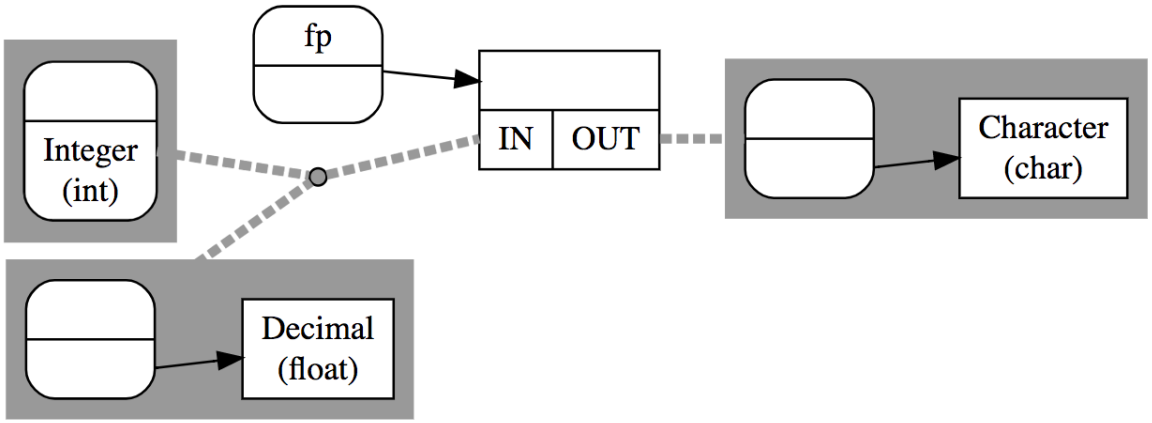
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017



Project Title:	**A Visual C++ analysis program for educational use**
Student:	**Ravi B. Woods**
CID:	**00939309**
Course:	**EIE3**
Project Supervisor:	**Dr Thomas J. W. Clarke**
Second Marker:	**Mrs Esther Perea**

Acknowledgements

Firstly, I must acknowledge the help and support of Dr Clarke the most. He was patient with me every week, even during my stress. I also must thank my testers; Karol, Peter, Zsofi, Tom, Salomon and Ben. Your time means a lot. Finally, my parents; Peter & Anita, thanks for all the support, proofreading and advice you have given me during this process.

Table of Contents

Abstract

When beginners learn C or C++, they often find declarations difficult to understand, especially those containing pointers or arrays, due to complicated syntax. So, in this project, these declarations can instead be visualised as diagrams, to aid novice programmers. The visualisations were developed, through testing, to be as clear as possible, even if the user has never learnt programming before. The result is a standalone cross-platform desktop application in Node.js, with source files parsed through LibClang - an API to the LLVM compiler. The application is currently a simple code editor, that allows the user to view visualisations while programming, as well as load and save source files. However, the application has been developed to, in the future, integrate as a plugin for Integrated Development Environments, such as Visual Studio Code.

CHAPTER 1

INTRODUCTION

In the 21st century, it is becoming increasingly likely that children and young adults will need to program in their lifetime, and many programming classes start by teaching C or C++. However, often when young people come to programming, they are used to interacting with computers through well-designed user interfaces, and so they find it difficult to understand how to interface to a computer without a visual interface.

One example of this occurs when declaring variables in C and C++. While basic declarations are easy for beginners to understand, when they learn about other constructs, such as pointers, arrays and functions, declarations can often become much more difficult to understand. In the past, techniques have been proposed to help programmers more easily understand these declarations, for example the Clockwise Spiral Rule (Anderson, 1994). However, while these rules may help those who already have a good grasp of the language, they can often be a hindrance, rather than an aid, for novices. Thus, this project aims to develop an intuitive visualisation of these declarations.

