

Using <stdio.h> in the output of the Fortran to C
translator, f2c

G F LEVY

NAG Ltd, Wilkinson House,
Jordan Hill Road,
Oxford, OX2 8DR, U.K.
(email:george@nag.co.uk)

Abstract

The automatic conversion of Fortran 77 to C can be accomplished using f2c, but the generated code is difficult to read and maintain, and needs to be linked with various non-standard run-time libraries which provide input/output and mathematical routines. This paper describes tools that have been developed at NAG Ltd to automatically replace non-standard run-time f2c input/output functions with their equivalent <stdio.h> functions.

Key Words f2c Fortran 77 Formats <stdio.h> Translation

Introduction

The automatic conversion of Fortran 77 to C can be accomplished using the tool f2c [1,2]; however, the generated code is difficult to read and maintain and has to be linked with the appropriate non-standard f2c run-time libraries which provide input/output and mathematical routines.

The chief reasons why the f2c output is difficult to read are that matrix elements are accessed using pointer notation/calculations, and because all Fortran 77 READ or WRITE statements are transformed into several statements involving calls to various f2c run-time routines.

Tools have been developed at NAG which, through the automatic creation of macro definitions, allow matrix elements to be accessed using C code that closely resembles the original Fortran [3]. However, the problem of input/output was not addressed. To obtain code that used standard C input/output routines then required manual replacement of f2c run-time routines with their <stdio.h> equivalents.

This paper describes new tools which deal with this issue. They minimise, and in most cases eliminate altogether, the occurrence of f2c run-time input/output routines in the code. When combined with the previous tools, these tools produce output that requires only minor modifications to obtain code which uses standard C libraries and which can easily be compared with the original Fortran 77.

The tools were created using standard UNIX utilities [5,6,7,8] and have been thoroughly tested using the example programs and stringent tests associated with the NAG Fortran 77 Library [9]. Validation of the processed code was performed by automatically checking that the generated result files were identical to those produced by the original Fortran.

An example

An example of the type of processing that the tools provide is now given. The following Fortran 77 code fragment contains both an “implied do loop” and an integer repetition applied to a set of paired brackets.

The original Fortran

```
WRITE (NOUT,*) 'Eigenvalues'
WRITE (NOUT,99998) (' (',WR(I),',',',WI(I),')',I=1,N)

99998 FORMAT (1X,4(A,F7.3,A,F7.3,A))
```

The “raw output from f2c”

```
#include "f2c.h"
/* Format strings */
static char fmt_99998[] = "(1x,4(a,f7.3,a,f7.3,a))";

/* Builtin functions */
integer s_wsle(), do_lio(), e_wsle(), s_wsfe(), do_fio(), e_wsfe();

/* Fortran I/O blocks */
static cilist io___21 = { 0, 6, 0, 0, 0 };
static cilist io___22 = { 0, 6, 0, fmt_99998, 0 };

s_wsle(&io___21);
do_lio(&c__9, &c__1, "Eigenvalues", 11L);
e_wsle();
s_wsfe(&io___22);
i__2 = n;
for (i = 1; i <= i__2; ++i) {
    do_fio(&c__1, " (", 2L);
    do_fio(&c__1, (char *)&wr[i - 1], (ftnlen)sizeof(doublereal));
    do_fio(&c__1, ",", 1L);
    do_fio(&c__1, (char *)&wi[i - 1], (ftnlen)sizeof(doublereal));
    do_fio(&c__1, ")", 1L);
}
e_wsfe();
```

It can be seen that the input/output format string in the C code generated by f2c is essentially the same as it was in the original Fortran 77. However, this format string is now interpreted by the f2c run-time library routines `s_wsle`, `do_lio`, `do_fio`, etc.

The first stage of processing by the tools removes all format repetitions from the format string and changes it to an expanded C format character array. The processed C code includes the header file `nag_io.h`. This file contains the macro `LINE` and global index `i__ind`, which is used to access the appropriate C format strings at run-time.

The macro `LINE` controls end-of-record line-feeds, and monitors the progress through a given C format character array by updating `i__ind` after each item is output. The arguments to `LINE` are first the format length, then the number of items to output and lastly the value of `i__ind` if wrap-round of a Fortran format occurs.

