A Brief Literature Review on Errors & Misconceptions in C Programming, and effective interventions

Paper 1: Common Logic Errors for Programming Learners (2021) [1]

This survey reviewed 47 publications from 1985–2018, identifying 166 common logic errors, many of which are relevant to C programming. Errors were classified into 11 categories:

- **Variables**: Uninitialized variables, wrong data types, misuse in I/O operations.
- **Computation**: Integer division instead of floating-point, misuse of operators, incorrect precedence.
- **Conditions**: Confusing `=` and `==`, missing logical operators (`&&`, `||`), improper boolean usage.
- **Loops**: Off-by-one errors, infinite loops, incorrect initialization or update of loop counters.
- **Arrays**: Out-of-bounds access, indexing errors, incomplete initialization.
- **Functions**: Ignoring return values, missing return statements in non-void functions, parameter mismatches.

Although the paper provides a broad literature survey, it often lacks detailed contextual information such as the exact programming language or course settings in which the data were collected. Despite this limitation, many of the highlighted logic errors are language-agnostic and map directly to issues frequently encountered in C programming. For example, integer division mistakes, off-by-one loop errors, and ignoring return values are both common and time-consuming to fix in C.

Paper 2: Catalogs of C and Python Antipatterns by CS1 Students (2021) [2]

This study analyzed code submissions from students in a distance-learning course to identify recurring *antipatterns* in C and Python.

- Identified 95 misconceptions in C, with 21 consolidated as common antipatterns.
- Frequent issues included:
  - **Input/Output**: Misuse of `scanf`/`printf`, incorrect use of `&` in `scanf`, missing format specifiers.
  - **Arithmetic**: Integer division mistakes, uninitialized variables, incorrect formulas.
  - **Control structures**: Misplaced braces, dangling `else`, extra semicolons.
  - **Functions**: Missing or incorrect return statements, parameter mismatches.
  - **Style**: Poor indentation, naming, or unnecessary redundancy.

Pointer / address-related notes:

- Missing `&` in front of variable in `scanf` (program crashes or reads garbage).
- Using `&` in `printf`, printing the address instead of the value.
- Using `&` instead of `%` in format specifiers.
- Printing uninitialized variables (garbage from memory).

A screenshot of the complete C antipattern list from the paper can be included below:

