Teaching Flowcharting with FlowC

T. Grandon Gill
IS & DS Department, CIS1040
University of South Florida
Tampa, FL 33620-7800
ggill@coba.usf.edu

ABSTRACT

When detailed logic flowcharting fell out of favor as a commercial design tool starting in the mid-1970s, it was discarded by many IS educators. In doing so, however, we may have thrown the baby out with the bathwater. Many of the disadvantages of flowcharting as a commercial tool—such as the immense size of flowcharts of large programs—are not necessarily serious drawbacks in introductory programming classes. Several researchers have also found benefits from the use of flowcharts as a teaching tool. The challenge is to develop approaches whereby learning to program—not learning to flowchart—is emphasized. FlowC, a Windows-based flowcharting application, is an example of a tool that can be used to minimize the challenges of teaching flowcharting while retaining its benefits in the formative stages of learning to program. In addition to guiding the user through the creation of diagrams, FlowC also allows the user to view the code (or pseudocode) implied by each construct drawn in the flowchart. The user may also generate complete applications that may then be compiled and run in MS Visual Studio .NET. FlowC has been used for three semesters to teach introductory programming (in C) to undergraduate MIS majors. The students have found the program easy to use and have reported that flowcharting has been an important component of their overall learning in the course. In addition, analysis of survey data gathered from students suggests that learning flowcharting early in the course has benefited their learning in subsequent programming assignments.

Keywords: Flowchart, Programming, C Programming Language, Undergraduate Education

1. INTRODUCTION

The use of flowcharts to design and document program logic was nearly universal in information systems (IS) and IS education through the mid-1970s. Since that time, however, complaints voiced by practitioners (e.g., Brooks, 1975) and concerns regarding the educational value of flowcharts (e.g., Schneiderman, et al., 1977) have led to their virtual elimination in commercial settings and their near-elimination from academic curricula.

Just as the use of flowcharts was disappearing, studies started to appear suggesting that flowcharts could be of considerable value in an educational setting, both in terms of student preferences (e.g., Scalan, 1989) and in terms of educational outcomes (e.g., Crews and Butterfield, 2002). There is even some evidence that using flowcharts provides selective benefits when teaching programming to women (e.g., Crews, et al., 2002), whose declining representation in the programming field has been a source of considerable concern (Camp, 1997).

The present paper intends to accomplish two objectives. First, it reviews the existing literature on flowcharting in an effort to clarify the strengths and weaknesses of the technique from a commercial and pedagogical standpoint. Second, it introduces a tool, called FlowC, that has been used to teach C-language programming to MIS majors at a large state university for over a year. This leads to a discussion of how the design of FlowC attempts to address the traditional deficiencies of flowcharts, and a presentation of preliminary results relating to the tool's classroom effectiveness.

2. A BRIEF HISTORY OF FLOWCHARTING

The intellectual origins of flowcharting are generally attributed to John von Neumann, who advocated the use of flowcharts (or ideograms) in designing program logic (Chapin 1970). Through the 1960s and up to the early 1970s, the need for logic flowcharts in designing programs was largely taken as a matter of faith—with approximately a dozen texts devoted entirely to the subject of teaching flowcharting (Shneiderman, *et al.* 1977).

By the mid-1970s, a number of serious concerns had been voiced regarding the value of flowcharts. On the practitioner side, questions were raised regarding whether or not the use of flowcharts served any practical purpose. A well-written structured program, it was argued, was every bit as clear as a flowchart and was much more space-efficient—since a detailed flowchart takes 3 to 10 times as much space as the code it depicts. Moreover, experienced programmers usually created flowcharts only after they had written their programs. In part, this was to avoid the excruciatingly painful process of modifying flowcharts to accommodate code changes—an activity that was almost universally ignored (Hosch, 1977). With post-coding flowcharts being the rule, rather than the exception, it seemed reasonable to question the flowchart's status as an "indispensable design tool" (Brooks, 1975). Furthermore, although ANSI standards for flowcharting had long existed, they were generally ignored in practice (Chapin, 1970). To confound matters further, a radically different style of flowcharting—sometimes referred to as "structured flowcharts" (Haskell, et al. 1976; Nassi and Shneiderman, 1973)—was introduced around the same time period.

In parallel with the increasing industry concerns, some researchers began to question the pedagogical value of flowcharts. Several studies found that flowcharts were no more effective than pseudocode (a.k.a. program description language, or PDL) for a number of algorithmic tasks (e.g., Ramsey, et al. 1983; Shneiderman, 1982). One of the earliest and most influential of these studies (Shneiderman, et al. 1997) went so far as to systematically examine different stages of the programming process—composition, debugging, comprehension, and modification—and could find no benefit of using flowcharts compared with a PDL presentation. In addition, it had long been noted that students frequently failed to conform to flowcharting standards (Hosch, 1977). Couldn't the time it would take to teach rigorous flowcharting be better spent working on learning to program?

The combined onslaught of professional and academic critiques of flowcharting led to a decline in the practice during the 1980s. Nonetheless, it did not disappear completely. To begin with, despite its many apparent weaknesses, the flow diagram is an extremely convenient tool for introducing concepts such as branching. Thus, the use of informal flowcharts—particularly in introductory programming texts—continued for illustrative purposes (and continues to the present day). Perhaps more significantly, research began to emerge with findings that appeared inconsistent with the earlier research that questioned the value of flowcharts (e.g., Scalan, 1989; Zhao and Salvendy, 1996; Crews and Butterfield, 2002).

The new research on flowcharting focused primarily on educational applications, and called into question a number of aspects of the design of earlier studies. In particular, many of the earlier tests had not controlled for the time taken to complete assigned tasks—nearly all of which were completed by subjects at high levels of performance (e.g., 95% scores and above). In doing so, it was argued, they failed to measure the speed at which the task was performed. Another complaint was that the earlier studies failed to adequately control for task types. Both these design concerns proved to be significant. In one experiment (Zhao and Salvendy, 1996), task completion times, error rates, and perceived difficulty were reduced significantly by the use of flowcharts (in comparison with alphanumeric program code)—but only for tasks involving significant conditional branching.

Another concern relating to the earlier research was that it considered flowcharting only in the context of understanding a particular algorithm. The tool's potential value in learning to program was ignored. Even earlier attacks on the technique, however, had allowed for its potential value in a teaching role (e.g., "The detailed blow-by-blow flow chart, however, is an obsolete nuisance, suitable only for initiating beginners into algorithmic thinking", Brooks, 1975, p. 168). Subsequent research found that students showed an overall preference for flowchart vs. verbal presentation of algorithms (Scanlan, 1988). Research also found that the use of flowcharting in an introductory programming course, presented using a flowchart interpreter software tool, led to significant improvement in student performance over the course of a semester (Crews and Butterfield, 2002). There was even some indication that the benefits of the tool were more pronounced for female students than for male students (Crews, *et al.*, 2002). Such an exploratory finding would be of considerable interest, given current concerns regarding the declining percentage of female enrollments in computer-related programs (Camp, 1997).

