UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

An Online C Programming Tutor

Dissertation presented by
Nicolas OOGHE

for obtaining the Master's degree in
Computer Science
Option(s): Networking and Security
& Software Engineering and Programming Systems

Supervisor(s)
Olivier BONAVENTURE

Reader(s)
David LEBRUN, Marc LOBELLE

Academic year 2015-2016

Abstract

This thesis is built around the desire to develop a tool to help Computer Sciences students learning the C programming language. The developed tool is aimed to be added to the existing online `INGInious` platform developed within the `INGI` department of the Louvain School of Engineering. The students can already submit their C programs on this tool, to be evaluated. Nevertheless, it only provides a binary feedback (failed or succeeded). The idea is to provide a better feedback on the submitted codes, based on the analysis of the programs executions and on a graphical description of the memory used in them. In that way, the students could better understand the errors in their programs and use these feedbacks to correct their submissions and thus, improve their skills in the C programming language.

This is greatly inspired by a personal project of the Dr. Guo of the University of Rochester, the `Online Python Tutor` which is a tool providing an excellent way to learn `Python` programming online.

The main technology used for this thesis is the `Valgrind` framework along with its powerful `Memcheck` tool used for tracking memory leaks and bugs inside C programs.

Acknowledgments

First of all, I would like to thank my thesis supervisor, Professor Bonaventure, for his availability, his patience and his help.

A special thanks to David Lebrun, for his patience when helping me debugging problems, and for his support during the realization of my thesis.

Thanks to Pierre Reinbold for his help with the INGInious platform and for the web server.

Thanks to my parents, Werner Ooghe and Nathalie De Wijngaert for their support.

Thanks to my friends Paul Vanhove, Xavier Planckaert, Laurent Mondry, Florent Gos, and Thibault Delaey for their support and for keeping myself motivated.

Contents

Chapter 1

Introduction

1.1 Learning C programming

To briefly introduce the language, we can say that it is a general-purpose programming language, originally developed by Dennis Ritchie. C is "low-level". That means that it deals with the same sort of objects that most computers do. These objects are numbers, characters, and addresses [14]. One of the language features is that the programmers have to manually manage the memory used by the programs they develop. This is in contrast to high-level programming languages like Java or Python. This characteristic is interesting because it means coders have to make an effort to properly manage the memory used by their programs. They have the ability to dynamically allocate memory for their variables, data-structures, etc. It is important to correctly compute the memory needed by a specific data-structure, and to free the memory that is no more used at a point of the execution to avoid memory leaks.

The memory management in C has a steep learning curve. Lots of students often have problems to understand how to handle pointers, those variables that contain other variables addresses. This can lead to various errors like dereferencing a freed pointer. However, different tools exist to help programmers track memory leaks, unauthorized memory accesses, etc. Such tools are for example `Valgrind` or `gdb`. However, those tools are often hard to handle for people who just started diving into the language.

This thesis aims to help Computer Sciences students learn the C programming languages and more particularly to help them understand errors they encounter while trying to manage the memory of the computer. The goal is to develop a tool giving an illustration of how the memory evolves during the execution of a program in a step-by-step manner. This is inspired by existing tools discussed in section 1.3.

1.2 INGInious

Eventually, the developed tool will be included in the INGInious platform in order to provide an improved feedback on the solutions submitted by students on C programming exercises dealing with memory management. Typical exercises involve programs manipulating data-structures. The INGInious platform is a tool developed by the INGI department of the Louvain School of Engineering which aims to automatically grade students exercises submissions for various Computer Sciences courses. The submitted codes are checked using scripts responsible for compiling (if needed), executing, and testing them. Figure 1.1 shows an example of exercise that could be made on the platform. The feedback provided by INGInious is binary. It is success or failure. As a result of this, students do not have precise information on what is causing the failures assuming that the submitted code can be compiled without errors.

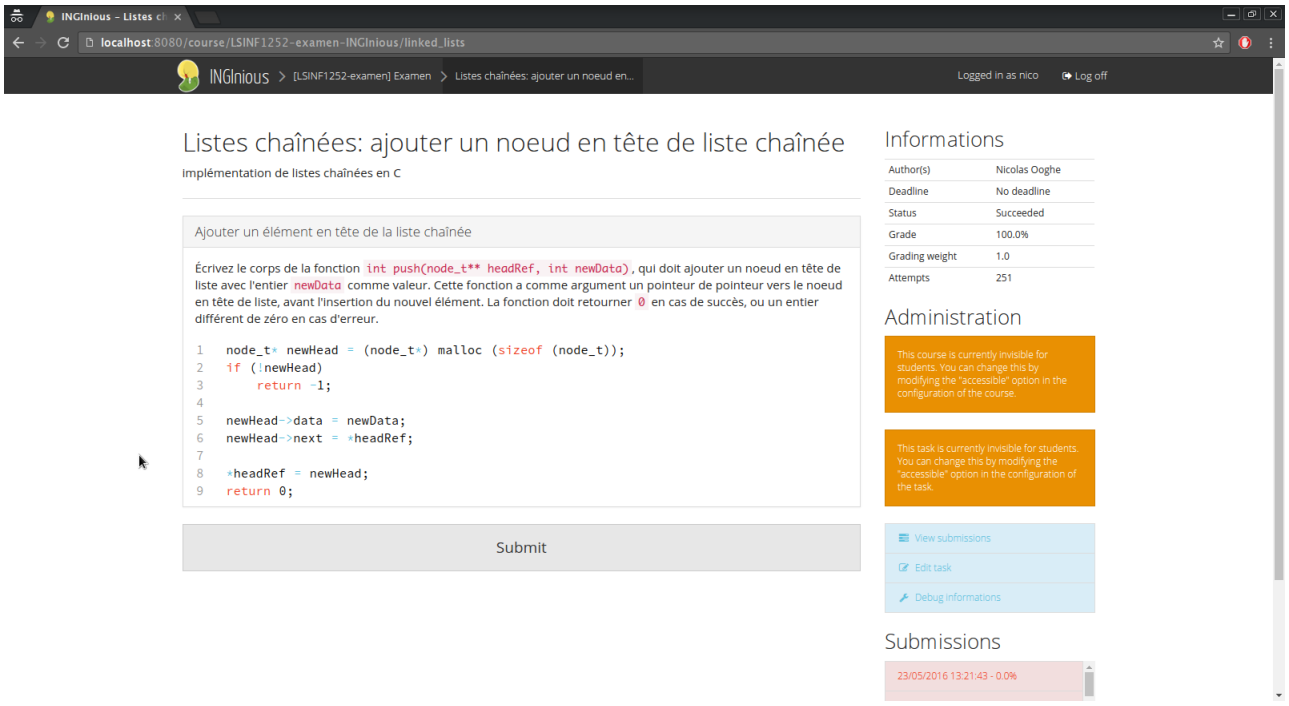


Figure 1.1: The INGInious platform

Even though the first goal of the tool was to be integrated into INGInious to improve the feedback given to the students, it is totally independent of INGInious. It can be used as a standalone tool. The idea is to make a free software out of it, enabling external contribution and integration into other tools of the same kind as INGInious.

1.3 State of the art

Nowadays, with the growing importance of Information Technology in our lives, there is an increasing demand for people proficient in coding. Consequently, lots of people want to learn programming and various tools to learn online begin to appear to help those people. These

existing tools differ on some points. There are applications that let you learn to code in different programming languages. An example is `Code School` [20], an interactive website that let you type and test your code directly in the application. It also provides lessons in video that you can watch before exercising in the proposed exercises. Another original example is `Scratch` [4], an online tool to learn to think creatively, systematic reasoning, and coding. It is designed for children at first, but it can be used by people of all ages. With `Scratch`, you can program little games and animations using images, and basic programming structures like `if ... else ...` statements, loops, and so on. It is intended to inspire children to code. It is visual, so users can observe what their code is actually doing.
To stay in the topic of visualization, there exist applications for actual programming languages like `Ruby`, or `JavaScript` that show you what your computer is doing when it executes your programs. Such an application is for example, the `Online Python Tutor` tool that let you type `Python` code directly in your web browser and see step-by-step what your code is doing when it is being executed. Using images along with graphs, allows the user to observe the evolution of his variables and data structures. It is a personal project of the professor Philip Guo from the University of Rochester [11]. Dr. Guo has developed this application to help Computer Sciences students, or anyone who wants to learn programming with `Python`. It could be an excellent supplement to the Computer Sciences courses. Indeed, seeing programs executions in images helps people understand what their code is actually doing and improves their development skills. For me, this is one of the best available educational project for Computer Sciences. There are still other interesting projects of the same kind to mention. `codecademy` [22], for example, is a free online application teaching how to code in various programming languages. It requires to sign up for a free account, so the progression of the user can be saved in it. Therefore, it allows the user to keep track of his progression.
Big tech companies like `Google` also propose educational tools to learn programming and other Computer Sciences related topics. So it is fair to mention the great `Codelabs` that `Google` offers to its users via the `Google Developers` website [6]. They propose coding tutorials along with exercises to allow users to learn by experience.

1.4 The C tutor tool

To build an application that provides a visual feedback on C programming exercises, it is necessary to divide the problem into smaller subproblems and link the solutions to these problems together to build the final tool. There is no need to modify INGInious in this process. The different problems to solve are the following ones :

- Getting the execution logs of the C program
- Transforming the execution logs
- Building graphs representing the memory manipulated by the programs

- Feedback construction
- Integration in INGInious

We will see how these problems are resolved throughout this thesis and how the solutions have been implemented.

1.5 Contributions

In section 1.3, I have presented existing tools that provide online learning. It is a good idea to try to include such tools in the learning materials of the Computer Sciences courses in schools or universities. This is the aim of this thesis. I extend the previously defined INGInious platform of the Computer Sciences department of the Louvain School of Engineering which is used by teachers and their assistants to correct codes submitted by their students. For my thesis, I focus on the development of the tool for the Computer Systems course. In that course, students have to write codes in the C programming language. The extension gives a better feedback to the students, showing them the execution steps of the submitted C codes. As already mentioned, the tool is independent of INGInious, so it could be used in other environments. It could be integrated on another platform of the same kind as INGInious. The source-code of the application can be reused or modified for this. This is an efficient way to benefit from the assets of the educational tools like those defined in the previous section directly in the courses materials of universities and schools.

Chapter 2

Valgrind and Memcheck

2.1 Introduction

Valgrind and more particularly its Memcheck tool [5] is the central piece of the developed application. It is used to get logs file describing the execution steps of the analyzed C programs from which an improved feedback can be given to the students. Such a feedback enables to visually observe the execution of their program along with the evolution of the memory, to better understand the operations of the program. To produce those logs files, Valgrind has been slightly modified. For my thesis, I use the modified version proposed by Dr. Guo of the university of Rochester of his `Online Python Tutor` project [11]. One of the original goal of the thesis was to carry on with the modification of the Valgrind code in a way to get more information on C codes manipulating function pointers. However, this exercise is more difficult than expected due to the extreme complexity and the size of the Valgrind code. In any case, it is still interesting to have an overview on how this famous programming tool works. This allows us to better understand how it can be used to get information on the memory used during the execution of the C programs.

This chapter aims to explain the inner working of Valgrind and the modifications in the version used in the application. In my attempt to better understand how Valgrind works, I have used some research papers that discuss many subjects about Valgrind.

2.2 Valgrind

Valgrind is a dynamic binary instrumentation framework designed to develop dynamic binary analysis tools. Dynamic Binary Analysis tools are used to analyze programs at runtime [18]. This is the opposite case of unit testing or integration testing that perform tests on the source-code directly. Such analysis are performed by executing the analyzed program on a host processor or virtual processor. Binary analysis does not require the source code of the analyzed programs. In the context of the application developed for this thesis, it is interesting to have the ability to

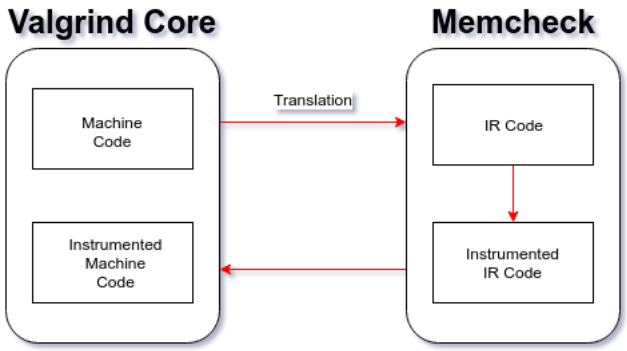


Figure 2.1: Valgrind and Memcheck

test the programs in a different way, since one of the goal of the project is to be included in the `INGInious` platform that currently only provides unit tests. Valgrind was first released in the year 2002 and is written in C programming language [15].

The Valgrind architecture is composed of two main elements. The Valgrind core is the dynamic binary instrumentation framework. The second element is constituted by the tools which are pluged into the Valgrind core. The most famous Valgrind tool is certainly `Memcheck`. But other `Valgrind` tools exist. I will detail this tool later in that chapter. To dynamically generate code to instrument the programs, Valgrind uses a `JIT` compiler and dynamic binary recompilation. Instrumenting programs using this technique affects the performances of the hosted programs [9]. The Valgrind core translates the machine code into an intermediate language called the Intermediate Representation (`IR`) which is architecture independent. This code representation technique is called `disassemble-and-resynthetise`. After the machine code is converted to the `IR` language, it is being instrumented by adding additional instructions to it. This representation is then reconverted to machine code. [18]. This is needed by the tools like memcheck to be able to instrument the program and thus perform analysis on it. The fact that the intermediate language manipulated by the Valgrind tools is platform independent has the consequence that the plugin tools like Memcheck are also architecture independent [17]. Figure 2.1 shows the architecture of that system [13].

Valgrind uses Shadow memory techniques. This technique is used to get information on the memory of the computer during the execution of a program. Dynamic Binary Analysis tools use this technique to mark the memory used by the programs during their execution with `shadow memory values` that store information about the memory [17]. It is by using this technique that tools like Memcheck can detect illegal memory access like segmentation faults.

Various other Valgrind tools could be presented. An interesting example is Helgrind, which aims to help programmers dealing with multi-threaded programs. It is useful for detecting data races, meaning that memory accesses between threads are performed without adequate locking

```
1  int main (int argc, char** argv)
2  {
3          int i; // Uninitialized!
4          while (i < 2)
5          {
6                  printf("I use an uninitiazed variable i : %d\n", i);
7                  i++;
8          }
9          return 0;
10 }
```

Figure 2.2: C program using an undefined value in loop condition

or synchronization [1]. In Computer Sciences studies, programming multi-threaded programs is often required. In this thesis, we do not discuss the multi-threading issues and instead focus on pure memory management.

