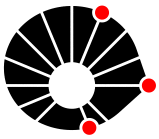

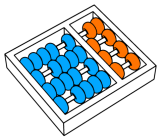

An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C

Ricardo Caceffo *Breno de França* *Guilherme Gama*
Raysa Benatti *Tales Aparecida* *Tania Caldas*
Rodolfo Azevedo

Technical Report - IC-17-15 - Relatório Técnico
October - 2017 - Outubro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C

Ricardo Caceffo, Breno de França, Guilherme Gama,
Raysa Benatti, Tales Aparecida, Tania Caldas, Rodolfo Azevedo

Instituto de Computação Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176
13083-970 Campinas-SP, Brasil
caceffo@ic.unicamp.br; breno@ic.unicamp.br, guilheremegama@gmail.com;
raysa.benatti@students.ic.unicamp.br; tales.aparecida@gmail.com; unicamp2010@ig.com.br; rodolfo@ic.unicamp.br

Abstract This work is a partial report related to the development and assessment of a Concept Inventory to Introductory Programming Courses. A Concept Inventory is a set of multiple-choice questions addressing specific misunderstandings and misconceptions of the students. In previous works, through instructor interviews, exam analysis, an online pilot test and interviews with students, we identified a list of 33 misconceptions related to 7 programming topics in C language. On this report we describe each one of these misconceptions, following an antipattern template composed by: code (a label to identify the misconception); name; description; rationale (the reason why we hypothesize the misconception happens); consequences; detection (where and how the misconception appears); and improvement (how to prevent the misconception).

Keywords: Misconception, Introductory Programming, Concept Inventory, Antipattern

1. Introduction

This report presents an ongoing work related to the development and assessment of a Concept Inventory (CI) to CS1 Introductory Programming Courses. A Concept Inventory is a set of multiple-choice questions addressing specific misunderstandings and misconceptions of the students [3].

In previous work [2] we analyzed two CS1 course exams (based on C programming language), classifying the students' errors according to misunderstandings (misconceptions) that might explain incorrect answers. We also conducted semi-structured interviews [5] with five instructors, asking them about the main misconceptions they believed to be common for students. Through this process, we identified seven topics and their misconceptions for a CI: (A) Function Parameter Use and Scope (B) Variables, Identifiers, and Scope; (C) Recursion; (D) Iteration; (E) Structures; (F) Pointers; and (G) Boolean Expressions. A total of 19 misconceptions were identified, two each for Topics G and E, and three each for the others. Figure 1 illustrates the methodology used in the study [2]:

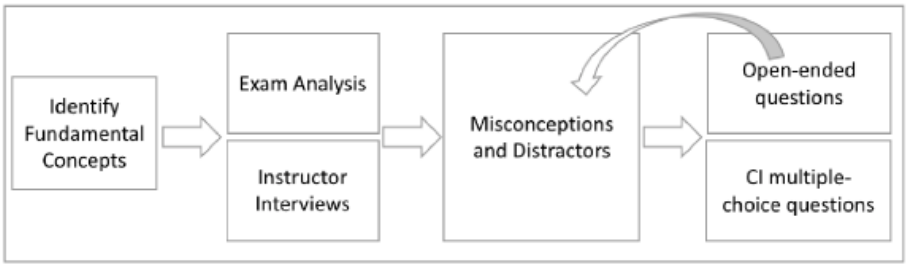

Figure 1) Methodology used in previous work [2] to map student's misconceptions

We have since developed a pilot test for the CI with three goals: (a) to validate the proposed misconceptions; (b) to identify additional misconceptions; and (c) to test the hypothesis that is possible to use analogy questions, not dependent on any particular programming language, to assess the concepts related to the introductory topics we identified.

Three questions were created for each topic, producing a pilot test for the CI with 21 questions. The first question for each topic was an open-ended language dependent question to identify new misconceptions. The second question was either a multiple-choice or an open-ended (for all other topics) analogy not specific to any programming language. The third

question (Q3) was multiple-choice and language-dependent. The work [4] presents the pilot test for the CI, as well as the data collected after a study conducted at the University of British Columbia (UBC). In that study, we found another 12 misconceptions, thus totalizing a list of 33 misconceptions (21 from the study [2] and 12 from the study [4]) related to CS1 programming in C.

In this report, we describe each one of these 33 misconceptions, following the Antipattern [1] method. This method shows the user how to achieve a good solution from a fallacious solution and how to avoid the specious solution in the future. In our context, the goal is to document the misconceptions found, also describing how to avoid them to happen in the future.

The Antipattern template used on this work includes:

- Code: a label identifying the misconception;
- Name: the misconception's name. It briefly describes the misconception. For example: "Parameter value set by an external source".
- Description: a more detailed explanation about what consists the misconception.
- Example: an example of a programming code in which the misconception happens (*i.e.*, it is a **wrong programming code**).
- Rationale: the reason why we believe the misconception happens.
- Consequences: what will happen if this misconception arises.
- Detection: where and how the misconception appears;
- Improvement: how to prevent the misconception.

2. Misconceptions

In this section we describe, through Antipatterns [1], all the 33 misconceptions found in previous studies [2, 4]. The seven categories and its misconceptions are:

- A) Function Parameter Use and Scope (6 misconceptions – from A.1 to A.6).
- B) Variables, Identifiers, and Scope (4 misconceptions – from B.1 to B.4)
- C) Recursion (3 misconceptions – from C.1 to C.3)
- D) Iteration (6 misconceptions – from D.1 to D.6)
- E) Structures (5 misconceptions – from E.1 to E.5)
- F) Pointers (5 misconceptions – from F.1 to F.5)
- G) Boolean Expressions (4 misconceptions – from G.1 to G.4)

A) Function Parameter Use and Scope

A.1

Code:	A.1
Name:	Parameter value set by an external source.
Description:	Setting a parameter value through a source outside the function scope.

