1530

MH

Bell Laboratories Cover Sheet for Technical Memorandum

Title: DOC: A Dialect of the Programming Language C

Date: February 14, 1980

TM: 80-1359-2

Other Keywords: preprocessor
programming language
reliable programming

Author(s)	Location	Extension	
D. K. Sharma	MH 7B224	2914	Charging Case: 39394 Filing Case: 39394

ABSTRACT

This paper describes the programming language DOC, a dialect of C. DOC provides control statements that have clean semantic properties, are easier to understand and use, and allow the program text to be formatted neatly. These control statements are modelled after the guarded commands proposed by E. W. Dijkstra and are extremely useful in developing correct programs.

As compared to their C counterparts, DOC programs are expected to be easier to read and write, and therefore easier to understand and maintain — program properties whose importance can not be over emphasized. It has been the author's observation that DOC programs are at least as efficient as the C programs, and frequently they are indeed more efficient.

DOC programs are translated into C using a preprocessor, whose output is then compiled by a regular C compiler. They can contain the C preprocessor commands, such as `#define` and `#include`, and their object modules can be linked with those of C programs.

This paper also contains the manual pages for the preprocessor, the command to compile and link DOC programs, and the filter program to underline DOC keywords.

Pages Text: 9	Other: 12	Total: 21
No. Figures: 0	No. Tables: 0	No. Refs.: 2

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

ALLEN,ROBERT B
ALLES,H G
ANSELMO,DONALD H
ARNOLD,THOMAS F
<BARON,ROBERT V
BERGLAND,G D
BLAHUT,D E
+BLEWETT,C DOUGLAS
BOHNING,J B
BROWN,W STANLEY
CAMLET,J V,JR
CANADAY,RUDD H
CHOW,W F
CHRISTENSEN,C
CLOGSTON,ALBERT M
COPP,DAVID H
DE GRAAF,D A
DOLOTTA,T A
DWORAK,F S
ELDREDGE,GARY F
EPSTEIN,ROBERT H
FISHER,EDWARD R
FRASER,A G
FREENY,STAN L
+GEHANI,NARAIN H
<GIORDANO,PHILIP P
HAIGHT,R C
HALL,ANDREW D,JR
+HANNAY,N B
HOFMANN,A M
+JAGANNATHAN,ANAND
+JUDICE,C N
+KATZENELSON,JACOB
<KEESE,W M
+KERNIGHAN,BRIAN W
KOGELNIK,H
LIMB,JOHN O
LUCKY,R W
LUDERER,GOTTFRIED W R
MARANZANO,J F
MARTELLOTTO,N A
MAST,C A
MAXEMCHUK,NICHOLAS F
<MC DONALD,H S
+MC DONNELL,J P
MCILROY,M DOUGLAS
MILLER,STEWART E
MOLINELLI,JOHN J

COMPLETE MEMORANDUM TO

MORGAN,SAMUEL P
NETRAVALI,A N
NINKE,WILLIAM H
PENZIAS,A A
+PRIM,R C
RALEIGH,T M
REED,E A
REUDINK,D O
+RITCHIE,D M
ROBERTS,CHARLES S
+ROSIN,ROBERT FISHER
ROVEGNO,HELEN D
ROWLAND,BRUCE R
SABSEVITZ,A L
<SCHONFELD,TIBOR J
SHARMA,D K
SIMONE,C F
SINOWITZ,NORMAN S
SLANA,M F
SLICHTER,W P
SO,H C
<STORER,JAMES A
SWARTZWELDER,J C
TAGUE,BERKLEY A
TERRY,MILTON E
<TEWKSBURY,S K
THOMAS,LEE C
WELLER,D R
+WETHERELL,CHARLES S
YACOBELLIS,ROBERT H
78 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ABATEMARCO,TERESA M
ABATE,JOSEPH
ACKERMAN,A FRANK
ACKERMAN,J T
ACKLAND,BRYAN D
ACKROFF,JOHN M
AHO,ALFRED V
AHRENS,RAINER B
AHUJA,SUDHIR R
ALBAGLI,V R
ALBERALLA,RICHARD J
ALCALAY,D
ALEXIS,A D,JR
ALKONS,FREDERICK
ALLISON,C E,JR
AMIN,ASHOK T
AMITAY,N

COVER SHEET ONLY TO

AMRON,IRVING
ANDERSON,FREDERICK L
ANDERSON,KATHRYN J
ANDERSON,MILTON M
ANDERSON,R E
ANDERSON,R R
ANDERSON,ROBERT V
ANDERSON,W A
ANDREWS,A J
ANTOLICK,DAVID R
APPELBAUM,MATTHEW A
ARCHER,RUSSELL E,JR
ARMSTRONG,D B
ARMSTRONG,F O,JR
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,JAMES Q
ARNOLD,PHYLLIS A
ARVIDSON,= P
ASELTINE,EDWARD G
ASMUTH,RICHARD L
ASTHANA,ABHAYA
ATAL,BISHNU S
AXELSON,A L
BABU,RAJESH RATILAL
BAILEY,CATHERINE T
BAKER,DONN
BAKER,MITCHELL B
BALDWIN,GEORGE L
BALENSON,CHRISTINE M
BALLANCE,ROBERT A
BARBATO,ROBERT R
BARCLAY,DAVID K
BARNS,A L
BAROFSKY,ALLEN
BARR,DAVID L
BATTAGLIA,FRANCES
BAUER,BARBARA T
BAUER,H C
BAUER,HELEN A
<BAUER,WOLFGANG F
<BAUGH,C R
BAUMAN,STEVEN M
<BAXTER,LESLIE A
BAYER,D L
BEACHY,MILTON
BEBLO,WILLIAM
BECERRA,PEDRO D
BECKER,CURTIS A
BECKER,GARY E
BECKER,JACOB I
BECKER,RICHARD A
BECKETT,J T
BEDNAR,JOSEPH A,JR
BEGLEY,ALOYSIUS A
BEIGHLEY,KEITH A
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,RICHARD L

