International Journal of Computer Science and Information Technology
ISSN: 3005-9682 (Print), ISSN: 3005-7140 (Online) | Volume 8, Number 3, Year 2026
DOI: https://doi.org/10.62051/ijcsit.v8n3.01
Journal homepage: https://wepub.org/index.php/IJCSIT/index

From Everyday Reasoning to Program Construction: Instructional Episodes from an Introductory C Programming Course for Non-Computer Majors

Wei Gao, Fengqi Zhang

Department of Computer Science, North China Electric Power University(Baoding), Baoding, China

ABSTRACT

Introductory programming is particularly challenging for students from non-computer majors, not only because of syntactic complexity but also due to the lack of a cognitive transition from everyday reasoning to algorithmic problem solving. This study presents a series of instructional episodes from an introductory C programming course to illustrate how abstract programming concepts can be grounded in familiar real-life experiences. These episodes show how students transitioned from intuitive understanding to structured reasoning and ultimately to independent program construction. Classroom observations indicate increased willingness to initiate coding, stronger persistence in debugging, and an emerging ability to transfer solution patterns to new problems. The findings suggest that connecting algorithmic concepts with everyday cognitive experiences reshapes students' perception of programming from knowledge memorisation to meaningful problem solving. This narrative provides practical and transferable teaching insights for programming education in similar contexts.

KEYWORDS

Algorithmic thinking; Cognitive transition; Everyday reasoning; Instructional episode; Programming course

1. INTRODUCTION

Introductory programming courses are widely recognised as challenging for students from non-computer majors. The difficulty is often attributed to the complexity of syntax and abstract computational concepts. However, classroom experience shows that many students can understand individual statements yet remain unable to construct a complete program when faced with a new problem. This phenomenon suggests that the primary obstacle lies not in syntactic knowledge but in the absence of a cognitive pathway from problem understanding to algorithm construction [1].

Computational thinking involves breaking down problems, abstracting core components, and representing solutions in a procedural manner [2, 3]. For novice learners without prior programming experience, these processes are highly abstract and cognitively demanding. When programming instruction begins directly with formal syntax and predefined program structures, students tend to rely on imitation rather than genuine problem solving [4]. As a result, they hesitate to start coding independently and easily give up when encountering errors.

Our long-term teaching practice consistently shows that students engage more actively when programming concepts are introduced through familiar everyday experiences. This recurring observation raises a pedagogical question: how can such experiential connections support the cognitive transition from understanding a problem to constructing a program?

This study addresses this question by presenting a series of classroom instructional episodes. By tracing how familiar experiences were used to introduce different programming constructs, the paper demonstrates the gradual transformation of students' learning behaviour from passive imitation to active problem solving and reveals a coherent cognitive progression in their understanding of program construction.

2. TEACHING CONTEXT AND INITIAL CHALLENGES

The course was the first programming-related subject for all students, and most of them had no prior exposure to algorithmic thinking. At the beginning of the course, several recurrent learning patterns were observed:

① Students were able to understand given programs but were unable to articulate the underlying problem-solving strategy or explain how such a strategy could be constructed [5].

② When required to write programs independently, they often did not know how to get started.

③ When debugging attempts failed, they tended to abandon the task quickly.

④ They tended to perceive programming as a process of memorising isolated knowledge points rather than solving problems.

These patterns indicate that the primary difficulty lies not in acquiring syntactic knowledge but in the absence of a cognitive transition from problem understanding to algorithm construction. In particular, students lacked an intermediate representational stage through which everyday reasoning could be gradually transformed into algorithmic thinking.

This diagnosis motivated the adoption of a progressive instructional pathway. Instead of introducing programming through formal syntax and predefined program structures, the teaching began with familiar experiential contexts, guided students to externalise solution processes as ordered steps, and only then introduced syntax as a representation of an already formed algorithmic structure. In this way, a cognitive bridge was constructed between problem understanding and program implementation.