Example:	`1. int func(int n){`
	`2.  scanf("%d ", &n);`
	`3.  (…)`
	`4. }`

Rationale:	Students do not consider that the parameter has some value attributed to it already. In this situation, they set the parameter value with an external source, such as a `scanf`.
Consequences:	The correct parameter value will be lost.
Detection:	**Where:** o At any statement of a function that receives a parameter.
	How: o The parameter value is inappropriately modified.
Improvement:	Students should be oriented, as an exercise, to verify, for example using a `printf`, the value of the parameter inside a function. If they are convinced the parameter has a value, they are unlikely to reassign it. Also, students should be explained the purpose of creating and using functions, and the importance of code modularization.

A.2

Code:	A.2
Name:	Parameters passed as if by reference.
Description:	Parameters passed as if by reference, instead of by value (*i.e.*, changes made inside a function would be reflected outside it).

Example:	`1. void addFive(int n){`
	`2.  n = n + 5;`
	`3. }`
	`4.`
	`5. int main(){`
	`6.  int x = 10;`
	`7.  printf("x is %d ", x);`
	`8.  addFive(x);`
	`9.  printf("x plus 5 is %d ", x);`
	`10.  return 0;`
	`11. }`

Rationale:	Students believe the function parameters refer, in memory, to the respective variables in the caller function.
Consequences:	The value of the variable(s) which the student expects to change will remain the same. This confusion can be aggravated when students learn pointers and use them as parameters. In this situation, both the parameter and the variable in the caller function points to the same memory address.
Detection:	**Where:** o In the next instruction that uses a variable previously passed as parameter to some function.
	How: o The value of some parameter is modified and it is expected that such change will reflect on the original variable, inside the caller function.
Improvement:	Students could be guided to verify and compare, for example with a `printf` or a debugger, the value of some variable before and after a function is called. Using different variable names (in the caller and for the parameter) would also help stress that the variables are not the same.

A.3

Code:	A.3
Name:	Parameters accessible outside their scope.
Description:	Assumption that parameters could be accessed from outside their scope, anywhere in the program.

Example:	`1. int func1(int n){`
	`2.  n = n + 5;`
	`3.  return n;`
	`4. }`
	`5.`
	`6. int func2(int x){`
	`7.  return x + n;`
	`8. }`
	
	There is an error on line 7, where an attempt is made to access the parameter `n`, which is local to `func1`, from within `func2`, which is outside its scope.

Rationale:	Students consider that the scope of parameter variables is global, and therefore they could be accessed from anywhere in the code.
Consequences:	- The program will not compile or; - If there is a variable (local or global), anywhere in the code, which has the same name of some parameter, the student could consider the program will use that parameter value instead of the local value, leading to unexpected behavior.
Detection:	**Where:** o Inside a function, when a parameter is used outside the function scope.
	How: o Student wants to access a parameter variable not visible in the current function scope.
Improvement:	Students should be oriented, through examples and exercises, to understand the difference between local and global scope of variables.

A.4

Code:	A.4
Name:	Incorrect order of function parameters.
Description:	Assumption that programs could identify the context of how variables are used, thus automatically adjusting the parameter order to correctly achieve the goal of the program or function.

Example:	`1. int subHelper(int a, int b){`
	`2.  return a - b;`
	`3. }`
	`4.`
	`5. int main(){`
	`6.  int a = 5;`
	`7.  int b = 7;`
	`8.`
	`9.  printf("a - b = %d", subHelper(a, b));`
	`10.  printf("b - a = %d", subHelper(a, b));`
	`11.  return 0;`
	`12.}`
	
	There is an error on line 10. The correct instruction would be:
	`10. printf(" b - a = %d ", subHelper (b,a));`

Rationale:	Students believe the program will understand the semantics related to the variable names and adjust their order properly.
Consequences:	The program will work accordingly the written order, which can output misleading results.
Detection:	**Where:** o When calling a function with multiple parameters.
	How: o The function call uses a wrong order for the parameters.
Improvement:	Students should be oriented to verify and compare, for example with a `printf`, debugging or assertive programming, the parameters values received by a function. Also, examine what would happen when the parameters order is changed.

A.5

Code:	A.5
Name:	Not catching the return value from a function
Description:	Assumption that return values are automatically handled by the caller function.

Example:	1. int squareOf(int n){
	2. return n * n;
	3. }
	4.
	5. int main(){
	6. int a = 5;
	7.
	8. squareOf(a);
	9. printf("%d", a);
	10. return 0;
	11. }
	
	There is an error on line 8. The correct instruction would be:
	8. a = squareOf(a);

Rationale:	Students believe the program is intelligent, i.e., it will understand the semantics related to a function call, automatically adjusting the logic and results.
Consequences:	The function's return value will not be received by the calling instance, rendering the function useless.
Detection:	**Where:** o In a function call that returns some value.
	How: o The returned value is not correctly handled.
Improvement:	Students should be oriented to always check whether a non-void function's return value is being properly assigned. Also, students should be told the difference between the input parameters and return value.

A.6

Code:	A.6
Name:	Logic error related to parameters when calling a function.
Description:	Presence of logic errors in the function call. The order of values of the parameters are wrong, different than those defined in the function interface/signature.

Example:	Create a main function that prints the result of `2a - b if a >= b`. Otherwise it prints the result of `b - a`. Use the `subHelper` function to process the subtractions.
	
	`1. int subHelper(int b, int a){`
	`2.  return b - a;`
	`3. }`
	`4.`
	`5. int main(){`
	`6.  int a = 5;`
	`7.  int b = 7;`
	`8.  int r;`
	`9.  if (a >= b){`
	`10.  r = subHelper(a, 2*b);`
	`11.  }`
	`12.  else{`
	`13.  r = subHelper(b, a);`
	`14.  }`
	`15.  printf("%d ", r);`
	`16.  return 0;`
	`17. }`
	
	There is an error on the line 10. The correct instruction would be:
	`10. subHelper(2*a, b);`

Rationale:	Students consider the program will understand the semantics related to a function call, automatically adjusting its logic and values.
Consequences:	Although the program will compile, it will not run as expected. The function call will return a wrong result, different than planned by the student, leading the program to a wrong (and unexpected) behavior.
Detection:	**Where:** o In a function call.
	How: o The parameter values or their order are not compatible with the function interface/signature.
Improvement:	Students should be oriented, for example with comments, about the purpose and role of each parameter in the function logic. The parameter names could be meaningful, referring to what the parameters are expected to represent.

