INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

THE EFFECTS OF LEARNING "C" PROGRAMMING ON COLLEGE STUDENTS' MATHEMATICS SKILL

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

Doctor of Philosophy

By

William Franklin Stockwell
Norman, Oklahoma
2002

UMI Number: 3045840

UMI Microform 3045840

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

THE EFFECTS OF LEARNING "C" PROGRAMMING ON COLLEGE STUDENTS' MATHEMATICS SKILL

A Dissertation APPROVED FOR THE
DEPARTMENT OF INSTRUCTIONAL
LEADERSHIP AND ACADEMIC CURRICULUM

BY

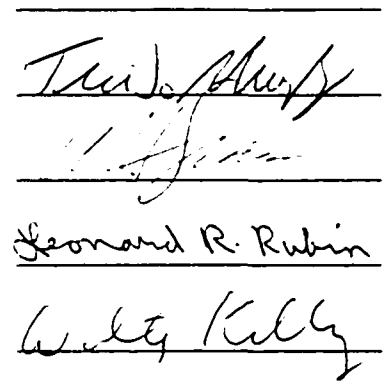

Table of Contents

CHAPTER 1 – INTRODUCTION.....1

CHAPTER 2 – REVIEW OF THE LITERATURE.....6

CHAPTER 3 – PROCEDURE.....20

CHAPTER 4 – ANALYSIS OF THE DATA AND RESULTS.....25

CHAPTER 5 – DISCUSSION, SUMMARY AND CONCLUSIONS.....27

REFERENCES.....30

APPENDIX A – PRE-TEST.....34

APPENDIX B – POST-TEST.....37

APPENDIX C – DATA.....41

APPENDIX D – ELEMENTS OF THE C PROGRAMMING LANGUAGE...43

APPENDIX E – ON THE CHOICE OF THE C PROGRAMMING LANG.....72

CHAPTER 1

INTRODUCTION

Do computer programmers have it easier at learning mathematics? Over the last thirty years there have been a number of research efforts that attempted to show a positive effect on mathematics learning from computer programming. Johnson and Harding (1979) stated:

> One of the major areas of interest is how computer programming by students in a laboratory-like context contributes to the learning of selected concepts and problem solving. Hatfield reported on an extensive review of the literature in this area and indicated that although the evidence from empirical research is minimal at best, there is considerable support for this approach (a computing laboratory) as an aid in the study of many mathematical concepts and for mathematical problem solving. This view is also supported by the Conference Board of the Mathematical Sciences (CBMS) recommendation that there is a need for "the development of short modules (or units) using computers in problem-oriented situations that emphasize the application of mathematics to the range of problems relevant to today's society. (p. 37)

Evidence that learning computer programming is expected to enhance students' problem solving skills as well as their development of other mathematical skills is the large number of textbooks on programming that have the words "problem solving" in their title. Further, it is reasonable to suggest that students in programming classes must exercise their abstract thinking abilities. In addition, if the students' programs do not

work, debugging the logic of the programming code is often a major problem-solving task in itself. Chin and Zacker (1985, p. 9) state that "Perhaps the most time intensive and frustrating part of programming is debugging, which requires a great deal of persistence and effort."

The literature shows that there is a large body of opinion that computer programming should enhance cognitive skills. In spite of this, however, there is scant empirical evidence supporting that belief.

A pilot study was conducted in an attempt to gather empirical evidence. An introductory freshman course on computers with a component of programming was offered at a state-supported university in the central portion of the USA. An especially designed pre-test and post-test of selected mathematics problem-solving skills was administered to several sections of the introductory class. The pre- to post-test comparative results showed no increase in problem-solving skills.

The amount of programming actually taught in the introductory classes was minimal. The motivation level of the students was generally low. As a result not much learning of programming took place. The type of cognitive activity involved in programming that might have contributed to mathematical development thus possibly also did not take place.

It was hypthesized that a more motivated group of students might demonstrate a stronger relationship. The study was repeated with sections of the computer science course Programming I in which the C programming language is taught. Nearly all the students who take this course are computer science majors or minors (or otherwise motivated persons who need some programming knowledge for their work). Such

students no doubt have much higher desire to learn programming than those in the introductory course had. Further, rather than testing for problem solving skill enhancement, it was decided to test for increased skills in selected mathematical areas presumed to be enhanced by learning C programming. It was felt that while problem solving skills would be enhanced, more than a semester course might be required to see those kind of results, even though other areas that C presumably enhances might well show up earlier.

What are these mathematical areas that C programming might enhance? One such area is that of functions. In C programming, all subroutines (callable code units) are actually what are called functions. These are subroutines that return a value to the caller and are invoked in a way that looks and acts a great deal like functions do in mathematical usage. For example, consider the mathematical function defined by $f(x) = x^2$. In C one might write this as

```
double f(double x) { return x * x; }
```

Here 'double' is a data type in C that corresponds roughly to the idea of 'real number', the braces enclose the body of the function code, and the '*' denotes multiplication. To actually use the function, a statement such as

```
y = f(x);
```

would suffice, where y and x are variables of type 'double' and x has been assigned some value. For more information about the C language, see Appendix D.

A second example of an area in which some transfer might be expected is in the area of sequences in mathematics compared to arrays in C. A mathematician might speak of a sequence of values $\{a_n\}$ with $a_1 = 3$, $a_2 = 7$, $a_3 = 11$. In a similar context, the C programmer could use a statement like

```
int a[] = {0,3,7,11};
```

Thus the C programmer would be declaring 'a' to be an integer array with a[1] = 3, a[2] = 7, and a[3] = 11. Indeed, the notation here differs from that for a sequence only in that historically, programming languages only dealt with text of typed characters having no subscripts or superscripts. This practice is essentially standard for programmers now and unlikely to change. Some C programmers pronounce a[1] as "a bracket one" while others would just say "a sub one", thus revealing arrays in C as coming from mathematical sequences.

A third potential area of transfer from C programming to mathematics is in algebraic expression evaluation. Programming students must code and understand algebraic expressions and precedence of operators if they are to write meaningful, working programs. This seems almost certain to help them as they deal with such expressions and order of operations in their mathematics classes.

Why was C chosen for this study among the possibilities in more demanding programming languages? Why not some other language? It was determined that the specific choice of programming language is not essential in this connection as long as the programming involved is cognitively demanding and involves analogies to mathematical skills. However C is only language taught at the researcher's university to large enough numbers of students to allow for a reasonable study. Further it was felt that C requires more dedicated students as C is generally harder to learn than many other languages. This seems likely to contribute to finding a positive result in concomitant growth in mathematical skills. Appendix E contains more on the reasons for choosing the C language for this study.

Problem Statement

This study was designed to determine if undergraduate college students show improvement in selected mathematical skills from learning computer programming in C. It was desired to determine if any such improvement was likely due to learning and practice in programming independently of whether mathematics was being studied simultaneously. A comparative observational study was designed (see Chapter 3) to gather data that might help to answer this question. Variables considered in the analysis of these data were:

1. performance on a pre-test of mathematics skills;
2. performance on a corresponding mathematics skill post-test;
3. whether the student was taking a mathematics course in the same semester in which they studied programming in C; and
4. performance on the final exam of the programming course.

CHAPTER 2

REVIEW OF THE LITERATURE

The literature search concentrated primarily on what effect C programming has on a student's mathematics skills, and secondarily on related areas such as the effect of computer programming in general on mathematics skills, and the effect of computer programming on general problem solving. Only one article was found that was specific to the effects of C programming, and its concern was with students of physics. Hence the references found relate to the relationship between mathematics skills and computer programming in general.

The only research article found that had anything to do with the C programming language was Saiz (1994, p. 28). He showed how the C language was an excellent tool for physics students to explore physics ideas, and investigate phenomena for themselves. Most likely, an easier language such as BASIC might be better for most students interested in these things; however, any reports of the use of C in this way are welcome. It is not hard to guess why: to be good at C requires a fairly high level of programming skill, or at least considerable perseverance.

Studies that found positive effects from programming

Nowaczyk (1983, p. 8) found "...evidence that problem-solving ability is an important component in (computer) programming." To those in the field of teaching software construction, this seems obvious, hence all the attempts at providing empirical evidence.

Others looked at the effect of computer programming on students' algebraic skills: Hart (1983, p. 2) stated, "There is now a background of research and development which indicates that there are very strong psychological arguments for approaching (algebraic) concepts through the use of computer programming." Again, ideas such as this played a strong part in the research selections made in this paper. The rationale is simple: the writing of high-level computer programming code involves coding of lots of algebraic expressions. Many times the way these should be coded varies from mathematical usage, but that doesn't change the fact that computer programming students absolutely must learn about algebraic expressions -- order of evaluation, precedence of operators like + and * (times), use of parentheses, and so on.

Powerful support for the idea that learning computer programming might be a route to more general reasoning skills came from Lee & Pennington (1993, p. 131), who found that "extensive training in programming can develop individual component skills, such as general diagnostic skill, which can transfer across domains...". Further, "with an emphasis on descriptions of knowledge and specific methods of training for transfer, this research has found what may be considered a general skill that can be acquired by learning programming." This is only one of several studies that found that learning programming is good for one intellectually. Casey (1997, p. 47) argues that "designing and writing a computer program engages students in the challenge of presenting ideas to and through a computer. Making that program work as intended, however, often tests the real mettle of the student as problem solver." Still another such source is Soloway (1993, p. 21), who argues the case for teaching programming "to the masses" essentially because it is good for them in terms of logical thinking, and benefits everyone.

Another source demonstrating some transfer effect due to programming was Goldenson (1996, p. 154), who reports on three studies he did which "demonstrated some intriguing transfer effects attributable to (computer) programming." In this case he was referring to transfer to the area of writing essays. This is important to the things this researcher is doing because of some important connections between human languages such as English, and computer languages, and also the large amount of similarity in the mental processes required for writing essays and writing computer programs. A little known fact to many is the relationship between linguistics and computer science; sometimes the English and computer science faculty work together!

McAllister (1985, p. 27) found a relationship between problem solving and beginning programming (in LOGO) for children in middle school.

Another positive result was found by Foreman (1988, p. 10), who did a study relating to cognitive style as well as ability relating to computer programming in BASIC at the college level. Foreman concludes: "...field independence, logical reasoning, spatial ability, and direction following were found to be related to computer programming."

