Extending an In-Browser C Interpreter With an Abstracted Model of the Memory

Ardalan Samimi Sadeh
Abstract

Extending an In-Browser C Interpreter With an Abstracted Model of the Memory

Ardalan Samimi Sadeh

Traditionally, computer science concepts have been taught using blackboards. In recent years, however, online learning platforms have become an alternative for educators as a way of illustrating complicated concepts. This thesis describes the extension of one such platform, Codecast, with the purpose to provide educators at Uppsala University with a tool that can teach students an abstracted model of the memory of a C program. An evaluation performed on second year students indicates that the tool can be beneficial when trying to illustrate memory related concepts. They showed a better understanding of memory pointers than their peers, and were also more inclined to focus on low level details of the memory. However, as the evaluation was small and limited in scope, a more in-depth study is required to determine whether tools such as these can help the students' understanding of this subject.
# Contents

1 Introduction ............................................. 1  
  1.1 Background ........................................... 1  
    1.1.1 The Memory Model in C .......................... 2  
    1.1.2 Modelling the Heap Memory ...................... 3  
    1.1.3 Codecast .......................................... 5  
  1.2 Purpose ................................................ 5  

2 Related Work ............................................ 6  

3 Description of Codecast ................................. 7  
  3.1 Technical Details ..................................... 7  
    3.1.1 React and Redux .................................. 7  
  3.2 Design Overview ...................................... 8  
    3.2.1 The User Interface of Codecast .................. 8  
    3.2.2 The Components That Make up Codecast ........ 10  
  3.3 Execution of a Program ............................... 12  
  3.4 Memory Allocation in Codecast ....................... 13  
    3.4.1 Deviations From the C Standard ................. 14  

4 Requirements ............................................ 15  

5 Implementation ......................................... 17  
  5.1 Visual Representation ................................ 18  
  5.2 Data Representation .................................. 20  
    5.2.1 The Stack Object ................................ 20  
    5.2.2 The Heap Object ................................ 21  
    5.2.3 The Data Object ................................ 22  
  5.3 Building the Stack, Heap and Data Objects .......... 22  
  5.4 Changes to Persistent-C ............................. 23  

6 Evaluation ............................................... 24  
  6.1 Method ................................................. 24  
    6.1.1 Procedure ....................................... 24  
  6.2 Results ................................................ 25  

7 Conclusions ............................................. 27  

8 Future Work ............................................. 28  

Appendices ................................................. 29
A Evaluation Questions 30
B Improved Evaluation Questions 33
C Usage Guide 36
1. Introduction

The technological advancements made during the last couple of decades have created new opportunities for programming education in the form of online environments that facilitate self-learning. Over the past few years, interest for online learning platforms has increased [1], and research indicates that the format could prove beneficial for programming education [2].

One such tool is Codecast [3]. Designed to teach beginners and novice programmers the C programming language, Codecast combines "a tutorial style video and an Integrated Development Environment" (IDE) [4] allowing teachers to record tutorials that the students can directly interact with.

The tool has sparked the interest of educators at the course Imperative and Object-Oriented Programming Methodology at Uppsala University. Currently, they use blackboards and presentation software to visualize basic data structures and help students understand memory management, but have expressed a wish to switch to a digital platform that would automate the visualization and let students experiment on their own.

This thesis will extend Codecast, by implementing a visualization module that would help novice programmers understand concepts such as memory management and dynamic memory allocation on the heap, as well as pointers and data structures in general. The overall goal is to provide the course Imperative and Object-Oriented Programming Methodology with a complementary teaching aid.

1.1 Background

Taking an introductory course to computer science can be a daunting experience for many students, as it is often their first serious foray into the world of programming, where they need to learn new theoretical and complicated concepts. Apart from the basics of programming and language syntax, these courses typically also include concepts regarding data structures, memory management and program state. However, as these are not easy subjects to teach, many students may have a hard time grasping computer programming [5, 6]. According to one study, 33 percent of all students fail these basic courses [7]. The reasons are many, and can range from the student’s lack of basic logical and mathematical skills to the teacher’s inability to adapt to different learning styles [8]. It is not uncommon for students to struggle
with or harbour misconceptions about core concepts such as variable assignments, recursion, or pointers and references [5, 9].

Traditionally, educators have taught such topics in classrooms by drawing diagrams or using presentation software. Raeder states that humans are visually oriented beings that can obtain information faster through pictures than text [10], and research suggest that illustrations and animations can facilitate learning [11]. However, static and non-interactive material may not always be optimally suited to convey the dynamic nature of programming [12, 8].

This method is currently used by instructors at the course Imperative and Object-Oriented Programming Methodology when teaching students about data structures and the memory layout of a C program.

An alternative approach to illustrating dynamic concepts may be through the aid of visualization software [13], which would automate the process of mapping code to an abstracted model. A review by Sorva, Karavirta and Malmi shows that, during the last three decades, numerous visualization tools have been produced for educational purposes, although most of them are not being actively developed today [9]. Visualization tools have been shown to have a positive impact on the learning outcome for novice programmers [6], as they decrease the cognitive effort put on the users [14, 15].

