Programming Tips for Plugins

Chad Neufeld

Centre for Computational Geostatistics
Department of Civil & Environmental Engineering
University of Alberta

Working in a university based research environment is dynamic and fast-paced. The driving force is pressure to devise new algorithms and techniques; the driving force in most industry applications is different. Development of new ideas and methods takes place with any available tool. In particular, software is never limited to one particular platform or programming style. This presents an issue for the companies that sponsor our research and wish to implement it.

Although a company may want to try some of our new tools, they are often limited to implementing the programs in a command line based setting. This is less than ideal for most users who have become comfortable with a particular piece of commercial software. Software vendors are making it easier to integrate additional functionality into their software. The Petrel Ocean API is one example.

This paper outlines some program tips for writing tools that will be included into a larger software package. An example is included showing how Fortran can be called as a subroutine or function from C#. In most cases, only small programming changes are required so that the code can be incorporated into other software.

Introduction

Software plays an interesting role in geostatistics. Some see it as an unfortunate necessity of what we do, while others build their careers and companies on it. For example, universities and research groups have a very different perspective, and need, of what software should do compared to a practitioner. In a university setting, software is used to test new ideas, prototype algorithms or solve a very specific problem. Contrast this to a practitioner that requires a stable and flexible tool that can be used to solve multiple problems.

At this point it would be good to clarify a few points. At the university, we divide software into 2 separate categories: (1) tools for performing a specific task and (2) software that can perform multiple tasks. An example of a tool is SGSIM. It has a very specific function and cannot do anything else. An example of software is a commercial software package: Petrel, Vulcan, GOCAD, Gemcom, etc. A tool, such as SGSIM, is bundled as part of the software package and it is used when needed from within the package. From here on in, large software packages with be referred to as "software" or "packages" and tools will be referred to as "tools" or "programs."

Our vision at the university *is to be the unquestioned leader in the education of geostatistical practitioners and the development of geostatistical tools*[1]. However, developing the most advanced tools is no good to our sponsor companies unless they can implement the tools themselves. And, most research ideas are never included into a commercial software package or may take years to become common enough that they will be included; Figure 1 shows an estimate of how many research ideas actually make it into main stream use. Some commercial software packages have just released conditional simulation modules. Simulation has been around for over 20 years or more.

[1] Introduction to the Sixth Annual Meeting of the Centre for Computational Geostatistics. September 2004.

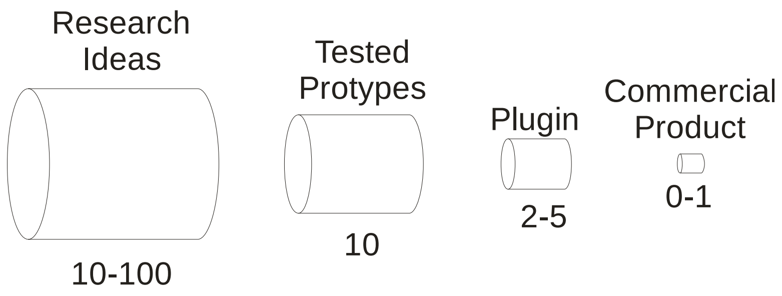


Figure 1: Estimate of the number of research based tools that will be included in a commercial software.

Almost all of our tools are based on GSLIB. This means that companies are limited to implementing new tools and programs in a command line based setting. This is less than ideal for most users who have become comfortable with a particular piece of commercial software. In addition, the exporting and re-importing of data and results is error prone and time consuming.

Software vendors are making it easier and easier to integrate new tools right into their commercial packages. The new Ocean API is Petrel's answer to incorporating additional code right into Petrel. This paper outlines some programming tips for tools that will be included into a larger software package. An example is included showing how Fortran can be called as a subroutine or function from C#. In most cases, only small programming changes are required so that the code can be incorporated into other software.

A New Coding Style

The existing coding style has been based on single stand-alone programs. All programs can be broken down into 3 steps: (1) reading the input parameters and data, (2) performing the numerical calculations and (3) writing the output. This is perfectly acceptable for a one-use executable, but it is less than ideal when the numerical portion will be reused in other code. It would be much better to adopt a slightly different coding style when writing the progam.

