Structured-Programming Concepts Revisited
By Demetria Ennis

Abstract
Learning to use a programming language effectively can be aided by an understanding of structured-programming principles, problem-solving techniques, testing and debugging strategies, and algorithm design. The concepts of structured programming, an old methodology, have been recognized for their ability to shorten development time, create more readable code, and better organize the design process. This article addresses the profitable aspects of structured programming and discusses the benefits and shortcomings of this methodology when used with teams of students in a beginning C programming course. The introduction of programming language instruction occurred during the mid-1980s and stemmed from the need to help students learn to use hardware to solve practical problems. Many people thought instruction in a programming language was essential because it could teach thinking skills, increase problem-solving and transfer abilities, instill logical principles, and teach decomposition and structure (Clement, Kurland, Mawby, & Pea, 1986; Faye & Mayer, 1988; Lehrer & Smith, 1986; Linn, 1985; Mann, 1986). Early studies on the benefits of programming instruction substantiated many of these claims, but other studies refuted some of them (Palumbo, 1990; Shaw 1984). Before structured programming was widely accepted, coded solutions were developed in an arbitrary manner; they were based solely on the organizational skills and abilities of the programmer. A programmer’s abilities still greatly influence design, but the introduction of organized programming principles makes program creation more productive, more formal, and less arbitrary. This article focuses on the benefits of the structured-programming methodology and discusses the results from a classroom exercise.

Modularity
One form of problem decomposition, modularity, essentially means breaking problems into smaller, more manageable units, subroutines, or functions that perform tasks when invoked. Each unit or module is an independent entity that represents one unique and logical program function: It is bound by one entry and one exit point. These independent entities are typically developed by individual programmers or teams of programmers after the main or controlling module has been created. The main module or driver typically contains the libraries that define all of the data structures used throughout the program. It also specifies the arguments that are needed by subroutines. The driver contains the overall structure of the programming solution.

Background on Structured Programming
Structured programming is a methodology that lends order and discipline to the creation of computer programs. It first appeared in the late 1960s as a series of rules for program construction. Several computer scientists are credited with this methodology: Dijkstra, Hoare, and Wirth (Luehrmann, 1983). The principles of this model can be applied to many aspects of the programming process, but it is most of-
modules, and the data passed between these entities. Flowcharts, automated charts, data flow diagrams, and pseudocode are often used to express the activities that are performed inside modules. Saxe (1985) defines other design tools that aid development: visual table of contents; hierarchy—input, process, and output; and logic structures. These tools help communicate the purpose of the code to those who will use, maintain, or modify the final product.

Brainstorming occurs often during modular designs. This activity permits programmers to think through their solutions before any code is written. Actual coding takes place after solutions are well defined and designed. This keeps programmers from rushing into narrowly defined solutions that might prove disastrous by causing additional bugs, requiring extensive revision, and limiting program functionality.

Many researchers have studied modularity’s effect on programming. The benefits reported include greater program quality, fewer repairs, and easier modification (Banker et al., 1993; Korson & Vaishnavi, 1986; Vessey & Weber, 1984).

Four experiments conducted by Korson and Vaishnavi (1986) with two teams of experienced programmers revealed that programmers following modular principles made faster code modifications in three out of four experimental conditions: information hiding, reusable modules, and chunking. Although this is a positive finding, it is important to remember that only 16 participants were studied.

**Reliability**

Often synonymous with software quality, reliable code performs as expected each time it is invoked. Faultless code is impossible to create, but structured programming techniques, careful planning, and program decomposition can help make code more tolerant of small mistakes. Other methods that improve reliability are the use of formal languages, redundancy, statistical correlations, and mathematical proofs (Littlewood & Strigini, 1992). According to Conte, Dunsomre, and Shen (1986), several characteristics of reliability are consistency, completeness, and robustness. Software solutions that are consistent, complete, and robust save time and are cost-effective because they limit the number of repairs and modifications.