In summary, then, flowcharting has been proposed useful for three main purposes: as a form of program documentation, as a means of enhancing algorithm understanding, and as a tool for teaching programming. In each of the three areas, certain strengths and weaknesses have been proposed in the literature. These are contrasted in Table 1.

The pattern of strengths and weaknesses apparent in Table 1 illustrates how the attractiveness of flowcharts varies considerably according to usage. As a tool for documentation, weaknesses clearly outnumber strengths. Furthermore, some weaknesses—such as their size and lack of value to skilled programmers—would seem to be inherent to the representation: unless we change what we mean by a flowchart, they will always be much larger than the code they describe. Their practical value in algorithm comprehension is also doubtful. Even if they do allow algorithms to be understood more quickly (e.g., Scanlan, 1989), how often—outside of a classroom—are we faced with a situation where so many people need to understand a particular algorithm that a flowchart's incremental benefits in speeding comprehension justifies the time required to draw the chart?

The value proposition for flowcharts in teaching programming is very different from the other two uses. Not only have researchers found many possible benefits of the technique, but its weaknesses seem less insurmountable when used for teaching purposes. Indeed, the main drawbacks seem to involve the mechanics of flowchart

Table 1: Flowcharting Strengths and Weaknesses

Document	• High level flowcharts may clarify program organization	• Detailed flowcharts too large and cumbersome to use • Too hard to modify • Little evidence that they benefit a skilled programmer in designing an application
Algorithm comprehension	• May speed understanding of complex algorithms • May enhance error detection in complex algorithms • May reduce perceived difficulty in understanding complex algorithms	• Do little or nothing to improve comprehension of simple programs • Benefits are reduced as code becomes more structured • Size becomes a barrier to comprehension for large programs • Are not well suited for representing some programming techniques, such as recursion • Much more difficult to prepare than PDLs
Teaching programming	• Acknowledged to be a natural way of introducing common constructs, such as branches and loops • May increase overall performance in introductory programming classes • May provide particular benefits to underrepresented groups (e.g., women) • Relatively language independent • May be perceived as helpful by students	• Somewhat difficult to prepare • Very difficult to modify • Easy to do incorrectly (i.e., not according to standards) • Learning time can be substantial—taking away from other class activities

creation/modification and the tradeoff of teaching flowcharting versus going directly into teaching programming in a specific language. In principal, at least, both of these weaknesses can be minimized through better tools and teaching strategies. Towards this end, the FlowC tool was developed.

3. THE FLOWC APPLICATION

FlowC is a flowcharting application that was developed as a learning aid for use in an introduction to programming course, taught to MIS majors at a large state university using the C programming language. The tool is a medium-sized standalone Windows application (roughly 20,000 lines of C++ code) developed by the course instructor, with the initial version representing approximately 4 programmer-months of effort. Approximately 300 students have used the tool in the course since it was introduced, in January 2002.

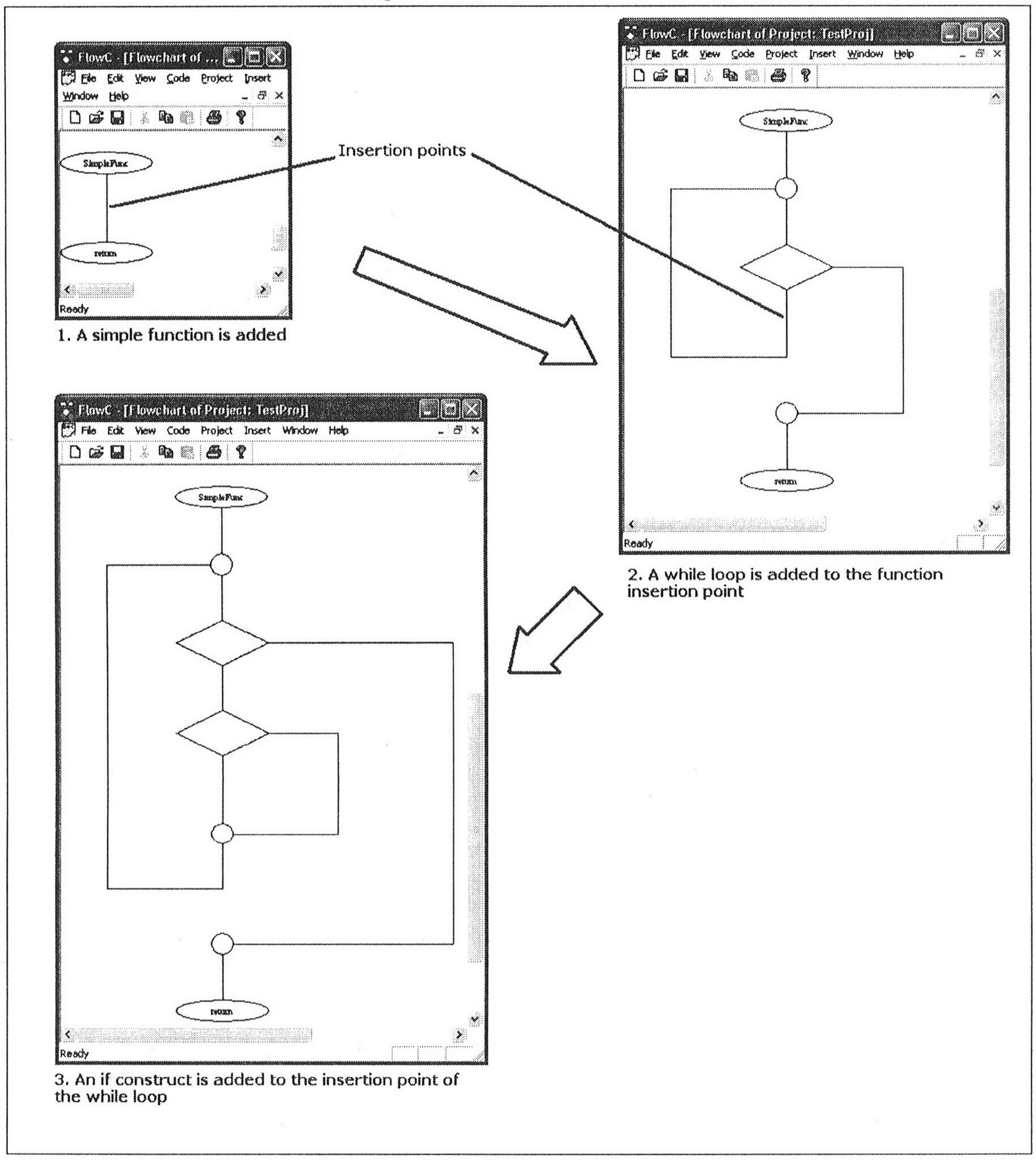
The FlowC interface is mouse-driven. Construction of a flowchart begins by adding one or more empty functions to the top-level. Constructs are then added by right-clicking vertical lines within a function, referred to as code blocks, opening a menu that identifies constructs that can be inserted (e.g., branches, a variety of loop types) within the code block. As constructs are added, further code blocks are exposed. In addition, the list of constructs that is presented in the right-click menu is context sensitive. If a code block is not in a loop, for example, the "continue" option is not presented. Text within construct graphics (e.g., rectangles, ellipses, and diamonds) can be edited by double-clicking the graphic. Presentation styling (e.g., color, font, shadowing) and object sizes can also be controlled though a variety of menu options. To assist the user getting started, and for the purposes of documentation, extensive online help is provided.