The proposed coding style is based around DLLs (dynamic-link libraries). A core algorithm is coded and compiled to a DLL. A program, be it stand-alone executable or a software package, assembles the data and program parameters, calls the DLL to do the numerical calculations, and receives the output from the DLL. The new approach requires 5 steps: (1) assembling the parameters and data, (2) calling the DLL and passing it the information from step 1, (3) the DLL does the numerical calculations, (4) the DLL returns the output, and (5) the calling program saves the output. Figure 2 shows a flowchart with the old and new coding styles.

There are several advantages and disadvantages to adopting the new coding style. The biggest advantage is the ability to use a central piece of compiled code. A simple stand alone executable can be written using the DLL or a commercial package can use the exact same DLL. This means that both processes are using the exact same algorithm. There is no worry about introducing bugs or making a mistake by copying and pasting source code from one file to another. In the event that a bug is found, it is a lot easier to fix and update a single DLL compared to updating multiple programs that use the same code. The disadvantages are a more complicated coding style and that it is harder to distribute the DLL compared to a simple executable. For a stand-alone program based on a DLL a minimum of 3 files are required: (1) the executable that reads the input and writes the output, (2) the DLL itself and (3) a run-time library for the DLL (compiler specific). Other files may be needed.

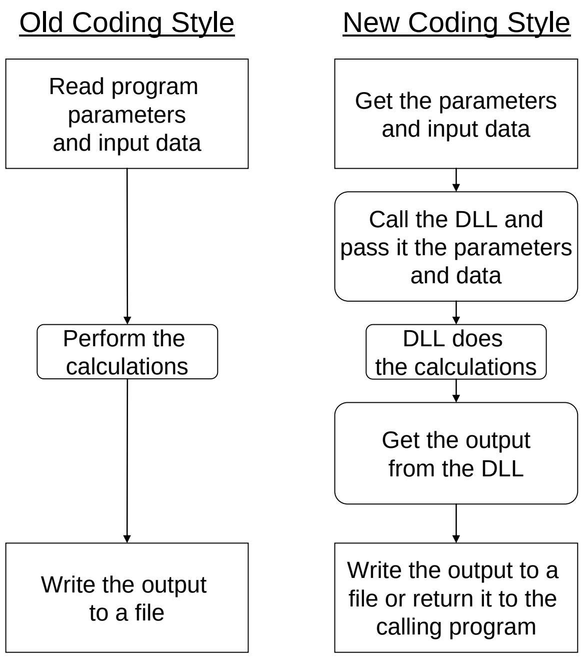

Old Coding Style
New Coding Style
Read program parameters and input data
Perform the calculations
Write the output to a file
Get the parameters and input data
Call the DLL and pass it the parameters and data
DLL does the calculations
Get the output from the DLL
Write the output to a file or return it to the calling program

Figure 2: Comparison of the workflow between the old and new coding styles.

Mixed Language Programming

Moving to a modular type coding standard introduces some interesting complications. Consider using an algorithm written in Fortran compiled in a DLL. It is easily called from another Fortran executable. However, calling the same DLL from a C program is not as simple. An interface is required telling the C program what the Fortran DLL is expecting for input and what the DLL will return as output. With the proper setup, a Fortran subroutine contained in a DLL can be called just like a built in C function.

There are 2 main differences between Fortran and C: (1) the way variables are passed when functions or subroutines are called and (2) the order that arrays are stored in the computers memory. When calling subroutines or functions in Fortran, the variables are passed by reference. In contrast, when calling functions in C, variables are passed by value unless it is specifically specified to pass them by reference. This creates a problem when calling a Fortran DLL from C. This will be discussed in more detail below. The second difference is how arrays are stored in memory. Fortran stores arrays in row-order while C stores arrays in column-order. This will be discussed in more detail later.

These are by no means the only issues that you may encounter with mixed language programming. Visual Basic, C++, and C# are just some of the programming languages that people have used for building applications that call Fortran DLLs. Each language will have its own issues and quirks. The remainder of this paper focuses on calling Fortran DLLs from C#. C# is a fairly new programming language developed as part of the Microsoft .NET framework. New GUI applications, including the Petrel API, are being written with C#.

