Using executable flowchart for teaching C language

Sehyeong Cho
Professor,
MyongJi University
Yong In, Korea
E-mail: shcho@mju.ac.kr

Conference Key Areas: Please select three Conference Key Areas

Keywords: programming instruction, visual programming, flowchart

INTRODUCTION

Teaching computer programming to students is a daunting task, especially to those without any background or experience in computer programming. Even simple assignment statements or arithmetic operations can be difficult for them to understand. In our experience, roughly 25% of students fail the course and get frustrated that they are not fit for programming after all.

There are many reasons why programming is so difficult for beginners. First, there are linguistic issues. The syntax of a programming language is very different from that of a natural language. Trivial grammatical errors can result in cryptic error messages that are hard to interpret. The students also encounter semantic difficulties. It is a challenge to get an accurate understanding of the operational semantics (i.e., effects) of the programming language constructs, which makes it difficult to predict the accurate result of a program code. This, in turn, makes it difficult to write a program. Second, regardless of the difficulty of programming language at hand, the problem-solving process itself is inherently complicated.

This paper proposes to use an extended flowchart called CFL as the first programming language before beginning to teach general programming language such as C, and argues that it can help students learn C programming language better. We describe the language and system, then how it was applied, and show the results.

1 RELATED WORK

There are many different approaches to facilitating the acquisition of programming languages. For instance, in order to avoid the complexity of full-fledged programming languages, one can use simplified programming languages, such as Mini-Java[1]. In fact, mini-languages have been used for quite a long time[2]. However, this approach does not meet our requirements, since we need to teach a full language after all. Iconic programming languages, such as Mindstorm NXT-G[3] or Alice[4], can be another possibility. Iconic programming is suitable for a gentle introduction to computer programming and it has been reported to help keep students interested[4]. Flowchart programming (called FLINT) has been used elsewhere as an aid to help students understand the concept and to improve their problem-solving skills[5]. Raptor[6] is another well-designed flowchart programming tool used for teaching the concept and problem-solving skills, but there is a big gap between flowcharts and commercial programming languages, and therefore transition to actual programming language such as C is time-consuming and requires extra effort. Neither FLINT, nor Raptor were suitable for our purpose because we wanted the students to smoothly transition to C language.

2 CFL: C-LIKE FLOWCHART LANGUAGE

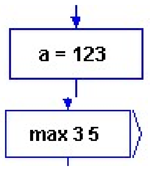
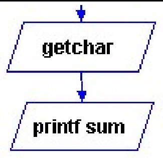
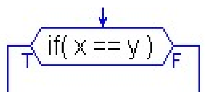
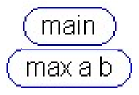
2.1 Overview of CFL

CFL stands for "C-like Flowchart Language," and is an executable flowchart-based programming language and system. CFL is developed to help students learn programming, as well as to understand what's happening during program execution. In particular, it is designed to help prospective C-language learners. CFL is basically a flowchart-based language. Therefore it is easy to understand the flow of execution. CFL is also an integrated development environment, and therefore has execution and debugging capabilities. There is no compilation process

2.2 Elements of CFL

A CFL program consists of nodes and directed arcs connecting the nodes. There are basic nodes and composite nodes. There are four types of basic nodes: processing, I/O, decision, and function nodes. However, decision nodes can only be used in composite nodes such as 'while' composite, to enforce structured programming. Table 1 summarizes CFL basic nodes. For easier transition from CFL to C language, the syntax of CFL is intentionally aligned with C language syntax. These include arithmetic expressions, array notation, and input/output functions such as "scanf" or "getchar."

Table 1. CFL basic node types

type	processing	I/O	Decision	start/end
typical example(s)				
note	+, -, *, /, % function call, return	putchar, scanf printf	!=, ==, > ,<, >=, <= , &&, ||	only one main function

Composite nodes are used to group particular combination of basic nodes to give structures, such as "if-then-else", "for", "while", and functions. Structured programming is naturally enforced by composite nodes. Arbitrary connections (i.e., 'goto') are not allowed.

