, the restriction that there be only one declaration for an external object/function is relaxed; but, in a given header file, only one declaration for a given object/ function is allowed; the possible constraints (qualifiers, array size etc) have to keep constant or increasing in order of inclusion of header files.

Finally, the scope of any global object/type/tag can be terminated before the end of the compilation unit by use of the /∗~Undef(Tag) <identList> ∗/ d-pragma.

Inclusion of header files

Header files are often included at the wrong level. They, most of the time, should be included at least at the body file level (and at the beginning of it), so that all used services are easy to spot. If they are not included at that level (and one of the functions/objects they declare is used), a warning is issued.

To cater to "composed" header files (header files offering services including other service(s)), the /∗~ComposingHdr∗/ d-pragma is supplied. For example, if one wants to provide a <superMath.h> service, giving all services of <math.h> plus some others such as Bessel and Jacobi functions, one will write the following <superMath.h> header file :

```
#include <math.h> /*~ComposingHdr*/

extern double bessel( ... );
extern double jacobi( ... );
```

and then no warning will be incurred in a body file including only <superMath.h>, and yet using 'sin' (or any other <math.h> function/object).

Encapsulation

To still improve encapsulation, structures/unions/enums declared in header files may be qualified by the /∗~PrivateTo <files list>∗/ d-pragma, which renders their member's names invisible, except from macros/

functions defined in the indicated files. So a type may be exported without its components being disclosed.

D-Pragmas

As seen previously, Disciplined C often needs advice or information, to be conveyed by pragmas. But it cannot use the pragma facility of C, for two reasons:

- there might be an ambiguity with an already existing local pragma, or a compiler might warn about an unknown pragma,
- more significantly, C pragmas can only be at the beginning of a (logical) line.

For those reasons, it was decided to define "d-pragmas", and to make them look as comments, hoping that the '/*~' prefix does not happen too often in existing programs.

Miscelleanous

Many other improvements of C have been incorporated in Disciplined C (and are verified by the checker program):

- 'if', 'else', loop operators: if they are followed by several statements on the same (physical) line, it is asked whether all these statements are part of the 'if'/'else' arm, or loop body,
- as an option, line indentation is checked against current block level; this makes it possible to detect early missing left/right braces, or bad 'if'/loop bodies,
- declarations have to be separated from statements by white line(s), except if the first statement is empty (";;"),
- only 'void' type expressions can be used as statements (or as first argument of the comma operator, or first and third arguments of the 'for' operator), the only exceptions being the following:
  - expression whose top operator is an assignment or increment operator,
  - functions marked as /*~PseudoVoid*/ (only functions whose main effect is a side effect, such as 'printf' or 'strcpy', should be so marked).

  This way, probable errors such as "`fct;`" (instead of "`fct();`"), or "`a[i,j]`" (instead of "`a[i][j]`") are located. Also, it encourages programmers to test the value returned by I/O functions ('scanf', 'fputc', etc), so as to detect I/O errors,
- a non-void function must terminate in all cases either via a 'return exp', whose type is (a descendant of) the type of the function, or via a call to a function marked as /*~NeverReturns*/ (such as 'exit' or 'abort'); to that end, a simple control flow analysis is performed, which also detects unreachable statements,
- since Disciplined C is only aimed at ANSI C with "new-style" functions, a function with no parameter can be declared/defined as f( ) (no 'void' keyword), and still be considered only as a parameterless function,
- if a parameter name is given in a function prototype, the same name must be used for the corresponding parameter in the function definition, this to ensure that the meaning of the prototype is/stays the same as the implementation,
- a function name is not a pointer on that function, and pointers on functions are to be dereferenced before use (for the sake of regularity),
- switches: the 'switch' statement must control a block; a missing break is flagged, unless a /*~NoBreak*/ d-

pragma has been used; a 'default' case (which must come at the end of the switch) is expected, unless a /*~NoDefault*/ d-pragma has been used or, if the switch expression is of enum type, all enum constants of that type have been used as case values; the /*~FullEnum*/ d-pragma can be used to get a warning if not all (distinct-valued) constants of a given enum are used as cases values, and a default case (to catch, for example, spurious values) has been used,

- in an effort to clarify the distinction between type attribute ('extern'... 'register') and type qualifier ('const', 'volatile'), the qualifier must lexically come after the attribute,
- 'const'/'volatile' qualifiers are strictly obeyed (and literal strings considered as const char [ ] !),
- there is a warning if objects modified within the reach of a setjmp( )/ longjmp( ) pair are not qualified 'volatile',
- pointers on local objects cannot be returned or assigned to global/external variables, unless the /*~LocalAdr*/ d-pragma is used (this check is not perfectly foolproof, however, because of the possibility of pointers on pointers),
- except inside array and enum initialization, numeric constants (barring -1, 0, 1) must be named ('#define'd); this forces much semantic to flow from the programmer's mind to the program ! There is a special case for array bound expressions, where unnamed constants cause warning only if (subsequently) a non constant expression is used to index the array.

