The Implementation and Organisation of Work Arrays in Numerical Algorithms

Tim Hopkins and Les Hatton
Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, U.K.

April 28, 2004

Abstract

We have observed, in a wide variety of circumstances, the difficulties posed by the need for temporary work storage in numerical algorithms. Practical implementations in such libraries as ACM's Collected Algorithms (CALGO [1]) vary widely and frequently obscure the details of the algorithm itself. In this paper, we review typical practices and possible implementation solutions in various languages in an effort to guide the numerical software developer.
Keywords: Workspace, Fortran, C, C++, Ada

1 Review

Numerical algorithms are commonly written in some dialect of Fortran, Ada or one of the C-like languages, C, C++ or Java. The dialects include Fortran 66, 77, 90 and 95, Ada83 and 95, C90 and various flavours of C++ which has only recently become standardised. In each of these languages, handling work arrays is obviously a problem for implementors. We will not discuss Java further in this paper owing to the relative paucity of published numerical routines which have been implemented using it.

Recent work in the testing and updating of the Collected Algorithms of the ACM (CALGO now published in ACM Transactions on Mathematical Software) has highlighted a general lack of consistency in the way in which writers of scientific and numerical software implement work arrays. These arrays provide the temporary storage required by the algorithm for intermediate calculations and are 'unnecessary' as far as problem defining data is concerned.

When machines were far slower work arrays were allocated statically within the routines requiring them. The amount of storage was usually chosen to enable what, at the time, was considered an exceedingly large problem to be solved; unfortunately, huge increases in both processor power and memory have made many of these limits laughably small. With standard Fortran 66 this involved the use of explicit numerical constants to define the sizes of the arrays and upgrading these codes to operate on larger problems often involves changing numerical constants throughout the code as well as the obvious dimensioning information. Very rarely do these codes include checks to ensure that enough workspace has been allocated often causing dramatic failure when the available space is exceeded and array bound checking has been switched off.

Although static space allocation is still found in software submitted (but not published) in CALGO, this practice has been largely superseded by a number of other methods using common blocks and wrapper routines as detailed below. However, before giving details of these methods, it is worthwhile mentioning a couple of practices that were used and still appear from time to time in production code.

In an attempt to minimize the number of changes required to a Fortran package, work array storage of the correct size is provided in labelled common blocks in the main program but is declared to be of length 1 in the package code. Worryingly this requires both the user (and, presumably, the developer) to switch off all array bound checking within the package. Since

```
PROGRAM  MYNUM
COMMON /CWORK1/ WORK1(1000)
...
SUBROUTINE DOALG( X, N )
INTEGER N
REAL X(N)
COMMON /CWORK1/ WORK1(1000)
...
```

Figure 1: Use of COMMON for work arrays in Fortran 77

reading and writing data to addresses outside the requested array bounds is one of the most common run-time errors this practice certainly does not bode well for the quality of the software. It also does not adhere to any of the published Fortran standards.

Another method is to provide all the required work arrays of a particular type as a single array which is carved up internally and the slices passed to the requisite routines. This simplifies the interface to the user. The second common disreputable practice is to over simplify this interface by using a single, usually floating-point, array for both integer and real data. This requires the compiler to contravene the prescribed standards by abandoning the checking of actual and dummy argument types, another common programming error (see Hatton [5]) which may lead to unexpected results being generated. Unlike the illegal use of common blocks mentioned above, removing this error often requires completely redesigning and rewriting large sections of the software.

Given the obvious difficulties which the authors of numerical algorithms have experienced with Fortran, we will look at the ways in which a number of the more common scientific programming languages allow for the provision of workspace arrays and provide 'best buy' approaches for each language.

The essence of the work array is that it is certainly intended to be re-usable and it generally has a more limited life-time than other kinds of data structures. In an ideal world, we would like to be able to spirit work arrays into existence only when needed and for them to disappear immediately when they are no longer required, preferably automatically so we don't have to bother with the house-keeping. We call this kind of storage dynamic in contrast to static storage which is allocated at the beginning of execution and persists through the entire lifetime of the running program. Automatic deallocation of storage is also known as garbage collection.