The project was, primarily, a software project, with the final deliverable being a desktop application. However, important parts of the project concerned the development of visualisations designed for the user. In addition, testing was done throughout the process to make sure that novices could understand the visualisations in a way they couldn't with the raw code. While this report finishes with a desktop application housing a code editor and a declaration visualiser, an important piece of future work is to migrate the declaration visualiser into a plugin for an Integrated Development Environment already used by programmers, so the project can be used without needing a dedicated application.

This report is structured as follows:

- **Requirements:** Technical and user-interface design requirements
- **Research:** Background research, which helped to tailor the final product around novice programmers
- **Analysis & Design:** Decisions regarding the design in light of the project's requirements
- **Implementation:** A high level description of the processes behind the system's production, including the backend and the user interface
- **Testing:** Testing the product with users, both quantitatively and qualitatively
- **Evaluation:** The results of testing; codifying how well the product holds up against the original requirement

CHAPTER 2

Requirements Capture

The initial specification and context for this project was broad. As such, many different ideas and areas of work were considered in the early stages. Parts of these ideas are discussed as extension work, in Conclusion and Extension Work, for the scenario where the main body of work was completed early. However, the reason for focusing on visualizing C/C++ declarations is twofold. Firstly, the issue of complicated declarations in C/C++ is encountered quickly as a beginner, and is still present when you become proficient. Also, no tool was found that fully aids beginners in this area. Secondly, on the whole, the problem is bounded. While extensions and additions can be made to the core body of work, it is possible to produce a tool that successfully visualises any C declaration, in the given timeframe.

The split in requirements is between Technical, and Visual. While of course they intersect, they are distinct in many ways. Technical Requirements concern the extensibility of the final product into a plugin for IDEs, and portability for use on different machines, with different operating systems and screen sizes. In addition, Technical Requirements concern what can be parsed, and what information is present in the final graphs. Visualisation and User Interface Requirements are crucial to this project, since the visual output of the project is the sole aid given to the user. Firstly, the requirements concern the usability of the final product, to make sure that the user can see the graph when they need to. In addition, they concern the look of the graphs themselves, so the user can better understand C++, and their own code, as a result of use of this product.

2.1 Technical Requirements

While the future of this project will likely be as a plugin for IDEs, such as Visual Studio or Eclipse, for now the graphing engine must be built on top of a code editor. So, the final deliverable will need to have a Code Editor, where a user can type code, load source files and save the output. When it comes to the parsing, this must all be done statically and live, while the user is typing.

The final package must be self-contained, and must not rely on features of one operating system, allowing for use on Windows 10, and MacOS El Capitan and above, as a minimum. It is also desirable that the system can be run on Ubuntu 16. The layout of elements should not be confusing at different screen sizes. However, we can assume that smaller screen sizes (such as tablet or phone sizes) will not be used.

The system should successfully parse a sufficient set of C++ for beginner and intermediate use. While parsing C++ is a difficult task, there must be support for Object Orientated constructs and basic use of templates. For declarations, the system should parse and show a graph for a sufficient range to aid a beginner. This includes:

- Support for *char*, *int*, *float* & *double* datatypes. It is desirable that all primitive data types are supported.
- Support for arrays. With arrays, the size of the array and the type of the array elements must be shown.
- Support for functions. With functions, the type of the input parameter, and the type of the output must be shown. There must also be support for functions that don't have inputs, and for void functions.
- Support for pointers. This includes pointers to primitive data types, arrays, or functions.
- Support for arbitrary variable names. This includes names of declarations of primitive data types, arrays, functions or function parameters.
- Support for the keyword *const*. It is desirable that *volatile* is supported also.
- Support for any combination of these, with the ability to recurse to multiple levels (pointers to pointers, for example).