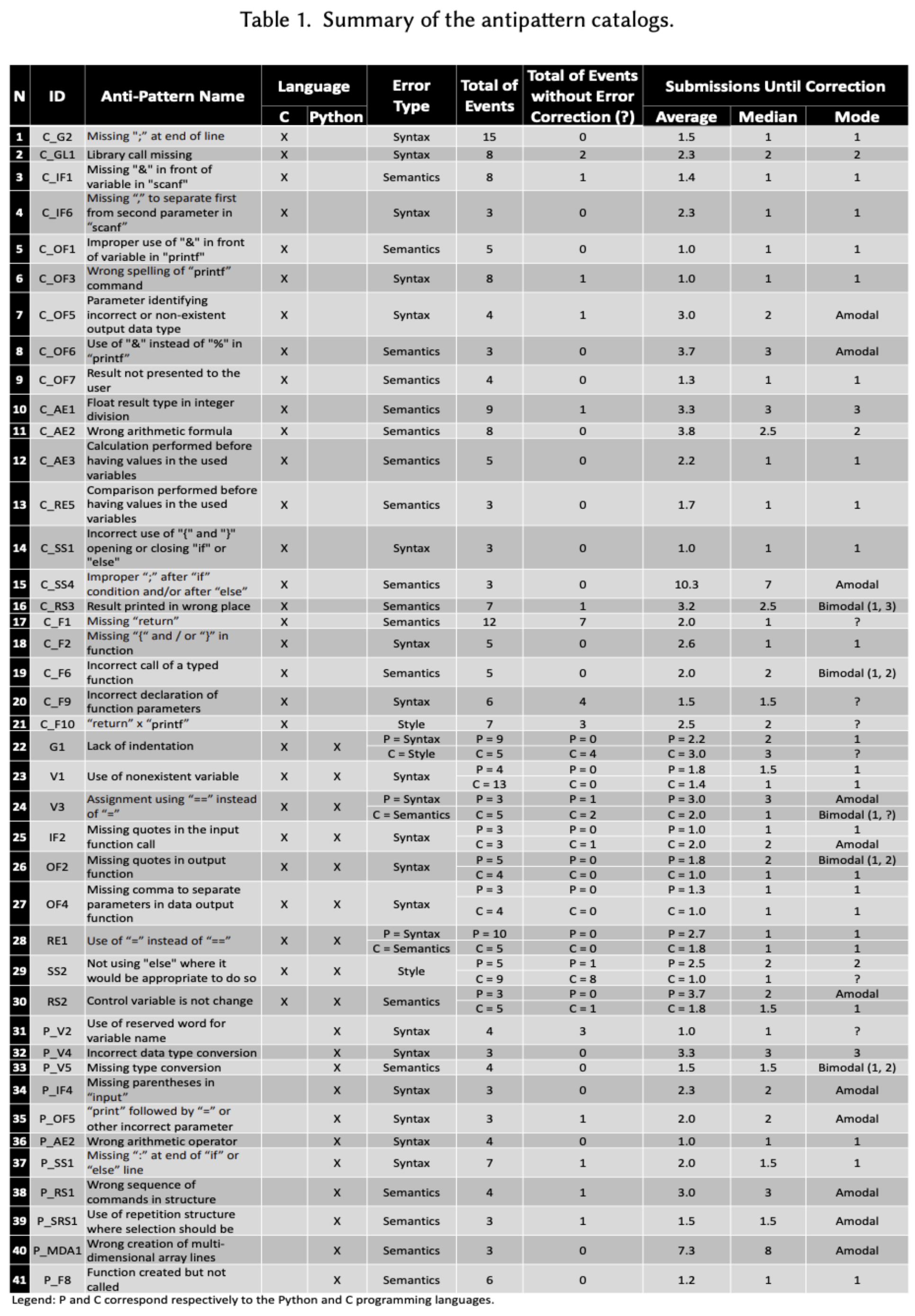


Table 1. Summary of the antipattern catalogs.

N	ID	Anti-Pattern Name	Language		Error Type	Total of Events	Total of Events without Error Correction (?)	Submissions Until Correction		
			C	Python				Average	Median	Mode
1	C_G2	Missing ";" at end of line	X		Syntax	15	0	1.5	1	1
2	C_GL1	Library call missing	X		Syntax	8	2	2.3	2	2
3	C_IF1	Missing "&" in front of variable in "scanf"	X		Semantics	8	1	1.4	1	1
4	C_IF6	Missing "," to separate first from second parameter in "scanf"	X		Syntax	3	0	2.3	1	1
5	C_OF1	Improper use of "&" in front of variable in "printf"	X		Semantics	5	0	1.0	1	1
6	C_OF3	Wrong spelling of "printf" command	X		Syntax	8	1	1.0	1	1
7	C_OF5	Parameter identifying incorrect or non-existent output data type	X		Syntax	4	1	3.0	2	Amodal
8	C_OF6	Use of "&" instead of "%" in "printf"	X		Semantics	3	0	3.7	3	Amodal
9	C_OF7	Result not presented to the user	X		Semantics	4	0	1.3	1	1
10	C_AE1	Float result type in integer division	X		Semantics	9	1	3.3	3	3
11	C_AE2	Wrong arithmetic formula	X		Semantics	8	0	3.8	2.5	2
12	C_AE3	Calculation performed before having values in the used variables	X		Semantics	5	0	2.2	1	1
13	C_RE5	Comparison performed before having values in the used variables	X		Semantics	3	0	1.7	1	1
14	C_SS1	Incorrect use of "{" and "}" opening or closing "if" or "else"	X		Syntax	3	0	1.0	1	1
15	C_SS4	Improper ";" after "if" condition and/or after "else"	X		Semantics	3	0	10.3	7	Amodal
16	C_RS3	Result printed in wrong place	X		Semantics	7	1	3.2	2.5	Bimodal (1, 3)
17	C_F1	Missing "return"	X		Semantics	12	7	2.0	1	?
18	C_F2	Missing "{" and / or "}" in function	X		Syntax	5	0	2.6	1	1
19	C_F6	Incorrect call of a typed function	X		Semantics	5	0	2.0	2	Bimodal (1, 2)
20	C_F9	Incorrect declaration of function parameters	X		Syntax	6	4	1.5	1.5	?
21	C_F10	"return" x "printf"	X		Style	7	3	2.5	2	?
22	G1	Lack of indentation	X	X	P = Syntax	P = 9	P = 0	P = 2.2	2	1
					C = Style	C = 5	C = 4	C = 3.0	3	?
23	V1	Use of nonexistent variable	X	X	Syntax	P = 4	P = 0	P = 1.8	1.5	1
						C = 13	C = 0	C = 1.4	1	1
24	V3	Assignment using "==" instead of "="	X	X	P = Syntax	P = 3	P = 1	P = 3.0	3	Amodal
					C = Semantics	C = 5	C = 2	C = 2.0	1	Bimodal (1, ?)
25	IF2	Missing quotes in the input function call	X	X	Syntax	P = 3	P = 0	P = 1.0	1	1
						C = 3	C = 1	C = 2.0	2	Amodal
26	OF2	Missing quotes in output function	X	X	Syntax	P = 5	P = 0	P = 1.8	2	Bimodal (1, 2)
						C = 4	C = 0	C = 1.0	1	1
27	OF4	Missing comma to separate parameters in data output function	X	X	Syntax	P = 3	P = 0	P = 1.3	1	1
						C = 4	C = 0	C = 1.0	1	1
28	RE1	Use of "=" instead of "=="	X	X	P = Syntax	P = 10	P = 0	P = 2.7	1	1
					C = Semantics	C = 5	C = 0	C = 1.8	1	1
29	SS2	Not using "else" where it would be appropriate to do so	X	X	Style	P = 5	P = 1	P = 2.5	2	2
						C = 9	C = 8	C = 1.0	1	?
30	RS2	Control variable is not change	X	X	Semantics	P = 3	P = 0	P = 3.7	2	Amodal
						C = 5	C = 1	C = 1.8	1.5	1
31	P_V2	Use of reserved word for variable name		X	Syntax	4	3	1.0	1	?
32	P_V4	Incorrect data type conversion		X	Syntax	3	0	3.3	3	3
33	P_V5	Missing type conversion		X	Semantics	4	0	1.5	1.5	Bimodal (1, 2)
34	P_IF4	Missing parentheses in "input"		X	Syntax	3	0	2.3	2	Amodal
35	P_OF5	"print" followed by "=" or other incorrect parameter		X	Syntax	3	1	2.0	2	Amodal
36	P_AE2	Wrong arithmetic operator		X	Syntax	4	0	1.0	1	1
37	P_SS1	Missing ":" at end of "if" or "else" line		X	Syntax	7	1	2.0	1.5	1
38	P_RS1	Wrong sequence of commands in structure		X	Semantics	4	1	3.0	3	Amodal
39	P_SRS1	Use of repetition structure where selection should be		X	Semantics	3	1	1.5	1.5	Amodal
40	P_MDA1	Wrong creation of multi-dimensional array lines		X	Semantics	3	0	7.3	8	Amodal
41	P_F8	Function created but not called		X	Semantics	6	0	1.2	1	1

