Heterogeneity in Computational Environments

Seongjai Kim*

Abstract

In teaching, learning, or research activities in computational mathematics, one often has to borrow parts of computational codes composed by colleagues or uploaded in public domains. Most of these codes are written in C++, C, Java, F77, or F90, or their combinations. Manipulating these heterogeneous modules seems a necessary experience for students in modern computational environments. This short note discusses strategies for an effective management of such heterogeneous modules, in particular, mixtures of C++, C, and F77. In order to help unexperienced students, automatic generators written in shell scripts are introduced: *CSTART* that generates the main and input/output routines in C++ and C, respectively, and *MKMK* that makes a makefile for an efficient incorporation of various objects, depending on the operating system.

Key words. Data management, multi-language computation, shell script, public domain, software engineering.

AMS subject classifications. 65-01, 65F20, 65Y10, 68N19

*Department of Mathematics and Statistics, Mississippi State University, Mississippi State, MS 39762-5921 USA Email: skim@math.msstate.edu The work is supported in part by NSF grants DMS-0107210, DMS-0312223, and DMS-0609815.

Contents

1. Introduction

Lots of computational routines are available free from public domains. A few examples are NETLIB, MGNET, SPARSKIT, and SMLIB. The public-domain routines have been extensively utilized in both teaching numerical methods and solving various important industrial problems, and saved human efforts from rewriting the same or similar routines again and again.

In writing a code whether utilizing public-domain routines or not, indispensable components for programmers are as follows:

I. Implementation of the main routine that declares variables and arrays and calls required functions and subroutines.

II. Implementation of the input/output (IO) routines to read the data and save the results.

III. To implement the converter routines that produce the algebraic system out of the given data and operators. (Data are often saved in files in a certain format; the programmer has to implement a converter routine that reforms the data to fit the computation routines.)

IV. To get/implement the core computation routines.

A typical structure of computational algorithm is plotted in Figure 1.

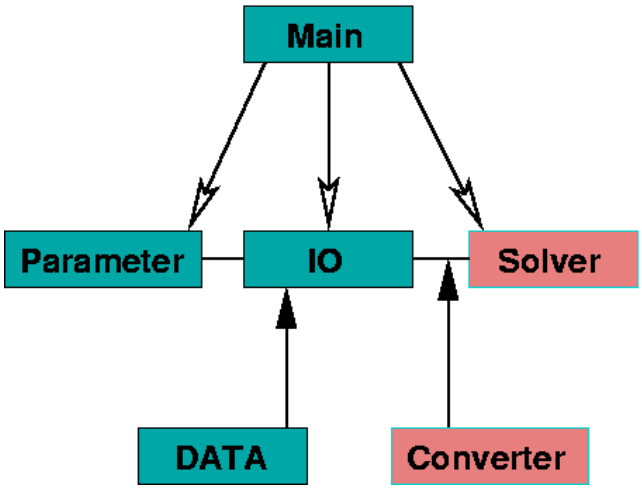

Figure 1: A typical structure of computational algorithm.

Tasks III and IV are the main concerns in classrooms and research activities in computational mathematics. Different numerical methods require different experiences in implementation. Researchers in computational mathematics spend most of

their implementation time on these last two tasks. They often borrow modules, rather than writing all personally, in order to save time and to enjoy a better performance of numerical methods under consideration. However, the available codes may be written in computational languages different from the one the user is familiar with. We will discuss strategies of calling FORTRAN subroutines from a C++/C function, and vice versa.

The first two tasks are related to the algorithm settlement and data management. Whenever the algorithm variables and the data are changed, the programmer should revise the corresponding parts of the program. For experienced programmers whose main interests are not in these tasks, these are just "chores" on which they do not want to spend their priceless time. (I never mean that these are trivial. In connection with data management, the algorithm-settling is very important in software engineering, because it is strongly connected with convenience, portability, reusability, and performance of the program.)

Here the question is: *how can we minimize the time on chores and focus on the main subjects?* The main subjects can be either the development of core computation algorithms or data management, or both. Answering the question seems problematic, but it is clearly important for efficient classroom experiences and research activities as well, in particular, for unexperienced students. We will introduce an automatic code generator, so-called *CSTART* that deals with the first two tasks. A generator of such kind should be able to handle various types of algorithm parameters/variables and heterogeneous data structures. Furthermore, its generated modules should require minimized extra work for revision, preferably, touch-free.