B) Variables, Identifiers, and Scope

B.1

Code:	B.1
Name:	Local variables accessed outside their scope.
Description:	Out of scope assignment, considering local variables as if they were global variables.

Example:	`1. int func(int a, int b){`
	`2.  return a + b + c;`
	`3. }`
	`4.`
	`5. int main(){`
	`6.  int a = 5;`
	`7.  int b = 7;`
	`8.  int c = 10;`
	`9.`
	`10.  int r = func(a, b);`
	`11.  return 0;`
	`12. }`
	
	The error is in Line 2: the local variable `c` from the main function is accessed inside the function `func`.

Rationale:	Students believe the scope of local variables is global, so they could be accessed at any point of the code.
Consequences:	- The program could not compile or; - If there is a local variable in some function that has the same name as another local variable (in a different function), the student could consider the program will share the values of these variables, leading to unexpected behavior.
Detection:	**Where:** o In any function with local variables.
	How: o The function tries to access a variable declared inside another function.
Improvement:	Students should be guided regarding the existing variable scopes in the C language and how they work. Examples to illustrate this might include comparisons between similar codes with different variable names, stressing that the choice of names in no way affects their scopes.

B.2

Code:	B.2
Name:	Global variables considered as local in the current scope.
Description:	Out of scope assignment, considering or accessing global variables as if they were local to current scope.

Example:	`1. int max = 10;`
	`2. int func(int a, int b){`
	`3.  if (a > = b){`
	`4.  max = a;`
	`5.  else{`
	`6.  max = b;`
	`7.  }`
	`8.  return max;`
	`9. }`
	`10.`
	`11. int main (){`
	`12.  int a = 5;`
	`13.  int b = 7;`
	`14.  int r;`
	`15.  printf("The max value is %d ", max);`
	`16.  r = func(a, b);`
	`17.  printf("The max value is still %d ", max);`
	`18.  return 0;`
	`19. }`
	
	The program logic considers the global variable `max` as local to the function `func`. Therefore, the attributions made in the lines 4 and 6 would be local, not affecting the value of the global `max` variable. This would lead the `printf` in lines 15 and 17 to display the same value: `10`. However, as `max` is a global variable, the `printf` in line 17 will print the greatest number between `a` and `b` (in this case, the number `7`).

Rationale:	Students consider the variables scope as local, and so when a global variable value is changed inside a function, it would not affect other parts of the code.
Consequences:	Although the program will compile, it will not run as expected. In fact, the program logic will be wrong, as the students expect the value of global variables will change accordingly to the function manipulating it.
Detection:	**Where:** o In any function that accesses and assigns a value to a global variable.
	How: o The program logic is built upon the assumption that global variables behave like local variables.

Improvement:	Students should be oriented about the existing variable scopes in the C language and how they work. Debugging and assertive programming should also be employed.

B.3

Code:	B.3
Name:	Parameter confusion with same-name variable outside the function.
Description:	The wrong variable is accessed when a parameter has the same name of some local variable in the caller function.

Example:	`1. int addTwoInt(int a, int c){`
	`2.  return a + c;`
	`3. }`
	`4.`
	`5. int main(){`
	`6.  int a = 5;`
	`7.  int b = 7;`
	`8.  int c = 10;`
	`9.  int r = addTwoInt(a,b);`
	`10.  printf("The result value is %d", r);`
	`11.  return 0;`
	`12. }`
	
	The student expects the parameter `c` in function `addTwoInt` would have the value of `c` in the main function (10), instead of the value of `b` (7). Therefore, the `printf` on line 10 would output the number 15 instead of 12 (actual output).

Rationale:	Students consider variables with the same name refer to the same variable and memory values.
Consequences:	Although the program will compile, it will not run as expected. In fact, the program logic will be wrong, as the students will consider the parameter will refer to a different value.
Detection:	**Where:** o In any program that has at least two variables with the same name.
	How: o Student accesses or assign a value to some variable, but in fact they are accessing another variable in the memory but with the same name.
Improvement:	Students should be oriented to understand that, as C language allows variables in different scopes to have the same name, they must consider the scope when referring to them.

B.4

Code:	B.4
Name:	Global variables not accessible within a function.
Description:	The scope of any global variable is assumed to be invalid within the functions of a program.

Example:	`1. int max = 10;`
	`2. void compare(int a){`
	`3.  if (a > max)`
	`4.  printf("%d i s greater than %d ", a, max);`
	`5.  else`
	`6.  printf("%d is not greater than %d ", a, max);`
	`7. }`
	
	In this example, students would say the program fails to compile as the `max` variable cannot be accessed inside the `compare` function.

Rationale:	Students believe functions are unaware of external variables not passed as parameters. As a global variable is declared outside the scope of any given function, it follows that no function would be able to access it.
Consequences:	Students would feel compelled to pass global variables as arguments to functions, or even to avoid their use altogether.
Detection:	**Where:** o Any program making use of global variables.
	How: o Student fails to use a global variable, replaces it with a local counterpart, or passes it as an argument.
Improvement:	Students should be oriented to understand that any variable declared outside a function (a global variable) is accessible throughout the program.

C) Recursion

C.1

Code:	C.1
Name:	Wrong expression used to calculate the return value of a recursive function.
Description:	A recursive function returns a wrong value, leading to an improperly sequence of recursion calls.