FlowC was designed to ameliorate the two main challenges to using flowcharting as a teaching tool: the difficulty of creating/modifying flowcharts and the tradeoff between time spent teaching rigorous flowcharting technique versus time spent teaching actual programming.

Challenge 1: Overcoming the difficulty of creating and modifying flowcharts

FlowC addresses the challenge of creating/modifying flowchart primarily through interface design. The tool represents a significant departure from the typical graphic design tool (such as MS Visio or MS PowerPoint), which simplifies drawing shapes and connectors but does little to help the student conceptualize a program (and leads to charts that are very difficult to modify). Rather than focusing on graphical design elements (e.g., rectangles, boxes, diamonds, connectors), it focuses on structured programming constructs, such as 2-way branches (e.g., if constructs), multi-way branches (e.g., case constructs) and loops (e.g., while, until and for). When a student creates a function in FlowC, a single connected entry and exit point is provided. Subsequent constructs then need to be added to the code block (*always* a vertical line) between the entry and exit points. When a construct has been added, additional code blocks may become available (e.g., an if-

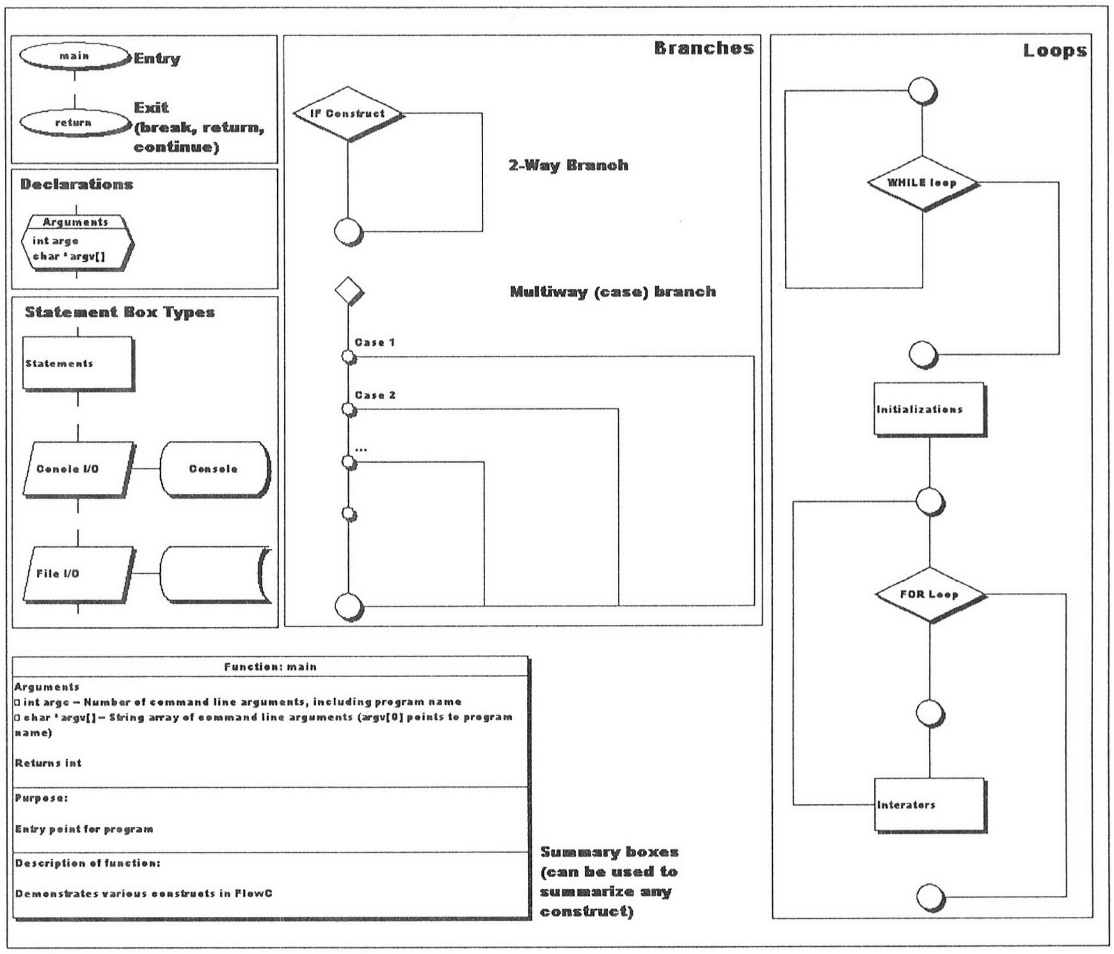
Figure 1: FlowC Insertion Process


FlowC - [Flowchart of ...
File Edit View Code Project Insert Window Help
SimpleFunc
return
Ready
Insertion points
1. A simple function is added
FlowC - [Flowchart of Project: TestProj]
2. A while loop is added to the function insertion point
3. An if construct is added to the insertion point of the while loop

Construct will have two: the *true* branch and the *false* branch). Further constructs may then be inserted into these blocks. This process is illustrated in Figure 1, where a flowchart starts with an empty function, into which a while-loop is placed, followed by an if-construct inserted within the while-loop.

Constructs are also be deleted (or cut-and-pasted) in the same manner. Individual graphic elements (e.g., the diamond in an if-construct) cannot be removed individually. Only an entire construct (and any constructs nested within it) can be removed. As a consequence of this interface, any flowchart prepared in FlowC is necessarily structurally valid. Structural validity is, of course, no guarantee of logical validity. It does, however, prevent users from creating charts that are likely to generate more confusion than learning—such as charts containing if statements with more than two branches, loops that don't