These tools are commonly, though, either plugins for IDEs or external software on their own. It is also rare for visualization software to present a high-level graphical representation of the heap memory [16]. Even though e-learning platforms have gained in popularity in recent years, there are still few examples of web based, and thus cross-platform compatible and easily accessible environments that provide the user with visual models of a program’s execution. They do, however, exist. One example is the open-sourced Python Tutor [17], which has served as an inspiration to the work described in this thesis. The tool both allows the user to run their code step-by-step and gain insights into a program’s dynamics, but also provides the user with a graphical representation of the evolution of a program’s memory.

1.1.1 The Memory Model in C

The memory model of a C program is typically divided into five segments. The first one is the text (or code) segment, which stores the machine instructions, and the following two are used for initialized and uninitialized global and static data such as string literals [18]. The last two segments of the C memory model are the heap and the stack memory segments [18]. Figure 1.1 shows a typical representation of the memory model, where the stack and the heap segments can dynamically grow.

The stack memory can be considered a temporary storage location, where short-lived data are placed, e.g. local variables, return values or function arguments [19, 20]. As the name suggests, the stack memory functionally resembles its namesake data structure, and each element of the stack is called a stack frame. Whenever a new function is called, a new frame is added to, or pushed onto the stack, and when a function call returns, that frame is removed, or popped from the stack [20]. An important aspect of C is that variables are not initialized by default [20], and may
contain any value that may already be present at that memory location, before being assigned a value by the programmer [20]. For example, when a new frame is pushed to the stack, it may occupy the same memory space as an old stack frame. Thus, variables that have not been given a value may contain remnant data. Values of uninitialized variables are therefore regarded as undefined [20].

The stack memory provides, however, only a limited form of storage, as it is automatically managed by the system and is set on compile time [19]. If the programmer needs to store values meant to outlive the current function, they need to manually allocate memory on the heap [19, 20]. Allocating memory is typically done using the standard library procedure `malloc`. This procedure takes as input the number of bytes needed, and returns a pointer to the beginning of the memory location [19] that now has been reserved by the operating system. When the programmer no longer needs the reserved memory space, they should deallocate it by using the `free` function included in the C standard library [20]. This function takes as input a pointer to the beginning of the allocated memory block and will let the operating system know that the user no longer needs it, freeing it up for use later during the execution of the program [19].

1.1.2 Modelling the Heap Memory

In the von Neumann architecture, which laid the foundation for the modern computer [21], memory is organized into a sequence of cells ordered internally by their addresses [18]. Although there may exist various different ways to model the memory, depending on the level of abstraction needed, the structure described by von Neu-
mann makes tables and diagrams a natural way of illustrating the memory.

As shown in figure 1.2, which models an integer array of size 6, each memory cell is represented by a row in the table, growing downwards from the memory address 0x104. To differentiate between different sequences of cells, that do not belong together semantically, one can draw several, separated tables.

![Figure 1.2: Memory represented as a diagram.](image)

Figure 1.2: Memory represented as a diagram.

When introducing pointers, i.e. entities storing addresses to other memory locations, the memory structure can be drawn as a graph, where the memory blocks are the nodes and the pointers are edges between these, as shown in figure 1.3. This notation, or a variant thereof, is often used in many text books and by educators. Python Tutor uses a similar notation, as well.

![Figure 1.3: Memory represented as a graph.](image)
1.1.3 Codecast

Codecast is an online learning environment, written mainly in Javascript using the frameworks React, Redux and NodeJS. The tool was developed by researchers at Télécom ParisTech in cooperation with the non-profit organization France-ioi, with the purpose of facilitating the teaching and learning of the basics of the C programming language. Consisting of "a C language interpreter, integrated with visualization tools, a code editor and an event and voice recorder for the teacher and a player for the learner," [4] Codecast enables teachers to record simple coding tutorials, while allowing students to interact with the tutorial and modify the code at any time.

Similar to Python Tutor, Codecast is a free and open-sourced tool that also offers visualization, although somewhat limited. The visualization provided is a fairly simple, low-level representation of both the stack and the heap memory.

A more detailed description of the system can be found in section 3.

1.2 Purpose

Although Codecast already provides some form of visualization of the heap memory, it is not deemed adequate to be used for the course Imperative and Object-Oriented Programming Methodology at Uppsala University. As Codecast does not offer an abstracted overview of the memory, it does not fit the needs of the instructors of the course.

The purpose of this project is to extend the tool with a module that provides an abstracted and graphical representation of the memory of a C program to help students understand concepts such as dynamic memory allocation, pointers and basic data structures in accordance with the models used by their teachers. The tool is to be used as a complementary teaching aid.
2. Related Work

Section 1.1 mentioned briefly other, existing tools for program visualization. These include, for instance, Bradman, a visual debugger developed in 1991 that provides a conceptual model of the execution of a C program [6]. The Bradman tool, which is not being actively developed today, shows how the execution of each program statement changes the program state. Bradman, as with most visualization tools, do not, however, give an abstracted, graphical representation of the memory. Commonly, these tools are either plugins for IDEs or standalone programs that needs to be installed on the client computer [22].

Python Tutor [17] is a browser-based application that does provide an abstracted model of the memory. Using Python Tutor, the users can visualize their code and in real-time watch the evolution of the memory as they step through the code. While the tool have several features related to visualization, it does not include the ability to capture video or audio that the students can replay, as Codecast does. Nor does the graphical representation of the memory correspond to the model used by the educators of the course Imperative and Object-Oriented Programming Methodology.
3. Description of Codecast

The following section gives a brief description of the Codecast tool. As it consists of several different elements, only the most relevant parts of the code editor will be discussed here. Details on the recorder, the player and the backend server will be omitted.