Example:	`1. // My recursive function`
	`2. int recursive(int a){`
	`3.  if (a == 0)`
	`4.  return 1;`
	`5.`
	`6.  return recursive(++a);`
	`7. }`
	`8.`
	`9. int main () {`
	`10.  int r = recursive(1);`
	`11.  return 0;`
	`12. }`
	
	In this example, the wrong call in line 6 would lead to an infinite loop or a stack trace error. The correct instruction to be replaced in line 6 is:
	
	`6.  return recursive(--a);`

Rationale:	This misconception relates to a lack of understanding about how recursion works in C, specifically how the recursion instances are stored into and retrieved from the stack.
Consequences:	The recursive function will display unexpected behavior, leading to an infinite loop or a wrong number of iterations and recursive calls.
Detection:	**Where:** o Within the recursive function, specifically in the statement (parameter) of the recursive call.
	How: o The recursion either fails to converge to its base case (leading to an infinite loop) or does converge, but with a wrong number of iterations.
Improvement:	Students should understand that recursion is handled by a stack. In this way, each recursion's instance is stored in the current topper position of the stack. To complete the recursion a base case should be satisfied, *i.e.*, at some point the stack must cease to increase and each stored instance retrieved and resumed *(last in, first out* scheme). For example, this understanding could be reached through debugging techniques, such as the use of the `printf` function to identify each instance position in the stack.

C.2

Code:	C.2
Name:	Recursive function never calls itself.
Description:	A (presumably) recursive function lacks a recursive call to itself. The function is called just once, thus not being used recursively.

Example:	`1. // My recursive function`
	`2. int recursive(int a){`
	`3.  if (a == 0){`
	`4.  return 1;`
	`5.  }`
	`6. }`
	`7.`
	`8. int main(){`
	`9.  int r = recursive (10);`
	`10.  return 0;`
	`11.}`

Rationale:	Students have a weak grasp on the concept of recursion itself, leading to the belief that the function does not need to contain a recursive call.
Consequences:	As the presumably recursive function contains only the stopping condition, the expected recursive calls will not happen, causing the program not to execute as expected.
Detection:	**Where:** o In any recursive function.
	How: o The recursive function lacks a recursive call to itself.
Improvement:	Students should understand that in C language recursion is handled by a stack. In this way, each recursion's instance is stored in the current topper position of the stack. To complete the recursion a base case should be satisfied, *i.e.*, at some point the stack must cease to increase and each stored instance retrieved and resumed *(last in, first out* scheme). For example, this understanding could be reached through debugging techniques, like the use of the `printf` function to identify each instance position in the stack.

C.3

Code:	C.3
Name:	Function does not terminate at the base case.
Description:	A recursive function lacks a base case (stop condition).

Example:	1. // My recursive function
	2. int recursive(int a){
	3.
	4. return recursive (--a);
	5. }
	6.
	7. int main () {
	8. int r = recursive (10);
	9. return 0;
	10. }

Rationale:	The recursive function is written upon the assumption that it will indistinctly stop at a given time. The student does not have the proper knowledge to define stop conditions within the recursive function, which would allow the function to halt as required.
Consequences:	The program will not stop until it reaches a stack overflow error.
Detection:	**Where:** o Within recursive functions.
	How: o The recursive function does not have a stop condition.
Improvement:	Students can be oriented on how the recursive calls mechanism works (aspects such as the memory stack and how the compiler interprets recursive calls), so that they understand that stop conditions are crucial.

D) Iteration

D.1

Code:	D.1
Name:	Improper update of a loop counter
Description:	A loop counter is improperly updated, leading to an incorrect number of iterations.

Example:	`1. int i = 0;`
	`2. int sum = 0;`
	`3. while (i < 10){`
	`4.  sum = sum + i;`
	`5.  i = 1;`
	`6. }`
	
	The issue is shown on line 5 (the correct counter update would be `i++`).

Rationale:	Students don't understand how the loop counter should be increased or decreased to reach the desired number of iterations.
Consequences:	Infinite loop or wrong results, if the output uses values calculated inside the loop.
Detection:	**Where:** o In any loop structure (`for`, `while` or `do/while`).
	How: o The loop counter is not updated or is wrongly updated.
Improvement:	Students should be able to check (*e.g.* through the debug or a `printf`) the counter value at each iteration. In this way, they would be able to realize the counter is not updating as expected.

D.2

Code:	D.2
Name:	Use of loop result before loop completes.
Description:	Partial results found during loop iterations are used or considered as if they were final.
Example:	1. int sum = 0, i; 2. for (i = 0; i <= 9; i++){ 3. sum = sum + i; 4. printf ("The sum is %d", sum); 5. } 6. printf("The sum is %d", sum); For each iteration, the partial result is needlessly printed (line 4).
Rationale:	Students feel inclined to group all instructions related to a given loop inside the body of that loop.
Consequences:	The loop will not produce the expected result.
Detection:	**Where:** o Wherever a loop is used and there are operations related to its final result.
	How: o Students include in the body of the loop instructions that are meant to use its final result.
Improvement:	Students should be oriented to check each instruction within a loop and determine whether it depends on a partial result or a final result, moving the latter outside of the body.

D.3

Code:	D.3
Name:	Improper initialization or stop condition of loop counter.
Description:	Improper initialization of the loop counter or improper construction of its stop condition, leading to an unexpected number of loop iterations.

Example:	Example:
	Write a program that prints the sum of all numbers from 1 to 9:
	
	`1. int sum = 0, i;`
	`2. for (i = 1; i < 9; i++){`
	`3.  sum = sum + i;`
	`4. }`
	`5. printf ("The sum is %d ", sum);`
	
	In this case, the `for` stop condition should be `i <= 9` instead of `i < 9`.

Rationale:	Students does not understand how to construct the loop structure (counter initialization or stop condition) to support a specific number of iterations. Related to the former, students do not know how to properly build the mathematical comparison (e.g use the >= or the >).
Consequences:	The number of times the loop iterates will be wrong, leading to unexpected program behavior.
Detection:	**Where:** o In the initialization of the loop counter or in the loop stop condition.
	How: o Either the loop counter initialization or the loop stop condition are wrong, leading to an unexpected number of iterations.
Improvement:	Students should be oriented to check how many times the loop will iterate, also verifying the value of the loop counter in each loop iteration.