Palumbo and Palumbo (1993) did one of a number of studies on how Logo programming enhances elementary students' problem solving skills; this particular study, like most of those, did find a positive result, and is also a later study than most in its group. Palumbo and Palumbo (1993) stated:

> Programming language instruction is another area that has been said to enhance the development of problem-solving skills. The teaching of computer programming may generate transferable problem-solving and thinking skills since

programming deals with both structure and abstraction. Students are required to place items in a logical and precise order. Students must also plan and use organization and design skills when programming. (p. 310)

Another Logo study, this time with college students, was done by Denenberg (1988), who states:

> The last five years of experience teaching CSC111 "Computation, Reasoning, and Problem Solving" clearly shows mixed results on the development of students' reasoning skills. A few never get it at all (these students seem to need a remedial course to learn even more basic skills such as reading, multiplying fractions, etc.), a few swallow the course whole, stretching their minds and confidence in themselves as effective problem-solvers, while most gain an excellent grasp on the Whimbey and Lochhead problem-solving techniques as well as a fairly good understanding of the two types of reasoning and the power of applying them within the context of a computer programming language such as Logo. (p. 12)

Mayer (1975) in studying 176 non-programmers who were taught a programming language, found

> Apparently, the strongest overall pretest predictor in this study was the Algebra Story Test – a test requiring subjects to translate story problems into formulas. This skill seems closely related to skills required in computer programming in general and the high correlations with performance for all six posttest tasks support this view. (p. 732)

Again, this result hardly seems surprising to this researcher, as the process of determining the steps to carry out in order to accomplish a desired goal (as in computer programming) would seem to demand some of the same skills needed for solving "story problems".

Palumbo and Reed (1991, p. 369) showed that "...some problem solving skills of high school students can be positively increased through systematic exposure to and interaction with the BASIC programming language." Another positive result is shown by Soloway et al. (1982), using a mix of languages with college students.

There is a fair amount of evidence linking mathematical and computer skills; in 1994 a study of over one hundred students of a microcomputer class showed that the most significant factors relating to success in that course were grade point average and mathematics ACT score (Seymour, et al., 1994, p. 1338).

Bernardo and Morris (1994) looked at the effect of a high school programming course in various areas of mathematical achievement (including problem solving). They mention that George Polya's problem solving heuristics are extremely similar to the processes employed in the construction of a computer program. In their study they did find a statistically significant impact of learning to program on their problem solving ability (at the .01 significance level).

Reed (1988, p. 56) speaks of the "content-voidness" of programming languages, and speaks to the issue of the power programming languages have to help develop thinking processes in students. The term "content-voidness" refers to the notion that programming languages are generic with respect to content; they can be used in a very wide variety of applications. Reed goes on to state that "programming languages are based on operational logic, an area that modern education has overlooked". Reed

distinguishes operational logic from deductive logic in that operational logic "involves the logic of planning and the rational execution of action". Of course, designing and writing computer programs involves careful, logical planning and precise determination of what is needed in order for the program to carry out the appropriate actions at the appropriate time. Thus the term "rational execution of action" applies very well here.

Harding and Johnson (1979) tried to determine the effect programming has on "selected concepts and problem solving", for university students. They found that programming had a favorable effect on problem solving abilities. They were also interested (for future research) in determining

> what is learned from solving problems with a computer that makes major contributions. Is it:
>
> - the type of thinking demanded for successful programming?
> - the analysis of a problem that is necessary before a program to solve that problem can be written?
> - the trial-and-error behavior that is characteristic of successful programming?
> - the opportunity or provisions for partial solutions and feedback provided by running of programs and studying output?" (p. 54-55)

McCoy and Orey (1988) did a study with secondary school students learning the BASIC language. In their study, programming achievement was significantly related to both problem solving ability and verbal reasoning ability. They also noted improved problem solving ability after one semester of programming instruction.

Negative (or questionable) results on the effects of programming

McCoy (1996, p. 443) summarized a number of studies involving how programming affects problem-solving, and states that the conflicting results found are "possibly explained by two difficulties that have made programming less appealing to mathematics educators: inherently hard-to-measure problem-solving skill and wide variation in programming experiences." Factors such as these played a part in this researcher's choices for experimentation. For example, this researcher believes that one can cut down on the "variation in programming experiences" by teaching a harder language such as C.

Many studies have been done trying to find a link between programming ability and mathematical ability; the results do not always agree. For example Mazlack (1980) found no significant relationship here, while in Fletcher (1984, p. 4) we have "Another interesting aspect of the analysis is that while mathematical reasoning is highly correlated with overall programming ability, it is relatively insignificant for the more fundamental aspects of programming. This tends to indicate that mathematical reasoning is more closely related to primary abilities than the other cognitive processes tested." In other words, the higher the IQ, the more likely the student is capable of mathematical reasoning.

Alspaugh (1972, p. 89), although dated, nevertheless has some important things to say about programming aptitude and especially its relationship to mathematical aptitude. What was noticed is something that has been seen several times, which is that programming and mathematics aptitude don't always act as good predictors of each

other. It was pointed out also, rather interestingly, that one primary assumption in putting computer science as part of the mathematics department (which was done a lot in the past and is still sometimes the case) is that there is a strong relationship between the two. Nevertheless, in the testing that Alspaugh did, results showed that achievement in programming (in FORTRAN and assembly language in this case) had as its major influencing component the mathematical background of the student in question. This conclusion seems to be at odds with the observation that math aptitude is not a good predictor of programming aptitude. Another possibility is that those with a strong mathematics background do have programming aptitude -- it just may not be developed.

The literature shows something of a mix on results relating to the effects of programming on cognitive skills. Salomon and Perkins (1987) have stated that differences in programming instruction is one possible reason some research has yielded significant results in this area while other research has not.

Pea and Kurland (1984, p. 162) tried to see what if any cognitive transfer occurred with children learning Logo. They were not encouraged; however the researcher has come to believe that generally only high school students and above are likely to show any real benefits from programming. Such belief comes from studying Piaget, who discusses a stage of development called "formal operational thinking" which some never attain, but for those who do, it generally occurs on or after age twelve (see http://www.psych.ualberta.ca/~mike/Pearl_Street/Dictionary/contents/P/piaget's_stages.html).

Miscellaneous studies relating to programming

Important to the success of a computer programmer is skill at "debugging" – the process of finding (the hard part) and removing "bugs", or program errors, from written code. At least one study relates to this: Chin and Zacker (1985) state:

> Perhaps the most time intensive and frustrating part of programming is debugging, which requires a great deal of persistence and effort. This may explain why internality is the strongest personality trait associated with high programming performance. The high positive correlation found between internality and programming performance supports the notion that internals would be more likely to succeed at program implementation than externals. (p. 8)

Many of this researchers' sources are teachers of programming and/or mathematics, who in the process of teaching programming become convinced of its power to enhance mathematical learning. One such was Day (1973), who taught a 3-hour class in her mathematics department called "Introduction to Computing". She spent the first month of the course teaching the BASIC programming language, and then had the students choose a programming project to work on for the rest of the semester. Many of the projects explicitly involved a lot of mathematics. After the course, Day wrote:

> Evaluating, I am optimistic about the potential of programming assignments as an aid to learning mathematics. The students seemed to grasp a concept more firmly and be much more interested after writing a program using the principles involved. One somewhat unexpected benefit of this course was that it stimulated

creativity, which was due mostly I think to each student being encouraged to choose his own project. (p. 12)

There is quite a lot of disagreement in the literature as to whether teaching computer programming helps students develop their thinking skills. One source seeks to explain this as follows: Jones (1988, p. 10) says that many studies target elementary students, but this is the wrong group to study according to Piaget. Piaget says the great majority of children can achieve the level of formal operations only at age twelve and beyond. Jones goes on to say that "Effective programming requires that the student (a) understand the problem, (b) design and plan a solution (program), (c) code, and (d) debug the program. Surely, Piaget would classify programming as a formal operations activity...It seems clear that most research has targeted the wrong group." This is one reason that this researcher chose to test college level students, although it is worth pointing out that motivated high school students would also have worked out for these purposes.

Kurtz (1980, p. 114) defined an "ID" level (intellectual development) in his study of students in an introductory programming course. He states

> ID level explained over 80% of the variance in the total test scores, yet only 39% of the variance in programming scores. This result was predicted by the researcher and should not be surprising to anyone who has taught an introductory programming course. Students have many sources of help when writing programs: the instructor, lab consultants, other students, and of course, the computer itself. Programs are graded on a mastery basis, the number of attempts

> or the actual time spent on the program are not considered in assigning a grade. In a batch environment there is no way to control these factors. It would be interesting to see if ID level would be a better predictor of programming performance if programs had to be completed individually, in a certain amount of time, and under the supervision of the instructor. Although this is an interesting research question, it is not a recommended teaching strategy.

On that, this researcher can only agree whole-heartedly.

Greeno (1978, p. 264), in discussing the nature of problem-solving abilities, notes that "The most demanding intellectual problems are problems of composition. In composing a piece of music, a painting, a sculpture, a poem, a novel, or a theory (or, sometimes, an experiment) a person must create an arrangement of ideas whose structure incorporates some significant new understanding." Indeed, this researcher would describe composition of a new and ambitious computer program in much the same way.

Anderson (1982) uses the ACT theory of learning to discuss how the acquisition of cognitive skill takes place. This system, described by Anderson in 1976,

> consists of a set of productions that can operate on facts in a declarative database. Each production has the form of a primitive rule that specifies a cognitive contingency, that is, a production specifies when a cognitive act should take place. The production has a condition that specifies the circumstances under which the production can apply and an action that specifies what should be done when the production applies. The sequence of productions that apply in a task correspond to the cognitive steps taken in performing the task. (p. 369)

Anderson goes on to say:

> The claim is that the configuration of learning mechanisms described is involved in the full range of skill acquisition from language acquisition to problem solving to schema abstraction. Another strong claim is that the basic control architecture across these situations is hierarchical, goal structured, and basically organized for problem solving. This echoes the claim made elsewhere (Newell, 1980) that problem solving is the basic mode of cognition. (p. 369)

A number of references concerned themselves about testing for programming aptitude. In (Bauer, et al., 1968) for example a number of tests for programmer aptitude were correlated with the final grade in a programming course for Army trainees, in an effort to see which tests (if any) would turn out to be good predictors of programmer aptitude. The correlations with standard tests came out fairly high. In (Besetsny, et al., 1993), another such study (this time in the Air Force) showed that the Air Force test for programming aptitude was not a good predictor of such. Bloom (1978) discusses the need for "testing the test" for programming applicants; he mentions that different tests have different objectives, and that use of a test without careful evaluation can lead to serious problems. Another study, this one by Correnti (1969), tended to show that tests for programming aptitude do work, at least up to a point. Ledbetter (1975) did an interesting study on programming aptitude in which he states

> The major conclusion ... from this study is that a very large proportion of college-level students probably possess sufficient aptitude for successful computer programmer training. This aptitude does not appear to be a characteristic of an elite few, as some had surmised, but rather, is probably very widely held in the college population (p. 166).

Lemos (1980) is concerned with the problems inherent in attempting to measure a student's proficiency in a programming language. He found

> a direct relationship between the ability to read programs and ability to write programs. This finding is important since the measurement of reading ability is shown to require a significantly smaller time investment in evaluating student performance (p. 272).

Mazlack (1980) did a long term study in which many students, over a number of years, took the same introductory programming course. Performance in this class was found to correlate at a low level with the IBM Programmer's Aptitude Test.