3.1 Technical Details

Codecast consists of a server and a client part. The server-side application is written in NodeJS, and on the client-side Codecast uses the frameworks React and Redux (see section 3.1.1 for a description on these frameworks). The compilation of a program is done by an external tool, C-to-JSON, invoked by the server. The output of this process is a tree representation of the syntactic structure of the source code where each node in the tree represents a construct such as an expression or a statement. This is commonly known as an abstract syntax tree (AST).

The execution of a program is done by iterating the AST, using the external library Persistent-C, created by the Codecast team. Persistent-C is a C evaluator written in Javascript, that makes it possible to step through a C program represented as an AST. The library creates and maintains a context for the C program, which includes, amongst other things, the memory of the program at a certain point of time in the execution, as well as which instructions are to be executed next. Codecast does not directly manipulate the context. This is done by invoking different operations provided by the Persistent-C library.

3.1.1 React and Redux

React is a Javascript library used for developing user interfaces (UI) for web applications. First developed by Facebook in 2013, it has since seen a rise in popularity, and is today considered one of the most used frameworks on the web alongside the Vue and Angular libraries [23, 24].

Similar to both Vue and Angular, the building blocks of a React application are called components, and together the components make up the entire application. A component is a self-contained small piece of UI that can be used by and combined with other components [23]. For instance, designing a simple login screen could be
done using at least two different components: One for describing text fields, and one for buttons. A wrapper component could then use these two smaller components to create the login screen.

The components can also contain both a local state and other logic associated with the interface, as well as receive arguments, called properties. Going back to the above example, the state of the text input component could consist of the value of the text field. The button component could, in the same way, define some action to be performed whenever the user clicks on it.

While React does provide a way to keep a global state, many developers today use the standalone library Redux to manage the state of an application. With Redux, the application state is contained in a single object [25]. Updating the state can be done only by dispatching actions, or passing a Javascript object to it, describing an event [25]. The state changes are made by pure functions, called reducers, which takes as input the current state and the action, and outputs the new state [25].

One of the more important features of React and similar frameworks is its reactive UI rendering, meaning a state change does not require the user to reload the web page for the new information to be loaded onto the screen [23].

3.2 Design Overview

Below follows a description of the user interface of Codecast, followed by an overview of the different components that make up the tool.

3.2.1 The User Interface of Codecast

The default view of the code editor, when first loading the web application, consists of a text area where the user will input the source code, a smaller view used to display a running program’s stack frames and a control panel that provides options for compiling or executing a program. At the bottom of the window, an input/output view is placed.

The user can write, compile and execute code directly in the source code editor, which supports syntax highlight. Compiling code in this context means that a server-side parser processes the code and sends back an AST. Having the program represented as an AST allows for the user to either step through the code in a manner similar to a debugger such as The GNU Project Debugger (GDB) or run a program in full, using the control panels.

The editor also includes other views that are only visible through the use of so-called directives. A directive can be considered a plugin for the Codecast editor that provide some additional features. These plugins must be explicitly activated by the user. To activate a directive, the user must type in a command in the form of a comment, directly in the C source code. For example, to activate a directive named Foo, the source code should include the line `//!showFoo()`.
Figure 3.1: The Codecast editor with the memory directive active. The cells in the bottom most area represent the memory of the program. The 4 bytes highlighted there are the memory cells containing the address to the string that the stack variable `output` points to.

One such directive, the memory directive, provides a graphical representation of the heap memory. This view allows the user to investigate the evolution of the memory during execution. However, this view offers a rather low-level representation of the memory. As seen in 3.1, the memory is drawn as a continuous sequence of cells, each representing 1 byte. The 4 bytes that are marked in the figure represents the location of the stack variable `output` in the memory. As `output` is a string pointer, the contents of the memory cells is an address, 0x104, which is the location of the actual string literal. Not shown in the figure are the the memory cells starting from the address 0x104, where the ASCII code associated with each character of the string is stored.

While this is a correct representation of the memory, it is a rather concrete way of drawing the memory compared to the abstracted model described in section 1.1.2. A more abstracted view of the memory, showing the contents of and relations between the blocks, does not yet exist. This increases the workload put on the user to figure out what is actually stored at a certain memory slot, or how different blocks are interconnected.
3.2.2 The Components That Make up Codecast

As the client-side application of Codecast utilizes the React framework, it is built using the component-based design principles of React. This means that the web application consists of a series of components that each provide some part of the user interface along with some functionality. Figure 3.2 gives a concrete view of how the different components together make up the editor.

Figure 3.2: The components of Codecast. The StepperView component consists of the sub components StackView, BufferEditor, DirectivesPane and IOPane (not shown in the figure).

The structure and hierarchy of the components can be seen in figure 3.3. Note that only the most relevant components have been drawn in the figure.

