TARTARE: Automatic Generation of C Pointer Statements and Feedback

Géraldine Brieven
Université de Liège, Montefiore Institute
Liège, Belgium
gbrieven@uliege.be

Valentin Baum
Université de Liège, Montefiore Institute
Liège, Belgium
valentin.baum@student.uliege.be

Benoit Donnet
Université de Liège, Montefiore Institute
Liège, Belgium
benoit.donnet@uliege.be

ABSTRACT

This paper addresses the difficulties students face when learning and practicing pointers (i.e., variables storing the memory address of another variable as its value) in a computer programming class. To improve their understanding and practice, we have developed TARTARE, an automatic C pointer statement and feedback generator. By creating statements with automatic feedback, students are given the opportunity to practice at will, each time on a different instance. In addition, if the statement must be done remotely and accounts in the final grade, TARTARE discourages academic dishonesty since each student faces their own statement to solve.

This paper describes the techniques implemented in TARTARE, relying on a pattern template-based approach. The statement variety of TARTARE is evaluated. Finally, current limitations and further improvements are discussed. We believe our approach for TARTARE can be transposed for automatic exercises generation in various other fields.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Software and its engineering** → *Source code generation.*

KEYWORDS

C programming language, pointers, statement generation, TARTARE

ACM Reference Format:
Géraldine Brieven, Valentin Baum, and Benoit Donnet. 2024. TARTARE: Automatic Generation of C Pointer Statements and Feedback. In *Australian Computing Education Conference (ACE 2024), January 29-February 2, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3636243.3636264

1 INTRODUCTION

Each year, we organize a computer programming class (hereafter called CP class) in which students are exposed to specific C programming language concepts and algorithmic aspects. Mastering the C language requires a deep understanding of memory management, a key skill that every computer scientist must acquire [9]. For that purpose, one chapter of the course is dedicated to *pointers* (i.e., variables storing the memory address of another variable as its value). In regards to Goldman et al. [19] classification, pointers belongs to the "*Memory model, references, pointers*" class (abbreviated as MMR). Compared to the other key concepts in Computer Science, this topic appears among the most difficult ones for students [19], which is confirmed by other studies [1, 12, 30] as well as through our own teaching experience. Therefore, in our class, in addition to traditional theoretical and exercise sessions, students have the opportunity to practice pointers through an online homework that is automatically corrected with feedback and feedforward [10, 28].

However, it often appears that the performances at the exam – which contains a question identical, in its overall shape and concept, to the pointer homework – do not match those of the remote homework. Fig. 1 illustrates that observation by providing the correlation between students' performance in the remote homework and the exam for academic years 2019 and 2022. We have identified three potential explanations for this discrepancy: (*i*) some students miss practice because they do not have enough exercise instances to train on (see Fig. 1, bottom left square – students who have failed in both the homework and the exam would probably perform better with more exercises and feedback); (*ii*) some students may cheat [22] or collaborate to find solutions to their remote homework (see Fig. 1, bottom right square – students who succeeded in their homework but failed at the exam); finally, (*iii*) some students may rely on the help of the course material or even external tools like CHATGPT [29], without fully grasping the concepts (see Fig. 1, bottom right square).

To address those concerns, we believe each student must have the possibility to practice as often as wanted, each of them solving a new C pointer statement instance. However, manually creating such statements, correcting them, and quickly providing feedback does not scale, particularly for a large audience, as often met in an introductory CP class. To address those issues, we wanted to develop a tool for automatically generating C pointer statements and providing personalized feedback. More precisely, our goals are:

- Goal 1: Providing students with the ability to practise as much as they want by automatically generating C pointer statements.
- Goal 2: Supporting student improvement through feedback provided after an automatic correction of students' answers. This feedback must help students to identify and understand their mistakes, so that they are not repeated.
- Goal 3: Reducing the possibility of cheating by giving each student a different set of statements of the same difficulty level.
- Goal 4: Reducing the workload of the supervisors by defining a scalable system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACE 2024, January 29-February 2, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1619-5/24/01…$15.00
https://doi.org/10.1145/3636243.3636264

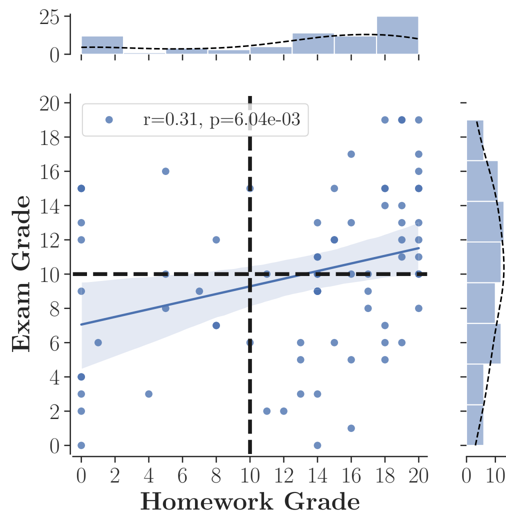


Figure 1: Correlation between results for online homework and exam for academic years 2019 and 2022. Students who did not participate in one or both activities are not accounted in this graph. Years 2020 and 2021 are not reported since exams were organized remotely due to CoVID-19 pandemic [13] and results would not be representative. A student succeeds a C pointer statement if the mark is greater or equal to 10. $N_{2019} = 44$ **and** $N_{2022} = 32$.

These goals are achievable if statements always follow a predefined schema so that it can then be modeled and implemented. Furthermore, this modeling can be used to anticipate students' possible answers and match them to pre-defined misconceptions, enabling personalized feedback.

In this paper we instantiate this process for C pointer statements. In particular, an analysis of the composition of past C pointer statements in our CP class is conducted in order to derive patterns from which new instances can be automatically generated. Those patterns are then used as input to TARTARE (au**T**om**A**tic C pointe**R** statemen**T** gener**A**to**R** with f**E**edback), our automatic C pointer statement generator. TARTARE can be parameterized to constrain the C pointer statement generation. Besides statement generation, in order to provide personalized feedback to students, a collection of possible erroneous answers is computed by TARTARE, on the basis of common predefined pointers misconceptions.

We assess TARTARE's quality along two dimensions: its ability to generate unique C pointer statements and its ability to generate a variety of expressions within a C pointer statement. Finally, the paper also shows that the generated C pointer statements are relevant, with a level of difficulty requiring a deep understanding of pointers.

The remainder of this paper is organized as follows: Sec. 2 models the C pointer statements in our programming homework and final exam; Sec. 3 introduces TARTARE; Sec. 4 evaluates its performance; Sec. 5 discusses our generated patterns relevance; Sec. 6 positions this work with respect to the state of the art; Sec. 7 discusses current TARTARE's limits and various directions for future works; finally, Sec. 8 concludes this paper by summarizing its main achievements.