Two of the studies (Cavaiani, 1989, and Cheney, 1980) concentrated on cognitive style – a personality variable. For example, Cavaiani found

> There is some evidence that certain personality variables are related to performance in computer programming. Studies have shown that subjects identified as having a global cognitive style have greater difficulty solving certain types of problems than subjects having an analytic cognitive style. (p. 418)

A somewhat different approach, also involving high school students, was done by Ennis in 1994. Her intent was to improve students' problem solving ability by combining programming instruction and problem solving instruction. This is interesting as it is right down the alley of that which I am trying to do, however by mixing the two types of instruction it makes it quite difficult to know what to attribute any increase in problem solving ability to. However, it must be noted that she was NOT merely trying to increase problem solving ability for its own sake, but also for the benefits it might bring to her

programming students, that they might become better programmers. Once again, we see the intertwining threads of problem solving vs. programming that appear so interrelated. Interestingly however, in trying to find any significant increase in problem solving ability due to the mix of instruction, none was found (but she did have some "interesting qualitative observations".)

McCoy (1989) discussed problems students have understanding variables and translating English word problems to algebraic equations. For example, if there are 6 times as many enlisted men as officers in a room, is this represented as O = 6E or as E = 6O? McCoy noticed that many students had a hard time with this. This intrigued this researcher to such an extent that it was decided to put such a question into the researcher's pre- and post-tests for mathematical achievement. See Appendices A and B for more about the results of those tests.

While the literature contains a lot of references to the effect of computer programming on problem-solving and mathematics performance, it contains virtually nothing on the specific effects of C programming in these areas. Considering what a dominant computer language C has been over the last two or three decades, and the fact that it has typically been used on many very high level projects (nearly all modern operating systems are written in C), this is rather surprising. The door appears open to research of this nature.

CHAPTER 3

PROCEDURE

As indicated by the literature, a wide variety of attempts to show how computer programming affects problem-solving as well as other mathematics skills have been made, with varied results. Noting the lack of research studies of this type in which the programming was done in the C language, this researcher devised a study in which students receiving instruction in C programming would be tested on their ability to perform selected mathematics exercises at both the start and the end of the C programming course.

Pilot Study

A pilot study was conducted to develop and refine the instruments used in this research project. This study was conducted during Fall 1997 and is referred to in the following discussion of the development of the instruments.

Development of the Instruments

1) The Pre-test (Appendix A) contained 14 questions constructed by the researcher and administered at the beginning of the semester of the study to determine the students' skill level on certain mathematics problems at that point. These questions began as similar to questions asked in the pilot study, but in their final form were quite different, since the type of mathematics and problem-solving skills for which it was being used was different.

2) The Post-test (Appendix A) contained 14 content questions, which had been constructed by the researcher and administered close to the end of the semester of the study to determine any change in the students' mathematics skill levels from the Pre-test.

Definition of Terms

Mathematics Achievement Measure. This is defined as the PostTest score minus the PreTest score for each student. Each test had 14 multiple choice questions, such that a score of −14 to 14 could be obtained. For purposes of this research, we take these difference scores to be meaningful, even though the items on the PreTest and PostTest were not the identical.

Took Math Variable. For each student who took the Pre- and Post-Tests, this was recorded as a one if the student was taking a mathematics course that semester and a zero if not. The students were asked on the PostTest to indicate if they were taking a mathematics course that semester and, if so, to list their mathematics teachers' names on the back of their answer sheet.

Purpose of the Study

This study attempted to determine if there is a positive relationship between the learning of computer programming in C and performance on selected types of mathematics exercises. Variables considered in the analysis of the data were the PreTest score, the PostTest score, the TookMath value, and the score on the final exam of the students' computer class.

Independent Variables

The TookMath value (whether or not each student was taking a mathematics course) is an independent variable. Also the score on the PreTest was an independent variable.

Dependent Variables

The dependent variables were the final exam scores and the scores on the PostTest of mathematical achievement.

General Research Hypothesis

The achievement in selected mathematics areas of students who are taught C programming will see a net increase.

Design of the Study

Instruments

The instruments used in the study were:

1) The PreTest of mathematical skill
2) The PostTest of mathematical skill

Each student's final examination score for their C programming class during the semester of the study was made available to the researcher by the teacher of the student's class.

Population

The one-semester study was conducted at a medium sized state supported university in South-Central U.S.A. Six sections of Programming I took part as the main study group with an original sample size of 173.

Minimum prerequisites for Programming I include two years of high school algebra and also Beginning Programming (in which Pascal is taught).

Design

This study involved a Pre-Test and Post-Test of some selected mathematics skills of students taking a course in C programming. See Appendix A and Appendix B for the actual questions on these tests. The Cronbach alpha coefficient was computed to be 0.63 for the pre-test and 0.61 for the post-test.

Procedures for Data Collection

At the beginning of the study semester (Spring 1998), a pre-test (Appendix A) of fourteen researcher-selected mathematical problems was administered to the students in the six sections of Programming I. A total of 152 students took the pre-test out of 173 who were enrolled at that point.

Near the end of the study semester, a posttest of fourteen questions was given to all the remaining students in the tested sections. See Appendix B for the post-test questions.

The posttest had a question that asked if the student took any mathematics course that same semester, as an additional question 15 that is not included as part of the score. This was used to determine the "TookMath" value for each student.

Final examination scores were obtained from the teachers of each Programming I section for all of the students, so that this data could be used.

Data Analysis

To determine the effect of C programming on the selected mathematics skills, a t-test on matched pairs was used to see if a difference was found between the pre- and post-tests. The t-test was run against the set of C programming students who did not take a mathematics course that semester, but who did take both the pre-test and the post-test so those scores could be paired. Additionally, a multiple regression analysis was done using the model

$$PostTest = b_1 FinalExam + b_2 TookMath + b_3 PreTest + c$$

This was run against the full data set of students that took both the pre- and post-tests, whether they took a mathematics class that semester or not. The idea was to try to glean more information about the relationship between the variables in the study.

CHAPTER 4

ANALYSIS OF THE DATA AND RESULTS OF THE STUDY

This study was designed to determine if there is a positive relationship between learning to program computers in C and selected areas of mathematical achievement. Overall results on the pre-test and post-test are shown in Appendix C.

The group of students used for the t-test was the C programming students who took both the Pre-Test and the Post-Test of mathematical skills but who were not taking a mathematics course that semester (sample size of 40). The scores these students obtained were matched into pairs. The mean difference score between Pre- and Post-Test was 0.6. Although the Pre- and Post-Test were not the same test, for purposes of this research the difference score is taken to be a meaningful statistic. The standard deviation of the differences was 1.82. The t-value obtained was 2.06 (39 degrees of freedom), which is significant at the .05 level.

The data used here constitute a subset of the full set of data for the multiple regression that is done below, and may be viewed in Appendix C. That data set includes all 76 students who took both the pre-test and the post-test, regardless of whether they took a mathematics course during the study semester. In the listing is a column that indicates whether a math course was taken (1 for yes, 0 for no) and the students in the matched pair analysis above are precisely the 40 in the full listing who did not take a math course.

A simultaneous multiple linear regression was performed using Post-Quiz as the

dependent variable and final exam, Pre-Quiz, and the dichotomous variable of TookMath as predictor variables. The first iteration of the analysis showed two variables contributing significantly to the prediction of the Post-Test score. The variable TookMath was not significantly correlated with the criterion variable, Post-Quiz, when considered by themselves, and its partial contribution diminished as the other two variables were introduced. Therefore, in the interest of presenting a parsimonious equation, the analysis was re-run excluding TookMath. Those results indicated a prediction equation of

$$\text{Post-Quiz} = 0.369*\text{Pre-Quiz} + 0.083*\text{FinalExam} - 0.958.$$

The adjusted R^2 value was 0.665, significant beyond the 0.01 level. See Appendix C for a table showing the data that were used in the multiple regression.

CHAPTER 5

DISCUSSION, SUMMARY, AND CONCLUSIONS

During the study semester at the researcher's university, pre- and post-quizzes were administered to students in several sections of the Programming I course in C programming to assess any possible improvement in their mathematics quiz scores over the semester. Students who were taking any mathematics course at the same time were not counted in the result. A t-test was performed against the scores from the pre- and post-quizzes, and for the Programming I students, an increase (from pre to post) was noted which was significant at the .05 level. Additionally, a multiple regression was performed presuming a linear relationship of the form

$$PostQuiz = b_1 FinalExam + b_2 PreQuiz + c$$

and this came out with a good fit ($R^2 = .66$). This was anticipated from the fairly high correlation between the Post-Quiz and the FinalExam scores from the students' Programming I course. Since the correlation between the Post-Quiz scores and the TookMath variable was quite low, this variable was eliminated from the final analysis. Since the quiz questions related to functions, mapping properties, sequences, and expression evaluation, which have analogs in C programming, the conclusion is that there appears to be a strong relationship between the learning of C programming and the learning of certain mathematics concepts.

These results are encouraging, however replication and broadening of the scope of the experiment is called for. It should be possible to expand on the types of mathematics skills where improvement might be expected. Another thing that could be done is to experiment with the effect of high level programming in other languages besides C,

especially in a context where several computer languages might be compared and any significant differences noted. Another area of interest is the relationship between computer programming and problem-solving. However, it isn't always easy to show good problem-solving enhancement in only one semester, and secondly one has to define "problem-solving" in an appropriate way so as to objectify the process. Still, computer scientists are convinced on this relationship – indeed, it is not hard to come up with a strong argument that computer programming is problem-solving. To wit: designing, writing, and debugging non-trivial computer programs requires considerable skill in problem-solving, and a strong case can be made for the notion that such skills are rather widely applicable to many problem-solving areas. Casey (1997, p. 47) makes this point, and the quote is on page 7 of this paper. Also, Reed (1988, p. 56) discusses the "content-voidness" of programming languages, and speaks to the issue of the power programming languages have to help develop thinking processes in students. What Reed is referring to with respect to "content-voidness" is the fact that what a program deals with is abstract and general, and not necessarily specific to any problem domain. This also helps to answer some critics who doubt that instructional courses actually improve students' problem-solving skills. In the area of experimental psychology, there is increasing evidence that good problem-solving requires plenty of domain-specific knowledge (Mayer, '83). However, as was mentioned, the content void aspects of programming are precisely what allow programmers to be problem solvers without a great deal of specific knowledge about the application area.

Many point out the problem-solving aspects of debugging: Chin & Zacker (1985, p. 8), for example (their quote is on page 14 of this paper). In addition, debugging

employs many aspects of general problem solving. Logic, experience, and skill in determining the crux of a problem all contribute. More generally, programming involves integrating correctly implemented algorithms with well-chosen data structures for the problem at hand; opportunities for problem-solving abound. Knowledge of the right data structures and algorithms to use in a situation relates directly to the quality of the implementation, and to how well it solves a given problem. In many programming situations, there is a space-time tradeoff that is affected by the programmer's implementation; finding the right balance between memory use and execution time can be critical. Hence, there are many problems to be solved in a typical programming situation. In order to be successful at programming non-trivial applications, a programmer should have some problem-solving skills.

From the above, one can see the need for more empirical research on the influence of computer programming on developing problem-solving skills.

REFERENCES

Alspaugh, Carol A. Identification of Some Components of Computer Programming Aptitude. Journal for Research in Mathematics Education, 1972, **3**, 89-98.

