Learning memory management with C-Sim: A C-based visual tool

Baltasar García Perez-Schofield[a], Matías García Rivera[a], Francisco Ortin[b], María J. Lado[a]

[a]*Higher School of Computer Science Engineering, University of Vigo, Campus As Lagoas, Ourense, Spain*

[b]*University of Oviedo, Computer Science Department, c/Calvo Sotelo s/n, 33007, Oviedo, Spain*

Notice: This is the authors' version of a work accepted for publication in Computer Applications in Engineering Education. Please, cite this document as:

Baltasar G. Perez-Schofield, Matias Garcia Rivera, Francisco Ortin, Maria J. Lado. Learning memory management with C-Sim: A C-based visual tool. Computer Applications in Engineering Education, volume 27, issue 5, pp. 1217-1235, September 2019, doi: 10.1002/cae.22147.

Learning Memory Management with C-Sim: A C-Based Visual Tool

Abstract. Nowadays, Computer Science (CS) students must cope with continuous challenges related to programming skill acquisition. In some occasions, they have to deal with the internals of memory management (pointers, pointer arithmetic and heap management) facing a vision of programming from the low abstraction level offered by C. Even using C++ and references, not all scenarios where objects or collections of objects need to be managed can be covered. Based on the difficulties identified when dealing with such low-level abstractions, the **C-Sim** application, aimed at learning these concepts in an easy way, has been developed. To support the tool, the C programming language was selected. It allows to show concepts, remaining as close as possible both to the hardware and the operating system. To validate **C-Sim**, *pre-* and *post-tests* were filled in by a group of 60 first-year CS students, who employed the tool to learn about memory management. Grades of students using **C-Sim** were also obtained and compared to those that did not use the tool the former academic year. As main outcomes, 82.26% of students indicated that they had improved programming and memory management knowledge, and 83.64% pointed out that the use of this type of tools improves the understanding and quality of the practice lessons. Furthermore, marks of students have significantly increased. Finally, **C-Sim** was designed from the ground up as a learning aid, and can be useful for lecturers, who can complement their lessons using interactive demonstrations. Students can also employ it to experiment and learn autonomously.

Keywords: Education, Memory Management, Systems Programming, Visual Tool, C Programming Language.

1 Introduction

According to the Computer Science (CS) Curricula 2013 (CS2013), fundamental skills and knowledge that all computer engineering graduates must possess must be insistently sought and carefully identified [1]. The learning of CS includes several topics in Programming Languages.

Great efforts about learning programming are continuously being made to teach students from primary school [2,3,4] to University students [5,6,7]. In this education level, introductory CS courses are aimed at developing programming skills. As an example, the University of Oxford (UK) includes in the Bachelor and Master Programs in CS the learning of Functional and Imperative Programming (first year), and Concurrent Programming (second year) [8]. The Carnegie Mellon University (USA) recommends students with no programming experience the course Fundamentals of Programming at the beginning of their studies [9]. The Programming Methodology course is proposed as a first step in learning about CS in the University of Standford (USA) [10]. The University of Toronto (Canada) offers in the first year the course Introduction to Computer Programming [11], similar to the University of Swinburne (Australia), which includes the course Introduction to Programming [12]. In the École Polytechnique Fédérale de Lausanne (Switzerland), students must pass in the preparatory year the subjects of Introduction to Programming and Practice of Object-Oriented Programming [13]. The curriculum of the CS Degree of the University of Vigo (Spain) also includes different programming subjects in the first and second year [14]. This degree is conducted in the Escuela Superior de Ingeniería Informática, where a traditional approach (imperative-first, students firstly learn procedural programming) is taken, contrasting with the objects-first one, followed by also many other faculties.

In a first programming subject (Programming I), students learn basic programming by means of the C++ [15] programming language. In fact, they are taught C [16], and some selected concepts from C++, such as references. In a second subject (Programming II), students learn object-oriented programming through Java [17]. Time is a limiting factor to teach low-level mechanisms related to memory storage and operating system in the first Programming courses [18]. These concepts are taught in the Computers Architecture I subject, during the second semester.

To learn memory management, students must face a vision of programming from the low abstraction level offered by C. Even using C++ and references, lecturers find it impossible to cover all scenarios where objects or collections need to be managed. The concept of pointer and other topics related to memory management must be taught. Students have to learn about memory addresses, stack vs. heap, word size, among other important concepts [19].

When learning memory management, and in accordance with other published works [20], several complex concepts have been detected: a) memory as a one dimension array; b) codification of types, with their different sizes and how the word size of the machine affects that codification; c) the concept of pointer being just an integer (representing a memory address), and a type (the type of the pointer, which denotes the number of bytes occupied by the value); and d) memory access from the address stored in a pointer, sometimes involving the & and * operators, together with pointer arithmetic (specially for running over arrays).

Several applications, all including Graphical User Interface (GUI), aimed at helping students to acquire memory management skills and deal with pointers can be found in the literature. Table 1 shows the main characteristics and limitations of these tools. Their most meaningful trait is the set of views of the program being executed they offer, being these source memory representation, variable relationship diagrams, etc. The column "Integration" shows whether these views are used as a whole instead of separate tools. The column "Program animation" shows whether all these views are updated with each executed instruction or not.

The main contribution of this work is to present C-Sim, a visual programming tool devoted to learn memory management concepts, and focussed on visual representation on memory storage of variables, and the relationships set from their addresses (i.e., pointers), and allows students to interact with memory in a controlled sandbox. In contrast to a common debugger, students do not need to have previous knowledge about the specifics of memory management, or the effects of pointer arithmetic; results for each instruction are reported in all views; and finally errors occur without being catastrophic. Therefore, users can safely simulate the consequences of using pointers, or the * and & operators, learning the relations set among variables through live diagrams. Moreover, our students can also interact with a given program running it step by step, observing the effects of each instruction. They can also use our tool to change any values of variables (specially including pointers). Thus, **C-Sim** opens a wide spectrum of educational possibilities. With this tool, we are not pretending to present a tool to learn C, but to have a support to deal with memory management concepts, while deepening in C concepts.

It must also be remarked that **C-Sim** can be a very useful tool to explain concepts such as physical distribution of data structures in the computer's physical memory, this being necessary for later learning of string management, dealing both with dangling and wild pointers, learning of copy constructor and assigning operator, passing by value and reference parameters, working with insecure code and vulnerabilities, such as buffer overflows/underflows, or detecting the adequate moment to free memory in different applications.

The rest of the paper is organized as follows: Section 2 explains the fundamental aspects about the software tool, including main features, requirements, implementation and use. Section 3 presents the methodology employed to validate the application. Section 4 shows the results obtained, as well as the corresponding discussion. Finally, conclusions are presented in Section 5.

2 Software Description

2.1 Main Features

C-Sim is a C-based visual application designed for learning memory management. The main goal is to provide students with an easy tool to enable them a better understanding of memory assignment to variables and arrays, and specially how pointers hold the memory address of the pointed variables. An initial version of the tool was first published in 2014, but with very limited functionality [30].

C-Sim has followed an important evolution over the last years, and initial existing limitations related to lack both of GUI and representation of arrays, were overcome in this new version.

The main features of present **C-Sim** are:

- It creates a dynamic visual scheme of the variables in the program, depicting their locations in memory and their relationships.
- It allows users to create or modify variables through a friendly GUI, which is globally updated with each interaction.
- It presents viewers for existing variables, history and watches. The latter allows to dynamically follow the variations in value for a given variable.
- It offers immediately updated views, to show them as a quick answer to users' instructions.
- It includes pointers and references management, as well as memory states.
- It allows both to execute each instruction separately and the whole program.

2.2 Hardware and Software Requirements

In this Section, main hardware and software requirements are presented.

Basic hardware requirements to run **C-Sim** are the following:

- Arm Cortex A53, Intel Pentium SU4100, or AMD E350 processors.
- 1GB of RAM.
- 1MB of free space on the hard disk.

On the other hand, software requirements are:

- GNU Linux Distributions, 32 and 64 bits. It has been executed showing complete functionality both in Arch Linux and Ubuntu Linux.
- Windows, 32 and 64 bits. Total functionality has been tested in Windows 10.
- Mac OS, 32 and 64 bits.
- Mono software platform [31]. While Windows executes this tool without the need of any other dependency, in Linux and Mac the Mono implementation of the .NET framework is needed.

2.3 Implementation

The architecture employed to develop **C-Sim** is shown in Figure 1. Main parts are described below.

2.3.1. Machine

Class central to the design, being the most important properties the word size and the endianness (explained in Section 2.4, Session 2). These two characteristics of the machine make it mandatory to create new objects when changed: **ByteConverter**, **TypeSystem**, **MemoryManager**, **SymbolTable**, and the **SnapshotManager** (in order to manage the different states of the machine for each instruction).

2.3.2. StdLib object

Standard library, which contains all the functions in the system. It can be maintained for all executions. For each instruction, a new **Parser** and **Lexer** (for the instruction's lexical and syntactical analysis), as well as an **ExecutionStack** must be created too, when the instruction is valid. In that case, the parser generates a list of opcodes to execute, and after their completion, all the affected variables are updated in memory. While executing, when an opcode results in an error, all pending opcodes are dismissed and the error is reported to the user. This means that, although it seems to be an interpreter, this tool actually behaves as an integrated compiler and a virtual machine.

