



Learning C language Programming with executable flowchart language

Prof. Cho Sehyeong, Myong Ji University

1992 Ph.D. in Computer Science, Pennsylvania State University 1992-1999 Electronics and Telecommunications Research Institute 2000-2014 MyongJi University

Prof. Yeonseung Ryu, Myongji University

Prof. Sang-Kyun Kim, Myongji University

Sang-Kyun Kim (BS '91, MS '95, PhD '97) received his BS, MS, and PhD degrees in computer science from the University of Iowa in 1991, 1994, and 1997, respectively. In 1997, he joined the Samsung Advanced Institute of Technology as a researcher. He was a senior research staff member as well as a project leader on the Image and Video Content Search Team of the Computing Technology Lab until 2007. He is now an associate professor in the Department of Computer Engineering of Myongji University. His research interests include digital content (image, video, and music) analysis and management, fast image search and indexing, color adaptation, 4D, sensors, VR, and multimedia standardization. He serves as a project editor of International Standards, that is, ISO/IEC 23005-3, 23005-4, 23005-5, and 23005-7.

Learning C language programming with executable flowchart language

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 30% of students fail the course and get frustrated that they are not fit for programming after all.

There are at least two reasons why programming is so hard 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 hard to get an accurate understanding of the operational semantics (i.e., effects) of the programming language constructs, which makes it difficult to predicting the accurate result of a program code. This, in turn, makes it hard to write a program. Second, regardless of the difficulty of programming language at hand, the problem-solving process itself is difficult. For beginners, linguistic difficulty matters, while, as you acquire more experience, the problem solving matters more.

In this paper, we will focus only on the linguistic issues, even though the problem-solving capability may have a greater impact in the long run.

We hypothesize that the use of proper visual aid can improve students' learning speed. One of the most popular visual aids for programming learning is a flowchart. In a programming class, it is conventional to explain the meaning of control structures, such as 'if-then-else' or 'do-while' by means of flowcharts. However, we propose to use an extended flowchart as a visual programming language, which is designed to enable smooth transition to commercial programming languages such as C, C++, or Java.

We developed a flowcharting tool that can be used for actual programming, as well as for executing and for debugging/visualizing. We conducted an experiment, the results of which support our hypothesis. In what follows, we will briefly introduce the tool used, and proceed with the discussion of the experiment and the results.

Related Work


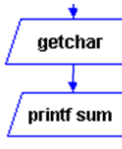
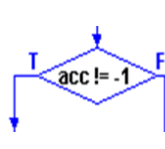

There are many different approaches to ease the problem of programming language learning. For instance, in order to avoid the complexity of full-fledged programming languages, one can use simplified programming languages, such as Mini-Java¹. In fact, mini-languages have been used for quite a long time². Iconic programming languages can be another possibility, such as Mindstorm NXT-G³ or Alice⁴. Iconic programming is suitable for a gentle introduction to

computer programming, and it has been reported to help keep students interested⁴. Flowchart programming has been used elsewhere as an aid to help students understand the concept and to improve their problem-solving skills⁵. In this study, we also use flowchart programming, but focus rather on short-term effects. Also, instead of using a flowchart to enhance overall problem-solving skills, we employ it for teaching specific language constructs, such as repetition structure or function calls.

A Brief Introduction to CFL

CFL, which stands for “C-like Flowchart Language,” is an executable flowchart-based programming system. CFL is developed to help students learn programming, as well as to understand the mechanism of program execution. In particular, it is designed for helping C-language learners, although it is useful to other language learners as well. CFL has four types of nodes related to execution: processing, I/O, decision, and function nodes. Table 1 summarizes CFL node types. CFL is executable, and, therefore, has features related to execution. These features include: one accumulator register, one floating point accumulator, 6 integer variables, 6 floating point variables, the input buffer, the output window, and two execution buttons – one for single stepping and the other for running the whole flowchart until it reaches the end. Integer variables are initially named from ‘a’ to ‘f’ and floating point variables are named ‘u’ to ‘z’, but they can be renamed. During the execution, students can watch the inner workings of the program: the control flow by a moving red dot, changing values by flashing colors, and the function call stack by a stack of parameters. CFL is tightly integrated into a web-based instruction system for efficient assigning of exercises, submitting, and grading⁶.

Table 1. CFL node types

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

All operations are intentionally aligned with C language statements for easier transition from CFL to C language. For instance, input statements “scanf” has not only the same name as in C language, but also has the same semantics: the return value is stored in the temporary memory – the accumulator – and has exactly the same meaning: the number of successful input items. To further relieve students from the burden of syntactic detail, node types change automatically. For example, if one types in ‘putchar’ inside a processing node, the node

automatically changes into an I/O node; if you type 'a=b' in an I/O node, it changes into a processing node. For another example, if one types in a function name, the processing node is marked as a function call node for easy visual identification (see in Figure 1).