Anderson, John R. Acquisition of Cognitive Skill. Psychological Review, 1982, **89**, 369-406.

Bauer, R., Mehrens, W. A., & Vinsonhaler, John F. Predicting Performance in a Computer Programming Course. Educational and Psychological Measurement, 1968, **28**, 1159-1164.

Bernardo, M. & Morris, J. Transfer Effects of a High School Computer Programming Course on Mathematical Modeling, Procedural Comprehension, and Verbal Problem Solution. Journal of Research on Computing in Education, Summer 1994, **26**(4), 523-536.

Besetsny, L., Ree, M., & Earles, J. Special Test for Computer Programmers? Not Needed: The Predictive Efficiency of the Electronic Data Processing Test for a Sample of Air Force Recruits. Educational and Psychological Measurement, 1993, **53**, 507-511.

Bloom, Allan M. Test the Test for Programming Applicants Datamation, October 1978, 37-39.

Branca, N. Problem Solving as a Goal, Process, and Basic Skill. Problem Solving in School Mathematics, 1980 NTSM yearbook, p. 3-8.

Casey, Patrick J. Computer Programming: A Medium for Teaching Problem Solving. Using Technology in the Classroom, 1997, by the Haworth Press, p. 41-51.

Cavaiani, T. Cognitive Style and Diagnostic Skills of Student Programmers, Journal of Research on Computing in Education, Summer 1989, p. 411-420.

Cheney, P. Cognitive Style and Student Programming Ability: An Investigation. Assocation for Educational Data Systems Journal, 1980, **23**, 285-291.

Chin, John P., & Zacker, Steven G. Personality and Cognitive Factors Influencing Computer Programming Performance. ERIC Report No. ED 261 666, March 1985.

Clements, Douglas & Sarama, J. Computers Support Algebraic Thinking. Teaching Children Mathematics, February 1997, p. 320-325.

Correnti, R. Predictors of Success in the Study of Computer Programming at Two-Year Institutions of Higher Education (Doctoral dissertation, Ohio University, 1969). Dissertation Abstracts International, 1969, **30**, 3718A. (University Microfilms No. 70-4732).

Day, Jane M. A Course Which Used Programming to Aid Learning Various Mathematical Concepts. ERIC Report No. ED 081 230, 1973.

Denenberg, Stewart A. Developing Reasoning Skills in College Freshmen Using Computer Programming, Collaborative Problem-Solving, and Writing. ERIC Report No. ED 302 231, May 1988.

Ennis, D. Combining Problem-Solving Instruction and Programming Instruction to Increase the Problem-Solving Ability of High School Students. Journal of Research on Computing in Education, Summer 1994, **26**(4), 488-496.

Fletcher, Stephen H. Cognitive Abilities and Computer Programming. ERIC Report No. ED 259 700, 1984.

Foreman, Kim Hyun-Deok. Cognitive Style, Cognitive Ability, and the Acquisition of Initial Computer Programming Competence. ERIC Report No. ED 295 638, January 1988.

Goldenson, Dennis. Why Teach Computer Programming? Some Evidence About Generalization and Transfer. National Educational Computing Conference 1996, Minneapolis, MN.

Greeno, James G. Natures of Problem-Solving Abilities. In W.K. Estes (ed.) *Handbook of Learning and Cognitive Processes Volume 5: Human Information Processing.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1978, p. 239-270.

Hart, Maurice Computer Programming and Algebra. The Nottingham Programming in Mathematics Project. ERIC Report No. ED 227 002, 1983.

Johnson, D. & Harding, R. University Level Computing and Mathematical Problem-Solving Ability. Journal for Research in Mathematics Education, Jan 1979, p. 37-55.

Jones, Preston K. The Effect of Computer Programming Instruction on the Development of Generalized Problem Solving Skills in High School Students. ERIC Report No. ED 302 221, June 1988.

Ledbetter, William N. Programming Aptitude: How Significant Is It? Personnel Journal, March 1975, 165-175.

Lee, Adrienne Y., Pennington, Nancy. Learning Computer Programming: A Route to General Reasoning Skills? Empirical Studies of Programmers, December 1993, p. 113-136.

Lemos, R. Measuring Programming Language Proficiency AEDS Journal, Summer 1980, 261-273.

Linn, M. The Cognitive Consequences of Programming Instruction in Classrooms. Educational Researcher, May 1985, **14**(5), 14-16, 25-29.

Mayer, R. E. Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models. Journal of Educational Psychology, 1975, **67**(6), 725-734.

Mazlack, L. Identifying Potential to Acquire Programming Skill. Communications of the ACM, January 1980, **23**(1), 14-17.

McAllister, Alan. Problem Solving and Beginning Programming. ERIC Report No. ED 259 032, March 1985.

McCoy, Leah P. General Variable Skill, Computer Programming, and Mathematics. ERIC Report No. ED 304 330, April 1988.

McCoy, Leah P. Use of Variables: Algebra-Computer-English Translation. ERIC Report No. ED 308 071, March 1989.

McCoy, Leah P. & Dodl, Norman R. Computer Programming Experience and Mathematical Problem Solving. Journal of Research on Computing in Education, Fall 1989, p. 14-25.

McCoy, Leah P. & Orey III, Michael A. Computer Programming and General Problem Solving by Secondary Students, 1988, Haworth Press.

Nowaczyk, Ronald H. Cognitive Skills Needed in Computer Programming. ERIC Report No. ED 236 466, 1983.

Palumbo, D. & Reed, M. Effect of BASIC Programming Language Instruction on High School Students' Problem Solving Ability and Computer Anxiety. Journal of Research on Computing in Education, Spring 1991, **23**(3), 343-372.

Palumbo Debra & Palumbo, David. A Comparison of the Effects of Lego TC Logo and Problem Solving Software on Elementary Students' Problem Solving Skills. Computing in Childhood Education, (1993) **4**(4), 307-323.

Pea, R. & Kurland, D. On the Cognitive Effects of Learning Computer Programming. New Ideas in Psychology, 1984, **2**(2), 137-168.

Pea, R. & Kurland, D. On the Cognitive Prerequisites of Learning Computer Programming. ERIC Report No. ED 249 931, 1983.

Pea, R., Kurland, D., Clement, C. & Mawby, R. Study of the Development of Programming Ability and Thinking Skills in High School Students. Journal of Educational Computing Research, 1986, **2**(4), 429-459.

Reed, W. M. A Philosophical Case for Teaching Programming Languages. Educational Computing and Problem Solving, 1988, p. 55, The Haworth Press.

Saiz, David (1994) PC Possibilities. The Science Teacher, March 1994, 28-31.

Seymour, Judy, Goings, Douglas & Vincent, Annette Factors Contributing to Success in a Microcomputer Course. Perceptual and Motor Skills, 1994, 79, 1338.

Salomon, G. & Perkins, D. (1987) Transfer of cognitive skills from programming:When and How? Journal of Educational Computing Research, **3**, 149-170.

Soloway, E., Lochhead, J. & Clement, J. Does Computer Programming Enhance Problem Solving Ability ? Some Positive Evidence on Algebra Word Problems. In R. Seidel, R. Anderson & B. Hunter (Eds.), Computer Literacy: Issues and Directions for 1985. New York: Academic Press, 1982.

Soloway, E. Should We Teach Students to Program? Communications of the ACM, October 1993, **36**(10), 21-24.

Appendix A – Pre-Test

This section contains the actual pre-test administered to the subjects of the observation that was done in Spring 1998.

Mathematics Assessment Quiz

Please answer each of the following questions on your Scantron form using a #2 pencil. There are 14 questions in all. This quiz will in no way count AGAINST you in the class, but each question you get right will add extra credit to be determined by your instructor. Be sure to identify yourself on your Scantron form. You will have 25 minutes.
(Note: the "Diff. Index" to the right is the percentage of students in the study who scored correctly on the question).

Question	**Diff. Index**
1. Let a, x and y be real variables. Recall that in mathematics, placing 2 variables next to each other means they are to be multiplied. Suppose a = 4, x = 14 and y = 5; evaluate the expression $a + xa - y(x + a)$ (a) -90 (b) -68 (c) -30 (d) 74 (e) None of the above	84
2. Again let a, x, and y be real variables with a = 17, x = 8, y = 9. Evaluate the expression $\frac{y(a - x)}{(a + x)}$ (a) 3 (b) –9 (c) 81 / 25 (d) 9 (e) 0	95
3. Suppose $\{a_n\}$ is a sequence of integers such that $a_1 = 1$, $a_2 = 1$, $a_3 = 2$, $a_4 = 3$, $a_5 = 5$, and $a_6 = 8$. From this pattern, what is a_7 likely to be ? (a) 10 (b) 13 (c) 11 (d) 17	74
4. Suppose f (x) is a real function of a real variable such that $f(x) = x^2$ for any x. If a and b are real variables, evaluate f(a + b) : (a) $a^2 + b^2$ (b) $a^2 + 2ab + b^2$	46

(c) $(a + b)^2$
(d) Both (b) and (c)
(e) Can't be done without knowing a and b

21

5. A **function** f mapping a set D into a set R (written f:D→R) is a rule of correspondence that associates to each element x of the domain D a unique element of the range R, which we denote by f(x). Such a function is said to be one-to-one provided
(a) anytime $f(x_1) = f(x_2)$, then $x_1 = x_2$
(b) anytime $x_1 = x_2$, then $f(x_1) = f(x_2)$
(c) f maps each x in D to a unique value in R
(d) f maps 1 to 1, that is, f(1) = 1.