2 C POINTER STATEMENT FRAMEWORK

This section describes the general framework used to define a C pointer statement in our CP class. The statement framework comprises three parts: (*i*) the variable declarations (Sec. 2.1), (*ii*) the memory state (Sec. 2.2), and (*iii*) the list of expressions to evaluate (Sec. 2.3). A complete example of a C pointer statement is provided in Appendix A. It will be referenced throughout this section.

Given a C pointer statement, the students must evaluate each expression (i.e., rvalue-expression) using the memory state and variable declarations. If an expression evaluation leads to an address that is outside the range of addresses provided by the memory state, students must respond that the expression evaluation returns a segmentation fault (symbolized by `SF`). It is assumed that each expression is evaluated with respect to the initial given memory state, to avoid the waterfall effect in case of an error.

Students have been exposed to such statements between 2016 and 2022, through online homework and exams. A total of 24 statements have been manually created during this period. In this section, all those past statements are analyzed (see Fig. 2) and their general shape is described.

2.1 Variable Declaration

First, the variable declarations are provided to students. The general shape of such declarations is illustrated in Listing 1.

Listing 1: General view of variables declaration.

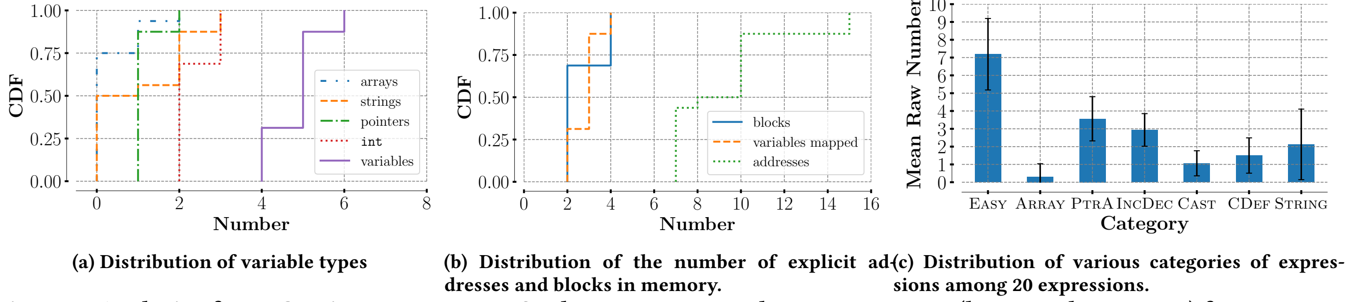
```
1 #include<stdio.h>
2 int var1;
3 int *var2 = &var1;
4 int var3[] = {...};
5 char *var4 = "...";
```

These declarations introduce all the required variables for the statement, some of them being initialized. Only integer data types (i.e., `int` and all its variations) and arrays (of `int` or `char`) are considered. The declarations also provides the `#include` directive for allowing input/output (typically, a call to `printf()`). In the example in Appendix A, `w` and `v` are declared as integer variables, `b` is declared as a pointer containing the address of variable `v`, and `a` and `l` are strings initialized to "Invariant" and "ariant" respectively.

More generally, Fig. 2a gives some insight about the amount of variables and the distribution of their types over the past statements. It shows that about 56% of statements introduced five variables (solid purple line). Among them, the integers are the most frequent ones (dotted red line rightmost). It varies from two to four occurrences. On the opposite, arrays were declared less frequently (only 25% of the past statements introduced at least one array).

2.2 Memory State

Fig. 3 outlines the general representation of a memory state in our C pointer statement. It covers a specific range of addresses ([lower bound, upper bound] – [20, (1008+3)] in Appendix A). Not all addresses in the range are necessarily present. In such a situation, the memory is divided in several blocks. For instance, in Appendix A, there are two blocks, [20, 36+3] and [1000, 1008+3]. Variables (first column) are mapped to addresses (second column – they are represented in decimal for simplicity), and variables may have a defined value (third column). If the value at a given address



(a) Distribution of variable types

(b) Distribution of the number of explicit addresses and blocks in memory.

(c) Distribution of various categories of expressions among 20 expressions.

Figure 2: Analysis of past C pointer statements. Students were exposed to 24 statements (homework or exams) from 2016 to 2022.

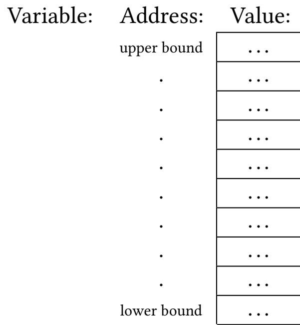


Figure 3: General view of the memory state.

is undefined, it is represented by "??" (see address 1008 in Appendix A). Usually, we store values on 32 bits, and addresses are aligned to the word size, meaning that they are multiples of four.

All the declared variables (see Sec. 2.1) may not necessarily be mapped to memory. Typically, to evaluate the expressions, if simple variables (i.e., `int`) are initialized upon declaration, only their value should matter, not their address. On the contrary, if a variable is not initialized upon declaration, it means both its address and value have to be considered when evaluating the expressions. In such a situation, the variable is mapped to the memory and some random value is assigned to it (see for example `w`, in Appendix A, that is mapped to address 1004 with value 28). Finally, all pointers are initialized at declaration time (`b` receives the address of `v` in Appendix A).

It is worth noticing that this memory shape does not reflect the actual memory organization, as it does not show the stack, the heap, BSS, and so on, and does not position variables in a realistic way. The objective here is not to be realistic, but rather to allow students to improve their understanding of pointers and memory arithmetic by relying on a simplified memory model.

Fig. 2b analyzes the memory framework from previous C pointer statements. In most of the cases (nearly 75% of the cases), the memory is divided in two blocks (the maximum being four – see the blue line). The number of addresses explicitly shown oscillates from seven (nearly half of the case) to fifteen (see the green dotted line). Finally, two to four variables that have been declared are explicitely mapped to a memory address (see the orange dotted line).

2.3 Expressions to Evaluate

The last part of a C pointer statement exhibits the different expressions students have to evaluate according to the variable declaration and the memory state. A statement is always made of twenty rvalue-expressions (see Appendix A).[1] The objective of those expressions is to practice pointers and memory arithmetic, even through expression that do not follow good programming practice rules (see Expression 2 in Appendix A).

We model each expression as belonging to a given *category* (see Sec. 2.3.1). Within each category, we have identified *patterns*. A pattern is composed of variables (with a specific type), specific kinds of operations and casting. Patterns allow to generalize all the expressions that have been defined in the past.