D.4

Code:	D.4
Name:	Wrong control flow in a loop.
Description:	The number of times a loop is executed is wrong.

Example:	Write a program that prints the sum of all integer numbers, from 0 to 9:
	`1. int sum, i;`
	`2. for (i = 0; i <= 10; i++){`
	`3.  sum += i;`
	`4. }`
	`5. printf ("The sum is %d", sum);`

Rationale:	Students do not understand how to write a loop structure correctly, *i.e.*, how to represent in a loop command structure a situation that must be executed a specific number of times. This issue relates to understanding issues about the initialization of the loop iterator variable as well as its increment or stop condition.
Consequences:	The loop will not behave as expected. The number of iterations will be greater or lower than it should be.
Detection:	**Where:** ○ *For* iterations: The issue can be found in the *for* structure declaration, specifically in the initialization of the loop control variable, or in its increment or stop condition steps. ○ *While* iterations: The issue can be found before the *while* structure declaration, when the loop control variable is (or not) declared and initialized; it also can be found in the *while* condition, that determines when the loop should be executed and; it can be found in the *while* code, that can have a wrong (or absent) control variable increment.
	How: To detect this misconception it is necessary to analyze the intended use of the loop, *i.e.*, determining of what the loop is supposed to do (or calculate) and how many times it is to be executed.
Improvement:	Students should be oriented to check if the loop control variable has been correctly initialized and incremented. Also, they should check if the loop stop condition has been correctly defined. In order to correctly control how many times the loop is executed, students could insert a `printf` statement into the loop block, printing the values of the loop control variables.

D.5

Code:	D.5
Name:	No loop, only one simple iteration.
Description:	The program is able to automatically create and configure a loop, dispensing the need to write the respective instruction.

Example:	1. int isDivisibleBy(int n, int x){
	2. if (n % x == 0) {
	3. return 1;
	4. }else {
	5. return 0;
	6. }
	7. }
	8.
	9. int main(){
	10. int x, isPrime;
	11. scanf ("%d ", &x);
	12. isPrime = isDivisibleBy(x,1);
	13. if (isPrime == 1)
	14. printf (" Number %d is prime", x);
	15. else
	16. printf(" Number %d is not prime", x);
	17. return 0;
	18. }
	
	In this example, the call to `isDivisibleBy` is made only once (line 12), without any iteration.

Rationale:	The call to `isDivisibleBy` is written upon the assumption that the loop statement and its definitions will be automatically imposed by the compiler.
Consequences:	In most of the cases the loop absence will make the program not work as expected. In specific situations (when the loop would iterate only one time) the program will work successfully.
Detection:	**Where:** o Anywhere throughout the code where it appears that a repetition structure was intended.
	How: o The code logic requires some block to be repeated several times. However, the program does not have any repetition loop statement.
Improvement:	Students should be oriented on how the loop statement is interpreted by the compiler and that is mandatory to use a loop command (`for`, `while` or `do while`) to have a loop.

D.6

Code:	D.6
Name:	Loop construction does not consider the logic or its connection with other parts of the code.
Description:	Although the loop is internally well constructed, there is a logic error when the big picture is analyzed, *i.e.*, how the loop interacts with the code before and after it.

Example:	`1. int isDivisibleBy(int a, int b){`
	`2.  if (a % b == 0)`
	`3.  return 1;`
	`4.  else`
	`5.  return 0;`
	`6. }`
	`7. int main(){`
	`8.  int x;`
	`9.  int c = 2;`
	`10.  int foundDivisible = 1;`
	`11.  scanf("%d", &x);`
	`12.  while (!foundDivisible && c < x){`
	`13.  foundDivisible = isDivisibleBy(x, c);`
	`14.  c++;`
	`15.  }`
	`16.  if (!foundDivisible)`
	`17.  printf("Number %d is prime");`
	`18.  else`
	`19.  printf("Number %d is not prime");`
	`20.  return 0;`
	`21.}`
	
	The `while` conditional expression on line 12 is evaluated as false on the first attempt, leading to a program that classifies all numbers as prime. The correct instruction is:
	`10.  int foundDivisible = 0;`
	
	Another variation for this example would be if the line 16 contained the following instruction:
	`16.  if (foundDivisible)`
	
	In this situation, even if the loop correctly identifies if a number is prime, the loop output (`foundDivisible` variable) is wrongly used by other program parts.

Rationale:	Students build the loop considering it is an independent part of the code, not related to any previous and former code.
Consequences:	The program behavior will not be as expected, leading to logic errors.
Detection:	**Where:** o Whenever a loop is used within the code.

	How: ○ If considered independently, the loop is well constructed. However, when considering the whole function that contains the loop, there is a logic error in the data that is used by the loop (input) or generated by the loop (output) and used in other parts of the code.
Improvement:	Students should be oriented to understand the loop is not an independent entity, thus being influenced by previous code and also influencing the code that is after it.

E) Structures

E.1

Code:	E.1
Name:	No fields specified in comparison of `structs`.
Description:	Attempt to compare entire `structs` variable using only their identifiers in a Boolean expression, rather than comparing each of their respective fields.

Example:	1. typedef struct{
	2. int x, y;
	3. } Point;
	4.
	5. int main(){
	6. Point p1, p2;
	7.
	8. // values are attributed to fields of
	9. // p1 and p2.
	10.
	11. if (p1 == p2){
	12. printf("The points are equal.");
	13. (…)
	
	Line 11 would result in a compilation error. The correct instruction would be:
	11. if ((p1.x == p2.x) && (p1.y == p2.y)) {

Rationale:	Students consider the program automatically compares all fields of the `structs`, thus not being necessary to specify any of them on the sentence.
Consequences:	The program will not compile. Also, even if it were possible to compile, the evaluation of the comparison expression would be wrong, leading to incorrect program behavior.
Detection:	**Where:** o Whenever `structs` are compared to each other.
	How: o `Structs` are compared directly through their identifiers, no fields are specified.
Improvement:	Students should be oriented to understand that in C language there is no implicitly comparison of `structs`, and so it is mandatory to specify all desired fields.