Figure 2: Examples of FlowC Constructs



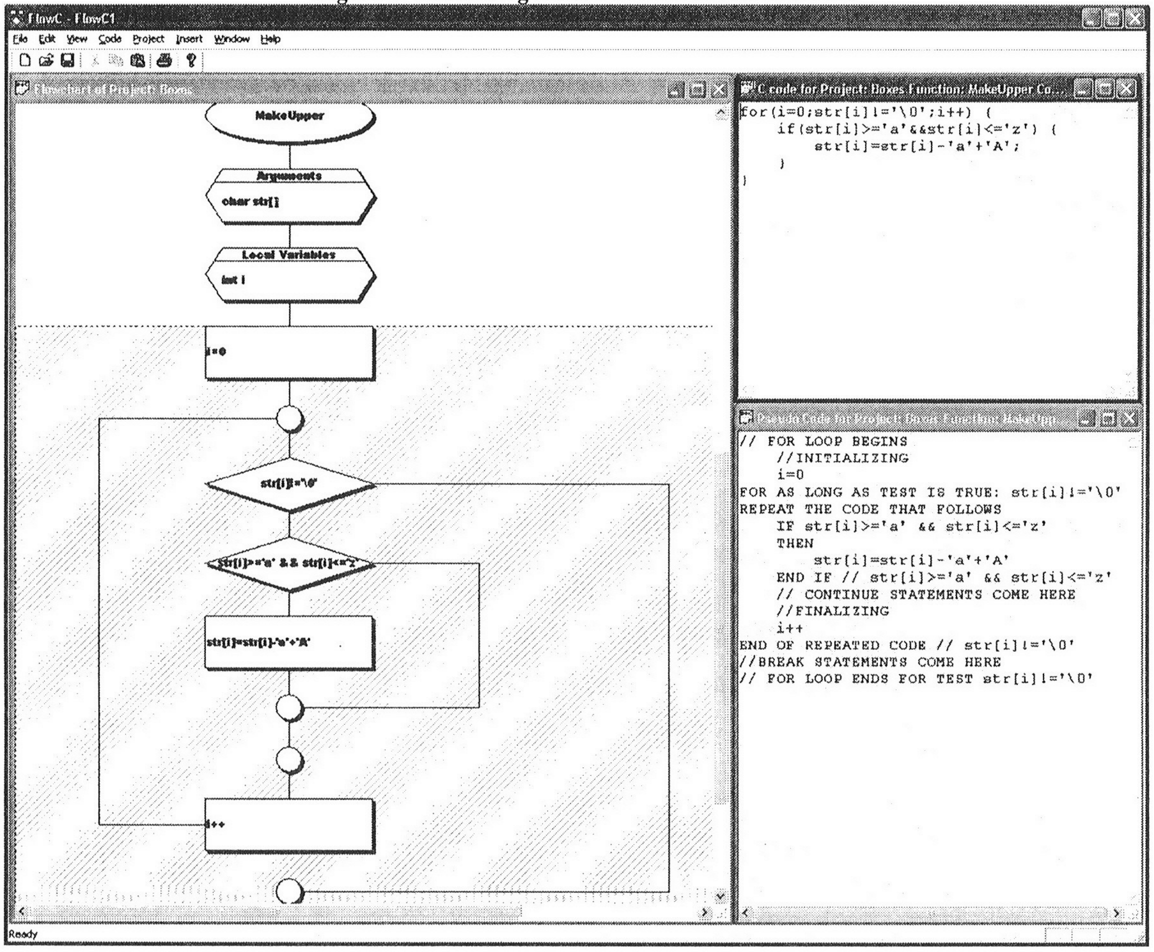
guarantee of logical validity. It does, however, prevent users from creating charts that are likely to generate more confusion than learning—such as charts containing if statements with more than two branches, loops that don't loop back on themselves and "Roach Motel" statement boxes (where flow checks in but it doesn't check out).

Another important aspect of FlowC's interface design is that it provides users absolutely no discretion as to the topological structure of the flowchart. Each construct is presented in a specific way and, while users can control global display properties (e.g., box sizes, coloring, fonts, etc.), each construct is displayed in one—and only one—

manner. Examples of the presentation of some of the constructs are shown in Figure 2. These constructs are, for the most part, consistent with the ANSI standard (excepting that flow is not allowed to pass though I/O objects, such as documents and files and multiple statements can be placed in statement boxes, to conserve space). Building ANSI compliance into FlowC was not an important objective, however, since the standard has never been widely adhered to (Chapin 1970).

As first glance, the total lack of flexibility associated with FlowC representation and positioning might seem unduly restrictive. This design decision, however, provides a number of important benefits. First, it eliminates line crossings and other common practical problems encountered when drawing flowcharts. Second, it allows a series of simple rules for interpreting the chart to be specified. For example, the line coming out of the bottom of a diamond is always the "true" branch, the line coming out of the left hand side is always the "false" branch. Similarly, any time a connector is to the left of the object it connects, it is traveling up (instead of the usual down) and the only entry point for any non-node (circle) object is the top. Such rules dramatically reduce the need for labeling in the chart and are quickly internalized by the user. Finally, the complete consistency required leads users to focus on the constructs themselves, rather than on finding the most efficient layout. Naturally, one price the user pays for lack

Figure 3: Generating Code and Pseudocode in FlowC


FlowC - FlowC1
File Edit View Code Project Insert Window Help
Flowchart of Project: Boxes
MakeUpper
Arguments
char str[]
Local Variables
int i
i=0
str[i]!='\0'
str[i]>='a' && str[i]<='z'
str[i]=str[i]-'a'+'A'
i++
C code for Project: Boxes Function: MakeUpper Co...
Pseudo Code for Project: Boxes Function: MakeUpp...
Ready

```
for(i=0;str[i]!='\0';i++) {
     if(str[i]>='a'&&str[i]<='z') {
          str[i]=str[i]-'a'+'A';
     }
}
```

```
// FOR LOOP BEGINS
     //INITIALIZING
     i=0
FOR AS LONG AS TEST IS TRUE: str[i]!='\0'
REPEAT THE CODE THAT FOLLOWS
     IF str[i]>='a' && str[i]<='z'
     THEN
          str[i]=str[i]-'a'+'A'
     END IF // str[i]>='a' && str[i]<='z'
     // CONTINUE STATEMENTS COME HERE
     //FINALIZING
     i++
END OF REPEATED CODE // str[i]!='\0'
//BREAK STATEMENTS COME HERE
// FOR LOOP ENDS FOR TEST str[i]!='\0'
```

of layout control is that FlowC charts can grow even larger than traditional flowcharts, with a ratio of roughly 10:1 compared with the code represented being common.

