Chapter 2

Developing a C* Program

A C* program can consist of:

- standard serial C code
- C* code; see Section 2.1
- header files; see Section 2.2
- calls to the CM timing utility; see Section 2.3
- calls to CM library functions; see Section 2.4
- calls to CM Fortran subroutines; see Section 2.5

In addition, C* programs can be called from CM Fortran programs. See Section 2.6.

2.1 C* .cs Files

All C* code must appear in files with the suffix **`.cs`**. C* code consists of any of the extensions that C* adds to Standard C; see the *C* Programming Guide* for a discussion of these extensions. Standard C code can appear in either **`.c`** or **`.cs`** files; putting it in **`.c`** files speeds up compilation, as discussed in Section 3.2.

2.1.1 C* Keywords

C* adds these new keywords to Standard C:

```
allocate_detailed_shape
allocate_shape
bool
boolsizeof
current
dimof
everywhere
overload
pcoord
physical
positionsof
rankof
shape
shapeof
where
with
```

C* code must not use these words, except as prescribed in the definition of the language.

2.1.2 Reserved Identifiers

Identifiers beginning with **CM** are reserved for use by the Connection Machine system software. Do not create identifiers beginning with **CM** in your programs.

2.2 Header Files

C* substitutes its own header files for some Standard C header files. To find out the location of these and other C* header files at your site, issue this command:

```
% cs -cm5 -dirs
```

The files are in the include directory listed in response to this command; see Section 3.4.4 for more information. Appendix A contains the man pages for some of these files.

C* accepts other Standard C libraries and associated header files, as long as they are ANSI-compliant. Exceptions include:

- header files that use C* keywords; see Section 2.2.4

- header files containing syntax that is not accepted by the C* compiler (for example, **<a.out.h>**)
- header files that depend on internal compiler support not provided by C* (for example, **<alloc.h>**)

In addition to its versions of some Standard C header files, C* includes these header files:

- **<cscomm.h>**, which is the header file for the communication functions described in Part III of the *C* Programming Guide*.
- **<csshape.h>**, which is the header file for a group of routines used in obtaining information about shape layout, as described in the *C* Programming Guide.*
- **<cstable.h>**, which provides routines used in C*'s table lookup utility, also described in the *C* Programming Guide.*
- **<csfort.h>**, which you use when calling CM Fortran routines from a C* program. See Section 2.5.

2.2.1 The <math.h> File

The C* version of **<math.h>** declares all the functions in the UNIX serial math library and extends all ANSI serial functions with parallel overloadings. No special library is required to use these functions.

2.2.2 The <stdlib.h> File

The file **<stdlib.h>** contains scalar and parallel declarations of the function **abs**, **rand**, and **srand**; the parallel versions of **rand** and **srand** are named **prand** and **psrand**. The file also contains the declarations of **palloc**, **pfree**, and **deallocate_shape**, which are described in the *C* Programming Guide*. No special library is required to use these functions.

Note that **prand** returns a parallel variable in which each element is sampled independently from the uniform distribution.

2.2.3 The <string.h> File

The file **<string.h>** contains parallel declarations of the functions **memcpy**, **memmove**, **memcmp**, and **memset**. In addition, it contains declarations of boolean versions of these functions, called **boolcpy**, **boolmove**, **boolcmp**, and **boolset**. Use the boolean versions to maintain source compatibility with CM-200 C*.

The boolean versions take pointers to **bool**s for arguments and return pointers to **bool**s (except for **boolcmp**). If you are dealing with arguments that are not **bool**s, you must cast them to be pointers to **bool**s. Also, note that the size argument for **memcpy** and related functions is in terms of **char**s, while the size argument for the boolean versions is in terms of **bool**s.

For example, in this code fragment, both **memcpy** and **boolcpy** copy **source** to **dest**:

```
#include <string.h>
/* ... */
int:current dest[2], source[2];

memcpy(dest, source, 2*sizeof(int:current) );
boolcpy( (bool:current *)dest,
    (bool:current *)source, 2*boolsizeof(int:current) );
```

2.2.4 Header Files and C* Keywords

A difficulty can occur when you want to include a standard header file that also makes use of a C* keyword. For example, the X Window System header file **<X11/Xlib.h>** uses the C* keyword **current** as the name of a structure member. Including this file would result in a syntax error. In such a situation, you can do this:

```
#define current Current
#include <X11/Xlib.h>
#undef current
```

This redefines **current** to be **Current** while **<X11/Xlib.h>** is being included, then undefines it. Of course, if you subsequently want to use the **<X11/Xlib.h>** structure member in your program, you must refer to it as **Current**.

