CSTART: An Automatic Generator for Main and Input/Output Routines

Seongjai Kim*

Abstract

In teaching computational mathematics, the instructor is often tempted to assign a project to the students that addresses a realistic problem or important aspects in computational mathematics. Among others, *data management*, *multi-language implementation*, and *industrial applications* are quite attractive subjects; these are components of the *general-purpose programming* where the code is reusable, portable, and easy to maintain and apply. However, a project incorporating such tasks is hard for most students to accomplish within a certain amount of time. Also it will take a long time for the instructor to organize a project dealing with one or more of these subjects. In this paper, we introduce an automatic generator *CSTART* that has been developed to make the tasks easy for both the instructor and the students. CSTART produces main and input/output (IO) routines (written in C++ and C, respectively) that are touch-free, portable, and easy to maintain and extend for various structures of data and variables. It does not exclude the inclusion of libraries such as codes in public domains and C++ classes for object-oriented programming. Examples are demonstrated to show its effectiveness in classrooms and research activities.

Key words. data management, multi-language implementation, shell script, general-purpose programming, object-oriented programming, public domain, software engineering.

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

*Department of Mathematics, University of Kentucky, Lexington, Kentucky 40506-0027 USA Email: skim@ms.uky.edu

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, the indispensable components for programmers are as follows:

1. To get/implement the core computation routines
2. To implement the converter or matrix-composer routines that produce the algebraic system out of the given data and operators (The data are saved in a certain format; the programmer often has to implement a converter routine that reforms the data to fit the computation routines.)
3. Implementation of the input/output (IO) routines to read and save the data
4. Implementation of the main routine that declares and calls algorithm parameters and variables, arrays, and the required routines

Tasks 1 and 2 have been mainly concerned 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 first two tasks.

The last two tasks are related to data management and algorithm settlement. Whenever the data or the algorithm variables are changed, the programmer should revise the corresponding parts of the algorithm. For experienced programmers who are not interested in the tasks, these are just "chores" on which they do not want to spend their priceless time. (I dit not mean that these were trivial. In connection with data management, the algorithm-settling is very important in software engineering, since it can improve convenience, portability, 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 this article, we will try to answer the question by introducing an automatic code generator, so-called *CSTART* that deals with tasks 3 and 4. 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

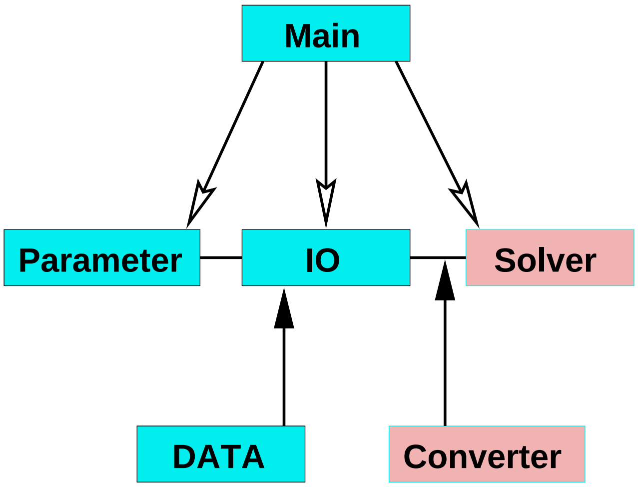

Figure 1: A typical structure of computational algorithm.

minimized extra work for revision, or preferably, be touch-free. It is very important to design an automatic generator to be able to produce touch-free modules, in particular, for efficiency reasons.

A typical structure of computational algorithms is plotted in Figure 1. 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 (blue-colored) parts in the figure such as *Main, Parameter, IO*, and *DATA* and its generated routines are touch-free. Whenever the structures of data and variables are changed, the programmer can regenerate the routines by running CSTART. The routines can be compiled and linked without any other additional work, since they are touch-free. CSTART is so efficient! The main purpose of the article is to introduce the generator CSTART and share its efficiency with computational mathematicians and engineers. It is hoped that CSTART drives both the instructors and students to an efficient and joyful computation world!

The source code of CSTART written in Bourne shell and example codes are available from `http://www.ms.uky.edu/~skim/GRADE`.