CSTART generates three routines: (1) the main function in C++, (2) a settlement routine in C which reads user-specified variables and settles other required parameters, and (3) a read-data module in C that reads data files, if any. It handles the components such as *Main, Parameter, IO*, and *DATA* in Figure 1, and its generated routines are touch-free. Whenever the structure of data and variables are changed, the programmer can regenerate the routines by running CSTART one more time. To compile and link such heterogeneous modules efficiently, a makefile maker called *MKMK* is introduced. The source codes of CSTART (in Bourne shell) and MKMK (in C shell), their user manual, and example codes are available from `http://www.msstate.edu/~skim/GRADE`.

An outline of the article is as follows. In §2, we consider the interfaces between different computational languages, and implementation strategies for multi-language computation, focusing on the interfaces between C++, C, and F77. In §3, we present

the scope of CSTART, exemplifying a Least-Squares problem to be solved by the QR algorithm. Section 4 discusses the makefile maker MKMK. The last section includes conclusions.

2. Multi-language computation

We begin with how to compile and link C-modules (functions in C++ and C) together with FORTRAN-modules (functions and subroutines in F77 and its later versions) for an executable.

2.1. General remarks

For an efficient programming, multi-language implementation seems indispensable. Due to the time limitation, it is not relevant to implement every line of routines for the given problem. Even if time is enough, we do not have to waste energy in duplicating former works! Public domain codes are often well written, due to experiences and strategies from lots of computational mathematicians, and show satisfactory numerical performances. Most of such codes are implemented in F77 in 70s and 80s and still being utilized extensively in scientific computation.

Even though F77 and later versions (F90 and F95) show advantages in scientific computation, they are still not so convenient in system cooperation and dynamic array allocation as much as in C++. Many modern computation packages incorporate C++ headers for a dynamic array allocation, object-oriented programming via class libraries, and system cooperation.

A computation algorithm can be defined as a set of well-organized functions. Different functions can be written in different languages. Since different languages may have different conventions in the function interfaces, a special attention is required in combining these routines. As far as the function interfaces are explicitly matched each other, any objects produced from any different languages can be linked for an executable. The interface conventions built in C, C++, and FORTRAN are:

	C and C++	FORTRAN
Function Arguments	value or pointer	pointer, always
Function Name	the same as called	occasionally, post-pend under bar (_)

Roughly speaking, FORTRAN compilers in IBM and HP machines do not post-pend an under bar, while the others do. However, GNU FORTRAN compilers (`g77`) post-pend an under bar in most operating systems.

2.2. When C calls FORTRAN

One can easily figure out from the following example the core of the interface convention when a FORTRAN-module is called from a C-module (the main in C++):

```
#include<iostream.h>
#include<assert.h>

// declaring f77-functions
#if defined (_IBMR2) || defined (AIX) || defined (hpux)
    extern "C" void f77name(int*,float*,double*);
#else
    extern "C" void f77name_(int*,float*,double*);
#endif

main(){
    ......
    // calling f77-functions
    #if defined (_IBMR2) || defined (AIX) || defined (hpux)
        f77name(&nx,&ax,A);
    #else
        f77name_(&nx,&ax,A);
    #endif
    ......
}
```

Here `ax` and `A` are declared as a float variable and a double array, respectively. Except called from the main function in C++ or C, FORTRAN-modules do not have to be declared as an external object. On the other hand, a function in C called from the main function in C++ should be declared as an external object, e.g.,

```
extern "C" void Cfunction(int*,float*,double*); //in C++ main
```

2.3. When FORTRAN calls C

Now, we consider the case that a FORTRAN-module calls a C-module. Assume that the C-module to be called begins with

```
void pcgilu0(nx,ny,itmax,idpcond,level,iterPCG,ierr,tol,
             a,ailu,x,b,wksp)
```

```
int nx,ny,itmax,idpcond,level;
int *iterPCG,*ierr;
double *tol;
double a[],ailu[],x[],b[],wksp[][nx*ny];
{
   ...
}
```