2.3 Timing a Program

CMOST provides a timing utility that you can use in a C* program to determine how much time any part of a program takes to execute on the nodes.

The timing utility has these features:

- A timer calculates total time the processing nodes were active, with microsecond precision.
- Multiple timers can be active at the same time.
- Timers can be nested. This allows you, for example, to start one timer that will time the entire program, while using other timers to determine how different parts of the program contribute to the overall time.

You can have up to 64 timers running in a program. An individual timer is referenced by an unsigned integer (from 0 to 63 inclusive) that is used as an argument to the timing instructions. Instructions with the same number as an argument affect only the timer with that number.

To start timer 0, for example, put a call to the **CM_timer_start** routine in your program, using 0 as an argument:

```
CM_timer_start(0);
```

You can subsequently stop timer 0 by calling the **CM_timer_stop** routine later in your program:

```
CM_timer_stop(0);
```

This function stops the timer and updates the values for total elapsed time and total node idle time being held by the timer. You can subsequently call **CM_timer_start** again to restart timer 0; the timing starts at the values currently held in the timer. This is useful for measuring how much time is spent in a frequently called subroutine. The timer keeps track of the number of times it has been restarted.

You can start or stop other timers while timer 0 is running; each timer runs independently.

To get the results from timer 0, call this routine after you have called **CM_timer_stop**:

```
CM_timer_print(0);
```

CM_timer_print prints information like this to your standard output:

```
Starts: 1
CM Elapsed time: 27.7166 seconds
CM busy time: 23.1833 seconds
```

These routines return specific information from the timer for use in a program:

- **CM_timer_read_starts** returns an integer that represents the number of times the specified timer has been started.
- **CM_timer_read_elapsed** returns a double-precision value that represents the total elapsed time (in seconds) for the specified timer. "Elapsed time" refers to process time, not wall-clock time.
- **CM_timer_read_cm_idle** returns a double-precision value that represents the total CM idle time (in seconds) for the specified timer.
- **CM_timer_read_cm_busy** returns a double-precision value that represents the total time (in seconds) the CM was busy for the specified timer. CM busy time is the total elapsed time minus the CM idle time.
- **CM_timer_read_run_state** returns 1 if and only if the specified timer is running. Otherwise, the routine returns 0.

If you use any of these **CM_timer_read_***xxx* routines, include the file **<cm/timers.h>**.

In addition, **CM_timer_set_starts** takes a timer number and an integer as arguments. It sets the number of starts for the specified timer to the specified value. This is useful if you want to write a function that can query a running timer without changing the number of starts. Not changing the number of starts is important if you want to know how many times a large chunk of code was called, but you also want to get sub-timings within that block.

To clear the values maintained by a timer, call **CM_timer_clear**. For example, to clear the value maintained by timer 0, put this call in your program:

```
CM_timer_clear(0);
```

This zeroes the total elapsed time, the total node idle time, and the number of starts for this timer.

NOTE: For compatibility with CM-200 C*, CM-5 C* also provides versions of the timing routines that begin with **CMC** instead of **CM**. If your program contains the **CMC** versions of the routines, make sure you include the file **<cstimer.h>.**

2.3.1 Hints on Using the Timing Utility

The elapsed time reported by a timer includes time when the process is swapped out on the partition manager. The more processes that are running, the more distorted this figure will be. Therefore, we recommend that you use a partition manager that is as unloaded as possible.

If you can't guarantee that you will have exclusive use of the CM-5, try to run the process several times; the minimum elapsed time reported will be the most accurate.

In addition, we recommend that you avoid stopping a process that is being timed.

Note that the inclusion of calls to the timer functions can change the generated code somewhat, and therefore itself affect performance.

Finally, note that if you are using Prism to analyze the performance of a program that includes timer calls, Prism performance data will include the overhead assigned to these calls; thus, the elapsed time reported by Prism will be somewhat greater than the elapsed time reported by the timing routines.