Legend: P and C correspond respectively to the Python and C programming languages.

Figure: C antipatterns from Paper 2

Paper 3: Common Logic Errors Made by Novice Programmers (2018) [3]

This study analyzed over 15,000 student submissions from a first-year engineering programming course that taught MATLAB first and then C. The focus was exclusively on logic errors in the C portion of the course, providing a rich empirical dataset of errors specific to novice C programmers.

Errors were grouped into three categories:

- **Algorithmic errors**: Flawed or incomplete algorithms, e.g., not handling equal values when finding the maximum.
- **Misinterpretation errors**: Misunderstanding task requirements, e.g., returning 0 instead of -1 when a search fails.
- **Misconceptions**: Fundamental misunderstandings of C programming, e.g., array indexing or uninitialized variables.

Key recurring C-related misconceptions included:

- Integer division truncates results unless explicitly cast to double.
- Assuming uninitialized variables default to 0 rather than containing indeterminate garbage values.
- Off-by-one errors in array traversal, such as using `<=` instead of `<` in for loop conditions.
- Forgetting that array indexing in C starts at 0, leading to out-of-bounds errors.
- Misuse of `printf` instead of returning values, confusing side effects with function outputs.

Pointer / memory-related notes:

- Out-of-bounds array access often led to reading invalid memory, producing unpredictable garbage values or crashes.
- Accessing uninitialized variables exposed raw memory contents, reinforcing misconceptions about memory safety in C.

Paper 4: Student Difficulties with Pointer Concepts in C (2016) [4]

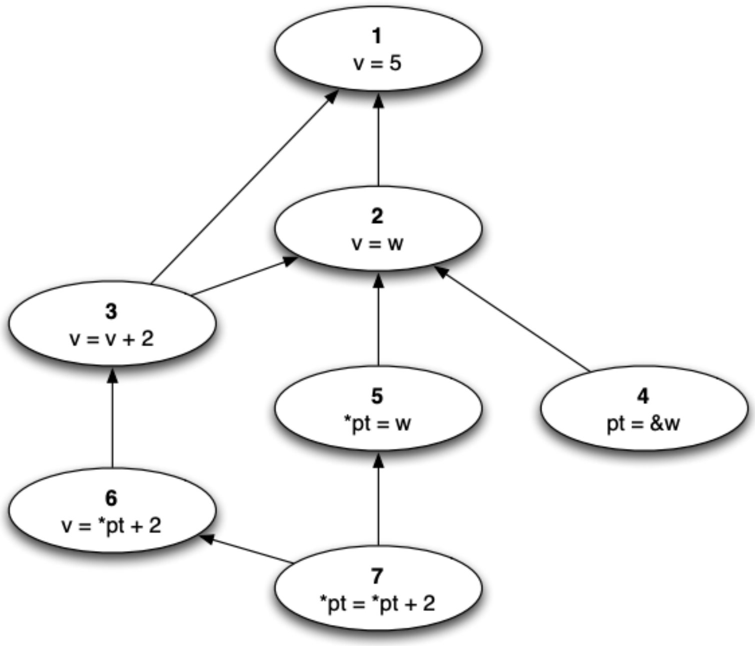
This study by Craig and Petersen analyzed difficulties students face with pointer concepts in C. The authors built a taxonomy of pointer operations, then evaluated their difficulty using pre-/post-tests and coding exercises in a second-year systems programming course.

Methodology

- **Participants:** 341 students (second-year, prior experience in Python/Java).
- **Materials:** Lab with seven coding tasks and three multiple-choice (MC) questions.
- **Tests:** “Select all that apply” MC questions before and after the lab, covering assignment, dereferencing (LHS/RHS), symmetric dereference, arrays, and double pointers.

- **Analysis:** Compilation logs, abstract syntax trees, and error/warning messages were analyzed to identify misconceptions.

Figure 1: Expected Hierarchy of C Pointer Concepts. An arrow indicates a dependency.



Observations and Misconceptions

The results revealed consistent misconceptions about how pointers work in C:

- **Relative concept difficulty:** Simple assignments without pointers (e.g., `v = w`) were easiest, while dereferencing operations were hardest. Surprisingly, students performed better on `*p = *p + i` (symmetric dereference) than on isolated LHS/RHS dereferences, suggesting alternative mental models.

- **Address vs. value confusion:** Students often omitted or misapplied the `&` operator, treating variables and their addresses interchangeably.

- **Pointer-to-pointer complexity:** Declaring and using `char **` was highly error-prone, with fewer than 20% succeeding on first attempt.

