THE FRESHMAN PROGRAMMING COURSE: A NEW DIRECTION

William H. Jermann
The University of Memphis

INTRODUCTION

For decades typical Electrical Engineering curricula have included a freshman-level course in computer programming. In earlier days, this course included segments related to operating a card punching machine as well as detailed coverage of the FORTRAN programming Language. Now the course frequently involves use of a more modern programming language such as c or **c++** operating under a system that supports integrated developmental environments [1], [2].

Typical engineering freshmen may already be highly computer literate. Not only may they be competent internet surfers, but they frequently enter college experienced in using many software packages, and may even have developed significant skills in program writing. A traditional freshman-level course in computer programming may no longer be appropriate.

An introductory course that emphasizes program modularity and code reusability may have sound educational benefits. The concept of code reusability not only relates to developing subprograms that can be used subsequently, but incorporates use of previously written code that may have been developed in a different programming language. Course material that emphasizes these concepts may introduce both sound problem solving techniques and principles applicable to engineering design.

WHERE WE ARE AND HOW WE GOT HERE

The University of Memphis is a somewhat traditional academic institution. Rather than being just a major network node connected to student nodes on the network, we still have classes for real-time students, and still view education rather than information interchange as our primary mission.

Yet, the onset of the information age has not completely passed us by. The introductory freshman-level course in electrical engineering at one time concentrated on types of engineering jobs with an introduction to the engineering profession. Now it includes searching the internet for information, turning in assignments both on floppy

disks and via electronic mail, using the Schematics version of Pspice both as a CAD tool and for network analysis, and even developing and running a few C programs.

In this course students develop skills that can yield both steady-state and transient solutions to complicated network configurations. Yet, it is doubtful that they have much understanding of network principles, and surely could not write a set of state equations whose solution yields the dynamic response of a network. If information retrieval were the objective of this course, students would not need to take subsequent courses in mathematics or circuit theory. Clearly information retrieval is not our primary objective.

During the second semester, freshmen take a course in computer programming called: ELECTRICAL ENGINEERING COMPUTATION. Students refer to this course as C or C programming, which is a more accurate title. A few words related to the historical development of this course are appropriate.

At one time, engineering students were formally given instructions on how to use a slide rule. Academic credit was generally not awarded for these seminars. Shortly after computers became popular tools in engineering curricula, FORTRAN or derivations of FORTRAN were taught to engineering students in courses bearing academic credit hours. The courses emphasized syntax and semantics. Since awarding credit for a "skill course" is not appropriate, the titles of these courses frequently involved words such as "problem solving." As courses evolved, some were taught independently of programming languages, using FLOW CHARTING and subsequently PSEUDO code. In many electrical engineering curricula, FORTRAN was replaced by Pascal, a language intended for teaching programming concepts. In order to be more in line with industry, many universities now teach C or C++ during freshman year.

To this date, many in the academic world still associate computer programming with code generation, and therefore consider it a skill. This is why our freshman programming course is called ELECTRICAL ENGINEERING COMPUTATION. Yet, it is questionable as to how much electrical engineering computation can be done by students who have taken no substantive courses in electrical engineering.

THE FRESHMAN PROGRAMMING COURSE

At one time, we taught a course in FORTRAN programming to freshmen. Our dean mandated that programming be incorporated into **all** subsequent engineering courses. But times have changed. ABET still requires computer usage across the entire curriculum. But this does not necessarily mean that a traditional programming language need be taught. In addition to a variety of word processing, spreadsheet, and data base packages, the following packages are available to our students.

LabView
Matlab
Mead
DADiSP/32
Tk-solver
Orcad
Pspice

Some of these are used extensively in required courses. In addition, several other packages are used in electrical engineering electives. And this is just the tip of the iceberg concerning what is available and what will be available in the near future.

It certainly appears that both educational objectives and ABET criteria can be satisfied without teaching a general purpose programming language such as BASIC, FORTRAN, Pascal, C, c++, or Ada. Should a general purpose programming language be taught? Is it still necessary to have a freshman programming course?

We do not have an answer to this question. We teach a first-year course in C. The decision to do this was based primarily on observations of local employment trends. Although educational objectives are our primary goal, we feel we must also be concerned about employment opportunities for our graduating seniors. Our decisions regarding the freshman programming course are based on the following opinions.

1. A programming language should not be taught in a freshman course unless it is used in subsequent courses.

2. A college-level programming course should be considerably different from a high-school course.

3. In an electrical engineering course, software concepts should not be completely isolated from hardware concepts.

4. Only those who are highly competent in the subject matter should teach the course. In addition, the instructor should have demonstrated teaching competence.