E.2

Code:	E.2
Name:	Field of a `struct` is compared to the identifier of another.
Description:	Comparison of a specific field of some `struct` variable against the bare identifier of another.
Example:	`if (structVar1.field == structVar2)`
Rationale:	Students consider that, because they have specified a field of the first `struct`, the program automatically knows they would like to access the same field in the other `struct` variable.
Consequences:	The program will not compile. Also, even if it were possible to compile, the evaluation of the comparison expression would be wrong, leading to a incorrect program behavior.
Detection:	**Where:** o On the line where a field of some `struct` is compared against another `struct` variable.
	How: o A field of some `struct` is compared against another `struct` variable.
Improvement:	Students should be oriented to understand that in C language there is no implicitly comparison of `structs`, which makes it mandatory to specify all desired fields in both `structs`. Also, they should check the `struct` declarations, verifying whether the fields to be compared exists and are compatible.

E.3

Code:	E.3
Name:	`Struct` variable is considered and accessed as a pointer.
Description:	Assumption that `struct` variables are always pointers, even when they are not declared as such.

Example:	1. typedef struct {
	2. int x, y;
	3. } Point;
	4.
	5. int main() {
	6. Point p1;
	7.
	8. p1 -> x = 10;
	9. (*p1).y = 5;
	10. return 0;
	11. }
	12.
	
	Lines 8 and 9 would result in a compilation error. The correct instruction would be:
	8. p1.x = 10;
	9. p1.y = 5;

Rationale:	This misconception can be explained because students learn that a function conceived to handle a `struct` variable and modify some of its fields, also reflecting this change in the caller function, receives a pointer to the `struct` variable. For example:
	
	1. typedef struct {
	2. int x, y;
	3. } Point;
	4.
	5. void changePoint(Point* p, int x, int y){
	6. p -> x = x;
	7. (*p).y = y;
	8. }
	9. int main() {
	10. Point p1;
	11. changePoint (&p1, 10, 20);
	12. return 0;
	13.}
	
	Therefore, students generalize the use of the `->` and `*` operators even in situations where the variable is not a pointer to a struct.

Consequences:	- Compilation error ("base operand has non-pointer type" or similar) and; - Students with this misconception will possible have issues when accessing or manipulating pointer variables.

Detection:	**Where:** o In a code with a `struct`, when there is an attempt to access one or more of its fields(s).
	How: o Although the `struct` variable is not a pointer, it is considered and accessed as one.
Improvement:	Students should understand the specific situation where a pointer to a `struct` variable is passed as a function parameter.

E.4

Code:	E.4
Name:	`Struct` field accessed as an array index.
Description:	`Struct` fields called upon using array index syntax.

Example:	`1. typedef struct{`
	`2.  int x, y;`
	`3. } Point;`
	`4.`
	`5. int main(){`
	`6.  Point p1;`
	`7.  p1[0] = 10; // Setting the field x with 10`
	`8.  p1[y] = 20; // Setting the field y with 20`
	`9.  return 0;`
	`10.}`
	
	`Lines 7 and 8 are wrong. The correct would be:`
	`7. p1.x = 10; // Setting the field x with 10`
	`8. p1.y = 20; // Setting the field y with 20`

Rationale:	Students interpret the `struct` type as an array that could have its fields accessed by position (indexing from 0 to N) or name.
Consequences:	- The program will not compile and; - Students with this misconception will possible have issues when accessing or manipulating array and pointer variables.
Detection:	**Where:** o Anywhere throughout the program where a `struct` field is accessed.
	How: o The `struct` field is accessed through the bracket syntax (`[ ]`).
Improvement:	Students should be oriented on how `structs` and arrays are stored and accessed in memory by the compiler.

E.5

Code:	E.5
Name:	`Struct` variable name preceded by `struct` type after declaration.
Description:	The `struct` type is added before variables name.

Example:	`if (StructName structVar1.field == StructName structVar2.field)`
	or
	`if (StructName (struct.field == struct.field) )`

Rationale:	Students consider it is necessary to repeat the `struct` type everywhere, not just when declaring a variable.
Consequences:	The program will not compile due to a syntax error.
Detection:	**Where:** o Anywhere a `struct` is used after it has been declared.
	How: o Student writes the `struct` type, alongside the variable name.
Improvement:	Students should be orientated regarding the right context to use `struct` types.

F) Pointers

F.1

Code:	F.1
Name:	Using & instead of * to dereference pointer.
Description:	Attempt to dereference a pointer with the operator `&`, rather than `*`.

Example:	`1. void addTen(int* a){`
	`2.  &a = &a + 10;`
	`3. }`
	
	Line 2 above contains a compilation error, as the `&a` operator indicates a hexadecimal number (a memory address). For example, if the address of variable `a` is `10FF21`, the instruction `&a = &a + 10;` would be translated by the compiler as `10FF21 = 10FF21 + 10;` which is an invalid statement. The correct instruction would be:
	`2.  *a = *a + 10;`

Rationale:	Students consider the memory address referenced by a pointer variable is accessed through the `&` operator.
Consequences:	The statement is invalid. The program will not compile.
Detection:	**Where:** ○ Whenever it is necessary to directly manipulate the values pointed by pointer variables.
	How: ○ Student attempts to dereference a pointer variable with the `&` operator.
Improvement:	Students should be oriented to understand the difference between the `&` and `*` operators, and to consider, when writing code, whether they wish to use the address or the pointed value.

F.2

Code:	F.2
Name:	Not dereferencing a pointer variable to get the pointed value.
Description:	Absence of the `*` operator when dereferencing a pointer variable.