2.3 Memcheck

Memcheck is a tool implemented using the Valgrind framework to detect memory errors in programs. It is also written in C like Valgrind. Memcheck is able to detect lots of memory errors such as undefined value errors [16], illegal memory access, double free, and memory leaks. It is the most popular Valgrind tool. As explained in the previous section, Memcheck uses Shadow memory technique to tracks those memory errors. This technique allows Memcheck to have additional information on the memory used by the running programs. Memcheck stores information about the memory of the computer during the execution of the compiled C programs. All the bytes used by programs are marked with a single `A` bit. This bit is responsible to verify that the program has the right to access those bytes of memory. Bytes are also marked with eight `V` bits that says if those parts of memory are properly initialized or not. The `V` stands for "validity". Data bits are shadowed with `V` bits that indicate if their corresponding bits are properly defined or not. When a program performs an operation that uses values as input and if the operation alters the behaviour of the program, the `V` bits of those values are checked. This is to avoid a program to perform operations on undefined values [16]. Such operations could be disastrous and often really difficult to debug. Memcheck can then issue an error in those cases. The C code in figure 2.2 is an example of C program that uses an uninitialized variable inside the condition of a while loop. Here, the value of `i` could be any integer. When running Memcheck on the compiled program, it raises an error telling the programmer that some operations in his program depends on uninitialized values 2.3. It is crucial to initialize variables when programming in C because they are not automatically initialized.

As another example, running the code from figure 2.4 with the Memcheck tool of Valgrind provides the output in figure 2.5, showing that the programmer attempted to write in a zone of

```
==10702== Conditional jump or move depends on uninitialised value(s)
==10702== at 0x400513: main (cond_jump.c:7)
==10702==
==10702== Conditional jump or move depends on uninitialised value(s)
==10702== at 0x4E7D570: vfprintf (in /usr/lib/libc-2.23.so)
==10702== by 0x4E84C78: printf (in /usr/lib/libc-2.23.so)
==10702== by 0x40050A: main (cond_jump.c:9)
==10702==
==10702== Use of uninitialised value of size 8
==10702== at 0x4E7A1CB: _itoa_word (in /usr/lib/libc-2.23.so)
==10702== by 0x4E7D89D: vfprintf (in /usr/lib/libc-2.23.so)
==10702== by 0x4E84C78: printf (in /usr/lib/libc-2.23.so)
==10702== by 0x40050A: main (cond_jump.c:9)
```

Figure 2.3: Valgrind output in case of undefined value error

```
node_t* aNode = (node_t*) malloc (sizeof (node_t));
aNode->data = 42;
aNode->next = &nextNode // Defined earlier
free (aNode);
       ...
aNode->data = 43; // Bad !
```

Figure 2.4: Common error

the memory that was previously freed.

Memcheck also keeps track of the dynamically allocated memory on the heap to check for double frees or memory leaks (missing free operations). To do that, it keeps the location of each heap blocks in a hash table [17]. Errors are also raised in case of these kind of memory management errors. Figure 2.6 shows an example of a little C program in which I try to deallocate the memory block pointed by the `pNode` pointer two times. In that case, Memcheck outputs the error depicted in figure 2.7.

Memcheck is therefore a really efficient tool to detect errors related to memory management in C programs. It is hard to avoid using it when developing programs in C. On the contrary, I would say it is totally worth using it. In the next section I will detail how we can get the

```
==24196== Invalid write of size 4
==24196== at 0x400579: main (exemple.c:10)
==24196== Address 0x51d7040 is 0 bytes inside a block of size 16 free'd
==24196== at 0x4C2BCEA: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==24196== by 0x400574: main (exemple.c:9)
==24196== Block was alloc'd at
==24196== at 0x4C2ABD0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==24196== by 0x40054E: main (exemple.c:6)
```

Figure 2.5: Valgrind output exemple

```
1  int main (int argc, char** argv)
2  {
3    node_t* pNode = (node_t*) malloc (sizeof (node_t));
4    free (pNode);
5    free (pNode); // free an already freed variable!
6    return 0;
7  }
```

Figure 2.6: Double free in C program

```
1  ==11172== Invalid free() / delete / delete[] / realloc()
2  ==11172== at 0x4C2BCEA: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
3  ==11172== by 0x40056A: main (double_free.c:8)
4  ==11172== Address 0x51d7040 is 0 bytes inside a block of size 16 free'd
5  ==11172== at 0x4C2BCEA: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
6  ==11172== by 0x40055E: main (double_free.c:7)
7  ==11172== Block was alloc'd at
8  ==11172== at 0x4C2ABD0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
9  ==11172== by 0x40054E: main (double_free.c:6)
```

Figure 2.7: Valgrind output in case of invalid free operations

memory information used by Valgrind for every execution steps of C programs. This part is needed by the application to be able to represent the state of the memory during the execution of the programs.

2.4 Modifying Valgrind and Memcheck

The Valgrind code is really hard to understand and there is almost no official documentation excepting the presence of comments in the code. Moreover, it is highly recommended by the developers not to use anything from the `libc`. The reasons for that are not very clear, but I guess it's a precaution to avoid conflicts between the client program (the one that is being analyzed) and the Valgrind process. In consequence of that, the Valgrind developers have re-implemented a subset of the functions of the standard library.

The modifications and additions into the source code of Valgrind and Memcheck are mostly used to collect information about the local variables in the stack frames, global variables and allocated heap blocks at each step of the execution of the client C program. These data are written in log files. The majority of the modifications aims to print information into a file instead of into the terminal. The log files are written using JSON syntax. We will show examples of such files in the next chapter.

2.5 Conclusion

In this chapter, I have described the inner working of Valgrind, a Dynamic Binary Instrumentation framework and Memcheck, a dynamic binary analysis tool implemented with Valgrind. The latter is used to detect memory errors in programs during their execution. We have also seen some examples of memory errors that Memcheck can detect. The power of Memcheck is well established. It is a very powerful tool that every C programmer has to own in his toolbox. For this thesis, Memcheck is used to obtain information about the memory manipulated by C programs in order to build a tool visually showing the step-by-step evolution of the memory used by students codes. In the next chapters we will see how this information is used to construct the application.

Chapter 3

The Valgrind logs and their processing

3.1 Introduction

This chapter talks about the execution log files created by Valgrind and their processing. In order to develop a tool graphically showing the memory evolution during programs executions, we need to obtain the complete execution trace of any C program in a format that allows us to easily process them. But we need to transform these raw traces into an easier processable format and use them to build the student feedback. In this chapter, I show what kind of traces Valgrind outputs in different situations and what information we can find in those traces. After that I explain the format type we need to build from these raw traces in order the construct the feedback application. The Valgrind traces are obtained using Philip's Guo modified version [10]. The next chapters will talk about the visualization part of the application namely the construction of the graphs showing the memory state at each execution steps, and the building of the final application. The complete architecture of the tool is depicted in figure 3.1. The source code is first obtained from a student's exercise on the INGInious platform. Note that this is optional, we could also directly use the application on any C code. The code is then compiled and passed to Valgrind's Memcheck tool in order to get the execution logs. Then these log files are sanitized to obtain the format required by the developed application.

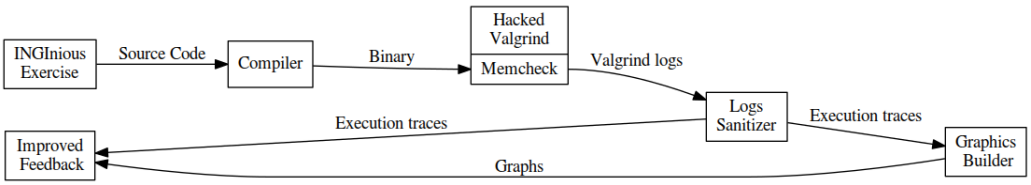


Figure 3.1: Architecture

3.2 Programs execution logs

The execution log files describe the execution of C programs in a step-by-step manner. Each execution point record contains information about the memory, the function that is currently being executed, the corresponding line number in the source code, the stack frames local variables, etc. The log files can be large in function of the size of the programs in lines of code. Although, the size of these files is not really proportional to the amount of memory used by the analyzed program. Of course, the size of the stack frames records can grow fast since they depend on the number of local variables they contain but the state of the heap is not depicted in the execution logs. Other information on events like function calls, function returns, program exits, segmentation faults, etc are not present in these raw execution traces. However, we have all the required data to infer those details from the Valgrind logs. We will see later how we can define the state of the heap and the other missing information so we can have them in the final execution trace format.

In order to detail the Valgrind execution logs, I will use simple C programs that use my implementation of linked list data structures. I will perform some operations on them to see what happens in the execution logs when a function is called, or when a runtime error arises.

Let's see what information we can obtain using a program that simply initializes a linked list with one node and then adds a new node to the beginning of the list. The second operation is performed using a special function that makes the new node the head of the linked list, and the initial head node, its "next" node. With this program example, we can already observe the execution of a code manipulating pointers and performing function calls and returns. The figure 3.2 depicts the implementation of this program.

The figure 3.3 shows the first execution step of the program in the corresponding Valgrind execution log. This record shows the function that is currently being executed. Since it is the beginning of the execution, we are in the main function of the program. It says that this state corresponds to the 6th line in the source code of the program. The address of the instruction pointer, also called the program counter, is present as well. The `kind` field is a simple bit that is set to "1" if it corresponds to the main function of the program, and "0" otherwise. It also contains all the already declared global variables. Obviously they are none for now and it will remain like that since we do not declare any global variable in the studied program. Information regarding the stack frames are present as well. We can observe detailed information about the local variables and functions arguments such as their addresses, kind, types, sizes and values. The corresponding stack pointers and frame pointers are available in the stack frames description. As a reminder, a stack frame always correspond to a subroutine. It contains the arguments of the called function, its local variables, and its return address. The stack pointer is the address of the top of the call stack. The frame pointer contains the address of the start of the stack

```
1  int main (int argc, char** argv)
2  {
3    node_t* head = malloc (sizeof (node_t));
4    head->data = 18;
5    head->next = NULL;
6    push (&head, 17);
7    free_list (&head);
8    return 0;
9  }
10 ...
11 int push (node_t** headRef, int newData)
12 /* insert a new node at the beginning of the list with 'newData' as data
13    returns 0 if the function succeeds
14    returns -1 on failure */
15 {
16   node_t* newHead = (node_t*) malloc (sizeof (node_t));
17   if (!newHead)
18     return -1;
19
20   newHead->data = newData;
21   newHead->next = *headRef;
22   *headRef = newHead;
23   return 0;
24 }
```

Figure 3.2: Initialize a linked list and push a new node at the beginning of it

```
1  === pg_trace_inst ===
2  {
3  "func_name": "main", "line": 6, "IP": "0x4006D6", "kind": 1,
4  "globals": {},
5  "ordered_globals": [],
6  "stack": [
7  {"func_name":"main", "line": 6, "SP": "0xFFF000368", "FP": "0x400F80", "locals": {
8    "argc": {"addr":"0xFFF00034C", "kind":"base", "type":"int", "size":4, "val":"<UNINITIALIZED>"}
9  , "argv": {"addr":"0xFFF000340", "kind":"pointer", "size":8, "val":"<UNINITIALIZED>"}
10 , "head": {"addr":"0xFFF000358", "kind":"pointer", "size":8, "val":"<UNINITIALIZED>"}
11 },
12 "ordered_varnames": ["argc","argv","head"]}]
13 }
```

Figure 3.3: Execution step in Valgrind trace

frame of a subroutine. Figure 3.4 shows a typical call stack (from `http://bit.ly/1YajkJt`).

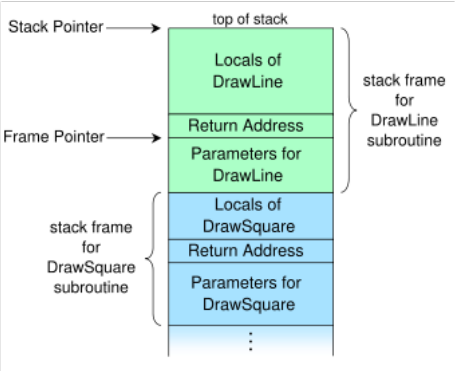


Figure 3.4: Call Stack

Now let us see what is shown in the Valgrind logs when a function call event happens and more particularly when the "push" function call is performed during the execution of our present program. The figure 3.5 shows the execution step corresponding to this function call. (Obviously, I am not talking about a push operation onto the call stack of the program. It corresponds to the call to the "push" function that adds an element at the beginning of the list in the analyzed program). The stack now holds a new stack frame. The one corresponding to the "push" function. As for the "main" stack frame, information regarding the local variables and arguments are found. It is also interesting to observe that at this point of the execution, the local variables and arguments of the "main" function are all initialized. It is stated that the "head" pointer now points to a heap block containing the data of the first node of the linked list. So there is no information regarding the entire state of the heap. Instead, the detailed descriptions of the dynamically allocated variables are directly located in the descriptions of the pointer variables. Using those data, it is possible to construct a detailed description of the heap memory at each point of the execution. We will see how this is achieved when the Valgrind log traces are processed to construct the trace files used by the application.

At program exit, the logs look like the first execution step of the program except for the "main" local variables and arguments which are now initialized. Actually, events like function calls, function returns and program exits are not precisely written in the Valgrind log files. We can infer those events by observing how the call stack of the program evolves. A growing call stack corresponds to a function call. As shown in the last example, a new stack frame is added onto the stack when a function is called. A narrowing call stack corresponds to a function return, obviously. And the last execution step always corresponds to the termination of the program execution.