2.4 Calling CM Libraries

You can call routines from the standard CM libraries from within a C* program. Specifically:

- Call routines from the CMX11 library to display images on an X Window System. See Section 2.4.1.
- Call routines from the CM/AVS library to create modules that you can use in a distributed visualization application within the AVS visual programming environment. See Section 2.4.2.
- Call routines from the CMFS library to perform standard I/O functions — for example, reading data into the processing nodes from a scalable disk array or other I/O device. See Section 2.4.3.
- Call routines from the CMMD library to do node-level message passing on CM-5s with vector units. See Section 2.4.4.

- Call routines from the Connection Machine Scientific Software Library (CMSSL) to perform data parallel versions of standard mathematical operations such as matrix multiply and Fast Fourier Transform. You currently call the CM Fortran versions of CMSSL routines, as described in Section 2.5.

NOTE: These libraries are not available if you compile with the **`-cmsim`** option to run on a Sun-4.

2.4.1 CMX11

You can make calls to the CMX11 library from C*. This library provides routines that allow the transfer of parallel data between the CM and any X11 terminal or workstation. See the *CMX11 Reference Manual* for information on the C* interface.

2.4.2 CM/AVS

CM/AVS is an extension of the Advanced Visualization System (AVS) to the CM-5. AVS provides a graphic programming environment in which you can build a distributed visualization application. CM/AVS enables such an application to operate on data that is distributed on CM-5 nodes and to interoperate with data from other sources.

You can write your own modules for CM/AVS in C*. You can then combine these modules with standard CM/AVS and AVS modules to create your visualization application. See the *CM/AVS User's Guide* for information on how to call the CM/AVS routines in a C* program.

2.4.3 I/O

CM-5 C* provides synchronous parallel I/O support to the DataVault and scalable disk array via an interface to the CMFS functions; I/O to other devices is not currently supported. The interface is the same for CM-5s with and without vector units. For complete information on these functions, see the documentation for the CM-5's I/O system..

Other interfaces for I/O may exist in the future.

The CMFS I/O library is not available if you compile with the **-cmsim** option to run on a Sun-4.

Note these points in using the current interface:

- Do not include **<cm/paris.h>** or **<cm/cmtypes.h>**.
- You must include **<cm/cmfs.h>**.
- Link with **-lcmfs_cs -lcmfs_cm5**, in that order.
- These calls are specific to the CM-2 and are not supported in the CM-5 CMFS library:

```
CMFS_cm_to_standard_byte_order
CMFS_convert_vax_to_ieee_float
CMFS_convert_ieee_to_vax_float
CMFS_partial_read_file_always
CMFS_partial_write_file_always
CMFS_transpose_always
CMFS_transpose_record_always
CMFS_file_geometry
CMFS_twuffle_to_serial_order_always_1L
CMFS_twuffle_from_serial_order_always_1L
```

- There are C*-specific versions of **CMFS_read_file_always**, **CMFS_read_file**, **CMFS_write_file_always**, and **CMFS_write_file**. The declarations (from **<cm/cmfs.h>**) are:

```
overload CMFS_read_file, CMFS_read_file_always;
overload CMFS_write_file, CMFS_write_file_always;

int CMFS_read_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_read_file_always (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file (int fd, void:void *dest,
                    int bytes_per_position);
int CMFS_write_file_always (int fd, void:void *dest,
                    int bytes_per_position);
```

  These interfaces provide basic compatibility with CM-200 C* code that calls CMFS.

The functions are called with pointers to parallel variables. A pointer to a parallel variable of any type may be used. The specified length may be any number of bytes, but performance is significantly diminished when the length is not a multiple of four bytes. See below for a further discussion of I/O performance.

- The lengths passed to and returned by these functions are always in bytes. For the C* interface, they indicate the number of bytes read or written in each position of the parallel variable. Note that on the CM-2/200 the **`CMFS_read_file`** and **`CMFS_write_file`** functions take bit lengths, and that in either case **`boolsizeof`** should be used to specify the length; this will make the program portable.
- There is currently no difference between the regular and the "always" versions of these functions. This is a temporary restriction. Users should only use the "always" versions until this restriction is lifted.
- Streaming and buffered I/O are not supported.
- You can use standard UNIX I/O routines to do serial I/O on CMFS files if the CMFS file system is NFS-mounted. See the CM-5 I/O documentation for more information.
- The total size of a file on a CMFS file system (that is, on the DataVault) will always be rounded up to be a multiple of 512 bytes.
- I/O performance may be significantly diminished if any of the following is true:
  - The size specified to the CMFS functions is not a multiple of 4 bytes.
  - The total amount of data being read or written is not a multiple of 16 bytes on the SDA, or 512 bytes on the DataVault.
  - The file position is not on a 4-byte boundary.
  - The parallel data passed to the CMFS function is an address that is not on a 4-byte boundary (for example, when the pointer points to a member of a parallel structure).
- The **`CMFS_lseek`** routine when called from C* seeks into a file the number of bytes you specify multiplied by the number of positions in the current shape. The routine **`CMFS_serial_lseek`** seeks an absolute number of bytes into a file.