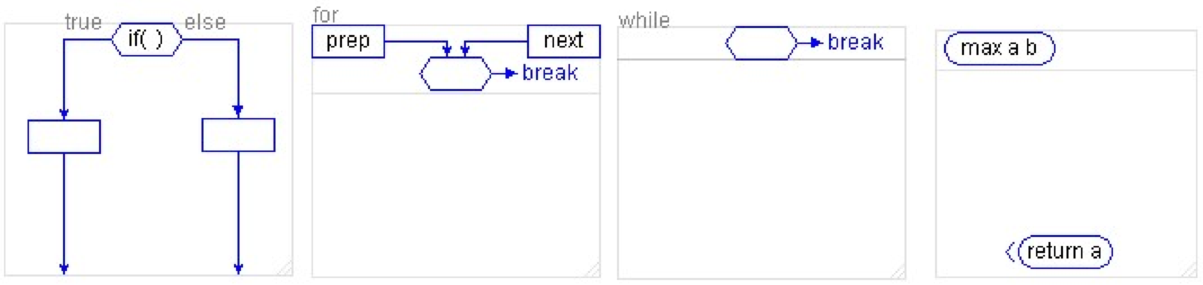
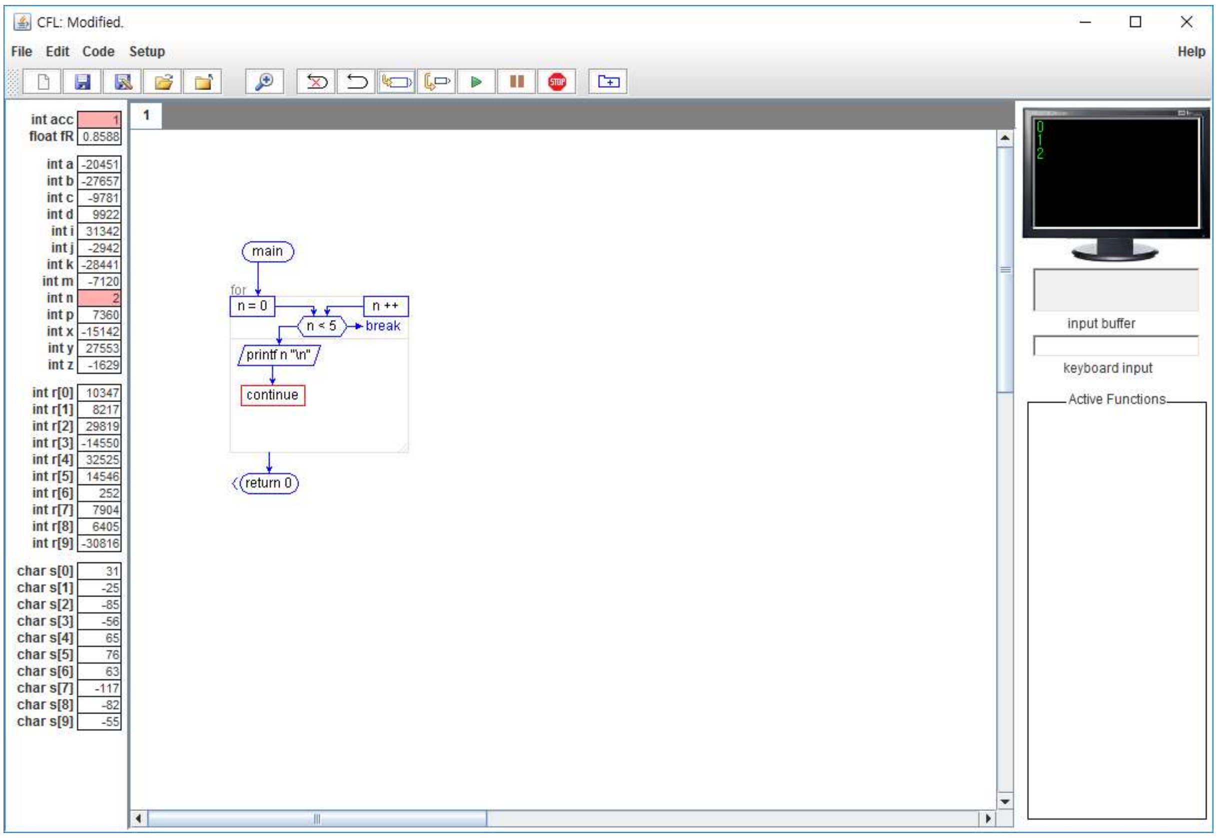


Fig. 1. Composite notes: "if", "while", "for", and function

2.3 Executing CFL

CFL is executable, and, therefore, has features related to execution. These features include: one accumulator register, one floating point accumulator, 12 integer variables that can be changed to float variables, two predefined arrays, the input buffer, the output window, and debugging buttons – step-in, step-over, run, and stop. During the execution, students can watch the inner workings of the program: current node is marked red, changed values are also marked, and the function call stack is depicted with actual parameters.