2.3.3. Opcodes

Understood by the virtual machine, and shown in Figure 2, apart from **SubOpcode**, **MulOpcode**, **DivOpcode** and **ModOpcode**, which are hidden in order to save space. The former are the ones providing functionality for mathematical operations, as in expressions such as "*3 * 4"* or "*2 + (4 * 5)"*. Some of them are:

- **CreateOpcode:** supports the creation for all types of variables. It is the one used in an instruction like "*int x;*" meaning that a variable *x* of type *int* is going to be created.
- **AssignOpcode:** used in instructions such as "*x = 5;*", in which the value of the variable *x* is changed to the *rvalue* (right value, represented by the **RValue** class, shown in Figure 3), at the right of the assignment operator (hence its name). Indeed *rvalues* (Figure 3) can be literal values (i.e., "*42*" or "5.0"), variable identifiers (i.e., "*y*" or "*x*"), the value of a variable, or even a type (the **TypeType** class). The later values allow to include types in expressions, as in "*sizeof(int)*", avoiding the need of a special implementation.
- **AddressOfOpcode** is the opcode for the '&' operator, used in expressions like "*&x*", in which the memory address of *x* is taken, presumably to be stored in a pointer variable.
- **AccessOpcode** (opcode for the '*' operator), complementary of the later, and used to access the memory address stored in a pointer variable, as in "**ptr*". In contrast to the '&' operator, the '*' operator can only operate on pointers again presumably intending to access the value of the pointed variable.

2.4 Working with C-Sim

The overall visual design of **C-Sim** is of a REPL (read-eval-loop) tool. Since the **C-Sim** core is an interpreter[1], it fits that usage very well, immediately updating all views in order to show them to users as an answer to their instructions. The main layout for the application is shown in Figure 4, in which the main parts of the window are highlighted in bright red. The user enters C instructions in the console input entry box, and the immediate result is outputted in the console immediately above it. The diagram viewer represents the state of memory in the simulated machine, and when the instruction does not result in an error, it is added to the history viewer, at the right panel. When a new variable is created, it will be added to the symbol viewer, in a panel at the left.

1 Actually, it is possible to execute **C-Sim** as a console interpreter (a REL or read-eval-loop), using the *--no-gui* parameter. The code shown in this Sections is taken from this view of the tool, while screenshots, diagrams and results are taken from the GUI view. Moreover, commands starting with a dot ("."), are only available in the console interpreter, since they are not needed in the GUI environment (the ">" prompt should not be entered, either).

The central viewer can change its representation through the appropriate tab selection. The default tab shows a diagram for all variables in the machine, while the other one shows a grid with the values for each position in memory. Moreover, a click in an entry of the symbol table will lead the user to the corresponding memory address for that variable in the memory grid. The grid for the whole memory just shows a continuous representation of the memory in the emulated machine, while the diagram can also relate pointers and variables.

Users can check the evolution of the values of variables in the watches panel. The history panel accounts the successful instructions entered. When the user chooses an instruction in the history panel, **C-Sim** updates the memory diagram and the memory grid to show the status of the emulated machine, up to that step. Also, pressing the play button in the toolbar will perform a step by step execution of the whole program, in which the user can see the results of each instruction for about a second.

To learn memory management employing **C-Sim**, a workshop consisting of four sessions was organized. Details are given in the following paragraphs.

Session 1 - Starting with the tool

The main objective of this introductory session is twofold. Firstly, students should be able to understand the basics of the tool, inviting them to experiment by themselves until they feel comfortable with the environment. Secondly, the a) and b) difficulties in Section 1 are addressed.

In this session, students are told to start entering specific instructions that will make them feel comfortable with the system.

```
> int square_side = 4
> int area = square_side * square_side
16
> printf("%d\n", area)
16
> printf("&square_side: %p, &area: %p\n", &square_side, &area)
&square_side: 0x04, &area: 0x08
> .dump
00 00 00 00 04 00 00 00 10 00 00 00 00 00 00 00 ................
```

C-Sim works primarily by using the console input (Figure 4). Just after the user enters a C instruction, the tool builds a diagram with the resulting memory scheme. Also, the memory grid (a raw view of memory), will be updated, giving the opportunity to analyze the consequences at byte level.

They see in the memory grid where variables *square_side* and *area* are located, the basic behavior of the diagram viewer as well, and the utility of the output viewer.

For each instruction, the tool presents an answer in the form: *<vble_id> (<vble_type> [<vble_address>]) = <vble_value>*. A variable definition always returns the created variable, while a function call will return a value (which is assigned to a temporary variable in the form of *_aux__x*). Specifically, *printf()* always returns the number of characters printed. Note that in the following code examples the auxiliary variables may be omitted.

The tool also provides a watches functionality, and a tree diagram in which all variables created in the machine are listed. For example, when clicking on variable '*area*' on the tree diagram, the memory grid opens as shown in Figure 5. As we found out before, variable *square_side* sits on position 4 with a value of 4, while *area* sits on position 8, storing a value of 16[2].

Next step is to transform the calculation of the area into a function call, specifically a call to *pow(a, b)*, which returns *a* to the power of *b*. In this way, students are introduced to the set of available functions of the standard library.

```
> square_side = 5
> area = pow(square_side, 2)
> printf(area)
25
> .dump
00 00 00 00 05 00 00 00 19 00 00 00 00 00 00 00 ................
```

The result (Figure 6), would be perceived as a change in the value of the *area* variable, from 16 (0x10) to 25 (0x19).

Session 2 - Working with pointers

As the first step when dealing with pointers, we clearly state that a pointer is just a matter of two concepts: a memory address, and a size (which is given by the type of the pointer). Indeed, the purpose of the program[3] below is to assist lecturing that very nature of pointers. Its result is shown in Figure 7.

```
> int x = 5
> int * ptr = &x
> printf("x: %d\n&x = %p\nptr: %p\n&ptr = %p", x, &x, ptr)
x: 5
&x = 0x04
ptr: 0x04
> .dump
00 00 00 00 05 00 00 00 04 00 00 00 00 00 00 00 ................
```

The previous code creates a simple integer variable *x* with value 5, and a pointer *ptr* pointing to it. **C-Sim** will draw a diagram with a box for *x* containing a value (5), and another box for *ptr* in a

2 A similar session can be found in video format here: https://youtu.be/dpKxLcuyUGo
3 A similar session can be found in https://youtu.be/R207-2SRBsA

lower row containing the memory address for *x*. As shown in the central panel of Figure 7, an individual variable (variable *x*), is represented by a first text line containing the type, the number of bytes occupied, and the memory address it starts on. Just below, a box containing its value (5) in decimal or hexadecimal (**C-Sim** defaults to hexadecimal) is presented, and immediately below the name of the variable ('*x*').

A similar representation scheme for the pointer variable *ptr* is also shown in Figure 7. The interesting part here is that the value of the *ptr* variable is equal to the start address of the *x* variable (0x04), and that the type of the pointer is the same as the type of *x* ('*int*'). That is why **C-Sim** draws an arrow between them.

A slighter complex program is given below.

```
> int a = 5;
> int * ptr1 = &a;
> int **ptr2 = &ptr1;
> printf("a: %d\n&a = %p\nptr1: %p\n&ptr1 = %p\nptr2: %p\n&ptr2 = %p", a, &a, ptr1, &ptr1, ptr2, &ptr2)
a: 5
&a = 0x04
ptr1: 0x04
&ptr1 = 0x08
ptr2: 0x08
&ptr2 = 0x0c
> .dump
2c 65 fc 0b 05 00 00 00 04 00 00 00 08 00 00 00 ,eü............
```

Memory addresses are given by default in ascending order, always considering aligning. However, depending on given settings (Figure 8), memory can also be randomly assigned. This means that by default, for a 32 bit machine (a four-byte machine word, the default machine type in **C-Sim**), four **int** variables *a*, *b*, *c* and *d*, will be given 4, 8, 16, and 32 addresses respectively. For a 16 bit machine (a two-byte machine word), those same variables will be stored at 2, 4, 6 and 8. However, this is not the only possibility: while **C-Sim** defaults to an ordered and aligned memory set to zeroes, it is possible to change that to a memory model in which random aligned addresses are assigned, and memory is set with garbage contents. The latter would be the opposite extreme in the range of possible configurations (alignment can be set in the configuration options, while a blank or a memory with random contents can be chosen on each reset).

Another issue is the so called *endianness*. Processors are said to be *little endian* when follow LSB (Least Significant Bit) ordering or *big endian* when follow MSB (Most Significant Bit) ordering for bytes in the values stored. That is, depending on from which byte they begin to read or write a given value in memory. A little endian approach would mean in practice to start considering the byte collection for any value taking the LSB first (as Intel processors do). Thus, for a *little endian* 32 bit processor, a value of 5 will be stored as 5, 0, 0, and 0, while a *big endian* 32 bit processor would store it as in 0, 0, 0 and 5.