3. INSTRUCTIONAL APPROACH: A PROGRESSIVE COGNITIVE PATHWAY

Our teaching followed a progressive pathway that supported students in moving from understanding a problem situation to constructing a complete program. A major obstacle for novice learners was that, although they could understand individual statements, they were unable to enter the problem space when faced with a new task. When programming was introduced through formal syntax at the outset, their attention was immediately drawn to syntactic form and correctness, which prevented them from analysing the structure of the problem itself.

To create an accessible cognitive entry point, programming concepts were first grounded in familiar everyday experiences. In these contexts, students were able to reason about the situation using their existing knowledge and to propose solution strategies without the pressure of syntactic accuracy. The focus of learning was thus shifted from "how to write the code" to "how the problem can be solved", allowing students to form an initial understanding of the problem structure [6].

Once this experiential understanding had been established, students were guided to describe the solution process in natural language and to organise it into explicit and ordered steps. This externalisation of reasoning created an intermediate representational stage in which the underlying problem-solving strategy became visible and discussable.

Only after the algorithmic structure had been formed was programming syntax introduced as a means of expressing an already constructed solution. Code was therefore perceived not as a set of rules to be memorised but as a representation of a reasoning process.

Through repeated application across different topics, this sequence gradually enabled students to internalise a stable pathway from problem understanding to program construction. Rather than reducing the cognitive demand of programming, the approach redistributed it by establishing a clear transition from experiential understanding to formal algorithmic expression.

4. CLASSROOM INSTRUCTIONAL EPISODES

4.1. Conditional Structures: From Partitioning a Whole to Ordered Decision Paths

When conditional structures were first introduced, most students were able to state the grading rules in natural language but were unable to organise multiple branches into a coherent program. Although the problem appeared simple, they hesitated to begin coding because they could not establish an ordered decision path.

Instead of presenting the syntax of nested if statements directly, the lesson began with a concrete question drawn from everyday experience: how to cut a rope into several segments. Students proposed different cutting strategies, such as dividing the rope into equal parts or separating a small segment first. Although all strategies produced the required segments, they followed different cutting sequences, which helped students recognise that a multi-branch problem must be organised as an ordered decision process. Classification was thus understood not as a static partition but as a sequence of decisions.

Students were then asked to return to the grading problem and determine the order in which score ranges should be evaluated. At this point, the conditional structure emerged naturally as the representation of this ordered decision path. Writing the program was no longer perceived as constructing a syntactic form but as expressing an already established reasoning process. Table 1 summarises this cognitive mapping from everyday experience to algorithmic structure and code.

Table 1. From Problem Context to Code: Cognitive Mapping for Conditional Structure

Stage	Description
Problem	Input a score on a 100-point scale and output the corresponding grade: A (≥90), B (80~89), C (70~79), D (60~69), E (<60).
Everyday experience	Cutting a rope into several segments. Different cutting strategies imply different decision orders, and each cut determines the next possible operation.
Algorithmic structure	Organising multiple conditions as an ordered decision path, in which score ranges are examined sequentially from higher to lower (or vice versa).
Code	A nested or sequential if_else structure that evaluates conditions step by step and outputs the corresponding grade.

4.2. Iteration: Existential Verification in Prime Testing

When iteration was introduced, students were able to accept the definition of a prime number but had difficulty understanding why the program had to examine a sequence of possible divisors and why the conclusion of primality could only be made after the loop had finished. They tended to treat the loop as a fixed syntactic pattern rather than as a representation of a verification process.

To make the underlying logic visible, the lesson began with a familiar question from everyday reasoning: how to prove that all swans are white. Students quickly realised that confirming such a statement would require examining every swan, whereas rejecting it would only require finding a

single counterexample. This contrast made it clear that different types of conclusions correspond to different verification processes.

This reasoning was then mapped onto prime testing. Searching for a divisor was understood as looking for a counterexample: once a divisor was found, the conclusion could be reached immediately; otherwise, the examination had to continue until all possibilities were exhausted. The loop thus emerged as a process of systematic verification rather than a repetitive control structure.