Consistency refers to the way the software performs over time. Are the desired results produced each time the program executes? If not, then major revisions may be needed.

Completeness addresses the functionality of the software. Does the program accurately address the needs of the user or client? If not, the software may be useless to a client. It may be considered an expensive development effort that serves no purpose.

Robust code performs without error in unexpected situations. For example, if unexpected user data are entered, the software denigrates gracefully because it contains data-validation procedures. Instead of "crashing," an invoked error-recovery routine may direct the user to enter appropriate data.

**Less Complex Code**

Complexity increases the possibility of program errors (Littlewood & Strigini, 1992). Many factors make programs complicated: number of lines of code, types
of control structures, composition time, numbers of programmers involved, and the types of data structures used (Shneiderman, 1986). Other factors like the nature of the programming solution and the operating environment may also contribute to program complexity. Branching or jumping to other segments within the same program via goto or call actions also adds intricacy. Unstructured coding practices permit each statement to be the target of such branching. This multiplies the number of possible paths and makes the code more perplexing. The structured philosophy avoids the use of goto and reduces complexity by having one main module begin and end program execution. Control is returned to the invoking module, and each module represents only one unique logical program function.

A Classroom Example
This article reports the results of a classroom project undertaken with students in a beginning C programming course. This activity was initiated to introduce team dynamics and structured-programming concepts and to analyze students’ final products to determine whether they reflected elements of structured programming. In addition, it permitted discovery of an advanced data structure and examined students’ perceived ability to apply what they learned. One group of students participated. A five-month programming course provided the setting. Instruction in the fundamentals of the structured paradigm and the basic constructs of the C programming language occurred before the structured exercise. Information about the participants, data collected from students, and the results follow.

Participants
A class of 21 students engaged in this activity. Of these students, 11 had prior programming experiences. Many had used Fortran, BASIC, Pascal, Assembly, QBASIC, COBOL, or C. One student had studied C for six months. Twenty percent of the experienced students had never been introduced to structured-programming precepts, but 80% had knowledge of or experience with this philosophy. Nine other students indicated that they did not have prior programming experience or knowledge of structured-programming concepts. Every student except one began the course with the same level of knowledge regarding C. The prior programming experience of one student surveyed, though, is not known.

The Problem
After a three-month introduction to C—which included weekly worksheet exercises, programming assignments, discussions, in-class laboratory time, and an introduction to structured programming—students were given a multipart programming assignment taken from their textbook. The program required interactive input, performed calculations, permitted students to learn about C’s data-structure type, and required output. It was selected because it was challenging, and because it enabled students to engage in some discovery and collaboration. Further, the text provided additional resources—programming exceptions, clarifications, and other examples that could aid students.

Students were told to organize themselves into five teams of four members each. They were also told that this team approach would simulate some of the activities of a large corporate programming environment. Each team member had a specific role and function: Team Leader, Documentation Specialist, Programming Component 1, and Programming Component 2. The teams used class time to organize, assign tasks, and debug their source code. Each team had to develop the individual tasks required for each of the four roles, identify criteria for evaluating performance, and complete its assigned module.

Following a modular approach, Team 1 functioned as the root or main module; its job was to collaborate with other team leaders, set deadlines, diagram the entire program, and define the data structures used. Every other team was required to collaborate with Team 1, provide a working version of its module, and explain any parameter passing that occurred. As an aid, each team was given the choice of writing its code from “scratch” or using an incomplete template that represented some of its design duties.

Data Collected
The programming project continued for seven weeks, after which a student survey was administered. The final program and all schedules, documentation, and flowcharts were collected and analyzed.

Results
This project was initiated for the following reasons:

- To introduce students to team dynamics and structured-programming concepts.
- To analyze students’ final products to determine whether students’ used structured-programming precepts.
To permit discovery of an advanced data structure.
To examine students' perceived abilities to apply structured-programming concepts.