- **Arrays and pointers:** Common issues included assigning array elements directly to pointers (`int *back = a[4];`), off-by-one errors, and confusion about pointer arithmetic.

- **Function parameters:** Students frequently attempted to return values instead of modifying arguments via pointers, and tended to apply operators symmetrically to both parameters.

- **Compiler feedback:** Many ignored or misinterpreted warnings such as "initialization makes pointer from integer without a cast," leading to more errors in subsequent attempts.

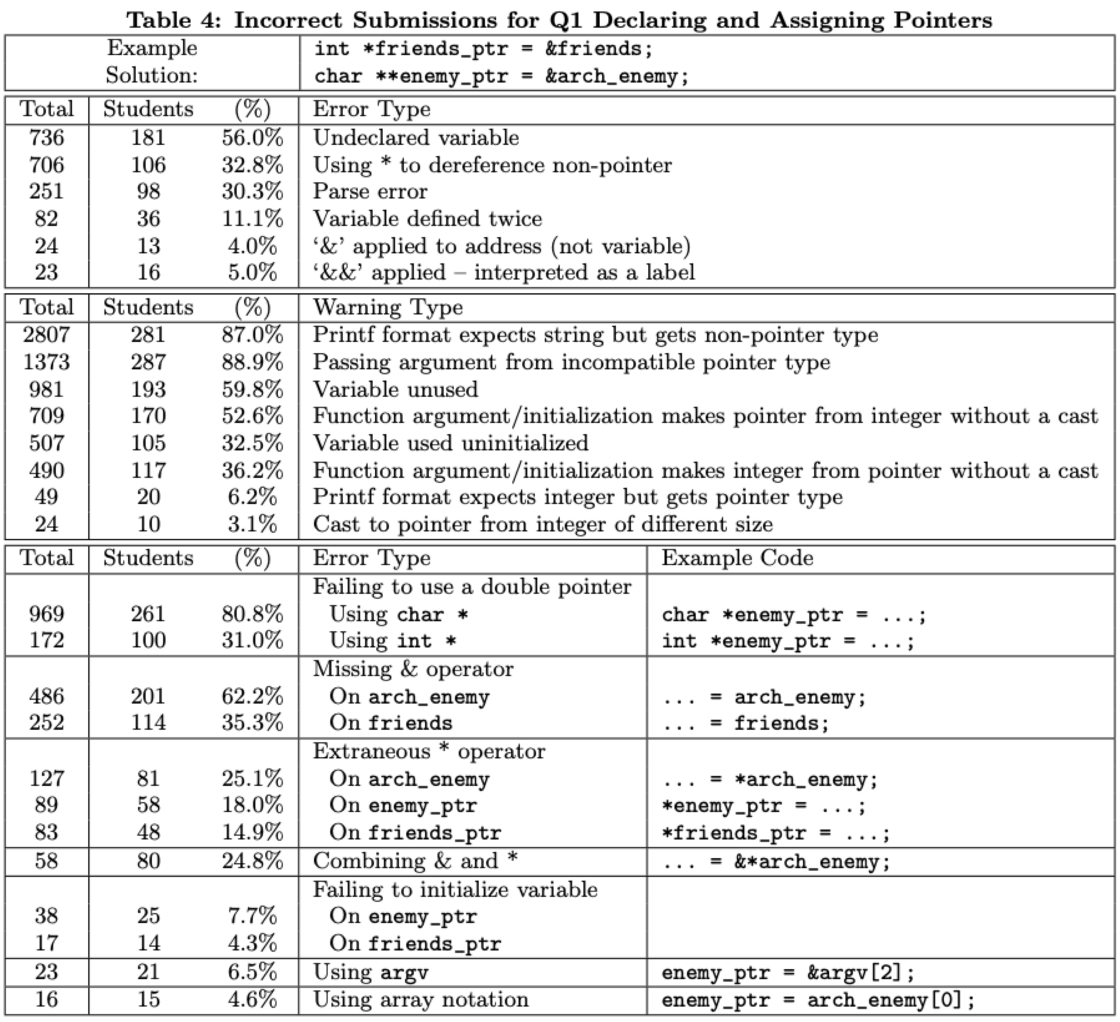


Table 4: Incorrect Submissions for Q1 Declaring and Assigning Pointers

Example Solution:	
	`int *friends_ptr = &friends;` `char **enemy_ptr = &arch_enemy;`

Total	Students	(%)	Error Type
736	181	56.0%	Undeclared variable
706	106	32.8%	Using * to dereference non-pointer
251	98	30.3%	Parse error
82	36	11.1%	Variable defined twice
24	13	4.0%	'&' applied to address (not variable)
23	16	5.0%	'&&' applied – interpreted as a label

Total	Students	(%)	Warning Type
2807	281	87.0%	Printf format expects string but gets non-pointer type
1373	287	88.9%	Passing argument from incompatible pointer type
981	193	59.8%	Variable unused
709	170	52.6%	Function argument/initialization makes pointer from integer without a cast
507	105	32.5%	Variable used uninitialized
490	117	36.2%	Function argument/initialization makes integer from pointer without a cast
49	20	6.2%	Printf format expects integer but gets pointer type
24	10	3.1%	Cast to pointer from integer of different size