  Since this constraint sometimes proves clumsy, the following alleviates it:

  • parenthesized unnamed constants can be used inside macros,

  • if a (numeric) parallel type is qualified by the /*~LiteralCst*/ d-pragma, use of unnamed constants inside expressions of this type does not elicit warnings,
- underflows/overflows in constant expressions are detected,
- in the 'scanf'/'printf' family, argument types are checked against (constant) format string specifications,
- unclosed comments are flagged (detection of '/*' inside a comment),
- labels and tags must obey the rule of other identifiers, that is disappear outside the defining block or structure/union; but, to stay compatible with C, the same label cannot be defined in different blocks of the same function body,
- backward branches, which may cause unstructured loops, must be documented via the /*~BackBranch*/ d-pragma,
- unless the /*~DynInit*/ d-pragma is used, non-static structure/array initializations are flagged (because they slow down function entry and waste memory),
- external identifiers are checked for non ambiguity for the local linker,
- tests for an unsigned quantity to be negative are flagged,
- constant boolean expressions (used elsewhere than in an assignment, and outside macros) are flagged, because they probably signal a coding error,
- parenthesization problems with macro bodies are detected (for example `Diff(a,b+c)*d`, with `Diff(x,y)` defined as "`x-y`" and not "`((x)-(y))`"),
- side effects via macro parameters used more than once are flagged,
- unless qualified by the /*~NotUsed*/ d-pragma, unused identifiers/objects are signaled,
- uninitialized local objects are flagged,
- unused variable values are signalled (an only modified object, e.g. `i++ ,` is not considered as used),
- an attempt to detect potentially dangerous side effects has been unsuccessful; for, if it is easy to detect "`a[i++] = i`", it is harder to detect "`a[i++] = *(&i)`", and much harder to detect "`a = f() + g()`", where each (external) function f and g depends on side effects of the other function.

Compile-time checking tool

A d-pragma, /*~zif <boolExp> "<msgString>"*/ is provided, that causes emission of <msgString> if <boolExp> is true. It is the main tool (besides "information" types and the fact that integer constants must be named) that allows a programmer to indicate his constraints (for example, that a quantity should not exceed a certain value, or that a structure member should be placed at such a position). <boolExp> must be a constant expression, but can contain 'sizeof' operators, enum constants, casts, and also the following functions:

- _ _member(<membName>): only to be used during structure/union initialization; answers true if currently initializing the member whose name is <membName>, or at end of the structure/union (empty <membName>),
- _ _extent(<enumType>): answers the "extent" of the enumType, i.e. the distance between its greatest enum constant and its smallest one (type: 'int'),
- _ _index( ): only to be used inside array initialization; answers the index value of the current array element being initialized (type = type of the bound),
- _ _sametype(x, y): 'x'/'y' may be types or expressions; answers true if both (expression) types are the same. The type equivalence used is the same as for parameter passing (x: formal parameter, y: actual parameter). This function can for example be used to type macro parameters.

The 'defined' function is also accepted. Of course, undefined identifiers are flagged (not replaced by 0L !).

This d-pragma permits a program to be much less fragile, by allowing one to build into it mechanisms to check at compile-time that related data structures are kept coherent through modifications; for example, it is easy to check that the length of parallel arrays are the same, or that the length of an array is the same as the extent (+1) of an enum used to name its elements, or that a structure member/array element is initialized with the right value.

Some words about the Disciplined C checker

"dcc", the C-checker, performs full syntactic and some static semantic analysis of a C program, one compilation unit at a time. It is itself written in Disciplined C, is approximately 15000 lines long, uses a recursive descent method, and is about ten times faster that the 'gcc' C compiler (on a DECstation 5000 running ULTRIX); so error detection is quite swift, and only a small time cost is added to regular compilation if the program is correct.

Unless the '-zcc' option has been used, control is automatically transfered to the local compiler if no error is detected (the philosophy of dcc, as of any C compiler, is to issue errors only when the standard is violated).