The code editor, labeled as SandboxApp in figure 3.3, consists of the two components StepperControls and StepperView. The former component provides controls for compilation or step-by-step execution, while the latter comprises of the actual source code editor BufferEditor, the stack view StackView and the output view IOPane. As directives are implemented as components as well, these are part of the DirectivesPane component under the StepperView.
In the source code for Codecast, the SandboxApp component is defined in the file `frontend/sandbox/index.js`, which imports the subcomponent StepperView from `frontend/stepper/views/main.js`. Figure 3.4 shows an excerpt of the directory listing of Codecast, with annotations added to files defining the previously discussed components.

```
index.js
  frontend
    _buffers (BufferEditor)
    _index.js
  sandbox (SandboxApp)
    _index.js
  stepper
    IO
      _index.js (IOPane)
    views
      _directives.js (DirectivesPane)
      _main.js (StepperView)
      _memory.js (MemoryView Directive)
      _stack.js (StackView)
```

Figure 3.4: An excerpt of the directory structure of Codecast, showing where the components described in 3.2.2 are defined.
3.3 Execution of a Program

To execute a program in Codecast, the code first needs to be compiled into an AST. Upon compilation, the client-side application will send the source code to the backend server, which parses the code and generates an AST, using the C-to-JSON executable. The result is sent back to the client application.

As an example, consider the simple program in figure 3.5. The only function, named main, consists of two statements. The first one is a variable declaration, assigning the value 42 to the variable \( x \). The second statement is a return statement, that returns the number 0.

```c
int main() {
    int x = 42;
    return 0;
}
```

Figure 3.5: A small program with only two statements. The first statement assigns the value 42 to the variable \( x \), and the second statement terminates the execution of the function and returns the number 0.

Compiling this program in Codecast would result in an abstract syntax tree similar to the one shown in figure 3.6. The root of this tree, `FunctionDecl`, represents the main function. This node consists of a single child node, `CompoundStmt`, which represents the body of the function. As the function only contains two statements, a variable assignment and a return statement, the `CompoundStmt` node has two children: The `DeclStmt`, which together with its child nodes represents the variable assignment on line 2, and the `ReturnStmt`, which represents the return statement on line 3.

On compilation, Codecast also generates an initial state for the program. The state of the program is represented by a single Javascript object, comprised of several substructures. As seen in figure 3.7, these include, amongst other, the current and the previous program context, named `core` and `oldCore`, respectively. The context of a program keeps track of, for instance, the memory of the program at a certain point of time in the execution, as well which instructions are to be executed next. This context is created and maintained by the Persistent-C library.

Once the AST has been built, Codecast can execute the program by traversing and evaluating the nodes, one by one. The user has the option of running the program in full, or by stepping through the program.

The actual evaluation of a statement is done by the Persistent-C library. At each step, Codecast passes a reference to the current context to Persistent-C, which parses and evaluates the current node. The return value of this operation is a collection of so called effects. An effect can be thought of as the result of running the statement. Effects are Javascript arrays that consists of at least the name of the effect. The effect may also include one or more values. Codecast interprets and takes appropriate actions based on the effects.
Figure 3.6: An AST representing the program in figure 3.5. The root of the tree represents the main function. The nodes DeclStmt and ReturnStmt represents the variable assignment and return statement, respectively.

Figure 3.7: The state of a program in Codecast.

For example, executing the assignment operation on line 2 of figure 3.5 would send the context of the program to the Persistent-C library. The library evaluates the current node, which in this case is a VarDecl, and generates an effect describing the assignment operation. In this case, the effect would include the name of the effect, which would be store, the memory address of the variable $x$ (set by Persistent-C), and the value that is to be assigned to it. Upon receiving the store effect, Codecast writes to the memory the value assigned to the variable at the address provided in the effect. The memory write-operation is performed by Persistent-C.

3.4 Memory Allocation in Codecast

Codecast allows for dynamic memory management, by providing the standard library functions malloc and free.

The malloc function will align the memory on a 4-byte boundary. This means
that the memory that is allocated will be padded if the requested number of bytes is not a multiple of four. Furthermore, each memory block is preceded by 4 bytes that are used for book keeping.

The act of writing to and reading from the memory is mainly done by the Persistent-C library.

3.4.1 Deviations From the C Standard

Codecast deviates from the C standard in a major way with regards to the memory model. In contrast to standard C where the stack memory is never cleansed upon popping a frame, the value of all uninitialized and unevaluated stack variables and heap memory allocations in Codecast are set to zero. This is done by the Persistent-C library, and not Codecast itself.
4. Requirements

As this tool is supposed to be used by educators at the course Imperative and Object-Oriented Programming Methodology at Uppsala University, the design requirements were defined in cooperation with associate professor Tobias Wrigstad, henceforth denoted the client.

In his lectures, the client often uses an abstracted representation similar to the model described in section 1.1.2 to teach the inner workings of common data structures, such as linked lists.

One of the main requirements put forward was for the visualization module to adopt a similar notation. In this representation, allocated memory is drawn as rectangular boxes with their stored contents sometimes visible inside of them. Pointers to memory blocks are drawn as arrows, meaning that memory cells that are linked together have arrows between them. To differentiate between different blocks of memory, the boxes are sometimes marked with type information. The notation also indicates whether a memory block has been deallocated.

Figure 4.1: A representation of a linked list using the client’s notation.

Figure 4.1 demonstrates how a data structure can be represented using this notation. It shows three links in a linked list data structure, consisting of two fields: An
integer element, and a pointer to the next link in the sequence.

As seen in the figure, the arrows always point to the left side of a box, which represents the beginning of the memory slot. This is intentional and is an attempt to capture how C pointers operate in reality. The client requested a similar feature.

Another requirement put forward was that the size of a box should reflect the actual number of bytes allocated. For instance, an 8-byte block should be twice as large as a 4-byte block.