More detailed information concerning `LINE` and its use is given in the next section.

After first stage processing

```
#include <nag_io.h>

static char *polfmt_99998[20] = {"%s", "%7.3f", "%s", "%7.3f", "%s", "%s", "%7.3f", "%s",
"%7.3f", "%s", "%s", "%7.3f", "%s", "%7.3f", "%s", "%s", "%7.3f", "%s", "%7.3f", "%s"};

printf(" Eigenvalues ");
printf("\n");
i__ind = 0;
i__2 = n;
for (i = 1; i <= n; ++i) {
    printf(polfmt_99998[i__ind], " (");LINE(20,1,0);
    printf(polfmt_99998[i__ind], wr[i-1]);LINE(20,1,0);
    printf(polfmt_99998[i__ind], ",");LINE(20,1,0);
    printf(polfmt_99998[i__ind], wi[i-1]);LINE(20,1,0);
    printf(polfmt_99998[i__ind], ")");LINE(20,1,0);
}
if (i__ind) printf("\n");
```

The second stage of processing aims to identify simple structures in the processed C code which permit a reduction in the number of calls to <stdio.h> routines. When this occurs individual C format strings in the character arrays are gathered together to form more complex format strings, and this allows an <stdio.h> routine to input/output several items per call. If during such compression all the information contained in a given C format character array is extracted then it becomes unnecessary and is deleted (this in fact occurs in the code fragment under consideration).

After second stage processing

```
#include <nag_io.h>

printf(" Eigenvalues ");
printf("\n");
i__2 = n;
i__ind = 0;
for (i = 1; i <= n; ++i) {
    printf(" %s%7.3f%s%7.3f%s", " (", wr[i-1], ", ", wi[i-1], ")");LINE(20,5,0);
}
if (i__ind) printf("\n");
```

The main considerations

f2c produces C code in which the input/output format string is essentially the same as it was in the original Fortran 77. Since Fortran format specifiers are not part of the C language their meaning has to be interpreted by a set of non-standard f2c run-time library routines. In order to obtain C code in which the standard input/output routines are used, it is necessary to replace the f2c run-time routines and format strings with their <stdio.h> equivalents.

The method used here is to convert the f2c generated format strings into character arrays whose elements are the appropriate <stdio.h> C format strings. Formats are then selected at run-time by using a global index to access the required array elements.

However, because the C routines contained within <stdio.h> have limited capability, certain Fortran 77 run-time features (such as P and colon format specifiers) cannot be implemented. In such cases the format specifier will be ignored and thus left out of the generated C format string. This means that the processed C code may produce output results that differ from the original Fortran – this should not be a problem since these differences will be well defined and easy to interpret.

The Fortran format specifiers that the tools can process correctly are “/ G F D L E X A I (. .) ’ ”, where each specifier can be used with its full descriptor which, in general, may involve a repetition count, field width and number of significant digits.

The main issues that need to be considered are

- End-of-record line-feeds
- Fortran format repetitions and paired brackets
- Wrap-round of a Fortran format
- Literal strings within formats

End-of-record line-feeds

In Fortran a WRITE statement generates an end-of-record line-feed either when there are no more items to output (and there is still format string left) or when there are more items to output and the end of the format string has been reached.

In the processed C code the header file `nag_io.h` is used to control line feeds. This file contains the global index `i__ind` which monitors the progress through a given C format character array. It also contains the macro `LINE` which is used to increment `i__ind` after each item is output.

The definition of `LINE` is

```
#define LINE(A,B,C) i__ind = i__ind + B; if (i__ind == A) {i__ind = C; printf("\n");}
```

where B is the increment, A is the format length and C is the position after “wrap-round” has occurred.

This is illustrated, for the previously declared format character array `polfmt_99999` in the following code fragment:

```
i__ind = 0;
printf(polfmt_99999[i__ind],x);LINE(20,1,0);
```

```
.  
.  
printf(polfmt_99999[i__ind],y);LINE(20,1,0);  
if (i__ind) printf("\n");
```

When the end of the format string is reached the condition `if(i__ind == A)` within `LINE` provides an end-of-record line-feed. If there is still sufficient format string remaining, the end-of-record line-feed will be provided by the terminal statement `if (i__ind) printf("\n");`

Fortran format repetitions and paired brackets

In Fortran (and in f2c generated C code) it is possible to apply an integer repetition to either an individual format or, using paired brackets, a range of formats. Since paired brackets can be nested a Fortran format can be quite complicated.