Total	Students	(%)	Error Type	Example Code
			Failing to use a double pointer	
969	261	80.8%	Using `char *`	`char *enemy_ptr = ...;`
172	100	31.0%	Using `int *`	`int *enemy_ptr = ...;`
			Missing & operator	
486	201	62.2%	On `arch_enemy`	`... = arch_enemy;`
252	114	35.3%	On `friends`	`... = friends;`
			Extraneous * operator	
127	81	25.1%	On `arch_enemy`	`... = *arch_enemy;`
89	58	18.0%	On `enemy_ptr`	`*enemy_ptr = ...;`
83	48	14.9%	On `friends_ptr`	`*friends_ptr = ...;`
58	80	24.8%	Combining & and *	`... = &*arch_enemy;`
			Failing to initialize variable	
38	25	7.7%	On `enemy_ptr`	
17	14	4.3%	On `friends_ptr`	
23	21	6.5%	Using `argv`	`enemy_ptr = &argv[2];`
16	15	4.6%	Using array notation	`enemy_ptr = arch_enemy[0];`

Table: Frequent errors in declaring and assigning pointers

In summary, the study found that even second-year students with prior programming experience struggle with core C pointer concepts. Misconceptions centered on confusing addresses with values, difficulty with dereferencing, and ineffective use of compiler messages. The findings emphasize the need for explicit teaching of memory models, targeted pointer exercises, and instruction on interpreting compiler output.

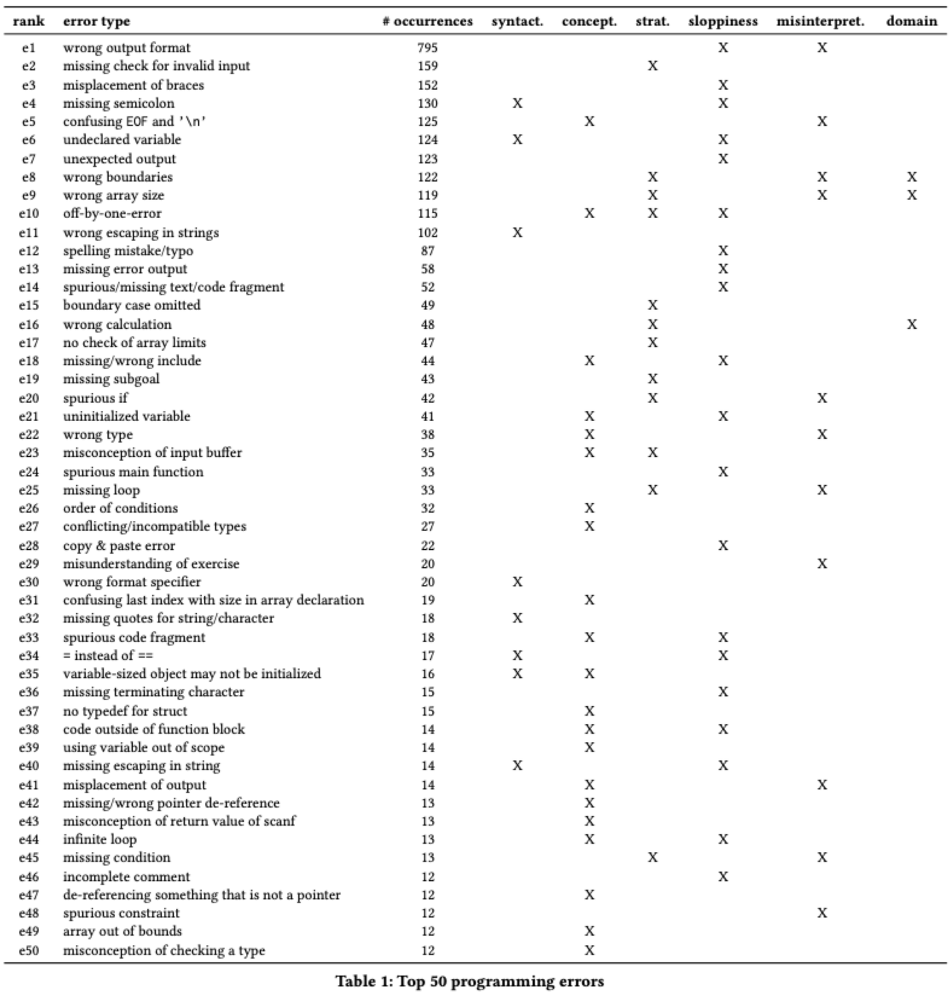
Paper 5: Sometimes It's Just Sloppiness – Studying Students' Programming Errors and Misconceptions (2020) [5]

This paper analyzed 12,371 incorrect submissions from 280 students in an introductory C programming course, collected via the SmartAPE online tool. Each incorrect program was manually classified into one of six categories: **syntactic**, **conceptual**, **strategic**, **sloppiness**, **misinterpretation**, and **domain knowledge**. Unlike compiler-only analysis, this approach captured both trivial slips and deeper misconceptions.

The results showed that nearly 17% of errors were due to sloppiness (typos, missing semicolons, careless oversights), but a significant portion were conceptual or strategic, reflecting misunderstandings of C semantics and problem-solving strategies. Strategic errors often involved missing boundary cases or failing to plan problem structure. Lastly, some of the conceptual errors have been highlighted below.