int acc	1
float fR	0.8588
int a	-20451
int b	-27657
int c	-9781
int d	9922
int i	31342
int j	-2942
int k	-28441
int m	-7120
int n	2
int p	7360
int x	-15142
int y	27553
int z	-1629
int r[0]	10347
int r[1]	8217
int r[2]	29819
int r[3]	-14550
int r[4]	32525
int r[5]	14546
int r[6]	252
int r[7]	7904
int r[8]	6405
int r[9]	-30816
char s[0]	31
char s[1]	-25
char s[2]	-85
char s[3]	-56
char s[4]	65
char s[5]	76
char s[6]	63
char s[7]	-117
char s[8]	-82
char s[9]	-55

```
0
1
2
```

Fig. 2. Screen shot of CFL

3 APPLYING CFL TO C LANGUAGE COURSES

3.1 Preparation

We had two classes of 4 credit-hour C-language programming courses with exactly the same class sizes - 35 students (maximum allowed size for a programming course with a lab). The students were taught by the same instructor on the same weekdays, but on different hours. One class - the control group - was taught by the regular C language syllabus. The other group - the treatment group, or the test group - was first taught CFL programming for four weeks, before they started learning C language. Table 2 summarizes the semester schedule of the two classes. Although the course is intended for freshmen, there were quite a few students with some prior experience in C programming, we excluded them from the analysis. This resulted in a total of 29 first-year students in the control group and 22 in the test group.

Table 2. Summary of schedules

Week	Control group (C only)	Treatment group (CFL + C)
1	Intro to Computers, Prep for laboratory (incl. Linux and vim)	Intro to Computers, CFL basics, operations, I/O
2	Beginning C programming	CFL conditional, for loop
3	Integers and I/O	CFL arrays, functions and recursion
4	conditionals	CFL graphics and game project
5	while/for loops	Linux and vim, Integers and I/O,
6	Functions	conditionals, while/for loop
7	Arrays and applications	Functions, arrays
8	Handling strings, mid-term	arrays, mid-term
9	2-dim arrays	strings, 2-dim arrays
19	files	files
11	structures	structures
12	bit handling	bit handling
13	recursion	recursion
14	pointers and dynamic allocation	pointers and dynamic allocation
15	linked lists	linked lists
16	Final Exam	Final Exam

3.2 The Experiment

To see if learning CFL has any effects, either positive or negative, on learning C language programming, we tested the students with three identical tasks in the final exam, and observed their overall performance throughout the semester. The students were not informed about the ongoing experiment.

Task A

Problem description:
Let X be a sequence of integers starting with 1, 2, 4, … and the differences between consecutive terms make an arithmetic progression of 1, 2, 3, ….

Let Y be a sequence of integers that starts with 1, 2, 4, 8, … and each pair of consecutive numbers in the sequence has a difference defined by sequence X.

Write a program to generate sequence Y less than 100.

Task A requires the understanding of how to write a loop. In the control group, 12 of 29 freshmen passed the test, compared to 15 of 22 in the treatment group. Thus, the performance in the treatment group was significantly higher than that of the control group (68.2% vs. 41.4%) as shown in Figure 3.

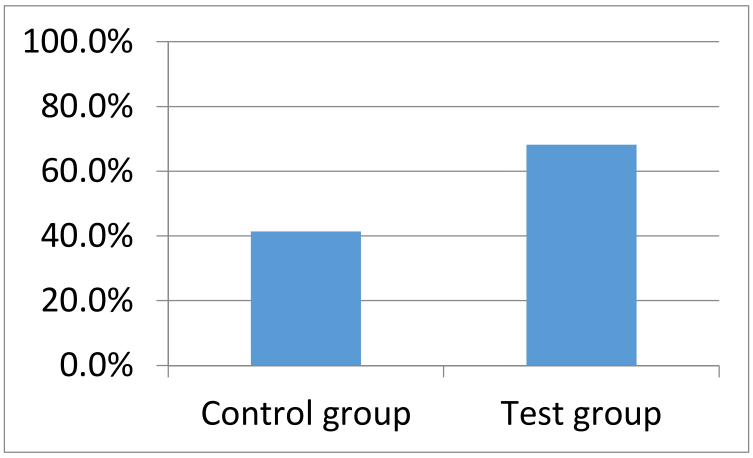


Fig. 3. Performance comparison: task A

Task B

Problem description:
Given an array of integers already in ascending order, write a program to insert an arbitrary new number into an existing set of numbers, such that the result is in an ascending order. However, you should follow the order given as follows: 1) determine what position the new number should go in; 2) shift all numbers greater than the new number; and 3) actually insert the new number.