In C these features do not exist, and it is therefore necessary to remove all such format repetitions when creating C format character arrays.

This is illustrated below where the f2c format consists of both a paired bracket and an individual format repeated twice.

```
static char fmt_99998[] = "(f6.3,2(a,2f7.3))";
```

The equivalent C format character array, which consists of 7 character strings and contains no format repetitions, is as follows:

```
static char *polfmt_99998[7] = {"%6.3f", "%s", "%7.3f", "%7.3f", "%s", "%7.3f", "%7.3f"};
```

Wrap-round of a Fortran format

In Fortran 77 when the end of the format string is reached a new record begins. The format is then repeated from the first embedded left bracket, using its repetition if any. As previously mentioned, `LINE` is used to both set the array index after wrap-round and output the required end-of-record line-feeds. In the above format string the first embedded left bracket corresponds to the second array element, and so `C = 1`. This means that after the output of each item the macro would be called using `LINE(7,1,1)`.

Literal strings within formats

On many occasions literal character strings are embedded within Fortran format statements; for instance they permit a complex number consisting of real and imaginary parts `RE` and `IM` to be output as `"(RE,IM)"`. The Fortran 77 code to do this is

```

WRITE (NOUT,99998) RE, IM
99998 FORMAT (' (',F8.4,',',F8.4,')')
```

The Fortran code consists of three literal strings and two format specifiers. The equivalent C code is a two-element character array, which contains a combination of C format specifiers and literal characters.

In Fortran (ignoring colon specifiers) “trailing” string literals are output even when there are no more items left to output. The following method is therefore used to create elements of the C format character array.

The first array element can be made up of three items (if they are all present) :

- A leading string literal
- The equivalent C format specifier
- A trailing string literal

Subsequent array elements do not incorporate the leading string literals.

The processed C code then looks like

```

static char *polfmt_99998[2] = {" (%8.4f,","%8.4f)"};

printf(polfmt_99998[i__ind],re);LINE(2,1,0);
printf(polfmt_99998[i__ind],im);LINE(2,1,0);
```

A complicated format

The following code fragment illustrates how the tools process a complicated format, which involves both embedded character strings and “wrap-round” of the format string.

Original Fortran 77

```

WRITE (NOUT,99998) NMESH, ERMX, IERMX, IJERMX,
+ (I,IPMESH(I),MESH(I),I=1,NMESH)

99998 FORMAT (/ ' Used a mesh of ',I4,' points',/ ' Maximum error = ',
+ D10.2,' in interval ',I4,' for component ',I4,/' Mesh p',
+ 'oints:',/4(I4,'(',I1,')',D11.4))
```

The “raw output from f2c”

```
#include "f2c.h"
static char fmt_99998[] = "(/\002 Used a mesh of \002,i4,\002 points\002\
,\002 Maximum error = \002,d10.2,\002 in interval \002,i4,\002 for compone\
nt \002,i4,/\002 Mesh p\002,\002oints:\002,/4(i4,\002(\002,i1,\002)\002,d11\
.4))";

static cilist io___19 = { 0, 6, 0, fmt_99998, 0 };

s_wsfe(&io___19);
do_fio(&c__1, (char *)&nmesh, (ftnlen)sizeof(integer));
do_fio(&c__1, (char *)&ermx, (ftnlen)sizeof(doublereal));
do_fio(&c__1, (char *)&iermx, (ftnlen)sizeof(integer));
do_fio(&c__1, (char *)&ijermx, (ftnlen)sizeof(integer));
i__2 = nmesh;
for (i = 1; i <= i__2; ++i) {
    do_fio(&c__1, (char *)&i, (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&ipmesh[i - 1], (ftnlen)sizeof(integer));
    do_fio(&c__1, (char *)&mesh[i - 1], (ftnlen)sizeof(doublereal));
}
e_wsfe();
```

It can be seen that the f2c generated format string looks rather “messy”. This is because f2c not only uses the same (limited length) string literals as Fortran 77, but also tends to introduce extra line breaks into the format string. Another f2c “feature” is the use of the escape sequence `\002` to represent a format quote.