The '+zsy' option permits one to have, at each block exit, the content of the symbol table for those identifiers local to the block; their kind, type (fully decoded) and size (for objects) are given, this last information so that programmers have an idea of the amount of memory their program uses.

Numerous other options are available, for example to limit dcc's scrutiny.

All error/warning messages can be adapted at will, being gathered in one single source file. In addition to a message, one is given: the number of the source line where the error/warning has been detected, the name of the corresponding file, the preceding and current source line text, and a caret showing the precise position of the error/warning in the source line. An option causes the printing in clear of the last processed tokens (sometimes very handy !).

Three levels of warnings, corresponding to benign (mostly readibility), serious, quasi-certain error, are implemented; a screening mechanism (desactivated by some options, e.g. '-zcc') prevents a warning from showing up if its level is lower that the last message level. In the same vein, unless option '+zae' is used, messages are not repeated if they pertain to the same cause.

There is a way to adjust local system header files (without modifying them), mainly in order to change some library functions return types (such as 'getc' or 'isalpha'), to define NULL as `(void *)0`, or to mark functions as /*~PseudoVoid*/ or /*~NeverReturns*/.

A symbol, '_ _dcc', is defined by the checker; it can be used to turn off its scrutiny in selected areas of source code (besides the /*~NoWarn*/ and /*~Warn*/ d-pragmas).

This program, developed on a VAX/VMS machine, has been ported in less than a day on a HP715/UNIX, a DECstation 5000/ULTRIX and an ALPHA 3000/OSF1.

It is available via anonymous ftp at 'ftp.supelec.fr', in the sub-directory 'pub/lang/dcc'.

Conclusion

The ambition of Disciplined C is to be a really high-level programming language, with all the confidence and ease of use the term "high-level" should convey.

The first idea behind it is that programmers ought to be given a tool permitting to render the semantics and constraints of their programs more apparent, while not hampering or restraining their creativity, nor inducing any run-time loss of efficiency.

Another idea is that a tool should serve its user, instead of randomly disseminating traps in his way; so everything that seems "strange" is flagged. But there are ways (via d-pragmas) to tell that a generally erroneous situation is perfectly valid in this specific case.

A third idea is that a tool should exhibit clear concepts, so that the user can easily master it; the last idea is that error messages should be indicative enough to put programmers back on the right track, and the less numerous possible (don't detect twice the same error; try to avoid induced errors...).

Those ideas have been in the mind of the author for a long time [NOY 88], but it took him about ten years to realize that, instead of regularly complaining about C weaknesses and pitfalls, it would be better (and possible) to write a program that reports most of the problems, and so, last but not least, facilitate the teaching of C with due regard to good programming practices and conceptual purity.

The C standard library is another place where some improvements could be brought: for example, the string handling function 'strcpy' could return a pointer on the ending NUL character, and give an easy way to prevent buffer overflow; or there could exist a function telling whether a pointer points on a 'free'able area. But this is unfortunately out of the reach of Disciplined C.

I am much indebted to my colleagues of our computer science department, with whom discussions were often illuminating (and animated !). Special thanks to C. Bocage, F. Boulanger, D. Marcadet and F. Mullet for their advice and patience. Also to W. Briscoe (freelance, UK), for numerous comments/advices.

References

[K&R 88] B.W. Kernighan and D.M. Ritchie
The C Programming Language
Prentice Hall, Englewood Cliffs, N.J., 1988

[MOD 91] R.P. Mody
C In Education and Software Engineering
SIGCSE Bulletin, Vol 23 n° 3, September 1991, pp 45-56

[NOY 88] Y.L. Noyelle
La Saga du LSE (et de ses cousins LSD/LSG/LST)
Colloque sur l'histoire de l'Informatique en France, Vol 2, May 1988, pp 301-310