Error cases like segmentation faults are not described in the Valgrind log traces. If an error is encountered during the execution of the analyzed program, the last execution point in the logs

```
=== pg_trace_inst ===
{
"func_name": "push", "line": 218, "IP": "0x400BBE", "kind": 0,
"globals": {},
"ordered_globals": [],
"stack": [
{"func_name":"push", "line": 218, "SP": "0xFFF000338", "FP": "0xFFF000360", "locals": {
  "headRef": {"addr":"0xFFF000318", "kind":"pointer", "size":8, "val":"<UNINITIALIZED>"}
, "newData": {"addr":"0xFFF000314", "kind":"base", "type":"int", "size":4, "val":"<UNINITIALIZED>"}
, "newHead": {"addr":"0xFFF000328", "kind":"pointer", "size":8, "val":"<UNINITIALIZED>"}
},
"ordered_varnames": ["headRef","newData","newHead"]},
{"func_name":"main", "line": 10, "SP": "0xFFF000340", "FP": "0xFFF000360", "locals": {
  "argc": {"addr":"0xFFF00034C", "kind":"base", "type":"int", "size":4, "val":1}
, "argv": {"addr":"0xFFF000340", "kind":"pointer", "size":8, "val":"0xFFF000448"}
, "head": {"addr":"0xFFF000358", "kind":"pointer", "size":8, "val":"0x51D7040", "deref_val":
{"addr":"0x51D7040", "kind":"heap_block", "val": [
  {"addr":"0x51D7040", "kind":"typedef", "type":"node_t", "val":
  {"addr":"0x51D7040", "kind":"struct", "type":"node", "val": {
  "data":{"addr":"0x51D7040", "kind":"base", "type":"int", "size":4, "val":18},
  "next":{"addr":"0x51D7048", "kind":"pointer", "size":8, "val":"0x0"}}}}]}}
},
"ordered_varnames": ["argc","argv","head"]}]
}
```

Figure 3.5: Function call

is the one just before the execution point where the error arises and thus, program exits. We will see in the next section and in the implementation chapter how we detect runtime errors in the context of the application.

In these log files, we have all the data regarding the memory used by the programs at each step of their execution. We have seen that some data is missing. This data is details about the various events occurring during the execution of a program. Such event are function calls, function returns and runtime errors. We will see in the logs processing section that we can deduce when those events arise during the execution of the C programs.

3.3 Valgrind logs transformation

This section details the intermediate format of the execution traces that we need in order to be able to build a visual representation of the memory during the execution of the programs. This section also discusses the architecture of the part of the application responsible for the transformation of the Valgrind log files into this intermediate format. Figure 3.6 is a global vision of the presented architecture.

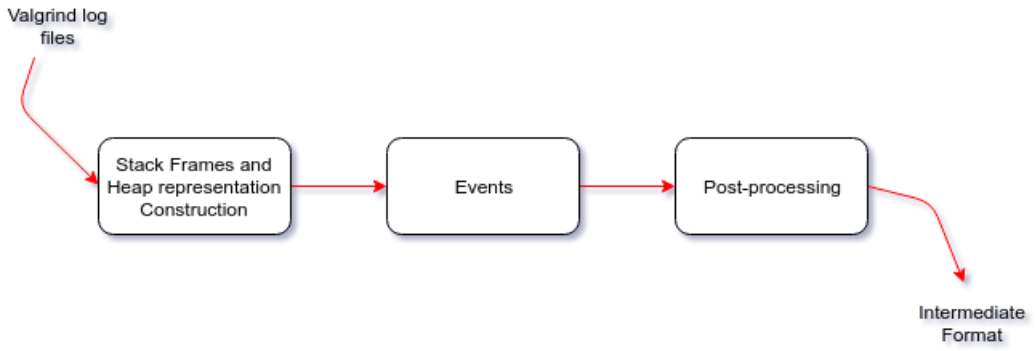


Figure 3.6: Valgrind traces transformation

We need files that contain all the information that we already have in the Valgrind log traces but in a simplified form. Furthermore we need to have information related to the events encountered during the execution of the programs like function calls, function returns and runtime errors which we have already discussed in the previous section of the present chapter. In that way we can display this information in the students feedback on the INGInious platform. The main additional item we need, is a description of the evolution of the heap state along with the program execution. It is mandatory to have that information, since the main goal of the tool is to show students how their solutions to INGInious exercises manipulate pointers to dynamically allocate memory for their data-structures. Building graphs depicting the heap blocks and stack frames variables would be a painful task to manage if we use the log files in their current format. What we need is an intermediate format that would allow us to build the graphs in the most efficient way. Moreover, as it is indicated in the architecture of the tool in figure 3.1, this intermediate format is not only used to build the images used in the application. It is also required in the application itself. Effectively, we also need this data inside the feedback tool so as to display suitable information to support the graphs.

3.3.1 Heap and Stack Frames states

To describe this intermediate format, I will reuse the little C program of the previous section so that we can compare the results with the already described Valgrind traces. The intermediate format has to contain all the required information to run the feedback application. The construction of the images and the data displayed in the tool only use the format explained in this section. So it needs to be as complete as possible. This format has been adapted from the format used in the open source `Online Python Tutor` application [11] that I have already mentioned in the previous chapters. It has been simplified since the most important information we need for the developed tool is the information about the memory used by the programs. Figure 3.7 is an example of the format we want to obtain after the Valgrind log files transformation. This example shows us the state of the memory in the first execution point of the C program of figure 3.2. It looks quite different from the corresponding execution step description in the

Valgrind log files. As required, there is a heap representation (In the example, it is empty since this is the description of the first step of the execution of the program). There is also a more precise and readable description of the stack frames content. Each stack frame description has a complete picture of the state of its local variables and arguments with their types, addresses and values. We need the same information for the data located on the heap part of the memory. Therefore, the part of the application responsible for building the intermediate representation has to process the Valgrind logs in order to build a good description of the heap state. To achieve this, it has to search the Valgrind traces for pointer variables pointing to heap blocks. We can therefore use those heap blocks description to build a representation of the heap in the intermediate format. The result will look like in the figure 3.8. In that example, the heap contains a `node_t` structure with its `data` and `next` fields descriptions. We can observe that the value of the `data` field is set to "18" and the value of its `next` field is set to the "0x0" address, meaning that this node is the last element of the linked list at that point of the execution.

```
    "ExecutionPointNumber": 0,
    "event": "step_line",
    "func_name": "main",
    "globals": {},
    "heap": {},
    "line": 6,
    "Stack_Frames": [
      {
        "encoded_locals": {
          "argc": [
            "C_DATA",
            "0xFFF00034C",
            "int",
            "<UNINITIALIZED>"
          ],
          "argv": [
            "C_DATA",
            "0xFFF000340",
            "pointer",
            "<UNINITIALIZED>"
          ],
          "head": [
            "C_DATA",
            "0xFFF000358",
            "pointer",
            "<UNINITIALIZED>"
          ]
        },
        "frame_id": "0x400F80",
        "func_name": "main",
        "is_highlighted": true
      }
    ]
```

Figure 3.7: Intermediate format

```
"heap": {
  "0x51D7040": [
    "C_ARRAY",
    "0x51D7040",
    [
      "C_STRUCT",
      "0x51D7040",
      "node_t",
      [
        "data",
        [
          "C_DATA",
          "0x51D7040",
          "int",
          18
        ]
      ],
      [
        "next",
        [
          "C_DATA",
          "0x51D7048",
          "pointer",
          "0x0"
        ]
      ]
    ]
  ]
}
```

Figure 3.8: Heap representation

3.3.2 Events setting

As explained in the previous section, we need additional information regarding the events arising during programs executions as well. Each execution step description has to show which type of event occurs at that point. In the example of figure 3.7, no special event occurred at this point. So it simply tells that the program continues its execution with the next line of code. In case of a function call, a "call" event has to be mentioned in the description of the execution step. Such an event can be detected by checking the presence of an additional stack frame from one execution point to another. The same method can be applied for functions returns detection. If the stack frame of a function is present in an execution point but not in the following one, it means that the function has terminated its execution. That way, we know when an execution point corresponds to a function return. This execution points are marked with a "return" event. Figure 3.9 shows the stack frame state representation of the "push" function when it is being called in the program of figure 3.2. One can notice the description of its two arguments and its local variable : `newHead`.

```
"Stack_Frames": [
  {
    "encoded_locals": {
      "headRef": [
        "C_DATA",
        "0xFFF000318",
        "pointer",
        "<UNINITIALIZED>"
      ],
      "newData": [
        "C_DATA",
        "0xFFF000314",
        "int",
        "<UNINITIALIZED>"
      ],
      "newHead": [
        "C_DATA",
        "0xFFF000328",
        "pointer",
        "<UNINITIALIZED>"
      ]
    },
    "frame_id": "0xFFF000330",
    "func_name": "push",
    "is_highlighted": true
  }
]
```


Figure 3.9: Stack frame representation

3.3.3 Runtime Errors Detection

A type of event that is interesting to be included in the execution points of the intermediate trace format is the runtime errors and more particularly segmentation faults. Since we are building an application that will serve as a feedback for students exercises, it is useful to be able to inform the user when such errors arise during the execution of the program. With a complete execution point description corresponding to a runtime error, we are able to show students the state of the memory at that moment and the line of code that is responsible for the error. Segmentation faults are the kind of errors that are the most likely to happen during the execution of a program manipulating pointers. Therefore we also need to find a solution to handle those cases. Consider the bogus C program depicted in figure 3.10. The purpose of this program is to add a node at the end of a linked list data structure. The `addTail` function is typically the kind of function that we could ask students to implement as an exercise. In the example, the program will produce a segmentation fault error because it tries to set the `data` field of the new `node_t` structure using a pointer to it, while the value of that pointer has been mistakenly set to `NULL`. The last execution point of this program will contain an event telling that the program has terminated with a segmentation fault.

```
1  int addTail (node_t* head, int newData)
2  {
3    node_t* tail = NULL;
4    tail->data = newData; // Will procude a segfault error
5    tail->next = NULL;
6    node_t* curr = head;
7    while (curr->next)
8      curr = curr->next;
9    curr->next = tail;
10   return 0;
11 }
12
13 /* ... */
14
15 int main (int argc, char** argv)
16 {
17   node_t* head = (node_t*) malloc (sizeof (node_t));
18   head->data = 18;
19   head->next = NULL;
20   addTail (head, 89);
21   free_list (&head);
22   return 0;
23 }
```

Figure 3.10: Bogus C program

Figure 3.11 shows the (shortened) execution point representation where the segmentation fault is triggered, in the intermediate format. Using it, we can tell the application that the segmentation fault arises when executing line 233 and also provide a complete description of the heap and the stack frames at that exact moment.

```
1     {
2       "ExecutionPointNumber": 10,
3       "event": "segfault",
4       "func_name": "addTail",
5       "globals": {},
6       "heap": {
7                ...HEAP DESCRIPTION...
8       },
9       "line": 233,
10      "ordered_globals": [],
11      "Stack Frames": [
12        ...STACK FRAMES DESCRIPTION...
```

Figure 3.11: Segmentation fault in Intermediate format

3.4 Conclusion

In this chapter, I have detailed the type of log files describing the complete executions of C programs that we can get using the Valgrind framework and its powerful Memcheck tool. We

have seen that these log files do not contain enough information to build a sound tool showing the execution of C programs in a step-by-step manner. Therefore we needed to find a solution to get the missing information. Those data could be computed from the logs produced by Valgrind and Memcheck. The architecture of this solution has been detailed. The result is an intermediate format containing a complete representation the memory state and execution state of the programs at each point of their execution. One of the most important additional information that we needed was a representation of the heap, the part of the memory containing the dynamically allocated variables of the C programs. The events that occur during the execution of a program such as function calls and function returns are included in the intermediate format as well. Runtime errors like segmentation faults are also part of the events that could unfortunately happen during the execution of a program that misconducts the memory management. We have seen that they could be detected as well. The chapter dedicated to the implementation of the tool will provide more details about the construction of the execution trace files.

In the next chapter we will see the graphs that are used in the feedback tool in order to provide a visual representation of the memory used by the programs. These graphs are built using the intermediate execution trace format detailed in this chapter.

Chapter 4

Visualization

4.1 Introduction

This chapter introduces the graphs representing the state of the memory and their usage in the application that provide the feedbacks on the students exercises solutions from the INGInious platform. These graphs constitute one of the most important piece of the developed tool. Indeed, the goal of the application is to provide a feedback by showing the state of the memory at each point of the programs executions. Thoses images fill that requirement. They are built with the help of the files containing information about the execution of the programs. We have seen how we can obtain these files, in the previous chapter.

First, I will present the structure of the graphs and what kind of information we can find in them. I will then show some practical examples of graphs used in the application and how they are exploited in the improved feedback tool.

4.2 Structure

4.2.1 Graphs construction

The images describing the memory evolution are all attached to a specific execution point of the programs execution. Therefore, they are built one execution point at a time so that the final result could be compared to a movie or a sequence representing the evolution of the different parts of the memory over time. When used with a program that correctly manipulates dynamically allocated memory, the last image of the "movie" should depicts an empty representation of the heap. And if the "movie" ends badly i.e. with a non-empty heap representation, the students should understand that they forgot to free the associated memory blocks.

4.2.2 Design of the images

The structure of the graphs had to reflect the states of the call stack and the heap. We want to show how they evolve throughout the execution of the programs manipulating variables, heap

blocks and pointers. They have to use a simple way to link the pointers and the data they point to. So the structure of the images is split in two main sections. There is a section containing a visual representation of the stack frames of the active subroutines. Therefore, this section is itself split into as much sections as there are active subroutines at a particular point of the execution of the analyzed program. Each stack frame representation is labeled with the name of the corresponding function. In the figure 4.1 we can see an example of the stack of a C program at the beginning of its execution. Thus, there is a representation of the main stack frame and its local variables and arguments. The nodes describing these variables in the stack frames have the following form : a type field, showing the type of the variable, a field showing the name of the variable, its current value and its address in the memory of the computer. The heap section is also present in the example, simply showing that the program has not dynamically allocated heap blocks for now.

When a function call happens at a point of the execution, a new subsection representing the stack frame of the called function is added in the Stack Frames section. Then, the state of this new stack frame evolves as the corresponding function is being executed. At the point where the function returns from its execution, its stack frame is removed from the graph. The same procedure is followed along the program execution.