COVER SHEET ONLY TO

BENNETT,WILLIAM C
BENOWITZ,P
BENSING,JAMES EDWARD
BERENBAUM,ALAN
BERGH,A A
BERKEY,M A
BERK,DONALD A
BERNHARDT,RICHARD C
BERNOSKE,BEVERLY G
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BERNSTEIN,PAULA R
BERRYMAN,R D
BERZINS,ALEXANDER H
<BEYER,JEAN-DAVID
BEYLER,ERIC
BHATIA,RAJIV
BIANCHI,M H
BICKFORD,NEIL B
BILASH,TIMOTHY D
BILLINGTON,MARJORIE J
BILOWOS,R M
BIREN,IRMA B
BISHOP,J DANIEL
BISHOP,THOMAS P
BISHOP,VERONICA L
BITTNER,B B
BITTRICH,MARY E
BLAKE,GARY D
BLAZIER,S D
BLECHMAN,RONALD I
BLEIER,JOSEF
BLINN,J C
BLOSSER,PATRICK A
BLUM,MARION
BOCHULA,EDWARD J
BOCKUS,ROBERT J
BOCK,NANCY E
BODDIE,JAMES R
BODEN,F J
BOEHM,EARL W
BOEHM,KIM R
BOESE,J O
BOGART,F J
BOGART,THOMAS G
BOIVIE,RICHARD H
BOLSKY,MORRIS I
BONANNI,L E
BOND,F C
BORDELON,EUGENE P
BORG,KEVIN E
BORISON,ELLEN A
BOSE,DEBASISH
BOSTON,RONALD E
BOSWELL,PAULA S
BOUMA,HERMAN J
BOURNE,STEPHEN R
BOWEN,E G
BOWYER,L RAY
BOYCE,W M

COVER SHEET ONLY TO

BOYER,PHYLLIS J
BOYLE,GERALD C
BRADFORD,EDWARD G
BRADLEY,M HELEN
BRADLEY,R H
BRAINARD,R C
BRANDT,RICHARD B
BRAUNE,DAVID P
BRAUN,DAVID A
BREILAND,JOHN R
BRENSKI,EDWIN F
BRESLER,RENEE A
BRIGGS,GLORIA A
BRITT,WARREN D
BROAD,MARTHA M
BRONSTEIN,N
BRONZO,JOSEPH A
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROVMAN,INNA
BROWNING,JASON DAVID
BROWN,EARL F
BROWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BROWN,MARK S
BROWN,STUART G
BROWN,W R
BRUECKNER,DOUGLAS E
BRYANT,DAVID J
BUCK,I D
BULLEY,R M
BURGESS,JOHN T,JR
BURG,F M
BURIC,MILORAD R
BURKE,MICHAEL E
BURKE,R J
BURNETTE,W A
BURNETT,DAVID S
BURNET,ROSE M
BUROFF,STEVEN J
BURROWS,THOMAS A
BURR,DAVID JOSEPH
BUSCH,KENNETH J
BUTLETT,DARRELL L
BUTTON,BEVERLY
<BUTZIEN,PAUL E
BYERLEE,R W
BYCRICK,ROBERT S
BYRNE,EDWARD R
CALL,PETER F
CALVERT,KENNETH L
CAMPBELL,JERRY H
CAMPBELL,MICHAEL E
CANDREA,RONALD D
CANDY,J C
CANNATA,PHILIP E
CAREY,J E
CARRAN,JOHN H
CARRIGAN,RAYMOND J
CARR,DAVID C
...
1416 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION..

COMPLETE MEMO TO:
135-DPH 13-DIR 11-EXD 13-EXD 15-EXD 16-EXD 1359-AMTS 127-DPH 36-DIR 364-DPH
3641-SUP 3644-SUP 363-DPH 3631-SUP 3633-SUP

COVER SHEET TO:
135-AMTS

COPLGP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE
COPRMP = COMPUTER PROGRAMMING METHODOLOGY
UNPLCL = C LANGUAGE

HO CORRESPONDENCE FILES
HO 1A127

TM-80-1359-2
TOTAL PAGES 20

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

subject: DOC: A Dialect of the Programming Language C
Case: 39394
File: 39394

date: February 14, 1980

from: D. K. Sharma
MH 1359
7B224 x2914

TM: 80-1359-2

MEMORANDUM FOR FILE

1. INTRODUCTION

The programming language DOC is a dialect of C [1] in which the control statements have clean and simple semantic properties. We believe that if the formal semantics of control statements is easily grasped, they are easy to understand intuitively, and their usage is less error prone. The control statements in DOC are succinct and permit the control structure of the program to be displayed neatly. This makes DOC programs easy to read and write and therefore easy to understand and maintain. These program properties are crucial in developing reliable software, and their importance can not be over emphasized.