Session 2.1 - Pointer arithmetics

This session addresses the d) difficulty identified in Section 1, accessing memory using pointer arithmetic, by allowing the user to freely experiment with pointer's values, and the * and & operators.

It is important to consider what is known as pointer arithmetic and weak typing, in which pointers take a central role. Pointers are not limited to contain the start address of another variable. They are not even limited to point to a variable of their own type. Indeed, the technique shown in this exercise normally consists on pointing to a variable of a given type, with a pointer pertaining to another one.

```
> int x = 25857
> char *pch = &x + 1
> printf("x = %d\n&x = %p\npch = %p", x, &x, pch)
x = 25857
&x = 0x04
pch = 0x05
> .dump
d8 b6 1c f4 01 65 00 00 05 00 00 00 98 b1 e6 1f Ø¶.ô.e.......±æ.
> printf(*pch)
e
> printf("*pch = '%c'\n", *pch)
*pch = 'e'
```

The output is shown in Figure 9. The variable *x* has a value of 25857, coded as 6501 in hexadecimal (represented with the traditional C prefix "0x", so 0x6501), and since a *little endian*, 32 bit machine is used, it is written in memory as bytes 01,65,0,0. The pointer to char *pch* is assigned *&x + 1*. While in C one would need to convert the pointer from type *int* to *char* (as in *((char *) &x) + 1*), in **C-Sim** this is simplified, and the & operator always results in a type-less and simple byte offset taken from the base (0) memory address.

The value of *pch* is 5, since *&x* results in 4 and then it is incremented in one. Taking into account that the representation of 25857 is [01,65,0,0] in the default machine, and it starts in address 4, **pch* is dereferenced to 0x65, which is the ASCII value for letter 'e'. In the output above, the value of *pch* (second line) is not shown, as **C-Sim** tries to display a string (zero-ended sequence of characters) in the special case of a pointer to *char*.

Session 2.2 - References

This session addresses the c) and d) difficulties identified in Section 1, accessing memory through pointers and using the * and & operators, showing how references help to hide that complexities.

Although this is transparent to the user, references in our tool are implemented as simple pointers, for instance *int &ref = a* is roughly equivalent to *int * ref = &a*. In spite of being an implementation detail, we introduce students to references using the same similarity, remarking the differences: a) references must be mandatorily initialized in their creation, b) they cannot change the variable they are pointing to, and c) they do not need to use the pointer syntax, i.e. '&' and '*' operators. Another way to understand references, maybe simpler, is that they are a mechanism to create another name (an alias) for an already existing variable. This explains why *ptr* in the source below points to *a*, when it is initially created as a pointer to the address of *ref*.

```
> int a = 5
> int &ref = a
> int * ptr = &ref
> printf("a = %d\n&a = %p\nref = %d\n&ref = %p\nptr = %p\n&ptr = %p\n", a, &a, ref, &ref, ptr, &ptr)
a = 5
&a = 0x04
ref = 5
&ref = 0x04
ptr = 0x04
&ptr = 0x0c
> .dump
00 00 00 00 05 00 00 00 04 00 00 00 04 00 00 00 ................
```

The output of the previous source code[4] is shown below, and also in Figure 10. In practice, using *ref* is like using *a.* However, they are different variables. As we can see in the memory dump above (and check out in the GUI view), there are actually two variables with the memory address of *a* (0x04), at addresses 0x08 (*ref*) and 0x0c (*ptr*).

Session 3 - Heap management

This session addresses the d) difficulty identified in Section 1, of accessing memory using pointer arithmetics, by allowing students to freely experiment with pointers' values, the * and & operators, and the values in each array position.

C-Sim implements two ways to deal with the heap (dynamic memory): functions *malloc()* and *free()*, as well as C++ operators *new* and *delete*. While *free()* and *delete* are interchangeable in **C-Sim**

4 A similar session can be found here: https://youtu.be/1xcK3Fw73ao

(this would result in undefined behaviour in C++), there is an important difference between *new* and *malloc()*: the first is typed, and the second is not. This means that *new int* returns a pointer of type *int*, in contrast to *malloc(sizeof(int))* which always returns a pointer of type *char*. The implications are subtle, but nonetheless important: the pointer with *new* will store the start memory of an integer number, while with *malloc()*, the pointer will hold the start address of an array of type *char* of length 4[5]. This is exemplified with the code below. An array of *char* is pointed with pointer *ptr2* of type *int*, and therefore managed as an *int* variable, although four positions will still be shown in the diagram anyway.

```
> int * ptr1 = new int(5);
> int * ptr2 = malloc(sizeof(int))
> *ptr2 = 5
> printf("*ptr1 = %d\nptr1 = %p\n", *ptr1, ptr1)
*ptr1 = 5
ptr1 = 0x08
> printf("*ptr2 = %d\nptr2 = %p\n", *ptr2, ptr2)
*ptr2 = 5
ptr2 = 0x10
> .dump
dc d0 12 76 08 00 00 00 05 00 00 00 10 00 00 00 ÜÐ.v............
> .dump 16
05 00 00 00 5a 9d 9f fb 98 cf 7c 4d 76 2f ce e9 ....Z..û.Ï|Mv/Îé
> free(ptr1)
> free(ptr2)
```

The output for those instructions is shown in Figure 11 (the screenshot was taken just before freeing memory).

Session 4 - Arrays

Similar to session 3, the d) difficulty of Section 1 was addressed.

Heap management and pointers are two concepts intimately linked to arrays in C, due to their own design. However, as discussed before, there is an important difference between *new* and *malloc()*: while the first is typed, the second is not.

Figure 12 shows an interesting example in which an array of pointers to *int* is created, and then the two first positions are made to point to integer variables *x* and *y*. The input is listed below.

```
> int x = 55
> int y = 66
> int ** v = new int*[10]
> v[0] = &x
> v[1] = &y
> int ** pv1 = &v[1]
```

5 Note again that the default 32 bit machine is used (*sizeof(int)* would return 4).

```
> printf(**v)
55
> printf(*v[1])
66
> printf(**pv1)
66
```

This creates the diagram shown in Figure 12.

The example above shows how to use arrays, dynamic memory and pointers with a double level of dereferencing. Since *&v[0]* and *v* is the same thing (as well as *v[0]* and **v*), it is possible to access *x* in the following equivalent ways: *x*, **v[0]*, ***v*. The same thing happens with *y*, which can be accessed as: *y*, **v[1]*, ***pv1*. This is represented in the code above.

3 Validation Method

In order to evaluate the usefulness of **C-Sim** as an assistant to learn memory management, two tests (*pre-test* and *post-test*), were designed for appraising the satisfaction of the students using the tool. The purpose of the *pre-test* was to appreciate their actual knowledge before the workshop. The *post-test* had many questions repeated, aiming at evaluating the increase or decrease of confidence of the student, while some of them just concern their personal experience using **C-Sim**. Some other are just slightly different, so results can be checked out to be coherent or not.

These tests were presented during a workshop (of four sessions), with 60 undergraduate students. They were all enrolled in the subject Computers Architecture I, of the first year, second semester, at the CS Degree of the University of Vigo.

Authors selected the topics presented in Table 2 as the central ones for assuring that students have really achieved a good and deep understanding of memory management. In this way, the *pre-test* and *post-test* were built around them.

The first topic of Table 2 is about evaluating how deep students thought their knowledge about memory management was and how it evolved. This is a self-evaluation question, aimed at capturing the subjective improvement in their knowledge, as well as second topic, who indicates whether students think memory management is useful for learning or not. Third and fourth topics are objective, and address the issue of assimilation of word size and endianness. The fifth subjective topic deals with students' opinion about the benefits of C-Sim as an aid to improve memory management understanding. The last topic is about students appraisal of how the use of the tool in the workshop has improved their knowledge about memory management.

In addition to these tests, grades obtained by students when learning memory management through the use of **C-Sim** were compared to those got by students (same number, 60 students) that followed traditional classroom, and did not use the tool, the former academic year. In the case of participating in the workshops, no extra time on the course was required: the ease of use of **C-Sim** and the functionalities provided by the tool allowed students to learn the same concepts (and even more) in the scheduled time. The use of **C-Sim** and the workshops were the only differences between both courses. In this way, evaluation systems, learning methodologies and teaching staff were the same.

Related to the evaluation system, it consisted of two parts: 1) acquisition of theoretical concepts (2 paper-based exams, 60% of the total mark), and 2) practical skills (2 computer-based tests, 40% of the total mark). It was just in this last part where skills acquired with **C-Sim** were evaluated the second year.

To verify if statistically significant differences existed between grades obtained when using/not using **C-Sim**, hypothesis test was applied, after verifying normal distributions for grades in both academic years.

4 Results and Discussion

In this work, a visual tool to deal with memory management learning has been presented. The tool was validated with 60 CS Engineering students, enrolled in an undergraduate course of Computers Architecture. Participants had to fill in two questionnaires, previous to the usage of **C-Sim**, and a further one after the learning of memory management with the tool.