A further question concerned how to determine the result after the loop terminated. Since the loop might end either through early exit or after exhausting all candidates, a persistent record of the search outcome was required. A simple gesture-based activity was used to represent this idea: a raised hand indicated that a divisor had been found, while no gesture indicated its absence. This representation was mapped onto the flag variable, which records whether a specific event has occurred and is checked after the loop to reach the final decision. In this way, the flag was understood as a state recorder rather than as an abstract programming construct.

Table 2 summarises this cognitive mapping from everyday reasoning to the algorithmic meaning of iteration in prime testing. Through this transition, the loop was no longer perceived as a syntactic pattern but as the formal expression of a verification process.

Table 2. Cognitive Mapping for Iteration in Prime Testing

Everyday reasoning	Algorithmic meaning
counterexample	divisor
universal verification	full loop
existential falsification	early termination
gesture	flag variable

4.3. Linear Search: Task-Dependent Search Strategies

When linear search was introduced, students were able to describe the basic action of examining elements one by one, but they tended to treat each task as requiring a different program. Determining whether an element existed, stopping as soon as it was found, and reporting its position were initially perceived as three unrelated problems rather than as different interpretations of the same computational process.

To make the search process meaningful, the lesson began with a familiar situation: looking for a specific person in a classroom by asking students one by one. This scenario provided a natural sequential structure and enabled students to describe the procedure using everyday language. By changing the task requirements—confirming the presence of the person, leaving immediately after finding the person, or reporting the seat number—the same action was performed under different goals.

Through this comparison, students gradually realised that the underlying process did not change. In all cases, the search proceeded sequentially from one student to the next; what differed was the stopping condition and the interpretation of the result. Early termination not only indicated that the target had been found but also implicitly provided its position: stopping inside the classroom corresponded to success, whereas reaching the position beyond the last student indicated failure.

This realisation unified the three tasks within a single algorithmic structure. The loop index was no longer seen as a simple counter but as a carrier of the search state, simultaneously representing whether the target had been found and where it was located. Linear search thus emerged not as a fixed code template but as a task-dependent strategy in which different problem goals correspond to different interpretations of the same process. Table 3 summarises this cognitive mapping from everyday reasoning to algorithmic meaning.

Table 3. Cognitive Mapping for Task-Dependent Linear Search

Everyday reasoning	Algorithmic meaning
asking students one by one	sequential traversal
leaving immediately after finding	early termination
seat number at stopping point	index as position
reaching the position beyond the last student	search failure

4.4. Functions: From Task Decomposition to Program Organisation

When functions were introduced, students were able to follow examples of function definition and invocation, but they tended to regard functions as a syntactic requirement rather than as a means of organising a complex program. Most of them placed all operations inside the main function and were unable to design functional structures independently.

To make the organisational role of functions explicit, the lesson began with an analogy to project management. The main function was described as a coordinator responsible for dividing a large task into manageable subtasks and assigning them to different members. Through this analysis, each subtask naturally corresponded to a function.

Once the division of labour had been established, attention shifted to how these functional units cooperated. The main function provided the original data required for each subtask, while the called function either returned a result or updated existing data. Parameter passing was thus understood as the allocation of resources, and the return value as the outcome delivered back to the coordinator.

A further distinction emerged among different types of subtasks. Some produced new results that had to be reported back, some transformed existing data without generating a separate output, and others only performed an operational action. In the project analogy, these corresponded respectively to a team member submitting a report, a member updating a shared resource, and a member completing a logistical task. This idea was reinforced through familiar devices: a dumpling maker that produces a new product, a washing machine that transforms the clothes in place, and a device such as a light or an air conditioner that changes the state of its surroundings..

Through this transition, a program was no longer perceived as a linear sequence of statements but as a structured system of cooperating components connected by data flow. Functions thus emerged not only as a mechanism for task decomposition but as the organisational units through which the interaction and coordination of different parts of a program are realised. Table 4 summarises this cognitive mapping between project organisation and program structure.

