Improving RELAP5-3D Data Management

George Mesina

December, 2014

Data handling improvements increase RELAP5-3D reliability and prevent future errors.

Introduction

Many errors are created by insufficient handling of data between the point where it is created and its first usage. Examples include memory leaks, hanging of a computer during parallel operations, and destruction of data. When data is first allocated, it should be initialized. A feature of FORTRAN 2003 allows this through a keyword in allocate statements. Pointers should be nullified upon creation. A systematic approach to handling data better from the outset is underway for RELAP5-3D.

Background

The abilities to allocate memory, create pointers, and create derived types were added to FORTRAN in the 1990 ANSI standard. Enthusiastic programmers incorporated these constructs into legacy codes, such as RELAP5-3D. But without standards to guide their usage, many errors resulted. Initial use of allocation and deallocation, particularly where complex derived types were involved, resulted in indexing errors, calculations that randomly change from one run to the next, memory leaks (loss of access to memory during a run due to incorrect memory allocation), and core dumps. Equally deleterious effects have resulted from the use of pointers.

Many error reports have been tracked back to the failure to properly allocate, initialize, and deallocate arrays and pointers and/or to nullify pointers. Periodically, new errors of this sort are reported. These bugs are often difficult and time-consuming to track down.

Examples

Consider User Problem 14023. A HTTF problem failed immediately upon restart from steady state to transient. The linear equation solver correctly reported a singular matrix that included a zero pivot row, zero pivot column, and NaNs (Not a Number). This was traced to the exercise of an allowable restart option to extend the table of an insulator (material table 9). When the table was not extended or when certain values were used, the error did not occur. The error was eventually tracked to subroutine QFMOVE which allocated numerous arrays without initializing them. This caused the restart to produce various NaNs. The problem was resolved by initializing the arrays when they were first allocated.

Another example is User Problem 14018. An IRUG member reported that a multi-case flexible wall problem core dumped on the third case, but when the multiple-cases were run separately, all 8 cases ran correctly. At INL, this problem ran to completion with the “Windows Release” version but failed for the Linux and “Windows Debug” versions. The error traced to subroutine PRESEJ where the subscript of array COEFM, namely COEFM(JCT(4)%OFFDS(2)), had random values like -5360683942457357825, which all appeared to be real number bit-patterns contained in the integer index JCT(4)%OFFDS(2). Initializing array element overcame the issue of bad values in that location, but did not solve the problem.

In fact, it was insufficient to initialize the entire array JCT(:)%OFFDS(2) because the next case failed for a different indexing error. The solution was to zero out the entire element of the JCT-array before allowing subroutine RSNGFW to assign values to various components of the JCT element. This was accomplished by writing an initialization routine in JUNMOD that initializes any set of JCT-like array elements from a start index to an end index. It is called from RSNGFW immediately after CMP(nc)%JUNP(1) is allocated.

In User Problem 14001, an INL user discovered memory leaks when running RELAP5-3D in conjunction with PHISICS. These occurred in subroutines RRKINO and RRADHT. For a "large" problem they amount to 1MB to 6MB lost during the run.

Two instances of memory leaks were traced to allocation of the components of derived type arrays. There was a proper test of whether or not the entire array was allocated, so it was only allocated if it did not already exist. However, statements that allocated some of its derived type components lay outside the protection of that if-test and executed unconditionally. Therefore, when any of these derived type components existed, an error occurred because reallocating something that already exists (without first deallocating it) is not allowed. This can happen for instance, on the second case of a multi-case input deck. It also occurs and is exacerbated in parallel when more than one process attempts to allocate the derived type components separately as is done when running with PHISICS.

These runtime failures and memory leaks were solved by including the allocation of the components within the scope of protection provided by the derived type allocation test. It was also necessary to test the existence of each component before allocating it.

Systematic Improvement

These examples illustrate that debugging problems caused by allocation, initialization, nullification and deallocation errors can be time consuming and difficult to find and fix. Rather than wait for new error reports to surface, and repeat the usual resolution process each time, proactive measures are being undertaken. Application of the lessons learned from these and other user problem resolutions will save debugging time and increase code robustness and reliability.

From these debugging efforts, certain rules have been developed for using allocation, deallocation, initialization, nullification, and use of pointers. The rules are as follows:

1. Array and pointer allocation must be tested before it is exercised.
    a. The size must make sense (positive and not overlarge).

   b. The array or pointer must not already be allocated.
   c. For derived types, check the allocation status before proceeding to allocate components.
2. Memory must be initialized whenever it is allocated one of two ways:
   a. With a loop, or
   b. With the FORTRAN 2003 "source" keyword on the allocate statement.
3. Pointers must be nullified or assigned immediately upon creation.
4. Arrays and pointers must be deallocated before the run terminates.
   a. The allocation status of arrays must be tested before deallocating.
   b. The association status of pointers must be tested before deallocating.
   c. Derived types must be deallocated from bottom up to prevent memory leaks.

Now that the rules have been established, a systematic effort is being undertaken to apply them. Two separate modules of allocation and deallocation routines for arrays have been written. In ALLOMOD, the call sequences are simple, while those in ALLOCMOD add call parameters that pinpoint the location of the call and provide an error message in case of failure.

The starting point is the other RELAP5-3D FORTRAN 90+ modules, these are being visited in alphabetical order. Many modules have constructor and destructor subroutines for derived types and pointers; however, these need improvement according to the rules presented above. For simple arrays and those derived-type arrays with fixed substructure, the source keyword is introduced on the allocate statement. For the latter, a "derived-type zero" data element, comprised primarily of floating point and integer zeroes in the appropriate positions, is created immediately. This derived-type zero is placed immediately after the declaration of the derived type and is used on the allocate statement.

For large and complex derived types of fixed structure, such as in JUNMOD and VOLMOD, the interference of pre-compiler directives and alternation of real and integer statements (does so that related data is grouped together) makes the creation of a derived-type zero difficult and time consuming to construct. Yet due to proximity with the declaration of the derived type, once complete, they are easy to update when new data is created.

Besides JUNMOD and VOLMOD, modules beginning with the letters A-J have been upgraded to conform to the rules. After the modules are completed, subroutines will be examined in like manner.

This project works on individual modules and subroutines (generally, there are a few exceptions of linked modules and subroutines), and will be an ongoing effort as time allows.