Main results are shown in Figure 13 and Table 3, that presents the results obtained in both tests, as well as the questions asked to students, arranged in a way that the related ones in both tests are compared together. Some questions were included in both tests, to compare results before and after the memory management skills acquisition. However, other specific inquiries were only asked either in the *pre-* or the *post- test*. The first column shows the question numbering in the original tests as *x/-, -/y* or *x/y*. Firstly appears the question number in the *pre-test* (*x,* provided if it exits), and secondly the corresponding question in the post-test (*y,* provided it if exists).

4.1 Analysis

In the *pre-test*, questions 1 and 2 refer to the level of basic knowledge about memory management. Only 7.27% recognized to have no knowledge about memory management, while around 80%

considered they had some expertise. Both questions were evaluated together, since they were highly related; unsurprisingly, results were similar. In the *post-test* (question 4), students claiming to have good memory management understanding increased from 9.09% to 16.13%, while the percentage of students with no memory management knowledge drastically reduced to zero, which is a remarkable achievement. Question 1 was a complementary question in the *post-test*, dealing with previous experience in this matter. A percentage nearly to 59.48% of students thought they were high or medium experienced, while the remaining (40.32%) were unexperienced. These counter-intuitive numbers are probably explained by the depth of the sudden knowledge obtained after the workshop, a depth they simply just did not know about.

The third question in the *pre-test* (repeated as question 2 in the *post-test*), is related to the usefulness of the knowledge of memory management as a good complement in CS education. In the *pre-test*, 87.27% of students thought that it was useful, a rather intuitive concept indeed. This percentage increased up to 91.40% in the *post-test*, being the rest of answers “no” or “don’t know”. Though the sheer numbers are very good, it is unfortunate that still a small percentage was not sure, or even worse, answered “no”. Maybe the explanation is that they are used to high-level programming languages such as Java, in which only a shallow knowledge of low-level concepts is needed. It should be remarked that according to the CS curriculum of the University of Vigo, students learn C programming language in the first semester, in Programming I. In the second semester, students have to deal with Programming II (taught in Java), and with Computers Architecture I, among other courses.

The previous appreciation is probably related to the results of question 6 in *pre-test* (12 in *post-test*) about the real utility of the matter, i.e., whether memory management is just highly theoretical (only useful for lecturing), or not (also useful for real use in industry). The percentage of students considering that it was useful in theory and practice rose from 56.36% in *pre-test* to 59.68% in *post-test*. In addition, only a percentage of 11.29% thought it was just a theoretical asset in the *post-test*. It can be inferred by these numbers that an important number of students thought it was generally useful, but also that it became important for them to have acquired that knowledge.

Question 7 in the *pre-test (*6 in the *post-test*) deals with the importance of memory management for students training. In the *pre-test*, a percentage of 85.45% thought that it was important, increasing to 90.32% in the *post-test*, a net improvement of nearly a 5%. This last figure is related to those who were not able to decide about the usefulness before the workshop; they were reduced by more than a 4% in the *post-test*, meaning they realized that it had been productive for them. Overall, the sheer number of students satisfied with the knowledge acquired in the workshop is very good.

In question 8 (in both tests), in the *pre-test*, a percentage of 83.64% of students thought that **C-Sim** would improve their understanding, while a surprising 10.91% thought that their understanding would neither improve nor worsen. In the post-test, 87.10% of students thought that this visual tool had improved their understanding (a slight improvement in reference to the previous results), while a 9.68% (another slight improvement) thought that their understanding had neither improved nor worsen. Again, a surprising 3.23% thought that blackboard exercises would be more appropriate.

We can appraise how students, in general, consider it useful for both education and industry, and how they thought that learning this model was useful for they studies.

There are two questions that are central in these tests, and thus repeated (with slight variations) in both the *pre-test* (questions 4 and 5) and the *post-tests* (questions 5 and 3, respectively). In the first pair (questions 4/5), the student was asked to determine which concepts are central when transmitting data from one computer to another, with three possible meaningful answers: endiannes and wordsize, wordsize, and finally "don't know". In the *pre-test*, a 63.34% indicated the first answer , while the third option had an important share: 20%. In the *post-test*, those giving the correct answer rise to 85.48%, while students who did not know decrease to a mere 6.45%.

The second central question (questions 5/3), more specific, complements the previous one. In the *pre-test*, 81.82% of students selected the correct answer, increasing the percentage to 87.10% in the *post-test*. Furthermore, none of them answered "none of the above".

The remaining questions were specific for the *post-test*, and were related to the perception of the student about the software itself. In question 9, a percentage of 64.52% of students considered the software "Simple", while in question 10, 47.77% liked the language being simple, 9.68% considered that the best of the tool was memory grid, and 27.42% the way it shows data. In addition, 43.55% liked it (question 11). These results are very good in general, though the memory grid can be inferred to be not very popular at all.

Finally, questions 7 and 13 in the *post-test* asked students about whether this workshop had modified their conception about the matter. Attending to question 7, 82.26% of students recognized that the workshop had improved their understanding of memory management. For question 13, a percentage of 16.13% admitted that their conception had considerably changed. This already gives considerably merit to **C-Sim**: more than 82% of students thought they had a better understanding, and about 84% admitted that the workshop had somehow changed their conception, results which we think are a complete success for our objectives.

4.2 Evaluation

The most important result is probably the one obtained in the *post-test* about question 7, designed in order to know whether students considered that the workshop (and therefore the use of this tool), had improved their knowledge about memory management. A total 82.26% of them considered that their knowledge had improved.

Questions 1/4 (*knowledge of memory*), 3/2 (*useful for learning*), 4/5 (data transfer), and 5/3 (key components), were designed to indirectly verify the usefulness of the tool by evaluating students' comprehension of memory management. The number of students thinking that they had an important knowledge about memory management (*knowledge of memory*) rises, as well as students considering it useful for learning (*useful for learning*). It is definitely possible to appreciate the improvement in the number of students selecting the correct answer for *data transfer* and *key components*, which in case of questions 4/5 (*data transfer*), is certainly impressive.

Questions 10 and 11 in the *post-test* were designed to remark the strong and weak points of the software. It is really interesting and encouraging that nearly half of students had found this software very good by selecting the "I like it all in the software" option.

Regarding solely the software, questions 8/8 (*tool for learning*) and 9 (*knowledge improvement*) are remarked here because they are especially interesting. Question 8 gave students the opportunity to evaluate the usefulness of **C-Sim** in contrast to traditional classroom. Finally, question 9 was designed to evaluate the thoughts of students about the simplicity of **C-Sim**. Results indicate that the tool was simple to use and easy to manage for students.

4.3 Evaluation of Students Grades

Related to the programming skills acquisition and students' appraisal, Table 4 shows the different marks obtained by students that used/did not use **C-Sim** to learn memory management in both academic years considered.

Grades obtained by students improved in a significant way when **C-Sim** was used to learn concepts about memory management. In this way, the percentage of students that failed the test was drastically reduced in 25%. Moreover, the number of students getting marks between C and B+ also increased from 36.67% to 61.67%, this implying a strong improvement in the learning process. In addition, results of the hypthesis test yielded statistically significant differences when comparing both collections of marks (p-value < 0.001). In particular, the average grade for students who used **C-Sim** was considerably greater than the one obtained by students that did not try the tool (4.70 vs. 3.56 out of 10). This indicates that **C-Sim** can be a useful tool for learning memory management, since grades obtained by students that followed this methodology is better than those obtained by those following traditional classroom.

5 Conclusions and future work

In this paper, difficulties in learning various memory management concepts are identified. Both a specific lecturing strategy and its support by our educative tool have been discussed. The goal of **C-Sim** is to ease learning memory management, by means of behaving as an interpreter and live debugger for the C programming language. The C programming language (with a few C++ bits) is used due to its proximity to the representation level of the machine, without any intermediate layer or virtual machine.

The advantages of using this approach with our own students have been demonstrated with the use of two tests, in which students show both an improvement in their knowledge about pointers in particular and memory management in general, and their satisfaction with **C-Sim** as a support tool for education. The benefits of **C-Sim** as an aid to the learning process was also assessed when comparing grades of students that use/did not use the application, since a significantly greater percentage passed the exam when memory management was learned through the use of the tool.

Future work will be focused on two fronts: 1) to develop new lecturing strategies supported by the use of **C-Sim**, improving this tool with new functionality when needed, and 2) to advance in the support of more complex programs, involving functions and structs.

References

1. Joint Task Force on Computer Engineering Curricula, ACM, IEEE Computer Society. Computer Science Curricula Recommendation and Guidelines 2013. ACM New York, NY, USA (2013).
2. Kalelioğlu F. A new way of teaching programming skills to K-12 students: Code. org. *Comput Human Behav,*2015;52:200-210.

3. Bers MU, Flannery L, Kazakoff ER, Sullivan A. Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Comput Educ*,2014;*72:*145-157.

4. Mayer RE. *Teaching and learning computer programming: Multiple research perspectives*. *New York: Routledge; 2013.*

5. Law KM, Lee VC, Yu YT. Learning motivation in e-learning facilitated computer programming courses. *Comput Educ,*2010;55:218-228.

6. Esteves M, Fonseca B, Morgado L, Martins P. Improving teaching and learning of computer programming through the use of the Second Life virtual world. *Brit J Educ Technol*,2011;*42:*624-637.