The client has also expressed a wish to be able to toggle between different modes of details. The user should be able to switch between different levels of abstraction and conceal or reveal more information such as the data type of a structure.

The module should visualize the stack as well. Similar to the heap memory notation, stack frames should be drawn as tables with each row representing a stack variable, with the name and the data type visible. When assigned a value, the visual model should reflect the change by updating the value of the current variable. As with the heap model, pointers on the stack should be represented with arrows. For pedagogical reasons, as string literals are not stored on the heap, nor the stack, these should be shown in an adjacent area.

Furthermore, the client would like the Codecast environment to more closely follow the C standard in regards to uninitialized variables. By default, the tool zero sets variables and memory cells that has not been assigned a value, which deviates from the C standard.
The implementation of the visualization module, named *Memory Map*, can be split into two different steps. The first one involving the design of the graphical representation of the memory, in as close accordance as possible to the specification described in section 4. The second step was to find an appropriate data representation for the memory, and to gather the correct data from the Codecast environment to be able to construct such a data structure.

As Codecast can be used for more than understanding memory management and data structures, the visualization module was developed as a plugin. This means that in order to use it, the user must explicitly activate it using a directive, `!showMemoryMap()`.

The Memory Map module can be found in the directory `./frontend/stepper/memorymap`. Figure 5.1 shows the directory structure of the module.

```
  .
  |-- frontend
  |   [...]
  |-- stepper
  |   [...]
  |   `-- memorymap
        |-- components
        |   |-- blocks.js
        |   |-- circles.js
        |   |-- data.js
        |   |-- detailedgraph.js
        |   |-- frames.js
        |   |-- graph.js
        |   |-- lines.js
        |   |-- helpers.js
        |   |-- index.js
        |   |-- memorycontent.js
```

Figure 5.1: The Directory Structure of the Memory Map Module.
5.1 Visual Representation

The Memory Map module consists of two different modes: A simplified, highly abstracted overview of the heap memory, and a slightly less abstracted but more detailed view of the heap and stack memory. The user can toggle between the different views with the use of a group of buttons, placed at the top left.

By default, the detailed view is chosen. This view is made up of three different areas, representing the different parts of the memory: The stack, the heap and the data section.

The stack area displays the stack frames and their contents, while the heap area shows the allocations made by the user. The data area lists all string literals defined in the code. Pointers between memory addresses are drawn as arrows. Figure 5.2 shows an example of the visual representation of the memory. The stack frame for the `main` function is visible at the left, with its stack variables `baz`, an integer with the value 42, and `bar`, a pointer to a structure on the heap (seen in the middle). The only field of the structure, `str`, is a string pointer, pointing to the string `Hello World!` in the data area to the right.

![Figure 5.2: A graphical representation of the memory of a program. The stack area shows the stack frame for the function main, which contains two variables. The structure on the heap has one field, `str`, which is a string pointer, pointing to the string `Hello World!` in the data area to the right.](image)

Allocating a single block of size X, using `malloc`, will draw a box of size X multiplied by a factor. As Codecast uses 4-byte aligned memory addresses, if X is not a multiple of 4, the size of the allocated block and the box drawn will be the next closest integer that is a multiple of 4. The allocated block seen in figure 5.2 is 4 bytes large.

As figure 5.3 shows, the allocations grow downwards, instead of horizontally as per the model discussed in section 4. The discrepancy is due to the issue of screen real estate. By having a vertical design, more information can be displayed without the need to scroll too much.

The heap boxes consists of two labels: One smaller in red, and one larger in black. Depending on the data types, these labels differs. For a structure, the small, red label marks the name of a field, while for other data types the label represents the index of a memory cell in the block. Figure 5.3 demonstrates the differences. The first box represents a link in a linked list data structure with two fields, and the second box represents an integer array of size 2, where the value of first item of the array has been set to 42 and the second to 9. Both of these blocks are 8 bytes large.
Figure 5.3: A structure and an integer array allocated on the heap. The red label represents either the names of a structure’s fields, or the indices of a memory block’s cells. The black labels denotes the contents of a memory cell.

The black label denotes the contents of a memory cell. In figure 5.3, the first field of the structure, named `element`, contains the integer 42. As this is a scalar value, it is printed directly in the box. The second field contains a pointer to the memory address 0x104, which in this case is the beginning of the structure itself. The arrow indicates where the pointer points.

The same applies to stack variables in the stack area. The value of the variable is printed after the equals sign, as seen in figure 5.2. If the value is an address, i.e. the variable is a pointer, then an arrow will be drawn from the variable to the target of the pointer.

In the simplified view, shown in 5.4, only heap allocations are drawn. These are represented as simple circles, with the type information displayed above them. Pointers are, as in the detailed view, represented by arrows.

Figure 5.4: The simplified view.
5.2 Data Representation

To build the visual representation of the memory, a new data structure named memoryMap was introduced. The memoryMap object was made a part of the the state object, described in section 3.3, and consists of three child objects, stack, heap and data, each containing information used by the different areas of the Memory Map view. These three data structures will be described in more detailed in subsequent sections, followed by a description on how the data is gathered.

5.2.1 The Stack Object

The properties of the stack object are shown in figure 5.5. The first property, frames, is an array of StackFrame objects. StackFrame, a Javascript class, is defined in the file memorymap/memorycontents.js and represents a stack frame. Whenever a new function is entered, a StackFrame object is created and pushed onto the frames array, and whenever that function has returned, the StackFrame object is removed from the array.