The simple structure of the generated C code allows the first four output statements to be compressed into a single `<stdio.h>` routine call. The remaining output is performed using a “for loop” and, because the formats contain embedded strings, further compression cannot take place. This means that the macro call `LINE(16,1,4)` is required after each item is output. (Note: It is now not necessary to declare the full array `polfmt`, since only part of it is accessed via the index `i__ind`. However, the declaration of the full array was not considered a significant disadvantage and no extra processing to reduce format array sizes was incorporated into the tools.)

The last statement `if (i__ind-4) printf("\n");` is used to provide an end-of-record line-feed if format “wrap-round” (to index position `i__ind = 4`) has not just occurred.

The processed code

```
#include <nag_io.h>

static char *polfmt_99998[16] = {"\n Used a mesh of %4i points\n Maximum error = ",
"%10.2e in interval ", "%4i for component ", "%4i\n\n Mesh points:\n", "%4i(", "%1i)",
"%11.4e", "%4i(", "%1i)", "%11.4e", "%4i(", "%1i)", "%11.4e", "%4i(", "%1i)", "%11.4e"};

i__2 = nmesh;
i__ind = 0;
```

```

printf("\n Used a mesh of %4i points\n Maximum error = %10.2e in interval %4i for \
component %4i\n\n Mesh points:\n", nmesh, erm, ierm, ijer);LINE(16,4,4);
for (i = 1; i <= i__2; ++i) {
    printf(polfmt_99998[i__ind], i);LINE(16,1,4);
    printf(polfmt_99998[i__ind], ipmesh[i-1]);LINE(16,1,4);
    printf(polfmt_99998[i__ind], mesh[i-1]);LINE(16,1,4);
}
if (i__ind-4) printf("\n");

```

Other considerations

So far no mention has been made of the Fortran READ statement, and it has been assumed that the format identifier associated with a Fortran WRITE statement is the integer label of a format statement. However, the format identifier can also be a character string giving the format description or an asterisk indicating list-directed output.

Also nothing has yet been said about the output of character string array sections and the use of character field widths.

Character string format identifiers

The following Fortran 77 code fragment is used to illustrate how character string format identifiers are processed.

Original Fortran 77

```
WRITE (*, '(1X,F6.3,2(3X,F7.3))') TSTART, (YSTART(L), L=1, NEQ)
```

The “raw output from f2c”

```

#include "f2c.h"
/* Fortran I/O blocks */
static cilist io__2 = { 0, 0, 0, "(1X,F6.3,2(3X,F7.3))", 0 };
io__2.ciunit = nout;
s_wsfe(&io__2);
do_fio(&c__1, (char *)&tstart, (ftnlen)sizeof(doublereal));
for (l = 1; l <= 2; ++l) {
    do_fio(&c__1, (char *)&ystart[l - 1], (ftnlen)sizeof(doublereal));
}
e_wsfe();

```

It can be seen that f2c has placed the format string directly into an I/O block structure of type cilist. This is in contrast to what happens when the format identifier is a format label (see the first example). Here a two stage process occurs: first a separate character array is generated for the format string and then this array is referenced within the I/O block.

After first stage processing

```
#include <nag_io.h>

static char *polfmt_S0[3] = {" %6.3f   ", "%7.3f   ", "%7.3f"};
i__ind = 0;
printf(polfmt_S0[i__ind], tstart); LINE(3,1,1);
for (l = 1; l <= 2; ++l) {
    printf(polfmt_S0[i__ind], ystart[l-1]); LINE(3,1,1);
}
if (i__ind-1) printf("\n");
```

Since the method of processing requires all formats to have associated arrays, the character array `polfmt_S0` is generated automatically. (The next array to be generated, if required, would be named `polfmt_S1` and so on.)