Common conceptual/semantic C errors (from Top 50 list):

- Using `=` instead of `==` in conditions.
- Assuming uninitialized variables default to zero.
- Off-by-one errors in loops and array indexing.
- Confusion between pointers and array elements (e.g., assigning value instead of address).
- Missing `&` in `scanf`, leading to crashes or garbage input.
- Misinterpreting operator precedence in expressions.
- Using wrong format specifiers in `printf`/`scanf`.
- Forgetting that arrays in C start at index 0.
- Returning values instead of modifying data through pointer parameters.



rank	error type	# occurrences	syntact.	concept.	strat.	sloppiness	misinterpret.	domain
e1	wrong output format	795				X	X	
e2	missing check for invalid input	159			X			
e3	misplacement of braces	152				X		
e4	missing semicolon	130	X			X		
e5	confusing EOF and '\n'	125		X			X	
e6	undeclared variable	124	X			X		
e7	unexpected output	123				X		
e8	wrong boundaries	122			X		X	X
e9	wrong array size	119			X		X	X
e10	off-by-one-error	115		X	X	X		
e11	wrong escaping in strings	102	X					
e12	spelling mistake/typo	87				X		
e13	missing error output	58				X		
e14	spurious/missing text/code fragment	52				X		
e15	boundary case omitted	49			X			
e16	wrong calculation	48			X			X
e17	no check of array limits	47			X			
e18	missing/wrong include	44		X		X		
e19	missing subgoal	43			X			
e20	spurious if	42			X		X	
e21	uninitialized variable	41		X		X		
e22	wrong type	38		X			X	
e23	misconception of input buffer	35		X	X			
e24	spurious main function	33				X		
e25	missing loop	33			X		X	
e26	order of conditions	32		X				
e27	conflicting/incompatible types	27		X				
e28	copy & paste error	22				X		
e29	misunderstanding of exercise	20					X	
e30	wrong format specifier	20	X					
e31	confusing last index with size in array declaration	19		X				
e32	missing quotes for string/character	18	X					
e33	spurious code fragment	18		X		X		
e34	= instead of ==	17	X			X		
e35	variable-sized object may not be initialized	16	X	X				
e36	missing terminating character	15				X		
e37	no typedef for struct	15		X				
e38	code outside of function block	14		X		X		
e39	using variable out of scope	14		X				
e40	missing escaping in string	14	X			X		
e41	misplacement of output	14		X			X	
e42	missing/wrong pointer de-reference	13		X				
e43	misconception of return value of scanf	13		X				
e44	infinite loop	13		X		X		
e45	missing condition	13			X		X	
e46	incomplete comment	12				X		
e47	de-referencing something that is not a pointer	12		X				
e48	spurious constraint	12					X	
e49	array out of bounds	12		X				
e50	misconception of checking a type	12		X				

Table 1: Top 50 programming errors

Figure: Extract from the Top 50 student errors in C (adapted from Paper 4).

Overall, the study stressed the importance of distinguishing superficial mistakes from fundamental misconceptions when providing feedback.

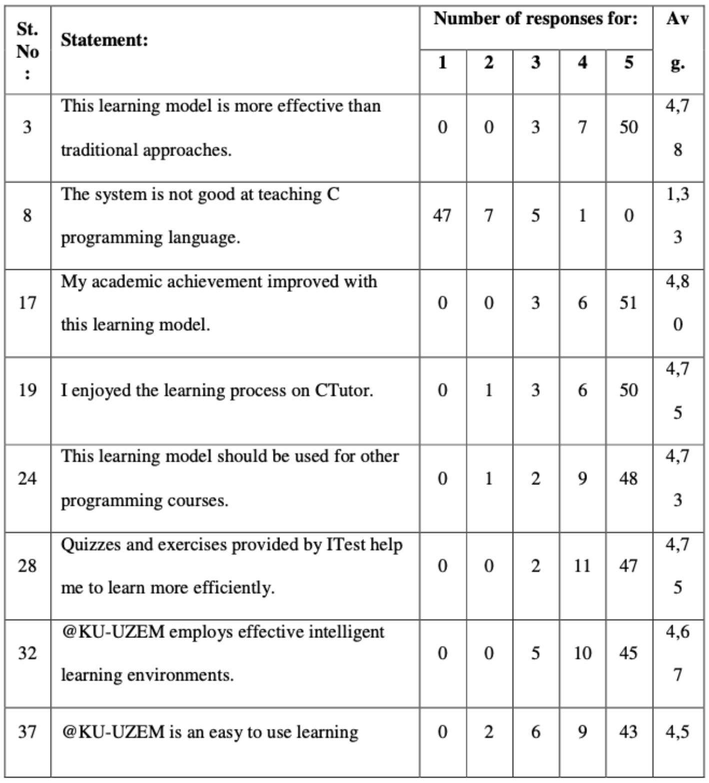
Papers on Interventions:

Intelligent Learning Environments within Blended Learning for Ensuring Effective C Programming Course (2012) [6]

This paper presents a blended learning approach that combines face-to-face instruction with e-learning, enhanced through intelligent learning environments integrated into the *@KU-UZEM* Learning Management System. Two main tools were developed: **CTutor**, a problem-solving environment that evaluates students' knowledge levels, provides feedback, and helps overcome misconceptions in C programming; and **ITest**, an adaptive assessment system that generates quizzes according to each student's learning level.

The model was applied to the *C Programming* course at Afyon Kocatepe University over two academic terms, involving 120 students divided into experimental and control groups. The experimental group used the intelligent learning environments along with traditional classes, while the control group received only face-to-face instruction.