2.3.1 Categories. Seven categories emerge from the analysis of the previous C pointer statements:

- EASY: Refers to basic notions of variable, address, and value of a variable. See for example Expression 1 in Appendix A.
- ARRAY: Refers to continuous addresses in memory. As mentioned in Sec. 2.1, arrays are unusual in our statements and Appendix A does not exhibit an expression belonging to this category.
- PTRA: Refers to operations between an address and a value or another address. See for instance Expression 10 in Appendix A.
- INCDEC: Refers to the manipulation of the increment and decrement operators. See for instance Expression 7 in Appendix A.
- CAST: Refers to the transformation of an address of one type to an address of another type (i.e., casting operation). See, for instance, Expression 14 in Appendix A.
- CDEF: Refers to dereferencing addresses that are obtained from other expressions. See for instance Expression 6 in Appendix A.
- STRING: Refers to character access (from array) and difference between characters. See for instance Expression 16 in Appendix A.

Fig. 2c illustrates the average frequency (with standard deviation) of the categories across all the previous C pointer statements.

[1] In the following text, we will use "expression" and "rvalue-expression" interchangeably.

Table 1: Notations supporting the expressions.

(Pattern) Component	Meaning
data	simple variable
ptr	pointer variable
tab	array variable
cTab	string variable
let	char variable
val	integer value
unOp	unary operation (++ or --)
binOp	binary operation (+ or -)
*(type *)*	casting ((short *) or (long *))

From one statement to another, on average, seven (±2) expressions were part of the EASY category. For the other categories, that average ranges between one and four, except for the ARRAY category. This last type of expression has been introduced later in our C pointer statements, explaining why it is less frequent when all the statements (from 2016 to 2022) are taken into account.

2.3.2 Patterns. Within each category, one can model the expressions through *patterns*. Since only the variable type matters to build expressions, the various variable names are represented through notations introduced in Table 1. Similarly, some notations are also defined to represent various kinds of operations and casting.

It is worth noting that some variables must have specific values for some expressions to make sense when they are evaluated. This is discussed further in Sec. 3 where pattern components replacement is described.

Table 2 maps the expression patterns from the previous C pointer statements to the different categories. The patterns in bold are those from which the 20 expressions in Appendix A were derived.

The patterns described in Table 2 cover a subset of the C programming language. This subset excludes exotic expressions such as `x++ + ++x` whose evaluation is not defined by the standard. Further, the restriction also ensures that the produced expressions are valid in C and can be compiled without any warning.

Table 2: Classification of patterns. Each pattern is uniquely identified through its category and a number. See Table 1 for the notations meaning in the various patterns. Patterns in bold are present in the C pointer statement in Appendix A.

Expression Category	Patterns
EASY	**(E.1) data**
	(E.2) *(int *)data
	(E.3) &data
	(E.4) &*ptr
	(E.5) *(int *)*ptr
ARRAY	(AR.1) tab[val]
	(AR.2) ptr[data]
	(AR.3) ptr[val] *binOp* ptr[val]
	(AR.4) ptr[tab[val] *binOp* tab[val]]
	(AR.5) ptr[ptr[val] *binOp* val]
PTRA	**(PA.1) &data *binOp* ptr**
	(PA.2) &data *binOp* val
	(PA.3) ptr *binOp* data
	(PA.4) (int*)data *binOp* val
	(PA.5) ptr *unOp binOp unOp* data
INCDEC	(ID.1) *unOp* *ptr
	(ID.2) **unOp* ptr
	(ID.3) (*ptr) *unOp*
	(ID.4) *ptr *unOp*
	(ID.5) *(*unOp* ptr)
CAST	(C.1) data *binOp (type *)* ptr
	(C.2) *(type *)* &data *binOp* val
	(C.3) *(type *)* ptr *binOp* val
	(C.4) (int *)ptr *binOp* ptr[val]
CDEF	**(CD.1) *unOp* ptr *binOp* *(&data *binOp* val)**
	(CD.2) *(ptr= ptr)
	(CD.3) *(&data *binOp* data)
	(CD.4) *(ptr *binOp* val)
	(CD.5) &*(ptr *binOp* val)
STRING	(S.1) *cTab
	(S.2) cTab *binOp* cTab *binOp*
	(S.3) printf("%s", cTab)
	(S.4) cTab[let - let]
	(S.5) cTab[val] - cTab[val]

3 TARTARE

This section introduces TARTARE (au**T**om**A**tic C pointe**R** state-men**T** gener**A**to**R** with f**E**edback). As illustrated in Fig. 4, TARTARE is made up of two main components: the statement generation (Sec. 3.1) in charge of creating the C pointer statement (see an example in Appendix A) and the feedback generation (Sec. 3.2) responsible for identifying a set of possible student answers and mapping them to some predefined pointers misconception.

3.1 Statement Generation

The statement generation is depicted in Fig. 4 (top rectangle). It relies on patterns defined beforehand (see Table 2).

Before running the statement generator, the educational team can customize several parameters: (*i*) the number of statements (one per student typically); (*ii*) the variable distribution (set to two integers, one pointer, and two strings by default, as suggested by Fig. 2a); (*iii*) the number of memory blocks (set to two by default, as suggested by Fig. 2b); (*vi*) the number of visible addresses (set to ten by default, as suggested by Fig. 2b); (*v*) the memory interval (set to [4;2000] by default); (*vi*) the number of expressions per statement (N) (set to 20 by default); (*vii*) the category distribution (set to 35% of EASY expressions, 15% of PTRA expressions, 15% of INCDEC expressions, 5% of CAST expressions, 15% of CDEF expressions and 15% of STRING expressions by default, in accordance with the means given by Fig. 2c.); (*viii*) the maximum number of Segmentation Fault over the N expressions (set to one by default). The default values of those parameters are determined based on the analysis of the previous statements.

Once those parameters have been fixed, statements can be generated, as illustrated in the top rectangle of Fig. 4. For a given statement, first, variables are declared (and initialized), with respect to the variable distribution parameter. Next, considering those variables, a memory state is generated based on the number of memory blocks, the number of visible addresses, and the memory interval that are specified through the parameters. After this, N different

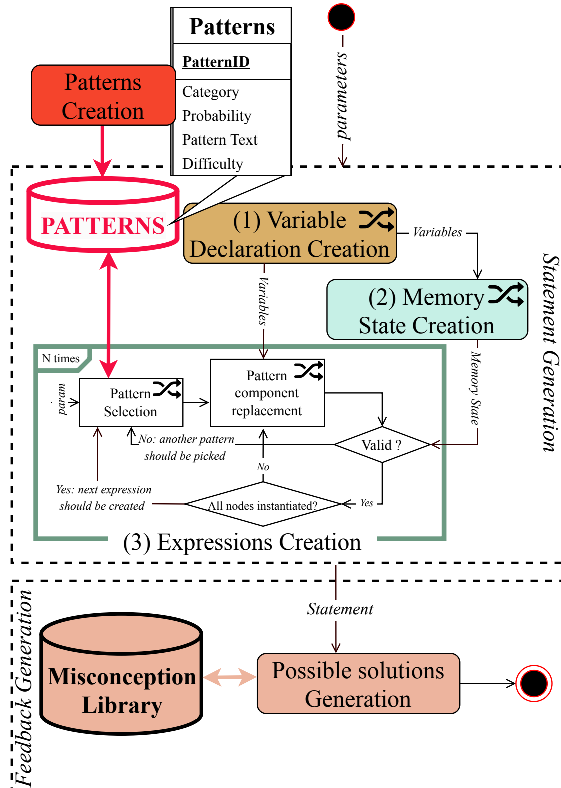


Figure 4: Tartare high level overview. It is made of two parts: (*i*) the statement generation (top rectangle) and (*ii*) the feedback generation (bottom rectangle).