Work Arrays in Fortran

Until Fortran 90, there was no dynamic storage in Fortran. The programmer had to create all the workspace that a program would ever require at program initiation and pass it around the program to the places where it was needed. This can be done essentially in two ways in Fortran 66 and Fortran 77, via COMMON and via argument lists. In terms of simplicity of interface, the COMMON method is preferred as in Figure 1.

In the case of the COMMON method, work arrays do not appear in any argument lists thus keeping the interfaces as clean as possible. The drawback is both that the workspace is global and therefore open to corruption from other parts of the program and, that it is static and allocated for the life-time of the program.

Changes to the size of work arrays defined via COMMON statements necessitate changes to be made in each routine that includes the COMMON block. Additionally the actual length of the arrays may also be used within the body of the subprograms, for example, when ascertaining whether enough workspace has been reserved for a particular problem. If the same COMMON blocks are used throughout the software these changes may be minimized by the use of named constants as illustrated in Figure 2.

```
PROGRAM  MYNUM
INTEGER WKLEN
PARAMETER (WKLEN=1000)
COMMON /CWORK1/ WORK1(WKLEN)
...
SUBROUTINE DOALG(X,N)
INTEGER WKLEN
PARAMETER (WKLEN=1000)
COMMON /CWORK1/ WORK1(WKLEN)
...
IF (N.GT.WKLEN) THEN
  CALL ERROR(...)
  ...
ENDIF
...
```

Figure 2: Use of COMMON for work arrays in Fortran 77 using named constants

```
PROGRAM  MYNUM
INTEGER NAMX, MMAX
PARAMETER (NMAX=100, MMAX=200)
COMMON /CWORK1/ SOLN(NMAX), SQARRAY(NMAX+MMAX, MMAX+1)
```

Figure 3: Use of COMMON for work arrays in Fortran 77 using components

Further improvements to the readability of the code may be made by explicitly naming the various parts of the workspace required and defining the component lengths in terms of the maximum values of problem dependent parameters, see Figure 3 for an example.

To minimize both the effort required and the chances of making errors when changing either the sizes or the make up of work arrays, the same names should be used in each occurrence of the COMMON block.

Many pre-Fortran 90/95 compilers allow the use of include files as an extension to the published standards. This appears, at first sight, to be an excellent solution to the problem of changing the size and/or the constitution of work arrays since only one copy needs to be edited and the changes automatically occur throughout the code on recompilation. However, there are serious portability issues to consider. First, the include statement was not part of the ANSI standard [2], and thus differs from system to system, for example, the keyword INCLUDE may start in either column 1 or column 7. Second, the file names will be operating system dependent. While the INCLUDE statement is now part of the Fortran 90/95 standard the second problem above still applies and the use of modules is a far cleaner solution.

An early attempt to implement a portable mechanism for the provision of work arrays using Fortran 66 was made by Fox et al. [3]. Their approach was to build a small library of, almost standard, Fortran 66 routines which used an area of labelled common as a stack. Although the library interface was quite simple its use required wrapper routines to ensure readability. Figure 4 shows an example of the use of the routines to allocate an integer and real workspace array.

The use of equivalence statements allows the user to acquire workspace of the correct type from a double precision array. To adhere strictly to the Fortran 66 and 77 standards the library routines should be edited and recompiled if the amount of workspace required was to exceed 500 double precision storage units. The recommended work-around given in the paper is to declare the array to be of the required length in the main program, turn off array bound checking and hope that the compiler does not provide enough information for the linker to check the consistency of labelled common block lengths!

```
      SUBROUTINE DOALG (X, N)

      DOUBLE PRECISION DSTAK(500)
      COMMON /CSTAK/ DSTAK

      INTEGER ISTAK(1000)
      REAL RSTAK(1000)
      REAL X(N)

      EQUIVALENCE(DSTAK(1), RSTAK(1))
      EQUIVALENCE(DSTAK(1), ISTAK(1))

C.. On return elements ISTAK(II)..ISTAK(II+2*N-1) are
C.. reserved for integer data, second argument = 2.
      II = ISTKGT(2*N,2)

C.. On return elements RSTAK(IR)..RSTAK(IR+N-1) are
C.. reserved for real data, second argument = 3.
      IR = ISTKGT(N,3)

      ...

C.. Release the space obtained by the last 2 calls
C.. to the routine ISTKGT
      CALL ISTKRL(2)
```