54

6. Which of the following equations correctly represents the statement: "There are six times as many students as professors at this university". Use S to represent the number of students and P for the number of professors.
(a) P = 6S
(b) S = 6P
(c) S = 6 + P
(d) P / S = 6

14

7. Which of the following equations represents the statement: "At a recent party, for every 6 people who drank Coke, there were 11 who drank Pepsi". Use C for the number who drank Coke, P for the number who drank Pepsi.
(a) 11C = 6P
(b) 6C = 11P
(c) P / C = 11 / 6
(d) P – C = 5

43

8. In a certain computer program the relationship T = 2C is enforced, where C is the number of children in a situation, and T is the number of telephones. What is the correct English description of the relationship between C, the number of children, and T, the number of telephones ?
(a) There are twice as many children as telephones
(b) There are twice as many telephones as children
(c) The number of telephones equals one half the number of children
(d) None of the above

25

9. Functions f that are one-to-one (as in #5) have an ***inverse*** function f^{-1} (read f inverse), defined as follows: if f:D→R is one-to-one, then f^{-1}:R→D "undoes" the action of f in that, for any y in R, $f^{-1}(y)$ is defined to be the (unique, by one-to-oneness) value x in D such that f(x) = y. Example: 2^x and $\log_2(x)$ are inverse functions of each other. Use this fact to compute $\log_2(128)$ (no calculators!).
(a) 0.693147
(b) 0.303031

(c) 6
(d) 7
(e) none of the above

---

10. A function f is defined on real numbers by $f(x) = x^2 - 9$. What is f(f(4)) ? 55
(a) 7
(b) 21
(c) 40
(d) 16
(e) none of the above

---

11. If A and B are sets, the ***Cartesian product*** **AxB** is defined as the set of all possible ordered pairs (a,b) with a in A, b in B. For any such product, there exist ***projection mappings*** (functions) such as π_A:**AxB** →**A** so that for any (a,b) in **AxB**, the action of π_A on (a,b) gives just a. In a similar way, $\pi_B(a,b) = b$. Now with **Z** = the set of integers and **R** = the set of real numbers, then π_R:**ZxR** →**R** is the projection mapping onto **R**. What then is $\pi_R(2, 3.14)$? 54
(a) 2
(b) 3.14
(c) 6.28
(d) 1.57
(e) none of the above

---

12. A ***root*** of a function f of one real variable is a value r such that f(r) = 0. What is the root of the function defined by f(x) = 5x – 20 ? 71
(a) 4
(b) 5
(c) –4
(d) 0
(e) none of the above

---

13. Suppose f is a function of real numbers with the property that f(ab) = f(a) + f(b) for all real a and b, and suppose f(2) = 3 while f(3) = 5. Then f(12) = 21
(a) 8
(b) 10
(c) 11
(d) 12
(e) none of the above

---

14. The ratio of the circumference of any circle to its area is always identifiable as 39
(a) the diameter of the circle
(b) π
(c) the radius of the circle
(d) twice the reciprocal of the circle's radius
(e) None of the above

Appendix B – Post-Test

Here is the post-quiz that was administered toward the end of the Spring semester, 1998: (Note that although the post-quiz says it has 15 questions, in reality there are only 14 math questions, just like the pre-quiz. The last question was used to determine if the student was taking any math courses. In order to help ensure truthfulness, the student was asked to list all math courses currently being taken and also the instructors' name).

Mathematics Assessment Post-Quiz

Please answer each of the following questions on your Scantron form using a #2 pencil. There are 15 questions in all. The quiz will in no way count against you in the class, but each question you get right will add extra credit to be determined by your instructor. Be sure to write down your name on the Scantron form. You will have 25 minutes.

Question	Diff. Index
1. Suppose x is a real variable, with a value of 4. Evaluate $\frac{x^3 + 1}{x + 1}$ (a) 17 (b) 16 (c) 14 (d) 13 (e) none of the above	86
2. Let a, x, and y be real variables with a = 14, x = 20 and y = 5. Evaluate $\frac{a(x^2 - y^2)}{x + y}$ (a) 195 (b) 210 (c) 205 (d) 215 (e) none of the above	79
3. Suppose $\{a_n\}$ is a sequence of integers with $a_1 = 2$, $a_2 = 5$, $a_3 = 10$, $a_4 = 17$, $a_5 = 26$, and $a_6 = 37$. From this pattern, what is a_7 likely to be ? (a) 46 (b) 47 (c) 49 (d) 50 (e) 51	81

4. Suppose f(x) is a real function of a real variable such that $f(x) = x + 1/x$ for all non-zero values of x. Also suppose a is a real variable with a value of 5. Which of the following is true? (a) f(a) = f(1/a) (b) f(x) = f(1/x) for all real, non-zero x (c) Both a and b (d) Neither a nor b	26
5. A function f with domain D and range R is said to be ***one-to-one*** provided that anytime $f(x_1) = f(x_2)$ for some x_1, x_2 in D, then it follows that $x_1 = x_2$. (In other words, f is 1-1 provided f maps different points to different points). In the following, some functions are given whose domain is all of **R** (the set of real numbers). Which is one-to-one? (a) $f(x) = 1 / (1 + x^2)$ (b) f(x) = 5x + 2 (c) Both a and b (d) Neither a nor b	39
6. Which of the following is the correct way to express the statement : There are 100 civilians in this town for every police officer. (Here 'civilian' is used to mean 'a citizen who is not a police officer). Use C for the number of civilians, and P for the number of police officers. (a) P = 100C (b) C = 100P (c) P/C = 100 (d) Both a and c (e) None of the above	40
7. Which of the following equations represents the statement: "At a recent party, for every 3 people who preferred drinking whiskey there were 8 who preferred beer". Use W for the number who like whiskey and B for the number who like beer. (a) 8W = 3B (b) 3W = 8B (c) W/B = 3/8 (d) Both a and c (e) None of the above	26
8. In a computer program the relationship G = 3P is enforced, where G is the number of guns in a certain situation, and P is the number of personnel involved. What is the correct English description of the relationship that	33

exists between G and P? (a) There are three times as many guns as personnel. (b) There are three times as many personnel as guns. (c) The number of guns equals one-third the number of personnel. (d) Both b and c (e) None of the above	
9. Functions f that are one-to-one (as in #5) have an ***inverse*** function f^{-1} (read: f inverse), defined as follows: if $f:D \rightarrow R$ is one-to-one, then $f^{-1}:R \rightarrow D$ "undoes" the action of f in that, for any y in R, $f^{-1}(y)$ is defined to be the unique value x in D such that $f(x) = y$. (Such an x exists since y is in the range of f; and x is unique because f is 1-1). A function and its inverse act as follows: $f(f^{-1}(y)) = y$ for all y in R, and $f^{-1}(f(x)) = x$ for all x in D. Now, given $f(x) = 7x + 1$, which of the following represents $f^{-1}(x)$? (a) $g(x) = 1 / (7x + 1)$ (b) $g(x) = (x - 1) / 7$ (c) $g(x) = x/7 + 1/7$ (d) $g(x) = 7x - 1$ (e) None of the above	26
10. A function f is defined on real numbers by $f(x) = x^2 + 7$. What is f(f(5)) ? (a) 1024 (b) 1031 (c) 1017 (d) 632 (e) None of the above	55
11. For sets A and B, the cartesian product AxB is defined to be the set of all ordered pairs (a,b) with a in A, b in B. Suppose A = {1,2,3,4,5} and B is the set of 26 letters {a,b,c,....z}. Then, how many pairs are in AxB in this case? (a) 31 (b) 100 (c) 105 (d) 110 (e) None of the above	67
12. A root of a function f of one real variable is a value r such that f(r) = 0. What is the root of the function defined by f(x) = 9x + 72? (a) -8 (b) 8 (c) -9	60

(d) 9
(e) None of the above

13. A function f is ***logarithmic*** provided f(ab) = f(a) + f(b) for all positive real a,b. Now, the function $f(x) = 2^x$ is one-to-one over the whole real line, and so it has an inverse defined on the positive reals called $\log_2$, defined by $\log_2(y) = x$ whenever $y = 2^x$. The function $\log_2$ is logarithmic. Since $\log_2(8) = 3$, and $\log_2(32) = 5$, use the above to find out $\log_2(16384)$ (note, 16384 = 8 x 8 x 8 x 32). 46
(a) 76
(b) 16
(c) 15
(d) 14
(e) None of the above

14. A piece of tin which is 8 inches square can be made into a box with no top by cutting out 4 equal corners and folding up the 4 sides. If x is the length of such a cut (in inches), what is the formula for the volume of the box (in cubic inches)? 19
(a) $x(8 - x)^2$
(b) $x(8 - 2x)^2$
(c) $8x(8 - x)^2$
(d) $(8 - 2x)^3$

15. On the back of the Scantron form, please list any mathematics courses you are taking this semester, and the name of the course instructor as well. Just write NONE if you are not taking any math courses this semester.

Appendix C – Data

Obs#	Quiz Score	Predicted	Error	Final Exam	Took Math	PreQuiz
1	8	6.35	-1.65	46.9	0	10
2	8	6.30	-1.70	60.0	0	7
3	7	7.40	0.40	62.5	1	8
4	7	7.43	0.43	82.5	0	5
5	8	8.32	0.32	68.8	1	9
6	10	8.61	-1.39	90.6	1	5
7	9	6.79	-2.21	64.4	1	6
8	10	9.02	-0.98	86.3	1	7
9	4	4.92	0.92	61.9	0	3
10	9	8.56	-0.44	91.3	0	6
11	14	10.37	-3.63	93.1	1	9
12	7	5.85	-1.15	63.8	0	5
13	11	9.02	-1.98	86.3	1	7
14	9	9.28	0.28	75.6	1	10
15	8	7.84	-0.16	63.1	1	9
16	7	6.74	-0.26	63.8	1	6
17	11	9.76	-1.24	91.9	0	9
18	4	7.24	3.24	60.6	1	8
19	9	7.46	-1.54	76.9	1	5
20	9	7.98	-1.02	83.1	1	5
21	5	6.53	1.53	61.3	1	6
22	8	9.48	1.48	79.4	0	11
23	6	6.97	0.97	58.8	0	9
24	7	8.27	1.27	78.8	0	8
25	4	3.90	-0.10	40.6	0	5
26	7	5.82	-1.18	57.5	1	5
27	13	10.19	-2.81	81.9	1	11
28	9	5.83	-3.17	68.1	0	4
29	7	8.11	1.11	80.0	1	6
30	9	8.75	-0.25	83.1	1	7
31	6	9.26	3.26	80.0	1	9
32	12	11.23	-0.77	95.6	0	12
33	11	7.06	-3.94	78.1	0	5
34	4	2.94	-1.06	33.8	0	4
35	5	6.76	1.76	70.0	0	6
36	6	8.26	2.26	68.1	1	9
37	5	4.17	-0.83	43.8	0	5
38	4	3.59	-0.41	35.6	1	4
39	7	6.46	-0.54	75.6	0	4
40	9	9.36	0.36	91.7	0	8
41	7	7.58	0.58	75.1	0	7
42	9	9.70	0.70	91.2	0	9
43	8	8.89	0.89	80.1	1	8
44	8	6.77	-1.23	70.1	0	6
45	2	2.31	0.31	25.0	1	3
46	11	10.32	-0.68	89.3	0	11
47	8	8.08	0.08	75.1	1	7
48	7	6.87	-0.13	65.3	1	6
49	7	8.13	1.13	81.7	0	7
50	8	8.44	0.44	85.4	0	7
51	10	10.02	0.02	85.8	0	11
52	12	10.91	-1.09	96.3	0	11
53	6	8.80	2.80	94.2	0	6
54	6	8.21	2.21	72.1	1	8
55	12	10.69	-1.31	93.7	0	11
56	7	8.82	1.82	99.0	0	5
57	7	7.56	0.56	73.5	1	6
58	10	9.40	-0.60	90.8	1	7
59	10	9.35	-0.65	85.6	1	8

60	8	9.24	1.24	94.8	0	7
61	4	6.87	2.87	65.3	1	6
62	10	9.72	-0.28	91.4	0	9
63	14	12.54	-1.46	96.0	1	14
64	3	4.10	1.10	47.6	0	4
65	11	10.50	-0.50	96.1	0	10
66	8	8.66	0.66	89.0	0	7
67	6	6.65	0.65	73.2	0	5
68	9	10.83	1.83	95.4	0	11
69	10	10.59	0.59	95.8	1	9
70	9	10.44	1.44	94.0	1	9
71	12	10.67	-1.33	92.1	1	10
72	10	10.37	0.37	90.0	0	11
73	9	9.98	0.98	99.0	0	8
74	10	10.15	0.15	96.5	0	9
75	7	8.09	1.09	79.8	1	6
76	11	10.54	-0.46	90.6	1	10

Appendix D
Elements of the C programming language

Introduction

The C computer programming language was developed at Bell Labs in the early 1970's by Kernighan and Ritchie, who had been working on an operating system which became known as UNIX and were wanting a higher level programming language to write it in. Until that time (and for some time after), all computer operating systems were written in assembly code, which is code done at the most basic level possible – direct manipulation of registers and memory locations. A higher level computer language allows the programmer to create abstractions and implement these in a way that solves a problem without worrying about low level details of machine operation.