Survey Results:
A post-course survey of 60 students assessed their acceptance of the blended model and attitude toward the intelligent environments. The majority of students expressed strong satisfaction: over 90% agreed that the learning model was more effective than traditional approaches, 88% found CTutor particularly helpful, and 87% were satisfied with the e-learning activities overall. Students also reported improved self-confidence and motivation to study.

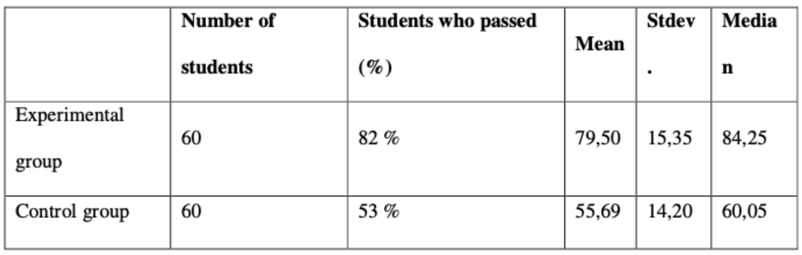


St. No:	Statement:	Number of responses for:					Avg.
		1	2	3	4	5	
3	This learning model is more effective than traditional approaches.	0	0	3	7	50	4,78
8	The system is not good at teaching C programming language.	47	7	5	1	0	1,33
17	My academic achievement improved with this learning model.	0	0	3	6	51	4,80
19	I enjoyed the learning process on CTutor.	0	1	3	6	50	4,75
24	This learning model should be used for other programming courses.	0	1	2	9	48	4,73
28	Quizzes and exercises provided by ITest help me to learn more efficiently.	0	0	2	11	47	4,75
32	@KU-UZEM employs effective intelligent learning environments.	0	0	5	10	45	4,67
37	@KU-UZEM is an easy to use learning	0	2	6	9	43	4,5

Experimental Comparison:
A comparative evaluation was conducted between the experimental and control groups. Results

showed that students using the intelligent learning environments performed substantially better: **82%** of students in the experimental group passed the course, compared to only **53%** in the control group. The mean term grade for the experimental group was **79.5**, while the control group averaged **55.7**. Statistical analysis using an independent-samples t-test confirmed a significant difference between the two groups at a 95% confidence level.

Table 5. Comparison of the results between experimental and control groups



	Number of students	Students who passed (%)	Mean	Stdev.	Median
Experimental group	60	82 %	79,50	15,35	84,25
Control group	60	53 %	55,69	14,20	60,05

Limitations/Criticisms:

- The study was conducted at a single university and within one course, limiting the generalizability of its findings.
- The evaluation period covered only two academic terms, without long-term observation to measure sustained learning outcomes.
- Student feedback relied primarily on self-reported survey data, which may introduce subjective bias.
- The system architecture, involving CTutor, ITest, and multiple LMS modules, is considerably more complex than traditional instruction or standard program tracing, requiring additional technical setup, integration, and maintenance effort.

Classroom practice for understanding pointers using learning support system for visualizing memory image and target domain world [7]

This paper investigates the use of the TEDViT visualization system to improve learners' understanding of pointers in C programming. The motivation for the study stems from the persistent difficulty novice programmers experience when learning pointer concepts, largely because existing visualization tools tend to obscure the actual memory locations and values of variables. TEDViT addresses this issue by providing simultaneous and synchronized visualizations of three key elements: the program code, the memory image displaying concrete variable states, and the target domain world representing logical data structures.

The research involved fifteen software engineers with limited experience in C programming who attended four 90-minute training sessions organized by the Hamamatsu Embedded Programming Technology Consortium (HEPT). Each session integrated lectures, TEDViT-based observations, coding exercises, and instructor explanations, allowing participants to explore relationships among pointers, variables, and memory models.

Questionnaire results indicated high participant satisfaction and suggested that TEDViT enhanced comprehension of pointer behavior while helping to reduce unevenness in understanding among learners, thereby supporting more effective class management. However, the study's limitations include a small participant sample, the absence of objective performance tests, and reliance solely on self-reported data. Additionally, since participants were professional engineers rather than students, the findings may not fully generalize to typical educational contexts. Despite these constraints, the authors conclude that TEDViT has significant potential as a visualization-based support tool for programming education, meriting further study through continued classroom applications.

Pedagogy of Teaching Pointers in the C Programming Language using Graph Transformations [8]

This paper proposes a new visual pedagogy for teaching the concept of pointers in the C programming language. The motivation stems from the persistent difficulty that students encounter in understanding pointer manipulation. The authors argue that visual learners benefit from graphical and dynamic representations rather than textual or static explanations, leading to the use of graph transformation systems as a pedagogical tool.

The study involved undergraduate students enrolled in the *Introduction to C Programming* course at the University of New Haven. The experimental group (Spring 2023) was taught pointers using visual simulations created with the GROOVE graph transformation tool, while the control groups (Spring 2022 and Fall 2022) were taught using traditional textbook-based methods.

In the experimental setup, the GROOVE tool was employed to model and simulate pointer operations, using type graphs, transformation rules, and visual animations to represent relationships between pointers, addresses, and data objects. The instruction included slides, in-class demonstrations, and practice exercises that visually reinforced pointer behavior.