The StackFrame object is used for drawing the stack frames seen in figure 5.2. Beside the function name and the number of variables that resides in its scope, the StackFrame object also contains the properties uninitialized and variables. The former property holds a collection of all variable declarations that has not yet
been evaluated. As these have not yet been evaluated, they do not have a memory address. Variables that have been evaluated, and thus been written to the memory object of the C program by the Persistent-C library, are stored in the variables property, which is a collection of StackVariable objects. Similar to the StackFrame, the StackVariable object is a Javascript class representing an aspect of the visual model.

The actual value of a variable is found in a separate place, the values properties of the stack object. This property holds a collection of either ValueType or PointerType objects. These objects simply describe what value is written at what address in the memory.

5.2.2 The Heap Object

![Figure 5.6: An overview of the heap object.](image)

The heap object has four properties, seen in figure 5.6. The last one, values, is identical in both form and purpose to the values property of the stack object, described in section 5.2.1. The first one, allocatedBlocks, holds a collection of MemoryContent objects, describing allocations made on the heap. MemoryContent, also a Javascript class, represents a single cell or a group of memory cells. The fields property of the MemoryContent object holds information on all memory addresses that semantically belong to the allocated block.

For example, if the user allocates an integer array with two elements on the heap, with starting address X, the fields collection would have two entries. Both would
have the same type and size, but the names of the first and second fields would be 0 and 1, respectively. The addresses would be X and X+4, respectively, as an int data type is 4 bytes large.

For a record type like the one defined in figure 5.7, the names of the fields entries would be the same as the names of the fields of the structure, that is, bar and baz.

The fieldAddresses property of the MemoryContent object holds a key-value store containing the addresses associated with an allocated block. The key of each entry is the address of the field, while the value is the starting address of their block. In the example with the integer array, the addresses would be X and X+4, both mapped to X. This is a temporary data structure used only after the MemoryContent object has been created, and is merged with the cellMapping object of the heap object. The cellMapping object contains all addresses, mapped to the beginning of their respective block.

```c
1 struct foo {
2     int bar;
3     char *baz;
4 }
```

Figure 5.7: An example structure named foo, with the fields bar and baz.

5.2.3 The Data Object

The last member of the memoryMap object is data. This data structure simply holds one substructure, literals, which is a dictionary mapping an address to a string. The data object is used for the visualization of the data section.

5.3 Building the Stack, Heap and Data Objects

The three objects described above are built and updated at different times in the execution of the C program.

As data and a part of stack contains only static information, these objects are built on compilation. This is handled by the function mapStaticMemory, found in the file frontend/stepper/memorymap/helpers.js. By iterating over all the nodes in the AST, the function gathers information on both the string literals defined in the program, and all function and variable declarations. The latter data is stored in the functions property of the stack object.

The heap object and the rest of the stack object are built as the execution proceeds. At each step of the execution, the objects are updated. The function mapMemory is responsible for creating the heap and stack representations, and is defined in frontend/stepper/memorymap/helpers.js.

The stack representation is built by going through a data structure provided by Codecast, which holds information on the different stack frames. By gathering the
necessary information from this data structure, and formatting them in an appropriate way, the mapMemory function generates \texttt{StackFrame} objects used in the visual representation.

The building of the heap representation is a two step process. As there is no easy way of knowing exactly what is stored in the memory, the function first iterates through all allocated memory cells, with the help of a function provided in Codecast. Once this process is done, the function can cross-check against a memory log, provided by Persistent-C, to determine what values are stored where in the memory.

5.4 Changes to Persistent-C

Changes were also made to the Persistent-C library, which by default zero sets variables and memory cells that has not been assigned a value. Instead, a random number or a string is now assigned to uninitialized memory, representing garbage data, in an attempt to mimic the behaviour of a C program.
6. Evaluation

This section will present the evaluation of the Memory Map module. First, a description of the method used and how the evaluation was performed will be given, followed by the actual results of the evaluation.

6.1 Method

To evaluate the Memory Map module, a small study were done with the help of a group of students at Uppsala University. The purpose of the evaluation was to determine whether (1) the students would understand the graphical representation provided by the Memory Map module, and (2) the tool does help the students in their understanding of a program’s memory and runtime.

The students had to individually answer a total of 13 questions (see appendix A) that would test them on their knowledge on the memory model of a C program, memory management and how pointers work. Three of these questions asked the students to draw a graphical representation of the memory of different programs, while the other ten questions were of a quiz-like nature, where they, for instance, had to determine the value of some variable. The number of correct answers will serve as an indication on whether the Memory Map module can be beneficial for students.

A total of nine second year students from Uppsala University participated. As this is a small study with a small time frame, it was determined better to select second year students that have already attended the course IOOPM during the previous semester, and are already familiar with the subject, as it would be too time consuming to teach these concepts to students with little or no knowledge of C.

To be able to determine if the Memory Map module does in fact help the students understand memory and pointers, the students were divided into two groups. One group was allowed to use Codecast, while the other group was only to use pen and paper while answering the questions.

6.1.1 Procedure

The evaluation were done in a computer lab at Uppsala University and took 50 minutes to complete. The students were not told beforehand how the evaluation
would be performed.