The control statements in DOC are: the `do` statement, `if` statement, `cif` (constant-if) statement, and `for` statement. Compound statements are enclosed by `begin` and `end`, but this is necessary only once per function definition. The `do` and `if` statements are similar to those proposed by Dijkstra [2], except for the following differences. The guards, defined in Section 5.4, are evaluated in the order of their appearance until a true guard is found; the statements following that guard are then executed. In [2], if several guards are true, the statements following any one of them may be selected for execution, thus giving rise to the nondeterministic DO and IF constructs. The `cif` statement is similar to the `switch` statement of C, and as far as its semantics is concerned, it is really a special case of the `if` statement. The `for` statement is almost the same as in Pascal; it is intended to execute a statement sequence while stepping through the values of a variable in steps of +1 or −1.

DOC may be used for program development in two ways.

1. Readers familiar with [2] may develop a program and its proof using the nondeterministic DO and IF constructs. They may run the resulting program as it is — the DO and IF constructs are correctly implemented by the `do` and `if` statements of DOC. Or on the other hand, they may choose to

   a. make use of the `else`-clause and the prespecified order of guard evaluation to eliminate redundant expression evaluations (Section 5.6), and

   b. replace certain `if` statements by `cif` statements to enable the compiler to generate more efficient code.

2. Alternatively, readers who do not wish to use the above method may develop DOC programs just as most programs are currently developed, that is, by the 'operational approach', in which the programmer thinks during program development as the computer would act during program execution.

In either case, it is quite natural to write DOC programs that are more efficient than what would be natural to write in C (Section 5.6). This, of course, is not to say that equally efficient C programs can not be written (such a statement would be false in view of labels and the `goto` statement in the language), but that they would be unduly cumbersome to write. Readers who are interested only in the main aspects of DOC may at this point skip to Section 5, where we describe the syntax and semantics of the various statements informally and give examples to illustrate their usage.

In addition to the control statements, DOC also differs from C in a few other respects. The names of a few operators have been changed. The syntax of structure and union declarations has been slightly altered, and additional data types `bool`, `boolptr`, `charptr`, `intptr`, `floatptr`, and `doubleptr` have been introduced.

DOC programs are translated into C using a preprocessor whose output is then compiled by a C compiler. The error messages from the C compiler can be easily traced back to the source programs in DOC, because the DOC preprocessor does not change line numbers and does not introduce new variables or remove the existing ones. DOC programs can contain the C preprocessor commands, such as `#define` and `#include`. The files thus 'included' must themselves contain DOC program segments, and the same holds for nested 'include' files also. The object modules of DOC programs can be linked with those of C programs.

Appendix A contains the manual pages for the preprocessor and for the command to compile and link DOC and C programs. The manual page for a filter program that underlines the keywords of DOC is also included. On hardcopy terminals with back-spacing capability, this program is intended to produce program listings that are easy to read: the control flow and the global structure of programs can be quickly discerned by looking at the underlined keywords. Appendix B contains a sample DOC program.

In the following, we assume that the reader is familiar with the C language [1] and describe only the aspects of DOC in which it differs from C.

2. COMMENTS

Comments begin with { and end with } .

3. DECLARATIONS

The syntax of declarations is essentially the same as in C, except for the following changes.

3.1 Variable Initializations

The assignment operator := is used to initialize variables, and (: and :) are used to begin and end aggregate initializers, respectively (instead of { and } in C). See also Section 4.8.

3.2 Pointer Declarations

The unary operator @ is used to declare pointers. For example,

```
int @xp, x:=1;
```

declares `xp` as an integer pointer and `x` as an integer which is initialized to 1.

Alternatively, pointers may be declared by using one of the *type-specifiers* `charptr`, `boolptr`, `intptr`, `floatptr`, and `doubleptr`. For example

```
intptr xp, fip(), (@pfip)();
```

declares `xp` to be an integer pointer, `fip` to be a function returning an integer pointer, and `pfip` to be a pointer to a function returning an integer pointer.

3.3 Structure and Union Declarations

In struct and union declarations, the left brace is omitted, and the right brace is replaced by end. The semicolon preceding end is optional (Section 5.8). Thus,

```
struct
   int i;
   char c;
end s;
```

declares a structure s with components i and c, and

```
struct
   int i;
   struct char a, b end p;
end s;
```

declares a structure s with another structure p as one of its components.

Identifiers can not be declared as *structure-tags* or *union-tags* which are described in the C-reference manual. Type definitions must be used for that purpose. In the following example, a data type complex is defined and then used to declare a variable z.

```
typedef struct
            float  real;
            float  imagin;
        end complex;
complex z;
```

The real and imaginary parts of z are accessed by z.real and z.imagin, respectively.

3.4 Boolean Variables

Boolean variables can be declared using the *type-specifier* bool. In addition, the two boolean values are available as the identifiers true and false. They can be assigned to boolean variables and used in logical expressions.