Figure 4: Example use of the Fox et al. dynamic storage library routine

```
PROGRAM  MYNUM
INTEGER WORK1(1000)
...
SUBROUTINE DOALG( X, WORK1 )
...
```

Figure 5: Passing work arrays through argument lists in Fortran

Finally we note that, unlike the routines described in the same paper to set machine dependent constants, the dynamic storage routines were almost completely ignored by authors of later CALGO algorithms.

The alternative to the use of COMMON is to place the work arrays in the top level of the program and pass them down through the interfaces as illustrated in Figure 5.

This is probably the most frequently occurring form used in the ACM CALGO library but it can lead to very obfuscated interfaces indeed, particularly if multidimensional arrays are used, if the programmer attempts to pass down dimension sizes for checking, or there are numerous layers in the calling tree. In addition the temptation for programmers to embed different numerical types within the same work array (mixing integers and floating point numbers is particularly popular and explicitly forbidden by the Fortran 77 standard) is often overwhelming with resoundingly negative effects on the programs potential reliability and portability.

In order to preserve a sensible name space when developing the software, this method often leads to wraparound routines (see Figure 6 for an example). These protect the user from having to provide detailed dimensioning information and simplify the interface to the provision of a single workspace array for each numerical type while at the same time allowing the programmer freedom to work with sensibly named component arrays.

This mechanism is frequently used both in the NAG Fortran Library [7] and in CALGO software. The ability to lie legally about the dimensionality of arrays passed as actual arguments means that the user is freed from the need to provide the leading dimension of multidimensional arrays.

In a similar way to that used for COMMON blocks, expressions involving problem dependent constants may be used to reserve just the amount of array space required to solve a particular problem as illustrated in Figure 7, although it is often the case that Fortran 77 does not allow the requisite expression within a PARAMETER statement when intrinsic functions (for example, MAX) are required.

A third possibility is to hard-wire work arrays to be of fixed size within the subroutines themselves but this is particularly inelegant and fraught with danger if the boundaries are accidentally exceeded or the sizes need to be changed to accommodate larger problems. The COMMON method is probably the best alternative given what is known about how Fortran programs fail.

Fortran 90 and 95 provide much improved facilities for generating workspace when and where it is required. This is an advantage in that programmers are less tempted to mix types and the onus is no longer on the user to ensure that correct amounts of workspace are available.

The simplest method is to use *automatic arrays* which exist only during the execution of the subprogram in which they are declared. The size of these arrays is define using a *specification expression*; this is a non-constant integer expression that may depend on any data that is defined on entry to the subprogram. This includes the use of intrinsic functions, like *size*, that can determine the actual lengths of dummy array arguments. With Fortran 95 a specification expression can use the full range of intrinsic functions along with a limited form of user defined functions. An example is shown in Figure 8.