Variables and Declaration

The most basic abstraction of most computer languages, including C, is the notion of *variable*. A variable can be thought of as a value stored in a named memory location. The name is given by the programmer, who also decides on what *type* it should have. A declaration statement is used and the variable is said to be *declared* by the statement. For example

```
int account_number;
```

declares a variable named 'account_number' to be of type 'int'. The computer (actually the C compiler, which is the program that translates the programmers code into machine language so it can run on the computer) decides where in memory the value for the variable is to be stored (so far no value has been established.) The data type, int, refers to the most basic type on a computer: within a certain word size (often 32 bits, or binary digits) a signed integer value is stored. Other types in the C language are *char* (short for

character), *float* and *double*. Char is really just a short int type as far as C is concerned; integers of size 8 bits are stored in a char; since ASCII codes for characters all fit into 8 bits, this is where the type 'char' gets its name, as characters can be directly placed into a char type variable. For example:

```
char c = 'A';
```

would declare a 'char' type variable called 'c' and initialize its value to be 'A'.

The types float and double both refer to floating point values. These are what can be used for representing values that approximate the mathematician's real numbers. The difference between these types has to do with the amount of precision used to represent the values; typically 'float' represents a 32 bit value while 'double' represents a 64 bit value. The C language allows for great portability across different computer systems, so the above 'typical' sizes are just examples of what may be found; for any given system, all that is enforced by C is that the size of type double be at least as large as the size of type float. So one might see declarations such as

```
float theta = 0.785;
```

or

```
double x = 0.6931472;
```

Of course in math the integers are a subset of the reals, but that isn't quite how it works in a computer; you can store an 'integer' into a double or a float, but the format in which it is stored varies dramatically from the corresponding 'int' value! In other words 5.0 is stored as a float much differently than 5 as an int. For most purposes this fact doesn't matter much but it is important for the mathematician who wants to know more about programming.

Functions

C programs consist of sequences of statements, each terminated by a semi-colon, in a largely free format. Statements can be used to control looping and selection blocks, perform calculations, declare variables, etc. Every C program consists of at least one *function*. A function in C is a subroutine, consisting of a type, then a name, then any parameters, then the body of the function enclosed by begin – end braces { }. For example:

```
double f(double x)
{
    return x * x;
}
```

would be a function that would correspond to the math function $f(x) = x^2$, and could be used in like manner in a C program. The name of the function is 'f', the return type is double, it takes one parameter of type double called 'x', and what it returns is x^2 (or x * x in C, where the asterisk means to multiply.)

There is one function whose execution comes before any other in a C program, and that function has the name *main*. This function always returns type int (zero to indicate success, other values to indicate relative degrees of failure) and execution always begins by executing main. Here is a simple example of a C program that prints the square root of its input:

```
#include <stdio.h>
#include <math.h>
int main()
{
        double x;
        puts("Please type in a positive number : ");
        scanf("%lf", &x);
        printf("The square root = %f\n", sqrt(x));
        return 0;
}
```

That's it! If you have a C or C++ compiler, such as Borland C++ ™, you could just say

bcc32 sample.c

(or whatever the name of your compiler is in place of bcc32) then that would compile the above code (had it been saved as sample.c) and would then create an executable called sample.exe, which you could run by just saying 'sample' from the command line.

Details in the above program: puts() is a standard C library call that prints a string of characters passed to it; scanf is used to read in characters from the keyboard and format those according to various, user-specified criteria (in this case, read in a floating point value) while printf is used to print a formatted result to the screen. The sqrt function is used to calculate the square root of a floating point value. C has a pre-processor that is used to handle statements like #include: #include <stdio.h> for example is a directive to the pre-processor to include the contents of this file as a part of the current program. Mostly such header files contain information as to the structure and calling sequences for standard C library functions such as those in math.h (like sin(), cos(), tan(), sqrt(), etc.) which the compiler needs to properly handle calls to these.

Use of printf and scanf

For formatted input and output, the scanf and printf functions use a *format string* to describe how the remaining list of arguments are to be handled. Suppose one needed to input an integer followed by a floating point (double) variable; then the format string would likely appear as

"%d %lf"

Each % field describes the format for the next value; %d describes the format for a signed integer while %lf is used for double precision floating point. Now, scanf() needs the

addresses of the variables to be input, and so to call scanf so as to actually do the above input, it might appear as

```
scanf("%d %lf", &n, &x);
```

where n and x are the int and double variables in question. Printf is similar, however it only uses %f for both float and double, and often uses width specifiers. For example, suppose we wanted 3 columns with the first column being 2 digit integer and the last 2 being floating point, with a width of 10 and 2 places after the decimal. Then the printf statement inside a loop might appear as

```
printf("%3d%10.2f%10.2f\n", n, x, y);
```

Here you can readily see the described format; note the \n at the end, this means to print a newline (carriage return/line feed) after the rest of the line is printed. Here is a short, incomplete but usable table of formats:

```
int      %d
char     %c
string   %s (array of characters)
float    %f
double %lf (needed only for scanf)
unsigned %u (for unsigned integers)
```

Expressions and Assignments

In order to write code in C one needs to understand *expressions*. Basically there are int-expressions and floating-point-expressions; both involve the use of variables, constants, operators like +, -, *, / and also parentheses, along with function calls. Examples:

```
n + 2
y * sqrt(2 * x + 1)
x * x + 2 * x + 1
```

Expressions in C return a value to the point where the expression actually occurs in the program. An example of this occurs with the assignment statement, which has the form

variable = expression

where *variable* is any variable of a type compatible with the expression on the right. What happens here is that the expression is evaluated, and the result mapped to the variable on the left. However, keep in mind that in C an assignment statement is just another expression, and can occur in the context of a larger statement; a simple example is

```
x = y = sqrt(2.0);
```

which assigns the square root of 2 to both y and x. Here y gets the square root of 2, and x gets the result of the assignment to y, which is the same value.

Logical expressions and If

In C, logical decisions are made based on the value of certain int expressions. The operators <, >, ==, <=, >=, and != are used in comparing 2 integer or floating point expressions as to greater than, less than, etc. and the result of these comparisons is calculated as 1 for true and 0 for false. For example, if you had

```
int a = 2, b = 3, c;
c = a < b;
```

then c would be set to 1 since a is less than b (2 < 3). Here is the complete list of operators that are used in this way:

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal to (do NOT use one equal sign for this!)
!=	not equal (actually, ! can be used to negate any of these)

```
||      OR
&&      AND
!       NOT
```

Programs that need to make comparisons and do different things based on the result can use the if statement. It works like this:

```
if (condition)
        Statement;
```

Here the condition is evaluated, and if it is true (non-zero) then the Statement is executed; otherwise the statement is skipped. Here is an example:

```
if (n % 2 == 1)
        puts("value is odd");
```

Here the % operator can be used with integer values to return the remainder; in general x % y is the remainder one gets when dividing y into x. So if n is an integer then n % 2 is zero if n is even and 1 if n is odd.

If one needs to execute multiple statements as a result of an if test, just surround the group of statements with { } – these are C's begin and end markers. Here is how to print a message and stop the program if z = 0:

```
if (z == 0)
{
        puts("Cannot divide by zero, ending program.");
        exit(0);
}
```

The if statement in C has an else clause, used in order to select one of two possible courses of action. Extending the above example, suppose we wanted to input 2 integers and print their quotient, but only if the denominator is non-zero.

```
scanf("%d %d", &a, &b);
if (b == 0)
```

```
        puts("Cannot divide by zero.");
else
        printf("Integer quotient = %d\n", a / b);
```

Of course either the if part or the else part can be a compound statement if need be, just surround with { } to have more than one statement in either part.

Some special operators

C is a language that is big on expressiveness, that is, allowing the programmer to write complex code quickly. One of the ways it does this is by having special operators that do common operations in a compact notation. For example, it is very common in programming to add 1 to a value. Thus x = x + 1, or alpha_beta = alpha_beta + 1. A nice shorthand for this is with the ++ operator:

```
x++;
alpha_beta++;
```

There is also a - - operator that is used to subtract one from a variable. You can place these either before or after the variable in question. So long as the operation of adding or subtracting one is all that is done in the statement, then using ++x or x++ makes no difference. However in a more complex statement, pre-incrementation (or decrementation) takes place *before* the rest of the expression, while post-incrementation takes place *after* the rest. Example: in

```
c = 5 + x++;
```

the variable c get the value of 5 + x, and then x is increased by one. On the other hand,

```
c = 2 * --x;
```

will cause x to be decremented, and then c gets the value of 2 * x (with x already decremented). Warning: do not use these operators with reckless abandon. It is very

easy to write tricky code with these, and you can easily fool yourself as to what is meant. However, if you have a long variable name it is certainly nicer to say

```
super_long_name_yadda_yadda++;
```

than

```
super_long_name_yadda_yadda = super_long_name_yadda_yadda + 1;
```

and it also makes for fewer spelling mistakes.

In this same vein, C has abbreviations for adding or subtracting or multiplying or dividing a variable by an expression:

```
x += expression;
```

The above would calculate the value of expression and add the result to x. It is equivalent to writing

```
x = x + expression;
```

but lots shorter if x happens to be a really long name. Likewise one can do this with the other arithmetic operators including % (the mod operator):

```
variable -= 5 + x;
variable *= x + y;
```

and so on. Note, the whole expression on the right is evaluated before the left hand variable gets involved. So doing

```
x *= a + b;
```

is the same as

```
x = x * (a + b);
```

Iteration with while

Looping, or iteration in C can be accomplished using the while statement. Here is the general form:

```
while (condition)
        Statement;
```

It differs from if in that the loop is executed over and over again so long as the condition is true. If the condition is false initially, the whole loop is skipped; but otherwise the statement is executed and then the while structure is repeated, until the condition becomes false. Here is an example that adds up the integers from 1 to n:

```
/* assume sum and k are ints, and n has been input already */
      k = 1;
    sum = 0;
     while (k <= n)
    {
       sum += k; /* same as sum = sum + k */
       k++;
    }
```

Here k is initialized to 1, and sum to zero, and the while loop keeps adding k to sum and 1 to k until k becomes greater than n. If n were input as 10, then this loop would compute the sum 1 + 2 + 3 + ... + 10 = 55. Note the use of comments: C uses /* to start a comment and */ to end it.