3.5 Function Declarations

Function declarations may be preceded by one of the keywords procedure and function. The keyword function should be used when the function is expected to return a value, and procedure should be used when no return value is expected. See Section 5.3 for an example.

4. EXPRESSIONS

The syntax of expressions is the same as in C, except that four predefined constants are available and a few operators are represented differently, as described in the following. The operators in C, and only those, that are not equivalent to the operators described below remain unaffected.

4.1 Constants

The keywords nullptr, nullch, true, and false are treated as constants. (They stand for 0, '\0', 1, and 0, respectively. The representations of these constants should not, in principle, be included in the language definition; this has been done here to facilitate interfacing DOC and C programs.)

4.2 Primary Expressions

The operator ^. replaces the arrow operator (->) of C.

4.3 Unary Operators

@*pointer-expression* : lvalue expression referring to the object that resides 'at' the location given by *pointer-expression*.

^*lvalue* : pointer to the object referred to by *lvalue expression*.

`not` *expression* : logical `not` of *expression*.

`bnot` *expression* : bit-wise `not` (i.e., one's complement) of *expression*.

The other unary operators, namely,

`- ++ --` (*type-name*) `sizeof`

remain the same as in C.

4.4 Arithmetic Operators

expression `div` *expression*
expression `mod` *expression*

The `div` and `mod` operators are the same as / and %, respectively; their use is recommended for clarity when both the expressions yield integer values. The / and % operators should be used for real valued expressions. All the arithmetic operators of C, i.e.,

`*   /   %   +   -   <<   >>`

remain unaffected.

4.5 Equality Operators

expression = *expression* : `true` when the two expressions are equal and `false` otherwise.

expression <> *expression*: `true` when the two expressions are not equal and `false` otherwise.

The *relational operators* of C, namely,

`<   >   <=   >=`

remain the same.

4.6 Logical Operators

expression `and` *expression* : logical `and` of the two expressions.

expression `or` *expression* : logical `or` of the two expressions.

The operators `and` and `or` should be used where their operands can be interchanged without affecting the program. If, however, that is not the case, the operators `cand` and `cor` (conditional `and` and conditional `or`, respectively) should be used. They are defined as follows: If x equals `false`, then the value of `x cand y` is `false` and y is not evaluated; otherwise, the value is y. Similarly, if x equals `true`, then the value of `x cor y` is `true` and y is not evaluated; otherwise, the value is y. For example, in the evaluation of `(i>0 cand a[i-1]=' ')` and `(i<>0 cand (j div i)>5)` for i=0, the operand following `cand` is not evaluated.

4.7 Bitwise-logical Operators

expression `band` *expression* : bit-wise logical `and` of the two expressions.

expression `bor` *expression* : bit-wise logical `or` of the two expressions.

expression `bxor` *expression* : bit-wise logical `exclusive-or` of the two expressions.

4.8 Assignment Operators

Simple assignment operator `:=`

Compound assignment operators

```
:+   :-   :*   :/   :%   :div   :mod   :>>   :<<
:band     :bxor     :bor
```

Here, "a := b" should be read as a becomes b, or a gets b; "a :+ b" should be read as a becomes a plus b; etc.

5. STATEMENTS

This is the syntactic category in which DOC differs the most from C and which was also the primary motivation behind creating DOC.

The statements of DOC discussed in the following are: `skip`, simple, compound, `if`, `cif`, `do`, and `for` statements. The `break` and `continue` statements are not available in DOC, and the `case` and `default` labels are not needed. The `return`, `goto`, labelled, and null statements are the same as in C and are not discussed below. Section 5.8 summarizes the treatment of semicolons.

5.1 Skip Statement

It has the form

```
skip;
```

and is equivalent to the null statement of C.

5.2 Simple Statement

A simple statement is

expression;

Note that expressions can have embedded assignment operators.

5.3 Compound Statement

We define *statement sequence* as a sequence of statements optionally separated by comments or white spaces (i. e., spaces, tabs, and newline characters). Then the compound statement has the form

```
begin
 [ declarations ]
 statement sequence
end
```

where the brackets delineate optional items. This is the same as in C, except that the braces are replaced by `begin` and `end`. For example,

```
procedure swap(x,y)
int  x, y;
begin
    int temp;
    temp := x;
    x := y;
    y := temp;
end
```

The part enclosed by `begin` and `end` is referenced quite frequently in the following, and for convenience we will call it *compound body*. That is, *compound body* is

[*declarations*]
statement sequence

5.4 If Statement

The `if` statement has the following form

`if` *expression* -> *compound body*
`||` *expression* -> *compound body*
.
.
.
`||` *expression* -> *compound body*
[`||` `else` -> *compound body*]
`fi`

To execute the `if` statement, the *expressions*, also known as the guards, are evaluated in the order of their appearance until a true *expression* is found. The *compound body* corresponding to that *expression* is then executed. If none of the given *expressions* are true, two cases arise: (1) if the optional `else`-clause is not present, the program is aborted with an error message, and (2) if the `else`-clause is present, its corresponding *compound body* is executed. Thus, `else` should be thought of as the complement of the disjunction of all the other *expressions*. If the guards in an `if` statement are all-inclusive, the last guard may be replaced by `else` to avoid unnecessary expression evaluation.