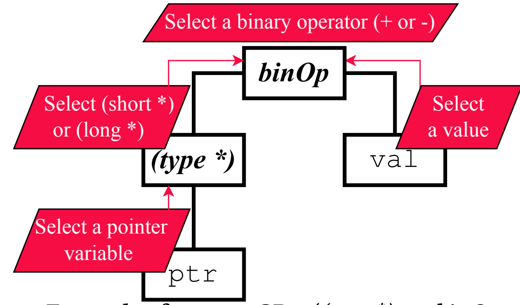


Figure 5: Example of pattern CD.3 ((*type* *) `ptr` *binOp* `val`) represented as a tree (in black) and the selection process (in red).

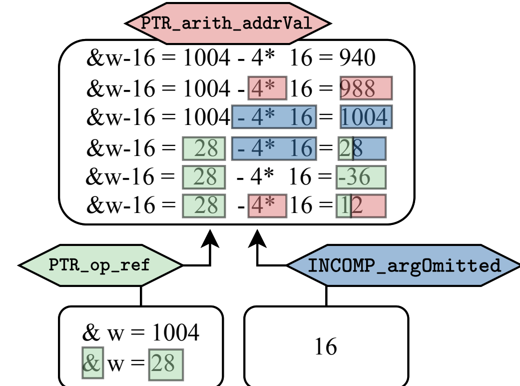


Figure 6: Example of feedback production based on the simulation of errors on the different intermediate terms. The example is built based on Expression 11 in Appendix A.

expressions can be created. The pattern selection is controlled by the category distribution. Once a pattern is selected, the pattern components (listed in Table 1) must be instantiated. To do so, the pattern is represented through a tree, as depicted in Fig. 5. The leaves are variables while the internal nodes either stand for operations or casting. If a variable needs to be selected, it must be one of the variables declared in the first step. For all nodes, if the dereferencing operator applies (this information is attached to the node) and a Segmentation Fault already occurs once as expected answer, the value of that node must be within the memory range defined in the previous step. If those restrictions make the selection impossible, a new pattern is selected and the process is repeated until a feasible expression is obtained. Fig. 5 illustrates the pattern component selection, with respect to a given pattern tree.

3.2 Feedback Generation

Once the N expressions have been generated, a corresponding feedback dictionary can be produced. The feedback consists of the student's possible answers, including both correct and incorrect ones, with potential feedforward to the course material.

The most intuitive way to generate a correct answer would be to run some C piece of code that would evaluate the different expressions. However, this approach has two main drawbacks:

(1) We do not have direct control over the allocated memory addresses. This poses a challenge in assigning values to the memory. All values in the memory are of type `int` but some values may need to be casted into addresses. We need to modify these values to correspond to the allocated addresses, rather than the value specified in the statement. From a student perspective, it is then difficult to simply print the expressions results as they would have to replicate the memory in some way beforehand.
(2) We cannot easily simulate students' reasoning based on some given pointers misconceptions to map some wrong solutions to a suitable feedback.

Those limitations are overcome by reusing the tree structure of each pattern that supported the expressions generation.

To find the correct answer, we run through the whole tree and compute the correct value for each node, starting from the leaves. To compute a node value, four pieces of information are required: (*i*) the result values the children are carrying; (*ii*) the current address; (*iii*) the type (to know if it can be dereferenced); (*iv*) the address size (as it could have been altered by some previous casting).

Besides this, to identify the common wrong solutions, we follow a similar process, except that miscomputations are simulated in such a way that some nodes carry wrong values. Each time a miscomputation is performed, the corresponding misconception is attached to the resulting wrong value, as illustrated in Fig. 6. Finally, based on those misconceptions, an appropriate text feedback can be built for each final value, depending on the miscomputations

```
"11": {
   "940": "The answer is correct.",
   "988": "You made one mistake. You should check the
   formula to compute the difference between an address
   and a value.",
   "1004": "You made one mistake. It seems that you forgot
   to compute the difference.",
   "28": "You made one mistake. The value of w shouldn't
   be involved here, we take the address.",
   "12": "You made two mistakes. (1) You seem confused
   with the referencing operator(&). It means that you
   take the address of the variable. (2) You should
   check the formula to compute the difference between
   an address and a value.",
   "-36": "You made one mistake. You seem confused
   with the referencing operator(&). It means that you
   take the address of the variable.",
  "default":"Your answer is incorrect."
}
```

Figure 7: Example of a generated feedback for Expression 11 in Appendix A. The produced dictionary mapped potential answers of students with the associated feedback.

Table 3: Examples of errors related to pointers management.

Error Code	Feedback Message
`PTR_numValChar`	A character holds an ASCII value. For example, the ASCII value of 'A' is 65.
`PTR_arith_addrVal`	Check the formula to compute the difference between an address and a value.
`PTR_op_ref`	You seem confused with the referencing operator (&). It means that you take the address of the variable.
`PTR_op_deref`	You seem confused with the dereferencing (*). It means that you take the value at the address stored in the variable.

they cumulated (see Fig. 7). This output can be interpreted by our learning platform [10], which supports automatic correction and feedback. For "simple" questions, the tool simply tries to match the student's answer with one of the answers identified by the generator. If there is a match, the appropriate feedback is sent to the student. Otherwise, a standard feedback message is addressed to the student, informing them that their answer is incorrect.

To implement such a process, a misconception library needs to be defined beforehand. Such a library contains the predefined feedback texts that are associated to common errors. Typical errors made by students in past C pointer statements (i.e., during online homework and exams from 2016 to 2022) are used to populate this library. Some examples of errors are presented in Table 3.