After second stage processing

```
#include <nag_io.h>
i__ind = 0;
printf(" %6.3f   ", tstart); LINE(3,1,1);
for (l = 1; l <= 2; ++l) {
    printf("%7.3f   ", ystart[l-1]); LINE(3,1,1);
}
if (i__ind-1) printf("\n");
```

Subsequent processing is now identical to that which occurs when the format identifier is a label. In this case compression has resulted in deletion of the character array `polfmt_S0`.

List-directed output

Several examples of list-directed output are given in the Appendix. Here the processed C code outputs floating-point numbers using the format `%12.6lf` and complex numbers using the statement `printf(" (%12.6lf, %12.6lf) ", comp.r, comp.i)`, where `comp` is a complex number. Integers are output with the format `%6ld`, and logical variables, which in C are TRUE if non-zero, are output using `printf("%6s", lvar ? "T" : "F")`, where `lvar` is the logical variable.

In Fortran a single `WRITE(*,*)` or `PRINT*` statement can be used to output all the elements of an array. When such statements are encountered during processing the tools generate auxiliary loops so that all the array elements can be printed. The example Fortran code in the Appendix illustrates this by using `PRINT*` to output the logical vector `LOGIC`, and the corresponding processed C code uses the loop variable `i__ind1`, defined in `nag_io.h`, to access individual array elements.

Reading data

At present the tools only process list-directed input and several examples of this are given in the Appendix.

Since Fortran input/output is record based, when the data specified by a `READ` statement has been input all further data on the same line (record) is ignored. It is because of this that the trailing comments in the data file given in the Appendix are ignored. This is illustrated by the statement `READ(*,*) N, TITLE` which is used to input the variable `N` and the character array `TITLE`, but ignores the text “:Value of `N` and `TITLE`”. In the corresponding processed C code the data is input using `scanf("%ld",&n)` and `scanf(" '%3s'",&title)`. The statement `gets(i_line)`, where `i_line` is a character buffer defined in `nag_io.h`, is used to skip any trailing comments.

Character string array sections and character field widths

Fortran 77, in contrast to C, allows character string array sections to be referenced using colon notation. For example `WRITE(*,*) STR(1:5)` outputs the sub-string consisting of the first five characters of the array `STR`. The processed C code translates the above statement by using the `<stdio.h>` function `strncpy` and the character buffer `i_line`, defined in `nag_io.h`. The situation is rather more complicated for situations in which a character output field width is specified. If a character (sub)string of length L is output using the format $A\omega$ then $\min(\omega,L)$ characters are output. These are right justified, with leading spaces if necessary, in the field.

This behaviour can be replicated in C by using the statement `printf("%\omega.*s",\min(\omega,L))`. The processed C code implements this by splitting each format string into two parts which are separated by a `NULL`. The first part is the usual format string whilst the second part is the character representation of the field width, ω . (If a Fortran format contains a character field width then the processed C code assumes that all character formats have an associated field width, and if they do not assigns the width a value of 999.)

The macro `AFIELD(id, l, STR)` calculates $\min(\omega,L)$ and creates the appropriate sub-strings by using the character buffer `i_line`. Here `id` is the identifier associated with the C format string `polfmt_id`, `l` is the length of the (sub)string, and `STR` is a pointer to the string.

An example of the processing of character field widths and array sections is given below.

Original Fortran 77

```
CHARACTER *30 G30

G30 = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'

DO 20 I = 2, 20, 2
  WRITE(*,90) G30(1:I-2),G30(1:I-1),G30(1:I+3),G30(1:I+4)
20 CONTINUE
WRITE(*,'(49X,A3)') G30, G30
```


Running the tool set

UNIX scripts are used to run the tools; they can process Fortran 77 files containing both main programs and multiple subroutines/functions.

NAG has made available a public domain version of the tool set which can be accessed using the URL <http://www.nag.co.uk:70/1h/public>

The area contains further instructions concerning their operation, and a UNIX script to run them.

Acknowledgements

The author would like to thank S Datardina for his help in testing the tools and J Du Croz for his comments.

References

[1] B W Kernighan and D M Ritchie *The C Programming Language*, Prentice Hall Software Series, 1988.

[2] S I Feldman, D M Gay, M W Maimone, and N L Schryer *A Fortran to C Converter*, Computing Science Technical Report, No.149, AT&T Bell Laboratories, Murray Hill NJ, March 1993.