For example,

```
if a=1 ->  c := 2;  {Initialize}
           d := 3;

|| a=2 ->  int temp; {Swap using a local variable}
           temp := c;
           c := d;
           d := temp;

|| else -> print("Improper value of a");
fi
```

5.5 Constant-if (or cif) Statement

The `cif` statement has the form

`cif` *expression*
`is` *constant expression list* -> *compound body*
`||` *constant expression list* -> *compound body*

```
   || constant expression list -> compound body
 [ ||   else    -> compound body ]
   fic
```

where *constant expression list* is a comma-separated list of *constant expressions* (the guards), and all the *constant expressions* in a `cif` statement must have distinct values. *Constant expression* is completely defined in [1]; briefly, it evaluates to a constant and can involve only `int` constants, `char` constants, `bool` constants, and `sizeof` expressions. Note that *constant expressions* do not contain embedded comma operators.

A `cif` statement is executed by evaluating the *expression* and then executing the *compound body* whose *constant expression* has the same value as *expression*. If the value of *expression* does not match any of the *constant expressions*, two cases arise that are analogous to the `if` statement: (1) if the optional `else`-clause is not present, the program is aborted with an error message, and (2) if the `else`-clause is present, the *compound body* corresponding to it is executed. The interpretation of `else` is the same as in the `if` statement described above. Notice that the `case` and `default` labels and the `break` statement of C are not available.

For example,

```
cif i
is    1     -> print("message 1");
||    2     -> print("message 2");
|| 3,4,5    -> print("message 345");
||  else    -> print("unknown i");
fic
```

5.6 Do Statement

The `do` statement has the form

```
do expression -> compound body
|| expression -> compound body
        .
        .
        .
|| expression -> compound body
od
```

Each iteration through the `do` statement involves evaluating the *expressions* (the guards) in the order of their appearance until a true *expression* is found. The *compound body* corresponding to that *expression* is then executed. This is repeated until all the *expressions* become false; the `do` statement is then terminated. Notice that the `continue` and `break` statements of C are not available.

For example, the following program counts the number of tab characters in the string `line`. It was obtained using the scheme described in [2]; the assertions related to its proof are omitted.

```
count := 0; i:= 0;
do line[i]<>nullch -> if line[i]= '\t' -> count++; i++;
                      || line[i]<>'\t' -> i++;
                      fi
od
```

By merging the guards of the `do` and `if` statements, we obtain the following program.

```
count := 0; i:=0;
do line[i]<>nullch and line[i]= '\t' -> count++; i++;
|| line[i]<>nullch and line[i]<>'\t' -> i++;
od
```

Note that both these programs do not rely upon the order of evaluation of the guards and that in the second program, the first component of the first guard is redundant. If we decide to make use of the order of guard evaluation, the second component of the second guard also becomes redundant, and the program reduces to

```
count := 0; i:= 0;
do line[i]= '\t'    -> count++; i++;
|| line[i]<>nullch -> i++;
od
```

This exemplifies how a DOC program obtained using the scheme described in [2] may be transformed to make it more efficient. It is, however, obvious that using the 'operational approach', the above program could have been obtained directly, but of course without a proof.

We now use this example to point out that certain DOC programs are more efficient than what would be natural to write in C to do the same task. Let t and n be the number of tab and non-tab characters, respectively, in `line`. Then, the number of comparisons done in the above program is t+2n. A C program to do the same task is

```
count = 0; i = 0;
while( line[i] != '\0' ){
        if( line[i] == '\t' )count++;
        i++;
}
```

The number of comparisons in this case is 2t+2n, as each character of `line` must go through two comparisons. The following C program is as efficient as the improved DOC program given above.

```
count = 0; i = 0;
for(;;){
        if( line[i] == '\t' )count++;
        else if( line[i] == '\0' )break;
        i++;
}
```

It is, however, more cumbersome to read and write than both the other programs. Note that all the three programs would benefit considerably from the use of pointers and register variables, but that was not done for the sake of clarity.

5.7 For Statement

The `for` statement is provided to step through a list of items in steps of +1 or −1. All other tasks requiring a looping construct are expected to use the do statement. The `for` statement has two forms:

`for` *var* := *expression1* `to` *expression2* `->`
 compound body
`rof`

and

`for` *var* := *expression2* `downto` *expression1* `->`
 compound body
`rof`

In the first (second) case, the variable *var* is first initialized to *expression1* (*expression2*); then, if *var<=expression2* (*var>=expression1*), *compound body* is executed; after this, *var* is incremented (decremented) by 1, and the control goes back to testing the inequality. The loop terminates if the inequality is not satisfied, and the value of *var* is left as *expression2*+1 in the first case and as *expression1*−1 in the second case.

In case of a transfer of control from the middle to the outside of the loop, *var* retains its value just before the transfer. Transfer of control from the outside to the middle of the loop should be avoided.

The example

```
sum := 0;
for i := 0 to n-1 ->
  sum := sum + a[i]
rof
```

does the same as

```
sum := 0;
for i := n-1 downto 0 ->
  sum := sum + a[i]
rof
```

except for the final values of `i`.

5.8 Semicolons

Semicolons followed by `end`, `fi`, `fic`, `od`, `rof`, or `||` can be omitted. This rule applies to structure declarations also. Semicolons may thus be treated as statement separators, in contrast with C, where they are treated as statement terminators.