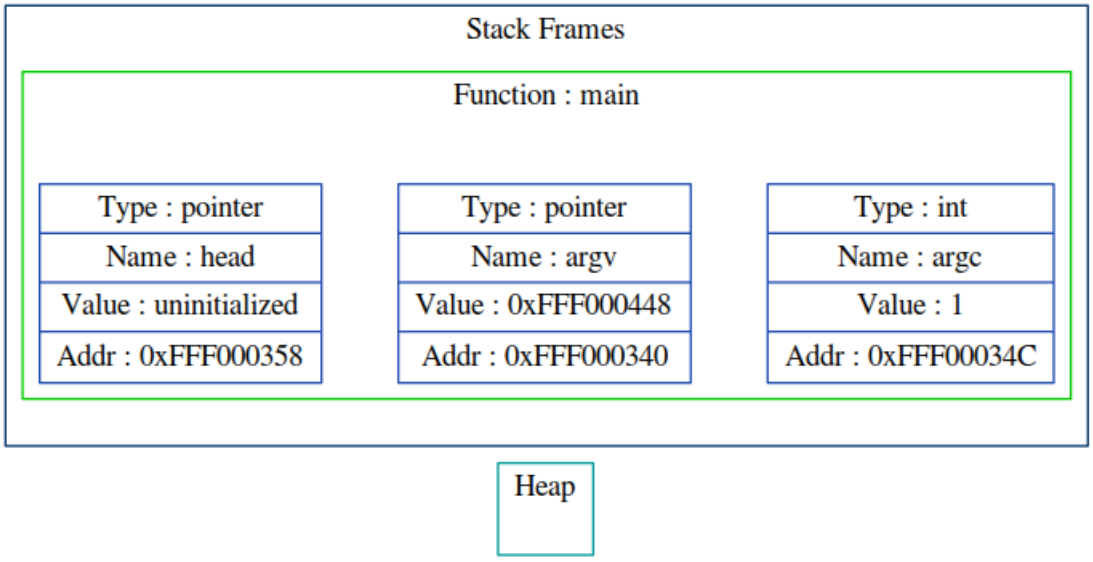


Figure 4.1: Call Stack representation

The other main section of the graph is the one representing the state of the heap. This section of the graph contains the dynamically allocated heap blocks of the programs. The heap blocks are represented with nodes containing a field to display the type of the associated data, the value of the variables, and their addresses. Other fields could also be present in the variables description in the case of structures for example. Arrows are also used in the graphs to show the relation between pointer variables and the data they points to. Figure 4.2 represents how

the graphs look like when heap blocks are in the heap section. We can also see that there is an arrow used to link a pointer variable in the main stack frame with its data on the heap.

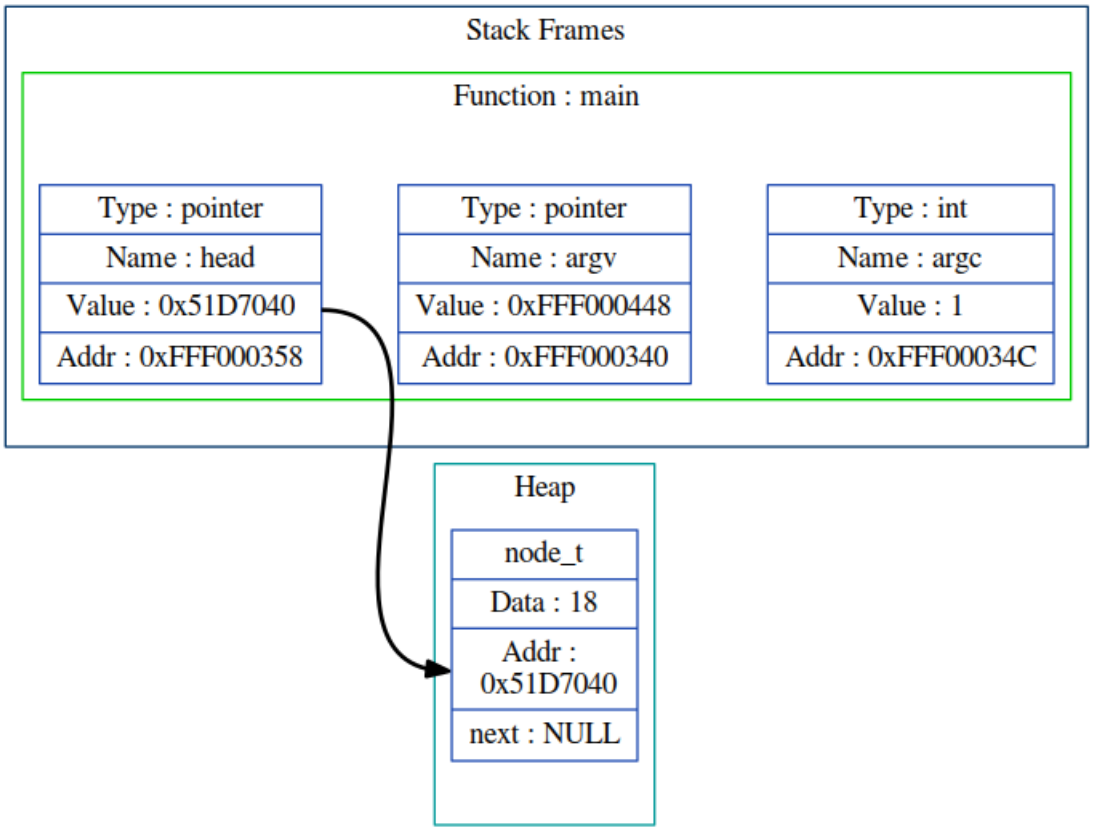


Figure 4.2: Call Stack and heap representations

4.2.3 Possible issues

Even if the feedback tool aims to be used with INGInious exercises submissions, it is difficult to keep control of the amount of data allocated on the heap. Consider an INGInious exercise for which the students have to dynamically allocate memory to add a single element in a data structure. There is a probability that some students could mistakenly allocate memory for a really big number of heap blocks. Therefore, the graphs representing the heap could grow really fast and become unreadable as in the figure 4.3. We have to prevent those cases because in addition to being unreadable, such graphs take a very long time to generate since all the intermediate states of the heap have to be represented as well. I will detail such cases in the performances chapter. It is possible to avoid that kind of issue by checking the submitted code using unit tests in the INGInious tasks or by verifying the heap state before the graphs generation. Nevertheless, it is still useful to double check that it won't happen.

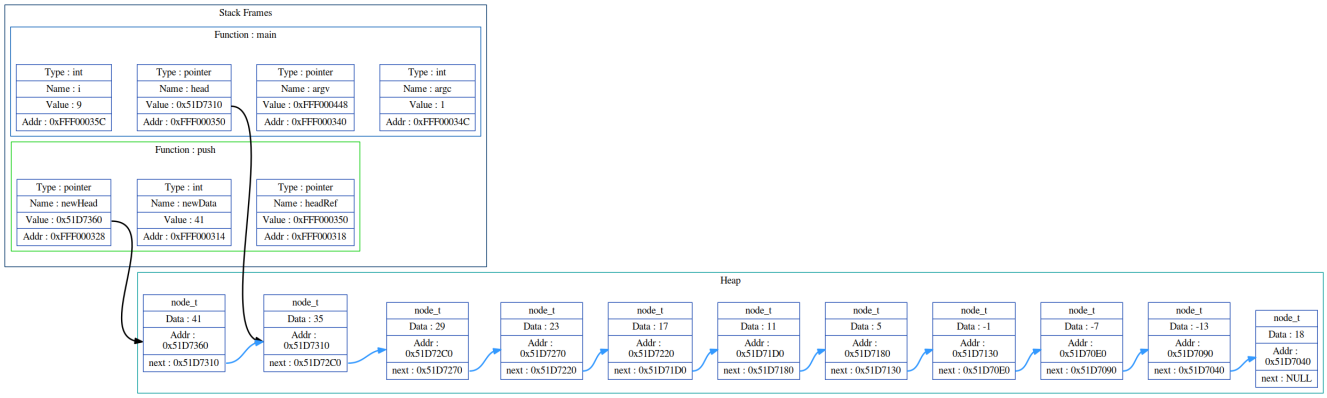


Figure 4.3: Too big heap state representation

This kind of problem could also happen if too much local variables are defined in the functions implemented by the students as shown in the figure 4.4. It is less likely to happen but checking for such cases is not pointless. Since the local variables and arguments of functions are directly present in the stack frames, a function with a really big amount of local variables does not increase the required time to generate the graphs. So the verification can be made on the intermediate trace format defined in the previous chapter, before starting the generation of the images.

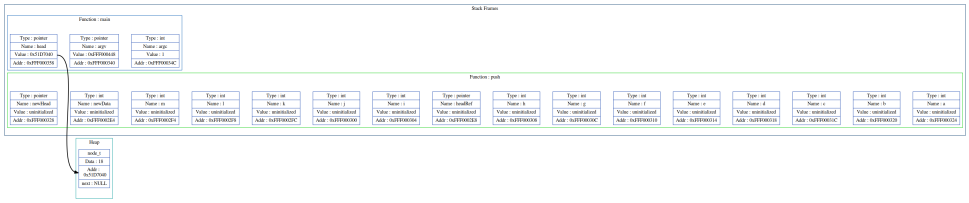


Figure 4.4: Stack Frame with too much local variables

4.3 Feedback

After generating the images that represent the memory state for each execution point, we have all the required elements to build the improved feedback pages presented to the users. The feedbacks are presented in the form of static web pages. The idea is to build the pages by putting a panel showing the code submitted by the student next to the graphs. The panel is constructed in the form of a browser that the user can use to navigate through the execution of the program one line of code at a time. It this also possible to go backwards in the execution or to directly jump at a specific execution point. The total amount of execution steps is displayed as well. Figure 4.5 shows the look of the execution steps browser used in the application.

Figure 4.5: Execution steps browser

While navigating through the execution, the graph corresponding to the current execution point is always displayed right to the code panel. The highlighted line of code in the browser is always the next line to be executed. So what the user sees in the graphs is always the state of the memory before the execution of the highlighted line. In the figure 4.6, there is an example of feedback page.

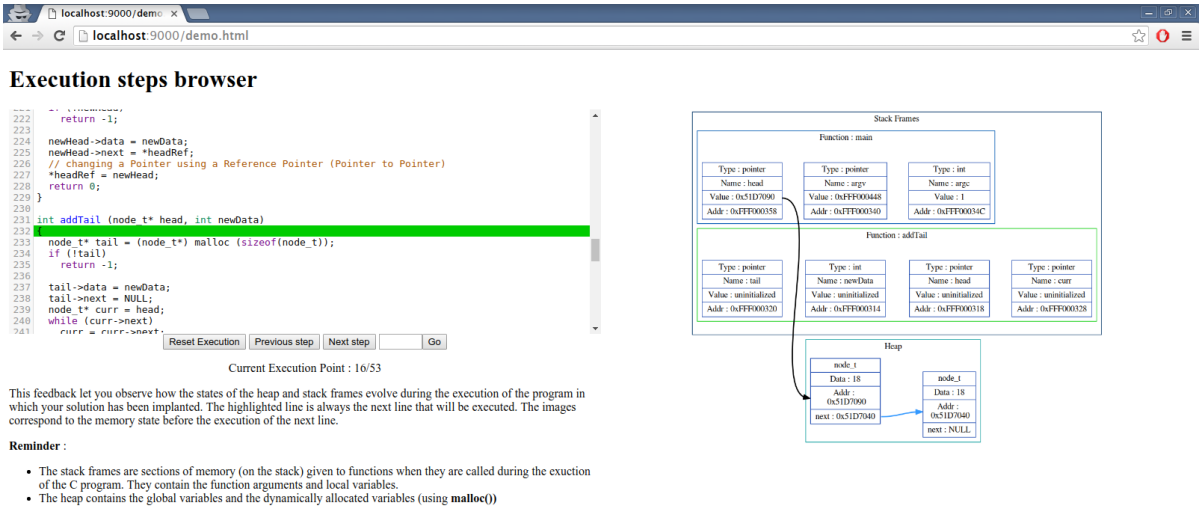


Figure 4.6: Feedback page appearance

The tool is also able to show when the code submitted by students triggers runtime errors. If a segmentation fault arises during the execution of the program, the tool shows the execution of the program until the execution point that is causing the error. The line of code that is responsible for the error is highlighted in red and the execution stops. That way, the user knows exactly which line of code is the root of the error and he can also have an overview of the memory when the program crashes. The figure 4.7 shows the tool when a segmentation fault arises during the execution of the program. The line is highlighted and a message is displayed to tell the user that this line is responsible for the error.

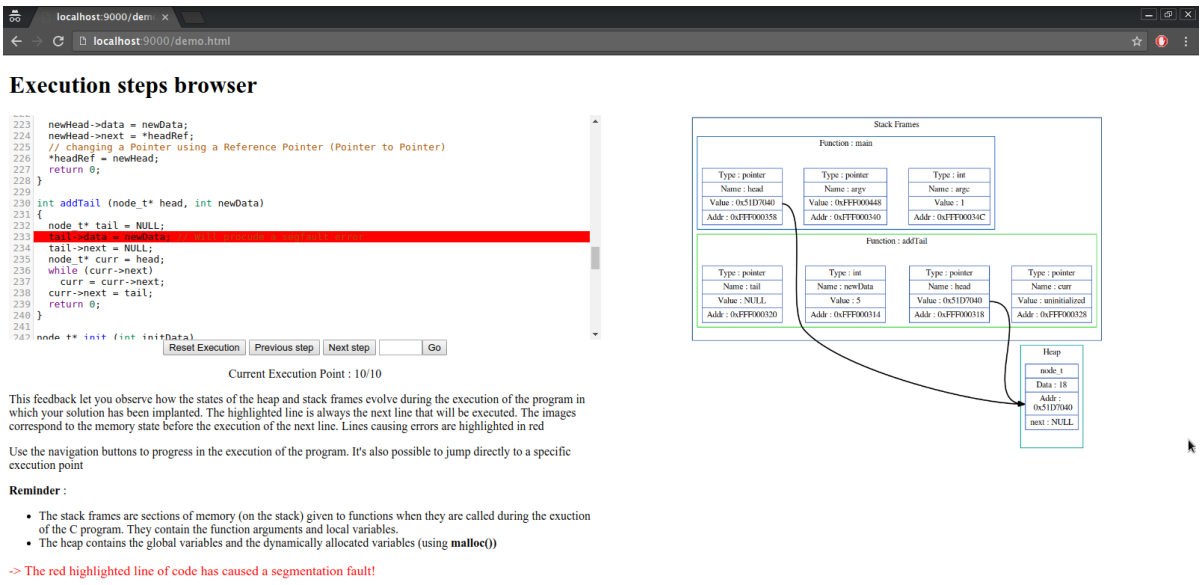


Figure 4.7: Segmentation fault in feedback

The example of figure 4.7 is a feedback simulation with an exercise submission. The user had to implement a `addTail` function to add a new node at the end of a linked list. In its implementation, the user tried to initialize the data field of the new node by using a pointer to it. The error occurs because the value of the pointer is set to `NULL`. We can clearly observe in the corresponding graph that the value of the `tail` pointer is `NULL` instead of the address of a `node_t` structure on the heap. Obviously, no arrow is present between that pointer and a variable in the heap section of the image. The user can therefore deduces that he forgot to initialize the pointer by allocating the memory for the new node.

As additional note : the tool does not handle compilation errors detection. The analyzed codes have to compile without errors to be able to give a feedback on their execution. On the INGInious platform it is already the case. Exercises submissions have to compile without errors to be tested.