2.4.4 CMMD

You can call routines in the CMMD communication library from C*, as of CMMD Version 3.0. For complete information, see the *CMMD User's Guide* and *CMMD Reference Manual.*

To call CMMD routines, you must compile your C* program with the **`-node`** and **`-cmmd_root`** options; see Section 3.4.8.

2.5 Calling CM Fortran

You can call CM Fortran subroutines from within a C* program. This section describes how. See Section 3.6 for a discussion of how to link in the CM Fortran program and other required files.

2.5.1 Overview

To call a CM Fortran subroutine, do the following:

- Include the file **`<csfort.h>`**.
- The name of the C* main routine must be **`MAIN_()`**.
- Use the function **`CMC_CALL_FORTRAN`** to call one or more CM Fortran subroutines. You must convert the subroutine name to lowercase and add an underscore to the end of it.
- To pass a parallel variable to a subroutine, create a scalar variable of type **`CMC_descriptor_t`**. Call the function **`CMC_wrap_pvar`**, with a pointer to the parallel variable as an argument, and assign the result to the scalar variable you created. Pass this scalar variable to the CM Fortran subroutine when you call it via **`CMC_CALL_FORTRAN`**.
- Pass scalar variables to a CM Fortran subroutine by reference.
- After you are finished with a descriptor, free it by calling **`CMC_free_desc`** with the scalar variable as an argument.

2.5.2 In Detail

Include File

As mentioned in the overview, you must include the file **<csfort.h>** if your program includes a call to a CM Fortran subroutine.

Calling the Subroutine

To call a CM Fortran subroutine, use this syntax:

```
CMC_CALL_FORTRAN(subroutine_(args), ... );
```

where:

subroutine is the name of the subroutine. It must be in lowercase (even if the original subroutine name is in uppercase), and you must add an underscore to the end of the subroutine name.

args are the subroutine's arguments, if any.

To call multiple subroutines, separate them with commas within the argument list. For example:

```
CMC_CALL_FORTRAN(subroutine1_(), subroutine2_());
```

The subroutine is not constrained by the current shape or the context as established by the C* program. When the call to **CMC_CALL_FORTRAN** returns, however, both the shape and the context are what they were before the function was called.

What Kinds of Variables Can You Pass?

You can pass both parallel and scalar variables as arguments to a CM Fortran subroutine. The parallel variables you pass can be of any shape. The variables can be of these standard types:

```
signed int
signed long int
float
double
long double
```

In addition, **`<csfort.h>`** provides **`typedef`**s for two new types: **`CMC_complex`** and **`CMC_double_complex`**. The **`typedef`**s are defined as follows:

```
typedef struct{float real, imag;} CMC_complex;
typedef struct{double real, imag;} CMC_double_complex;
```

Use these types to pass variables that can be treated as complex numbers by CM Fortran.

Passing Parallel Variables

A two-step process is required to pass a C* parallel variable to a CM Fortran subroutine.

First, declare a scalar variable of type **`CMC_descriptor_t`**. For example:

```
CMC_descriptor_t desc_a;
```

Next, make this variable a descriptor for the parallel variable by calling the function **`CMC_wrap_pvar`**, with a pointer to the parallel variable as its argument, and assigning the result to the scalar variable. For example, if **`p1`** is the parallel variable you want to pass, call the **`CMC_wrap_pvar`** function as follows:

```
desc_a = CMC_wrap_pvar(&p1);
```

You can wrap a parallel variable of any shape.

You can then pass the descriptor to the CM Fortran subroutine. For example:

```
CMC_CALL_FORTRAN(subroutine_(desc_a));
```

The descriptor stores the address of the parallel variable, and the parallel variable is passed by reference in this way. The CM Fortran subroutine can then operate on the parallel variable referenced by the descriptor.