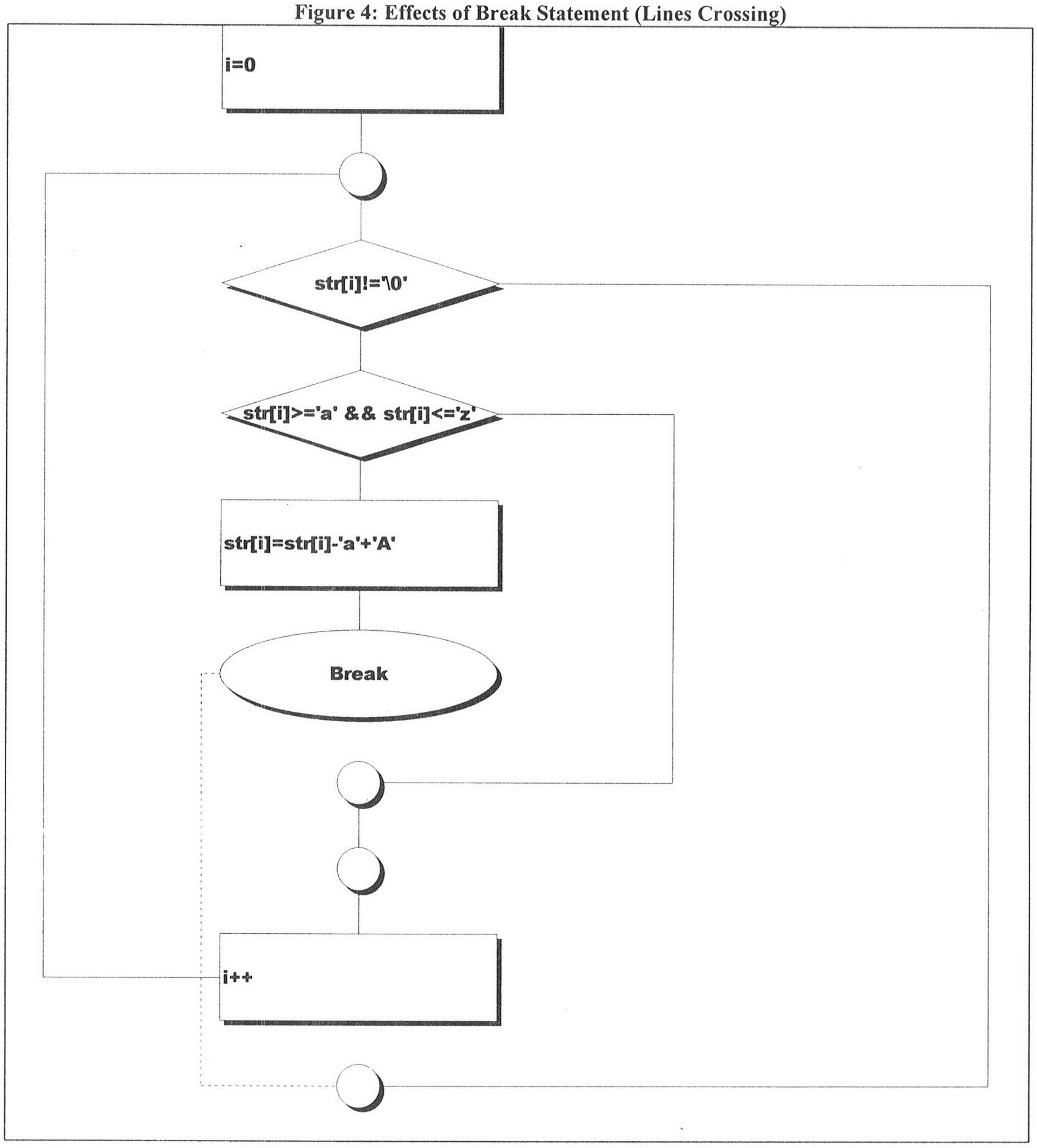
Challenge 2: Tradeoff between teaching flowcharting and teaching programming

Given that many researchers have found flowcharting no more effective than use of other techniques for describing code such as PDLs (e.g., Ramsey, et al., 1983) and that flowcharting has become uncommon in commercial documentation, it is reasonable to question the payback of teaching flowcharting in an introductory course. Wouldn't the time be better spent providing more instruction in the language being taught? FlowC addresses this challenge in two ways. The most direct impact of the tool comes from learning time compression. The structure provided by the FlowC interface leads to major reductions in the time required for students to learn rigorous flowcharting techniques (contrasted with traditional lectures on flowcharting technique). Indeed, over the past year students have been taught the how to use the tool primarily through about 30 minutes of multimedia tutorial files prepared by the instructor, combined with some handouts. There has been no feedback to the effect that this level of introduction was insufficient. Indeed, as will be discussed in the "Results" section, the tool was perceived to be relatively easy to use—particularly for students having no programming background.

The second mechanism for addressing the tradeoff between teaching flowcharting and programming is changing the nature of the tradeoff itself. Rather than teaching flowcharting and C programming in serial fashion, FlowC is used to teach flowcharting concurrently with teaching the C language. This approach is greatly facilitated by another capability built into FlowC: its ability to generate C code (and pseudocode).

The code-generation capability of FlowC is easy to access. Any time a construct is selected, a C code window or

Figure 4: Effects of Break Statement (Lines Crossing)

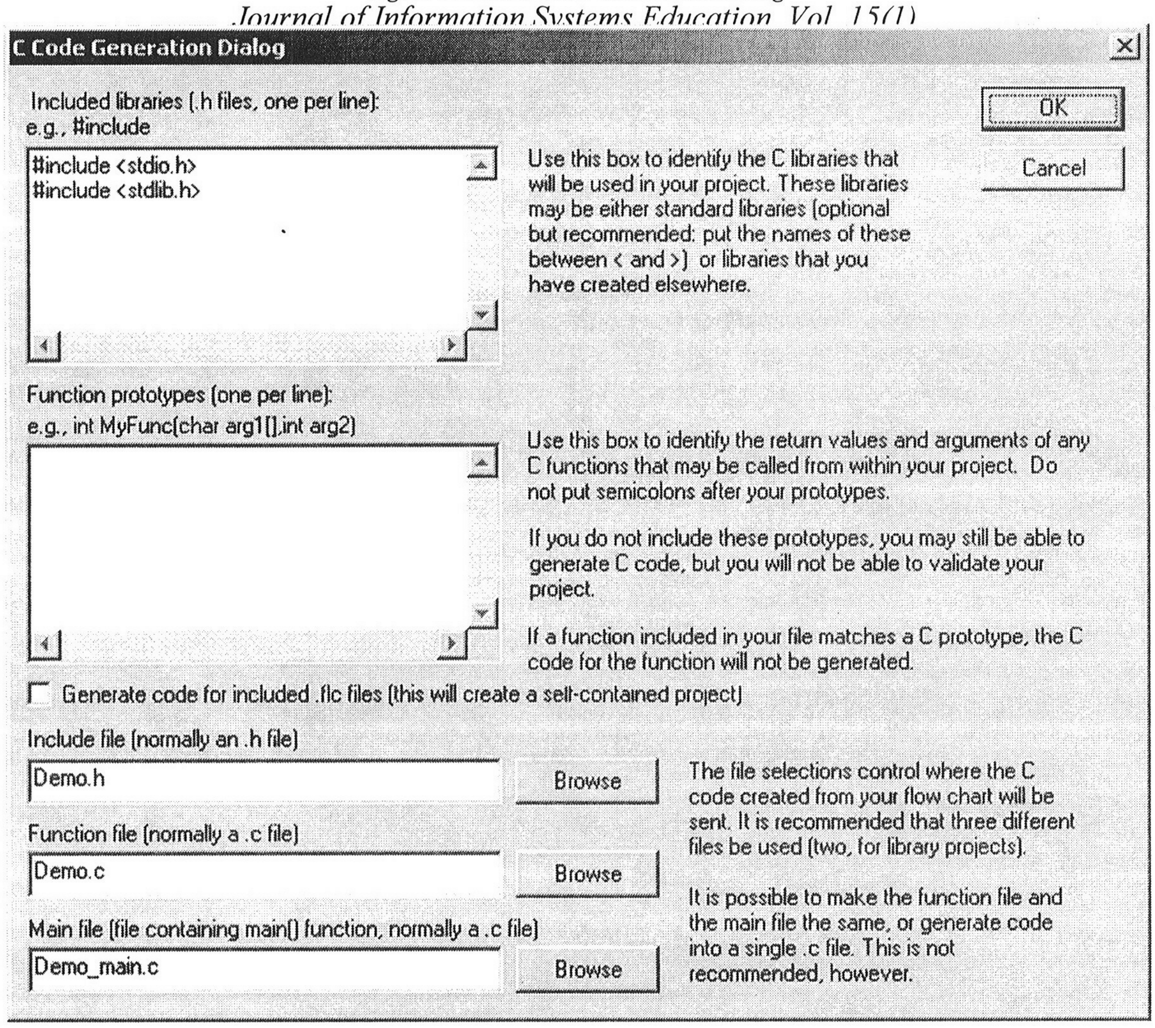


pseudocode window can be opened. These windows take the text entered into the diagram by the

user and places it in appropriate program/pseudocode form. This is illustrated in Figure 3, which shows both C and pseudocode renderings of a selected (shaded) for loop.