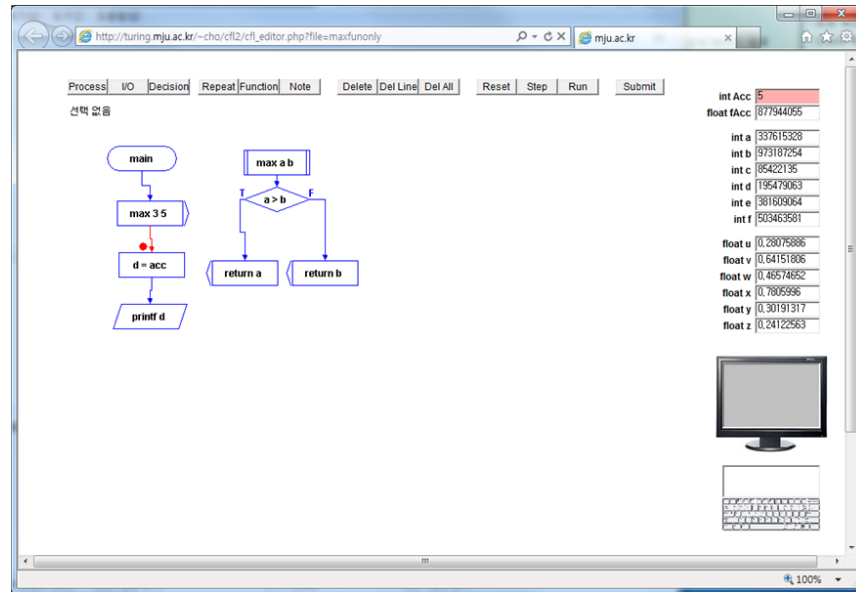


Figure 1. Snapshot of CFL editor/execution environment

The Experiment

First, we conducted an experiment in order to see if there is any enhancement in learning with CFL. Instead of measuring a long-term effect (e.g., one semester), we decided to measure rather a short-term effect. The reason is that long-term effects are not only difficult to measure, but can also be easily intervened by many unforeseen factors. Thus, in order to minimize such side effects, the study was planned as a very short-term and tightly-controlled experiment.

We had two classes of the C programming course, one with 32 students and the other with 33, all freshmen, CS majors. None of the students had taken programming courses before. They were taught by the same instructor on the same weekdays, but in different hours. The course is a 4 credit-hour course, taught twice a week, each class consisting of two consecutive class hours. One class – the control group – learnt the control structure of “while” loop in the C language syntax and was given two exercises in the class. The other class – the test group – first learnt the same concept by using CFL and was given two equivalent exercises in CFL, and then learned the C language syntax without actual exercises. The preparation took one class hour for each of the groups. Table 2 summarizes the activities during the first hour.

Table 2. Summary of preparation of the experiment

Groups	Control group	Test group
Instruction of the concept	by traditional flowchart	by traditional flowchart
Explanation of the syntax	“while” statement syntax explained	“while” statement syntax explained after CFL exercises
Exercises	two simple C exercises	two simple CFL exercises
Total time taken	50 minutes	50 minutes

After the preparation stage was accomplished, we conducted the main experiment in the second class hour of the same day. The students in both groups were given four tasks to write simple C programs requiring the use of “while” statement (see Table 3).

Table 3. Tasks used in the experiment

Task 1	Write a program to print all numbers from 1 to 99.
Task 2	Write a program to print all even numbers from 100 to 0 in descending order.
Task 3	Write a program to find the largest number N such that the sum of all integers from 1 to N (i.e., $1+2+3+...+N$) does not exceed 1000.
Task 4	Write a series of numbers such that the first number is 0, the second number is 1, and the difference between one number and the next number increases by 1 each time (progression of difference).

The system records the exact time when a student retrieves the problem and the time when the solution is submitted and graded. The tasks were so simple that we did not take into account the ‘quality’ of the solution; it is mechanically graded by the program and accepted if the result is correct. Since this was an in-class experiment limited in time, we could not afford enough time to allow all students to complete their tasks. Figure 2 shows the completion time of the first 23 students in each class, from fast to slow. Twenty-three was the minimal number of successful students across all four tasks.

In the first two tasks, the test group performed consistently slower than the control group. In the third task, the test group seemed to be catching up slightly. However, in the last task, the test group outperformed the control group by an average of 69 seconds (see Table 4).