7. Cedazo R, Garcia Cena CE, Al-Hadithi BM. A friendly online C compiler to improve programming skills based on student self-assessment. *Comput Appl Eng Educ*,2015;*23:*887-896.

8. University of Oxford: CS studies: https://www.ox.ac.uk/admissions/undergraduate/courses-listing/computer-science?wssl=1#. Accessed December 2018.

9. University of Carnegie Mellon: Bachelors Curriculum in CS: https://www.csd.cs.cmu.edu/undergraduate/bachelors-curriculum-admitted-2014-2015-2016#CS. Accessed December 2018.

10. University of Standford: Undergraduate in CS: https://cs.stanford.edu/degrees/ug/Considering.shtml. Accessed December 2018.

11. University of Toronto: CS: https://www.teach.cs.toronto.edu//cs_courses/current_course_web_pages.html. Accessed December 2018.

12. University of Swinburne: https://www.swinburne.edu.au/study/courses/units/Introduction-to-Programming-COS10009/local. Accessed December 2018.

13. École Polytechnique Fédérale de Lausanne: School of Computer and Comuunication Sciences: https://ic.epfl.ch/computer-science/study-plan_bachelor$_1$. Accessed December 2018.

14. University of Vigo: Escuela Superior de Ingeniería Informática: http://esei.uvigo.es. Accessed December 2018.

15. Stroustrup B. *The C++ Programming Language*. *New York: Addison-Wesley Professional; 2013.*

16. Kernighan BW, Ritchie DM. *The C Programming Language. New York: Prentice Hall; 1988.*

17. Gosling J, Bill J, Steele G, Bracha G, Buckley A. *The Java Language Specification. Redwood City: Addison-Wesley Professional; 2014.*

18. Tanenbaum AS, Bos H. *Modern Operating Systems. New York: Pearson; 2015.*

19. Jones R, Hosking A, Moss E. *The garbage collection handbook: the art of automatic memory management. London: Chapman and Hall/CRC; 2016.*

20. Milne I, Rowe G. Difficulties in learning and teaching programming—views of students and tutors. *Educ and Inf Technol 2002;*7:55-66.

21. Isoda S, Shimomura T, Ono Y. VIPS: A Visual Debugger. *IEEE Software*,1987;4:8-19.

22. Laffra C, Malhotra A. HotWire -- A Visual Debugger for C++. In *Proc. of the C++ Conference;*1994;109-122.

23. Mukherjea S, Stasktot JT. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger. *ACM Trans Comput Hum Interac*,1994;1:215-244.

24. Gries P, Mnih V, Taylor J, Wilson G, Zamparo L. Memview: A Pedagogically-Motivated Visual Debugger. *Procs of 35th ASEE/IEEE FIE Conference,*2005.

25. Kölling M, Quig B, Patterson A, Rosenberg J. The BlueJ System and its Pedagogy. *Comput Sci Educ,*2003;13:249-268.

26. Fernández A, Millán J. CGRAPHIC: Educational Software for Learning the Foundations of Programming. *Comput Appl Eng Educ,2003;11:167-178*.

27. García Perez-Schofield B, Ortín F, García Roselló E, Pérez Cota M. Towards an object-oriented programming system for education. *Comput Appl Eng Educ,2006;14:243-255.*

28. Guo PJ. Online python tutor: embeddable web-based program visualization for cs education. *Proc of the 44th ACM technical symposium on Computer science education,*2013;579-584.

29. Moreno L, González EJ, Popescu B, Toledo J, Torres J, Gonzalez C. MNEME: A memory hierarchy simulator for an engineering computer architecture course. *Comput Appl Eng Educ*,2011:19:358-364.

30. García Perez-Schofield B; Ortín Soler F. C-Sim, un simulador de manejo de memoria de C/C++. *Proceedings of JENUI,*2014.

31. Dumbill E, Bornstein NM. Mono: a developer's handbook. Boston: O'Reilly Media; 2004.

Tables

Table 1. Summarized comparison among systems.

System	Programming language	Pedagogical tool	Memory representation	Program animation	Limitations	Integration
VIPS [21]	Ada	No	No	No	High abstraction No explanation about representation of variables in memory models	No
HotWire [22]	C++	Yes	No	No	No explanation about memory storage of class instances	No
LENS [23]	C	Yes	No	Yes	Difficult to generate diagrams No explanation about representation of variables in memory models	Yes
MemView [24]	Java	No	No	No	No explanation about representation of variables in memory models	No
BlueJ [25]	Java	Yes	No	No	Object-oriented Scarce tools and little refactoring	Yes
CGRAPHIC [26]	C	Yes	No	No	Not focused on memory representation	No
Visual Zero [27]	Java	Yes	No	No	No explanation about representation of variables in memory models	No
Python tutor [28]	Python	Yes	No	Yes	High abstraction No deepening of pointers of memory representation	Yes
MNEME [29]	Java Swing	Yes	Yes	Yes	Memory address not directly shown Mainly for eviction algorithms and allocation page file	No
C-Sim v1.0 [30]	C, few bits of C++	Yes	Yes	Yes	Underdeveloped GUI No representation of arrays and pointers	Yes

Table 2. Topics to be assessed in the workshop of the prototype-based paradigm.

Key	**Topic**
Knowledge of memory management	Students' knowledge of memory management, both before and after the workshop.
Useful for learning	Students' opinion about the usefulness of memory management for learning programming.
Data transfer	The importance of word size and endianness for representing data.
Key components	The importance of pointers, word size and endianness in memory management.
Tool for learning	**C-Sim** is a valuable tool to learn the basics of memory management.
Knowledge improvement	Students' evaluation about how good **C-Sim** is.

Table 3. Pre-test and post-test questionnaires.

#	Question/options	Pre-test	Post-test	#	Question/options	Post-test
1/4	Do you think you have basic knowledge about memory management and computer architecture?			8/8	The fact of using a programming system that exemplifies memory management in the practice classes...	
	No	7.27%	01.61%		You think that classic blackboard classes would be better	05.45%
	Some	83.64%	82.26%		It will improve the understanding and the quality of the practice classes	83.64%
	Quite a lot	9.09%	16.13%		It won't improve nor worsen the understanding or quality	10.91%
2/-	Do you think you have basic knowledge about what memory management and computer architecture are useful for?			-/1	You had previous experience or formation	
	No	05.45%			Quite a lot	03.23%
	Some	78.18%			Some experience	56.45%
	Quite a lot	16.36%			None	40.32%
3/2	Do you think that memory management and computer architecture are useful for learning?			-/7	Do you think this workshop allowed you to improve your knowledge about memory management and computer architecture in particular, and programming in general?	
	Yes	87.27%	91.40%		Yes	82.26%
	No	05.45%	03.23%		No	04.84%
	I don't know	07.27%	04.84%		I don't know	12.90%
4/5	In order to send data between computers...			-/9	The software tool is:	
	Word size and endianness	63.34%	85.48%		Simple	64.52%
	Word size	13.33%	03.23%		Not simple nor complex	32.36%
	I don't know	20.00%	06.45%		Complex	3.23%
	None of the above	03.33%	04.84%			
5/3	In memory management and computer architecture, the concepts involved are...			-/10	The best of the software tool is:	
	Pointers	07.27%	09.68%		Simple programming language	
	Word size	01.82%	00.00%		Simple library	06.45%
	Endianness	05.45%	03.23%		How it shows data	27.42%
	All of above	81.82%	87.10%		Memory grid	09.68%
	None of the above	03.64%	00.00%		Nothing worth noting	00.00%
6/12	The usefulness of memory management and computer architecture is:			-/11	The worst of the software tool has been:	
	Both theoretical and practical	56.36%	59.68%		Simple programming language	01.61%
	Useful for lecturing	05.45%	25.81%		Simple library	06.45%
	Theoretical	30.91%	11.29%		How it shows data	22.58%
	I don't know	07.27%	03.23%		Memory grid	25.81%
	None of above	00.00%	00.00%		I liked it all	43.55%
7/6	Do you think it is going to be important for your studies to learn about memory management and computer architecture?			-/13	You think this workshop has changed the perception you had about programming	
	Yes	85.45%	90.32%		Quite a lot	16.13%
	No	01.82%	01.61%		Partly	67.74%
	I don't know	12.73%	8.06%		No	16.13%

Table 4. Percentage of students' grades obtained when using/ not using C-Sim.