**Team Programming**
On the whole, students responded positively to the team-programming approach: 13 rated this approach very good or good, 7 provided neutral responses, and one indicated that the team-programming experience was negative. Student comments on the value of the team-programming assignment follow:

I wish I could say that I learned the benefits of group work; however, our group began falling apart at the end leaving only a couple of us to complete the assignment. The project gave me an understanding of structures; however I feel I could have learned more from several smaller individual projects or assignments relating to structures.

I didn't need the interpersonal skills training—I already had those skills. I learned about the structure construct.

**Final Product**
The completed program contained some of the concepts discussed in class regarding structured programming (e.g., modular program modules, a single entry and exit point, only one parent invoking a subordinate module, and control returning to the invoked module). However, limited documentation was provided, and few flowcharts or other design tools were used by students. Many of them used the templates heavily without demonstrating much creativity in their final module. The final project was not as cohesive, creative, or well-documented as expected, but it did perform the functions required.

**Advanced Data Type**
Some students learned the purpose and applicability of the structure, one of C's derived data types. However, several students indicated that they would have had a greater appreciation for this data type if more discussions, more in-class time, additional assignments, and a longer activity were undertaken.

More time on the programming exercise was a sentiment expressed by several students. Time limitations seemed to produce anxiety and frustration among students. Even though this was a seven-week project, 15% of the students responded that time hindered their performance. Team 1's leader wrote:

We were] taxed with too much! Also, due to the time line, we had to wait till the last minute to pull it all together. We spent lots of time outside of class while the rest of the class only needed to spend class time. Bottom line: I would have rathered we work on a similar, scaled down project...individually and I could have learned the same concepts by making the program structured, well-documented, etc.
Another member of Team 1 wrote:

I would have preferred continuing lecture. I feel I personally missed out on information that would have helped me understand this complex language a great deal more. I learned from the project, but not as much as I could have from lecture. The project is a wonderful idea, but time limitations make it impossible to complete satisfactorily. As [Team 1], we felt the pressure of time limitations and were not completely satisfied with the final result.

Group sizes (15% would have preferred smaller groups) and expanded responsibilities for every group member were identified as necessary project additions. One student expressed the following sentiment:

Unless all the members are actually programming, the exercise is not valuable for the majority of the class. The person who actually did the program for our team really learned the concepts involved. The others did not.

In addition, 15% of the student's surveyed indicated that the templates for the modules should have been withheld, even though the choice to use these was left entirely to each team. One student expressed the use of templates this way:

Do not provide the code! I believe the concept of the group project is excellent, but I believe a much greater amount of knowledge would have been gained by doing it from scratch.

Application of Structured Programming

Lastly, this project was undertaken to determine whether students felt they could apply structured-programming concepts. This question requires further study. More than half of the participants had prior programming experience and knowledge of structured programming, which could have enabled them to apply this paradigm. Of the students with prior programming backgrounds, 90% responded positively when asked if they could apply structured-programming concepts, and 10% did not respond. Students without prior programming experience who were asked the same question responded in this manner: 60% affirmative, 30% negative, and 10% no response. No significant differences were noted in the perceived ability of either group (with and without programming backgrounds) to apply structured-programming precepts.

Conclusion

Students should be encouraged to use the structured-programming paradigm to take advantage of its benefits: modular programs, and readable, reliable, and less complex coding projects. It is beneficial to implement team-programming projects that are structured, solitaire programming concepts, and challenge students to develop interpersonal skills such as brainstorming, working with others, and leading projects. Longer timelines, specific individual responsibilities, limited use of templates, student organization, and problem decomposition can facilitate successful team-programming approaches and the acquisition of structured-programming concepts.

References


Contributor

**Dr. Demetria Ennis**, Assistant Professor of Technology and Cognition, University of North Texas, Denton, TX 76203-6857; 817/565-3536; Fax 817/565-2185; ennis@unt.edu