To substantiate our library, Fig. 8 shows to which extend we could cover students' errors by simulating the misconceptions we have identified so far. We confront the resulting potential students' answers with incorrect answers students actually provided in the online homework during academic year 2022. The objective here is to see the proporition of mistakes Tartare could catch based on our misconception library. On Fig. 8, the more to the right the red curve, the more robust the misconception library. Fig. 8 illustrates that a majority of students' errors can be mapped with the errors stored in our misconception library. More precisely, we can see that, in 50% of the cases, the proportion of wrong answers that could be

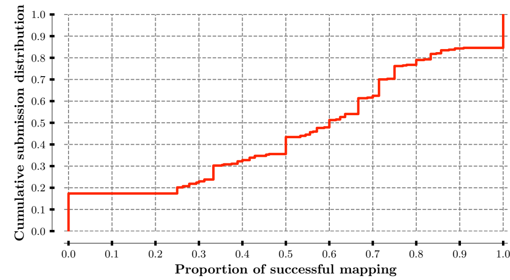


Figure 8: Misconception library robustness with respect to students incorrect answers (in the context of an online homework in academic year 2022).

mapped to misconception is higher than 60%. And for 15% of the submissions, all errors could be computed in advance.

4 TARTARE EVALUATION

This section demonstrates that Tartare fulfills the four goals stated in Sec. 1. We assess the diversity of statements (Goal 1 and Goal 3) and we discuss the feedback robustness (Goal 2) as well as the scalability (Goal 4) of Tartare. Regarding the feedback robustness, Tartare automates the mapping between misconception(s) and answer (for a given expression). In this way, more combinations of misconceptions can be handled (compared to manual simulations), leading to a longer list of possible answers with an appropriate feedback. As this list produced by Tartare at least includes answers that would have been manually computed, the resulting feedback accuracy reaches at least what we could get so far, based on a manual mapping (see Fig. 8). Then, for both the mapping (supporting the feedback) and the statement generation using Tartare, the supervisor workload significantly decreased, therefore meeting Goal 4. With Tartare, the only manual intervention consists in setting the parameters. Moreover, to generate 1,000 exercises (directly compiled in LaTeX, producing 1,000 PDF files), only 5 seconds are now required, while manually designing one C pointer statement took us, on average, four hours.

Regarding Goal 1 and Goal 3, C pointer statements produced by Tartare must respect the given setting while remaining sufficiently distinct from each other's. For that purpose, at each step of the generation process (see Fig. 4), various choices are kept random: (1) the relation between the variables and their initial values; (2) the bounds of the various memory blocks that are visible and the variables location in the memory; (3.1) the pattern selection within a category; (3.2) the component selection for a given pattern.

This section evaluates the Tartare randomness around those four aspects through 1,000 C pointer statements generated with parameter values as given in Table 4. In addition, Tartare has generated the corresponding 1,000 feedback dictionaries (see Sec. 3.2).

Fig. 9 shows the proportion of each pattern in a given category. The patterns distribution appears well-balanced, as expected. Only two patterns (CD.3 and CD.4) seem less represented, both belonging to the CDef category. It is likely because the addresses they need to access (`&data` *binOp* `val` and `ptr` *binOp* `val`) often falls outside the memory range of interest, leading to the selection of another pattern.

Table 4: TARTARE parameters values used for our evaluation.

	Parameter	Value	Meaning
Global	NBR_EXERCISES	1,000	Number of different statements to generate
	SIMILAR_EXERCISES	False	variable names, memory states, and expression selections differ
Variables	VARIABLE_REPARTITIONS	[3,2,1]	3 simple variables, 2 pointers, and 1 array
	VARIABLE_CHAR_REPARTITIONS	[2]	2 strings
Category	CATEGORY	[7,3,4,1,1,2,2]	7 expressions in EASY, 3 in INCDEC, 4 in PTRA, 1 in CAST, 1 in ARRAY, 2 in STRING, and 2 in CDEF
Memory	MAX_MEMORY_INTERVAL	[0,500]	lower and upper bounds for the memory range
	MEMORY_RANGE	1,000	memory range for each statement
	NBR_VISIBLE_ADDRESS	10	number of addresses that are visible within the memory range
	MAX_NBR_MEMORY_BLOCKS	4	the maximum number of distinct memory blocks inside the memory range

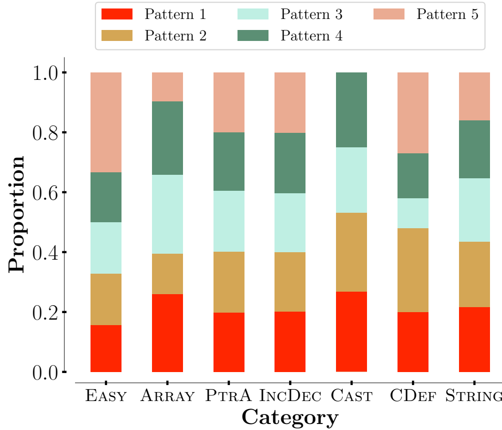


Figure 9: Distribution of patterns for each category. Each pattern is associated with the percentage of occurrences within the category.

Besides this, Fig. 10 reports how different the expressions emerging from the same pattern are. This metric is relevant as expressions are what students directly manipulate. To assess the difference between the expressions, rather than just considering their value, their whole intermediate results are taken into account. To do so, their corresponding tree structure is used. The values hold by each node are computed and compared. More precisely, we track both the memory address being accessed and the resulting value of each node. By aggregating these values for each node, starting from the leaf nodes, we create vectors representing the values obtained at different levels of the expression resolution. Two expressions are considered different if their vectors differ. This differentiation test was run on all the pairs of expressions derived from the same pattern, under three different conditions: (*i*) with variables and memory being fixed (yellow bar); (*ii*) with only variables fixed (red bar); (*iii*) without anything fixed (i.e., what is happening in practice, when students get their own statement to solve – green bar). By considering those different conditions, we can identify the impact of each step on the diversity of the resulting expressions. After running those tests, Fig. 10 shows that the expressions appear unique if they have been produced based on different memory configurations (and/or variables), except the three first ones belonging to the STRING category. Those three patterns only depend on variables (not constants) while only two different strings are declared. This explains why the diversity of expressions is extremely limited when the variable declaration is fixed. It is not true for patterns 4 and 5 since they also depend on constants (letters or value), which are independent from the declaration and memory state. Further, even if the variable declaration is different, the diversity of the three first STRING expressions remains limited. Considering pattern S.1 (`*cTab`), it suggests that lots of strings are initialized in such a way they start with the same letter. Looking at pattern S.2, it suggests that the substring of the first string that is declared sometimes starts at the same index (leading to the same addresses difference across the statements). With respect to pattern S.3 (`printf("%s", cTab)`), it suggests that the first string that is declared often contains the same number of letters. All of this gives us insight about how we could tune TARTARE for the future.