5. ONLY STANDARD PROGRAMMING LANGUAGE should be used. No implementation-specific programming statements are accepted. This means no C++ statements in C programs. Violation of this principle results in code that is not transportable. Later in the curriculum students are taught how to incorporate non-standard modules into a program.

6. Engineering applications consist of any programming tasks performed by engineers. These include just about everything except the traditional examples frequently given in textbooks with "engineering applications." (such as the dynamics of a bouncing ball.)

7. Computer programming is bona fide subject matter. A

freshman level course should be of comparable difficulty with traditional courses such as physics, chemistry, and calculus. Successful activities in this course will enhance both problem solving and design skills. Misguided activities can be a serious detriment to the development of good problem solving or design skills.

8. Human beings learn through redundancy. Not all fundamental concepts will be mastered in the first course. In order to learn through redundancy, fundamental concepts must be introduced early. In many textbooks, topics such as development of user-defined subprograms, pointers, data structures, and abstract data types are not covered until later chapters. This precludes the possibility of really learning these concepts in the course.

9. Course content in programming courses has changed significantly and will continue to change. Furthermore, no two competent and experienced teachers will ever agree on the specifics of the course content. But the specifics are not as important as the general educational objectives.

10. DO NOT let the freshman programming course be a NEGATIVE educational experience. Do not have students writing code to model systems they do not understand or design software to perform tasks that have not been well defined. These types of activities violate fundamental principles of problem solving and design.

Consider the following example of how some of the above ideas can be incorporated into the beginning of an introductory college level course. After introducing the organization of a computer as a CPU connected to memory by a data bus and an address bus, it is appropriate to discuss the types of information stored in memory. These consist of INSTRUCTIONS and DATA. Although there are many types of data, three of the most fundamental types are pointers (addresses or address identifiers), character codes, and integers. Shortly after this introduction, students are ready to compile, link and run a C program, and to obtain printouts of both source code and program output. This can be done using an integrated developmental environment or by using command lines. We encourage students to run the first program in more than one environment, such as Windows, Unix, and VMS. A typical first program looks something like the following:

```
#include <stdio.h>
        int main(void)
            {   printf("hello world\n");
                return 0;
            }
```

College students generally do not get excited about the above program. They may have even run the same program back in primary or elementary school. Yet the program involves a number of concepts that can be introduced or reinforced. These include:

Function definitions and declarations
Introduction to a system function that requires dozens of supporting functions
Function argument lists
Initialization of an abstract data type. (A character array terminated with a zero byte)
Transmission of a pointer value to a function
Development of a user-defined function (main)
Use of a function that accepts a variable number of arguments

The above concepts can be illustrated using the following example.

```
#include <stdio.h>
int main(void)
{ char *a = “hello world\n" , *b = “save the %s whales\n";
      printf(a) ;
      printf(a+6) ;
      printf(b,b) ;
      printf(b+8,b+12) ;
      *b = ’L’; *(b+1) = 'o';
      printf(b) ;
      printf("%c %d %d %d\n" ,*(a+10), *(a+10), *(a+11), *(a+12));
      return 0;
 }
```

This program not only reinforces basic concepts, but introduces some other concepts such as

Pointer arithmetic
Initial assignment of character pointer values to character pointer variables
Assignment of character values to variables in an array
Referencing variables through use of pointers
Identifying character codes used in strings

There appear to be good reasons for introducing these concepts very early. The author has found that his freshmen students have little difficulty using pointers to implement complex data structures if fundamental concepts are introduced early and often. On the other hand, he has observed very little success when the concepts are covered near the end of the course.

MODULARITY AND CODE REUSABILITY

Many introductory textbooks discuss modular program development very early but introduce the use of user defined functions much later. Consequently, students are given many assignments before they have the opportunity to incorporate modularity in their program design. Thus they develop the habit of ignoring modular development before they are able to develop and use separately compiled modules. The freshman programming

course should not serve as a medium for developing poor problem solving techniques.

After students have studied and run the "hello world" program, they should be ready to develop and use separately compiled modules. They have already used a function with an argument list (printf), and have developed a user defined function (main). Likewise, they have already been introduced to character arrays and pointers.

