

Understanding Notional Machines through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing

Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, Vladimir Lara-Villagrán

Universidad de Costa Rica, CITIC-ECCI

San José, Costa Rica, 2060

{jeisson.hidalgo, gabriela.marin, vladimir.lara}@ucr.ac.cr

Abstract

A correct understanding about how computers run code is mandatory in order to effectively learn to program. Lectures have historically been used in programming courses to teach how computers execute code, and students are assessed through traditional evaluation methods, such as exams. Constructivism learning theory objects to students' passiveness during lessons, and traditional quantitative methods for evaluating a complex cognitive process such as understanding. Constructivism proposes complimentary techniques, such as conceptual contraposition and colloquies. We enriched lectures of a "Programming II" (CS2) course combining conceptual contraposition with program memory tracing, then we evaluated students' understanding of programming concepts through colloquies. Results revealed that these techniques applied to the lecture are insufficient to help students develop satisfactory mental models of the C++ notional machine, and colloquies behaved as the most comprehensive traditional evaluations conducted in the course.

Keywords: programming learning, notional machine, lecture, constructivism, conceptual contraposition, cognitive dissonance, program memory tracing

1 INTRODUCTION

Unlike other types of texts, such as novels or scientific articles, source code of computer programs has a behavioral dualism. Source code is static when it is written in text editors, but it has a dynamic behavior when it is run by a computer. This dualism is obvious for experienced programmers, but not for programming learners [1]. A correct understanding of its dynamic behavior is mandatory for writing valid source code, and this ability is especially difficult for some students [1].

In order to understand the behavior of code at runtime, one needs an understanding how a computational machine works. It is common that some courses be included in undergraduate computing curricula in order to achieve this goal, such as digital circuits, computer architectures and assemblers [2]. Paradoxically these courses are usually not taught before the programming courses. However, mastery of the real machine is not mandatory in order to understand the dynamics of the programs at runtime. Programming languages, through its constructs, provide an abstraction of the real machine, Boulay et al. in 1981 called this concept a **notional machine** [3]. Each programming language provides a notional machine. For example, a C programmer could conceive that the machine has data types and is able to execute functions, but the actual machine does not have these constructs. A Java programmer would think that the machine is capable of running methods polymorphically, and that it has a garbage collector [4]. The right side of Fig 1 represents the relationship between the actual machine and the notional machine mediated through its programming language.

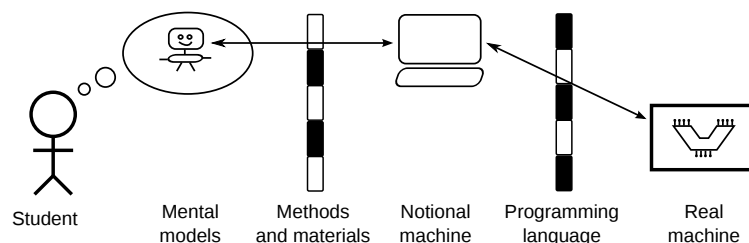


Fig 1 Process of creating mental models about notional machines

Students build their mental models about the notional machine through the teaching and learning methods and materials used in programming courses. The most commonly used teaching method around the world is the lecture [5], [6]. In its simplest form, the **lecture** is an oral speech performed by a teacher in front of a group of students [7]. Historically, teachers would read the material and students took notes, hence its name *lecture*. Over time, other activities and materials were incorporated in order to enrich the lecture, such as dialogues with students, illustrations on the blackboard, or projection of prepared slides.

Programming professors usually enrich lectures with other techniques in order to help students understand how notional machines execute programs. An example is the technique that Hertz and Jump called **program memory tracing** [8]. This technique consists in illustrating the distribution of the different program's concepts within the computer memory, and how they change while the program is executed [8]. We provide examples applying this technique in section 4.1. Hertz and Jump reported higher grades and higher motivation when program memory traces are produced by students [8]. But, no scientific evidence was found when program memory traces are produced by professors during traditional lectures, which is the most common scenario [5], [6]. In traditional instruction, professors commonly assess students' learning through written exams, and programming assignments. Fig 2 shows, in the left-most column, traditional instruction composed of the methods described previously.

There are concerns in the scientific community about the effectiveness of educational methods for programming learning, specially those used by traditional instruction [9]. Multinational statistics indicate that 67.7% of students pass their first programming courses [10], [11], and the majority of them do not know how to program [12]. The main objections are stated by other learning theories, such as constructivism. Traditional instruction is criticized mainly for students' passivity and lack of intrinsic motivation [13].

The authors of social constructivism suggest using several principles and techniques such as the ones listed in the middle column of Fig 2. In the same fashion that constructivism can adopt classical means of instruction such as lecturing and reading books [14], traditional instruction can be enriched with these constructivist techniques. This research applies the conceptual contraposition and the colloquy techniques to the traditional teaching. *Conceptual contraposition* is a technique to arouse students attention and interest to learn a new concept. *Colloquy* is an evaluation technique based in conversations that allows both, assessing and fostering, the learning process of students. We will further discuss these techniques in section 2.

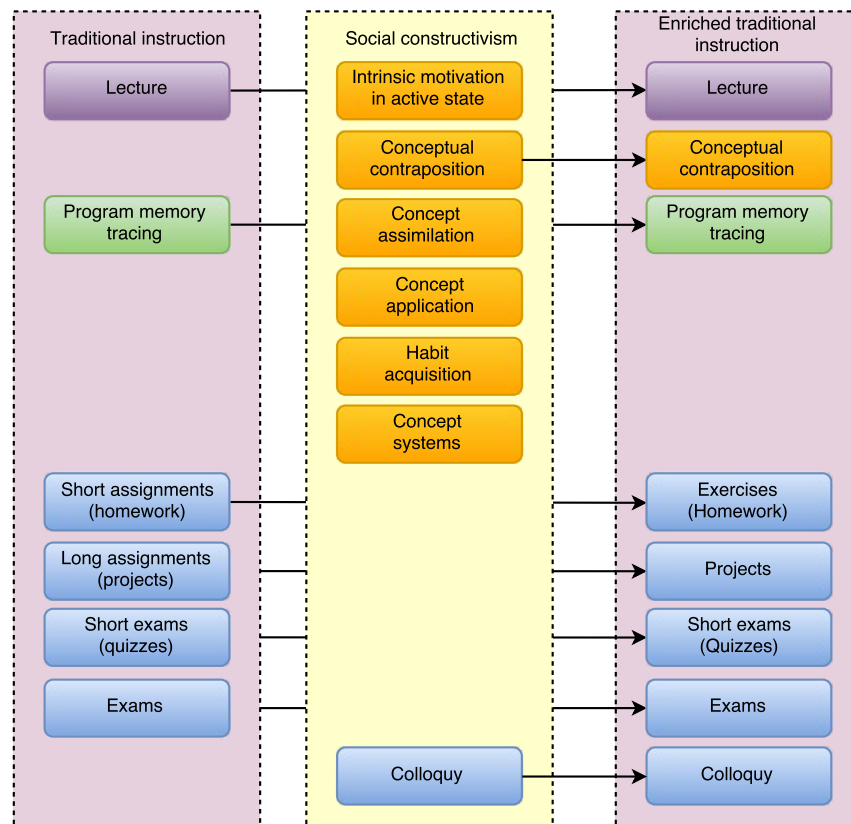


Fig 2. Teaching and evaluation methods used this research

A preliminary version of this work was presented in Spanish at the 2015 Latin American Computing Conference (CLEI 2015) [15]. This version has an extended discussion about theoretical background. A section about previous work includes a more systematic literature review. The methodology was extended and structured in two sections. Results were separated in qualitative and quantitative sections. Qualitative descriptions were slightly increased and scans were provided for all memory tracing drawings made by students. Quantitative results were merged in one table and explained deeper. Argumentation sections, i.e., discussion of results, conclusion, and future work, were significantly extended.