The second task is a part of a sorting program that requires knowledge of and competence in handling 1-dimensional arrays. Both groups experienced ‘bubble sort’ during the semester. Sorting program is so simple that it is quite possible to memorize the code. In order to prevent “memorizing the solution”, we provided specific constraints for sorting algorithm. In this test, the treatment group again outperformed the control group. Only27.6% of the freshmen in the control group solved the problem, while 45.5% in the treatment group solved it. Figure 4 compares the pass ratios for the second test.

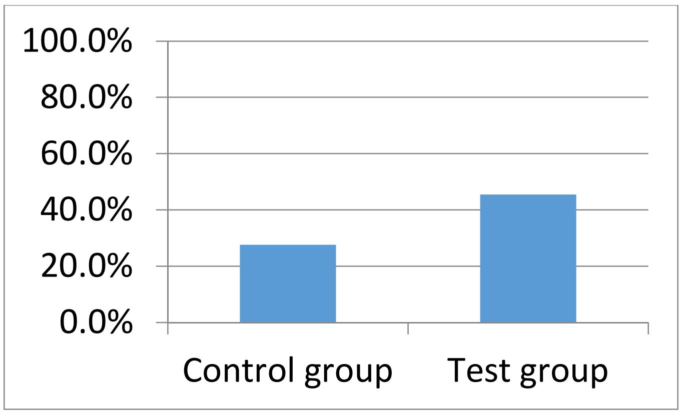


Fig. 4. Performance comparison on task B

Task C

Problem description: Given a representation of Omok (a.k.a., Gomoku) game in a 2-dimensional array, write a function to determine if a player has a winning configuration, i.e., 5 in a row.

Task C is to fill in some missing part of Omok game. For those who are not familiar with this game, omok is played by two players, black and white, taking turns to put a white or black stone until one of the player has 5 in a row, either straight or diagonally.

This problem requires handling of 2-dimensional arrays. On this task, the performance in the two groups was almost identical: 17 out of 29 students (58.6%) in the control group and 13 out of 22 students (59.1%) coped with the task.

Task C differed from tasks A and B in that the students were given the same problem as homework assignment before, which might explain why the performance of the two groups was similar.

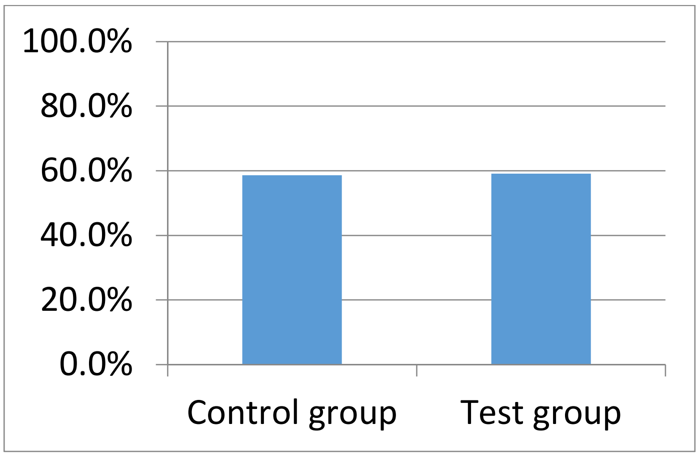


Fig. 5. Performance comparison on task C

The overall performance

The treatment group outperformed the control group on two out of three problems in the exam. More importantly, in the end of the semester, far more students in the treatment group passed the course. Specifically, 90.9% of the treatment group (i.e., those who learned CFL before learning C language) passed the course. By contrast, only 72.4% of the control group passed the course. The average scores of exams are 28.0 for control group and 40.9 for test group in a 100 scale. The pass ratio of the course in recent 3 years amounts to 74%. Therefore, 72.4% can be considered quite normal, while 90.9% can be considered exceptional.