Note that the first five arguments are expecting values; `pcgilu0` cannot be called directly from a FORTRAN-module. To call it from a FORTRAN-module, we need to introduce a connecting module in C, say `pcgilu0c`, given as follows:

```
void pcgilu0c(nx,ny,itmax,idpcond,level,iterPCG,ierr,tol,
              a,ailu,x,b,wksp)
int *nx,*ny,*itmax,*idpcond,*level;
int *iterPCG,*ierr;
double *tol;
double a[],ailu[],x[],b[],wksp[];
{
   pcgilu0(*nx,*ny,*itmax,*idpcond,*level,iterPCG,ierr,tol,
           a,ailu,x,b,wksp);
}
```

Now, a FORTRAN-module can call `pcgilu0c` and eventually `pcgilu0`. In practice, one should save another copy of `pcgilu0c` with the function name replaced by "`pcgilu0c_`" for the case that the calling name is post-pended by an under bar during the compiling of the FORTRAN-module.

In the case that all arguments of a C-module are pointers, the module can be called from a FORTRAN-module directly. Even in the case, one should a connecting C-module for the case that an under bar is post-pended. For example, let `cg` be a C-module in which every argument is a pointer. Then the connecting module can be of the form:

```
void cg_(arguments)
...   /*declarations*/
{
   cg(arguments)
}
```

3. CSTART: a main and IO routine generator

Tasks I and II in §1 can be carried out quickly by running CSTART. To explain its main features, we consider the following problem as an example: find the best-fitting quadratic polynomial $x(t) = x_1 + x_2 t + x_3 t^2$ utilizing the QR algorithm for the data

t_i	-1	-0.75	-0.5	0	0.25	0.5	0.75
y_i	1	0.8125	0.75	1	1.3125	1.75	2.3125

(The example can be found in [6, §3.3].) Let the available code for the QR algorithm be given as

```
void qr(int n,int m,int idata,int level,
        double A[][m],double b[],double wksp[][m],int*ierr)
{
  ...
 }
```

To utilize CSTART, we first save the data and its information in an appropriate format. It is more convenient (and common practice in most industries) to save the data and its information into separate files. Let us select `DATA@` for the data-file and `DATA` for the information-file. Let the data-file `DATA@` include

```
-1  -0.75   -0.5  0.0  0.25   0.5  0.75
 1  0.8125  0.75  1  1.3125  1.75  2.3125
```

Then, the information-file `DATA` can save

```
n1=2 n2=7
data=DATA
```

The information simply means that `DATA@` has data of size 2×7 and the data would be read and saved in the array `DATA` in the program.

Then, we have to save algorithm variables into a set-file, named as `SET_file`:

```
n=7 m=3
idata=1
level=2            /* print out level: necessary parameter */
ATYPE=double       /* array type */
array=A            /* array name */
```

```
narr=n*m          /* symbolic assignment for array size */
wksp=wksp         /* work-space name */
nwksp=2*m         /* symbolic assignment for wksp size */
DTYPE=float       /* data type */
data=DATA0        /* data array name */
dataext="@"       /* file-name extension for data-file */
Headers="A.h"     /* for header declarations */
Calls="B.h"       /* for calling functions */
```

Now we are ready to run CSTART:

```
cstart SET_file <return>
```

Then, CSTART first searches the current working directory to find files whose name ends with `dataext` (for this example, `dataext=@`). The files found are considered as the data-files if the directory has an information-file whose name is the same as the data-file except `dataext` at the end. Then, it produces three touch-free files:

```
Main.cpp  IOset.c  ReadDATA.c
```

(The file names can be differently selected by setting command line options for CSTART.) Here `Main.cpp` is the main routine written in C++, `IOset.c` is a C-routine to read `SET_file` and `DATA`, and `ReadDATA.c` is a C-routine to read data from `DATA@`. The total number of lines for these three routines is 200 to 400, depending on the structures of data and variables. The array opened in the program for the data is `DATA`, due to the user-assignment in the information-file. If the assignment of the data array name in the information-file is removed, the array in the program for the data would be `DATA0`, because of the 10th line of `SET_file`. If there is no data-file in the directory, `ReadDATA.c` will not be produced. Symbolic or string parameters in the set-file and information-files are used for delivering information to CSTART; only and every numeric parameter is run-time adjustable. So the programmer does not have to regenerate the routines for a different geometry of data in the data-file; simply change the information in the corresponding information-file. If the files `A.h` and `B.h` are in the same directory (even though they are empty), the produced routines will be compiled and linked without any error message from the compilers and the linker. Now, you have to edit `A.h` and `B.h` to include declarations and callings for the core computation modules, respectively. `A.h` can include

```
extern "C" void getmtx(int,int,int,int,int,
```

```
            int,DTYPE*,ATYPE*,ATYPE*,int*);
extern "C" void qr(int,int,int,int,
            ATYPE*,ATYPE*,ATYPE*,int*);
```