Before the evaluation, the students attended a 20 minute long lecture on memory management and pointers in C, held by associate professor Tobias Wrigstad. As these are concepts the students are already familiar with, the lecture served mainly as a refresher. During the lecture, Tobias Wrigstad talked about the memory model of a C program, pointers and pointer arithmetic, as well as arrays.

The students were then randomly split into two groups. Each student was handed two sheets of paper. The students were asked to write down their answers on these papers, as well as their names. They were not allowed to begin before everybody was ready.

The first group, denoted group A, consisted of five students. The other group, denoted group B, consisted of four students. Group B were given access to the computers and the Codecast application. They were told to use the tool to answer the questions. Group A was only allowed to use pen and paper to answer the questions.

6.2 Results

The results will be presented groupwise. The number of correct and incorrect answers of each group participant will be added to an overall score for the group as a whole. Of the 13 questions, three of these asked the students to draw graphical representations of the memory. These three questions will be presented separately from the other ten, as the answers were too diverse and not easily graded.

As group A consisted of five students, the total overall score for this group was 50. For group B, consisting of four students, the total score was 40. The results are shown in the table in figure 6.1.

Group A answered in total 25 out of 50 questions correctly. This group had most trouble with questions 1.3b and 2.2, which none of the participants answered correctly. Question 1.1d was challenging as well, as only one group member could provide a correct answer.

Group B answered 31 out of 40 questions correctly. This score might have been higher, as three of the incorrect answers reported were due to a misunderstanding. A student did not answer question 1.1a, 1.1b and 1.1d as she thought only one of the questions was to be answered. Another student in group B misunderstood question 2.2.

Group B had issues with question 1.1d, to which only one participant gave an acceptable answer.

The reason the students in group A answered 1.3b incorrectly seems to be in part due to a confusion on how program code is evaluated combined with the way the question was formulated. The question asked for the value of a variable during the execution of a given program at a certain line, containing a statement which would update the value of that variable. Viewing this in a machine-like manner, that statement would not yet have been evaluated as the program was executing that line. The answers given, though, indicates that some of the students in group A interpreted this as if the statement had been evaluated.
Group B were more literal in their answers in general. For instance, on question 2.2 they were asked to give the value of a pointer variable after performing an arithmetic operation on it. This can be interpreted as both the address of the pointer, or the value it points to. Group A interpreted the question asking for the actual value, while group B read the question more literally and wrote the actual addresses that the tool displayed. However, group B were more successful in understanding where the pointer actually pointed, and, thus, at what value, than group A.

On questions 1.2, 3.1 and 4.1 the students were asked to draw a representation of the memory of given programs. Students in group A consistently used the typical memory model, as shown in figure 1.1. This is most likely due to Tobias Wrigstad using it during his lecture when talking about the memory organization of a C program.

Furthermore, the drawings were generally not as detailed as group B’s and did not include, for example, the names of fields or type information. However, even though group B had access to the visualization tool, not everyone in the group seem to have taken advantage of this, and instead have been at least inspired by the model used by Tobias Wrigstad. Only two students in group B directly copied the graphical representation on display on their screens.
7. Conclusions

The purpose of this thesis has been to implement a visualization module for the e-learning platform Codecast, to be used in the course Imperative and Object Oriented Programming Methodology at Uppsala University. The Memory Map module was developed according to a set of specifications given by associate professor Tobias Wrigstad.

An evaluation was performed to determine whether the students would understand the graphical representation provided by the Memory Map module, and if the tool can help students get a better understanding of a program’s memory and pointers in C. However, as the sample size of the study was small, the results can only be seen as an indication.

The results presented in section 6 shows that group B had no difficulties answering the questions. This indicates that the students using Codecast could correctly interpret the graphical representation provided by the Memory Map module.

The question of whether the tool does help the students in their understanding of memory and pointers is more difficult to answer. Apart from the small sample size, the differences between the group were far too small. The results for question 2.2 does seem to indicate that the tool does make it easier for students to understand how pointers work, as none of the participants in group A gave a correct answer.

The tool seems to also have put a focus on more low level details, as the students in group B wrote down the actual memory addresses of variables when asked about the values of these. This is not an incorrect interpretation, as the questions were not specific enough when it came to pointers. However, no one in group A interpreted it the same way, as they had not access to Codecast. This would indicate that the tool can help students at least be more aware of such details.
8. Future Work

The focus of this thesis has been on implementing a visualization module that correctly shows the memory of a C program. However, the graphical design has not been prioritized. This is a possible area of further development. For example, text elements that are wider than enclosing boxes are not wrapped. Too many pointers in a program may also lead to cluttered screens, as all the arrows are drawn at once and may overlap each other. This can be improved by having the option to hide some or all arrows, or focusing on specific pointers.

Another point of improvement would be to move some of the logic of the Memory Map module from Codecast to the Persistent-C library. The reason being that the library contains the memory related logic.

A third area of improvement would be for Codecast and Persistent-C to better follow the C standard. For example, currently, uninitialized variables and memory cells are assigned some random garbage data. Instead, the variables and memory cells should be assigned whatever value was previously stored at that memory address.

Furthermore, the current implementation has a major drawback. Pointers pointing to areas of memory not allocated by the user are not drawn. The same applies to pointers pointing to addresses that are not aligned. This is a feature that would greatly benefit users when debugging faulty programs containing pointers.