Example Program

Here is a complete sample C program that inputs an integer n and prints a list of the prime numbers less than or equal to n, and also prints how many primes it found.

(See next page)

```
/*
 * Prime.c, by Bill Stockwell
 *
 */
int main()
{
        int n, divisor, k, prime, count = 0;
        printf("Enter n : ");
        scanf("%d", &n);
        if (n >= 2)
        {
                count = 1;
                printf("%d\n", 2);
        }
       k = 3;
        while (k <= n)
       {
                prime = 1;
                divisor = 3;
                while (prime && divisor * divisor <= k)
                        if (k % divisor == 0)
                                prime = 0;
                    else
                                divisor += 2;
              if (prime)
                 {
                        count++;
                        printf("%d\n", k);
                 }
                 k += 2;
        }
        printf("\nTotal primes = %d\n", count);
       return 0;
}
```

Arrays in C

As with most computer languages, C allows for the use of ***arrays***. These are indexed lists of items of the same type (that is, all the items in an array share the same type, say *int* or *double*). They are patterned after the mathematical notion of a *sequence*, as with $x_1, x_2, \ldots, x_n$, which in the case of a computer is necessarily finite. However in C the subscripts start at zero (0) rather than 1, and also since it is somewhat difficult on most systems to show subscripts easily, C uses square brackets for this purpose. Thus a reference such as x[k] refers to the kth element of array x. To declare an array variable, do it the same as for a simple variable of the same type, but include the *size* of the array in square brackets. Thus

```
int x[100];
```

declares x to be an int array with 100 elements, indexed from 0 to 99. There is no 100^{th} array element here; that is, a reference to x[100] is actually *out of range*. However C is a language that was developed by and for programming experts, and C does very little runtime checking for errors. In particular, C does not check for array references to be in bounds. That is up to the programmer to get right.

For the most part, declaring an array sets up in memory a contiguous list of like items, which (in the case of int x[100]) is like having declared 100 individual int variables, with space allocated and ready to be assigned to and used. However, having all 100 items together under the same name (x) gives the programmer great flexibility and cuts way back on the amount of code to be

written. Before seeing an example of this, we need to look at another type of loop statement in C.

FOR loops; useful with arrays

The *while* statement in C allows for iteration, but for many situations where the iteration to be done in order by equal steps, there is another form of the looping construct that is more useful. Here is the syntax:

for (init; condition; next)
Statement;

Here the *init* statement is done first (and only once for the whole loop); after that, the *condition* (a logical or Boolean condition) is checked for being true; if it is true, then the *Statement* is executed and the loop continues (checking the condition again and doing the Statement again so long as the condition remains true). *Next* is a statement that is carried out as the last step in each iteration, after the *Statement* is done. Let's see an example where we wish to add the integers from 1 to n:

```
int sum = 0;
int k, n;
/* get a value for n */
scanf("%d", &n);

for (k = 1; k <= n; k++)
   sum += k;

/* output result */
 printf("Sum = %d\n", sum);
```

Here the init statement is to set k equal to 1. Then the program would check if k <= n (true if the input was at least one), and then the statement sum += k would be executed. Then one is added to k, and the whole process repeats so long as k <=

n. For example, if the above code was made into a program and the user input 100 for n, then the program would print Sum = 5050.

Earlier it was mentioned that ***for*** loops are useful with arrays. Let us see how. Again let the array be int x[100], and suppose an integer n has been input that is to be the number of scores on a test, and we wish to input the array of scores.

```
for ( k = 0; k < n; k++ )
   scanf("%d", &x[k]);
```

Note the use of the address operator on x[k]; scanf ***always*** needs the address of the item to input. Consider now how the above could be done without an array. If there were 42 scores, we would have to input them with several statements such as

```
Scanf("%d %d %d %d %d %d %d %d %d %d",
   &x1, &x2, &x3, &x4, &x5, &x6, &x7, &x8, &x9, &x10);
```

and that only gets the first 10! Plus we would have to declare all those 42 variables, plus now our program is highly inflexible as to the possible number of scores – it would have to be rewritten just to do 43!

Almost all array processing is done with for loops. For example, suppose we wanted to average the array of scores mentioned earlier:

```
1)    double sum = 0, average;
2)    int k, n;
3)    int score[100];
4)    /* suppose we have code here to input n (# of scores)
5)       and the array */

6)    for (k = 0; k < n; k++ )
7)       sum += score[k];
```

8) average = sum / n;

Lines 6 and 7 above show the *standard C idiom for processing an array.* With n = the actual number of items, then to process items 0 through n-1 we use *for (k = 0; k < n; k++).* Inside the loop, references to x[k] then refer to the items we want, namely x[0] through x[n-1]. After adding all these up, line 8 then divides by n to get the average.

C also allows for higher dimensional arrays, pretty much whatever one needs. Arrays of one and two dimensions are by far the most common in actual code. A two dimensional array can be thought of as a table or spreadsheet, and is usefully thought of as having rows and columns. For example:

```
int table[10][20];
```

sets up a 2 dimensional int array having 10 rows and 20 columns. These are indexed from 0 to 9 and 0 to 19, respectively. FOR loops can be nested, and often are when dealing with higher dimensional arrays. For example, here is code to input the above table from the keyboard:

```
int table[10][20];
int row, col;

for (row = 0; row < 10; row++)
   for (col=0; col < 20; col++)
       scanf("%d", &table[row][col] );
```

If one needs access to all of the table values, a nested loop like the above is the way to go.

More array examples are coming, but most of them can wait until *functions* have been discussed.

Functions (Subroutines)

Essentially all computer languages allow for *subroutines*. These are blocks of code that can be *invoked* or *called* from any other point in a program. C is no different in this way, but C calls all such things by the name *function*. In many languages a function is a subroutine that returns a single value while a subroutine just does a computation in response to a command and does not return a value like a math function. In C, however, functions can play both roles. We have already seen a function without realizing it; in any complete C program, there is a *main()* function, which is what gets called initially when the program runs. Here is a skeletal outline of what a function looks like:

```
Return_type function_name(parameters...)
{
   the functions' code goes here...
}
```

The begin-end braces mark where the functions' code begins and ends. All functions have a *return_type*; this can be **void**, meaning there is no return value, or it can be any type such as int or double. All functions are written with () after the name, both in calling the function and writing it. If there are *parameters* to the function, these are placed in the parentheses. If not, the parentheses still must be present, but nothing is inside. Here is an example of a function that takes no parameters:

```
void status()
{
        printf("Error count = %d\n", error_count);
}
```

The above code assumes that 'error_count' is a global variable. Here no parameters are taken and nothing is returned. To call the function one merely mentions its name (with the parentheses) as in a statement:

```
status();
```

The effect of the code is that at the point where this instruction is encountered, the message

```
Error count = 5
```

would be displayed (assuming error_count was then 5). The keyword 'void' is used when one wants a non-specific or unknown type, or just no type at all. In the context of the return value for a function it is used to indicate that no value is returned. Let's now take a look at a function that *does* return a value, and also takes parameters. Long ago Euclid gave an algorithm for calculating greatest common divisors (GCDs). Given 2 positive integers, keep subtracting the smaller from the larger until they become equal; the result is the GCD of the original values. Here is a C function for that:

```
int gcd(int a, int b)
{
        while (a != b)
        if (a > b)
                a -= b;
          else
                b -= a;
        return a;
}
```

Here the return type is int, and the function takes two parameters, formally named 'a' and 'b'. Both of the parameters are also of type int. The while loop performs the intent of Euclid's algorithm and quits only when a and b become equal. At

that point, the common value 'a' is returned to the caller (we could have returned b as it has the same value). It is important to note that the parameters a and b are passed "by value" to the function. This means that when the caller invokes gcd(x,y), the values x and y (which could have been expressions and not merely variable references) are *copied* to the values for a and b in the code for gcd. When gcd returns to the caller, the values for a and b are **not** copied back. Thus, gcd can feel free to alter the values for a and b all it wants with no effect on the callers' parameters. As an example:

```
printf("GCD of 60 and 24 = %d\n", gcd(60,24) );
```

would print

```
GCD of 60 and 24 = 12
```

Here the values that are used by the caller are constant expressions (60 and 24) and could not possibly be updated from new values for a and b by gcd anyhow. The point is that C passes all parameters by value. Arrays are sometimes thought of as being a different case. Arrays are *effectively* passed by *reference* in C. Here is an example of a function that averages an int array:

```
double average(int x[], int size)
{
        int k;
        double sum = 0;
        for (k = 0; k < size; k++)
          sum += x[k];
      return sum / size;
}
```

C doesn't need to know the size of an array that is passed to a function, just to get the array. When passed to a function, an array 'decays' to a pointer to the first

array element. Hence all that is needed in the declaration of the array parameter is the name of the array followed by empty brackets (as with int x[]). This by the way is completely equivalent to int *x, which means that x is a 'pointer to int'. That is exactly how C treats an array parameter (at least for one dimensional arrays), as a pointer to the first array element. So, the size information is not made available. That is why one often sees the actual array size passed as a second (or additional) parameter. As in the code to average an int array, we saw that we had two parameters, int x[] and int size. The size needed to be passed so the function could know how many array elements to average. Here is another example that computes the 'geometric mean' of an array of doubles:

```
double geometric_mean(double x[], int n)
{
        /* computes (x0x1...xn-1)^1/n , the n^th root of the product of
         the array values. Assumption: all the xi are > 0
        */
       double prod = x[0];
       int k;
       for (k = 1; k < n; k++ )
            prod *= x[k];
       return pow(prod, 1.0/n);
}
```

In C, there is no operator for raising to a power, but there is a built-in function **pow** that can be used; pow(x,y) yields x^y for double precision values x and y. One should #include <math.h> in order to use **pow**. Here is another example that does the famous *linear search*:

```
int linear_search(int x[], int n, int item)
{
        /* Searches array x for the presence of the integer item, returning the first
index k such that x[k] == item, or -1 if item is not found. */
        int k;
```

```
    for (k = 0; k < n && x[k] != item; k++)
        ;
    /* The above semi-colon terminates the for loop and shows that this loop
has no body – all the code is in the for statement itself. But the semi-colon is very
important because without it, the next line of code would be taken as the for loop
body.
    Once the loop terminates, we are done – either the item is found or it is
just not present in the array at all. We only need to check whether k < n to see
which condition holds.
    */
    if (k < n) /* then we found item at array position k */
        return k;
    else
        return -1; /* else not found, return -1 */
}
```

Some 2D array examples

When passing an array to a function, the array name decays to a pointer to the first element. In the one dimensional case, no array size needs to be placed in the brackets after the array name. However, for any higher dimensions, the information is needed. Suppose we need a function to sum a row of a 2D table where the number of columns is 20. Here is an example of the code needed:

```
int rowsum(int x[][20], int row)
{
        int sum = 0, col;
        for (col = 0; col < 20; col++)
          sum += x[row][col];
        return sum;
}
```

Here the caller must provide a table with any number of rows but exactly 20 columns, and must specify which row is to be added up. More flexibility can be had by passing another parameter for the actual number of columns; for example, the caller may have an array with 20 columns but is only using the first 5 columns. Another change is that the actual number of rows to be used could be passed in and the row number to be added up could be tested for being in range, so as not to cause a runtime error.