C* code can operate on the parallel variable even after it has been wrapped.

You can reuse a descriptor in a program, but first you must free it; see below.

One restriction on passing parallel variables: You cannot pass a member of a parallel structure or an element of a parallel array to CM Fortran. Only simple parallel variables can be passed.

Passing Scalar Variables

Pass scalar variables to a CM Fortran subroutine by reference. For example:

```
int s1;

CMC_CALL_FORTRAN(subroutine_(&s1));
```

Freeing the Descriptors

When you are through using a descriptor, free it by calling **CMC_free_desc** with the descriptor as the only argument. For example:

```
CMC_free_desc(desc_a);
```

You can free a descriptor to a parallel variable of any shape.

An Example

Below is a C* program that calls a CM Fortran subroutine.

```
#include <stdio.h>
#include <csfort.h>

shape [16384]S;
CMC_descriptor_t desc_a;
int s1;
int:S p1;

MAIN_()
{
  with (S) {
    s1 = 1;
    p1 = 1;
    desc_a = CMC_wrap_pvar(&p1);

    CMC_CALL_FORTRAN(fortran_op_(desc_a,&s1));

    CMC_free_desc(desc_a);
    printf("Result for last position is %d\n",
           [16383]p1);
    }
}
```

And here is the simple CM Fortran subroutine it calls:

```
subroutine fortran_op(a,s)
integer a(16384)
integer s

a = a + s

return
end
```

In the future, we hope to provide a more transparent interface to CM Fortran. To minimize recoding when this interface is available, we recommend that you call the subroutine as if you were calling it directly, then use a stub routine to provide the correct syntax to make it work now. The example in Section 2.6.5 shows how to do this.

2.6 Writing Functions that Are Callable from CM Fortran

The previous section described how to call a CM Fortran routine from C*; this section describes how to write a CM-5 C* routine that can be called from a CM Fortran program. See Section 3.7 for information on compiling the CM Fortran program.

Functions callable from CM Fortran must include the header file **`<csfort.h>`**.

2.6.1 Names

Name the function according to Fortran conventions. Specifically:

- The name must end with an underscore (_).
- Any alphabetic characters in the name must be lowercase.

In the CM Fortran program that calls the function, the alphabetic characters may be either upper- or lowercase (since Fortran is not case-sensitive). The trailing underscore must be omitted.

2.6.2 Shapes

A C* shape and a CM Fortran "geometry" (that is, the shape and layout of a CM array) are not exactly the same, since C* shapes include context information as well as information on extents, rank order, and layout. Multiple shapes can share the same geometry.

Use the function **CMC_allocate_shape_from_desc** to dynamically allocate a shape in a C* function to be called from CM Fortran. It takes as an argument a descriptor for a CM array; see the next section. It returns a shape whose geometry is the same as that of the CM array, and whose context initially selects all elements of the geometry.

You must free this shape via the **deallocate_shape** routine before the C* function returns.

2.6.3 Parallel Arguments

Parallel variables are the equivalent of CM arrays in CM Fortran. The C* function must declare CM arrays incoming from the CM Fortran program with the type **CMC_descriptor_t**; this is a *descriptor* for the CM array. The C* function then turns the descriptors into pointers using this function:

```
void:void *CMC_unwrap_pvar(CMC_descriptor_t, shape);
```

(This is the opposite of the **CMC_wrap_pvar** function described in Section 2.5.2.) This function returns a pointer to a parallel variable, given an array descriptor and a shape. The geometry of the array must match the geometry of the shape, or a run-time error is signaled. A function to check that the geometries match is:

```
bool CMC_same_geometry(CMC_descriptor_t, shape);
```

Typically, called functions will create a single shape per geometry, but this interface allows otherwise for flexibility.

2.6.4 Scalar Arguments

All variables are passed by reference in CM Fortran, so the C* function must receive pointers to each of its scalar arguments. The data-type correspondence between CM Fortran and C* is shown in Table 1.

Table 1. Data type correspondence between CM Fortran and C*.

CM Fortran	C*
`REAL`	`float`
`DOUBLE PRECISION`	`double`
`COMPLEX`	`CMC_complex_t`
`DOUBLE COMPLEX`	`CMC_double_complex_t`
`CHARACTER`	`char *`

Character arguments are not guaranteed to be null-terminated, as C often expects. The lengths of all character arguments are appended to the end of the argument list (these lengths are passed by value, not by reference).