[3] G F Levy *Improving the output of the Fortran to C translator, f2c*, Software Practice & Experience, Vol 25, No.2, pp 217–227, Feb 1995.

[4] *The NAG C Library, Mark 4*, NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford, UK, 1996.

[5] M E Lesk *Lex - A Lexical Analyzer Generator*, Computing Science Technical Report, No.39, AT&T Bell Laboratories, Murray Hill NJ, Oct 1975.

[6] S C Johnson *Yacc - Yet Another Compiler*, Computing Science Technical Report, No.32, AT&T Bell Laboratories, Murray Hill NJ, July 1975.

[7] J R Levine, T Mason and D Brown *Lex & Yacc*, O' Reilly & Associates, Inc. 1992.

[8] A V Aho, R Sethi and J D Ullman *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

[9] *The NAG Fortran Library, Mark 17*, NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford, UK, 1996.

Appendix

Two complete Fortran subroutines are given to illustrate how a file is processed. The subroutine INPUT uses list-directed READ(*,*) statements to input data, and the subroutine OUTPUT uses list-directed WRITE(*,*) and PRINT* statements for output. The data file is also shown, and contains an integer, a character constant, a complex matrix, and a logical vector. It also contains a heading and trailing comments which are to be skipped.

The original Fortran source code

```
SUBROUTINE INPUT(N,A,LDA,LOGIC)

  INTEGER          N,LDA
  COMPLEX*16       A(LDA,*)
  LOGICAL          LOGIC(*)
  CHARACTER*8      TITLE
  INTEGER          I,J

  READ (*,*)
  READ (*,*) N, TITLE
  WRITE(*,('Example for '''' ''',A,''' '''' ''',/30(''=''),/)) TITLE
  READ (*,*) ((A(I,J),J=1,N),I=1,N)
  READ (*,*) (LOGIC(I),I=1,N)
  RETURN
END

SUBROUTINE OUTPUT(N,A,LDA,LOGIC)
  INTEGER          N,LDA
  COMPLEX*16       A(LDA,N)
  LOGICAL          LOGIC(N)
  INTEGER I,J

  PRINT*, 'LOGICAL VECTOR LOGIC = ', LOGIC
  DO 20 I = 1, N
    DO 30 J = 1, N
      WRITE (*,*) A(I,J)
30    CONTINUE
20    CONTINUE

  RETURN
END
```

The data file

First line of data file

```
2 'APPENDIX' :Value of N and TITLE
(1.0,1.0) (-1.0,1.0)
(1.4,1.6) (-3.0,2.1) :End of complex matrix A
T F T T :End of logical vector LOGIC
```