2.2 Visualisation and User Interface Requirements

Since this project is aimed at providing helpful information through visualizing code, the visual elements of the project are crucial to actually help the user. On the whole, the final deliverable

must be easy to use, without a tutorial needed. When it is possible for a graph to be shown, there must be some indication of this with the relevant declaration, and this indication must be clear and timely, but not distract the user. When a user is looking at a graph, they must know what declaration the graph refers to, and they must be able to edit the code at the same time. The graphs themselves must be easy to understand. For example:

- Functions and pointers must be obviously distinct.
- With datatypes, it must be clear what element the datatype refers to.
- With functions, it must be obvious what the inputs and outputs are, and there must be a clear difference between inputs and outputs.
- With functions without input, or void functions, it must still be clear that a function is present, whilst being obvious that there are not inputs/outputs present.
- With keywords, it must be clear what the keyword applies to, and what the keyword is.

Finally, the layout of the graph must be easy to see. If elements or words are small, there must be the ability to zoom. With complex declarations, no element should be obscured, so the graph can still be read.

2.3 Conclusion and Extension Work

From these requirements, we can see that the final product must be clearly pointed toward use as a plugin within other code editors. In addition, the final product must cover enough of a range of declarations to genuinely help novice programmers when learning C++. Finally, the product must visualize those declarations adeptly and helpfully, even for those who may never have written C++ before. While many of these requirements may be obvious given the context, they are useful reminders when looking at what tools to use, and when testing the product (at both intermediate and final stages). As well as these requirements, there are some further pieces of extension work that were identified as being useful to novice programmers:

1. Adding functionality for pointer assignments, for example `int* ptr = &a`.
2. Adding functionality for classes and structs. For example, given a class, graphs could be shown for member variables and member functions.
3. Actually converting the codebase of the graphing engine into a plugin for an IDE.

CHAPTER 3

BACKGROUND RESEARCH

3.1 User Interfacing

In his seminal book on the subject of User Interfacing, *The Design of Everyday Things*, the designer Don Norman says that there is an inherent paradox in modern technology, where new technologies should benefit our lives, but instead they often add stress, due to added complexity (Norman, 2001, p32). This, he says, is the challenge for the designer. In the case of this project, this holds true. There are many tools and resources that novice programmers can use to aid their learning, and this product has the opportunity to work alongside school and university courses. However, a badly designed product could hinder, rather than help, the path to learning. Programming concepts can be complex, but a well designed User Interface could help to simplify those concepts. In the first section, we first look more generally at user interfacing, and how to make a product that is built around the novice programmer, by being helpful but not distracting. Then, we move onto the specific area of programming, looking at how to create user interfaces that best support novice programmers, including how to make the visualisations of the implementation both clear and engaging.

3.1.1 Important Principles

In his book, Norman goes on to codify seven fundamental principles of designing interactions with users (Norman, 2001, p 72). In this project, three are most pertinent. So, we will look at those three in detail:

- Feedback

In the writer Steve Krug's book on web usability, *Don't Make me Think!*, he wrote that good assistance is timely. For this project, this means that live analysis is a must, with tooltips shown quickly after relevant code is typed. Secondly, good assistance is brief. This project has the option of giving lots of information about the user's code. However, keeping it brief, with an option of giving more information, limits distractions. Finally, good assistance is unavoidable. Using size, colour and placement, a design can be achieved where the user is always alerted to the new information available to them. (Krug, 2014, p 47)

- Mental Model

  The mental model of a system is the user's model of a system. While it may be abstracted away from the reality of the technology itself (for exactly a file directory on a computer), according to Norman, it does not matter if the model is abstracted, as long as it is useful to help the user understand and remember the system's functionality. In our case, the abstraction is between the model the system provides regarding data in C-like languages, and the underlying reality of how those systems are actually implemented.

- Signifiers

  Signifiers communicate what actions can take place on the system. Krug says that, for example, clickable elements must be obviously so (Figure 3.1). Every interface element that a user has to consciously think about takes away from the task at hand, so it must be obvious how signifiers can be interacted with.