Next, coming back to the patterns of the other categories, when both the variables and the memory state are fixed, the proportion of unique expressions varies, depending on how many components need to be replaced in the pattern. The more components, the more unique expressions, which makes sense since each component requires a random choice to be made.

To summarize, those results demonstrate that students get statements that are sufficiently different from each other's. It means that, from one instance to another, they get a fresh training experience to enforce their learning (meeting Goal 1). Further, their statements also differ enough from the ones of the other students, preventing them from easily cheating (meeting Goal 3).

5 GENERATED PATTERNS RELEVANCE

In this section, we investigate how easy it is for students to get correct answers from external tools, such as CHATGPT. To do so, we have confronted several C pointer statements with CHATGPT version 3.5. From the 1,000 C pointer statements generated by TARTARE (see Sec. 4), we randomly picked 100 of them with their associated feedback dictionary. Those statements have been proposed, in a row, to CHATGPT between September 18th, 2023 and September 19th, 2023. Each statement was fully provided, in LaTeXformat, to CHATGPT. We ended up each statement with the following question:

> What is the evaluation of each of those expressions, in CSV format? Please, indicate the expression and your answer.

CHATGPT answers were then copied and saved in a CSV file (one file per statement to solve) for further analysis, i.e., confronting those answers with the corresponding feedback dictionary.

Fig. 11 illustrates the performance of CHATGPT on the 100 statements. For each statement, 20 expressions had to be evaluated. An

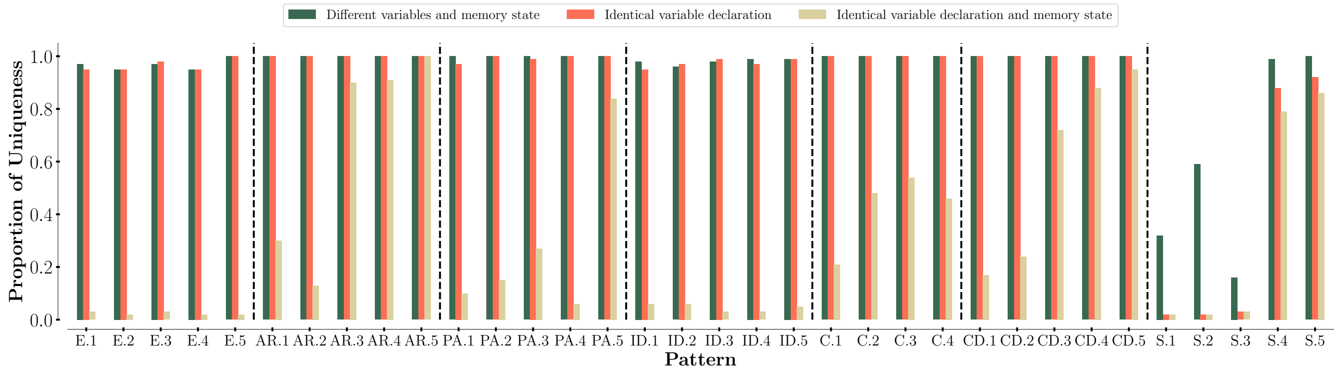


Figure 10: Proportion of unique expressions derived from a given pattern. Pattern identifiers on the X-Axis refer to those in Table 2.

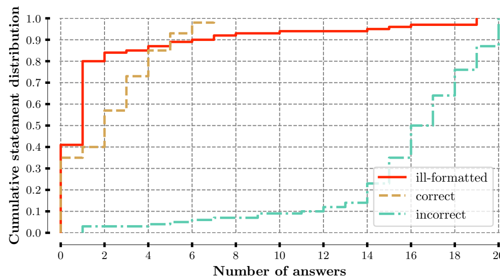


Figure 11: Performance of ChatGpt.

answer can be ill-formatted[2], correct, or incorrect. Fig 11 shows that, in 10% of the cases, ChatGpt could not provide any well-formatted answer. Next, for 60% of the statements, ChatGpt could only provide at most four correct answers out of 20. Among the incorrect ones, only some of them could be mapped to a pre-defined misconception, which shows how much unreliable ChatGpt is regarding our C pointer statement. However, it also shows that students cannot really take benefit from ChatGpt. Going further, we noticed that most of the correct answers given by ChatGpt are related to the Easy expressions, which is not much helpful for students.

Those results also highlight our C pointer statement generation and the fact that, to solve a statement, a student must have a deep understanding of theoretical concepts and must have trained with traditional exercises sessions or statements generated by Tartare.

6 RELATED WORK

With the growing number of students enrolled and the popularity of online courses, automated exercise generation and assessment has gained attention as a means of reducing instructor workload and offering additional opportunities for students to practice.

Different techniques for creating exercises have been presented in prior work: (*i*) Context-Free grammar [5, 11, 20, 44, 45]; (*ii*) **Templates**; (*iii*) Mutation [6, 39]; (*iv*) Construction from solution [3, 32]; (*v*) Natural language processing [2, 4, 8, 14, 26, 27, 31, 35, 40, 43, 46, 47]. This paper applies the Template approach through which patterns have been identified from previous statements to shape new instances. The same process was applied for other statement profiles, like identifying what a piece of code does [36], shaping a loop flow [16], writing SQL queries [24], or solving equations [41].

Then, different strategies to provide meaningful feedback have been introduced: (*i*) **Test-based feedback**; (*ii*) Path construction [7, 34, 37]; (*iii*) **Transformation model**; (*iv*) Peer-feedback [17]. In our work, Test-based feedback and Transformation model are combined. It compares the student's response to the expected, similar to what the first wave of automated grading tools supported. [15, 18, 21, 23]. To go further, we catch students' misconception(s) in case of error. To do so, transformation rules simulating typical errors are applied when a solution is computed to anticipate multiple students' solutions [38, 42]. The limitation of this method is that it fails if the student is too far from the solution [33].

Typical errors made by students in practice are the cornerstone of our misconception library. Some of those errors can be mapped with some C pointer concepts introduced by Craig and Petersen [12], including getting an address, dereferencing pointer, managing pointers arithmetic, or considering the array name as the address of the first element of the array. They also studied students' common mistakes in regards to those concepts. However, their students were asked to define their own expressions while our students are expected to assess given expressions. Therefore, some of their errors match with ours (e.g., Incorrect Pointer Arithmetic or Dereferencing pointer gives the value at the address stored in the pointer) while others do not (eg: '&' applied to address, undeclared variable, or a pointer is assigned to an integer). More generally, other studies identified common mistakes related to the "*Memory model, references, pointers*" class [1, 25]. Like Craig and Petersen [12], they focus more on how students manipulate pointers to solve a given problem rather than purely considering how students can evaluate some given pointers expressions. Because of that, their error taxonomy is more oriented towards dynamic allocation.