Grades (out of 10)	Learning with C-Sim	Learning without C-Sim
F (0.0-4.9)	33.33	58.33
C (5.0-5.4)	23.33	20.00
B- (5.5-5.9)	6.67	3.33
B (6.0-6.9)	25.00	8.33
B+ (7.0-7.9)	6.67	5.00
A- (8.0-8.9)	3.33	3.33
A (9.0-10.0)	1.67	1.67

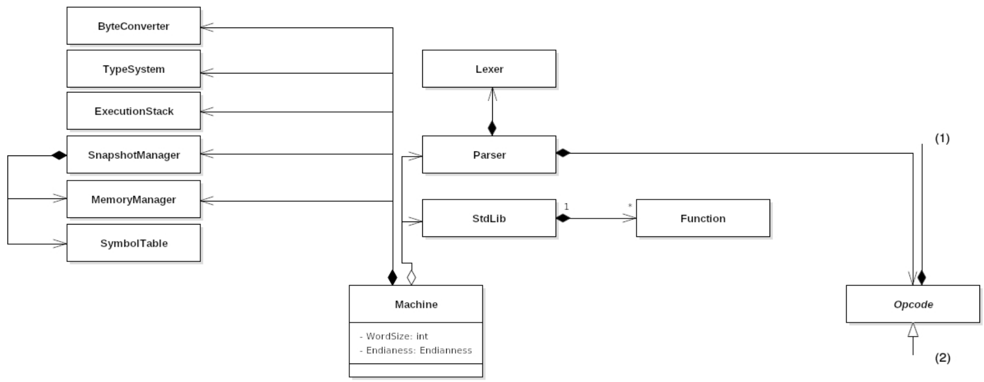
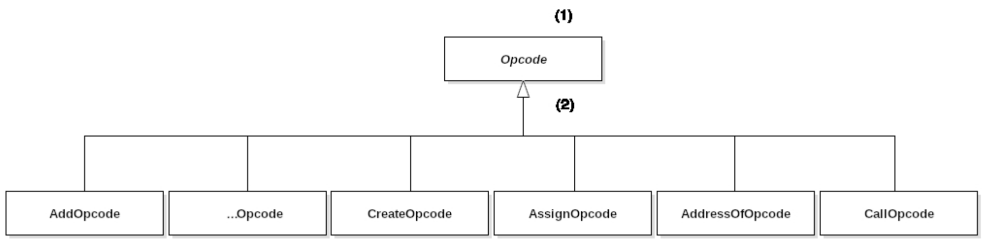


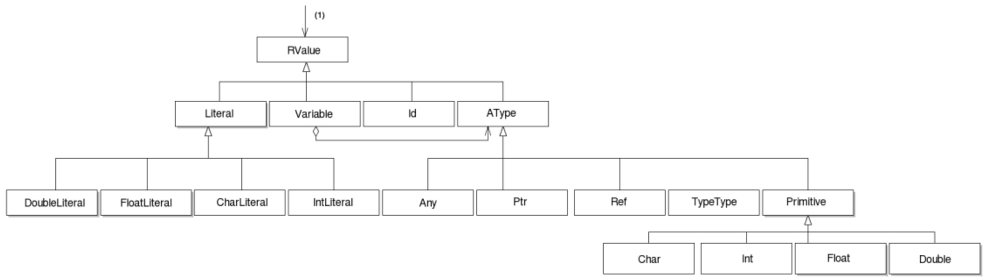
Figure 1. Lexer, Parser Functions and Opcodes do not vary for any machine model, while the ByteConverter, the TypeSystem, the ExecutionStack, the SnapshotManager, the MemoryManager and the SymbolTable are dependent of the word size and machine's endianness.

427x166mm (72 x 72 DPI)



Caption : Figure 2. The main opcodes supported.

363x89mm (72 x 72 DPI)



Caption : Figure 3. A selection of the types supported by any machine (this is a simplification). A few depend on its word size,14 while others are fixed in size. Types are a kind of RValue, which means that they can be part of a expression, as in sizeof(int).

629x189mm (72 x 72 DPI)

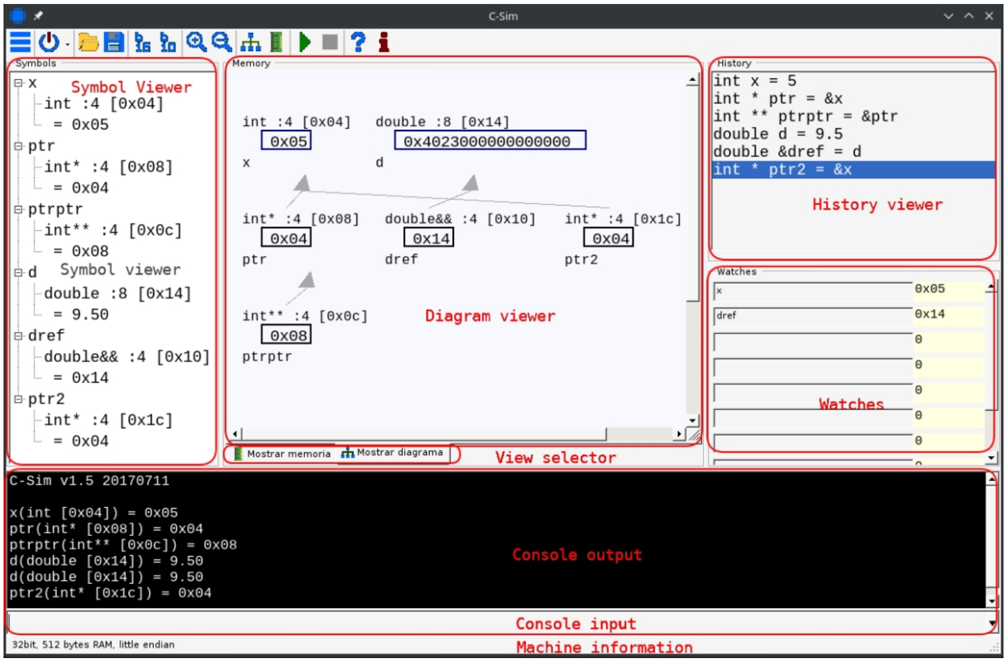


Figure 4. General layout of C-Sim, showing the four main parts of the interface, from left to right and top down: symbol viewer, diagram viewer, history viewer and watches manager, view selector, console output and input, and machine information.

418x273mm (72 x 72 DPI)

/	0	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
0	0	0	0	0	04	0	0	0	10	0	0	0	0	0	0	0
01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
03	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
04	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
07	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
09	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0f	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5. An example memory grid, in which two variables, one in position 4 with value 4, and another one in position 8, with value 16, are shown.

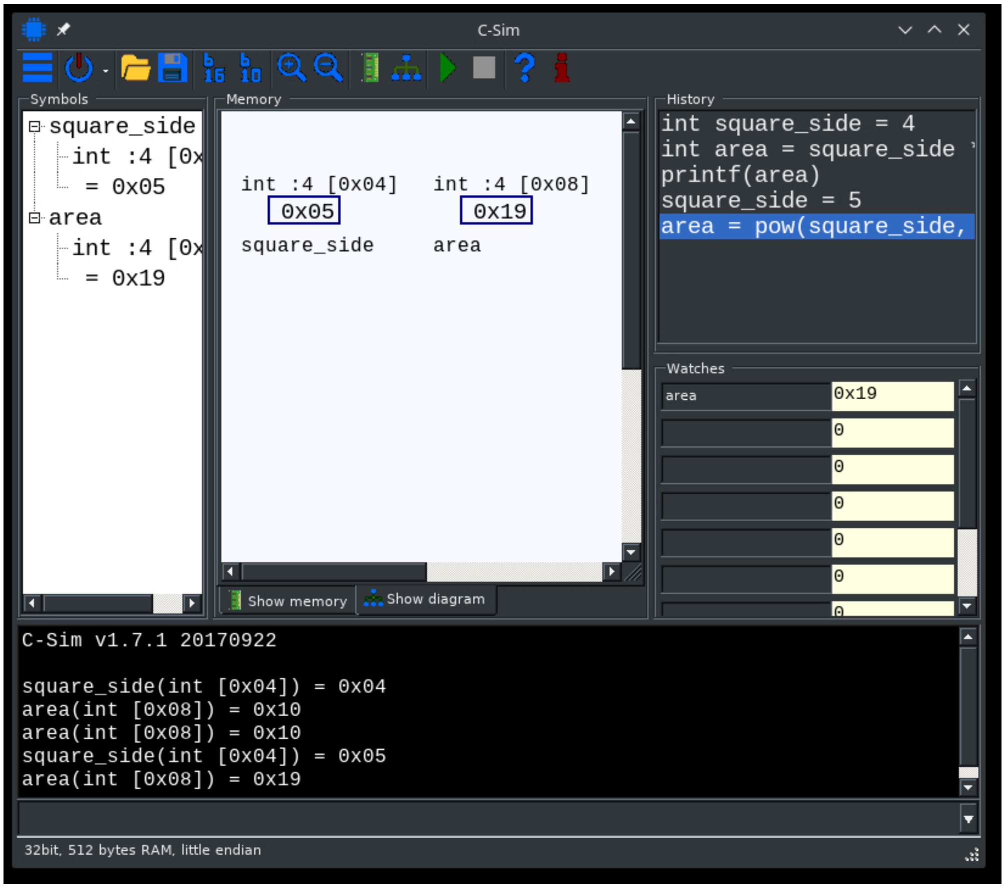


Figure 6. Starting with the tool.