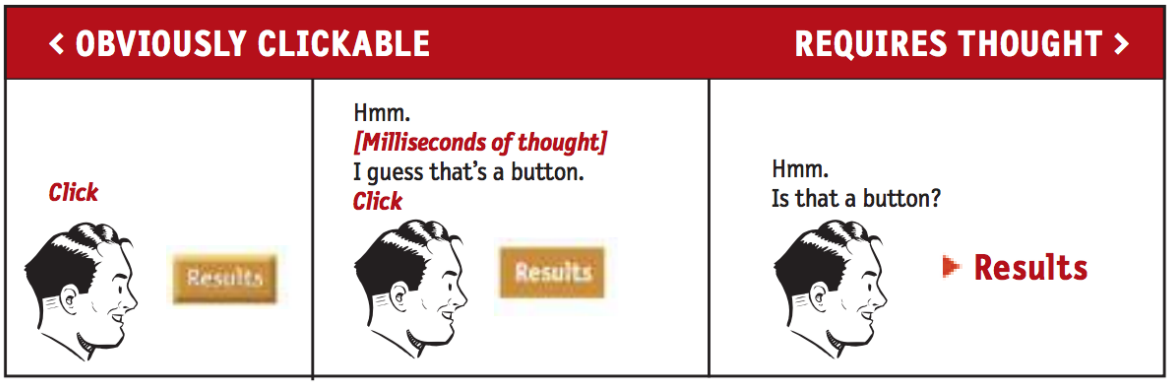


FIGURE 3.1. Obvious action in clickable elements. Figure reproduced from Krug, 2014.

3.1.2 User Interfaces in Programming Software

Pane and Myers undertook an overview of usability issues in different programming environments aimed at novices. They say that, since the research in this area is dispersed among different

sources, "most new [programming environments] are built primarily around technical objectives, perhaps considering only a subset of [relevant] usability issues" (Pane and Myers, n.d., p. 1). Thus, they collated a list of usability issues, so that system designers have a checklist to go through. While the list is broad, we will narrow our focus onto two points, which are pertinent to this project. Firstly, we will look at incremental running. Pane and Myers cite Goldenson (1991), where it is noted that incremental running is important to novices, and those that have a habit of testing their code incrementally are more likely to perform better. However, incremental feedback is not only to find bugs. Goldenson goes onto say that incremental running with graphical debuggers (like the tool my project provides) aids program comprehension in novices even if the code itself runs with no issue. From this, the conclusion was made that a principle for novice programming environments is not to need to press a button to run a visual debugger, instead allowing individual commands to be issued and executed immediately with visible feedback.

A second point made in the report concerns mental models (they use the term "metaphors", however). There are two requirements of the model. Firstly, it must be based on, and close to, a concrete system known to every user. So, with our example of the file directory on a computer, the concrete system is a filing cabinet of folders, perhaps. Secondly, there should be a consistency between the mental model, and the actual inner-workings of the system. There are three problems with consistency; the system has features not in the model, the model has features not in the system, and some features exist in both but are very different in their operation. For example, when computers are anthropomorphised, novices then expect computers to be similarly non-rigid, and thus are not as precise in their description. In the design of this project, therefore, care must be taken to ensure this mismatch does not occur.

3.1.3 Summary

From this study of the research, we can begin to create a picture of both the layout of the code editor, as well as what visual elements will be shown when visualising declarations. With regards to the code editor, we have seen that signifiers (such as interactive elements and buttons) must be clear in what can be achieved using them. This does not necessarily mean using words, since words can be a mental strain. Instead, care must be taken to avoid mental strain, making the user journey as simple as possible. With regards to the visualisations, we looked at mental models. While they are often abstracted away from the inner-workings of the system, they must be based on something concrete, yet still be consistent with the system itself. For example, arrays are commonly visualised as a column of a table. From this research, then, we can begin to understand the requirements of our interface, both in terms of the code editor, and of the visualisations.