2 THEORETICAL BACKGROUND

A correct understanding about how computers run programs is required in order to effectively program them. Students acquire this understanding through the educative methods used in introductory program courses. The most used teaching method worldwide in programming courses is the lecture. Professors explain how programs are run using verbal explanations and static visualizations. However, this method is criticized by the Constructivism learning theory if students stay inactive during lectures. Constructivism claims that knowledge must be actively rebuilt, instead of passively absorbed by students [13], [14]. Social constructivism and lectures are not considered incompatible [14], and some constructivist techniques may be applied to increase the activeness of students' mind during lectures. Constructivism also objects to quantitative methods, such as exams, for evaluating a cognitive process, such as understanding. In this paper, we apply two constructivist techniques: conceptual contraposition for arousing an active state of students' mind during lectures, and colloquies for evaluating students' understanding of how programs are run by computers.

The notion of *conceptual contraposition* is suggested by social constructivism in order to increase students' intrinsic motivation and prepare their minds for the assimilation of new concepts [13]. The **conceptual contraposition** technique consists in challenging students by posing to them a situation or problem such that existing concepts in their minds are insufficient or contradictory to resolve it [13]. Students enter into a state of cognitive uncertainty by experiencing that their beliefs or knowledge are wrong or incomplete [13]. In order to overcome the uncertainty and achieve a state of mental equilibrium, students must inevitably reorganize old concepts and build new ones. This mind restructuring inherently occurs without dependence on external reinforcements [13]. The conceptual contraposition technique has received other names in the Western literature, as theory of *cognitive dissonance* or *cognitive conflict*. The latter term is used by Ma et al. [16]–[18].

Social constructivism also questions the use of exams as an evaluation method for complex cognitive processes. Exams provide late feedback about a finished product, instead of immediate feedback about the learning process. Social constructivism's authors propose qualitative methods to evaluate students' mental models [14]. One of these methods is the colloquy, which allows achieving both functions of evaluation: diagnostics and learning.

According to Vygotsky, learned concepts reach the level of **knowledge** if students construct systems of concepts that reflect the relationships between objects and phenomena in the real world, and these systems are stable over time [13]. In the programming learning context, students' mental models must reflect valid relationships of the notional machine's concepts and these mental models must be stored in the students' long-term memory. Otherwise, students will only associate sparse or incomplete concepts called **fragile knowledge**, and they will be able to explain only pieces of the code [19]. Assessments done immediately after finishing a subject will be prone to evaluate fragile knowledge, therefore, they should be deferred or repeated in time in order to evaluate actual knowledge [13].

The **colloquy** is a formative evaluation method used by social constructivism authors [13]. It consists of a dialogue between the professor and the student while the latter performs an educational task. We suppose that colloquies can be used to assess the two development levels stated by Vygotsky's learning theory. In a first phase of the colloquy the professor only investigates the level of skill that the student has in solving the task, called **effective development level** [13]. In a second phase, the professor asks the student to reattempt the task, especially where deficient mental models (wrong concept systems) were detected. The professor provides key questions, information, tools, and tips to help the student overcome the deficiencies (scaffolding). That is, the professor helps the student understand or learn concepts that are in his or her zone of proximal development. The **zone of proximal development** is a metaphorical representation about the set of all concepts that a student may learn in the company of a more skilled collaborator (adapted from [13], [20], [21]). Afterwards the professor evaluates the **potential development level** that the student has reached with this aid [13]. The first phase is oriented to the evaluation, the second to student's learning.

3 PREVIOUS WORK

Constructivism has been suggested as the learning theory to face the fast technological changes in Computer Science education [22]. However, studies about constructivism in Computer Science are rather reduced compared with studies in related fields such as Science and Mathematics [14]. Studies about the application of conceptual

contraposition or colloquies to the teaching and learning of computer programming are even scarcer. As our previous work, we considered all relevant papers found in ACM Digital Library, IEEE Xplore and Scopus, that resulted from the following query string:

```
("cognitive dissonance" OR "cognitive conflic" OR "think aloud") AND programming AND (teach* OR learn*)
```

Although, the term *conceptual contraposition* is not found in the computer education scientific literature, a few results appear as synonyms. An abstract-only publication by Kearney and Nodine reports an experimental comparison of traditional lectures from a Fortran course against assignments with no formal instruction (i.e. *cognitive dissonance*) [23]. They found both methods as effective, but students reached a deeper understanding of Fortran using cognitive dissonance [23]. However, they did not study both methods applied together.

Ma et al. found positive results on student mental models when *cognitive conflict* was applied before program visualizations [16]–[18]. However, they did not apply the conceptual contraposition to the lecture, which is still the prevalent method of education around the world [5], [6].

Constructivism states that qualitative evaluations, such as colloquies, must be conducted in order to assess complex cognitive processes, such as understanding [13], [14]. Although the term “colloquy” was not found in programming learning literature, related methods can be found. Colloquies may be conceived as *interview* sessions where students are asked to perform a task while “*thinking aloud*”. During the interview, the professor observes, takes notes (*assessment*), and may provide some clues or questions when the student experiences problems or impasses (*scaffolding*).

Whalley and Kasto used *think-aloud* sessions to follow volunteer students up from an introductory programming course in Java [21]. Students were asked to solve programming problems that required processes of knowledge restructuration (*accommodation* in Piaget’s nomenclature). Students’ potential development levels were analyzed and classified using the theory of Perkins et al. about type of learners: stoppers, tinkerers and movers [21]. However, in their study, results were obtained directly from student’s performance in think-aloud sessions, and were not related to the teaching-learning methods or other evaluations used in the course.

Teague and Lister have conducted several think-aloud studies with introductory programming students. They used think-aloud protocols for discovering misconceptions of students resolving a simple assignment exercise [24], resolving a complex exercise about reversing the effect of a given program [25], and characterizing the stages of the learning process experienced by a student [26]. Their results are analyzed in light of the neo-Piagetian theory, but they did not relate these results to the teaching methods, such as lectures.

Arshad found positive opinions when think-aloud protocols were used by teaching assistants or expert programmers explaining, in front of the class, how they resolve a given programming problem [27]. However, this scheme is distinct to that proposed in this paper. We use think-aloud protocols as an assessment method instead of a teaching method.

Think-aloud protocols are mainly found in software engineering scenarios, where participants already know to program software applications. For example, Wiedenbeck and Engebretson conducted think-aloud protocols in order to know how secondary school teachers understand an existing application’s code that they must modify later [28]. Because our research is targeted to introductory programming students, these studies were out of scope.

In summary, we found evidence of conceptual contraposition applied to programming assignments or program visualizations, but not lectures. In the same fashion, we found think-aloud protocols used for teaching or evaluating students’ performance, but not related to the effect of lectures or other teaching methods.

4 METHODOLOGY

The treatment proposed in this paper was applied in two stages. First, each programming concept that has an effect in the C/C++ notional machine was introduced using conceptual contraposition along the course duration. Second, students’ understanding of the C/C++ notional machine was evaluated using colloquies at the end of the course for extra credit. Both stages are explained in the following subsections.

4.1 Conceptual contraposition

The conceptual contraposition was applied to a Programming II course (sometimes called CS2 in scientific literature). This course belongs to the Computer Science undergraduate program at the University of Costa Rica. The first author of this article taught the course from August to November 2014. By convention, the programming language used in this course is C++. The teaching method used for this course was the “enriched” lecture depicted in the third column of Fig 2. During the semester, each programming concept that has impact on the notional machine was introduced using conceptual contraposition and explained using program memory tracing. In addition to regular course’s assessments, the evaluation of students’ understanding of notional machine concepts was conducted at the end of the course using colloquies between the professor and each student who voluntarily participated.