FlowC's code generation capability allows the student (and the instructor, during lectures), to view the direct correspondence between a flowchart representation and the code implied by it. This reduces the problem of translating between flowcharts and code that has been identified by some instructors (e.g., Brewer, 1976). The ability to display the code implied by any flowchart also helps students grasp the brace-delimited block structure used in C (and many other languages, such as C++, Java, C# and JavaScript). The block structure of C is further reinforced by the nature of the FlowC interface itself, which requires the user to think in terms of construct blocks—since these are the only means of adding and removing flowchart elements.

Figure 5: FlowC Code Generation Dialog



Included libraries (.h files, one per line):
e.g., #include

```
#include <stdio.h>
#include <stdlib.h>
```

Use this box to identify the C libraries that will be used in your project. These libraries may be either standard libraries (optional but recommended: put the names of these between < and >) or libraries that you have created elsewhere.

Function prototypes (one per line):
e.g., int MyFunc(char arg1[],int arg2)

Use this box to identify the return values and arguments of any C functions that may be called from within your project. Do not put semicolons after your prototypes.

If you do not include these prototypes, you may still be able to generate C code, but you will not be able to validate your project.

If a function included in your file matches a C prototype, the C code for the function will not be generated.

☐ Generate code for included .flc files (this will create a self-contained project)

Include file (normally an .h file): Demo.h

Function file (normally a .c file): Demo.c

Main file (file containing main() function, normally a .c file): Demo_main.c

The file selections control where the C code created from your flow chart will be sent. It is recommended that three different files be used (two, for library projects).

It is possible to make the function file and the main file the same, or generate code into a single .c file. This is not recommended, however.

FlowC was also designed to illustrate some of the subtleties of the C programming language. For example, as illustrated in Figure 4, the presence of an unused node in a for loop can be used to discuss the effect of a C *continue* statement. In addition, the potential confusion that can result from using statements that violate structured programming principles (e.g., break and continue) can be graphically illustrated—as these statements can be shown to be the only elements in a FlowC program that can lead to lines crossing (see Figure 4).

The code-generation capability of FlowC is also used to enforce certain programming standards. For example, FlowC intentionally omitted *goto* statements and case statement fall through from its available options. It also establishes standards, at least by example, for code presentation formats, such as indenting and commenting.

A final capability provided by FlowC is the ability to save complete C projects in a form that can readily be compiled by MS Visual Studio .NET, the development tool used in the course. The default behavior of FlowC is to create three files from each flowchart: 1) a header (.h) file containing all prototypes, definitions and global declarations, 2) a source (.c) main file, containing the definition of the project's main function, and 3) a source (.c) file containing all remaining function definitions. If a project is designated a "library" project, no main function is included—making it possible to create files for use in multiple projects. FlowC also allows the user to import functions from other FlowC projects. The code generation dialog is illustrated in Figure 5.

The current version of FlowC does not perform any syntax checking. As a consequence, if users place code in FlowC boxes that is not legal C, the resulting project created by FlowC will not compile. Even without syntax checking, however, an advantage of using FlowC to generate code is that it limits the types of errors that the user is likely to encounter. Common errors that lead to particularly incomprehensible compiler messages—such as missing semicolons, braces and improperly declared functions—are not normally encountered in FlowC generated code. Thus, the user's initial introduction to compiler errors is normally limited to improperly formed expressions and errors resulting from improperly declared variables.

Teaching flowcharting (with FlowC) and C syntax concurrently eliminates most of the incremental time associated with teaching flowcharting (and takes substantially less time than teaching flowcharting without FlowC, based on the instructor's experience). In fact, the benefits of having FlowC create frameworks for functions appears to reduce the total elapsed time from introducing C to the point at which students are able to create their own, non-trivial, code. As a result, the "tradeoff" between teaching flowcharting and teaching programming is illusory.

Table 2: Class Assignments For Introductory Programming Course

Assignment Number	Title	Oral Exam?	Coverage/Comments
1 (5%)	"Hello world"	No	• Creating single file project in MS Visual Studio .NET • Creating multifile project in .NET • Walking through various debugging features and performing screen captures
2 (5%)	Numbering systems	No	• Transforming integers between different bases • Hexadecimal and twos complement properties, overflow • Bitwise and logical operators
3 (27%)	Flowcharting	Yes	• Creating flowcharts for qualitative tasks • Describing an algorithm from its flowchart • Creating flowcharts for simple functions and the generating and compiling code • Creating a flowchart from C code • Designing a complex function using a flowchart, then generating and testing the resultant C code
4 (5%)	Debugging	No	• Fixing compiler, linker and logic errors in code, capturing key debugger screens along the way
5 (5%)	Memory grid	No	• Mapping variables, arrays and structures to a hypothetical location in memory • Evaluating various value and address expressions
6 (27%)	C Functions	Yes	• Specifications are presented for 11 C functions, ranging from fairly simple functions to more complex ones • Project must be tested using a test data file supplied by the instructor
7 (27%)	CGI Applications	Yes	• Development of a CGI program that takes input from a web-based form containing loan amount, interest rate and term values, then produces a mortgage amortization table page in response. • Students are given a standalone tool that simulates a web server

4. TEACHING WITH FLOWC

The FlowC tool is currently used as the basis of a single assignment i n a se mester l ong c ourse. S tudents c omplete the assignment over a 3-week period in the early stages of the course. The course is designed such that nearly all students (more than 90%) typically meet the course requirements through completing assignments—although students have the option of taking midterm and final examinations for up to 50% of their grade. For students not taking exams, 80% of the course grade is determined by performance on three major assignments. Each major assignment is validated by an individual oral examination, administered by the instructor or teaching assistants. In the even a student does not pass the oral exam on the first try, all subsequent exams on that assignment must be taken with the instructor. A list of the assignments, and their respective weights, is presented in Table 2.