Fortran 90 also introduced dynamic storage allocation for the first time. This allows for both the generation of workspace arrays and the construction of data structures like linked lists. Fortran 95 can automatically release this storage on exit from the program unit that creates it; a vast improvement over the bizarre undefined allocation status possible in Fortran 90. Even so it

```
PROGRAM  MYNUM
INTEGER RWLEN, IWLEN
PARAMETER (RWLEN=1000, IWLEN=500)
REAL RWORK(RWLEN)
INTEGER IWORK(IWLEN)
INTEGER N
...
CALL DOALG(X,N,RWORK,RWLEN,IWORK,IWLEN)
...

SUBROUTINE DOALG( X, N, RWORK, RWLEN, IWORK, IWLEN)
REAL RWORK(RWLEN)
INTEGER IWORK(IWLEN)
INTEGER RST1, RST2, RST3, IST1, IST2, IST3
RST1 = 1
RST2 = RST1+N
RST3 = RST2 + N*N
IST1 = 1
IST2 = IST1 + 2*N
IST3 = IST2 + N
CALL MYALG(X, N, RWORK(RST1), RWORK(RST2), RWORK(RST3)
+ IWORK(IST1), IWORK(IST2), IWORK(IST3) ...)
...
END

SUBROUTINE MYALG(X, N, RHS, COEF, RESID, IWGHT, IPVT,
JVALS)
REAL RHS(N), COEF(N,N), RESID(N)
INTEGER IWGHT(2*N), IPVT(N), JVALS(N,N)
...
```

Figure 6: Use of wrap around routine for work arrays in Fortran 77

```
PROGRAM MYNUM
INTEGER N, M, RLEN, ILEN
PARAMETER (N=12, M=15)
PARAMETER (RLEN=N+(N+1)*(N+1)+M, ILEN=4*N+5)
REAL RWORK(ILEN)
INTEGER IWORK(ILEN)
...
```

Figure 7: Illustration of calculation of user provided workspace arrays

```
      SUBROUTINE SUB(A, B, X)
      REAL, DIMENSION (:,:) :: A
      REAL, DIMENSION (:) :: B, X
      INTEGER, DIMENSION(SIZE(B)) :: IPVT
      REAL, DIMENSION(INT(LOG(REAL(SIZE(B)))/LOG(2.0)) &
&                     +3*SIZE(B)+MAX(38,SIZE(B))) :: WKSPCE
      ...
```

Figure 8: Using automatic arrays to create workspace in Fortran 95

```
      INTEGER :: IERR
      REAL, ALLOCATABLE :: SOL(:,:)
      INTEGER, ALLOCATABLE :: IPVT(:)
      ...
      READ(...)N, M
      ALLOCATE(SOL(N,M), IPVT(N-1), STAT=IERR)
      IF (IERR/=0)THEN
        WRITE(...)'Failed to allocate workspace'
        STOP
      ENDIF

      ...

      DEALLOCATE(SOL, IPVT, STAT=IERR)
      IF (IERR/=0)THEN
        WRITE(...)'Failed to deallocate workspace'
        STOP
      ENDIF

      ...
```

Figure 9: Creating dynamic space in Fortran 90/95

is preferable that the user deallocate all allocated storage explicitly so that it is clear when such storage is no longer required; an example is given in Figure 9.

New Fortran numerical software should be developed using Fortran 90 to enable the use of these and many of the other useful facilities that have appeared with the new standard.

Work Arrays in C

Workspace in C can be handled in very similar ways to Fortran 66/77 but overwhelmingly the most popular method is by manipulation of dynamically created work arrays in a storage area known as the heap through the standard access functions malloc(), calloc(), realloc() and free().

These functions can, with a very sweet trick long known in the C community, be used in a highly portable way to manipulate multi-dimensional arrays as shown in Figure 10.

The underlying malloc family of functions perform exactly the same role as the dynamic allocation functions of Fortran 90/95 (indeed dynamic allocation in Fortran 90 and 95 was inspired by the fact that you could already do it in early dialects of C). This is somewhat unfortunate as it was also known that because the programmer bore the entire responsibility for consistent creation and freeing of the heap, this form of dynamic allocation suffered from the following failure modes