Building a Fortran DLL

A DLL does not contain a program. It cannot be run on its own. It must be called from a separate program. The calling program passes the DLL parameters and data to perform the numerical calculations. In other words, a DLL is a collection of numerical algorithms, there can be more than one algorithm in one DLL, that can be called by any other program.

Fortran DLLs typically contain subroutines that are exposed outside of the DLL. The DLL can also contain other subroutines and functions that are not exposed in the DLL. These private functions can only be used by other functions within the DLL. Consider the following subroutine in its own file; `testDLL.f90`. It is a simple subroutine that calculates the sine of an input number and returns it in a separate output number.

```
subroutine sineT(num, sineNum)
  real(8), intent(IN)  :: num
  real(8), intent(OUT) :: sineNum
  sineNum=sin(num)
end subroutine
```

The file testDLL.f90 can be compiled to a DLL by including a switch at compile time or by creating a special project within the compiler's development environment. For the Intel compiler it can be done by using the following:

```
ifort /dll testFLL.f90
```

or by creating a DLL project in Microsoft Visual Studio.

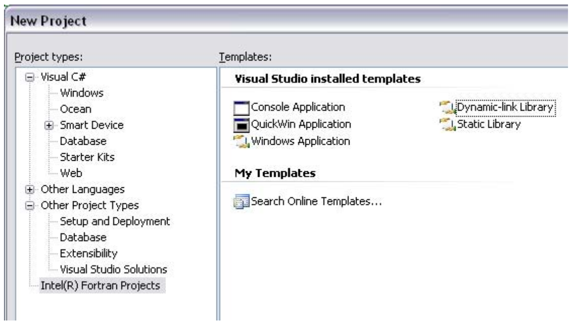
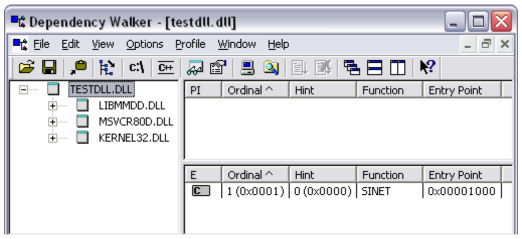


Figure 3: Creating a DLL project in Visual Studio.

There is a problem though. Nothing inside the DLL can be used by an external program. An error would be returned if a program were to call the DLL. A line needs to be added to the subroutine to export the subroutine and make it visible to a calling program. This is done with a compiler directive.

```
subroutine sineT(num, sineNum)
  !DEC$ ATTRIBUTES DLLEXPORT::sineT
  real(8), intent(IN)  :: num
  real(8), intent(OUT) :: sineNum
  sineNum=sin(num)
end subroutine
```

After compiling the DLL, the subroutine can be called from other programs. It would be beneficial to introduce a very useful tool right now. The tool is called Dependency Walker [1]. It is a free utility for scanning windows modules to determine exported functions and any dependencies that the module relies on. It is essential for diagnosing errors related to exported function names in DLLs. The output of dependency walker for `testDLL.dll` is shown in Figure 4. It shows the exported sine function and gives its name as SINET. Note that the case of the function name is not important for Fortran. It is important for C based applications. It also lists three required files for the DLL: `libmmdd.dll`, `msvcr80d.dll` and `kernel32.dll`. Two of these files are part of windows. The third file, `libmmdd.dll`, is from Intel and needs to be present for the DLL to run when called.



TESTDLL.DLL
LIBMMDD.DLL
MSVCR80D.DLL
KERNEL32.DLL

PI	Ordinal ^	Hint	Function	Entry Point

E	Ordinal ^	Hint	Function	Entry Point
C	1 (0x0001)	0 (0x0000)	SINET	0x00001000

Figure 4: Dependency walker output for testDLL.dll.