4.4 Conclusion

To conclude this part, we have seen that we are able to provide a efficient way to visualize the evolution of the memory used in C programs. It is a great way to help users understand what they are doing when they manipulate the memory of the computer with pointers algorithms. The stack frames and the heap are well represented. It is also possible to directly know which parts of the programs are causing runtime errors and to observe the state of the memory when such errors happen. The interface of the feedback pages is simple and easy to use. Focus has been given to the quality of the feedback pages. Nevertheless, the look of the pages could be improved if necessary but it is not really important. It the next chapters, we will study the implementation of the tool and its performances.

Chapter 5

Implementation

5.1 Introduction

In this chapter, I detail the implementation of the solution, explaining the written programs, the technologies used and how it can be modified to improve it. This is the most technical part of the thesis. I begin by briefly recall the architecture of the application and the problems to solve. Then, I discuss the implementation details of the different parts of the tool.

5.2 Architecture

As a reminder, those are the different subproblems to be solved in order to build the application.

Getting the execution logs of the C programs binaries This is achieved by using the modified Valgrind version developed by the previously mentioned Dr. Guo. Originally developed to extend the `Online Python Tutor` [11] application by adding C programs analysis to it.

Transforming the logs The Valgrind logs have to be sanitized to remove unnecessary information, and to format the needed ones so that the tool can easily parse and use them. Some analysis has to be done on these data to detect events like functions calls or runtime errors.

Building the graphs The graphs are used within the tool to present the memory state of the computer to the students using the extended feedback, during their programs executions. When graphs are grouped together, they constitute a little movie describing the stack and heap mutations caused by the execution of the program.

Feedback presentation The previous parts need to be interpreted and presented in a nice format. This is what the students will see on the final feedback. Thus, it has to be understandable and easy to use.

INGInious integration Even if the tool can be used outside the `INGInious` platform, the main goal of the tool is to be used as an improved feedback on the C exercises submitted on it. Therefore, an efficient way to link the two tools together has to be found. Practically, a little install guide will be provided.

Each part of the system is independent and can be modified without having to modify the whole architecture of the tool. So that we can easily extend it to use it for other programming languages for example. Figure 5.1 summarizes the architecture of the tool.

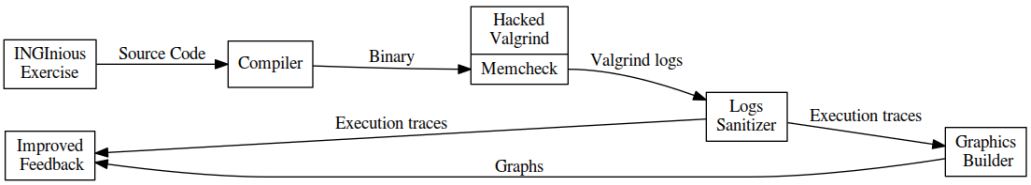


Figure 5.1: Architecture

5.2.1 Data encoding

To pass the data concerning the execution steps of the programs executions through the different parts of the system, an efficient data encoding technique is required. `JSON` seems to be the right choice. `JSON` stands for `JavaScript Object Notation`. Even if its format is mainly inspired by JavaScript objects at first, it is language independent. Thus, there are plenty of libraries allowing programmers to deal with `JSON` files with the programming language of their choice. Therefore, this format is a good choice if we are working on projects involving multiple programming languages. It is a light-weight key-value style data exchanging format [7]. The syntax is quite straightforward to understand. It is composed of dictionaries and lists describing objects. To update a `JSON` file, we can add or modify its objects, simply by adding key-value pairs or by retrieving objects values with the associated keys to modify them.
Figure 5.2 provides an example of JSON formatted file in which a simple object is represented with a dictionary containing key-value pairs separated by commas. As shown in the example, keys must be strings and values can be of any data types.

```
{
    "name": "Ooghe Nicolas",
    "tags": ["master", "thesis"]
}
```

Figure 5.2: JSON example

In the extension, the data describing the states of execution of the C programs and the memory are encoded in this format. I show other examples related to the project in the next sections.

5.2.2 Programming languages and technologies

Before diving deeper in the implementation of the project, I think it is a good thing to present the chosen programming language used to build the project. The two mainly used languages are obviously the C programming language to implement the exercises proposed to the students, and the Python programming language. The C codes are compiled using `gcc`. During the realization of this thesis, I focus on the Linked List data structures implementation in order to test the tool with exercises related to these specific data structures. Implementing Linked List in C is a good way for students to develop skills for complex pointer algorithms as indicated in a paper written by Nick Parlante from the Stanford University [19].

The Python programming language is used to develop the data processing and graphics building parts of the solution. Python is a high-level, interpreted programming language with dynamic semantics, dynamic typing and dynamic binding. It is principally used for scripting purposes but could also be used for larger development projects. It was originally created by the Dutch developer, Guido Van Rossum. The Python syntax is really simple to understand and is easy to learn. It is also right to mention that Python is widely used in universities and free online Computer Sciences courses. It is a good programming language for beginners [23, 11].

In order to construct the graphics that will represent the state of the memory during the execution of the programs, I use the GraphViz software. GraphViz is a collection of graph drawing tools that could be used in a various different applications [8]. The `dot` language is used to the generate the graphs. It has been chosen because it is easy to understand and it is scriptable. To dynamically generate the images using the data produced by the tool, I use the `PyGraphviz` [2] Python module to write code in order to automatically generate the graphs that I need for the student's feedbacks. This module is distributed with BSD license.

To build the feedback pages presented to `INGInious` users, I use JavaScript and HTML. JavaScript is also a high-level, interpreted programming language and is one of the main technologies used for web development. For now the extended feedbacks are built and hosted on an `Apache` web server inside the `INGI` network as static `HTML` pages and students are given a URL redirecting them to their personal feedback along with the `INGInious` binary feedback. I will go into further details in the next sections.

5.3 Valgrind logs processing implementation

In this section, I describe in details the implementation of the presented solution and how it can be extended to handle cases that are not yet supported. I will also demonstrate how to use the tool and how it is integrated into the `INGInious` platform to cover the improved feedback on the students submissions.

The program used to sanitize and process the execution logs provided by the modified Valgrind version is written in Python as mentioned in the previous section and was originally developed by Dr. Guo for his C/C++ backend of the `Online Python Tutor` application [11]. I have forked his project hosted on the `Github` website [10] and modified it to fit my needs. The code has been totally refactored and documented. Thus, the structure of the code is quite different from the original one. It had to be done to improve its readability and to make it more understandable for possible future contributors who want to continue the development of the tool. It is now divided into multiple functions which all have a specific purpose in the processing of the data. I have followed the `Google` Python docstrings [21] specifications to document the code. This specific part of the tool is located in the `generate_traces.py` file in the sources of the project.

It takes as input the traces produced by running Valgrind on the chosen C programs, encoded in the previously defined `JSON` syntax. The `json` Python module is used to process those files. In the Valgrind trace files, the execution points are separated by a special `=== pg_trace_inst ===` line, so we can process them one at a time. This useful feature is provided by the modifications brought by Dr. Guo to the Valgrind code. The result of the program is a file containing all the information in the previously defined intermediate format. Practically it is encoded in `JSON` syntax as well.

The first function of the script, the `process_record(lines)` function, is used to parse the different execution points of the Valgrind trace files. Each of them is appended to a global list `all_execution_points` after being processed by the second and more interesting function, the `process_json_obj(obj)` function. This first function throws an exception in the case of a parsing error which can be caused by a badly formatted input file. Such a case, is generally due to crashing programs.

I will now explain the working of the `process_json_obj(obj)` function that is used to process the execution points contained in these Valgrind log files. This function is responsible for transforming the execution traces in the intermediate format.

The function takes as argument a `JSON` object that represents an execution point of the execution of the C program. It begins by checking whether the stack entry of the currently processed object is empty or not. This stack entry is a list containing the stack frames to render. It is the stack that contains the information about the states of the active subroutines of our analyzed C program. So, this stack will grow when a function call event arises. If it is empty, it is an error because all the C programs have at least a main function. In that case, an exception is thrown. We then reverse this list to follow the convention of the growing down call stack like in figure 5.3. After that, a `top_stack_entry` variable is defined to contain the last element of the stack. It will be the next stack frame to be processed during the execution.

```
1         assert len(obj['stack']) > 0
2
3         obj['stack'].reverse()
4         top_stack_entry = obj['stack'][-1]
```

Figure 5.3: Reversing the stack

The script creates an execution point object in the intermediate format out of this `JSON` object. The entries composing an execution point in the intermediate format are all dictionaries or lists as well. The script begins by initializing these different entries. The `heap` entry describes the state of the heap which changes when memory is allocated using the C built-in functions `malloc()` or `calloc()`. The other entries are the following : `Stack_Frames`, `globals`, `line`, `func_name`, and `event`. These are quite obvious to understand. There are also entries that are specific to the stack frames contained in the stack list : `func_name`, `is_highlighted`, `frame_id`, and `encoded_locals`. For most of these entries, no specific processing needs to be applied. We simply initialize them with the same values as in the Valgrind logs file, like in figure 5.4.

```
1             ret['line'] = obj['line']
```

Figure 5.4: Some entries do not require processing

A specific case that needs further processing, is the encoding of the global variables of an execution point and the local variables of a given stack frame. The description of those variables will be contained in the `globals` and in the `encoded_locals` dictionaries, respectively. To deal with these cases, the recursive `encode_value(obj, heap)` function is used. I will explain its behavior in details because I think it is the most critical function of the script. For now, when it comes to the point of encoding global variables (resp. local variables), it iterates over the `g_var` and `g_val` (resp. `local_var` and `local_val`) pairs of the `globals` (resp. `locals`) dictionary of the current point of execution of the Valgrind log and encodes the state of the variables in the `enc_globals` (resp. `enc_locals`) dictionary using the following function call in figure 5.5.

```
1             enc_globals = {}
2             for g_var, g_val in obj['globals'].iteritems():
3             enc_globals[g_var] = encode_value(g_val, heap)
```

Figure 5.5: `encode_value()`

To treat the different stack frames that compose the current stack entry of the object, the script iterates over them and defines a new `stack_obj` dictionary for each of them. These dictionaries are appended to the `stack` list which is the `Stack_Frames` entry of the execution point. See figure 5.6.

The different entries corresponding to a specific stack frame are initialized in the same way as for the other entries of the execution point object.

```
1         for e in obj['stack']:
2           stack_obj = {}
3           stack.append(stack_obj)
```

Figure 5.6: Add stack frames on the stack

5.3.1 Encoding the variables states and updating the heap

Now I will describe in details the `encode_value(obj, heap)` function which I have already introduced in the previous section. The arguments of this function are `obj` which corresponds to the value (in the format of the Valgrind logs) of a local or global variable, and `heap` which is the dictionary that represents the heap during the execution of the C program. The local and global variables are encoded in function of their kinds and types, in the final trace format. The different kinds of variables are the following : `base`, `pointer`, `struct`, `typedef`, `array` and `heap_block`. This function also updates the heap entry of the execution point in the case of global variables or dynamically allocated variables with the `malloc()`, `calloc()`, or `realloc()` built-in C functions.

5.3.2 The basic data types

This kind groups the basic data types of the C programming language like : int, char, float, double, short and long. As an example, consider the short C program in figure 5.7 where basic local variables are declared in the body of the main function.

```
1         int main (int argc, char** argv)
2         {
3           int i = 1273823;
4           char c = 'a';
5           float f = 3.14;
6           double d = 5.42342422424;
7           short s = 2;
8           long l = 30132308203290382;
9
10          return 0;
11        }
```

Figure 5.7: Basic data types

At the end of the execution of that small program (when all the variables have been declared and initialized), their encoding can be found in the resulting trace file, in the last execution point, in the `encoded_locals` entry of the stack frame of the main function. Let's take the example of the `i` variable to show how it actually works. In the last execution point of the

Valgrind logs, we can find the variable state in the `locals` dictionary of the stack frame of the main function, in the following format: `"i": "addr":"0xFFF00060C", "kind":"base", "type":"int", "size":4, "val":1273823`

The variable identifier, `i`, is the key of the entry, and the value contains the information related to that variable. We find useful information such as its address, its kind, its type, its size, and its value. During the `encode_value(obj, heap)` function call with the object representing this value, it will match the condition in figure 5.8.

```
1        if obj['kind'] == 'base':
2        return ['C_DATA', obj['addr'], obj['type'], obj['val']]
```

Figure 5.8: Basic data types

The return value here, is a list containing some information about our variable. This value is entered in the `encoded_locals` dictionary of the main stack frame, using the identifier of the variable as key. That is the way the basic variables are encoded in the execution point objects. Figure 5.9 shows this encoding.

```
1        "i": [
2          "C_DATA",
3          "0xFFF00060C",
4          "int",
5          1273823
6        ]
```

Figure 5.9: Basic data types encoding

5.3.3 The pointers

Let us now examine how the pointers are encoded in the final trace format. Consider the following code example where an int pointer is declared using the `malloc()` function:

```
1       int main(int argc, char** argv)
2       {
3         int* pTest = (int*) malloc(sizeof(int));
4         *pTest = 4;
5         free(pTest);
6       }
```

Before the memory pointed by the pointer is freed, one can find it in the execution points of the Valgrind log of this program in the form depicted in figure 5.10.

```
"pTest": {"addr":"0xFFF000618", "kind":"pointer", "size":8, "val":"0x51D8040", "deref_val":
  {"addr":"0x51D8040", "kind":"heap_block", "val": [
    {"addr":"0x51D8040", "kind":"base", "type":"int", "size":4, "val":4}]}}
}
```

Figure 5.10: Pointer representation in raw format

We have the information concerning the pointer itself, but also the information about the data (on the heap) pointed by the pointer. For that, we have an additional entry in the value of our pointer object : the `deref_val` entry, which is not present if the pointer points to some memory that does not correspond to the address of a valid variable nor valid data. This new entry has itself three fields : `addr`, `kind` and `val`. The information concerning the data pointed by the pointer can be found in the `val` entry. This entry is a list because a pointer could point to the address of a char array (i.e a string), or to the address of a structure for example. Thus, in these cases, the first element of the list will correspond to the address of the first char of the string, or the first member of the structure. The next elements of that list will contain the address of the next chars of the string, or members of the structure. Let us see an example in figure 5.11.

```
int main(int argc, char** argv)
{
  char* cName = (char*) malloc(8*sizeof(char));
  *cName = 'N';
  *(cName+1) = 'i';
  *(cName+2) = 'c';
  *(cName+3) = 'o';
  *(cName+4) = 'l';
  *(cName+5) = 'a';
  *(cName+6) = 's';
  *(cName+7) = '\0';
  free(cName);

  return 0;
}
```

Figure 5.11: Example

In that code, I dynamically allocate memory for a character string of size 8 (the size of a char is 1), taking into account the terminating character. Then, I manually set the values of the successive characters of my string by dereferencing the `cName` pointer and accessing the contiguous memory addresses from it. The resulting data in the Valgrind logs of this little program has the same form as in figure 5.12.

```
1       "cName": {"addr":"0xFFF000608", "kind":"pointer", "size":8, "val":"0x51D8040", "deref_val":
2       {"addr":"0x51D8040", "kind":"heap_block", "val": [
3         {"addr":"0x51D8040", "kind":"base", "type":"char", "size":1, "val":"N"},
4         {"addr":"0x51D8041", "kind":"base", "type":"char", "size":1, "val":"i"},
5         {"addr":"0x51D8042", "kind":"base", "type":"char", "size":1, "val":"c"},
6         {"addr":"0x51D8043", "kind":"base", "type":"char", "size":1, "val":"o"},
7         {"addr":"0x51D8044", "kind":"base", "type":"char", "size":1, "val":"l"},
8         {"addr":"0x51D8045", "kind":"base", "type":"char", "size":1, "val":"a"},
9         {"addr":"0x51D8046", "kind":"base", "type":"char", "size":1, "val":"s"},
10        {"addr":"0x51D8047", "kind":"base", "type":"char", "size":1, "val":"\\0"}]}}
```

Figure 5.12: Previous example in Valgrind logs

The information about the characters of the string, are described in the `val` list of the `deref_val` dictionary in the value of the pointer object. Obviously, the first element of the `val` entry has the same address as the one pointed by the `cName` pointer. The next elements are located on the next addresses from that point.