Consider the following declarations and definitions, which may be incorporated in a user developed header file.

```
/*File name : cpx.h */

#define cpx double

      void cxadd(cpx *a, cpx *b, cpx *c ); /* c = a + b */
      void cxsub(cpx *a, cpx *b, cpx *c ); /* c = a - b */
      void cxmul(cpx *, cpx *, cpx * ) ;    /* c =a * b */
      void cxdiv(cpx *, cpx *, cpx * ) ;    /* c = a/b */
      void cxset(cpx *c , cpx r, cpx i) ;   /* c = (r,i) */
      void cxprint(cpx *) ;
```

The above function declarations define a set of operations on complex numbers, where a complex number is stored as an array of two real numbers and referenced by a pointer to the first real number in the array. If a few new concepts are introduced, and a little assistance is given, students can write and compile the corresponding functions. Examples of three of these functions are given below.

```
/***** Functions that operate on complex numbers.
       A complex number is represented as a array
       of two real numbers, and is referenced by a
       pointer to the first number in the array. *****/

#include <stdio.h>
#include "cpx.h"

void cxadd(cpx *a,cpx *b, cpx *c)
{ C[o] = a[O] + b[0]; c[1] = a[1] + b[1]; }

void cxset(cpx *a,cpx r, cpx i) /* assigns value to cpx number*/
{  *a = r;  *(a+1) = i; }

void cxprint(cpx *a)
{ printf("( %f  %f)\n",a[0],a[1]);
}
```

After these modules have been compiled, a main function can be written, compiled, and linked with the object code obtained from the above file. An example of a main program is:

```
#include "cpx.h"
int main(void)
{ cpx a[2],b[2],c[2];
       cxset(a,3,4);  cxset(b,5,-12) ;
       cxadd(a,b,c) ;
       cxprint(c) ;
       return 0;
}
```

By developing subject matter in this way, students learn to write independent program modules almost at the very beginning of a course. If subsequent assignments require use of complex number operations, the relationship between modular development and code reusability is clearly illustrated. But the concept of using previously written programs does not start with newly written code. Consider the DEFINITION statement in the header file cpx.h.

```
#define cpx double
```

Suppose this statement is changed to

```
#define cpx float
```

Then the computer representation of a complex number is the same as the FORTRAN representation of a complex data type. This opens the door to linking compiled C programs with FORTRAN developed object libraries. A hugh amount of scientific oriented software has been developed in FORTRAN.

Our students write C programs that use FORTRAN libraries. This is currently done in a subsequent course called Matrix Computer Methods in Electrical Engineering. But most of the concepts are introduced in the first programming course, and reinforced in the subsequent course. By using subprograms in the extensive IMSL FORTRAN library, students are able easily solve a variety of complex problems. A FORTRAN library is a compiled set of code designed for FORTRAN users. It certainly does not have to be written in FORTRAN. It appears that much of the IMSL FORTRAN library that we use was developed in C [3].

The following example illustrates using the the IMSL subroutine EVLRG in a C program to find the eigenvalues of a 5 by 5 matrix. Recall that arguments in a FORTRAN array are transmitted by reference, and that two-dimensional arrays are stored "columnwise" rather than “rowWise.”

```
#include "cpx.h"
#include “match”
#define D double *
#define F float *
/** Example illustrating use of the  IMSL FORTRAN subprogram,

      subroutine evlrg(n, a,ldm,evec)

 that finds eigenvalues of n by n matrix a with leading dimension
```

```
 ldm, and stores them in complex vector evec  **/

int main(void)
{   void evlrg(int *n, float * fa, int * rowdim, float * ans) ;
    double a[10][10];  int i,j,ten=10;
    float fa[10][10], fv[20];
    Cpx X[2];
      i = 5; /** find eigenvalues for a 5 x 5 matrix **/
         matread(i,i,(D)a,10) ;  matprint(i,i, (D)a,10);
         ctofor(i,i,(D)a,10,(F)fa,10) ;
         evlrg(&i,(F)fa,&ten,(F)fv) ;
         for(j=0 ;j<2*i;j+=2)    /** print complex eigenvalues **/
                 {   x[0] = fv[j];  x[1] = fv[j+1];
                     cxprint(x) ;
                 }
          return 0;

}
```

In the above example, the function CTOFOR is used to convert a C 2-dimensional array to a FORTRAN 2-dimensional array. Declarations for this function and other supporting functions are given below.

```
/** file is called matc.h **\
#define CASTC double
#define CASTF float

void cput(int i, int j, CASTC x, double * a,int coldim) ;
CASTC cget(int i, int j, CASTC *a, int coldim) ;
void forput(int i, int j, CASTF x, CASTF *a, int rowdim) ;
CASTF forget(int i, int j, CASTF *a,int rowdim) ;
void ctofor(int m, int n, CASTC * c, int coldimc, CASTF *f,
                                int rowdimf) ;
void fortoc(int m, int n,  CASTF *f, int rowdimf, CASTC *c,
                                int coldimc) ;
void matread(int m, int n, CASTC *c, int coldimc) ;
void matprint(int m, int n, CASTC *c, int coldimc) ;
void complexfortoc(int m, int n, CASTF *f, int rowf, CASTC *c,
                                int colc) ;
void complexctofor(int m, int n, CASTC *c, int colc, CASTF *f,
                                int rowf) ;
```