Table 4. Average completion time for each task

	Task1	Task2	Task3	Task4
Control group	91.2	179.9	478.7	601.1
Test group	101.7	228.2	496.4	530.4

What is more interesting than the average completion time, however, is the distribution pattern in the completion time, as illustrated in Figure 2. It is noteworthy that “excellent” students in the test group were slightly – if not significantly – slower than their counterparts from the control group, while “average” students in the test group were noticeably better than the

students in the control group. Two observations can be made from these experimental results. First, for “excellent” students, the relative complexity of task 4 does not appear to have mattered. In other words, it was an easy problem for them to solve anyway. Secondly, for “average” students, the complexity of task 4 seems to have mattered. We believe that this result implies that having exercised with CFL made the students solve problems better (faster) when encountered with relatively more difficult problems, which offset any prior disadvantage in familiarity with the C language syntax.

Conclusion and Future Work

The central aspect that distinguishes our experiment from most other experiments is that it has been conducted inside a single course session. Therefore, the experiment is very accurately conditioned. On the other hand, this is admittedly a small-scale experiment, which makes it difficult to predict long-term effects of teaching with CFL. In the coming year, we plan to scale up the experiment to spend the first 4 weeks during the 16-week semester only with CFL, and then transition to C language to investigate the medium-term effects of teaching with CFL.

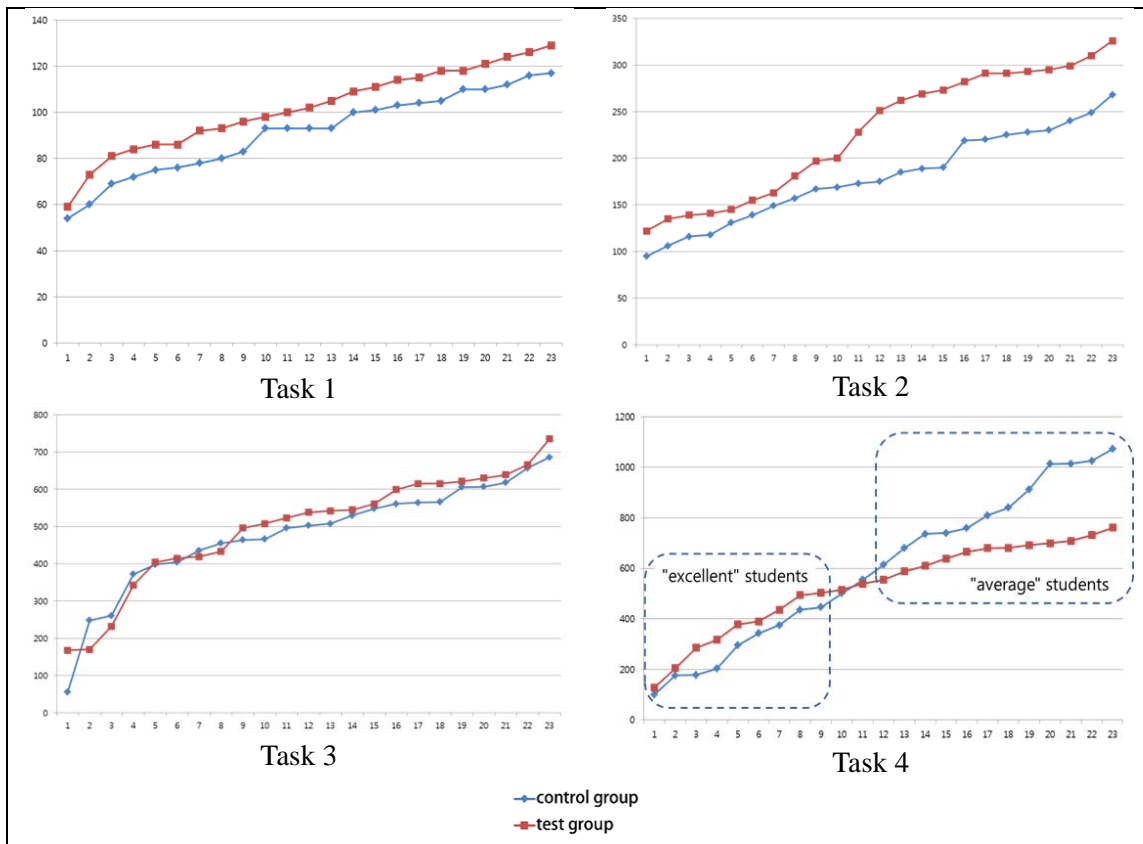


Figure 2. Time to complete the tasks (in seconds)

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. "Using a Flowchart Simulator in a Introductory Programming Course," <http://www.citidel.org/bitstream/10117/119/2/visual.pdf> Last accessed Dec.30, 2013
6. Sehyeong Cho, "A Web-based Tool for Teaching Computer Programming," Proceedings of Asian Conference on Engineering Education, 2013, pp.132-133