APPENDIX A

Disciplined C grammar (LL(2))

```
prog ::=           decl+
decl ::=           attrib? decl1
attrib ::=         extern | static | auto | register | typedef   -- auto/register allowed only inside function.
decl1 ::=          decl2 decl3  -- 'modifier' (called by 'decl2') must declare an identifier; exceptions: 'attrib' ≠ typedef,
                                   and bit-field (if no identifier => padding), or 'strun'/'enum' with both 'tag' and
                                   'member's/'enumElt's (in such a case, 'modifier' must be empty).
decl2 ::=          qualif? baseType qualif? modifier      -- only one qualif.
qualif ::=         [ const | volatile ]+     -- one of each at most; volatile incompatible with register.
baseType ::=       nativeType? |             -- may only be omitted in case of (field) padding.
                   ident                     -- "type" identifier.
nativeType ::=     void |       -- only if function or pointer declaration; then 'qualif' must be empty.
                   float | [ long ]? double |
                   [ unsigned | signed ]? [ char | short [ int ]? | int | long [ int ]? ] |
                   strun |
                   enum
strun ::=          [ struct | union ] tag? [ { member+ } ]?        -- error if no 'tag' nor 'member's, or if one of them
                                          exist, but the following 'modifier'(s) do not each one define an identifier;
                                          if no identifier definition, error if 'attrib' or 'qualif' non empty. Exception:
                                          a 'strun' with a tag only is legal.
member ::=         decl1        -- 'member' must not be a function.
enum ::=           enum tag? [ { enumElt [ , enumElt ]* } ]?    -- error if no 'tag' nor 'enumElt's; see also 'strun'.
tag ::=            ident
enumElt ::=        ident [ = cstExp ]?
modifier ::=       pointer* modif1 arrayFct*
pointer ::=        * qualif?            -- pointer declaration.
modif1 ::=         ( modifier ) |
                   ident?               -- error if 'ident' omitted and {'attrib' = typedef or 'baseType' ≠ 'enum', 'strun'}.
arrayFct ::=       [ cstExp? ] |        -- array declaration.
                   ( parList? )         -- function declaration; 'attrib' ≠ auto/register; return type not array nor function.
parList ::=        attrib? declPar [ , attrib? declPar ]* [ , ... ]?  -- only legal 'attrib' = register. If parameter type =
                                     void, parameter must not be named, nor 'attrib'uted, and must be alone; other-
                                     wise, parameter names must all be different. If 'parList' called by a 'modifier'
                                     followed by 'block', then each 'declPar' must declare an identifier.
declPar ::=        decl2        -- function type not allowed.
decl3 ::=          initOrSizFld? [ , modifier initOrSizFld? ]* ; |   -- no 'initOrSizFld' if 'attrib' = typedef , or if no
                                          identifier declared, or if function declared, or if called via 'member'.
                   block        -- 'modifier' (called by the 'decl1' that called this 'decl3') must have declared a function;
                                                                              'attrib' = extern or static only.
initOrSizFld ::=   = init |
                   : cstExp     -- allowed only if called via 'member'.
init ::=           condExp |    -- 'condExp' allowed only if 'qualif' = auto/register, and 'init' not called by itself.
                   cstExp |
                   { init [ , init ]* [ , ]? }
block ::=          { decl* stmt* }       -- no function definition allowed among 'decl's.
stmt ::=           label : stmt |
                   block | ifStmt | switchStmt | returnStmt | whileLoop | doLoop | forLoop | break ; | continue ; |
                   goto label ; | sideEffects ; | empty ;
label ::=          ident
ifStmt ::=         if ( boolExp ) stmt [ else stmt ]?
switchStmt ::=     switch ( exp ) { decl* [ [ case cstExp : ]+ stmt+ ]+ [ default : stmt+ ]? }
returnStmt ::=     return [ exp ]? ;
whileLoop ::=      while ( boolExp ) stmt
doLoop ::=         do stmt while ( boolExp ) ;
forLoop ::=        for ( sideEffects? ; boolExp? ; sideEffects? ) stmt
sideEffects ::=    [ sideEffect , ]* sideEffect
```

```
sideEffect ::=    asgnExp      -- warning if top operator of 'asgnExp' is not one of 'incDec', 'asgnOp', call to a
                                   function returning void, or cast to void.
boolExp ::=       exp          -- of boolean type.
cstExp ::=        condExp      -- computable at compilation time.
exp ::=           [ sideEffect , ]* asgnExp
asgnExp ::=       condExp |
                  condExp asgnOp asgnExp
asgnOp ::=        = | |= | ^= | &= | <<= | >>= | += | -= | *= | /= | %=
condExp ::=       term0 [ ? exp : condExp ]?
term0 ::=         term1 [ || term1 ]*
term1 ::=         term2 [ && term2 ]*
term2 ::=         term3 [ | term3 ]*
term3 ::=         term4 [ ^ term4 ]*
term4 ::=         term5 [ & term5 ]*
term5 ::=         term6 [ equalOp term6 ]*
equalOp ::=       == | !=
term6 ::=         term7 [ orderOp term7 ]*
orderOp ::=       > | >= | <= | <
term7 ::=         term8 [ shiftOp term8 ]*
shiftOp ::=       >> | <<
term8 ::=         term9 [ addOp term9 ]*
addOp  ::=        + | -
term9 ::=         term10 [ mulOp term10 ]*
mulOp ::=         * | / | %
term10 ::=        unaryOp term10 |
                  prim incDec* |
                  sizeof ( declPar )      -- 'declPar' must not declare any identifier.
unaryOp ::=       ~ | ! | * | & | addOp | sizeof | cast | incDec
cast ::=          ( declPar )             -- 'declPar' must not declare any identifier, and the result type may not be
                                             struct/union/function.
incDec ::=        ++ | --
prim ::=          prim1 primQualif*
prim1 ::=         ( exp ) | cst | ident
primQualif ::=    ( argList ) | [ exp ] | . ident | -> ident
argList ::=       [ asgnExp [ , asgnExp ]* ]?
cst ::=           ' character ' |
                  " string " [ " string " ]* |
                  sgndNb |
                  nb
sgndNb ::=        [ + | - ] nb
```