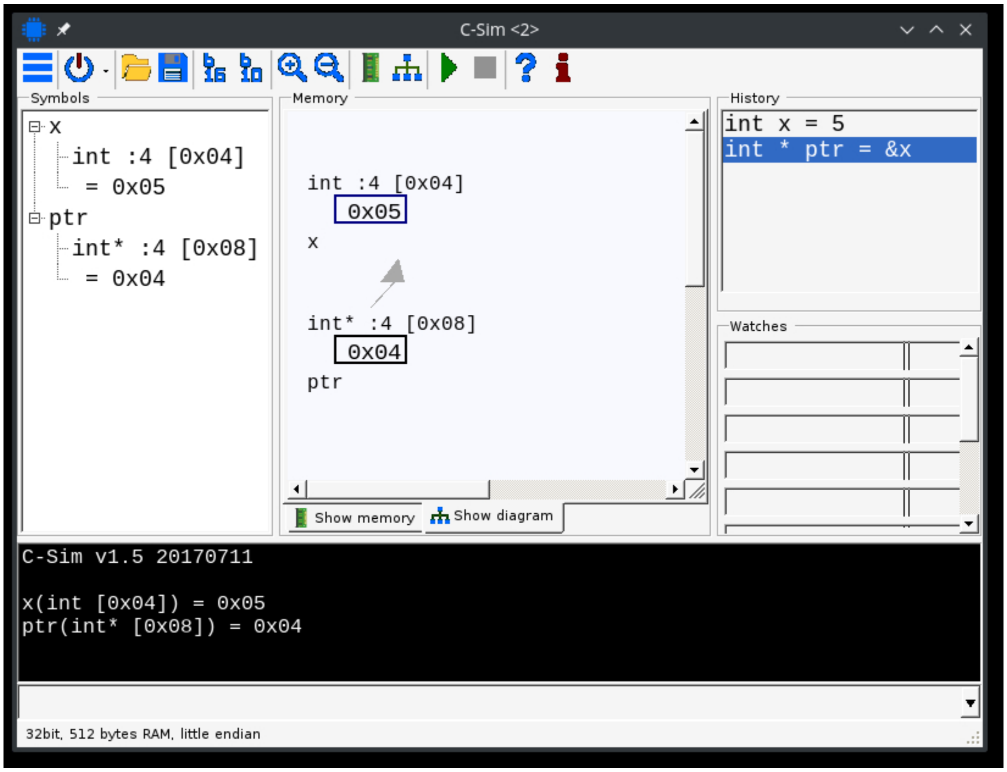


Figure 7. A first exercise showing a simple variable and a pointer pointed to it.

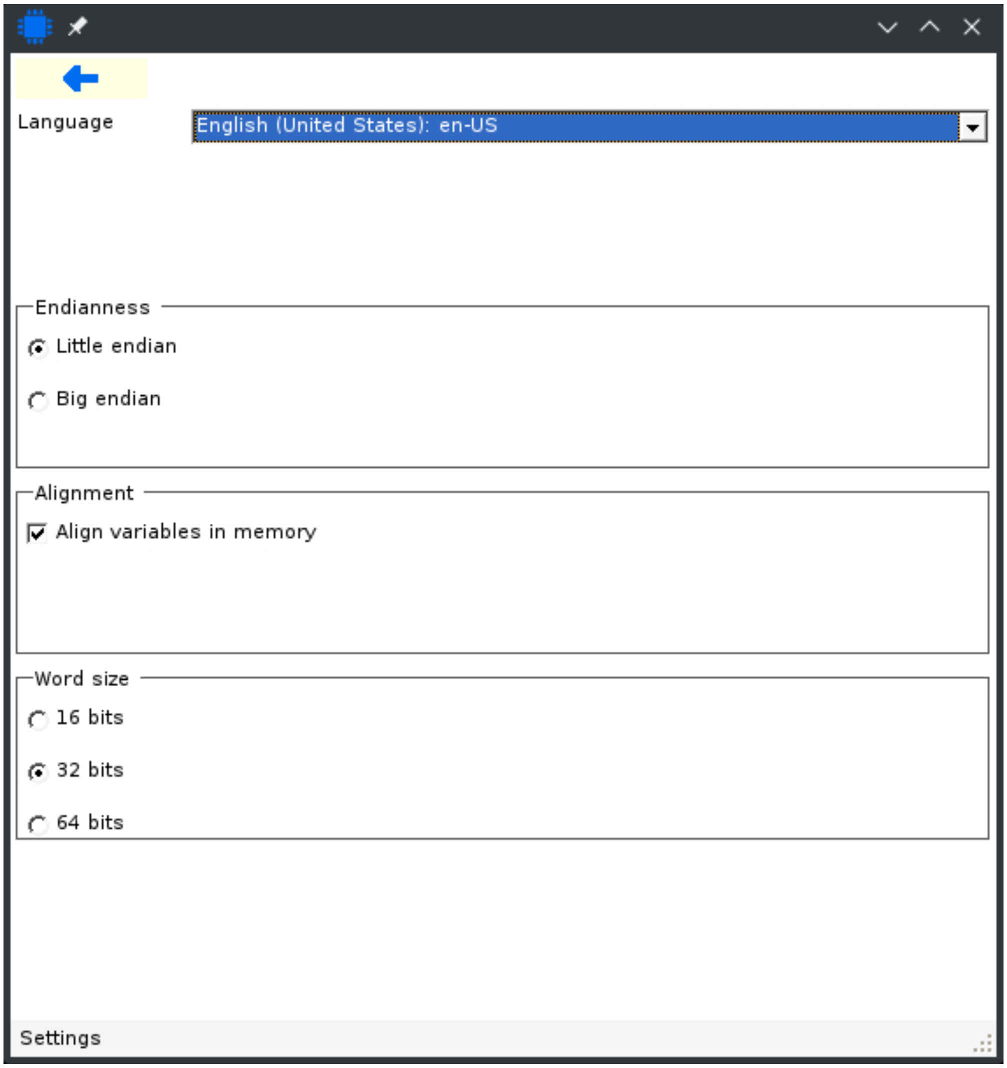


Figure 8. Available settings for the emulated machine.

201x213mm (72 x 72 DPI)

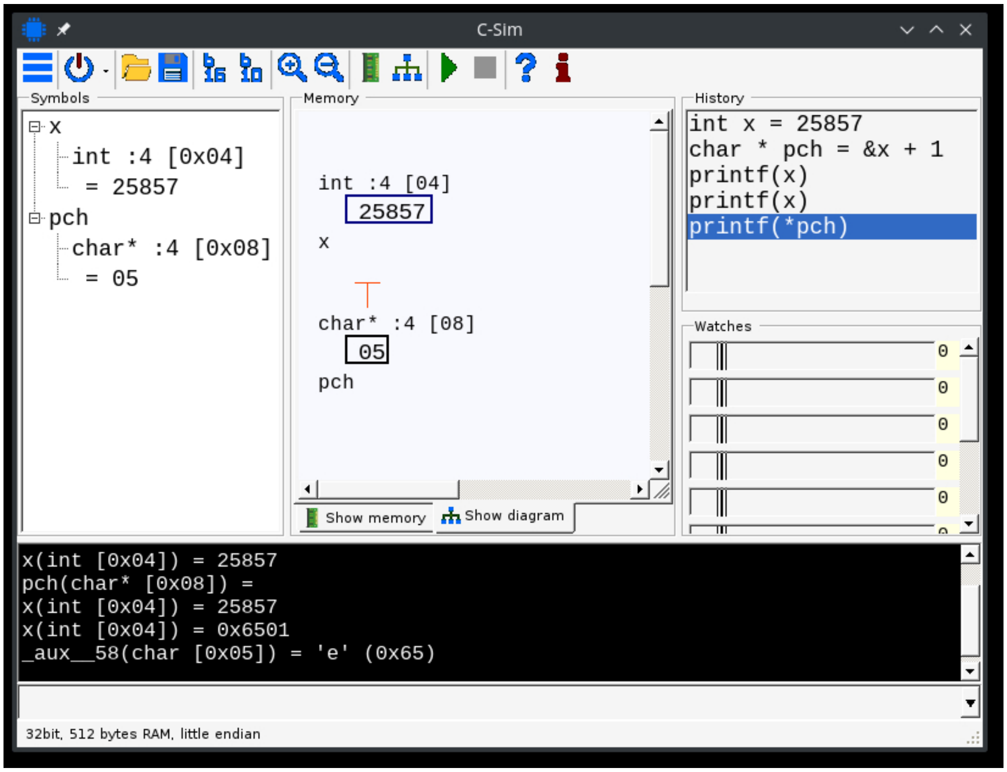


Figure 9. Pointer arithmetic and weak typing.