Prior to commencing the flowcharting assignments, students have been introduced to the flowchart representation of the major programming constructs, both in lecture and through the use of multimedia (.avi) files. Their introduction to the specific rendering of these constructs in C occurs during the first week of the assignment, which is organized into the following parts:

4.1 The Flowcharting Assignment

The flowcharting assignment consists of a series of questions organized as follows:

1. Students are required to flowchart generic tasks, such as locating the position of the maximum integer in a list and diagnosing a printer problem.
2. Students are given a flow chart of a simple algorithm (binary search), along with an overview of the algorithm, and are then asked to answer questions about the algorithm. Students then recreate flowchart in FlowC and verify their results by compiling and running the FlowC generated code.
3. Students are asked to create a series of simple NUL-terminated string functions (e.g., strcpy, strcmp) using FlowC, then test them.

4. Students are given the C-code for a moderately complex function (atoi) and are asked to flowchart and explain it.
5. Students are given a written description of an iterative algorithm for computing a mortgage payment from interest rate, loan amount and term, and are then required to flowchart it, then generate C code for the algorithm that can be compiled and run.

Students completing Parts 1-5 reasonably well get an A on the assignment, students completing Parts 1-4 receive a B, and students completing Parts 1-3 receive a C. Students may work in groups, but—prior to receiving credit for the assignment—must pass an individual oral examination on the work submitted with the instructor or a teaching assistant. Inability to pass the oral exam is treated as equivalent to not turning in the assignment.

As the organization of the assignment implies, the principal role of FlowC is to get students comfortable moving back and forth between logical (i.e., flowchart) and code (i.e., C) presentation of problems. This focus presents an interesting contrast with FLINT, another flowcharting teaching tool described in the literature (Crews and Butterfield, 2002). The FLINT tool, which was developed independently of FlowC, contains a fully functional interpreter (unlike the current version of FlowC)—meaning that students can actually run their flowcharts once they create them. This capability makes the FLINT tool particularly valuable for teaching elementary programming logic, applicable to any programming language. Initial research into the use of FLINT suggests significant net benefits in overall course performance are achievable from this approach (Crews and Butterfield, 2002). Moreover, those benefits proved to be particularly pronounced for students coming into the class with weaker programming backgrounds (Crews, *et al.*, 2002).

One limitation of FLINT is that many data types, statement types and constructs are not supported. This limitation falls precisely in FlowC's greatest area of strength, allowing the user to represent nearly any construct that would normally be programmed in C (excepting those constructs that are not implemented as a mater of coding style, such as case fall through and goto statements). Given the differences in design between FlowC and FLINT, it is reasonable to believe that benefits observed from the use of FlowC may prove to be complementary to, rather than a replication of, those already observed for FLINT (i.e., in Crews and Butterfield, 2002).

4.2 Results of FlowC Use

The introduction of FlowC into the C programming course was—like most curriculum innovations—not performed using a rigorous experimental design. As a consequence, evidence of its effectiveness needs to be coaxed from a variety of sources, including both quasi-experimental data and evaluation data gathered from students.

When FlowC was first introduced into the course, during the spring semester of 2002, the primary change to the course materials was the replacement of a casual flowcharting assignment with the much more rigorous (and demanding) current version of the assignment. The other assignments remained virtually unchanged. It is therefore reasonable to view the transition from a quasi-experimental perspective, with the introduction of FlowC being treated as the experimental manipulation. In this context, two outcome measures would seem to be particularly relevant: 1) overall course evaluations and 2) percentage of students completing all seven assignments. The changes to the two measures are presented in Table 3.

Table 3: Changes accompanying introduction of FlowC assignment

	Evaluations (1=worst, 5=best)	Percent of students completing all assignments
Fall 2001 (pre FlowC)	2.63	23%
Spring 2002 (post FlowC)	4.47	44%

Although both differences resulting from the transition were positive and highly significant, the reader is cautioned against attributing too much weight to these results. Accompanying the change in assignments was also a change to content delivery options—the introduction of web-based discussion boards for each assignment. Since subsequent surveys indicated that students *also* viewed the discussion board favorably (4.5 on a 1 to 5 satisfaction scale), it is impossible to separate the effects of the two manipulations (not to mention any other changes that weren't measured). Such ambiguity is almost inevitable when using quasi-experimental methods.

Another technique that has been used to assess the value of flowcharting is to survey students (e.g., Scanlan, 1989). In this context, starting in Spring 2003, the course instructor developed a comprehensive survey, derived primarily from three previously validated instruments, to gather data on all aspects of the class. Students were offered extra credit for completing the instrument (which took 30 minutes to 1 hour to fill out) and 45 students (60% of the students enrolled in the class) completed it. A number of items from the survey are informative with respect to role played by flowcharting and FlowC. The most direct of these had students rate the statement "Assignment 3 was a helpful learning activity" on an agree (5) to disagree (1) scale. The result of this was a mean score of 4.16 (SE of the mean 0.15).

Another direct test involved ranking the assignments according to their perceived value. These results, shown in

Figure 6: Analysis of assignment rankings

	N	Mean	Std. Deviation	Std. Error Mean
Assignment 6	44	2.5000	1.69129	.25497
Assignment 3	45	3.3778	1.96895	.29351
Assignment 1	45	3.4444	1.82851	.27258
Assignment 4	45	3.6889	1.60712	.23958
Assignment 7	42	4.0714	2.02897	.31308
Assignment 2	44	4.3864	1.58798	.23940
Assignment 5	43	5.1395	1.56725	.23900

One-Sample Test

	Test Value = 3.3778 (Assignment 3 mean value)					
Assignment	t	df	Significance (2-tailed)	Mean Difference	95% Confidence Interval of the Difference	
					Lower	Upper
1	.244	44	.808	.0666	-.4827	.6160
2	4.213	43	.000	1.0086	.5258	1.4914
3	.000	44	1.000	.0000	-.5916	.5915
4	1.299	44	.201	.3111	-.1717	.7939
5	7.371	42	.000	1.7617	1.2794	2.2441
6	-3.443	43	.001	-.8778	-1.3920	-.3636
7	2.216	41	.032	.6936	.0614	1.3259

Figure 6, indicate that FlowC ranked second best of all the assignments—with the observed difference being significant for three of the lower ranked assignments: 2, 5 and 7.

The fact that FlowC was designed with code generation in mind (rather than acting as an interpreter, as FLINT does) may account for some differences between its observed impacts and those observed for FLINT. For example, students exhibited a nearly universal preference (observed by the instructor and teaching assistants) for answering questions during Assignment 3 oral examinations using FlowC-generated C code than using source flowcharts. This would suggest that flowcharting's key role was in getting the student started writing programs, rather than in helping the student to understand complex code (somewhat different from the enhanced benefits of flowcharts for complex vs. simple problems reported by Crews and Butterfield, 2002). There may also have been a difference in the population of students involved. Whereas, gender-related differences in the efficacy of flowcharting that were observed in earlier studies (Crews, et al., 2002), gender did not appear to be a significant predictor of any outcome for the students in the FlowC class (including performance on assignments prior to FlowC's introduction in the class).