Restrictions with respect to ANSI C:

- no old-style function definition,
- imposed order for 'attrib' and 'qualif',
- no default 'int' baseType,
- a switch must control a block; 'default' must come as last case; 'case', 'default' not labels,
- no unparenthesized assignment in initializer,
- braces in initializer only to indicate initialization of a compound object (array/struct/union); if so, mandatory,
- enum, char : types by themselves (not considered as int),
- automatic arithmetic conversions only from narrower to wider type,
- identifiers may never be declared more than once in the same scope (except in header files),
- macros (#define) cannot be declared more than once (if not '#undef'ed beforehand),
- labels local to blocks (but cannot be duplicated in a given function),
- a function or array is different from a pointer on the same function/array,
- left operand of indexation must be array/pointer,
- void * only universal receiver for pointers (cast or d-pragma needed the other way around, except for NULL).

APPENDIX B

```
/**********************************************************************/
/*      Sum of two matrix of arbitrary size (without use of pointer)      */
/**********************************************************************/
#include <stdio.h>
#include "dynarray.h"

/* 'bool' type definition */
typedef unsigned int bool;
#define FALSE (0 != 0)
#define TRUE  (0 == 0)

/* type definition for matrix elements */
typedef float Num;
/* type definition for variable-size matrix */
typedef DynArray2(Num) DynArr;
typedef DynArray2(const Num) CDynArr;

/* Models (function profiles) */
static void print(CDynArr), read(DynArr);
static DynArr add(CDynArr, CDynArr);

int main()
{ bool ok;  unsigned int i, j;
  DynArr mat1 = NULL, mat2 = NULL, resMat;  /* matrix declarations (resMat
                                                         allocated by 'add' function). */
  do {
    for (;;) {
      printf("\nEnter matrix dimensions: ");
      if (scanf("%u%u",&i,&j) == 2) break;  /* exit loop if numbers read are correct */
      while (getchar() != '\n') {}}  /* flush input buffer */
    AllocDynArray2(mat1, i, j, &ok);  /* allocate space for matrix 'mat1', freeing
                                                              possible previous one. */
    if (! ok) printf("\n memory overflow for 1st matrix");
    else {
      printf("\nmat1=\n"); read(mat1);
      AllocDynArray2(mat2, i, j, &ok);
      if (! ok) printf("\n memory overflow for 2nd matrix");
      else {
        printf("\nmat2=\n"); read(mat2);
        resMat = add(mat1, mat2); print(resMat);
        FreeDynArray2(resMat);}}  /* free result matrix space */
  } while (i != 0);
  return 0;
}

static void read(DynArr x)
{ unsigned int i, j, l1, l2;

  LimDynArray2(x, &l1, &l2); /* get dimensions of matrix 'x' */
  for (i = 0; i < l1; i++)  {for (j = 0; j < l2; j++) {(void)scanf("%g", &x[i][j]);}}
}

static void print(CDynArr x)
{ unsigned int i, j, l1, l2;

  printf("\n");
  LimDynArray2(x, &l1, &l2);
  for (i = 0; i < l1; i++)  {for (j = 0; j < l2; j++) {printf("%g ", x[i][j]);} printf("\n");}
}

static DynArr add(CDynArr x, CDynArr y)
{ unsigned int i, j, l1, l2;  bool ok;  DynArr z = NULL;

  LimDynArray2(x, &l1, &l2);
  AllocDynArray2(z, l1, l2, &ok);  /* allocate space for result matrix */
  if (! ok) printf("\n memory overflow for result matrix");
  else for (i = 0; i < l1; i++)  {for (j = 0; j < l2; j++) {z[i][j] = x[i][j] + y[i][j];}}
  return z;
}
```