To be able to show that tools such as these actually help students, a much more in-depth and comprehending study should be performed. To be able to determine whether Codecast does help students in gaining a better understanding of C, the questions asked should demand more from the student than simply stepping through the code and copying whatever is shown on the screen. This means that the questions asked should test the students ability of abstract thinking. The questions should be designed in a way where Codecast, and the Memory Map module, can not give the full answer, only relevant information needed for the students to reach the answer on their own.

Appendix B includes a set of new questions that can be used starting-point for a new study.
Appendices
A. Evaluation Questions

Question 1

For the following program, answer the questions listed below.

```c
#include <stdlib.h>

struct bar {
    char *g;
};

struct foo {
    struct bar *f;
    int *y;
    int *z;
};

int main() {
    //ShowGraph()
    int z;
    struct foo *x = malloc(sizeof(struct foo));
    x->f = malloc(sizeof(struct bar));
    x->y = malloc(sizeof(int));
    x->f->g = "Hello World!";
    *x->y = 42;
    z = *x->y;
    return 0;
}
```

1.1. Where does the following variables point just before the main function returns? They can either point to the data section, the heap or the stack, or they can be NULL or uninitialized.

a. x
b. x→f

c. x→f→g

c. x→f→z

1.2. Assuming that line 21 was removed. Provide a graphical representation of the
the program’s heap memory using the notation used in the lecture.

1.3. In the previous program, what is value of the variable z at...

a. line 15

b. line 23

c. line 25

Question 2

For the following program, answer the questions listed below.

1 int main() {
2 //!showGraph()
3 char *a = "Foo";
4 char b[2] = "42";
5 char c[2] = "84";
6 a = c;
7 a += 2;
8 return 0;
9 }

2.1. Where are b, c and a allocated?

2.2. What is the value of a at line 8?

Question 3

For the following program, answer the questions listed below.

1 #include <stdio.h>
2 void bar(int *y) {
3 char *msg = “hello, world”;
4 puts(msg); // P
5 *y = 84;
6 }
7
8 int foo(int i, int *x) {
9 int z;
10 if (i > 1024)
1. Draw a graphical representation of the above program’s stack memory, just before line 4 is executed (at mark P).

2. What is the value of the variable z after the foo function returns?

Question 4

For the following program, answer the questions listed below.

4.1. Assuming the allocated memory that the pointer variable z points to contains the integer 42, draw a graphical representation of the program’s memory.
B. Improved Evaluation Questions

Question 1

For the following program, answer the questions listed below.

```c
#include <stdlib.h>
#include <stdio.h>

char *copy_string(char *src, int number_of_characters) {
    char *str = malloc(sizeof(char) * number_of_characters);
    while (*src) {
        *str = *src;
        str++;
        src++;
    }
    return str;
}

int main() {
    //!showGraph()
    char *string = copy_string("Hello World", 12);
    printf("%s\n", string);
    return 0;
}
```

1.1. The above program does not work as intended. Identify the problem.

1.2. Propose a solution to fix the problem identified.

Question 2

For the following program, answer the questions listed below.

```c
int main() {
    //!showGraph()
    char *a;
```
2.1 Where does the pointer a point to, after executing line 7? Why?

Question 3

For the following program, answer the questions listed below.

```c
#include <stdio.h>
#include <stdlib.h>

struct foo {
    struct foo *foo;
};

struct foo *new_foo() {
    struct foo *foo = malloc(sizeof(struct foo));
    foo->foo = NULL;
    return foo;
}

void add_foo(struct foo *foo) {
    struct foo *oof = new_foo();
    struct foo *ofo = foo;
    while (ofo) {
        ofo = ofo->foo;
    }
    ofo->foo = oof;
}

int main() {
    //ShowGraph()
    struct foo *foo = new_foo();
    add_foo(foo);
    add_foo(foo);
    add_foo(foo);
    return 0;
}
```
3.1. The above program should link together a series of foo structs. Why does it not work.

3.2. Fix the program to work as intended.

Question 4

For the following program, answer the questions listed below.

```c
#include <stdio.h>
#include <stdlib.h>

int fib(int n, int *d) {
    if (*d != 0) return *d;
    if (n < 2) {
        *d = 1;
    } else {
        *d = fib(n - 1, d) + fib(n - 2, d);
    }
    return 0;
}

int main() {
    //!showGraph()
    int n = 5;
    int *a = malloc(sizeof(int) * n + 1);
    for (int i = 0; i <= n; i++) {
        a[i] = 0;
    }
    a += n;
    printf("The n:th fibonacci number is \%d\n", fib(n, a));
    return 0;
}
```

4.1. The above program does not correctly calculate the fibonacci sequence. Find and correct the problem.
C. Usage Guide

To be able to run Codecast locally, the following tools must be installed on the system:

- Node Package Manager and NodeJS
- Docker
- Git

Once the above tool have been setup, the project must be downloaded from the hosting site Github.com. This can be done using the Git command line tool, with the following command:

```bash
$ git clone https://github.com/pkrll/Codecast
```

This will create a new folder, Codecast, and download the project to that folder. To start Codecast, navigate to the Codecast folder, and from the Terminal, type:

```bash
$ make start
```

Make sure the Docker application is running before running the above command. This will build the project and start the Docker container for the service.

The web application will, after the build and launch processes have finished, be reachable at http://localhost:8001.
Bibliography