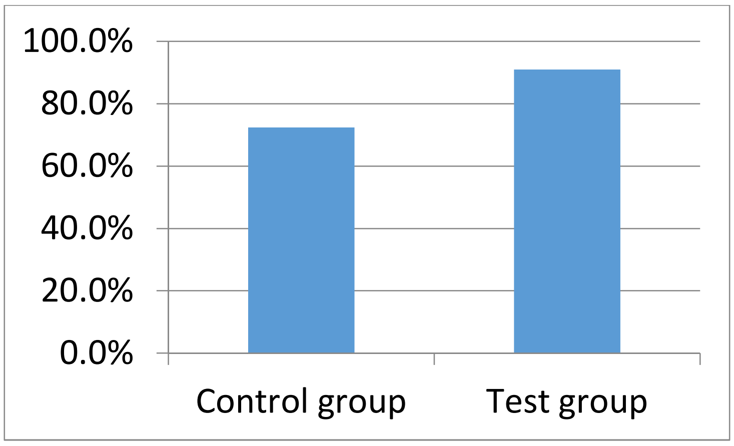


Fig. 6. Course pass ratios of the two groups

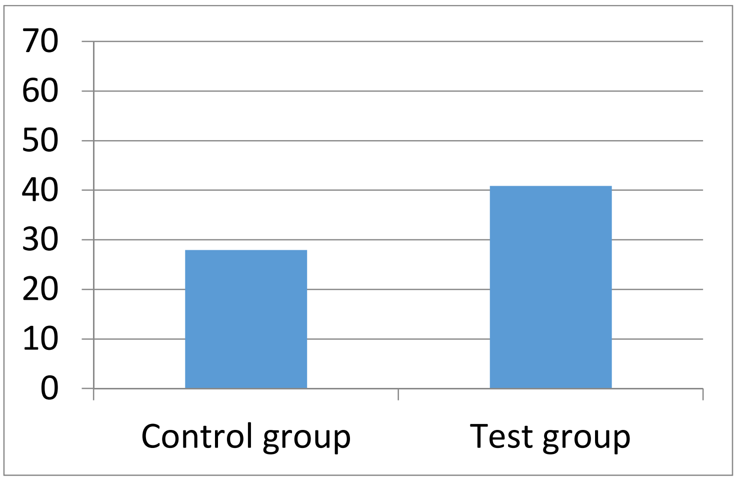


Fig. 7. Average scores of the two groups

4 SUMMARY AND FUTURE WORK

Our results show that the students who are taught CFL before learning C language eventually outperform those who learned C programming from the beginning. We ran a t-test analysis in Exel to see if the statistics is meaningful. The *t Stat* > *t Critical two-tail*, therefore we reject the null-hypothesis. In other words, it is unlikely that the difference of scores of the two classes is purely by chance.

Table 3. t-Test results

	Variable 1	Variable 2
Mean	20.44	13.98
Variance	93.45	67.47
Observations	22	29
Hypothesized Mean Difference	0	
t Stat	2.577183	
P(T<=t) one-tail	0.006511	
t Critical one-tail	1.676551	
P(T<=t) two-tail	0.013022	
t Critical two-tail	2.009575	

We believe that the reason why we benefit from CFL is because CFL is in fact a C language in (graphical) disguise. We experienced that the students who learn CFL have more fun and do better in terms of “thinking” of the solution, because of the graphical nature of the language and the visual execution environment. Furthermore, the competence acquired with CFL does not seem to be diminished later by the complexity of the C language syntax, for once one understands how to do something in CFL, one can easily do the same thing in C language.

Current version as of writing this paper is v.3.06, and we are currently working on Object-oriented flowchart, for teaching Java instead of C language.

REFERENCES

[1] Roberts, E. “An Overview of MiniJava”, ACM SIGCSE Bulletin 33 (1), 2001, pp. 1-5.

[2] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. “Mini-languages: A Way to Learn Programming Principles”, Education and Information Technologies 2 (1), 1997, pp. 65-83.

[3] Swan, D. “Programming Solutions for the LEGO Mindstorms NXT,” Robot magazine, 2010, p. 8.

[4] Sattar A., Lorenzen T. “Teach Alice programming to non-majors”, ACM SIGCSE Bulletin 41(2), pp. 118-121.

[5] Crews, T. “the Flowchart interpreter for Introductory Programming Courses”, Frontiers in Education Conference, 1998. Pp.307-312

[6] Martin C., Carlisle et al. “RAPTOR: introducing programming to non-majors with flowcharts”, Journal of Computing Sciences in Colleges 19(4), 2004, pp. 52-60