Example:	`1. int addTen(int* a){`
	`2.  return = a + 10;`
	`3. }`
	
	Line 2 above contains an error, as the value of the pointer variable `a` (a memory address) is increased by 10. The correct approach would be to first access the memory address pointed by the variable `a` – an integer – and then increase its value by 10:
	`2.  return *a + 10;`

Rationale:	Students consider the compiler automatically deferences a pointer variable.
Consequences:	Although the program will still compile, changing the memory address that is within the variable `a` can result in further segmentation faults if the new address is accessed or its value is changed.
Detection:	**Where:** o Whenever it is necessary to directly manipulate the values of pointer variables.
	How: o The pointer variable is used without the `*` operator, so the address stored in that variable is changed.
Improvement:	Students should be oriented to understand what a pointer variable is and what type of values pointer variables store.

F.3

Code:	F.3
Name:	Assigning invalid address to a pointer variable.
Description:	Assignment of an invalid address to a pointer variable.

Example:	`1. void doSomething(){`
	`2.  int* a;`
	`3.  int b = 10;`
	`4.  a = b;`
	`5  printf("The value pointed by a is %d", *a);`
	`6.  (…)`
	`7. }`
	
	On this example, the pointer variable `a` is set to an invalid address in line 4. The correct would be to set `a` to a valid address, such as that of the `b` variable:
	`4.  a = &b;`

Rationale:	Students does not understand exactly what a pointer variable should store.
Consequences:	The pointer variables will not contain the desired memory address, leading to an unexpected program behavior and a possible crash when the wrong address is accessed.
Detection:	**Where:** o When a value (*i.e.* not an address) is set to a pointer variable.
	How: o The pointer variable receives a wrong or no memory address.
Improvement:	Students should be instructed that pointer variables store memory addresses, and variable memory addresses can be obtained through the `&` operator.

F.4

Code:	F.4
Name:	Void function returns value.
Description:	A void function is able to return a value.

Example:	`1. void changeValues (int a){`
	`2.  a= 20 + a;`
	`3.  return a;`
	`4. }`
	
	In this example there is an issue in the function declaration (line 1), that has `void` as return type. The correct instruction is:
	`1. int changeValues (int a) {`

Rationale:	The students consider the presence of a `return` statement sufficient to determine a function's return type as non-void; they do not realize a function can only return the type determined at the function declaration. The proper concept of what a function does is unclear.
Consequences:	The program will not compile. Also, the program logic (algorithm) — and consequently its behavior — might not work as expected, as the program could be constructed on a misinterpretation of the role and use of functions.
Detection:	**Where:** o In void functions with the `return` statement.
	How: o Although the function returns a value, its declaration contains `void` as return instead of the proper returned type.
Improvement:	Students should be instructed on the role of functions, how they can be used to modularize the programs and what is the correct syntax make a function return a value.

F.5

Code:	F.5
Name:	Parameters are passed as if by reference.
Description:	Any modification of a parameter value within a function is reflected on the original variable on the caller function.

Example:	`1. void addTen(int a){`
	`2.  a = a + 10;`
	`3. }`
	`4.`
	`5. int main (){`
	`6.  int i = 10;`
	`7.  // Add 10 to i`
	`8.  addTen (i);`
	`9.  printf ("20 = %d", i);`
	`10.  return 0;`
	`11. }`

Rationale:	Students consider that parameters are pointers (thus containing the value of a memory address), so changing the value of some parameter within a function would also change the original value in the caller function.
Consequences:	Although the program will compile, its logic will be wrong, leading to an unexpected behavior.
Detection:	**Where:** o In any function that receives a parameter.
	How: o The parameter value is modified within the function, and the program logic considers that modification is reflected on the original caller function.
Improvement:	Students should be oriented that in C language all parameters (including pointers) are passed by value, not reference.

G) Boolean Expressions

G.1

Code:	G.1
Name:	Incorrect precedence for Boolean operators.
Description:	An incorrect order of precedence in a Boolean expression.

Example:	Consider the following definition to leap year:
	"A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400."[1]
	
	Write a function that receives an integer and returns 1 if it corresponds to a leap year, and 0 otherwise:
	
	`1. int isLeapYear(int year){`
	`2.  if ((year % 400 == 0 || year % 4  == 0)`
	`  && year % 100 != 0)`
	`3.  return 1;`
	`4.  else`
	`5.  return 0;`
	`6. }`
	
	Line 2 above contains an error. The correct expression would be
	`2.  if (year % 400 == 0 ||`
	`  (year % 4  == 0 && year % 100 != 0))`

Rationale:	Students consider that Boolean expressions can be transcribed directly from English and will be automatically understood by the program.
Consequences:	The expression will not evaluate correctly, leading to an unexpected program behavior.
Detection:	**Where:** ○ Whenever Boolean expressions are used, especially those involving more than one Boolean operator.
	How: ○ Students either fail to parenthesize the expression, or do so inappropriately.
Improvement:	Students should be oriented to review the order of precedence of C operators and, when writing Boolean expressions, to ascertain that they will be evaluated by the program in the desired order.

[1] Definition available at: https://www.programiz.com/c-programming/examples/leap-year
Acessed in: October 2017

G.2

Code:	G.2
Name:	Nested if-statements instead of a Boolean expression.
Description:	A Boolean expression is written as an `if` and `else` nested sequence.

Example:	`1. int main(){`
	`2.  int result = 0;`
	`3.  if (A){`
	`4.  if (B){`
	`5.  result = 1;`
	`6.  }`
	`7.  else`
	`8  if (C){`
	`9.  result = 1;`
	`10.  }`
	`11  }`
	`12.  if (result){`
	`13.  // Do something`
	`14.  }`
	`15.  return 0;`
	`16.}`
	
	The same above code should be written as:
	`1. int main(){`
	`2.  if (A && (B || C)) {`
	`3.  // Do something`
	`4.  }`
	`5.  return 0;`
	`6.  }`

Rationale:	Students do not know how to evaluate Boolean expressions with multiple variables, so they build a sequence of `if` and `else` statements, each containing a single variable to be evaluated.
Consequences:	Although the program will still work correctly with a sequence of `if` and `else` statements (*i.e.* this is not an error), longer Boolean expressions will lead to needlessly complex code, difficult to read, understand and debug.
Detection:	**Where:** o Whenever a Boolean expression must be evaluated within the code.
	How: o The Boolean expression is evaluated through a sequence of `if` and `else` statements.
Improvement:	Students should be oriented to understand how Boolean expressions are evaluated in C language, and also how to use the proper operators (`||`, `&&` and parenthesis) to build these expressions. Also, the equivalence to nested if-statements could be explored.