The previous example shows a very simple subroutine coded into a DLL. DLLs can be as simple or as complex as needed. The following code shows a subroutine that accepts a 3-dimensional array as input, allocates an additional array, does some calculations, and then deallocates the array and returns the result. This has been tested up to approximately 30 million cells in total.

```
subroutine arrtest3d(nx,ny,nz,arr,ierror)
    !DEC$ ATTRIBUTES DLLEXPORT::arrtest3d
    integer, intent(IN)    :: nx,ny,nz
    real(8), intent(INOUT) :: arr(nx,ny,nz)
    integer, intent(OUT)   :: ierror

    real(8), allocatable :: arr2(:,:,:)
    integer :: test
    allocate(arr2(nx,ny,nz),stat=test)
    arr2=arr*2
    where(arr2 > 1000) arr=arr2
    deallocate(arr2,stat=test)
    ierror=0
end subroutine
```

Recall that the functions and subroutines inside a DLL must be exported before they can be used. The way that the functions are exported must be specified in the DLL. There are several different conventions available for exporting DLLs. As you probably expected, FORTRAN and C based language have different default conventions. However, any convention can be used as long as the DLL and the calling program are aware what that convention was. The conventions define:

1. if the arguments are passed by value or reference,
2. how the function names are formatted, and
3. if a variable number of arguments is allowed.

The convention used depends mostly on user preference, or on how the DLL will be used. The default DLL convention is fine when Fortran will be used to call the DLL the majority of the time. This makes it easy to use the DLL from Fortran, but harder from other languages. If a different language, such as C# or VB, will be used to call the DLL, it may be beneficial to use a different convention. This makes it easier to use the DLL from something like C#, but not from Fortran. The conventions are: default, C, and STDCALL. Table 1 summarizes the differences between the DLL conventions.

Table 1: DLL convention summary

	Default	**C**	**STDCALL**
Arguments passed by value	No	Yes	Yes
Variable number of arguments	Yes	Yes	No
Case of external name	Upper	Lower	Lower
Leading underscore added to name	No	Yes	Yes
Number of arguments added to name	No	No	Yes

The default convention is used when function or subroutine is exported using the following command:

```
!DEC$ ATTRIBUTES DLLEXPORT::sineT
```

To use the C or STDCALL conventions, the following needs to be used:

```
!DEC$ ATTRIBUTES DLLEXPORT, C::sineT
```

or

```
!DEC$ ATTRIBUTES DLLEXPORT, STDCALL::sineT
```

Fortran and C pass information between program components differently. Fortran passes arguments by reference all the time, where as C passes some arguments by value and some by reference. There is no easy way to handle this difference between C and Fortran when building and using DLLs. When using the default conventions, you will need to pass all variables explicitly be reference from C. Conversely, when using C or STDCALL, you will need to tell Fortran to pass variables by value.

The exported function name changes depending on the standard used. For example, the exported name for `sineT` with the default convention is `SINET` and with the STDCALL convention the name will be `_sinet@16` The exported name can also be set manually by adding the `alias` statement to the declaration.

```
        !DEC$ ATTRIBUTES DLLEXPORT, STDCALL, ALIAS:'sineT' :: sineT
```

Recall that dependency walker can be used to verify the exported function names.

Passing Arrays

As mentioned earlier, large arrays can be passed back and forth between the calling program and a DLL easily. There is nothing to worry about when calling a Fortran DLL from a Fortran program. The arrays will be passed correctly. However, when passing an array from any other language to Fortran, the array storage format is important.

Fortran stores its arrays in a column-major order while C/C++/C# store their arrays in a row-major order. This means that Fortran arrays cycle starting with the first index while C starts with the last index. Consider two different arrays: a Fortran array A(2,3) and a C# array A[2,3]. The Fortran array is stored in memory in the following order:

```
      A(1,1)   A(2,1)   A(1,2)   A(2,2)   A(1,3)   A(2,3)
```

While the C array is stored in a different order:

```
      A[1,1]   A[1,2]   A[1,3]   A[2,1]   A[2,2]   A[2,3]
```

This means that when passing arrays from C to Fortran, the C array needs to be allocated and indexed in reverse order in the C program. The revised C# array is A[3,2]:

```
      A[1,1]   A[1,2]   A[2,1]   A[2,2]   A[3,1]   A[3,1]
```

Note that the order the array elements are stored is the same as the Fortran array, except that the subscripts are reversed.

Any dimension array can be used. Just remember to reverse the order of the indices in the array when declaring it and when using it in C. For example, a four-dimension array in Fortran could be defined as `SIMS(nreal, nx, ny, nz)` and the corresponding C# array would be `SIMS(nz, ny, nz, nreal).`

One dimensional arrays can be passed with no special consideration.