7 CURRENT LIMITS AND FURTHER WORK

By relying on a pattern-based approach, expression profiles are limited compared to what we could get using other techniques

[2] ChatGpt typically replied with text describing how the answer could be obtained rather than the actual value. For instance, with expression `*tab`, ChatGpt replies with `[value of s]`, which does not make sense and cannot be automatically evaluated.

such as a context-free grammar or a machine learning approach. However, the disadvantage of those techniques is that one loses control over the generated expressions. Therefore, it requires careful checking of each expression to ensure they are correct and relevant. For example, an expression that might result from a context-free grammar is `* (int *) * (int *) * (int *) x` while we would expect `*** (int ***) x`. Machine Learning techniques would likely lead to expressions that are even more inaccurate. Some alternative would be to use those techniques only to draw new patterns (rather than expressions directly), so that manual reviewing would be feasible as the pattern creation occurs only once. Moreover, by sticking to the pattern-based approach, expressions distribution can be more balanced since we can force each pattern to be used only once per C pointer statement.

Although pattern-based techniques provide more control, currently, Tartare does not come with a mechanism to fully make sure that the generated expressions are syntactically correct. However, given we are working on a subset of the C programming language, we believe the produced expressions could be included in a C-program and compiled. In case of incorrectness, warnings should be triggered and the expression(s) of concern should be replaced.

Finally, regarding the automatic mapping Tartare is making to support automatic feedback, it could still be improved by enriching the misconception library it relies on. To do so, uncaptured values provided by students should be analyzed as they directly highlight where we currently still lose track of students' errors.

8 CONCLUSION

This paper investigates the automatic generation of C pointer statement in the context of a computer programming class. Our aim is to develop a structured approach that generates a wide range of C pointer statements to help students practice and improve their understanding of this complex subject. The approach also aims at reducing the workload for teachers in manually creating C pointer statements and at reducing the likelihood of academic dishonesty coming from students copying each other's answers or using external helps. The objective also extends to providing a comprehensive overview of the field of exercise generation to inspire further development in the generation of exercises for various subjects.

A flexible framework for exercises using pattern templates has been developed. Patterns are sufficiently restrictive to allow control over the production of C pointer statements, ensuring that only relevant expressions are given to students and their concepts are consistent for all students. At the same time, the framework enables the creation of unique C pointer statements by exploiting the different choices that can be made in the pattern and the uniqueness of the memory and variables. That framework has been implemented in Tartare (au**T**om**A**tic C pointe**R** statemen**T** gener**A**to**R** with f**E**edback). We evaluated Tartare performance over two dimensions: unique C pointer statements generation and patterns distribution within a C pointer statement. The paper also discussed generated C pointer statements relevance.

SOFTWARE ARTEFACT

Tartare is written in Python 3. It requires the Pandas library for working properly. Tartare must be tuned according to parameters listed in Table 4. Tartare source code is available at this URL: https://gitlab.uliege.be/cse.

ACKNOWLEDGMENTS

Authors would like to thank Pascal Fontaine for his careful readings and insights on Tartare and on their manuscript.

This work is supported by the CyberExcellence project funded by the Walloon Region, under number 2110186.

REFERENCES

[1] B. Adcock, P. Bucci, W. D. Heym, J. E. Hollingsworth, T. Long, and B. W. Weide. 2007. Which Pointer Errors Do Students Make?. In *Proc. SIGCSE Technical Symposium on Computer Science Education.*

[2] M. Agarwal and P. Mannem. 2011. Automatic Gap-Fill Question Generation from Text Books. In *Proc. Workshop on Innovative Use of NLP for Building Educational Applications (IUNLPBEA).*

[3] U. Z. Ahmed, S. Gulwani, and A. Karkare. 2013. Automatically Generating Problems and Solutions for Natural Deduction. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI).*

[4] I. Aldabe, M. Lopez de Lacalle, M. Maritxalar, E. Martinez, and L. Uria. 2006. ArikIturri: An Automatic Question Generator Based on Corpora and NLP Techniques. In *Proc. Intelligent Tutoring Systems (TS).*

[5] J. J. Almeida, E. Grande, and G. Smirnov. 2016. Context-Free Grammars: Exercise Generation and Probabilistic Assessment. In *Proc. Symposium on Languages, Applications and Technologies (SLATE).*

[6] C. Alvin, S. Gulwani, R. Majumdar, and S. Mukhopadhyay. 2015. *Automatic Synthesis of Geometry Problems for an Intelligent Tutoring System.* cs.AI 1510.08525. arXiv.

[7] T. Barnes and J. Stamper. 2008. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Proc. Intelligent Tutoring Systems (ITS).*

[8] L. Becker, S. Basu, and L. Vanderwende. 2012. Mind the Gap: Learning to Choose Gaps for Question Generation. In *Proc. Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.*

[9] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. 2007. Threshold concepts in computer science: do they exist and are they useful? *ACM SIGCSE Bulletin* 39, 1 (March 2007), 504–508.

[10] G. Brieven, L. Malcev, and B. Donnet. 2023. *Training Students' Abstraction Skills Around a Café 2.0.* cs.CY 2309.09562. arXiv.

[11] O. Chinedu and A. Ade-Ibijola. 2023. Synthesis of Nested Loop Exercises for Practice in Introductory Programming. *Egyptian Informatics Journal* 24, 2 (July 2023), 191–203.

[12] M. Craig and A. Petersen. 2016. Student Difficulties with Pointer Concepts in C. In *Proc. Australasian Computer Science Week Multiconference (ACSW).*

[13] J. Crawford, K. Butler-Henderson, J. Rudolph, B. Malkawi, M. Glowatz, R. Burton, P. A. Mangi, and S. Lam. 2020. COVID-19: 20 Countries' Higher Education Intra-Period Digital Pedagogy Responses. *Journal of Applied Learning & Teaching* 3, 1 (2020), 9–28.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT).*

[15] C. Douce, D. Livingstone, and J. Orwell. 2005. Automatic Test-Based Assessment of Programming: A Review. *Journal on Educational Resources in Computing* 5, 3 (September 2005), 4–es.

[16] A. DuFrene. 2016. *Automatic Generation and Grading of Programming Exercises.* Technical Report. California Polytechnic State University.

[17] P. Ertmer, J. Richardson, B. Belland, D. Camin, P. Connolly, G. Coulthard, K. Lei, and C. Mong. 2007. Using Peer Feedback to Enhance the Quality of Student Online Postings: An Exploratory Study. *Journal of Computer-Mediated Communication* 12, 2 (January 2007), 412–433.

[18] G. E. Forsythe and N. Wirth. 1965. Automatic grading programs. *Communications of the 'ACM'* 8, 5 (May 1965), 275–278.