1. Multiple frees on the same pointer

```
/* Create a 2-d array using malloc */
void ** alloc2( size_t mrows, size_t ncols, size_t size )
{
 void **p;
 size_t irow;
 /* Create pointer for each row */
 if ((p = malloc(mrows*size)) == NULL )
 return NULL;
 /* Create the 2-d matrix and attach to p[0] */
 if ( (p[0] = malloc(ncols*mrows*size)) == NULL)
 return NULL;
 /* Initialise the row pointers  */
 for( irow = 0; irow < mrows; ++irow )
 p[irow] = (void *) (p[0] + ncols*irow*size);
 return p;
}

/* Function to cast it to the right type */
float ** alloc2float( size_t mrows, size_t ncols )
{
 return(float **) alloc2( mrows, ncols,
sizeof(float));
}

/* And in use ... */
int main(void)
{
 float **q;
 q = alloc2float( mrows, ncols );
 q[0][0] = 0.
 q[0][1] = 1.
 ...
```

Figure 10: Adroit use of malloc to create multi-dimensional arrays. A corresponding free function can be constructed relatively simply

```
p = malloc(...);
free(p);
free(p); /* undefined behaviour */
```

2. Memory leaks

```
p = malloc(..);
p = malloc(..); /* All reference to first malloc'd memory lost */
```

3. Memory bloat

```
p = malloc(..); /* Never bothering to free even when you can */
```

4. Messing up reallocation

```
realloc(p, 500);
```

   instead of:-

```
p = realloc(p,500); /* realloc is entitled to move p if it wants */
```

5. Not malloc'ing enough

```
p = malloc(12);
strcpy(p,"Edinburgh Academicals");
```

This latter overwrites the 12 byte space created happily blundering on into whatever happens to be next. If the programmer is lucky this will fall over straight away. If not, the programmer usually obtains a rather unhelpful error message from somewhere else claiming that the heap has been corrupted. Such errors can be exceedingly difficult to find and correspond to the difficult category of diagnostic problems described by Hatton [6].

So, dynamic heap storage solves one problem in providing the work array where the programmer needs it without the legacy of carrying it around in global data structures or interfaces. Unfortunately, it introduces a new class of failure which has proven to be particularly insidious.

There is another way of doing it in C. The downside is that it lies outside the remit of the C standard. The upside is that it works and is very commonly used by system functions. It relies on the existence of a function called alloca(). Unlike the standard functions described above, alloca() uses the system stack not the heap. The stack is not normally addressable by the user, it is used by the compiler itself to store argument information, local variable allocation and other housekeeping. The good news is that the compiler itself is responsible for allocation and deallocation so that anything on the stack (including any space allocated by alloca()) will be released automatically when a function ends.

In other words, use of alloca() provides us with the ideal of work space which can be created when needed and which disappears automatically when the function in which it is created exits. The programmer is freed from the responsibility of releasing the workspace safe in the knowledge that it will not accumulate. Of course, the programmer is still responsible for not overwriting array bounds, but that is a continuing obligation with most programming languages.

At the time of writing, the original ISO C standard, 9899:1990 has just been replaced by a new C standard, 9899:1999. The updated language is a superset of the old standard so all of the above comments apply. As no numerical software has yet been encountered in this new dialect, it will not be considered further at this stage.

```
void func( int length )
{
double *work = new double[length]
//
// Do things with work, and then delete it.
//
delete[] work;
}
```

Figure 11: An example of the use of the new / delete construction of C++ and the array deletion syntax

Work Arrays in C++

C++ has considerably superior mechanisms for handling work arrays than either C or Fortran. Unfortunately, this benefit is bought at the cost of complexity. C++ is probably the most complex language ever standardised and it is unusually difficult to use reliably with the corrective cost of failure thought to be significantly higher according to some reported data (see Hatton [4]). With this caveat in mind, C++ provides a superior mechanism to the malloc() / free() concept of C known as new / delete. Although allocation is still made from the heap with all of the problems mentioned earlier, the new / delete mechanisms automatically call constructors and destructors for the allocated objects (which have to be provided). In addition, C++ has a standard syntax for deallocating arrays.