The astute reader may have noticed that C array handling in the 2D case leaves something to be desired. This is true; the need to pass in all the size information for all dimensions but the first makes two and higher dimensional array handling a bit clunky. However, using pointer and dynamic memory allocation, we can get around this. An example follows in which a pointer to a pointer is set up to act as a 2D array pointer:

```
int **allocate_table(int rows, int cols)
{
        int **m;
        int k;
        m = malloc(rows * sizeof(int *));
        for (k = 0; k < rows; k++)
           m[k] = malloc(cols * sizeof(int));
        return m;
}
```

Here if one wants a 10 by 50 table, just do

```
int **p = allocate_table(10,50);
```

and you have it. Now you can pass p around just like a 2D array for all purposes.

Suppose one needed to find the *trace* of a square integer matrix of order n: then one might code

```
        int **x = allocate_table(n,n);

        printf("Trace = %d\n", trace(x,n));
```

with the code for *trace* being

```
        int trace( int **p, int n )
        {
                int sum = 0, k;
                for (k = 0; k < n; k++)
                        sum += p[k][k];
                return sum;
        }
```

Arrays however are only part of the answer to the need for ways to hold and access lot of data. Records are another way.

Structures (Records) in C

To understand the need for records is to understand that lots of information doesn't fit entirely or nicely into an int or double, or a character

string. Suppose one is writing a database to handle student records in a university setting. A simple example of such a record might need information such as

--the name of the student

--their ID

--number of completed credit hours

--grade points

--Grade point average thus far

Obviously other information would normally be needed, but you get the idea. In C programming, the **struct** is a vehicle for representing such information:

```
struct Student {
        char name[20];
        char id[14];
        int hours, points;
        float gpa;
};
```

The above forms a template (kind of a cookie cutter) for creating variables of this structure type. No variable has yet been declared. To do that we might say

```
struct Student X;
```

and this now makes X a student record variable. Now, how do we use it? Mostly through the *membership* operator which is '.' (period, or 'dot'). Thus we might say

```
strcpy(X.name,"John Jackson");

strcpy(X.id,"417891256");

X.hours = 65;

X.points = 197;

X.gpa = (float) X.points / X.hours;
```

Note the use of a "cast" in the calculation of X.points / X.hours; an int divided by an int in C will yield an integer result without it. The case causes the points field in X to be treated as floating point; then when divided by an integer, a float results.

In C, structures can be assigned one to another if they have the same exact structure type. Thus if we have

```
struct Student X,Y;
```

and some info had been placed into X, and we wanted Y to be an exact copy of X, we would only have to do

```
Y = X;
```

to make that happen. Also structures can be passed by value to functions (as well as by reference, by using a pointer) and also they can be returned from functions as a return value. Here is a sample function that prints the GPA of a student:

```
void print_gpa(struct Student X)
{
        printf("GPA of %s is %.2f\n", X.name, X.gpa);
}
```

If we had

```
struct Student someone;
```

Then to print that student's name and gpa we would only have to

```
print_gpa(someone);
```

Of course it is important to realize that the 'someone' variable would need to have some data placed in it first.

Sometimes, pointers to structures are needed. If p is a pointer to a struct, as in struct Student *p, then one can access the data in the record p points at like this:

Printf("GPA = %.2f\n", (*p).gpa);

Anytime one has pointer p in C, *p is ALWAYS to be thought of as "that which p points at". Since here p points to a struct Student, then *p IS that Student structure. Then we use the dot notation to access the fields therein. However, note the use of the parentheses around the *p, as in (*p).gpa. They are needed because the dot operator binds more tightly than * (has lower precedence) and so to leave out the parentheses is to refer to the wrong thing. But this type of reference is so common that there is another operator in C just for this purpose: That operator is -> (written as a dash followed by greater-than) and is simply read "pointer" or "pointer-to". With p a pointer to a struct Student, we could print the gpa field with

printf("GPA = %.2f\n", p->gpa);

At this point, we are ready to see how to pass a structure by reference in C. Consider the writing of a function that would calculate the gpa field in a student record. Suppose the name, points and hours have been initialized but the gpa is not yet calculated. Here is a C function to do that:

```
void calc_gpa(struct Student *p)
{
        if (p->hours > 0)
                p->gpa = (float) p->points / p->hours;
        else
                p->gpa = 0;
}
```

Here we first check before possibly dividing by zero, setting the gpa to zero if the student as yet has no hours completed, but otherwise calculating the gpa as the points divided by the hours. Either way the gpa field in the record that p points at is updated and it is the callers' record that is affected. Thus if you have

```
Struct Student X;  X.points = 15; X.hours = 5;
```

and you do

```
calc_gpa(&X);
```

then X.gpa would now be 3.00. Note the use of the '&'; this is the address operator in C, and when prefixed before a variable, yields the memory location of that variable. This is needed here since the function calc_gpa wants a pointer to a student record as its only parameter. The value of a pointer is precisely that, a memory address.

Linked Lists in C

Again, if p is a pointer, then p holds the address of something in memory (perhaps another pointer, or perhaps a data value of some kind). To get at what p points at we use the notation *p. Recall that *p is to be read "that which p points at". The inverse operator of * is &, the "address" operator. If p is a pointer to a student record, and X is a student record, we could set p = &X. Having done so, we would then have the identity *p == X.

A "linked list" is a chain of data values in memory such that each one has its own link to the next one. In this way, with only a pointer to the first one, we have an essentially self contained (perhaps huge) collection of values in memory

which are easily and quickly accessible. In C, what we need to set up something like this is a structure like:

```
struct node
{
   int data;  /* the data can be of any type – int is for  simplicity*/
    struct node *next;
};
```

So, the 'next' field in the record is a pointer to another such node. In order to be able to setup and use linked lists, we at least need a function for insertion of new values, and one for traversing the list we make so we can see if the values got there OK. Additionally I will show a function for averaging the values in such a list.

Linked List Sample Program

```
/*
 • Linked List Program by Bill Stockwell, 12/2001
 • Sets up a list of integers, with last in first out insertion
 */
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct node
{
   int data;
   struct node *next;
} Node;

/* Typedef allows creation of a named type, here Node, so we don't have to
repeat 'struct node' all the time */

Node *head = 0;
/* head will be global, the pointer to the start of our list */

void insert(int item)
{
    Node *newnode = malloc(sizeof(Node));
    /* sets up a pointer and allocates memory for it */
    newnode->data = item;
  newnode->next = head;
  head = newnode;
}

void traverse()
{
   Node *p = head;
    while (p)
    {
        printf("%d\n", p->data);
        p = p->next;
    }
}
```

```
double average()
{
    Node *p = head;
     double sum = 0;
    int count = 0;
    while (p)
    {
       ++count;
        sum += p->data;
          p = p->next;
     }
    return sum / count;
 }

main()
{
    double mean;
     insert(5);
     insert(7);
    insert(3);
    traverse();
    mean = average();
    printf("\nThe average of the values in the list is %.2f\n", mean);
    return 0;
 }
```

Appendix E
On the choice of the C programming language
As a vehicle for enhanced mathematics learning

First of all, the researcher is a faculty member in the computer science department of a mid-sized university in the south-central part of the United States and has been for over twenty years, and is expert in the use of quite a number of computer languages, including C, C++, FORTRAN, PL/I, Pascal, BASIC and several flavors of assembly language. He has taught most of these languages to college undergraduate students at some time or other over the last twenty years, including C for the last seventeen years and its descendant C++ for the last ten years.

In Chapter 1 it was mentioned that there were three areas of mathematical learning that it was thought might be enhanced from learning programming in C, and that these areas were 1) sequences, 2) functions, and 3) algebraic expression evaluation. The way sequences come into the picture via programming is through the idea of an array. An array in a computer program is a subscripted variable, capable of holding thousands of distinct values all under the same name, in which each value is addressable via a subscript. In C for example one might declare an array like

```
int scores[] = {71, 82, 95, 10, 17};
```

This would set up an array of integers (int = integer in C) called 'scores', with scores[0] being 71, scores[1] = 82, scores[2] = 95, and so on. Each scores[k] value (for some integer index k) is a full blown integer variable and can be altered or examined at any time. Most conventional computer languages have this array capability, so C has no advantage in this case.

In the second area, that of functions, a bit of explanation is needed. A function in mathematics represents how values in a domain set are transformed into uniquely defined elements of a range set. In some computer languages, one can define a type of subroutine called a *function* which is a body of code that can be called upon to perform its set of instructions, computing a value which is to be returned to the caller *in the manner of a mathematical function.* For example, here in C code is a function for computing the gcd of two integer values:

```
int gcd(int a, int b)
{
     while (a != b) /*as long as a and b are unequal*/
        if (a > b)       /* Keep subtracting */
            a = a - b;   /* the smaller from */
        else             /* the larger */
            b = b - a;
     return a; /* When equal, that is the GCD */
}
```

To call this function from another part of the program, something like

```
main()
{
   int value1 = 60, value2 = 24;
   printf("GCD = %d\n", gcd(value1, value2) );
   return 0;
}
```

Putting the above two pieces of code into one program and being sure to put

#include <stdio.h>

at the top, one could compile and run the code (with say Visual C++ for Windows), and it would come back and say

GCD = 12

which indeed is the greatest common divisor of 60 and 24. The point here is the mathematical function like behavior of the way in which the functions can be used in the

code that C programmers write. Can other languages do this? In many cases the answer is yes, however one must understand that in many computer languages, the notion of function is rather downplayed, to the point where many times programmers in those languages don't ever use functions. C is one of the few languages in which the *function* idea is the *only* subroutine that is available. Thus, if one writes any C code, one *will* be writing and using functions – not something one sees in other languages, except mostly C++.

The third and last area of expected transfer is that of algebraic expression evaluation. It should come as no surprise that just about any computer language has provisions for evaluation of algebraic expressions written by the programmer, so C would not appear to be preferable to other languages – at first glance. However, the nature of C is such that a higher level of care and overall knowledge is required to use it successfully. For one thing, C assumes that its' users "know what they are doing"; C compilers, the software that translates what the programmers write into machine code so it can actually run, assume that if there are questionable things in the C code, let it be. This is done for the sake of runtime efficiency (don't check on array subscripts being out of range, such checks are time-consuming) and also because C was originally developed for its users to write operating systems with – and people who write operating systems don't want the translator questioning their code.

Thus C is often thought of as better to learn as a second computer language than as a first one. Indeed, in the researcher's computer science department, those thinking of taking Programming I in C are generally asked what other computer languages they know, and are strongly recommended to take Beginning Programming (Pascal) if they

have no other computer language in their background. Hence, the researcher had the notion that students in Programming I – at least the successful ones – probably had a fairly high motivation and knowledge level, one that might serve well in attempting to learn some mathematical concepts.

The last reason the researcher had for choosing C is not the least of the reasons: C is the only computer language taught in large enough numbers at the researcher's university to allow for a decent study.