In `B.h`, we can open extra arrays and save function-callings:

```
ATYPE *b; b=new ATYPE[n]; assert(b!= 0);
getmtx(n1,n2,n,m,idata,level,DATA,A,b,&ierr);
qr(n,m,idata,level,A,b,wksp,&ierr);
```

Here `getmtx` is the converter which produces the matrix to be utilized in `qr`. Note that we have assumed that `getmtx` and `qr` are written in C.

Then, we have finished every implementation step required.

Remark: For any geometry of data sets, one or more, whether having the same structure or not, CSTART will work fine. If the number of data sets is larger than one, it is mandatory to include the variable `idata` in the set-file as in the example. Then, CSTART assigns the alphabetical ordering to the data sets. Details can be found from the user's manual [2].

CSTART has been utilized by junior to graduate students to carry out classroom projects partially in a research level: medical image processing [4], accuracy of ADI for parabolic PDEs [1], reservoir simulation [3], wave simulation [5], etc.. The source codes of the projects used in classes can be found from
`http://www.msstate.edu/~skim/GRADE`.

4. MKMK: a makefile maker

The final step in implementation is to make `Makefile`. If one saves or removes one or more of source files, `Makefile` should be edited to fit the new set of source files. Composing or editing Makefile is not difficult; however, it is often tedious. Here we introduce a makefile maker, called *MKMK* written in C shell. The generator first searches the current working directory as well as system variables to

- find source files (written in C, C++, FORTRAN, etc.),
- determine the language used for the main program,
- give explicit dependences for inclusion files (*.h),
- recognize the type of the operating system, and
- determine necessary libraries for linking.

Then, it produces a makefile, named `Makefile` which incorporates appropriate optimization parameters and libraries for the compilers and linker, depending on OSTYPEs.

The source code and the user's manual for MKMK can be found from the same place as for CSTART.

5. Conclusions

We have considered strategies of compiling and linking modules written in various computational languages. Different OSTYPEs (occasionally, different computers of the same OSTYPE) may require a different form of implementation for the function interfaces. However, I believe one can figure out necessary modifications by investigating inherited conventions after searching the system, even though it is often very tedious. We have introduced code generators, *CSTART* and *MKMK*, which work differently corresponding to the OSTYPE. CSTART generates the main and IO routines, incorporating the user-specified data structure and algorithm variables, while MKMK makes a makefile to compile and link heterogeneous modules efficiently. They are designed to minimize the time on chores and maximize convenience.

Acknowledgment

For usefulness of CSTART and MKMK, the author has been encouraged by positive feedback from students in classes: MA/CS321 and MA/CS422, Spring 1999; MA537, Fall 1999; and MA625, Spring 2000.

References

[1] J. Douglas, Jr. and S. Kim, *Improved accuracy for locally one-dimensional methods for parabolic equations*, Mathematical Models and Methods in Applied Sciences, 11 (2001), pp. 1563–1579.

[2] S. Kim, *CSTART: An automatic generator for main and input/output routines*, Technical Report #99-04, Department of Mathematics, University of Kentucky, Lexington, KY 40506, March 1999.

[3] ——, *Artificial damping in multigrid methods*, Appl. Math. Letters, 14 (2001), pp. 359–364.

[4] S. Kim, R. Cooper, and H. Atwood, *Assessing accurate sizes of synaptic vesicles in nerve terminals*, Brain Research, 877 (2000), pp. 209–217.

[5] S. Kim and S. Kim, *Multigrid simulation for high-frequency solutions of the Helmholtz problem in heterogeneous media*, SIAM J. Sci. Comput., 24 (2002), pp. 684–701.

[6] D. Watkins, *Foundations of Matrix Computations*, John Wiley and Sons, New York, Chichester, Brisbane, Toronto, Singapore, 1991.