A final result to be addressed is that of time taken to learn flowcharting (and, particularly, the degree to which it might take way from time programming). Here, the evidence would seem to suggest that the time demands of covering flowcharting are not excessive. The median reported time for Assignment 3 (15 hours, which presumably included the time required to become acquainted with the tool) was less than that of either of the two other major programming assignments (both 20 hours), despite the fact that all three assignments had equal weights. Further, there was slight agreement (3.33, on a 1 to 5 scale where 3 is neutral) with the statement "FlowC is easy to use". Surprisingly, the level of agreement was stronger among those who never taken prior programming courses (3.70 for first-timers vs. 2.95 for those who had taken prior courses, with a p-value bordering on significance at < 0.06). One possible interpretation of this finding is that the joint flowcharting-code generation nature of the assignment caused novice users to attribute difficulties to the coding aspects of the assignment, rather than blaming the tool. Also unexpectedly, reported FlowC ease of use did not correlate significantly with the many other software experience items contained in the survey, which included:

- An inventory of previously completed programming courses (both in any languages and in C/C++, specifically)
- An inventory of self-reported skills in various application packages and programming languages
- An inventory of self-reported skills in various categories of video game software.

5. DIRECTIONS FOR FUTURE RESEARCH

Since its inception, FlowC has continued to evolve. That evolution is taking place in two major directions at the present time. The first category of enhancements is allowing objects to be represented, using a subset of UML diagrams (particularly those describing class membership and inheritance). This capability is being implemented by extending the summarization box feature. The development of this particular capability is motivated by the desire to introduce elementary object-oriented programming capabilities at the end of the course in which FlowC is currently used. The prototype version of FlowC that incorporated this capability was introduced in fall of 2003, at which time the code generator was modified to produce C++ code, as opposed to pure C. A textbook incorporating the revised version of FlowC was published in the spring of 2004 (Gill, 2004)

The second category of enhancements is the incorporation of built-in syntax checking. This is being implemented through the use of a grammar file, modeled after the Prolog language, which is loaded into the application when it starts running and then allows the user to check the text entered into the flowchart for consistency with the grammar. The ultimate objective here is to route the user to an appropriate help page—explaining the appropriate contents of the box—in the event an error is encountered.

6. CONCLUSIONS

That IS practitioners will never again prepare large logic flowcharts is a virtual certainty. There is also little reason to expect that programmers will resume using flowcharts to explain complex code to each other (if they ever did). In discarding flowcharting, however, we must take care not to throw out the baby with the bathwater. Learning the basics of programming is a very different process from application development. There is no reason to believe that what is true for experienced developers will necessarily hold true for neophytes. The value of flowcharting appears to be a case in point.

When using flowcharting to teach programming, there are two major obstacles: the potentially tedious nature of flowcharting and the time it takes to teach rigorous flowcharting. The exploratory findings presented here support earlier exploratory findings (e.g., Crews and Butterfield, 2002) that the use of a tool designed to aid students in constructing flowcharts may reduce both tedium and time associated with teaching flowcharting. Given that tools such as FLINT and FlowC are now available, it makes sense to revisit how we teach programming. At a minimum, further research into the value of flowcharting in programming curricula is warranted.

7. REFERENCES

Brewer, R.K. (1976) "Documentation Standards for Beginning Students". The papers of the ACM SIGCSE-SIGCUE technical symposium on Computer Science and Education. February. pp. 69-73.

Brooks, T. (1975) The Mythical Man-Month: Essays of Software Engineering. Addison Wesley, Reading, MA.

Camp,T. (1997) "The Incredible Shrinking Pipeline" Communications of the ACM. 1997. Vol. 40(10). 103-110.

Chapin, N. (1970) "Flowcharting With the ANSI Standard: A Tutorial". Computing Surveys. June, Vol. 2(2), pp. 119-146.

Crews, T. and J. Butterfield. (2002) "Using Technology to Bring Abstract Concepts into Focus: A Programming Case Study". Journal of Computing in Higher Education. Spring, Vol 13(2), pp. 25-50.

Crews, T., J. Butterfield and R. Blankenship. (2002)"Right From the Start: Leveling (then Raising) the Playing Field". In D. Colton, M.J. Payne, N. Bhatnagar and C.R. Woratscheck (eds.) The Proceedings of ISECON 2002 v 19 (San Antonio): 343c.

Gill, T.G. (2004) Introduction to Programming Using Visual C++ .NET. John Wiley & Sons, New York.

Haskell, R.E., D.E. Boddy and G.A. Jackson. (1976) "Use of Structured Flowcharts in the Undergraduate Computer Science Curriculum". The Papers of the ACM SIGCSE Sixth Technical Symposium on Computer Science Education. September, pp. 67-74.

Hosch, F.A. (1977) "Whither Flowcharting?". Proceedings of the Eighth Technical Symposium in Computer Science Education. February.

Nassi, I. and B. Shneiderman. (1973) "Flowchart Techniques for Structured Programming". SIGPLAN Notices. August, Vol. 8, pp. 12-26.

Ramsey, H.R., M.E. Atwood, and J.R. Van Doren, (1983) "Flowcharts Versus Program Design Languages: An Experimental Comparison". Communications of the ACM. June, Vol. 26(6), pp. 445-449.

Scanlan, D. (1988) "Should Short, Relatively Complex Algorithms be Taught Using Both Graphical and Verbal Methods?: Six Replications". ACM SIGSE Bulletin, Proceedings of the nineteenth SIGSE technical symposium on Computer Science Education. February, Vol. 20(1), pp. 185-189.

Scanlan, D.A. (1989) "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison". IEEE Software. September, pp. 28-35.

Shneiderman, B. (1982) "Control Flow and Data Structures: Two Experiments". Communications of the ACM. January, Vol 25(1), pp. 55-63.

Shneiderman, B., R. Mayer, D. McKay and P. Heller. (1977) "Experimental Investigations of the Utility of Detailed Flowcharts in Programming". Communications of the ACM. June, Vol 20(6), pp. 373-381.

Zhao, B. and G. Salvendy. (1996) "Compatibility of Task Presentation and Task Structure in Human-Computer Interaction". Perceptual and Motor Skills. Vol. 83, pp. 163-175.

ENDNOTES

Instruments used as the basis of the class survey were:
1) "Student Opinion Survey" (located at http://oerl.sri.com/instruments/cd/studcourse/instr16.html on 5/4/2003),
2) "Computer Programming Survey" (located at http://oerl.sri.com/instruments/cd/studcourse/instr11.html on 5/4/2003) and
3) "Student Assessment of Learning Gains (SALG)" (generated from http://www.wcer.wisc.edu/salgains/instructor/ on 4/14/2003).

AUTHOR BIOGRAPHY

T. Grandon Gill is an Associate Professor at University of South Florida. His educational background includes three degrees from Harvard University: an undergraduate degree in Applied Mathematics (cum laude) from Harvard College, a Masters of Business Administration (high distinction) from Harvard Business School and a Doctor of Business Administration in the Management of Information Systems, also from Harvard Business School. His teaching areas have included programming, management of information systems, database design, the Internet and case method research. He has received numerous teaching awards, including the Florida Atlantic University award for excellence in undergraduate teaching. His research interests include expert systems, organizational learning and MIS education and include numerous publications in prestigious journals, such as *MIS Quarterly*. He has also done extensive programming, in a variety of languages, and has designed and programmed a number of commercial software applications.