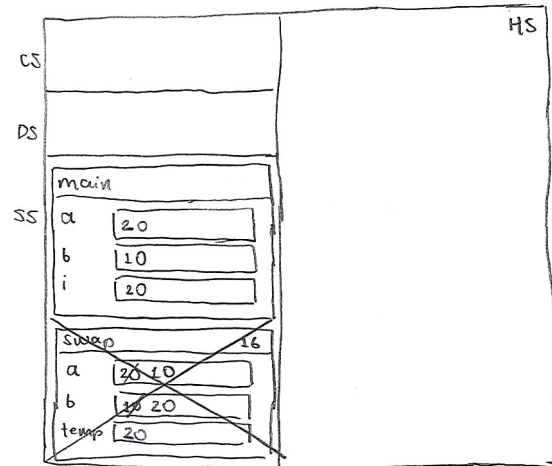
The conceptual contraposition technique was applied during lectures to introduce programming concepts that have effect on the C/C++ notional machine. For example, as a prelude to introduce the concept of pointer, the professor wrote the program in Fig 3(a) in the computer used for projection. The program prints all integer values between two numbers given by a user. If the user inputs an inverted range, the program is supposed to swap them and to continue as usual. This program will be referred as the “range program” in this document.

```

01 #include <iostream>
02
03 void swap(long a, long b)
04 {
05     long temp = a;
06     a = b;
07     b = temp;
08 }
09
10 int main()
11 {
12     long a, b;
13     std::cin >> a >> b;
14
15     if ( a > b )
16         swap(a, b);
17
18     for ( long i = a; i <= b; ++i )
19         std::cout << i << ' ';
20
21     std::cout << std::endl;
22 }

```

(a)



(b)

Fig 3. The “range program”, a C++ program that prints all integers within a given range:
(a) its source code, (b) professor’s program memory trace before finishing its execution (line 22)

The professor asked students about their predictions about the program. The program seemed perfect according to students. The professor ran the program on the computer and projected it in front of the class (Fig 4). The professor entered a valid range (10 20, underlined in Fig 4), and the program printed the expected values. Then, the professor executed the program again and entered an inverted range (20 10 in Fig 4). The program should have printed the same result, but its output was empty. Staring at the program that seemed correct, but was not, should foster a state of mental uncertainty, necessary for learning the new concept, according to the theory of social constructivism [13].

```

$ ./range
10 20
10 11 12 13 14 15 16 17 18 19 20
$ ./range
20 10
$

```

Fig 4. Two running examples of the “range program” on Unix. The \$ prompt indicates the operating system is waiting for commands. Underlined text is inputted by user. The remaining text is program output.

The professor explained the program behavior at runtime using the *program memory tracing* technique. This technique consists in traversing a program line by line reflecting the effect of each one on an abstract drawing [8]. The drawing (Fig 3(b)) represents the state of the program distributed among the memory segments of the notional machine [8]. The C/C++ notional machine is very close to the underlying architecture, with at least four segments: code segment (CS), data segment (DS), stack segment (SS) and heap segment (HS). Segments are represented in Fig 3(b) as rectangular regions. Interesting actions in the “range program” occur on the stack segment (SS).

The professor acted as the processor running each line, updating the memory drawing on the blackboard, and verbally explaining each effect in order to make the error evident. In the case of the “range program” (Fig 3(a)), the `swap` function receives two numbers as parameters by value. The function receives them in this way because parameter-by-value is the concept already present in the minds of students before introducing the concept of pointer.

After observing that the program failed and the cause, students are still in mental state of uncertainty and require knowing how to correct it. The professor, then, introduced the new concept, in this case, the pointer and its syntax. He corrected the program (Fig 5(a)) and executed it manually updating the program memory trace with the effect of the pointers (Fig 5(b)). Finally, he ran the program on the projection computer, which produced the expected result.

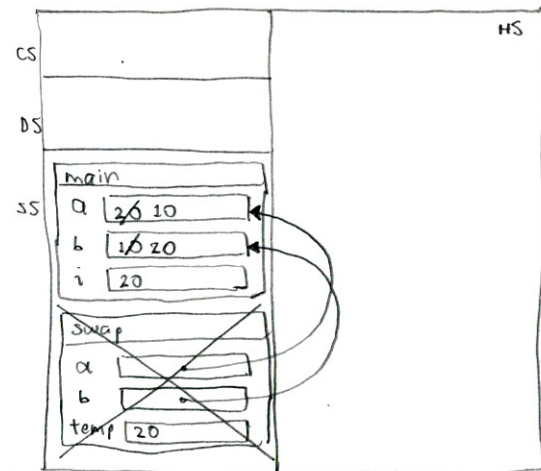
The described method, i.e., introducing a concept after a conceptual contraposition and its tracing on the notional machine, was used for each concept that had an effect on the program state. The underlining concern is: have students built correct mental models of the C/C++ notional machine with this method?

```

01 #include <iostream>
02
03 void swap(long* a, long* b)
04 {
05     long temp = *a;
06     *a = *b;
07     *b = temp;
08 }
09
10 int main()
11 {
12     long a, b;
13     std::cin >> a >> b;
14
15     if ( a > b )
16         swap(&a, &b);
17
18     for ( long I = a; I <= b; ++I )
19         std::cout << I << ' ';
20
21     std::cout << std::endl;
22 }

```

(a)



(b)

Fig 5. Range program fixed using pointers: (a) its source code, (b) professor's program memory trace before finishing its execution (line 22)

4.2 Colloquies

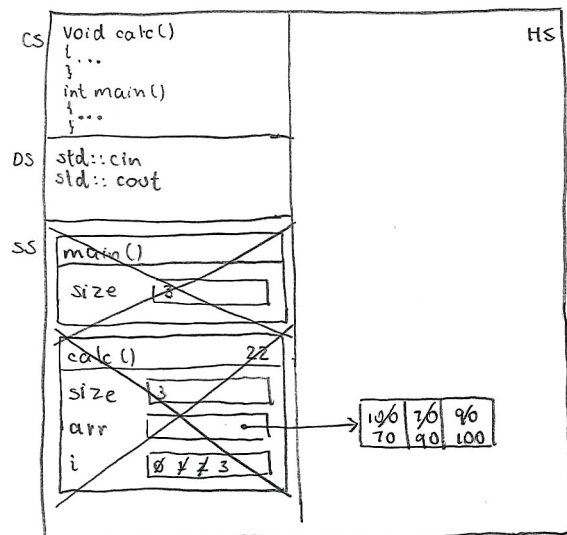
The chosen educational task for mental model evaluation was explaining how the C/C++ notional machine executes the “median program” show in Fig 6(a) line by line. A blank paper sheet was provided to each participant and he was asked to draw the program state as he was explaining its execution. An example of a memory trace drawing made by the professor is shown in Fig 6(b). Colloquies were recorded only on audio format. There were no time restrictions. Students had access to documentation of the algorithm `std::sort()` during the session.

```

01 #include <algorithm>
02 #include <iostream>
03
04 void calc(size_t size)
05 {
06     double* arr = new double[size];
07
08     for ( size_t i = 0; i < size; ++i )
09         std::cin >> arr[i];
10
11     std::sort(arr, arr + size);
12     if ( size % 2 == 0 )
13         std::cout << (arr[size/2] + arr[size/2 - 1]) / 2;
14     else
15         std::cout << arr[size / 2];
16 }
17
18 int main()
19 {
20     size_t size;
21     while ( std::cin >> size )
22         calc(size);
23
24     return 0;
25 }

```

(a)



(b)

Fig 6. The “median program”: a C++ program used to evaluate mental models; (a) its source code, (b) professor's program memory trace before finishing its execution (line 25)

The C++ program in Fig 6(a) calculates the median of sets of data entered in the standard input. The size is entered before the data set. Students were asked to use the values (3, 100, 70, 90, EOF) as standard input. The value 3 indicates that the data set has three values. The size for the second data set is the special end-of-file (EOF) mark. The “median program” reacts ending its execution when the EOF mark is entered. Every time the `calc()` method is called,

an array is created in dynamic memory (line 06), but it is not deleted, therefore a memory leak is generated. The only statements in this paragraph that were communicated to the students were the input data values.

Not all programming concepts covered in the course were assessed during the colloquies. We only evaluated the concepts involved in the “median program” in Fig 6(a). These concepts are listed below, along with a description of the expected actions that students must take for tracing them.