The "raw output from f2c"

```
#include "f2c.h"
/* Table of constant values */
static integer c__3 = 3;
static integer c__1 = 1;
static integer c__9 = 9;
static integer c__7 = 7;
static integer c__8 = 8;

/* Subroutine */ int input_(n, a, lda, logic)
integer *n;
doublecomplex *a;
integer *lda;
logical *logic;
{
    /* System generated locals */
    integer a_dim1, a_offset, i__1, i__2;

    /* Builtin functions */
    integer s_rsle(), e_rsle(), do_lio(), s_wsfe(), do_fio(), e_wsfe();

    /* Local variables */
    static integer i, j;
    static char title[8];

    /* Fortran I/O blocks */
    static cilist io___4 = { 0, 5, 0, 0, 0 };
    static cilist io___5 = { 0, 5, 0, 0, 0 };
    static cilist io___7 = { 0, 6, 0, "('Example for ' ' ,A,' ' ' ,/30('='),\
/) ", 0 };
    static cilist io___8 = { 0, 5, 0, 0, 0 };
    static cilist io___11 = { 0, 5, 0, 0, 0 };

    /* Parameter adjustments */
    --logic;
    a_dim1 = *lda;
    a_offset = a_dim1 + 1;
    a -= a_offset;

    /* Function Body */
    s_rsle(&io___4);
    e_rsle();
    s_rsle(&io___5);
    do_lio(&c__3, &c__1, (char *)&(*n), (ftnlen)sizeof(integer));
    do_lio(&c__9, &c__1, title, 8L);
    e_rsle();
    s_wsfe(&io___7);
    do_fio(&c__1, title, 8L);
    e_wsfe();
    s_rsle(&io___8);
    i__1 = *n;
    for (i = 1; i <= i__1; ++i) {
        i__2 = *n;
        for (j = 1; j <= i__2; ++j) {
            do_lio(&c__7, &c__1, (char *)&a[i + j * a_dim1], (ftnlen)sizeof(
                doublecomplex));
        }
    }
    e_rsle();
}
```

```

s_rsle(&io___11);
i__2 = *n;
for (i = 1; i <= i__2; ++i) {
    do_lio(&c__8, &c__1, (char *)&logic[i], (ftnlen)sizeof(logical));
}
e_rsle();
return 0;
} /* input_ */

/* Subroutine */ int output_(n, a, lda, logic)
integer *n;
doublecomplex *a;
integer *lda;
logical *logic;
{
    /* System generated locals */
    integer a_dim1, a_offset, logic_dim1, i__1, i__2;

    /* Builtin functions */
    integer s_wsle(), do_lio(), e_wsle();

    /* Local variables */
    static integer i, j;

    /* Fortran I/O blocks */
    static cilist io___12 = { 0, 6, 0, 0, 0 };
    static cilist io___15 = { 0, 6, 0, 0, 0 };

    /* Parameter adjustments */
    logic_dim1 = *n;
    --logic;
    a_dim1 = *lda;
    a_offset = a_dim1 + 1;
    a -= a_offset;

    /* Function Body */
    s_wsle(&io___12);
    do_lio(&c__9, &c__1, "LOGICAL VECTOR LOGIC = ", 23L);
    do_lio(&c__8, &logic_dim1, (char *)&logic[1], (ftnlen)sizeof(logical));
    e_wsle();
    i__1 = *n;
    for (i = 1; i <= i__1; ++i) {
        i__2 = *n;
        for (j = 1; j <= i__2; ++j) {
            s_wsle(&io___15);
            do_lio(&c__7, &c__1, (char *)&a[i + j * a_dim1], (ftnlen)sizeof(
                doublecomplex));
            e_wsle();
        }
    }
    /* L30: */
    }
    /* L20: */
    }
    return 0;
} /* output_ */

```

The final processed C code

```
#include <nag_io.h>

/* Subroutine */ int input_(n, a, lda, logic)
integer *n;
doublecomplex a[];
integer *lda;
logical logic[];
{
    integer i__1;
    static integer i, j;
    static char title[8];
#define LOGIC(I) logic[(I)-1]
#define A(I,J) a[(I)-1 + ((J)-1)* ( *lda)]
    gets(i__line);
    scanf("%ld",&(*n));
    scanf(" '%8s'",&title);
    gets(i__line);
    printf("Example for ' %s ' \n===== \n\n",title);
    i__1 = *n;
    for (i = 1; i <= i__1; ++i) {
        for (j = 1; j <= *n; ++j) {
            scanf(" ( %lf, %lf)",&A(i,j).r,&A(i,j).i);
        }
    }
    gets(i__line);
    for (i = 1; i <= *n; ++i) {
        scanf("%1s",i__line);LOGIC(i) = ((*i__line == 'T') || (*i__line == 't'));
    }
    gets(i__line);
    return;
}

/* Subroutine */ int output_(n, a, lda, logic)
integer *n;
doublecomplex a[];
integer *lda;
logical logic[];
{
    integer logic_dim1;
    static integer i, j;
    logic_dim1 = *n;
#define LOGIC(I) logic[(I)-1]
#define A(I,J) a[(I)-1 + ((J)-1)* ( *lda)]
    printf(" LOGICAL VECTOR LOGIC = ");
    for (i__ind1 = 0; i__ind1 < logic_dim1; ++i__ind1) {
        printf("%6s", LOGIC(i__ind1+1) ? "T" : "F");
    }
    printf("\n");
    for (i = 1; i <= *n; ++i) {
        for (j = 1; j <= *n; ++j) {
            printf(" (%12.6lf, %12.6lf) ",A(i,j).r,A(i,j).i);
            printf("\n");
        }
    }
    return;
}
```