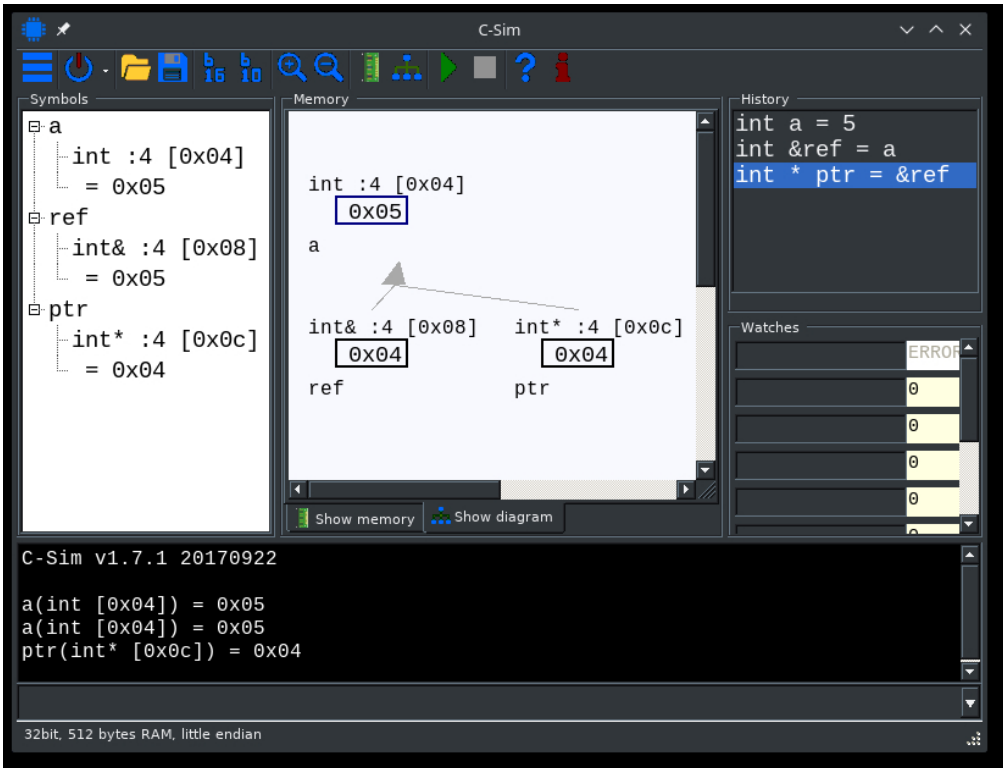


Figure 10: Using references.

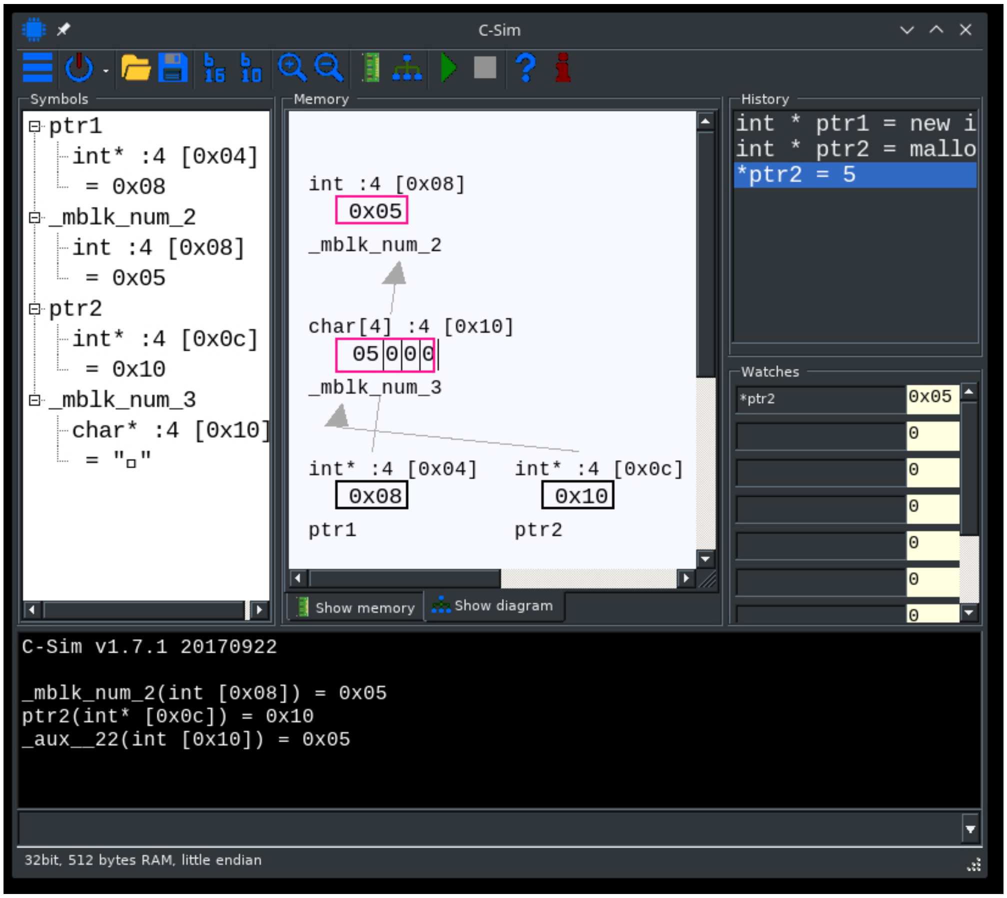


Figure 11: Difference between using new and malloc().

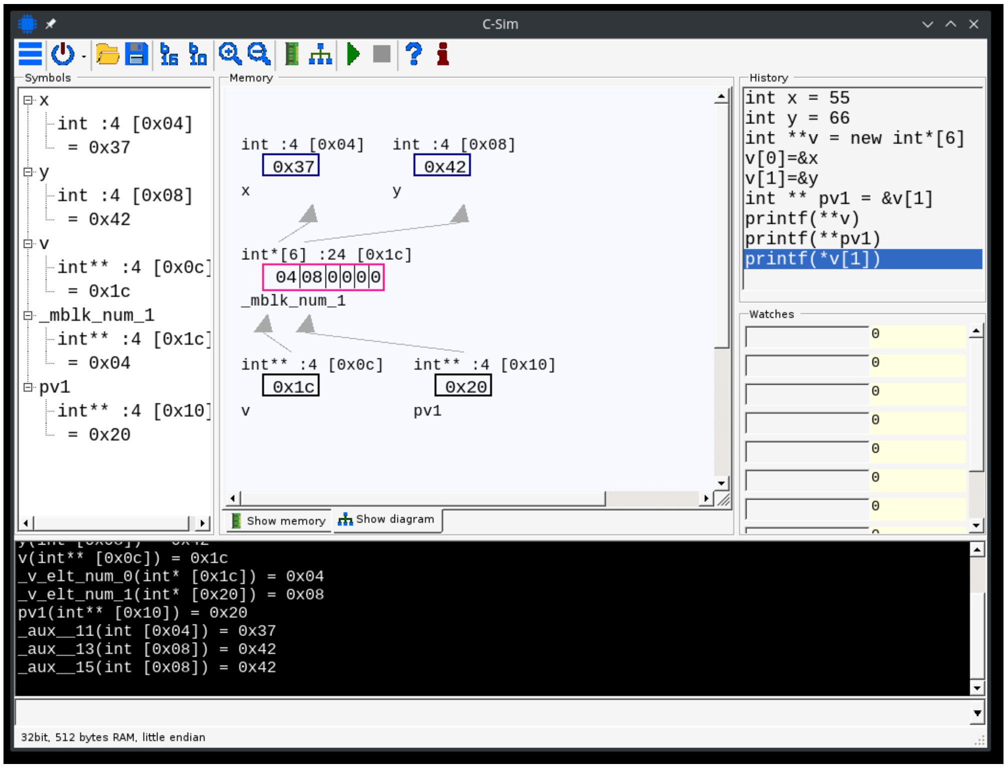


Figure 12: An example involving pointers, dynamic memory and arrays.

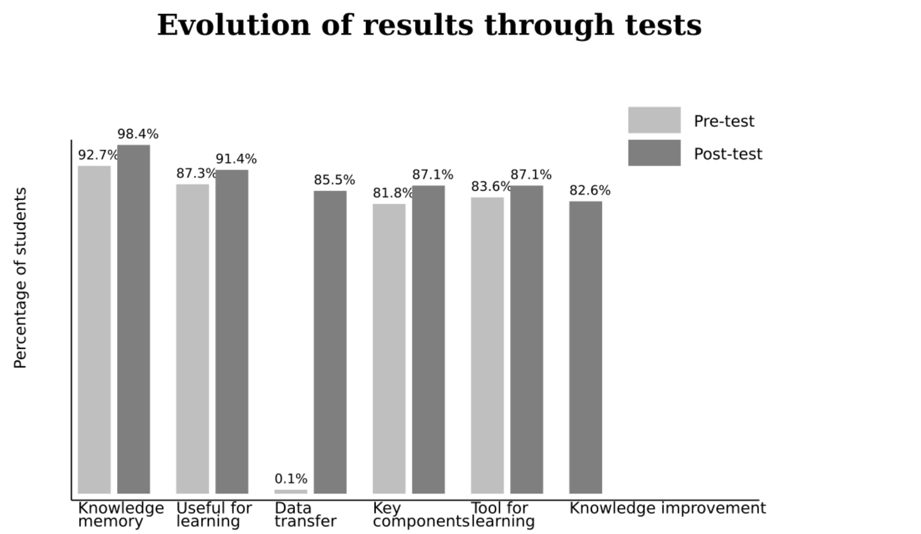


Figure 13: Bargraph representing the evolution of results for the main questions.

342x179mm (96 x 96 DPI)