An outline of the article is as follows. In §2, we present an example that can be taught in a numerical linear algebra class of junior/senior level, to illuminate the main features of CSTART. In §3, we consider an efficient way of constructing makefile and function interfaces in multi-language computation environments. Multi-language computation is often preferred in efficient programming. It should be noticed that most of codes in public domain were written in FORTRAN in 70s and 80s, while modern computation packages incorporate C/C++ routines for various reasons, in particular, dynamic array allocation, system cooperation, and object-oriented programming [2, 4]. §4 describes the conventions and limitations for CSTART. The last section includes conclusions and discussions.

2 Getting started

2.1 An example

As an example, let us consider the problem: find the best-fitting quadratic polynomial $x(t) = x_1 + x_2 t + x_3 t^2$ utilizing the QR algorithm for the following 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 [5, §3.3].) It is an over-determined problem. The algebraic system to be solved by the QR algorithm is

$$A\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where A is a 7×3 matrix, $\mathbf{x} = [x_1, x_2, x_3]^T$ the unknown, and $\mathbf{b} = [y_1, y_2, \cdots, y_7]^T$ the source vector. It is a typical example of least-square problems. We want to compose a code not only to solve (2.1) but also for similar problems of general data/matrix sizes and source vectors. Then, the implementation tasks are

1. Modules for QR algorithm
2. A converter routine to produce the matrix A
3. An IO routine to read the data
4. The main function

In the following sections, we present the details of implementation utilizing CSTART for tasks 3 and 4.

2.2 The first task: QR algorithm

For the first task, let us assume that we can download/implement a routine in C called `qr` having arguments as in

```
void qr(int n,int m,int idata,int level,
     double a[][m],double b[],double wksp[][m],int*ierr)
   /* INPUT:
```

```
           n,m:        the matrix size (n>=m)
           idata:      data specification indicator
           level:      stdout level
           a[n][m]:    the matrix of the over-determined system
           b[n]:       the source vector
           wksp[2][m]: the work space
           ierr:       error message indicator
        OUTPUT:
           b[m]:       the solution of least-squares problem
     */
......
```

2.3 The second task: converter

The programmer should implement a converter. Let `getmtx` be a converter routine in C beginning with

```
void getmtx(int n1,int n2,int n,int m,int idata,int level,
      float data[][n2],double a[][m],double b[],int*ierr)
......
```

2.4 Tasks 3 and 4: use of CSTART

Tasks 3 and 4 can be done quickly by running CSTART. To so so, we first save the data and required information. 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. Then, the data-file `DATA@` includes

```
-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
```

while the information-file `DATA` can save

```
n1=2 n2=7
```

The information simply means that `DATA@` has data of size 2×7. When the transpose of the data is saved in `DATA@`, the information-file should have a line: `n1=7 n2=2`. Then we have to save algorithm variables into a set-file called `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=DATA           /* 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>
```

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 a information-file whose name is the same as the data-files 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 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 giving 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 n1,int n2,int n,int m,int idata,
          int level,DTYPE*DATA,ATYPE*A,ATYPE*b,int*ierr);
extern "C" void qr(int n,int m,int idata,int level,
          ATYPE*A,ATYPE*b,ATYPE*wksp,int*ierr);
```

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);
```

Now, you have finished every implementation step required.

Remark: For any geometry of data sets, one or more, whether having the same structures 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 previous section. Then, CSTART assigns the alphabetical ordering to the data sets. Details will appear in §4.

2.5 Classroom experiences

If the example in §2.1 is assigned to junior or senior students (without a help of CSTART), most of them will fail to compose a general-purpose code on time. It seems difficult even for graduate students. The instructors in computational mathematics may have observed that students are good at implementing the core computation routines (of course, following the pseudo-code provided in the textbook), but most of them show weekness in data management and general-purpose implementation.

Utilizing CSTART, students in my junior/senior level class could accomplish a project, slightly more complicated than the example in §2.1, on time except three students out of 17. CSTART provides an implementation environment in which students can focus on the core computation routines. Also it can be used to teach techniques in data management, one of the most important components in software engineering.

CSTART has been utilized in classes for research-level problems: medical image processing [1], accuracy of ADI for parabolic PDEs [3], and reservoir simulation. The source codes of the projects used in my class can be found in
`http://www.ms.uky.edu/~skim/GRADE`.

3 Makefile and multi-language computation

3.1 MKMK: a makefile generator

The final step in implementation is to make `Makefile`. If you save or remove 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 I designed a makefile generator, 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++, and FORTRAN),
- 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 the makefile, named `Makefile` which incorporates appropriate optimization parameters for compilers and libraries for the linker, depending on the computers and OSTYPEs being used. To make the makefile, simply write `mkmk` and hit the return key. The following are the options you can apply with MKMK:

```
MKMK - Automatic Makefile Maker
Usage: mkmk [Options]
Options:
   -o[pt]     : to set the optimization parameter empty
   -l[ink]    : to set the linking parameter empty
   -h[elp]    : to get this message
```

3.2 Multi-language computation

For an efficient programming, multi-language implementation is desirable. Due to the time limitation, we cannot 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! It is often preferred to use ready-made codes from public domains. They are well written by many people's experiences and strategies, and they 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. For most programming, the author is utilizing C++ for main, C for IO routines, and f77 for the core computation modules.

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 implementing the routines. As far as the interfaces are explicitly matched each other, any objects from any different languages can be linked into an executable. The interface conventions built in C, C++, and FORTRAN are:

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

I believe that one can easily figure out the core of the interface conventions between the two main computation languages from the following example (C++ main routine):

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

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

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

Remark: Some of GNU FORTRAN compilers (`g77`) installed in HP machines work with the FORTRAN functions/subroutines post-pended by a under bar.

4 The set-file and usage of CSTART

CSTART reads a set-file to generate the main and IO routines. The set-file can include any variables and their initially assigned values as in the following example set-file

```
nx=81 ny=41                       (1)
ax=0.0 bx=2.0 ay=0.0 by=1.0       (2)
n1=21 n2=3 level=2                (3)
last="Done!"                      (4)
                                  (5)
Program=Poisson_Solver            (6)
author=Your_Name                  (7)
date=January_2000                 (8)
comment=""                        (9)
atype=double                      (10)
vtype=double                      (11)
array="SOL"                       (12)
narr="nx*ny*(nt+1)"               (13)
wksp=data                         (14)
nwksp=nx*ny                       (15)
dtype=float                       (16)
data=DATA                         (17)
ndata=n1*n2                       (18)
dataext="@"                       (19)
Headers=A.h                       (20)
Calls=B.h,C.h                     (21)
ConstData=""                      (22)
```

The conventions and limitations regarding to setting algorithm variables are given as follows:

- Each equality sign (=) assumes to have a variable on the left and an initial value of the variable on the right. The initial value should have at most one field. For the empty value, you have to set it explicitly as in the line (9). For the initial values having more than one field, you have to connect them as in the lines (6)-(8). If the right side has more than a value, you have to separate them using commas as in the line (21).
- Variables whose values start with the minus sign (-), dot (.), or numbers 0 to 9 are considered as numbers; the others are recognized as characters or symbolic

assignments.

- Numbers having a dot point (.) are considered as float-point numbers, and others as integers. For example, the variables in lines (1) and (3) will be viewed as integers, those in the line (2) as float-point numbers, and the others as characters or symbolic assignments. Due to the user-assignment in line (11), the float-point numbers are double variables.
- The order of setting does not matter; any ordering of the assignments will work fine.
- The integer variable `level` is mandatory to assign. It chooses a stdout level during the run time of the generated program.
- The variable `idata` should be assigned if the number of data sets which are used one on each run is larger than one. For example, suppose that there are two data sets `DATAa@` and `DATAb@`, and their corresponding information-files. Then, CSTART assigns the alphabetical ordering to the data sets. So, in run time, `idata=1` calls `DATAa@` and `idata=2` can be set to read `DATAb@`. Still you do not have to change the array name inside the program, if you do not want to. For the data sets that are to be utilized on every run of the program, see the special variable `CONSTDATA` below.
- The integer variable `ierr` cannot be assigned in the set-file. It is used as an inside variable to indicate errors, if any.
- The variables on lines (6) to (22) are treated in a special way. They will provide CSTART a set of various required information. They are optional; if they are not set, it will utilize default values and let the programmer know what values are defaulted. They are not case-sensitive; for example, `Author` will work in exactly the same way as `auTHor`.

Here we explain the special variables. The assigned initial values are the default built in CSTART.

Program="":	The program name.
Author="":	The programmer name(s).
Date="":	The date of programming. The default is empty.
Comment?="":	You may add comments at any place, in any line, of the set-file using variables `Comment`, `Comments`, `Comment1`, `Comment2`, etc.. Duplicative usage of one variable is also fine.