[19] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. Loui, and C. Zilles. 2008. Identifying Important and Difficult Concepts in Introductory Computing Courses using a Delphi Process. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE).*

[20] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (1970), 242–257.

[21] J. B. Hext and J. W. Winings. 1969. An Automatic Grading Scheme for Simple Programming Exercises. *Commun. ACM* 12, 5 (May 1969), 272—-275.

[22] G. Hill, J. Mason, and A. Dunn. 2021. Contract Cheating: an Increasing Challenge for Global Academic Community Arising from CoViD-19. *Research and Practice in Technology Enhanced Learning* 16, 1 (December 2021).

[23] J. Hollingsworth. 1960. Automatic Graders for Programming Classes. *Commun. ACM* 3, 10 (October 1960), 528−−529.

[24] E. Holohan, M. Melia, D. McMullen, and C. Pahl. 2006. The Generation of e-Learning Exercise Problems from Subject Ontologies. In *Proc. IEEE International Conference on Advanced Learning Technologies (ICALT)*.

[25] L. Kaczmarczyk, E. Petrick, J. East, and G. Herman. 2010. Identifying student misconceptions of programming. In *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*.

[26] T. Klein and M. Nabi. 2019. *Learning to Answer by Learning to Ask: Getting the Best of GPT-2 and BERT Worlds.* cs.CL 1911.02365. arXiv.

[27] Z. Liang, W. Yu, T. Rajpurohit, P. Clark, X. Zhang, and A. Kaylan. 2023. *Let GPT be a Math Tutor: Teaching Math Word Problem Solvers with Customized Exercise Generation.* cs.LG 2305.14386. arXiv.

[28] S. Liénardy, L. Leduc, D. Verpoorten, and B. Donnet. 2021. Challenges, Multiple Attempts, and Trump Cards – A Practice Report of Student's Exposure to an Automated Correction System for a Programming Challenges Activity. *International Journal of Technologies in Higher Education (IJTHE)* 18, 2 (June 2021), 45–60.

[29] K. Malinka, M. Peresini, A. Firc, O. Hujnak, and F. Janus. 2023. On the Educational Impact of ChatGPT: Is Artificial Intelligence Ready to Obtain a University Degree?. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

[30] I. Milne and G. Rowe. 2002. Difficulties in Learning and Teaching Programming—Views of Students and Tutors. *Education and Information Technologies* 7 (March 2002), 55–66.

[31] R. Mitkov and L. A. Ha. 2003. Computer-Aided Generation of Multiple-Choice Tests. In *Proc. Workshop on Building Educational Applications Using Natural Language Processing (HLT-NAACL-EDUC)*.

[32] A. Papasalouros. 2013. Automatic Exercise Generation in Euclidean Geometry. In *Proc. Artificial Intelligence Applications and Innovations (AIAI)*.

[33] P. M. Phothilimthana and S. Sridhara. 2017. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?. In *Poc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*.

[34] T. W. Price, Y. Dong, and T. Barnes. 2016. Generating Data-driven Hints for Open-ended Programming. In *Proc. International Conference on Educational Data Mining (EDM)*.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. 2019. Language Models are Unsupervised Multitask Learners. Last Access: September 21st, 2023.

[36] D. Radosevic, T. Orehovački, and Z. Stapic. 2010. Automatic On-line Generation of Student's Exercises in Teaching Programming. *Proc. Central European Conference on Information and Intelligent Systems (CECIIS)*.

[37] K. Rivers and K. Koedinger. 2014. Automating Hint Generation with Solution Space Path Construction. In *Proc. Intelligent Tutoring Systems (ITS)*.

[38] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proc. International Conference on Software Engineering (ICSE)*.

[39] D. Sadigh, S. A. Seshia, and M. Gupta. 2012. Automating Exercise Generation: A Step towards Meeting the MOOC Challenge for Embedded Systems. In *Proc. Workshop on Embedded and Cyber-Physical Systems Education (WESE)*.

[40] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proc. ACM Conference on International Computing Education Research (ICER)*.

[41] R. Singh, S. Gulwani, and S. Rajamani. 2012. Automatically Generating Algebra Problems. In *Proc. AAAI Conference on Artificial Intelligence*.

[42] R. Singh, S. Gulwani, and A. Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[43] Y. Skalban, L. Specia, and R. Mitkov. 2012. Automatic question generation in multimedia-based learning. In *Proc. International Conference on Computational Linguistics (COLING): Posters*.

[44] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller. 2022. Inputs From Hell:. *IEEE Transactions on Software Engineering* 48, 4 (April 2022), 1138–1153.

[45] P. N. Sovietov. 2021. Automatic Generation of Programming Exercises. *Proc. International Conference on Technology Enhanced Learning in Higher Education (TELE)* (June 2021).

[46] J. Weizenbaum. 1966. ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine. *Commun. ACM* 9, 1 (January 1966), 36—-45.

[47] J. H. Wolfe. 1976. Automatic Question Generation from Text – an Aid to Independent Study. *ACM SIGCSE–SIGCUE Outlook* 10, SI (February 1976), 104—-112.

A EXAMPLE OF C POINTERS STATEMENT

Consider the following memory state:

	1008	??
w:	1004	28
	1000	8
		…
a[0]:		…
		…
	36	1996
	32	19
	28	1200
	24	4
v:	20	1000

The first column gives variable names, the second one provides memory addresses (expressed, in this exercise, as decimal numbers), and the third column gives the value stored at this address (?? means the value is undetermined). We also assume variables are declared as follows (throughout the exercise `int` are on four bytes, `short` are on two and `long` on eight):

```
#include<stdio.h>
int w,v, *b = &v;
char *a = "Invariant";
char *l = a + 3;
```

In this exercise, you have to evaluate the following expressions. You must consider that, between each expression, the memory is reinitialized to its initial state, as illustrated above. In other words, each expression is evaluated on the same initial state of the memory, there is no history between expressions. If a memory access leads to an address outside the [20, 1008] interval, indicate the segmentation error with value `SF` (for `Segmentation Fault`). Addresses are also represented on four bytes.

(1) `v`
(2) `*(int*)v`
(3) `&v`
(4) `&*b`
(5) `*(int *)*b`
(6) `&*(b-4)`
(7) `*b++`
(8) `*(++b)`
(9) `b-- + --w`
(10) `&w - b`
(11) `&w - 16`
(12) `*(b+4)`
(13) `(long *) &w - 8`
(14) `(short*) b+3`
(15) `*++b`
(16) `a['c'-'a']`
(17) `printf("%s", l)`
(18) `a[4] - a[3]`
(19) `(a-l)`
(20) `++b + *(&v +3)`