1. *Memory segmentation*: Students must distribute the program’s memory into four segments; each one represented as a rectangular area in the drawing. Students must name them, and state their purpose.
2. *Program entry point*: Students must begin the program execution at `main()` function on line 18, and not at another line of the program.
3. *Function call*: Students must represent the invocation of both functions, `main()` and `calc()` as rectangles (stack frames) in the stack segment.
4. *Local variable*: Students must represent local variables within their function calls and not elsewhere. Variables must have a name and a value in the drawing.
5. *While loop*: Students must indicate that the loop body (the invocation of the function `calc()` on line 22) is called repeatedly until the loop condition becomes false (the user enters EOF, line 21).
6. *Standard input*: Students must indicate that the program expects a value from the standard input or keyboard in line 21 and 09. When a value is entered, it must be drawn on the destination memory space.
7. *Function parameter*: Students must represent parameters as local variables, and initialize them with the values provided when the function is called. It happens for lines 22 and 04.
8. *Dynamic memory allocation*: Students must draw space for variables in the heap segment when the `new` operator is executed on line 06.
9. *Array or vector*: Students must draw the exact number of continuous elements (3 when line 06 is executed).
10. *Pointer*: Students must draw pointers as integer variables, and connect them with the pointed data using arrows or hypothetical memory address values. In the case of line 06, the pointer must be represented as a local variable within the `calc()` function call.
11. *For loop*: Students must perform steps in order: initialization, condition, body, and increment; and repeat the last three in order until the condition becomes false. It is also valid to interpret the cycle at a higher level of abstraction. For example, students could explain the lines 08 and 09 as "here all array elements are read from standard input".
12. *Library function*: Students must indicate the purpose of the function and its effect on the program memory. Line 11 calls `std::sort()`, a function that is not defined within the program. This function is defined by the C++ standard library. The function was studied during course lessons, and students had access to its official documentation during the colloquy session. Students must state this function sorts the values of the array.
13. *Iterator*: Students must indicate that algorithms from C++ standard library work on subsets of data containers, and these subsets are delimited by iterators. An iterator is an object able to traverse the elements of a data structure, even if the elements are not continuous in memory. The iterator interface mimics a C/C++ pointer. The `std::sort(begin,end)` function receives two iterators `begin` and `end`, and sorts the elements in the range `[begin, end[`.
14. *Pointer arithmetic*: Students must find the pointed memory by the result of a pointer arithmetic expression. Line 11 adds an integer to a pointer (`arr + size`) to get a pointer to a nonexistent item (immediately after the last valid element), in order to build an iterator to the end of the array.
15. *If/else conditional statement*: Students must evaluate the condition of line 12. If the condition is true, they must run line 13 and not line 15. If the condition evaluates to false, they should ignore line 13 and run line 15.
16. *Expression evaluation*: Students must perform arithmetic or Boolean expressions according to the priority of operators, and get a unique value result of the evaluation.
17. *Array indexing*: When students run the line 13 or 15, they must evaluate an arithmetic expression and use the integer result to access an array element.

18. *Standard output*: Students must write or verbalize the value printed on the standard output (by default, the screen).
19. *Function return*: When students run line 16 or 25, they must indicate that the function ends its invocation, release its memory (stack frame) from the stack segment, and continue the execution where the function was invoked.
20. *Memory leak*: When students return from `calc()` function on line 16, they must remove the `arr` pointer memory but not the pointed memory. They must discover that a memory leak is generated, and indicate its consequences.
21. *Memory de-allocation*: If the student discovers the memory leak, the teacher asks the student how to correct it. The student must alter the program to add an invocation to operator `delete []`.
22. *Integer division*: Students must distinguish between integer division and floating point division. They must indicate the result of the integer division (`/`) or the module (`%`) according to the operator used. This concept was initially considered part of concept 16, but it was separated during data analysis.

Some programming concepts were not included in the above list because they do not have effect in the notional machine or they are already included in previous ones. Some examples are: increasing integer variable, namespaces, the `size_t` data type re-definition, and inclusion of header files.

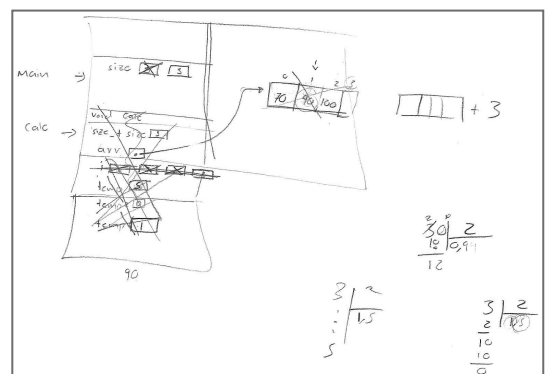
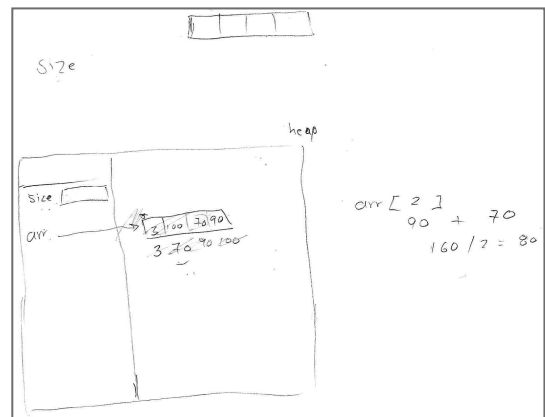
5 RESULTS

Courses in our Computer Science undergraduate program are offered in groups. A maximum of 20 students per group is allowed. The “enriched” lecture was applied to a group of students in the Programming II course in the second semester, 2014. From the 18 students enrolled in the group, 13 were active at the end of the semester (5 dropped the course at some point), and 11 participated voluntarily in colloquies for additional credit. Since the colloquy is a qualitative data collection method, the following subsection presents qualitative findings related to mental models of the notional machine. Then, a quantitative interpretation of these findings is presented in the second subsection.

5.1 Qualitative results

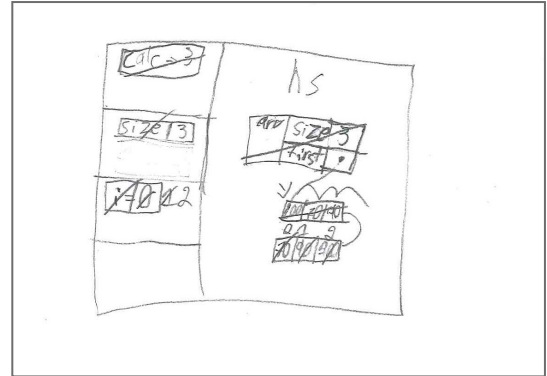
For each participant, the following list includes a brief description of the mental models for some programming concepts. It also includes a scanned image of the program memory trace made by each participant.

1. **Participant 1** distinguished the heap segment but not other segments. He placed two local variables within an unnamed segment, presumably the stack frame for `main()`, but he did not distinguish function calls at all. He reused the `main`'s `size` local variable instead of creating a `size` parameter for `calc()` function call. He added a copy of the local variable `size` as first element of the array on the heap, which led to wrong decisions and wrong program output even with scaffolding. He conceived the module 2 (`size % 2 == 0` in line 12) as asking if the dividend is even, which is a high level interpretation of the expression. He abruptly ended the program execution from line 16. He indicated that the machine automatically cleans the memory. He detected and corrected memory leak only with scaffolding.
2. **Participant 2** did not recall the names of memory segments, but he had a clear idea of usage of three of them. He clearly explained the function call and parameter passing process. He confused the `size` and capacity of a floating-point array with a C-style string, so he added space for another element but he did not use it. He considered pointer arithmetic as a sum of incompatible types and made wrong assumptions. He dedicated long time guessing the internal implementation of `std::sort`. He required a lot of

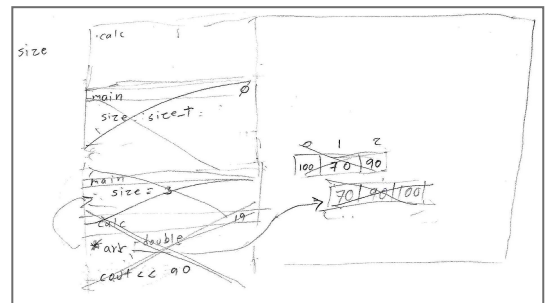


scaffolding in order to abstract the `std::sort` functionality. He made floating-point divisions ($3/2=1.5$) rather than integer divisions ($3/2=1$), and he was confused using this floating-point result to subscript the array (line 15). He managed to output the correct result through wrong assumptions. He stored temporary values in the stack segment for intermediate expression evaluations, but the notional machine does not store these values. He successfully detected and corrected memory leak.