There is no need to worry about the different starting indices between C and Fortran. Users familiar with C know that arrays are index from `0` to `nx-1` while Fortran is indexed from `1` to `nx`. The calling program simply passes the starting point of the array in the computers memory and the size of the array. Fortran will index the array from 1 to nx. That is why we can pass multi-dimensional arrays between C and Fortran by changing the index order [3].

Calling a Fortran DLL from a Fortran Program

This example illustrates calling a simple Fortran DLL from a Fortran executable. The first step was to write a subroutine and compile it to a DLL. The following subroutine requires an input double precision number and returns the sine of the number as output. The input number is not modified. The DLL was exported with the alias attribute to control the name exposed in the compiled DLL

```
      subroutine sineT(num, sineNum)
         !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sineT':: sineT
```

```
   real(8), intent(IN)  :: num
   real(8), intent(OUT) :: sineNum
   sineNum=sin(num)
end subroutine
```

The next step was to write a program that used the DLL to calculate the sine of a number. The following program imports the DLL in an INTERFACE block. The interface contains all of the input, output, and DLL information contained in the subroutine. The only information not contained, is the calculation part of the subroutine. Note that the DLLEXPORT has been changed to DLLIMPORT. After compiling the program, the program runs and calls the DLL as though it were embedded in the program.

```
program testFortranDLL
   implicit none
   real(8) :: number, sineNumber
   INTERFACE
      subroutine sineT(num, sineNum)
       !DEC$ ATTRIBUTES DLLIMPORT, ALIAS:'sineT':: sineT
        real(8), intent(IN)  :: num
        real(8), intent(OUT) :: sineNum
      end subroutine
   END INTERFACE
   number=73.0D0
   call sineT(number, sineNumber)
   write(*,'(2(f12.2))') number, sineNumber
end program testFortranDLL
```

Calling a Fortran DLL from a C# Program

There is a bit more work involved in calling a Fortran DLL from a C# program. The structure is very similar to the Fortran program, just with a different syntax.

The Fortran DLL from the previous example was used in the C# program. The first step in writing the C# program is to create a class to import the DLL function. The class states the DLL file name, the function name within the DLL to import and the input/output parameters for the DLL. It is important that the case of the function name matches the case of the exported function in the DLL. If it does not, a DLL not found exception will be reported at run time. The function was exported without the C or STDCALL attributes. That means that all of the variables will be passed by reference, not by value. Since this is different that the C default, the reference attribute is given to the input and output variables in the C# class.

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Text;

namespace testMixedC
{
    class ccgTestDLL
    {
        [DllImport("testDLL.dll")]
        public static extern void sineT(
            [MarshalAs(UnmanagedType.R8)] ref Double inputNumber,
            [MarshalAs(UnmanagedType.R8)] ref Double sineInputNumber);
    }
    class testProgram
    {
      static void Main(string[] args)
      {
         Double number = 73.0;
         Double sineNumber = 0;
         ccgTestDLL.sineT(ref number, ref sineNumber);
         Console.WriteLine("{0} {1}", number, sineNumber);
      }
    }
}
```

Conclusions

Multiple language programming has the possibility to bring many benefits to geostatistical software. They include, but are not limited to: (1) transfer of new algorithms to CCG sponsor companies, (2) integrating custom or new processes into commercial software, (3) building a re-usable base of common code, and (4) an easy was to update core algorithms. Transferring new code and algorithms to our sponsor companies is the most important. This has been done previously with standalone executables. The executables were not user friendly and required data to be exported and then imported back into a piece of commercial software. Now, the new algorithms can be written into a DLL and be incorporated right into a commercial software package. For example, Schlumberger has released an API that allows additional functionality to be added right into Petrel. This allows us to provide a graphical users interface to the end-user that looks and feels familiar to the software that they are used to using.

References

[1] Dependency Walker. http://dependencywalker.com Accessed on July 31, 2007.

[2] J. Sharp. *Microsoft Visual C# 2005 Step by Step*. Microsoft Press, Redmond Washington, 2006.

[3] Using C and C++ with Fortran. University of Utah, Department of Mathematics. http://www.math.utah.edu/software/c-with-fortran.html Accessed on August 21, 2007.