Code that implements these functions is given below.

```
#include <stdio.h>
#define CASTC double
#define CASTF float
#include "matc.h"

void cput(int i, int j, CASTC x, CASTC * a,int coldim)
/** puts x in position a[i-1] [j-1] in a 2-dim array
      whose second dimension is coldim ***/
{ a[coldim*(i-1) + (j-1)] = x; }
```

```
CASTC cget(int i, int j, CASTC *a, int coldim)
{ return a[coldim*(i-1) +(j-1)]; }

void forput(int i, int j, CASTF x, CASTF *a, int rowdim)
{ a[rowdim*(j-1) + (i-l)] = x; }

CASTF forget(int i, int j, CASTF *a,int rowdim)
{ return a[rowdim*(j-1) + (i-l)] ; }

void ctofor(int m, int n, CASTC *c, int coldimc, CASTF *f,
                          int rowdimf)
{ int i, ; CASTC x;
     for i = 1; i <= m; i++)
         for (j=1; j<=n; j++)
            { x = cget(i, j,c,coldimc) ;
              forput(i,j, (CASTF) x,f,rowdimf) ; }
}

void fortoc(int m, int n, CASTF *f, int rowdimf, CASTC *c,
                           int coldimc)
{ int i,j; CASTF x;
      for(i = 1; i <=m; i++)
          for(j = 1; j <= n; j++)
              { x = forget(i,j, f,rowdimf) ;
                cput(i,j,(CASTC) x,c,coldimc) ; }
}

void complexfortoc(int m, int n, CASTF *f, int rowf,
                           CASTC *C, int colc)
{int i,j; CASTF x;
    for (i=1;i<=m;i++)
        for(j=1; j<=n; j++)
          { x = forget(2*i - 1,j,f,rowf);
            cput(i,2*j - 1, (CASTC)x,c,colc) ;
            x = forget (2*i,j,f,rowf) ;
            cput(i,2*j,(CASTC)x,c,colc) ; }
}
void complexctofor(int m, int n, CASTC *c, int colc,
                           CASTF *f, int rowf)
{ int i,j; CASTC x;
     for (i=1; i<=m; i++)
         for(j=1; j<=n; j++)
           { x = cget(i,2*j -1,c,colc);
             forput(2*i -1,j, (CASTF)x,f,rowf) ;
             x = cget(i,2*j,c,colc) ;
             forput(2*i,j, (CASTF)x, f,rowf); }
}
void matread(int m, int n, CASTC *c, int coldimc)
{ int i,j; float x;
       printf(" Please enter %4d numbers: \n\n'',m*n) ;
       for(i = 1; i<=m; i++)
           for(j=1; j<=n; j++)
              { scanf("%f", &x);
                cput(i,j,(CASTC) x,c,coldimc) ; }
}
```

```
void matprint(int m, int n, CASTC *c, intcoldimc)
{int  i,j;
      printf("\n\n" ) ;
      for(i=1; i<=m; i++)
          { for(j=1; j<=n; j++)
                  printf("%8 .3f",cget(i, j,c,coldimc) );
             printf("\n") ;
          }
}
```

A more detailed discussion related to using FORTRAN libraries is given in reference [4].

CONCLUSIONS

We feel comfortable with the present objectives and presentation of our freshman course in computer programming and in use of computer material in subsequent courses that reinforces concepts developed in the first course. We are still searching for the "ideal" textbook, realizing we will never find one.

Based on observation of the recent history of computer related courses, it is not surprising that such courses are dynamic. But even though we anticipate significant changes in the details of subject matter, we still feel it is worthwhile to emphasize concepts related to modularity and code reusability.

REFERENCES

[1] N. Graham, LEARNING C, McGraw-Hill Inc., 1992.

[2] G. Bronson, C FOR ENGINEERS AND SCIENTISTS, West Publishing Company, 1993.

[3] FORTRAN SUBROUTINES FOR MATHEMATICAL APPLICATIONS, Version 2.0, IMSL Inc., Houston TX, 1991.

[4] W. Jermann, "Using FORTRAN Libraries in C Programs", TRANSACTIONS OF COMPUTERS IN EDUCATION, Autumn, 1995.

WILLIAM JERMANN is a Professor of Electrical Engineering at the University of Memphis, Memphis TN 38152.