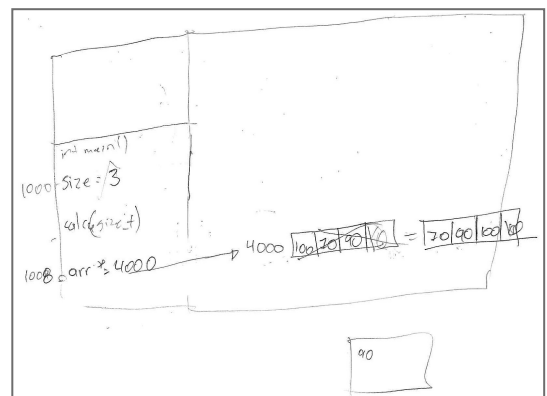
3. **Participant 3** only remembered the name for the dynamic memory segment. For the remaining segments, he explained their usage properly. He represented the `calc()` function call as a pointer to the value of `size` parameter, instead of a stack frame. He idealized both, the pointer and the array, as members of a container class, and located them in dynamic memory. He considered that the program causes an error when trying to access the element at the `arr + size` (line 11). He doubted if the integer division discards or rounds decimals. He favored rounding, so he managed to print the correct result. He identified the memory leak. He considered that `delete[]` operator must be called within a loop in order to delete all array elements. He finished the program execution correctly.



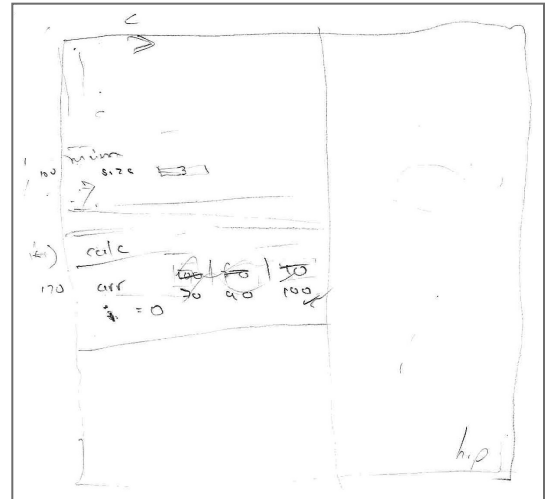
4. **Participant 4** indicated that the machine separates memory in subsections or spaces, and he only remembered the name of one of them: the "stack". He separated the dynamic memory from the stack. He considered the `size` variable (line 20) as global to the entire program. He duplicated the `main()` stack frame because "the `size` variable was twice, one time declared and another initialized", reflecting confusion with the code segment. He assumed that `calc()` can access a local variable from `main()`. He considered that a pointer is the same as an array. He evaluated pointer arithmetic correctly, but he considered that the program should fail when invoking `std::sort`, reflecting misconceptions about iterators. He correctly evaluated whole divisions and printed the correct value on the screen. He identified the memory leak, but tried to correct it with the C library function `free()`. He required a lot of scaffolding to remember the operator `delete[]`.



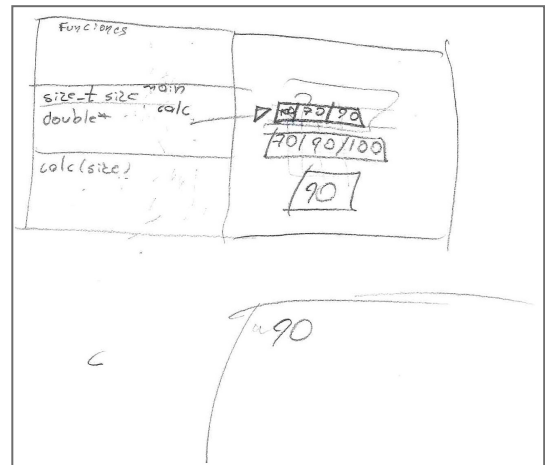
5. **Participant 5** did not recall names of the segments, but he indicated "there are three parts": "where dynamic things are", "where methods run", and "where global variables are saved". Although he verbally explained the mechanics of function calls, he did not reflect it in his drawing (stack frames). He separated between pointer and the pointed array correctly. He explained pointers drawing arrows, hypothetical memory addresses, and sizes of data types, reflecting mastery of these concepts. He incorrectly added the null string terminator ('\0') to the array of floating-point numbers. He explained the reading of array elements at high level of abstraction. He had serious difficulties doing the whole division $3/2$ and using the result for indexing the array. He said that the program should not compile. With scaffolding he argued that the program must fail at runtime. He did not eliminate the stack frame when executed line 16. He did not detect the memory leak. He finished the program execution but he was unable to explain how the while loop breaks on line 21; he required scaffolding to remember it.



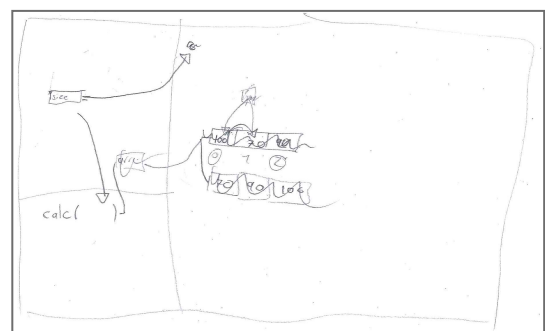
6. **Participant 6** sliced the program memory in two segments. He stated the first segment is intended for operating system resources (he labeled it as "hip", referring to the heap segment). The second segment is "where functions are". He placed all program variables in the latter segment because "I do not see [this program] use dynamic memory at all". He expressed he does not know how data from the keyboard arrive to the variable when it is read from standard input. He separated functions calls (stack frames) with their local variables, but omitted parameters. He did not distinguish between pointer and array; he drew them as a single entity. He explained array sorting at high level thinking that `std::sort` receives indexes instead of iterators. He evaluated correctly whole divisions, and printed the central element correctly. He could not explain how `calc()` ended its call and did not reflect it on his program trace. He believed that the C++ standard states that the program memory must be automatically released.

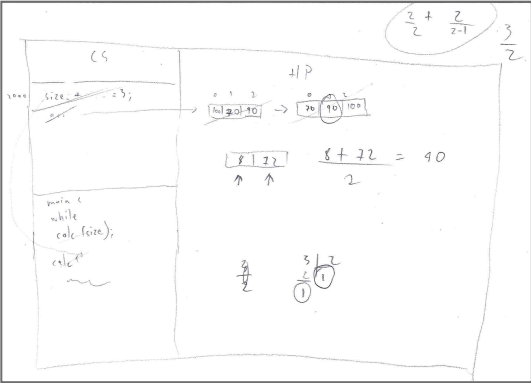
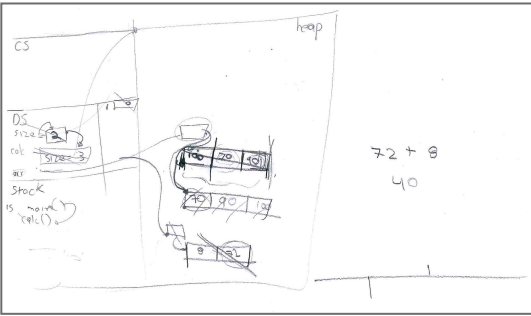
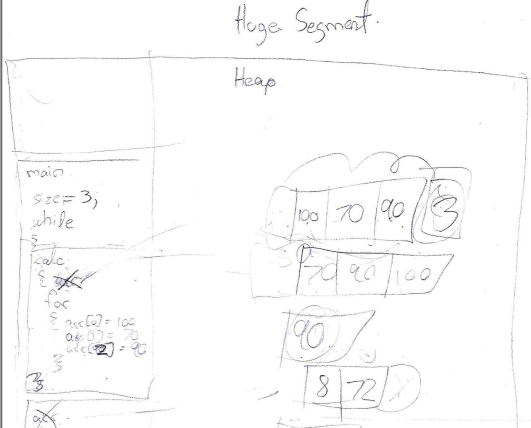


7. **Participant 7** separated three segments of memory: "the data segment that holds the local variables", another for functions but he could not remember its name, and he separated the "system memory" from the previous two, and he was unable to explain why. He created a pointer and a stack frame, both called `calc`, but he was unable to explain the difference. He did not read the `size` variable on line 21, and he could not explain how the `calc()` function got the value for `size` from standard input. He built the array within the "system memory" segment when he run the `new` operator, but he verbally explained it as an array of pointers to `double`, then as a matrix, finally as an array of floating-point values with some scaffolding. He explained the operation of `std::sort` at high level. He calculated whole divisions flawless. He printed the expected value to the standard output. He was very confused about how control breaks the cycle on line 21 in the first iteration; however, when the input was the `EOF` character, he recalled it spontaneously. He required scaffolding to detect the memory leak.