Table 4. Cognitive Mapping for Functions

Project organisation	Program structure	Cognitive meaning
project coordinator	main function	global control
task decomposition	function design	modularisation
resource allocation	parameter passing	input data flow
submitting a report	return value	producing new data
updating a shared resource	in-place data modification	state transformation
operational duty	void function	performing an action

5. CHANGES IN STUDENTS' LEARNING BEHAVIOUR

Long-term classroom observation across multiple cohorts revealed consistent changes in students' learning behaviour. These changes correspond closely to the initial learning patterns described in Section 2 and were repeatedly observed in different teaching cycles.

At the initial stage of learning, students tended to hesitate when asked to start coding independently and relied heavily on given examples. After guided practice grounded in familiar experiential contexts, they began to analyse problem structures before writing code and showed greater willingness to attempt independent debugging. As this problem-solving pathway became familiar, many students were able to transfer previously learned solution patterns to new tasks.

More importantly, students demonstrated markedly stronger persistence when facing errors. Instead of immediately asking for the correct answer, they tried multiple approaches to locate and correct the problem. This behavioural transformation indicates a shift from imitation-based learning to problem-oriented thinking.

6. PEDAGOGICAL IMPLICATIONS

The instructional episodes presented in Section 4 reveal a common cognitive pattern underlying the learning of different programming structures. In each case, the programming construct was introduced not as a syntactic form but as the representation of an already established problem-solving process. Conditional statements emerged as ordered decision paths, iteration as a process of systematic verification, linear search as a goal-dependent interpretation of a single traversal mechanism, and functions as the organisation of tasks and data flow within a program.

These observations suggest that the central difficulty in introductory programming for non-computer majors lies in the absence of a cognitive transition between everyday reasoning and formal algorithmic representation. Familiar experiential contexts provide an accessible entry point into the problem space, while stepwise externalisation of reasoning makes the underlying structure of the solution visible and discussable.

The instructional pathway described in this study therefore does not reduce the complexity of programming. Instead, it reorganises the learning process by ensuring that algorithmic meaning precedes syntactic expression. Syntax is thus repositioned as the final representation of a constructed solution rather than the starting point of learning.

This principle offers a transferable pedagogical insight: a wide range of programming topics can be approached through a similar cognitive progression from experiential understanding to structured reasoning and finally to program implementation.

7. CONCLUSION

This study addresses a central difficulty in introductory programming for non-computer majors: the absence cognitive transition from everyday reasoning to formal algorithmic representation. Drawing on long-term teaching practice, it shows how programming concepts can be introduced through familiar experiences and gradually transformed into structured reasoning and program implementation.

The observations reveal a clear shift in students' learning behaviour from imitation and hesitation to active problem solving, independent debugging, and the transfer of solution strategies. When algorithmic meaning precedes syntactic expression, programming is perceived as a meaningful problem-solving process rather than the memorisation of isolated rules.

The cognitive progression documented in this study provides a practical and transferable instructional pathway for supporting novice learners in similar educational contexts.

ACKNOWLEDGMENTS

This work was supported by 'the Fundamental Research Funds for the Central Universities (2023MS136)'

REFERENCES

[1] Soloway, E., & Spohrer, J. C. (Eds.). (1989). Studying the novice programmer. Psychology Press. https://doi.org/10.4324/9781315808321

[2] Wing, J. M. (2006). Computational thinking. Communications of the ACM, 49(3), 33-35. https://doi.org/10.1145/1118178.1118215

[3] Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. Educational Research Review, 22, 142–158. https://doi.org/10.1016/j.edurev.2017.09.003

[4] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. Computer Science Education, 13(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[5] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bulletin, 36(4), 119-150. https://doi.org/10.1145/1041624.1041673

[6] Kolodner J. L. (1997). Educational implications of analogy. A view from case-based reasoning. The American psychologist, 52(1), 57-66. https://doi.org/10.1037//0003-066x.52.1.57