When Translated Code Won't Run

It is not unusual for code produced by Mac F2C to compile without any problems, but crash and burn when run. In fact, if you get to the ccommand() dialog (the one that asks you to set up standard input and standard output), but crash immediately there after, it is almost certain that you are blowing out the stack space. The problem and its solution are described below.

The problem arises because FORTRAN allocates memory for all its variables statically. The C code produced by Mac F2C by default uses automatic storage for most variables. On the Macintosh automatic variables get stored on the stack. When the FORTRAN code declares large arrays (quite common in FORTRAN code), these become large arrays on the stack in the C version. Because less than 24K is normally allocated as stack space by default on the Macintosh, your program blows out the stack.

The quick fix to try is to translate the FORTRAN code with the Make local variables automatic vice static option in the C Output options dialog unchecked. This will move all of the large arrays from the stack to the global data space. You should also increase the memory partition for your program (either in the Finder's Get Info dialog if it is a free-standing application, or in the appropriate THINK, Symantec, or CodeWarrior dialog if it is under development) to allow for the extra memory required by the large global arrays.

This "fix" will often solve the problem. However, because THINK still imposes a 32K limit on the amount of global data per file, you can still have problems with files that have too many large arrays in them. If you are lucky, you can divide the file somehow so that the arrays are split among several files and now don't violate the 32K/file limit. However, that might not be possible if, for example, all the arrays appear within a single subroutine.

In such a case, the only solution is to edit the C code so that it allocates the memory dynamically.

Let's work an example. Consider the following simple FORTRAN program:

```
program example

double precision d
dimension a(100,100), b(100,200,20)
dimension d(50,20,10)
```

```
     a(100,100) = 1.0
     b(100,200,20) = 1.0
     d(50,20,10) = 1.0d0

     stop
     end
```

Basically all it does is allocate three arrays. Note that the arrays are not that large by FORTRAN standards. Selecting the option that allocates local variables automatically, this program translates into:

```
/* JUNK.f -- translated by f2c (version 19941113).
   You must link the resulting object file with the libraries:
     -lf2c -lm   (in that order)
*/

#include "f2c.h"

/* Main program */ MAIN__(void)

     /* Builtin functions */
     /* Subroutine */ int s_stop(char *, ftnlen);

     /* Local variables */
     real a[10000]   /* was [100][100] */, b[400000] /* was [100][200][20]
          */;
     doublereal d[10000] /* was [50][20][10] */;

     a[9999] = 1.f;
     b[399999] = 1.f;
     d[9999] = 1.;
     s_stop("", 0L);
     return 0;
 /* MAIN__ */

/* Main program alias */ int example_ ()  MAIN__ (); return 0;
```

Note how in the C version 410,000 reals and 10,000 doublereals are allocated on the stack—that's over 1.5MB on the stack. Your program will crash the moment it enters the MAIN__() function. If you tried using static vice automatic variables you would have over 32K of global data in one file and it wouldn't build under THINK C.

Multi-dimensional arrays can be quite large even when their individual dimensions are small. It's easy to blow out the stack or the THINK C limit on global data per file with even one or two arrays of three or more dimensions. I've also seen programs get into problems because they have lots of small arrays.

The solution is to replace the array variables with pointers and allocate the memory for them using `malloc()` or any of it's cousins.   Because C treats pointers and arrays similarly, the rest of the code works as is.

Applying this technique to the example yields the following modified code:

```
/* JUNK.f -- translated by f2c (version 19941113).
   You must link the resulting object file with the libraries:
     -lf2c -lm    (in that order)
*/

#include "f2c.h"

#define NEW_CODE                /* <========   to make the changes clear */

#ifdef NEW_CODE
#include <stdlib.h>             /* Get prototypes for malloc(), etc */
#endif

/* Main program */ MAIN__(void)

     /* Builtin functions */
     /* Subroutine */ int s_stop(char *, ftnlen);

     /* Local variables */
#ifdef NEW_CODE
     real          *a, *b;          /* Pointers instead of arrays */
     doublereal   *d;
#else
     real a[10000]    /* was [100][100] */, b[400000] /* was [100][200][20]
          */;
     doublereal d[10000] /* was [50][20][10] */;
#endif


#ifdef NEW_CODE                                          /* allocate the memory */
     a = (real *) malloc( 10000*sizeof(real) );
     b = (real *) malloc( 400000*sizeof(real) );
     d = (doublereal *) malloc( 10000*sizeof(doublereal) );
     /* Remember to check a, b, and d for NULL (e.g., out of memory) */
#endif

     a[9999] = 1.f;
     b[399999] = 1.f;
     d[9999] = 1.;

#ifdef NEW_CODE
     free( a );           /* Don't forget to free the memory */
     free( b );
     free( d );
```

```
#endif

     s_stop("", 0L);
     return 0;
 /* MAIN__ */

/* Main program alias */ int example_ ()  MAIN__ (); return 0;
```

This version of the program will work fine so long as there is enough heap space to allocate the memory requested in the `malloc()` calls.   You can add more heap space by simply increasing the partition size of the program (either in the Finder's Get Info dialog if it is a free-standing application, or in the appropriate THINK, Symantec, or CodeWarrior dialog if the code is under development).

This modified version will also work if the variables a, b, and d had been declared static vice automatic.