8. **Participant 8** separated between local variables, function calls, and dynamic memory. However, he also separated local variables from their values, placing values in the dynamic memory indicating that the variables point to their values. He filled the array with counter values instead of values read from standard input. He required scaffolding to notice it. He tried to explain the internal implementation of the `std::sort` algorithm, but then he abstracted its effect on the array. He hesitated how the module and integer division works. He could not explain how the computer indexes the array from a real number (the result of $3/2$, on line 15). He assumed that the computer takes the integer part of the division, then he hesitated, and he was unable to explain why the module $3\%2$ generates an integer value but not the division $3/2$. He stated that the computer automatically removes the dynamic memory.



9. **Participant 9** correctly distinguished between code segment and dynamic memory. However, he separated stack segment (not recalled its name) into two, one for variables and another for "method calls and pointers". He tried to start executing the code from line 1, and he required minimum scaffolding to discover the `main()` function. He mixed instructions and function calls in a segment. He explained `std::sort` properly at high level. He initially interpreted the module 2 (`size % 2 == 0` in line 12) as equivalent to asking if a number is even. Then he hesitated and said that the program should crash at runtime when attempting to access the nonexistent array index $3/2$. With scaffolding, he said that the program should print spurious values. He stated that vectors in dynamic memory are automatically destroyed. He abruptly ended the program execution from line 13. He printed incorrect results in standard output. He required explanation of whole division to print the correct result.
- 
10. **Participant 10** correctly recalled the four memory segments and their functionality. But he placed local variables in the data segment, and separated them from functions calls. He confused integer variable values with pointers and initialized them with the memory address 0. He conceived that the value of a pointer is another pointer stored in dynamic memory, and the value of the latter is the data array. He estimated sizes of variables and structures along his verbal discourse. He interpreted `std::sort` as "scrambling the data" until he read the official documentation. He hesitated about module and the whole division, although he evaluated the operations properly. He indexed the array starting at 1 and required scaffolding to realize it. He did not discover the memory leak; with scaffolding, he stated that dynamic memory is automatically released.
- 
11. **Participant 11** correctly recalled the purpose of three segments (omitted the code segment). He correctly interleaved local variables with function calls. However, he also included some code instructions in stack segment. He needed scaffolding to discover and start running from `main()` function. He doubted if the `>>` operator is overloaded by `std::cin` for reading variables of type `size_t`, demonstrating mastery of operator overloading. He did not create a local variable for the size parameter when called `calc()`. He copied the array values from dynamic memory into stack segment. He interpreted the expression `arr+size` as a sum of integers, instead of pointer arithmetic. After reading the `std::sort` documentation, he correctly inferred the range of values to be sorted. He evaluated some arithmetic expressions wrong, but did modules and entire divisions correctly. After requesting him to re-evaluate them, he did it correctly. He finished executing `calc()` without eliminating its stack frame; with minimum scaffolding, he did it correctly. He identified the memory leak, and properly fixed it.
- 

5.2 Quantitative results

A subjective summative evaluation prepared by the professor from colloquies is presented in Table 1. The student's understanding of each concept was graded with a real value between 0 and 1, where 1 indicates evidence of mastery of the concept and 0 absence of evidence of mastery. The color of each cell reflects its value in a range from red (0) to green (1). Blank cells correspond to values that could not be retrieved from audio recordings.

Each row in Table 1 contains the level of understanding of a programming concept. The Avg column lists the average understanding for each concept. At the bottom, the `colloq` row shows the average of each student, and the `Passed` row indicates if the student passed the course or not. Rows are in descending order by the average level of

understanding (i.e. Avg column). The concepts at the top of the table were more understood by students than concepts at the bottom. Concepts were grouped into three categories listed in column Acc. These groups indicate that students have:

- Excellent mental models of 7 evaluated concepts (31.8%): conditionals, array indexing, standard output, program entry point, for-loop, expression evaluation, and library function. None of these concepts is introduced in the Programming II course.
- Acceptable mental models of 6 evaluated concepts (27.3%): while loop, standard input, dynamic memory allocation, pointers*, pointer arithmetic*, and arrays. The two concepts marked with asterisks are introduced in the Programming II course.
- Deficient mental models of 9 evaluated concepts (40.9%): function return, function call, integer division, memory segmentation*, local variables, memory leak*, dynamic memory de-allocation*, function parameters, and iterators*. The four concepts marked with asterisks are introduced in the Programming II course.

Table 1. The results of professor's subjective grading for each concept

Concept	Participant											Avg	Acc
	1	2	3	4	5	6	7	8	9	10	11		
15 if	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,00	31,8%
17 arr[i]	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,00	
18 cout	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,00	
2 main	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	1,0	0,9	0,98	
11 for	1,0	0,8	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,98	
16 expr	1,0	0,7	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,9	0,96	
12 sort	1,0	0,9	0,9	1,0	0,9	0,9	1,0	0,8	1,0	0,9	1,0	0,94	
5 while	0,0	1,0	1,0	1,0	0,9	1,0	1,0	0,9	1,0	1,0	1,0	0,89	27,3%
6 cin	1,0	1,0	1,0	1,0	0,7	0,9	0,0	0,7	0,7	0,9	1,0	0,81	
8 new	1,0	1,0	0,5	0,8	1,0	0,0	1,0	0,9	1,0	0,7	1,0	0,81	
10 ptr	1,0	1,0	0,3	0,7	1,0	0,2	0,9	1,0	1,0	0,6	1,0	0,79	
14 ptr+i		0,1	1,0	1,0	1,0	0,3		0,5		1,0	0,9	0,73	
9 arr	0,6	0,7	0,3	0,7	0,7	0,4	0,5	1,0	1,0	1,0	0,8	0,70	
19 return	0,4	1,0	1,0	1,0	0,4	0,2	0,2	0,2	0,5	0,9	0,8	0,60	
3 call	0,0	1,0	0,1	0,5	0,8	1,0	0,8	0,4	0,2	0,6	1,0	0,58	40,9%
22 /%	1,0	0,0	0,2	1,0	0,1	1,0	1,0	0,1	0,0	0,9	0,9	0,56	
1 seg	0,3	0,7	0,4	0,3	0,6	0,3	0,2	0,4	0,6	0,8	0,8	0,49	
4 auto	0,2	0,7	0,5	0,3	0,7	0,7	0,2	0,1	0,7	0,3	0,9	0,48	
20 leak	0,4	1,0	1,0	1,0	0,0	0,0	0,1	0,0	0,0	0,0	1,0	0,41	
21 delete	0,7	1,0	0,7	0,8	0,0	0,0	0,1	0,0	0,0	0,0	1,0	0,39	
7 arg	0,0	1,0	0,0	0,1	0,2	0,0	0,1	0,1	0,0	0,8	0,1	0,22	
13 itr		0,0	0,0	0,0	0,0	0,0				0,0	0,6	0,09	
Colloq	0,68	0,80	0,68	0,78	0,68	0,59	0,66	0,63	0,68	0,75	0,89	0,70	
Passed	no	yes	no	no	yes	yes	yes	no	yes	yes	yes		

The "Programming II" course is a continuation of "Programming I" course. Concepts learned in "Programming I" are reused in "Programming II" (i.e. they are not introduced again). Fig 7 depicts the 22 evaluated concepts in terms of the course in which they are introduced. Concepts introduced in "Programming I" have blue background, and concepts introduced in "Programming II" have red background. Concepts are classified on the y-axis by the level of understanding that students had of them in colloquies.