`LOGICAL`, **`INTEGER`**, and **`DOUBLE PRECISION`** values are returned by value just as in CM Fortran. CM Fortran turns functions that return complex values into subroutines that pass a pointer to a place to write the result. Functions that return character values are turned into calls whose first argument points to a character variable in which to place the result, and whose second argument is the length (passed by value).

This is a CM Fortran program that passes scalar values to a C* program:

```
LOGICAL l
INTEGER i
REAL r
DOUBLE PRECISION d
COMPLEX c
DOUBLE COMPLEX z
CHARACTER*5 ch

l = .TRUE.
i = 2
r = 3.0
d = 4.0
c = (5.0,-5.0)
```

```
z = (6.0,-6.0)
ch = "abcde"

PRINT *,"Main Program (CMF)"
PRINT *,'l = ',l
PRINT *,'i = ',i
PRINT *,'r = ',r
PRINT *,'d = ',d
PRINT *,'c = ',c
PRINT *,'z = ',z
PRINT *,'ch = "',ch,'"'
CALL CMF2C(l,i,r,d,c,z,ch)

END
```

And here is a C* program that it calls. (Note that there is nothing parallel about this program; it merely uses the C* header file.)

```
#include <stdio.h>
#include <csfort.h>

void cmf2c_(lp, ip, rp, dp, cp, zp, chp, chl)    /* fn name
                         lowercased & underscore appended */
int *lp;
int *ip;
float *rp;                         /* all args are pointers */
double *dp;
CMC_complex_t *cp;
CMC_double_complex_t *zp;
char *chp;
int chl;                      /* last arg for string length */
{
    int i;
    float r;
    double d;

    putchar('\n');
    printf("CMF2C (C)\n");
    printf("l = %c\n", (*lp) ? 'T' : 'F');
    printf("i = %d\n", *ip);
    printf("r = %f\n", *rp);
    printf("d = %f\n", *dp);
    printf("c = (%f,%f)\n", cp->real, cp->imag);
    printf("d = (%f,%f)\n", zp->real, zp->imag);
    printf("s = %.*s\n", chl, chp);
    putchar('\n');

    /* modify some args */
    *lp = !(*lp);
    cp->real = -cp->real;
```

```
        cp->imag = -cp->imag;
        zp->real = -zp->real;
        zp->imag = -zp->imag;
        chp[0] = ' ';

    }
```

2.6.5 An Example

Here is a simple CM Fortran program:

```
REAL, ARRAY(7) :: J, I
REAL f
J = I + f
CALL CSTAR_FUNCTION(J,I)
END
```

The C* program below both calls this CM Fortran program and contains the C* function that the CM Fortran program in turn calls:

```
MAIN_()
{
    shape [7]s;
    float:s i, j;
    float x;
    int n;

    with (s)
         i = pcoord(0);
    CMF_ROUTINE(&j, &i, 1.0);
    for(n=0; n < positionsof(s); n++) printf("%f ", [n]i);
         printf("\n");
    for(n=0; n < positionsof(s); n++) printf("%f ", [n]j);
         printf("\n");
}


/* Eventually, the following stub will automatically be gener-
ated by the compiler given the appropriate declaration of
CMF_ROUTINE. For now, we need to write it by hand. */

#include <csfort.h>

CMF_ROUTINE(jp, ip, f)
float:void *jp, *ip;
float f;
{
```

```
    CMC_descriptor_t jp_desc, ip_desc;

    jp_desc = CMC_wrap_pvar(jp);
    ip_desc = CMC_wrap_pvar(ip);
    CMC_CALL_FORTRAN(cmf_routine_(jp_desc, ip_desc, &f));
    CMC_free_desc(jp_desc);
    CMC_free_desc(ip_desc);
}



/* This is an example of a C* function to be called by CMF.
Again, this is something that will eventually be handled
automatically. The above  #include <csfort.h> is necessary for
this as well. */

void cstar_function_(CMC_descriptor_t ip_desc,
CMC_descriptor_t jp_desc)
{
    shape s = CMC_allocate_shape_from_desc(ip_desc);
    float:s *ip = CMC_unwrap_pvar(ip_desc, s);
    float:s *jp = CMC_unwrap_pvar(jp_desc, s);

    *ip = -*ip;
    *jp = -*jp;

    deallocate_shape(&s);
}
```