6. MISCELLANEOUS

6.1 Preprocessor Command Lines

The preprocessor commands allowed in DOC programs are the same as in C. Note that these commands are acted upon by the C preprocessor, which is invoked before the DOC preprocessor during the process of compiling DOC programs. The DOC preprocessor ignores those commands. The files specified in `#include` commands must contain DOC program segments; the same holds for nested 'include' files, too.

6.2 Standard I/O Library

DOC programs that access the routines from the standard I/O library must 'include' the file `docio.h`, which defines appropriate macros and variables and is the DOC equivalent of `stdio.h`.

ACKNOWLEDGEMENTS

The author is grateful to N. Gehani and B. Dwyer for their comments on an earlier version of this paper.

MH-1359-DKS-dks

D. K. Sharma

Atts.
References 1-2
Appendixes A and B

REFERENCES

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[2] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Inc., 1976.

APPENDIX A

This appendix contains the manual pages for the following three programs:

(1) `docpp`: the DOC preprocessor to translate DOC programs into C.

(2) `dcc`: the DOC and C Compiler to compile DOC and C programs and link the resulting object modules.

(3) `ulk`: a filter program to under-line DOC keywords.

NAME

docpp − DOC preprocessor

SYNOPSIS

docpp [infile [outfile]]

DESCRIPTION

The DOC preprocessor accepts a DOC program from *infile*, converts it into an equivalent C program, and leaves the output in *outfile*. If *outfile* is absent or is −, it is assumed to be the standard output. If *infile* is absent or is −, it is assumed to be the standard input. Note that *infile* can not be absent without *outfile* also being absent.

Docpp produces one output line for each input line. That is, the line numbers in the resulting C file are the same as those in the input DOC file. Therefore, the error messages generated by the C compiler can be easily traced back to the source program in DOC, as they all contain line numbers in them. *Docpp* does not introduce new variable names or remove the old ones in the process of translation.

Docpp leaves unchanged the lines beginning with the character #. As a result, it does not expand macros, include files, etc. -- a task that is done by the C preprocessor.

Docpp prohibits the use of the following: (1) keywords of C not used in DOC, namely, **while**, **switch**, **default**, **case**, **break**, **continue**, and **entry**, (2) the operators: &, !, ˜, ==, !=, &&, and ||, (3) the composite assignment operators of C, which are of the form =*op* or *op*=, where *op* is +, -, *, /, %, >>, <<, &, ˆ, or |, and (4) the use of * for indirection in variable declarations and casts that begin with a predefined type specifier, namely, **char**, **bool**, **int**, etc. This is not the case if a type-specifier defined using typedef is used.

The keyword ebss is translated into **end**, and etext and edata are not modified. Recall that **ld** treats _end, _etext, and _edata as read-only reserved symbols.

For **if** and **cif** statements, to deal with the run-time condition when none of the specified alternatives apply, *Docpp* generates the calls IFERROR(filename, lineno) and CIFERROR(filename, lineno), respectively, where filename is a string pointer and lineno is an **int**. Standard version of these routines are available in **docerror.d**, and their object modules are in **libd.a**. They may be replaced by user supplied versions.

SEE ALSO

D. K. Sharma, *DOC: A Dialect of the Programming Language C.*
dcc(1), ulk(1)

NAME

dcc − DOC and C compiler

SYNOPSIS

dcc [option]... arg...

DESCRIPTION

This command has the following purposes: (1) to compile DOC and C programs, (2) to assemble assembly source programs, (3) to link the resulting object modules with libraries and previously obtained object modules, and (4) to run the C and DOC preprocessors in cascade in that order or individually.

It accepts several types of arguments:

Arguments whose names end with **.d** or **.c** are taken to be DOC or C source programs, respectively; they are compiled, and each object program is left on the file whose name is that of the source with **.o** substituted for **.d** or **.c**. If a single DOC or C program is compiled and loaded all at once, the **.o** file is deleted.

Arguments whose names end with **.s** are taken to be assembly source programs and are assembled, producing a **.o** file.

The **.d** files are converted to object files by running the C preprocessor, DOC preprocessor, and other passes of the C compiler in that order. The C preprocessor expands macros, include files, etc., and the DOC preprocessor converts the resulting DOC program into C. Thus, the files specified in the **#include** statements in **.d** files must themselves contain DOC program segments; the same holds for nested include files also.

The **.c** files are converted to object files just as in the *cc* command: by running the C preprocessor followed by the other passes of the C compiler. Thus, the files specified in the **#include** statements in **.c** files must contain C program segments; the same holds for nested include files also.

The following options are interpreted by *dcc*. See *ld*(1) for load-time options.

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startoff routine by one which automatically calls *monitor*(3C) at the start and arranges to write out a **mon.out** file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

−f In systems without hardware floating-point, use a version of the C compiler which handles floating-point constants and loads the object program with the floating-point interpreter.

−O Invoke an object-code optimizer.

−S Compile the named DOC and C programs, and leave the assembler-language output on corresponding files suffixed **.s**.