An example of the use of new and delete is given in Figure 11. In fact, C++ can go beyond this. For example, it is possible to overload the [] subscript operation (overloading means redefining the functionality of a particular operator when used with certain objects) to avoid array bound violations by defining run-time checking every time the subscript operation is invoked. The [] operator is one of the majority of C++ operators which can be overloaded in this way although there are some additional restrictions on a few operators including this one, for example any function overloading [] must be a non-static member of a class object. This requires careful handling in the language and is not for the faint-hearted or casual C++ programmer. Finally, it incurs a significant run-time penalty. If it is well designed however, it can provide an elegant way of handling work arrays alongside the new / delete mechanism with the repeated caveat that the programmer is responsible for managing allocation and de-allocation.

Work Arrays in ADA

We will only discuss Ada83 here as we have encountered too few numerical software packages written using Ada95 to offer any advice.

Ada83 uses a static model rather like Fortran but new objects may be created dynamically using the NEW construct. NEW is very like malloc() in C, although it is a little more general and contains a form which can also initialise the allocated storage (rather like calloc() in C but with the ability to initialise to other than zero). However, you cannot delete such objects easily in Ada83, the idea here being to make it harder to inject any of the heap abuse defects described earlier in conjunction with C. The compiler is then free to do any kind of automatic reclamation of heap (garbage collection) that it can. Unfortunately, some Ada implementations do hideously slow garbage collection and there is no requirement to do this at all.

There is also a pragma instruction which will hint to the compiler that NEW objects pointed at by a local type T can be freed when the local type goes out of scope. This must appear immediately after the declaration of T as follows

```
pragma Controlled (T);
```

Finally, as a last resort, Unchecked_deallocation may be used. This is just like the free() in C, and is a licence to risk any of the heap abuse problems described earlier. If you don't do this and the heap eventually runs out of allocatable space, the implementation is required to issue a Storage_error exception, so at least the user is informed what went wrong. (In C, if the heap runs out of space, the programmer will not know unless they test the returned value of malloc() or whatever. It is surprisingly common for programmers not to do this simple but fundamental housekeeping.) As can be seen, the mechanism in Ada83 is very similar to the other languages discussed with just a few attempts to make it somewhat more robust. However, there is still a strong onus on the programmer to maintain the allocated storage appropriately.

Conclusions

We have looked in detail at the ways in which work arrays are implemented in numerical software using a number of common implementation languages and it is clear that programmers of numerical software have significant difficulties with this. Practices vary considerably and many methods, especially when using the older dialects of Fortran, can lead to code that is difficult and error prone to change in order to solve problems that are larger than the original author contemplated.

We have provided a detailed description of the ways in which we believe work space may be more successfully implemented than it has in the past and we sincerely hope that we will see a far more consistent use of work arrays in future numerical software. It is interesting and somewhat ironic to note that in spite of our efforts to provide such facilities in programming languages, probably the easiest and most transparent and portable method to use (alloca() in C) is an admitted hack. In spite of this, arguably one of the most reliable and certainly most portable compiler in the world, the GNU C compiler, makes extensive use of this feature.

References

[1] ACM. *Collected Algorithms from ACM*, volume I – IV. Association for Computing Machinery, Inc., New York, 1980 – 1996.

[2] ANSI. *Programming Language Fortran X3.9-1978*. American National Standards Institute, New York, 1979.

[3] Fox PA, Hall AD, Schryer NL. Framework for a portable library. *ACM Transactions on Mathematical Software*, 4(2):177–188, June 1978.

[4] Hatton L. Does OO sync with the way we think? *IEEE Software*, 15(3):46–54, November 1997.

[5] Hatton L. The T experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, 4(2):27–38, January 1997.

[6] Hatton L. Repetitive failure, feedback and the lost art of diagnosis. *Journal of Systems and Software*, 47(2–3):183–188, July 1999.

[7] Numerical Algorithms Group, Ltd, Oxford. *NAG Fortran Library Manual, Mark 19*, 1999. ISBN 1-85206-169-3.