G.3

Code:	G.3
Name:	Arithmetic expression instead of Boolean expression.
Description:	A logic problem is evaluated through arithmetic instead of Boolean expression.

Example:	Consider the variables `a`, `b` and `c`, representing statements that can be true or false (meaning each variable would the value 1 or 0). Write a Boolean expression that evaluates whether at least two statements are true:
	
	`1. (…)`
	`2. if (a+b+c >=2) {`
	`3. (…)`

Rationale:	Students do not know how to evaluate Boolean expressions with multiple variables, so they some arithmetic property related to the problem as a shortcut.
Consequences:	Although the program will still correctly work with this approach, it is not a universal solution, *i.e.,* it depends on some arithmetical characteristic of the problem or expression.
Detection:	**Where:** o Whenever a Boolean expression must be evaluated within the code.
	How: o A logic problem is solved through a specific arithmetic property instead of a Boolean expression.
Improvement:	Students should be oriented to understand how Boolean expressions are evaluated in C language, and also how to use the proper operators (`||`, `&&` and parentheses) to build these expressions.

G.4

Code:	G.4
Name:	Attempt to evaluate a Boolean expression through loop iterations.
Description:	A situation that can be described as a Boolean expression is evaluated through a sequence of iterations within a loop.

Example:	Consider the variables `a`, `b` and `c`, which represents statements that can be true or false (meaning each variable would assume the value 0 or 1). Write a Boolean expression that evaluates whether at least two statements are true:
	
	`1. (…)`
	`2.  while ((a && b) || c){`
	`3.  // Do something`
	`4.  }`
	`5. (…)`
	
	The correct instruction is:
	`1. (…)`
	`2.  if ((a && b) || (a && c) || (b && c)){`
	`3.  // Do something`
	`4.  }`
	`5. (…)`

Rationale:	Students do not know how to construct a Boolean expression composed of a sequence of multiple statements. They do, however, the notion that the desired Boolean expression would comprise a sequence of statements that have some correlation among them. Therefore, students conclude that a loop iteration will somehow support this approach.
Consequences:	The Boolean expression will not be correctly evaluated and the program will not work properly. Also, as the loop contains logic errors, there is a chance the loop condition will always evaluate to true, leading to an infinite loop error.
Detection:	**Where:** o Whenever a Boolean expression with multiple statements must be evaluated within the code.
	How: o A logic problem is solved through a sequence of loop iterations instead of a Boolean expression.
Improvement:	Students should be oriented to understand how Boolean expressions are evaluated in C language, and also how to use the proper operators (`||`, `&&` and parenthesis) to build these expressions.

3. Conclusions and Future Work

We believe the documentation provided in this work will support the further development and assessment of a Concept Inventory (CI) for Introductory Programming Courses. Specifically, the next steps are: a) to replicate the pilot test study [4] at the Institute of Computing (Unicamp);[2] b) to map the misconceptions presented in this work to CI multiple-choice questions (each wrong answer would be related to a misconception); c) to create a website that allows instructors and researchers to provide the CI to computer science students, also using the data in this report to create personalized reports to all participants; d) to assess the CI's statistics internal consistency and; e) to adapt the CI to other programming languages (other than C), including Python and JAVA.

During the development of this work we identified some similarities among the misconceptions, specifically:

- D3, D4, and D6 are similar, describing issues related to improper initialization or stop condition of the loop counter, leading to a wrong control flow in a loop and failure in the program logic.
- A.2 and A.5 relate to the misconception that parameters are passed by reference in C language.

However, to keep consistency with previous works [2, 4], we opted in this Technical Report to preserve the original list of 33 misconceptions. In the next steps (replication of the pilot test study [4] at the Institute of Computing of Unicamp and misconception's mapping to CI multiple-choice questions) we will reanalyze and update the list of misconceptions.

We also believe the misconceptions detailed in this study can help researchers and instructors related to the Computer Science Education area to better understand and address the student's misconceptions on Introductory Programming Courses.

[2] This step is related to the MSc research of Tânia Alencar de Caldas.

4. Acknowledgments

This research is supported by grant #2014/07502-4, São Paulo Research Foundation (FAPESP). Additional support was provided by the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the National Counsel of Technological and Scientific Development (CNPq) and the University of Campinas (Unicamp).

We would like to thank Dr. Marco Aurélio Gerosa for the support and orientation on this research.

References

[1] Mohamed El-Attar and James Miller. 2006. Matching Antipatterns to Improve the Quality of Use Case Models. In *Proceedings of the 14th IEEE International Requirements Engineering Conference* (RE '06). IEEE Computer Society, Washington, DC, USA, 96-105. DOI=http://dx.doi.org/10.1109/RE.2006.42

[2] CACEFFO, R.; WOLFMAN, S.; BOOTH, K. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. *In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 364-369. DOI=http://dx.doi.org/10.1145/2839509.2844559

[3] G. L. Herman, M. C. Loui, and C. Zilles. Creating the digital logic concept inventory. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE '10, pages 102–106, New York, NY, USA, 2010. ACM

[4] CACEFFO, R.; WOLFMAN, S.; BOOTH, S.; AZEVEDO, R. A Pilot Test to Validate Misconceptions for Introductory Computer Programming. ***To be published.***

[5] LAZAR, J.; JINJUAN, H.; HOCHHEISER, H. (2010). Research Methods in Human-Computer. Interaction. Wiley Publishing.