−E[dcb] Run only the designated preprocessors on the named DOC and C programs, and send the result to the standard output. When the flag value is **d**, run DOC preprocessor only; when the flag value is **c**, run C preprocessor only; when the flag value is **b**, run C and DOC preprocessors in that order. When no flag value is specified, run DOC preprocessor on **.d** files and C preprocessor on **.c** files.

−P[dcb] Run only the designated preprocessors on the named DOC and C programs, and leave the result on corresponding files suffixed **.i**. See the description of **-E** option regarding which preprocessor is run.

−o *output*
Name the final output file *output.* If this option is used, the file **a.out** will be left undisturbed.

−C Comments are not stripped by the C preprocessor.

−D *name*=*def*
−D *name*
Define the *name* to the C preprocessor, as if by **#define**. If no definition is given, the name is defined as 1.

−U *name*
Remove any initial definition of *name*.

−I *dir* Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the *file* argument, then in directories named in −I options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.

−B *string*
Find substitute compiler passes in the files named *string* with the suffixes **cpp**, **docpp**, **c0**, **c1** and **c2**. If *string* is empty, use a standard backup version.

−t[pd012]
Find only the designated compiler passes in the files whose names are constructed by a −B option. In the absence of a −B option, the *string* is taken to be **/lib/n**.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *dcc* or *cc* run, or perhaps libraries of DOC- and C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.** The libraries **libd.a**, **libc.a**, and **liba.a** are searched in that order; **libd.a** currently has routines to print the run-time error messages for DOC.

FILES

file.d	input DOC file
file.c	input C file
file.s	input assembler file
file.o	object file
a.out	loaded output
/tmp/ctm*	temporary
/lib/cpp	C preprocessor
/usr/lib/docpp	DOC preprocessor
/lib/c[01]	compiler, *cc*
/lib/oc[012]	backup compiler, *cc*
/lib/ocpp	backup C preprocessor
/lib/odocpp	backup DOC preprocessor
/lib/fc[01]	floating-point compiler, *cc*
/lib/c2	optional optimizer
/usr/lib/comp	compiler, *pcc*
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/fcrt0.o	startoff for floating-point interpretation
/usr/lib/libd.a	DOC library
/lib/libc.a	C library, see (3)

/lib/liba.a	assembler library used by some routines in **libc.a**.
/usr/include	standard directory for **#include** files

SEE ALSO

D. K. Sharma, *DOC: A Dialect of the Programming Language C.*
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, NY, 1978.
B. W. Kernighan, *Programming in C—A Tutorial.*
D. M. Ritchie, *C Reference Manual.*
docpp(1), ulk(1), adb(1), ld(1), prof(1), monitor(3C).

DIAGNOSTICS

The diagnostics produced by the compiler are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular **m**, which means a multiply-defined external symbol (function or data).

NAME

ulk – underline keywords of DOC

SYNOPSIS

ulk

DESCRIPTION

This program can be used as a filter only: it reads from the standard input file and writes on the standard output file. The output of *ulk* is such that when it is printed on hardcopy terminals with back-spacing capability, certain keywords of the programming language DOC come out underlined. Each character to be underlined is replaced by an underscore, a backspace, and the character itself in that order.

Ulk is intended to produce listings of DOC programs that are easy to read: the control flow and global structure of the program can be quickly discerned by looking at the underlined keywords.

SEE ALSO

D. K. Sharma, *DOC: A Dialect of the Programming Language C.*
dcc(1), docpp(1)

APPENDIX B

This appendix contains the listing of `tab`, a sample DOC program that expands or compresses tabs according to the tabstops specified by one of its arguments. The listing was obtained by passing the file `tab.d` through the filter program *ulk* and then printing the output on a DASI 450 terminal.