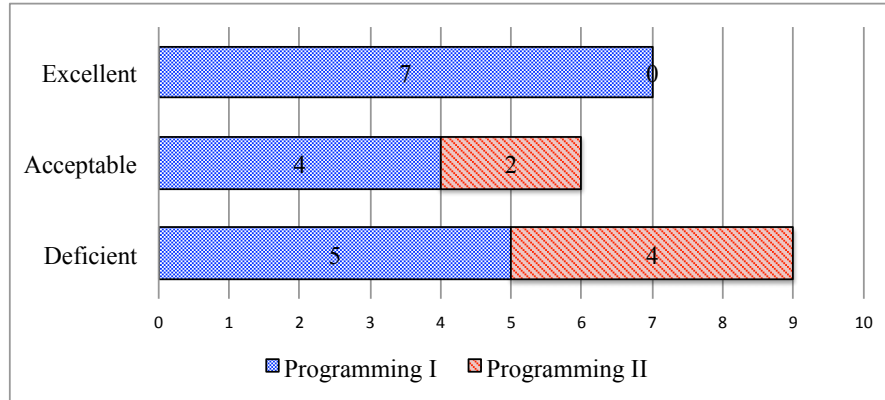
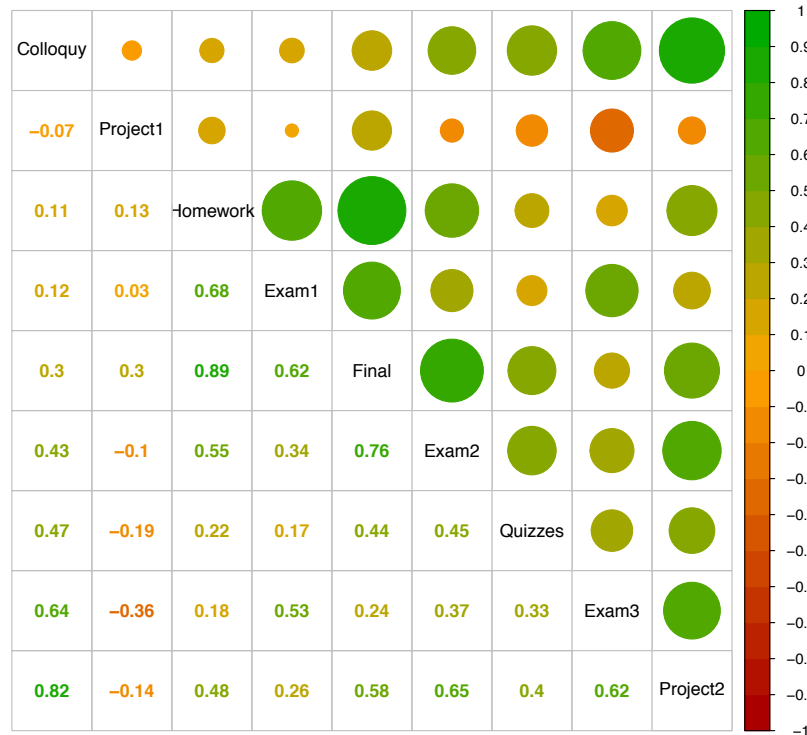


Fig 7. Level of understanding of the concepts introduced by course

Colloquies are suggested as an evaluation method by the constructivism learning theory [13], but they have been rarely used in introductory programming courses (as think-aloud protocols), and never as formal evaluation methods. We were interested in knowing if colloquies have potential to behave as some traditional evaluation methods. The quantitative data acquired from the 11 colloquies were compared against the grades that the same 11 participants obtained in the traditional evaluations conducted in the “Programming II” course.

Table 2 is the correlation matrix of the nine evaluations conducted during the course: colloquies (Colloquy), two long programming assignments (Project1, Project2), short programming assignments (Homework), three partial exams (Exam1, Exam2, Exam3), short exams (Quizzes), and course’s final grade (Final). The matrix is symmetric. Evaluation method names are stored in the diagonal. Correlation values are explicit in the lower triangular matrix, and proportionally sized circles in the upper triangular matrix. A cell is the correlation between the method located in its row and the method located in its column. Evaluation methods are sorted by correlation with colloquies. First project and homework assignments had the lowest correlations. Highest correlations show that colloquies behaved as the third exam or second project. This is an interesting result because these traditional evaluations are conducted near the end of the course, and they demand the highest levels of comprehension of programming concepts. These results will be interpreted in the next section.

Table 2. Correlation matrix of the conducted evaluation methods



6 DISCUSSION

The quantitative results in Table 1 show that students developed correct mental models for approximately one third of the concepts evaluated in colloquies (31,8% in column *Acc*). All of these concepts were introduced in the “Programming I” course, which is taught using Java as the programming language in our university (top bar in Fig 7). These concepts were reused and reinforced in the “Programming II” course (with C++). The traditional teaching method (the lecture) enriched with conceptual contraposition and program memory tracing seems appropriate for this reinforcement; but 40.9% of the evaluated concepts reflected incorrect mental models (the third category in column *Acc* in Table 1). Half of the concepts in the third category are also introduced in the “Programming I” course (bottom bar in Fig 7). That is, students have preserved incorrect mental models throughout two programming courses taught with lectures. A notorious example is the concept of integer division eclipsed by a generalization of the floating-point division. Approximately half of the participants experienced this problem, and suffered severe anxiety states because they were unable to resolve or explain a notional machine’s phenomenon that they considered elemental.

None of the evaluated programming concepts that were introduced in “Programming II” course had an excellent level of student’s understanding (top bar in Fig 7). These concepts are essential to achieve the objectives of the Programming II course, and are required by subsequent courses in our Computer Science undergraduate program. The conclusion from these results is that lectures enriched with conceptual contraposition and program memory tracing are insufficient to ensure proper construction of notional machine mental models by apprentices. This result is of importance when considering that lectures maintain their prevalence as a teaching method for computer programming.

Previous studies found positive results of these techniques when combined with other methods and materials; particularly conceptual contraposition combined with program visualizations [16]–[18], and program memory traces combined with active learning [8]. The differences between their results and ours may suggest that lectures are the main cause for students’ lack of understanding of notional machine concepts.

Excepting participant 11, qualitative results revealed that all students had serious deficiencies with at least three of the evaluated programming concepts. All students’ explanations lacked terminological precision and conceptual clarity. Most students had problems dividing the state of the program in memory segments (concept 1); the mechanics of calling, passing parameters and returning from functions (concepts 3, 19, 7); storing local variables (concept 4); and managing dynamic memory (concepts 20, 21). Comparing these results against the final course grades, six of these ten students passed the “Programming II” course. These results agree with the conclusion of McCracken et al., i.e. many students pass their first programming courses without a proficient level of programming [12]. Further research is required to determine if traditional instruction methods or traditional evaluation schemes are the main cause of this phenomenon.

Qualitative data also revealed that very few students (4 out 11) detected and corrected the memory leak in the “median program”. This is an advanced concept that must be compulsorily learned in order to program C/C++ effectively. We included this concept in our evaluation because we know our students have serious difficulties to grasp it, therefore we consider it is a good candidate to judge whether a teaching method is effective or not. A hypothesis that may be explored in future research is if memory leak understanding is negatively affected when C/C++ students have previously learned Java programming. The underlining rationale is the following. The creation of objects shares almost the same logic and syntax in Java and C++, but their deallocation is different. Java deallocates dynamic objects automatically using a garbage collector service. C++ delegates the responsibility to the programmer, but keeps quiet if programmer forgets doing it. This scenario encourages that students keep applying the object creation habit developed in the Java course (assimilation in Piaget’s nomenclature), inhibiting the learning process of C++’s dynamic memory deallocation system (accommodation in Piaget’s nomenclature).

Constructivism suggests qualitative evaluation methods for assessing the students’ learning process further than quantitative ones [14]. However, quantitative methods have historically been trusted in programming learning, and qualitative methods, such as colloquies, have rarely been used in programming learning research. We compared the subjective quantitative data gathered from colloquies against grades that participants obtained through traditional evaluation methods in our course. Correlations in Table 2 show that colloquies behaved very dissimilarly to the first long programming assignment (also know as “programming project” in our university) and short programming assignments (homework). Students had plenty of time to resolve these evaluations, and they may have obtained scaffolding from diverse sources. Therefore, these evaluations may not reflect effective student’s knowledge. In the other hand, colloquies behaved very similarly to the traditional methods conducted near the end of the course, i.e. the third exam and final project. These were the two most comprehensive quantitative assessments, because they demanded comprehension of most programming concepts learned in the course. These findings support the suggestion about qualitative assessments stated by constructivists [14], and this is the second most important contribution of this paper. However, further research is required in order to know if these results can be generalized.