During the execution of the script, if such a pointer object value is processed by the `encode_value(obj, heap)` function, it first checks if the `deref_val` entry is present in the value of the pointer object. If so, it means that the heap has to be updated by including the information about the data pointed by the pointer. To achieve this updating, we recursively call the function with the value of the `deref_val` entry. This value has the kind : `heap_block`. Therefore, it will match the following condition: `elif obj['kind'] == 'heap_block':`.

In the recursive call, we begin the updating of the heap entry by verifying whether the address pointed by our pointer is not already on the heap. Otherwise an exception is thrown. The script creates a list that will contain the information about the data pointed by our pointer and adds it into the heap entry of the execution point object. To get the information about these data in the intermediate format, it again, recursively calls the function for each element of the `val` entry and append the results of these recursive calls to the list. Once all the information are gathered in the list, it is entered into the heap dictionary of the execution point object with the address pointed by the pointer as key. After all the recursive calls return, the information concerning the pointer variable itself are added in the `encoded_locals` entry of the current stack frame. Figures 5.13 and 5.14 show the result of all the previously explained processing. The heap dictionary now contains the information about the pointed data and the pointer variable is in the `encoded_locals` entry of the `main` stack frame.

```
"heap": {
  "0x51D8040": [
    "C_ARRAY",
    "0x51D8040",
    [
      "C_DATA",
      "0x51D8040",
      "int",
      4
    ]
  ]
}
```

Figure 5.13: Resulting heap representation

```
"pTest": [
  "C_DATA",
  "0xFFF000618",
  "pointer",
  "0x51D8040"
]
```

Figure 5.14: Representation of the pointer variable in main stack frame

5.3.4 The structures

For the struct declarations in C, we consider two cases here. The case where `struct` are declared without the use of `typedef`, and the case where they are declared with it (declaration of a type alias).

Without using typedef

Let us begin with a small example. In the code snippet of figure 5.15, I define a new type : `struct student` and I instantiates it as a local variable in the main function of the program. (I also initialize the members of the structure but that's not really relevant here, so I omitted these lines to save place).

```
struct student {
  char* firstname;
  char* lastname;
  int noma;
  float grade;
};
```

Figure 5.15: Student structure

```
1       int main(int argc, char** argv)
2       {
3         struct student john;
4         /* ...members initialization... */
5       }
```

Figure 5.16: Structure instantiation

After getting the Valgrind execution logs for this little C program, we can see how the `john` variable looks like in it. The kind field of the object representing the variable is `struct` and the type field is `student`. The val field contains the information about the variable members of the struct. So the Valgrind trace contains the information we need : we see that the variable is a structure and that we have defined a new `student` type.

```
1       === pg_trace_inst ===
2       {
3       "func_name": "main", "line": 19, "IP": "0x4004DB", "kind": 1,
4       "globals": {},
5       "ordered_globals": [],
6       "stack": [
7       {"func_name":"main", "line": 19, "SP": "0xFFF000618", "FP": "0x4004E0", "locals": {
8         "argc": {"addr":"0xFFF0005EC", "kind":"base", "type":"int", "size":4, "val":1}
9       , "argv": {"addr":"0xFFF0005E0", "kind":"pointer", "size":8, "val":"0xFFF0006F8"}
10      , "john": {"addr":"0xFFF0005F0", "kind":"struct", "type":"student", "val": {
11        "firstname":{"addr":"0xFFF0005F0", "kind":"pointer", "size":8, "val":"0x400564", "deref_val":
12      {"addr":"0x400564", "kind":"heap_block", "val": [
13        {"addr":"0x400564", "kind":"base", "type":"char", "size":1, "val":"J"},
14        {"addr":"0x400565", "kind":"base", "type":"char", "size":1, "val":"o"},
15        {"addr":"0x400566", "kind":"base", "type":"char", "size":1, "val":"h"},
16        {"addr":"0x400567", "kind":"base", "type":"char", "size":1, "val":"n"},
17        {"addr":"0x400568", "kind":"base", "type":"char", "size":1, "val":"\\0"}]}},
18        "lastname":{"addr":"0xFFF0005F8", "kind":"pointer", "size":8, "val":"0x400569", "deref_val":
19      {"addr":"0x400569", "kind":"heap_block", "val": [
20        {"addr":"0x400569", "kind":"base", "type":"char", "size":1, "val":"S"},
21        {"addr":"0x40056A", "kind":"base", "type":"char", "size":1, "val":"m"},
22        {"addr":"0x40056B", "kind":"base", "type":"char", "size":1, "val":"i"},
23        {"addr":"0x40056C", "kind":"base", "type":"char", "size":1, "val":"t"},
24        {"addr":"0x40056D", "kind":"base", "type":"char", "size":1, "val":"h"},
25        {"addr":"0x40056E", "kind":"base", "type":"char", "size":1, "val":"\\0"}]}},
26        "noma":{"addr":"0xFFF000600", "kind":"base", "type":"int", "size":4, "val":16050900},
27        "grade":{"addr":"0xFFF000604", "kind":"base", "type":"float", "size":4, "val":12.000000}}}
28      },
29      "ordered_varnames": ["argc","argv","john"]}]
30      }
```

Figure 5.17: Structure in Valgrind logs

So now, what the script does when encoding the value of a struct variable like described in this execution point in the Valgrind logs, is returning a list with the information regarding it such as its address and its type, and the encoding of the variable members composing it. So

the script recursively calls the `encode_value` function with each value of the members of the instantiated struct (in the val field) and appends the result to the resulting list describing the whole struct variable. All this information will be well structured in the resulting intermediate format.

Using typedef

In the case where the coder defines an alias referring to the newly declared struct type (using the `typedef` keyword), like in the code example of figure 5.18, the Valgrind logs are a bit different. Let us see what happens. (Only the modified lines are shown here)

```
1       typedef struct student {
2         /* ...same as above... */
3       } student_t;
4
5       int main(int argc, char** argv)
6       {
7         student_t john;
8         /* ...initializing the members ... */
9       }
```

Figure 5.18: Alias definition

In the Valgrind logs:

```
1         "john": {"addr":"0xFFF0005F0", "kind":"typedef", "type":"student_t", "val":
2         {"addr":"0xFFF0005F0", "kind":"struct", "type":"student", "val": {
3       "firstname":{"addr":"0xFFF0005F0", "kind":"pointer", "size":8, "val":"0x400564", "deref_val":
4       {"addr":"0x400564", "kind":"heap_block", "val": [
5               ...
6         }
```

Figure 5.19: Structure in Valgrind logs with alias

Here, the kind field is now `typedef` and the type field contains the type alias defined by the programmer (So, `student_t` in the context of the example above). The reader may notice that in the val field, in the above partial Valgrind trace example, we find the exactly same information as those who define the stucture in the previous example when `typedef` is not used. Therefore, in the script, we set the type entry of the val entry to the same value as the first type value, and then we recursively call `encode_value` function with the val entry. In that way, in the final trace, `student_t`, the alias, will be used in the encoding of the structure variable. It is a choice that has been made here. The first type entry (`student`) should have been used as well. The code in figure 5.20 shows practically what has been explained in this specific context.

```
elif obj['kind'] == 'typedef':
    # pass on the typedef type name into obj['val'], then recurse
    obj['val']['type'] = obj['type']
    return encode_value(obj['val'], heap)
```

Figure 5.20: Structures representation encoding

5.3.5 The arrays

When processing an array, the script simply returns a list containing the address of the array and the encoding of the the elements of this array. To do that, again, recursive calls of the function are used with each of the array elements (as they are represented in the Valgrind trace) and their results are appended to the resulting list depicting the array in the final trace format. As an example, consider the char string containing only three characters in figure 5.21, its representation in the Valgrind logs in figure 5.22 and the result in the intermediate format, in figure 5.23.

```
char char_str[] = "Bob";
```

Figure 5.21: Char string

```
"char_str": {"addr":"0xFFF000610", "kind":"array", "size":4, "val": [
  {"addr":"0xFFF000610", "kind":"base", "type":"char", "size":1, "val":"B"},
  {"addr":"0xFFF000611", "kind":"base", "type":"char", "size":1, "val":"o"},
  {"addr":"0xFFF000612", "kind":"base", "type":"char", "size":1, "val":"b"},
  {"addr":"0xFFF000613", "kind":"base", "type":"char", "size":1, "val":"\\0"}
]}
```

Figure 5.22: Arrays in Vagrind logs

```
"char_str": [
            "C_ARRAY",
            "0xFFF000610",
            [
              "C_DATA",
              "0xFFF000610",
              "char",
              "B"
            ],
            [
              "C_DATA",
              "0xFFF000611",
              "char",
              "o"
            ],
            [
              "C_DATA",
              "0xFFF000612",
              "char",
              "b"
            ],
            [
              "C_DATA",
              "0xFFF000613",
              "char",
              "\\0"
            ]
          ]
```

Figure 5.23: Arrays in intermediate format

5.3.6 Postprocessing of the resulting trace

The functions that will be described in this section are the ones used to perform the post-processing on the resulting traces in the intermediate format. Indeed, we already have all the information needed about the variables, the stack-frames, the state of the heap, etc. But the correct event entries of the execution states have to be set and there is also the possibility that some duplicated execution points and bogus ones are still present at this point.

Execution points filtering

There is still a probability of bogus execution point presence after all the data processing. Such bogus execution points could contain frame pointers with `0x0` as address. These rare case arises in the event of errors during the Valgrind instrumentation. Execution points with duplicated frame ids have to be filtered out as well. This means that a frame id references two different stack frames on the stack. The `filterExecPoints()` function tracks bogus execution points in the `all_execution_points` variable containing all the execution points in the final format.

The events

The `setEvents` function is responsible for putting the right events in each execution step representation. This function takes a list of execution point as parameter and performs these modifications. This function was not initially present in the first version of the script even if the original code performed those operations already. I have optimized this part of the code to reduce its complexity and to make a function out of it. This function has an additional useful boolean argument that let the function know if the Valgrind trace parsing went well (during the execution of the `process_record` function). If the value of this boolean is set to `True`, it means that the program has not crashed during its execution. Thus, we can update the `event` entry of the last point of the execution with `return` as value because it corresponds to the termination of the program (return statement of the main function). Otherwise, it means that an error has been encountered and that the last entry of the execution trace is the point where the program has crashed. The entry, is thus set to `exception` and a special `exception_msg` entry is added to the last execution point record.

Removing the redundant records

It is still possible to find duplicated execution points at this point. It concerns the execution points that have their `event` entry set to `step-line`, the same `line` entry, and the same set of frame ids in their `Stack_Frames` entry. The `removeRedundantLines` removes these duplicated records to only keep one of them in all cases. In that way, we get only one execution point description per line from the source code, like a debugger would do. This function is quite straightforward to understand.

Runtime errors detection

The first version of the script was not able to detect segmentation faults. I needed to find a way to mark execution points where a segmentation fault arise. I have figured out a simple way to detect those errors so that the tool can mark the line of code causing the segmentation fault, in the student's feedback.
Assuming that C programs executions should always terminate with the return statement of the main function, the previously described `setEvents()` function can be used to guess if the termination of the program was caused by a segmentation fault. As already explained, if no error is detected during the parsing of the Valgrind logs file, the last execution point is marked with a `return` event that means that it is the last execution point of the program. A segmentation fault does not make Valgrind crash. So the last execution point, which is causing a segmentation fault, is marked with a `return` event saying that the execution has terminated here. The main difference is that the currently executed function will most of the time not be the main function at that point. Obviously, it is always possible to write code that will lead to a segmentation fault inside the main function. But in the context of the extended `INGInious` feedback, in the proposed

exercises, students have to implement functions, that will be included in another program and called inside the main function of that one to test the student's solution. Therefore it is possible to detect such errors by checking if the `func_name` entry of the last execution point does not correspond to the main function of the program where the execution should have terminated. If this condition is met, we can change the `event` entry from `return` to `runtime_error`.

5.4 Graphs construction implementation

I will now talk about the implementation of the part of the application responsible for the graphs construction. As already mentioned, I use the GraphViz software and the PyGraphviz Python module to dynamically generate the images representing the memory.

The code performing the graphs generation is divided into two parts. The main source file is the `generate_graph_from_traces.py` that takes an execution trace file as input. It is only composed by a `get_exec_point_info()` function which retrieves the needed entries in the execution point objects from the trace file and a `output_graph()` function to create the images and source files. The main function loads the data and does some pre-processing before calling the second Python script that contains the code responsible for generating the graphs. This second Python script depends on the type of exercises we want to create.