The underlined keywords in this listing make the control structure easily discernable and the program more readable. The reader is urged to closely examine the formatting of nested DOC statements: The branching conditions for `if` and `cif` statements and the looping conditions for `do` statements are displayed on the left and their corresponding statement sequences on the right.

```
{ Name          : tab
  Call sequence: tab option file1 file2
                 Where option is +[n] or -[n] or blank.
  Explanation  :
        If option is +n, tabs are expanded as if tab stops have
        been placed at columns 0, n, 2n, etc.  If option is -n,
        the inverse of the above is done by replacing
        consecutive spaces by tabs.  Blank option means +8, and
        absent n means 8.

        An absent file name stands for stdin or stdout. Note
        that infile can not be absent with out the outfile also
        being absent.
}
#define Linesize 257
#define FilopE   1
#define FilovrE 2
#define FlgcntE 3
#define FilcntE 4
#define LinlenE 5
#define FlgvalE 6
#include "header.h"
charptr progname;

procedure main(argc,argv) int argc; charptr @argv;
begin
    fileptr   infile, outfile,
              openwdef();  { Opens a file; returns a fileptr.
                             Aborts with a message if an existing
                             file may be overwritten. Returns
                             default fileptr supplied in the
                             call, if file name is null.
                           }
    charptr   infilnam:="", outfilnam:="", result, getl();
    int       filec := 0,  { File name count.}
              flagc := 0,  { Flag count.     }
              flagv := 8,  { Flag value.     }
              i, nvalue();
    char      line[Linesize];
    bool      exp := true;

progname := argv[0];        { Used by error() and errors().}
for i := 1 to argc-1 ->
    cif argv[i][0]
    is '-', '+' -> flagc++;
                   cif flagc
                   is  1  -> flagv := nvalue(argv[i]+1);
                             if flagv=0 -> flagv := 8 || else -> skip
                             cif argv[1][0]
                             is '-' -> exp := false;
                             || '+' -> exp := true;
                             fic
                   || else -> error(FlgcntE);
                   fic
    ||  else   -> filec++;
                  cif filec
                  is 1 -> infilnam := argv[i];
```

```
                        || 2 -> outfilnam:= argv[i];
                        || else -> error(FilcntE);
                        fic
    fic
rof

infile  := openwdef(infilnam,  "r", stdin);
outfile := openwdef(outfilnam, "w", stdout);

result := getl(line,infile); { Nullptr returned at the end of file! }
do result <> nullptr -> if exp -> expand(line, flagv, outfile);
                        ∏ else-> compress(line, flagv, outfile);
                        fi
                        result := getl(line, infile);
od

exit(0);
end { main }

    {    Convert an ascii string to an integer.}
function int nvalue(p) charptr p;
begin
    charptr q;
    char    c;
    int i;

    {    Note: The do construct following this comment critically
         relies upon the guards being evaluated in the order
         they are written. Considering it a nondeterministic do
         construct would be an error: examine the case when c =
         'a'. This loop is, however, more efficient than its
         nondeterministic counterpart, which is
         q := p; c := @q;
         do c <> '\0'-> if c >= '0' and c <= '9' -> q++; c := @q;
                        || else -> error(FlgvalE);
                        fi
         od
    }
    q := p; c := @q;
    do c >= '0' and c <= '9' -> q++; c := @q;
    ∏     c <> nullch        -> error(FlgvalE);
    od
    sscanf(p,"%d",^i); { i = 0 if p points to a null or blank string.}
    return(i);
end { nvalue }

    {    Read a line from iostream fp.  If the line in fp was
         longer than Linesize, abort with the error message
         corresponding to LinlenE.
    }
function charptr getl(line, fp) char line[]; fileptr fp;
begin
    charptr result, fgets();
    line[Linesize-2] :=  '\n';
    result := fgets(line, Linesize, fp);
    {    If the last character read is not '\n', the line in the
```

```
        file is longer than Linesize-1 -- an error.
    }
    if line[Linesize-2] <> '\n' -> error(LinlenE);
    ||           else           -> return(result);
    fi
end { getl }

    {    Expand tabs into spaces. It is inverse of compress.}
procedure expand(line,tabv,fp) char line[]; int tabv;  fileptr fp;
begin
    int nspaces, i, j, k;
    i := 0; j := 0;
    do line[i] = '\t'-> nspaces := tabv - j mod tabv;
                        for k := 1 to nspaces -> putc(' ',fp) rof
                        j :+ nspaces; i++;
    || line[i] <> nullch ->  putc(line[i], fp);
                             j++; i++;
    od
end { expand }

#define putl fputs

    {    This procedure is the inverse of expand.  }
procedure compress(line, tabv, fp) char line[]; int tabv; fileptr fp;
begin
    int nchtogo, nspaces, i, j;
    {    i and j are indices into the source and destination
         arrays which, in this case, are both line[]. nspaces is
         the no. of spaces transferred into the destination array
         after the last non-blank character.  If the source array
         has nchtogo  blanks after a non-blank character, they
         will be replaced by a tab in the destination array. Note
         that nchtogo is a function of i and tabv.
    }
    i := 0; j := 0; nspaces := 0; nchtogo := tabv;
    do line[i] =' '-> if nspaces=nchtogo - 1 ->  j :- nspaces;
                                                 line[j] := '\t';
                                                 nspaces := 0;
                                                 nchtogo := tabv;
                      ||        else        ->   line[j] := ' ';
                                                 nspaces++;
                      fi
                      j++; i++;
    || line[i] <> nullch  -> line[j] := line[i];
                             j++; i++;
                             nchtogo := tabv - i mod tabv;
                             nspaces := 0;
    od
    line[j] := nullch;
    putl(line,fp);
end { compress }

function charptr errmsg(i) int i;
begin
  charptr p;
  cif i
```

```
    is FlgcntE -> p := "too many flags present (max = 1)"
    || FlgvalE -> p := "improper flag value"
    || FilcntE -> p := "too many file names present (max = 2)"
    || LinlenE -> p := "an input line too long (max = 255)"
      {  The following error messages are used by errors(..).
         They contain one %s specification.
      }
    || FilovrE -> p := "file \"%s\" already exists"
    || FilopE  -> p := "can not open \"%s\""
    fic
    return(p);
end { errmsg }

    {    Print the i-th error message, the pointer to which is
         returned by errmsg().  The string must not contain any
         format specification.
    }
procedure error(i) int i;
begin
    charptr errmsg();
    fprintf(stderr, "%s: %s.\n", progname, errmsg(i));
    exit(i);
end { error }

    {    The string the pointer to which is returned by errmsg()
         must contain one %s.
    }
procedure errors(i,sp) int i; charptr sp;
begin
    charptr errmsg();
    fprintf(stderr,"%s: ", progname);
    fprintf(stderr, errmsg(i), sp);
    fprintf(stderr,".\n");
    exit(i);
end { errors }
```