Future research may answer if exams can be replaced by colloquies or both are only complementary methods. In this research, both methods had different purposes. Exams were intended to assess problem-solving abilities whereas colloquies were intended to assess understanding of the C++ notional machine. The authors have a positive judgment about the potential and benefits of colloquies. For example, if notional machine understanding was evaluated using standard questions, most students would have correctly answered a question that asked to predict the output for the program in Fig 6, and no evidence of whole division misunderstandings would have been detected. In the other hand, colloquies may have serious drawbacks, mainly with large populations of students, not self-confident participants, or students that reject being recorded. Further research is required in order to have a better picture of this method in the programming learning field.

7 CONCLUSIONS AND FUTURE WORK

From a review of the literature, it is known that a correct understanding of runtime behavior of programming concepts is mandatory to program computers [1], and students are expected to acquire this understanding mainly through the most used teaching method worldwide: the lecture [5], [6]. However there is a lack of evaluations for this method for programming learning. This research advances the knowledge with the following conclusions:

1. Lectures are insufficient to guarantee that students acquire a correct understanding of the notional machine of a programming language, even though, enriched lectures with concept contraposition and program memory tracing techniques are used.
2. McCracken et al. suggested that many students do not know how to program after passing the first programming courses [12]. Findings in the evaluated group of students in this research support the McCracken et al. statement, but at a more specific level: many students do not understand the runtime behavior of programming concepts after passing the first programming courses.
3. Colloquies behaved in the same role as the most comprehensive traditional assessment methods. Furthermore, they empowered researchers to detect and correct some important program misconceptions that exams or projects were unable to detect.

There are many interesting topics for future research in order to have a better picture about how students learn computer programming. The following is just a short suggestion list.

1. Can lectures be enriched with other techniques or materials to guarantee that students acquire a correct understanding of notional machines? For example, lectures enriched with program visualizations.
2. Are constructivism teaching-learning methods more effective than lectures to guarantee the student's correct understanding of notional machines? Are they more efficient?
3. Are mental models of notional machines stable in the student's long-term memory? In other words, do students actually learn notional machines?
4. How do students transfer notional machine understandings when the programming language is changed? For example, from Java in the Programming I course to C++ in the Programming II course. How is the notional machine transfer process affected if the programming paradigm is changed as well? For example from C++ to Lisp.
5. Is the ability of writing memory-leak free code in C++ negatively affected when students have learned Java programming previously?
6. Does an understanding of the physical machine help students transfer from one notional machine to another or not?
7. Are colloquies more effective to assess achievement of the programming course's objectives than traditional exams? In what contexts are they useful (e.g.: small number of students, cooperative learning, problem solving)?

Answering these questions will enrich our understanding of the process for learning computer programming. These answers can be also used by educational systems to enhance programming learning methods and materials, and ensure that students that pass programming courses actually know how to program.

Acknowledgements

This research is supported by the Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC), the Escuela de Ciencias de la Computación e Informática (ECCI), both from Universidad de Costa Rica (UCR), and the Ministerio de Ciencia Tecnología y Telecomunicaciones de Costa Rica (MICITT).

References

- [1] J. Sorva, “Visual Program Simulation in Introductory Programming Education,” Aalto University, 2012.
- [2] ACM and IEEE Computer Society, “Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science,” 2013.
- [3] B. du Boulay, T. O’Shea, and J. Monk, “The black box inside the glass box: presenting computing concepts to novices,” *Int. J. Man. Mach. Stud.*, vol. 14, no. 3, pp. 237–249, Apr. 1981.
- [4] J. Sorva, “Notional machines and introductory programming education,” *ACM Trans. Comput. Educ.*, vol. 13, no. 2, pp. 1–31, Jun. 2013.
- [5] T. L. Naps, S. Rodger, J. Á. Velázquez-Iturbide, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, and M. McNally, “Exploring the role of visualization and engagement in computer science education,” *ACM SIGCSE Bull.*, vol. 35, no. 2, p. 131, Jun. 2003.
- [6] E. Isohanni and H.-M. Järvinen, “Are visualization tools used in programming education?: by whom, how, why, and why not?,” in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research - Koli Calling '14*, 2014, pp. 35–40.
- [7] M. P. Sánchez González, *Técnicas docentes y sistemas de evaluación en educación superior*. Narcea Ediciones, 2010.
- [8] M. Hertz and M. Jump, “Trace-based teaching in early programming courses,” in *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*, 2013, p. 561.
- [9] A. Vihavainen, J. Airaksinen, and C. Watson, “A systematic review of approaches for teaching introductory programming and their influence on success,” in *Proceedings of the tenth annual conference on International computing education research - ICER '14*, 2014, pp. 19–26.
- [10] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming,” *ACM SIGCSE Bull.*, vol. 39, no. 2, p. 32, Jun. 2007.
- [11] C. Watson and F. W. B. Li, “Failure rates in introductory programming revisited,” in *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, 2014, pp. 39–44.
- [12] M. McCracken, V. Almstrum, D. Diaz, L. Thomas, M. Guzdial, I. Utting, and D. Hagan, “A multi-national, multi-institutional study of assessment of programming skills of first-year CS students A framework for first-year learning objectives,” *ACM SIGCSE Bulletin, Volume 33 Issue 4*, pp. 128–180, Dec-2001.
- [13] Luria, Leontiev, and Vigotsky, *Psicología y pedagogía*, 4th ed. Sevilla, España: Ediciones Akal, 2011.
- [14] M. Ben-Ari, “Constructivism in Computer Science Education,” *J. Comput. Math. Sci. Teach.*, vol. 20, no. 1, pp. 45–73, 2001.
- [15] J. Hidalgo-Céspedes, G. Marín-Raventós, and V. Lara-Villagrán, “Student understanding of the C++ notional machine through traditional teaching with conceptual contraposition and program memory tracing,” in *2015 Latin American Computing Conference (CLEI)*, 2015, pp. 1–8.
- [16] L. Ma, J. Ferguson, M. Roper, and M. Wood, “Investigating the viability of mental models held by novice programmers,” in *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, 2007, vol. 39, no. 1, pp. 499–503.
- [17] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood, “Using cognitive conflict and visualisation to improve mental models held by novice programmers,” in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, vol. 40, no. 1, pp. 342–346.
- [18] L. Ma, J. Ferguson, M. Roper, I. Ross, and M. Wood, “Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations,” in *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, 2009, vol. 41, no. 3, pp. 166–170.

- [19]D. Perkins and F. Martin, “Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22.,” Sep. 1985.
- [20]C. Crook, “Computers in the zone of proximal development: Implications for evaluation,” *Comput. Educ.*, vol. 17, no. 1, pp. 81–91, Jan. 1991.
- [21]J. Whalley and N. Kasto, “A qualitative think-aloud study of novice programmers’ code writing strategies,” in *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, 2014, pp. 279–284.
- [22]T. Greening, “Emerging Constructivist Forces in Computer Science Education: Shaping a New Future?,” in *Computer Science Education in the 21st Century*, T. Greening, Ed. New York, NY: Springer New York, 2000, pp. 47–80.
- [23]J. F. Kearney and J. Nodine, “Application of cognitive dissonance in the teaching of FORTRAN IV in an unstructured setting,” in *Proceedings of the annual conference on - ACM'73*, 1973, pp. 436.1–437.
- [24]D. Teague and R. Lister, “Manifestations of preoperational reasoning on similar programming tasks,” in *ACE '14 Proceedings of the Sixteenth Australasian Computing Education Conference*, 2014, pp. 65–74.
- [25]D. Teague and R. Lister, “Programming: reading, writing and reversing,” in *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*, 2014, pp. 285–290.
- [26]D. Teague and R. Lister, “Longitudinal think aloud study of a novice programmer,” in *Proceedings of the Sixteenth Sixteenth Australasian Computing Education Conference (ACE2014)*, 2014, pp. 41–50.
- [27]N. Arshad, “Teaching programming and problem solving to CS2 students using think-alouds,” in *Proceedings of the 40th ACM technical symposium on Computer science education - SIGCSE '09*, 2009, vol. 41, no. 1, p. 372.
- [28]S. Wiedenbeck and A. Engebretson, “Comprehension Strategies of End-User Programmers in an Event-Driven Application,” in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 2004, pp. 207–214.