Figure 5.24: Graphs construction

Currently the tool has been tested with exercises dealing with linked list data structures based on an implementation proposed in Nick Parlante's paper about linked list problems [19]. I have written a Python script to handle C codes implementing this data structure and several functions that perform operations on them. My linked list implementation is in the `thesis_LinkedList.c` file in the sources of the application.

The PyGraphviz module comes with a class defining methods to build the graphs : the `AGraph` class. All the methods and types used in the code constructing the graphs, come from this specific class [3]. When the program is executed, the resulting images and corresponding source files in `dot` language are generated and there is no additional modifications to perform on the generated `dot` files. Though these files are conserved for debugging purposes. The Python program used to build linked lists exercises based graphs is located in the `llist_graph_utils.py` module. It is imported by the `generate_graph_from_traces.py` program. The structure of the code is split into the following functions:

`build_graph_from()` : the main function of the script.

`make_heap_graph()` : to construct the block of the graph representing the heap state.

`make_stack_frames_graph()` : to build the Stack Frames block

`retrieve_heap_var_info()` : to parse the data needed to build the graphs

The module parses the `heap` and `frames` entries inside the execution point record in the trace file to get the needed data and build graph nodes in order to represent the variables. Then, it places them in the right graph block. The particularity of the `llist_graph_utils` module is that it analyzes the fields of the linked lists nodes to add arrows between the `next` fields and their next element in the list, in the heap block of the resulting graph. So that the users can observe the evolution of the linked list state when operations are applied on them.

Extending the tool to support other data structures specific graphs generation, so that it can handle other kinds of exercises, simply requires to write another module following the same structure as the `llist_graph_utils` one, using the PyGraphviz module. Then, in the `generate_graph_from_traces.py` file, the new module as to be imported and a call to `build_graph_from()` function can be performed as in the code snippet of figure 5.25.

```
1       i = 0
2     for exec_point in trace["trace"]:
3       if (args.verbose):
4         print ("Processing exec point number " + str(i))
5       obj = get_exec_point_info(exec_point)
6       graph_file_name = "exec_point_" + str(i)
7       exec_point_graph = llist.build_graph_from(obj)
8       output_graph(exec_point_graph, graph_file_name)
9       i = i + 1
```

Figure 5.25: From llist_graph_utils.py

5.5 Feedback pages construction

The feedback pages pages are hosted on an Apache web-server. For testing purposes and to use the tool independently of the `INGInious` platform, I wrote a script to handle everything from the programs instrumentation by Valgrind to the final feedback deployment on the server. The needed data are sent on the server via the `SSH` protocol using `rsync`. This solution has been chosen for the development of the tool during the realization of the thesis. It is a temporary solution because the tool needs a private key to be authorized to send the data on the web server. Naturally it is not the most secure solution since the key has to be stored in the project sources. A possible improvement would be to host the entire application on the server where the `INGInious` application is hosted to avoid having to copy the files from different hosts. For now, the application is dependent of the network connection of the department. That would be better if this could be avoided in the future.

The pages are built using pure JavaScript code and HTML. To display the source code on the page, I use `CodeMirror` [12]. It is a text editor designed for web browsers. It has similar features as other text editor. Those features are Syntax highlighting for a wide range of programming languages, auto indentation, etc. It is distributed under the `MIT` free software license. It has a powerful programming API that allows us to do almost whatever we want to do with it. It is also possible to create personal modules in case of special needs. It is the only external library that I use to develop the feedback pages.

The code is divided into lots of functions and is documented to make the code easily modifiable. It performs simple HTTP requests to retrieve the trace files and the images from the web server. The other functions ensure the proper functioning of the application.

5.6 Improved feedback on INGInious : Install Guide

In this section, I explain how to use the tool with the INGInious platform. This little guide assumes that the reader knows the basics of how the INGInious platform works. The exercises in INGInious courses are implemented in the form of task jobs. These tasks are located inside the courses directories of the platform. A task is created by creating a new directory in the course folder for which the exercise has to be created. This directory must contain a `task.yaml` file describing the purpose of the exercise. In this file, the settings of the exercise can be defined. The two important settings to be set in order to use the improved feedback tool are `environment` and `network_grading`. The codes submitted by the students for the exercises are executed inside `Docker` containers to provide a secure Linux environment to test and grade the programs. The `environment` field in the `task.yaml` file tells the task which `Docker` container it has to use in order to run the students submissions for the exercise. These containers are a bit like little operating systems with a filesystem and tools we need to tests the students submissions. For C programs, there is an already available container in the INGInious platform. So we can use it to compile and test C codes. In order to use the developed tool providing the extended feedback, we need to create a new `Docker` container with its dependencies. This is achieved by using a `Dockerfile` that will create our container. This file is shown in the figure 5.26.

```
1 FROM ingi/inginious-c-cpp
2
3 RUN yum -y install python-devel graphviz graphviz-devel
4 RUN pip install pygraphviz
5 RUN yum -y install openssh-clients
```

Figure 5.26: Dockerfile to build the environment

It inherits from the existing `cpp` container and installs `GraphViz` along with its dependencies and the `PyGraphviz` Python module to be able to build the graphs. It is also required to install

the SSH tools to have the possibility to send the data on the web server. This file is available in the sources of the application. The following command 5.27 is used to build the new container from it.

```
1       docker build -t ingi/inginious-c-graph ./
```

Figure 5.27: Dockerfile to build the environment

Now the environment option can be set accordingly with the name of the new container in the task.yaml file. The network_grading option has to be set to true to allow the container to use the network in order to copy the files on the web server 5.28.

```
1       accessible: true
2       author: Nicolas Ooghe
3       context: linked lists implementation
4       environment: graph
```

Figure 5.28: Dockerfile to build the feedback environment

The other pieces of the feedback application can be put directly in the task folder of the exercise. Typically, an improved_feedback directory with all the required programs can be created and put in the task folder. The required softwares are the generate_graph_from_traces.py, generate_traces.py, the graphs modules and the modified Valgrind used to get the execution logs of the C programs. The latter is contained in the valgrind-3.11.0 folder in the source code of the application. A C source file containing the code in which the students code has to be included must be present in the task folder as well. For example, if the exercise aims to deal with linked list functions implementation, a file containing an implementation of linked list and other functions has to be present in the task folder. Then, a specific function can be chosen to make an exercise out of it. The body of the chosen function has to be removed and the student code can be included instead, as in the figure 5.29.

```
1       int push (node_t** headRef, int newData)
2         /* insert a new node at the beginning of the list with 'newData' as data
3            returns 0 if the function succeeds
4            returns -1 on failure */
5       {
6       #include "push.c" // Include student code
7       }
```

Figure 5.29: Include student's code in C template

After that, the exercise directory is ready. The last and most important file to create is a run (bash) script used to specify the commands to be executed in the container in order to produce

the extended feedback on the students submissions. An example of `run` file can be found in the sources of the tool as well. It contains the commands to compile the code, run Valgrind, construct the trace file in the intermediate format, build the images, and construct the student's personal feedback page. So the first commands to put in the `run` script include the student's code in the `C` template and compile the resulting code. This can be done as in figure 5.30.

```
1   # To get student input :
2   getinput push > push.c
3
4   # Compile the code with the included student's solution
5   compiler_output=$(gcc -std=c99 -ggdb -O0 -fno-omit-frame-pointer template.c -o template.o 2>&1)
6   if [ $? -ne 0 ]; then
7     feedback --result failed --feedback "Errors were encountered during compilation::
8     $(echo "Error : $compiler_output")
9     "
10    exit 1
11  else
12    feedback --result success --feedback "Compilation ok"
13  fi
```

Figure 5.30: Get student's submission and compile

Additional unit tests can be performed to check if the submissions fill the exercises requirements, after the compilation. Then the following command must be used in order to get the Valgrind log files 5.31.

```
1   # Call the modified Valgrind to get the execution logs
2   export VALGRIND_LIB="/task/improved_feedback/valgrind-3.11.0/inst/lib/valgrind"
3   valgrind_output=$(improved_feedback/valgrind-3.11.0/inst/bin/valgrind --source-filename=template.c
        --trace-filename=template.vgtrace ./template.o 2>&1)
```

Figure 5.31: Getting the Valgrind log files

Then we can build the execution trace in the intermediate format as in figure 5.32.

```
1   # Build the execution traces out of the Valgrind logs
2   gen_trace_ouput=$(improved_feedback/generate_traces.py template > template.trace || true)
```

Figure 5.32: Building the execution trace file

And the graphs 5.33

```
1   # Build the execution graphs from the previously built exec traces
2   gen_graph_output=$(improved_feedback/generate_graph_from_traces.py template.trace 2>&1)
```

Figure 5.33: Building the memory graphs

Now, the feedback page is ready to be generated and sent on the web server. First, the `run` script has to create a unique directory to contain the data of the student's submission. The trace file containing the code and the description of the program execution are placed into it. The directory containing the graphs is added to the student's folder too. The name of the student's folder is unique. That way, each student doing the exercise has a unique feedback folder on the web server. A random integer in the range [0; 32767] is appended to the directory name to avoid students trying to access other students feedback. An improvement could be made here by finding a way to block access rights for students that are not owners of the personal feedback on the server. Since the web server based architecture is a temporary solution for testing purposes, I assumes this workaround is sufficient for now even though guessing the id of other students directory is a quite difficult problem. Once the student's directory is ready, it can be sent to the web server. This code can be found in the `run` file example in the source code of the application. If everything succeed with no errors, an INGInious feedback message can be returned to the students with the address of his personnal feedback like in the figure 5.34.

```
1        # Getting the feedback ready...
2
3        feedback --result success --feedback "Improved feedback at URL : http://130.104.78.197/$dir/
             demo_$id.html !"
```

Figure 5.34: Returning the address of the student's personnal feedback

5.7 Conclusion

In this chapter, the implementation of the application has been presented. The different technologies used to realize the tool have been introduced as well. The code has been written taking into account that it could be modified in the future. So it is easy to modify some part of it without the need to rethink its architecture. Documentation is provided in the source code. After reading this chapter, the reader should have a good overview of how everything works. For INGInious users wishing to test the tool, a brief install guide has been provided in order to explain how it has to be used inside INGInious tasks. For users who want to try it on other C codes, without INGInious as intermediate interface, the commands inside the `run` file described in the install guide can be used in a simple `UNIX` terminal. The complete source code of the application can be found on `Github` at the following address : `https://github.com/zorga/INGInious-C-Tutor`. The next chapter will discuss the performances of the tool with the implementation presented in this chapter.

Chapter 6

Performances

6.1 Introduction

This chapter is dedicated to the performance analysis of the tool. The analysis treats each part of the application independently at first, and then the entire application from the compilation of the C codes to the graphs generation.
The feedback construction, which depends on the network connection since the pages are hosted on a web server in the current version of the tool, is not analyzed in the tests. This part of the software serves as "linker" to put the generated traces in the intermediate format, the images, and the pages that contains the graphs and the execution steps browser together. Once everything is sent on the web server, there is no more computation to perform in order to construct the improved feedback. Therefore, I assume that this part can be skipped for the performances tests. The performances tests are realized using C codes dealing with a linked list data structure implementation. I have chosen to make the tests with linked lists because the implementation will mostly be used do make exercises on data structures manipulation. The studied variable is the amount of CPU time used by the different parts of the application. To perform the tests, I gradually increase the amount of dynamically allocated memory using `malloc()`. The tests have been realized on a 2,2 GHz Intel Core i7 processor with four cores and eight threads on an Archlinux environment and the `4.5.4-1` version of the Linux kernel. I have created different scripts to perform the tests described in this chapter. The code of these scripts will be located in the sources of the application on `Github`.
The presented running times are in milliseconds. The amount of CPU time is computed by adding the user time and the system time used during the process. User time being the total amount of CPU time used within the process and the system time being the total amount of CPU time used for system calls during the process. That way we are able to know how much CPU time is used by the tested parts of the application and by the entire application.

6.2 Experimentation

For the tests, I have used C programs that define linked list instances with an increasing number of nodes. The node structure used is defined as in the figure 6.1.

```
1  typedef struct node node_t;
2
3  struct node {
4    int data;
5    node_t* next;
6  };
```

Figure 6.1: `node_t` structure definition

It is only composed by an integer variable and a pointer to another node structure.
Different versions of the code manipulating the linked list have been realized. In each of these versions, I simply increase the number of nodes pushed into the list. Thus, the tool must deal with programs using more and more memory. As a result, the size of the Valgrind log files becomes gradually bigger, the tool generates bigger trace files in the intermediate format used in the application, and more graphs have to be constructed. Forty instances of the same program are used. The first one creates a linked list with only one node. The second one creates a linked list with two nodes, and so on. The last version of the program creates a linked list containing a total amount of forty nodes. At the scale of the application, it is not negligible because the size of the generated graphs keeps increasing. Also, each time a node is added into the list, a function call is performed to allocate the required memory for the new node and to modify the pointers. After creating the linked list, each program uses a special function to free all the memory used by it.
The figure 6.2 shows an example of program that has been used for the experiment.

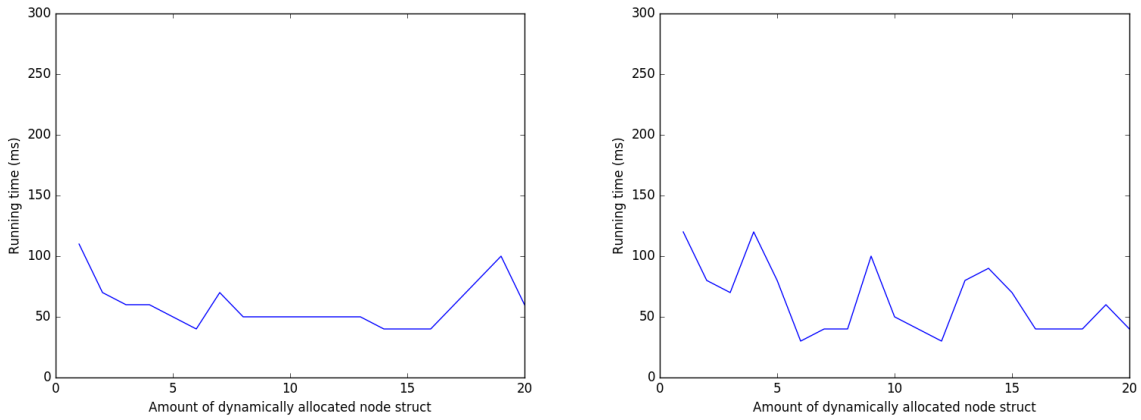
```
1  int main (int argc, char** argv)
2  {
3    node_t* head = (node_t*) malloc (sizeof (node_t));
4    head->data = 18;
5    head->next = NULL;
6    push (&head, 10);
7    push (&head, 10);
8    push (&head, 10);
9    push (&head, 10);
10   push (&head, 10);
11   push (&head, 10);
12   push (&head, 10);
13   push (&head, 10);
14   push (&head, 10);
15   free_list (&head);
16   return 0;
17 }
```

Figure 6.2: Example of used program

6.2.1 Compilation

The first action performed by the application when it is run on a C program, is the code compilation. For the compilation part, I use the version `6.1.1` of `gcc` without optimization in order to retrieve the best from Valgrind and Memcheck. This has the effect of reducing the execution time of the programs. The `-fno-omit-frame-pointer` flag is also used to ease the traces computation.