The **type graph** defined in the study models basic C pointer structures by representing how pointers relate to memory addresses and objects such as *int*, *char*, and arrays. Each address node represents a memory cell, connected by `succ` edges to indicate consecutiveness. Pointers use `ref` edges to reference addresses, while addresses use `cont` edges to contain objects. Constraints on edges, such as ensuring every array has a single `fst` edge to its first pointer and each address having at most one successor, ensure the well-formedness of the model. This explicit modelling of addresses allows the representation of operations such as dereferencing (*) and address-of (&) and the detection of errors such as dangling or null pointers.

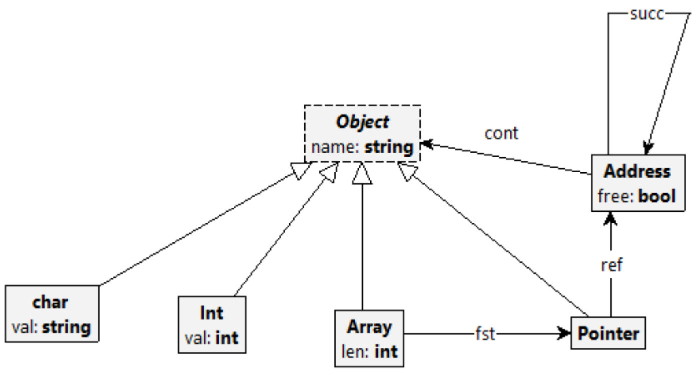
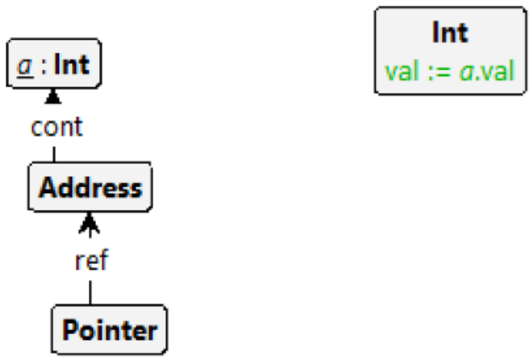


Figure 1: Type graph for pointer structures

Graph transformation rules encode the basic pointer operations in C. Each rule is represented as a combination of pre- and post-conditions integrated into a single graph with color-coded elements showing which parts are preserved, deleted, created, or forbidden. Examples include creating new pointers or integer objects, assigning addresses between pointers, or dereferencing pointer values. For instance, the rule that copies the value of the `int` referred to by a pointer into another object models the C statement `s = *pt;`.



(a) Copy value to int object

Example graph transformation rule modelling the statement `s = *pt;`

The results were evaluated through comparative test performance and survey responses. Using a value-added measure (VAM) analysis, the Spring 2023 class showed improved understanding and smaller negative VAM scores relative to earlier cohorts. Survey feedback indicated that students found the graphical approach helpful for visualizing memory and pointer relationships.

However, the authors acknowledge limitations in the study's statistical validity due to differences in student aptitude across cohorts and the small sample size. Moreover, the experiment only covered a subset of pointer operations and was limited to revision sessions rather than full-course integration. Future work aims to extend the rule set to include memory management functions, function calls, and distinctions between stack and heap memory, as well as to provide students with hands-on use of the GROOVE tool.

Where do we go from here?

Based on the kinds of errors and misconceptions identified in the earlier papers (especially around pointers, addresses, and memory behaviour), it's clear that many students still lack an accurate mental model of how C actually executes. Most existing interventions, like CTutor, TEDViT, or the GROOVE-based visualizations, do help with engagement and conceptual understanding, but I think they have limited semantic coverage or rely on complex setups that are difficult to replicate in a classroom.

Since the idea is to bridge the gap between compiler-level behaviour and what students imagine happens when they run their programs, we could start with defining a simple set of execution semantics for a core subset of C—covering expressions, assignments, pointers, arrays, and function calls. We could either design these semantics ourselves or adapt them from existing verification tools such as CompCert, Cerberus, or the K Framework.

Then we could implement a tool that runs programs step by step and shows corresponding state changes, as done with SimpliPy. Finally, we plan to test this system in a controlled classroom setting, comparing how students using the notional machine perform against those learning with traditional methods. Along with performance data, we'll collect feedback on usability and conceptual clarity to refine the model further.

What we want to study through this work is how such a system can improve students' conceptual grasp of low-level C execution. Specifically, we aim to observe whether the notional machine helps learners:

- form more accurate mental models of how pointers and memory interact,
- identify and correct misconceptions early (for example, about dereferencing or address-of operations),
- interpret compiler feedback in relation to runtime behavior, and
- reason about program state changes more systematically during debugging.

We also hope to gather insights into how learners use visual feedback when reasoning about C code—whether it supports long-term retention, improves debugging skills, or changes how they think about code correctness. By comparing learning outcomes between groups that use the notional machine and those that follow traditional methods, we can measure how far visual execution modeling contributes to genuine conceptual understanding rather than surface-level recall.

References:

[1] Common Logic Errors for Programming Learners

[2] Catalogs of C and Python Antipatterns by CS1 Students

[3] Common logic errors made by novice programmers

[4] Student Difficulties with Pointer Concepts in C

[5] Sometimes It's Just Sloppiness – Studying Students' Programming Errors and Misconceptions

[6] Intelligent Learning Environments within Blended Learning for Ensuring Effective C Programming Course

[7] Classroom practice for understanding pointers using learning support system for visualizing memory image and target domain world

[8] Pedagogy of Teaching Pointers in the C Programming Language using Graph Transformations