ATYPE=double:	It will assign the type of the arrays to be named by `ARRAY` and `WKSP`.

- **VTYPE=$ATYPE:** The type of scalar variables. It will be assigned same as the value for `ATYPE`, if not specified.
- **ARRAY="A":** The name of array.
- **narr=1:** The size of the array named by the `ARRAY` variable. It opens an array as in `$ARRAY=new $ATYPE[narr];`.
- **WKSP=wksp:** The name of the work space.
- **nwksp=1:** The size of the array named by the `WKSP` variable.
- **DTYPE=float:** The type of the array to be named by the data variable.
- **DATA="data":** The name of the data variable.
- **ndata="n1*n2":** The array size of `DATA`. The values `n1` and `n2` represents the geometry of the data; `n1` is the number of rows and `n2` is the number of columns. For the right-side value, you cannot use the multiplication sign (*) more than once. For more complicated assignment of the array size, you may set as follows: `ndata="[2*(n1+1)]*[3*nt2]"`. In the case, CSTART will understand that the data-file has the data of `2*(n1+1)` rows and `3*nt2` columns.
- **DATAEXT="@":** Each set of data can be saved in two separate files: one for the real data, `data-file`, and the other for the information of the data, `information-file`. Here the convention is that the data-file name is made up of the information-file name and an extension. For example, if the information-file name is `HeatSrc`, the data-file name can be `HeatSrc.dat`. In this case, the DATAEXT variable should have its value `.dat`. CSTART will search the current directory to find files whose names end with `DATAEXT`. The files found will be considered as data-files if the directory includes the corresponding information-file. When there is more than one data files, data-related variables can be assigned in individual information-file, instead of setting in the set-file.
- **HEADERS="":** It assigns inclusion files for declarations and libraries. You may assign more than one files by separating them by commas as in `HEADERS=A1.h,A2.h,A3.h`.
- **CALLS="":** It assigns inclusion files for opening extra arrays and calling functions. By utilizing `HEADERS` and `CALLS`, CSTART can generate the main routine touch-free. That is, your own additions

can be implemented in the files included through the two variables; you do not have to touch the main routine. For a new set of program variables or data geometry, you can re-generate new main/IO routines overwriting the old ones.

CONSTDATA="": Sometimes the current directory can include many sets of data. Some can be used optionally, one on each run of the program, while the others can be used on every run. We call such data a `constant data`. The variable `CONSTDATA` or `CDATA` assigns the constant data.

5 Conclusions

We have considered a code generator, called *CSTART* which generates the main and IO routines, depending on the user-specified data structures and algorithm variables. CSTART has been successfully utilized for students in classes of differential equations, numerical analysis, and numerical linear algebra, with applications to medical image processing, heat transfer, and reservoir simulation. It provides an efficient and convenient implementation environment to the students so that they can focus on the main subjects of the assigned project, which can be either core computation modules or data management, or both. CSTART can be adopted to active researchers. In checking new ideas in computational mathematics, researchers often have to spend a few days to a couple of weeks to compose the main and IO routines for given data and algorithm variables. Using CSTART, such chores can be finished within an hour or a day. CSTART and the makefile maker MKMK have been designed to minimize the time on chores and maximize the convenience, for both the instructor and the students!

The generators CSTART and MKMK are written in Bourne shell and C shell, respectively. So they work in Unix systems and Linux, but they may not work in window systems such as Win98 and NT. They will be translated in PERL to extend its portability up to Windows. The codes used in classroom projects are being organized for a library, called *Graduate Research and Applications for Differential Equations* (GRADE) that can be found from `http://www.ms.uky.edu/~skim/GRADE`.

Acknowledgment

I sincerely thank Prof. R.-C. Li and W.W. Symes for their helpful comments and encouragement. I have been encouraged by positive feedback from students in my classes: MA/CS321 and MA/CS422, Spring 1999, and MA537, Fall 1999.

References

[1] H. L. Atwood, R. Cooper, and S. Kim, *Assessing accurate sizes of synaptic vesicles in nerve terminals.* in preparation, 1999.

[2] T. Budd, *An introduction to object-oriented programming. 2nd Eds.*, Addison Wesley Longman, Inc., Reading, Harlow, Menlo Park, Berkeley, Don Mills, Sydney, Amsterdam, Tokyo, and Mexico City, 1997.

[3] J. Douglas, Jr. and S. Kim, *On accuracy of alternating direction implicit methods for parabolic equations.* Preprint, 1999.

[4] I. Pohl, *Object-oriented programming using C++. 2nd Eds.*, Addison Wesley Lonaman, Inc., Reading, Harlow, Menlo Park, Berkeley, Don Mills, Sydney, Amsterdam, Tokyo, and Mexico City, 1997.

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