As it is shown on figure 6.3, the compilation is not a critical part in the performance of the application. The compilation does not depend on the amount of memory used by the programs. On average, regardless of the amount of memory used, it rarely exceeds 100ms. The two charts result from two different tests with the same parameters.

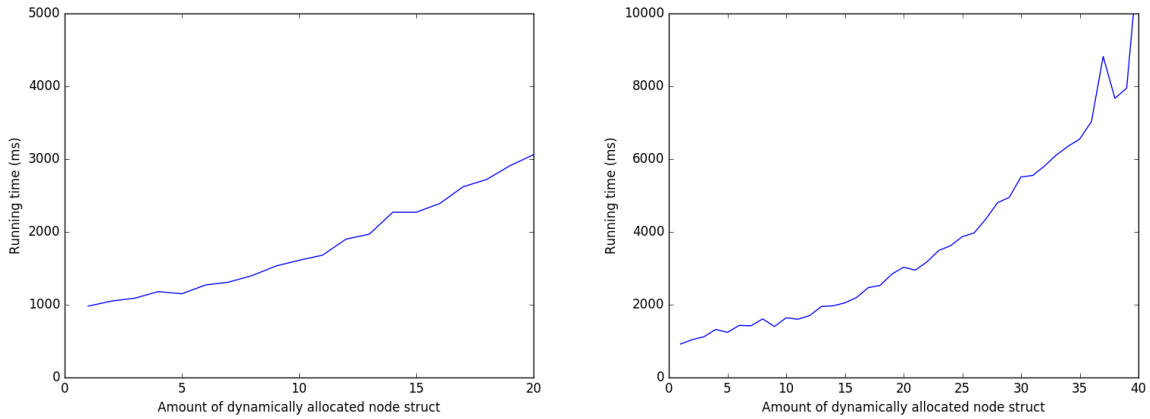


(a) (b)

Figure 6.3: Running time of programs compilation

6.2.2 Computation of the Valgrind logs

The next part of the application, after the successful compilation of the program, is the one that runs the modified Valgrind with Memcheck to get the execution logs of the program in raw format. This part is more sensitive to the quantity of memory used. As it is shown on figure 6.4. The left figure has been realized with linked lists containing from one node to 20 nodes. The right figure has been realized with linked lists containing from one node to 40 nodes. The computation time grows really fast when the amount of nodes in the linked list becomes higher.



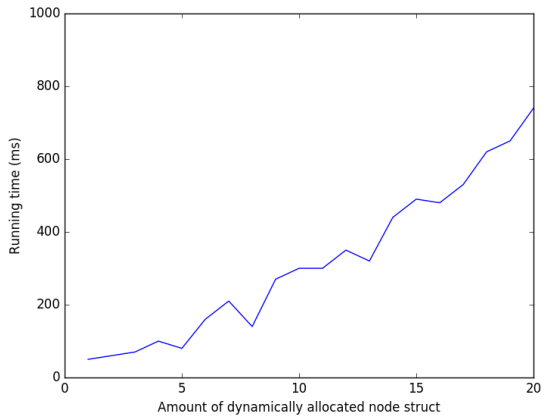
(a) Linked List with 1 node to 20 nodes

(b) Linked List with 1 node to 40 nodes

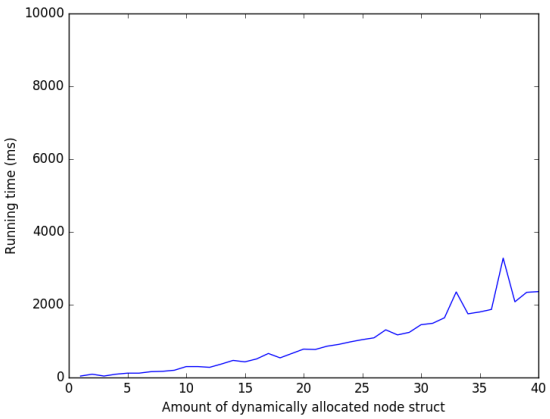
Figure 6.4: Running time of Valgrind logs computation

6.2.3 Transformation of the Valgrind logs

The transformation of the Valgrind log files into the execution trace files in the intermediate format is not a critical part of the application. As we can see on the figure 6.5 below, the computation time does not grow fast when increasing the amount of memory used by the programs. The complexity of this part seems to be linear though.



(a) Linked List with 1 node to 20 nodes



(b) Linked List with 1 node to 40 nodes

Figure 6.5: Running time of Valgrind logs Transformation

6.2.4 Construction of the graphs

The part responsible for the graphs construction is the slowest part of the application. In the figure 6.6, we can see that the running time of this part is already high with a small number of nodes in the linked list. Nonetheless, the complexity does not look really bad either. Moreover, generating graphs with lots of heap blocks has the effect of making the graphs difficult to read on the feedback pages. We have to be careful while checking the student's submissions to avoid such cases.

I have tried to make the code as most efficient as possible. This high running time could be due to the implementation of the PyGraphViz module. Note that for a given program, the graphs are generated for one execution step at a time. At each execution step, GraphViz is used to output the image files. This operation also takes time. In the future, this part could maybe be optimized.

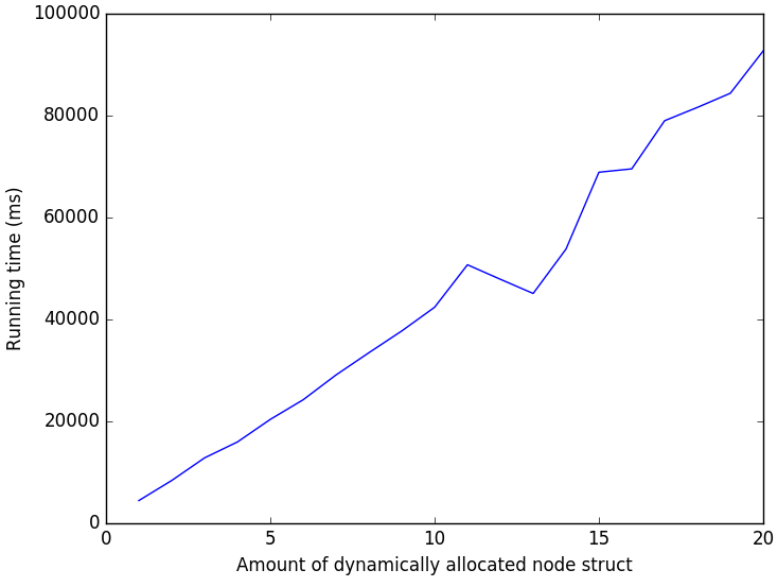


Figure 6.6: Running time of graphs construction
With a Linked List containing 20 nodes

6.2.5 Running time of the entire application

For this section, the tests have been realized with the same programs as in the other tests. But this time, the computed running time is the running time of the entire application, from compilation to graphs construction. The tests have been made with programs manipulating linked list with up to 20 nodes. Below this amount, the required time to perform the test becomes really high. Obviously, the graph construction part is the bottleneck of the application. The shape of the curve in figure 6.7 is similar to the one of figure 6.6.

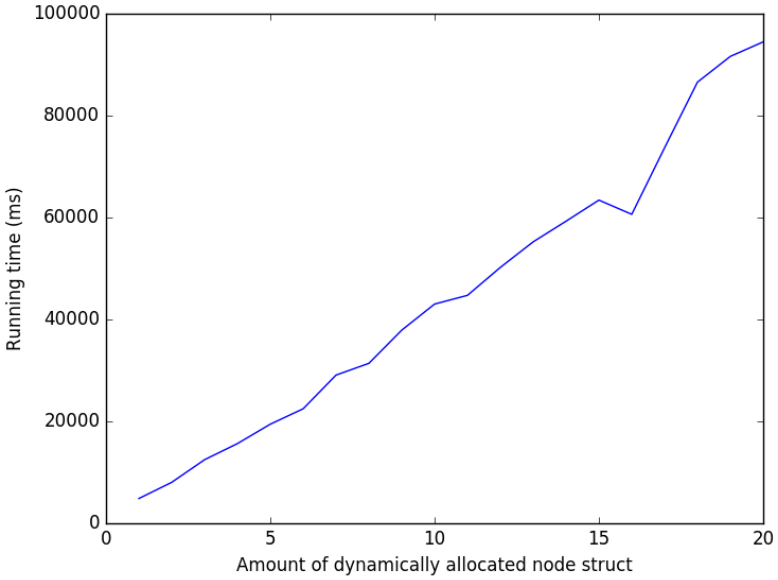


Figure 6.7: Running time of the application

6.3 Conclusion

The performances of the application are globally good. The most critical part of the application is the graphs construction module. This has the effect of considerably slow the entire application down. As already mentioned, this could be due to the GraphViz and PyGraphviz utilization. This external tool is called for each execution step of the programs in order to output the graphs representing the memory during its execution. The code has been carefully written to try to make it as fast as possible. But it is used to automatically write the graphs description in the `dot` language. Indeed, GraphViz is responsible for the graphs rendering part, not the code of the application. Thus, it would be difficult to speed this job up.
The performances do not depend on the quality of the analyzed codes either. Nevertheless, care should be taken when lots of memory are dynamically allocated, even if the memory is correctly freed during the execution.

Chapter 7

Conclusion

To sum up this thesis, the main point is that the goal to develop a tool providing an improved feedback on the C codes submitted on the INGInious platform has been reached. I hope this tool will help future Computer Sciences students to improve their comprehension of the memory management in C.
The main problem addressed by this thesis was the difficulties encountered by students who begin to learn C programming. Beginners often have problems to understand the inner working of pointer variables and memory management. The application gives users the opportunity to visualize the manipulation of the memory in their programs. It also enables to better understand why C programs often encounter segmentation faults without the help of a debugger. That constitutes the major improvement brought to INGInious. The feedback given to students is still "failure" or "success" but now, students have the possibility to explore the execution of their programs to see what went well or what went wrong. Understanding memory management in C is often a long processus for most students. This should help users to faster understand how to face the problems linked to the memory management in C. The sooner these concepts have been understood, the sooner other interesting topics can be studied.

7.1 Future Work

For the time being, the application could still be improved. For this reason, but not only, the source code has been made open-source. In that way, external contributions are possible.
The performance analysis have shown that the application runs correctly but the graphs construction is quite slow. Nervertheless, the integration of the tool into the INGInious platform should not be problematic, even if for now it makes use of an external server to host the dynamic feedback pages. This could be easily improved in the future.
The application has some limitations. First, it is focused on memory management, but there are other use cases that could be explored as well. It does not deal with signals handling, system calls, and multi-threaded programs. Maybe using other Valgrind tools like Helgrind could be a way to develop an extension, allowing it to handle threads visualization.

However, it is difficult to come up with a way to visualize signals handling or system calls. Memory is something easy to represent. Representing the different parts of the memory and the evolution of the entities that compose them, during programs executions was interesting to do. During the realization of this thesis, the power of the Valgrind framework and its famous Memcheck tool has been demonstrated. We have seen how we can use it to retrieve information on the memory used by the programs. This information forms the basis of the application. It is a tool that every C programmer should have in its toolbox.

In the future, the application could also be extended to support other programming languages. As the reader may know, the INGInious platform is not only used to submit C codes. A lot of other Computer Sciences courses are hosted on this platform for which different programming languages are used. It should be a good improvement if the feedback application could be used in all Computer Sciences courses. Obviously, it is difficult to think of another programming language than C when speaking about memory management. But it could still be interesting to observe it with other languages. The code has been written in a way that it is possible to modify the different parts that compose it without needed to review the whole architecture of the application.

Bibliography

[1] Helgrind: a thread error detector. http://valgrind.org/docs/manual/hg-manual.html.

[2] Dan Schult Aric Hagberg and Manos Renieris. Pygraphviz. https://pygraphviz.github.io/.

[3] Dan Schult Aric Hagberg and Manos Renieris. Pygraphviz agraph class. https://pygraphviz.github.io/documentation/pygraphviz-1.3rc1/reference/agraph.html.

[4] Lifelong Kindergarten Group at the MIT Media Lab. Scratch. `https://scratch.mit.edu`.

[5] Michael D Bond, Nicholas Nethercote, Stephen W Kent, Samuel Z Guyer, and Kathryn S McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. *ACM SIGPLAN Notices*, 42(10):405–422, 2007.

[6] Google Developers. Google codelabs. `https://codelabs.developers.google.com/`.

[7] Lidong CAO Dunlu Peng and Wenjie XU. Using JSON for Data Exchanching in Web Service Applications. December 2011.

[8] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2001.

[9] Bharadwaj Yadavalli Ramesh Peri† Gang-Ryung Uh, Robert Cohn and Ravi Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. 2006.

[10] Philip J Guo. C/C++ backend for Online Python Tutor. https://github.com/pgbovine/opt-cpp-backend.

[11] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[12] Marijn Haverbeke. Codemirror. https://codemirror.net/.

[13] Tomislav Janjusic, Christos Kartsaklis, and Wang Dali. Scalability analysis of gleipnir: A memory tracing and profiling tool, on titan. *Cray User Group*, 2014.

[14] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[15] Nicholas Nethercote and Julian Seward. Valgrind, A Program Supervision Framework. 2003.

[16] Nicholas Nethercote and Julian Seward. Using Valgrind to detect undefined value errors with bit-precision. April 2005.

[17] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. June 2007.

[18] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. June 2007.

[19] Nick Parlante. Linked List Problems. *URL : http://cslibrary.stanford.edu/105/LinkedListProblems.pdf*, 2002.

[20] Pluralsight. Codeschool : Learn by Doing. `https://www.codeschool.com/`.

[21] Rob Ruana. Example google style python docstrings. `http://sphinxcontrib-napoleon.readthedocs.org/en/latest/example_google.html`.

[22] Zach Sims and Ryan Bubinski. Codecademy. `https://www.codecademy.com/`.

[23] Guido van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**