3.2 User Testing

Related to the research into User Interface design is User Testing. While, later in this report, the exact details of the testing undertaken are shown, here we look at the background of how the best user testing is undertaken. This research comes from the work of the writer Steve Krug; both from his first book *Don't Make Me Think!* (Krug, 2014), which aims to show how websites and applications should let users undertake intended tasks directly, and from his second *Rocket Surgery Made Easy!* (Krug, 2010), where he goes into more details about professional usability testing. We will firstly examine the framework of user testing, aiming to ascertain when testing should be done, and how many people we should test. This can give us a framework for the rest of the design process. Then, we examine the nature of the tests themselves. This allows us to write interview questions that can gather as much information from participants as possible.

3.2.1 The Testing Framework

In *Don't Make Me Think!*, Krug points out seven important facts about user testing (Krug, 2014, p133-135). These facts can then help build a picture of the best way to go about testing the interfaces of the project. Three are most pertinent to us, so we will unpack those for our own needs.

- "Testing one user is 100 percent better than testing none." A simple point, but an important one. Often designers can be afraid of testing and criticism, or think it's too difficult, so Krug champions a simpler approach to encourage more user testing. In this project, since there is only one developer, it is very easy to remain in a bubble. However, for the project to be successful, and really help novice programmers, testing needs to occur.

- "Testing one user early in the project is better than testing 50 near the end." Krug makes the point that, while designers do not want to test early prototypes, users can feel freer to comment on unfinished products since it's obviously still subject to change. (Krug, 2014, p145) In addition, for testing to shape the design of a product, it must be done early.

- "Testing is an iterative process." Krug reasons that, with the help of Figure 3.2, multiple smaller tests are better for the final product. This may seem like an issue in this project, since the timescale between implementation and the final product is so small. However, as with any iteration design, the amount of change per iteration gets smaller and smaller. So, while an iterative process is important, it is likely that a few iterations will be needed to still gain a large improvement.

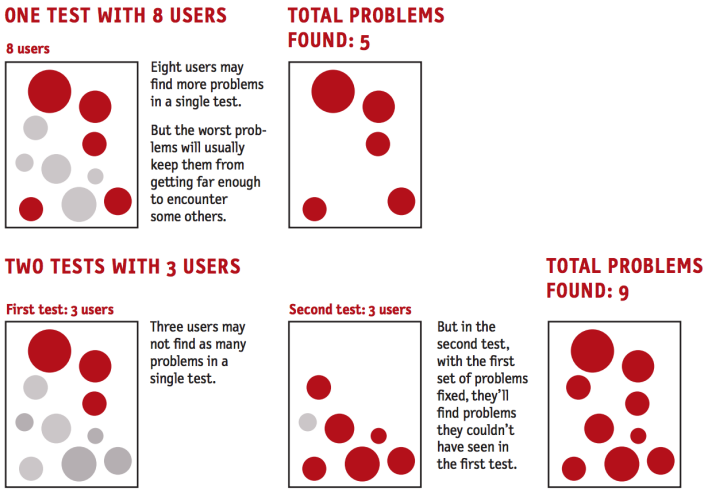


FIGURE 3.2. The case for multiple, smaller test iterations. Figure reproduced from Krug, 2014.

From these facts, we can see that Krug argues for simple, small iterations of testing, throughout the design of a system. In our case, the early stages of the design will likely be small scale, with early prototypes and 2-3 people. This can help refine the direction of the interface early. Then, towards the end of the design, tests will likely be somewhat larger. This does two things. Firstly, for those problems which may be unfixable at later stages, it still allows critical evaluation of the project as a whole. Secondly, it can also help to find and remedy issues which are fixable at later stages.

3.2.2 The Nature of Tests

Next, we will look at the testing itself. When it comes to recruiting participants, of course, the target audience is important. However, unless very specific domain knowledge is needed, it is better to recruit loosely, especially if it allows more testing to occur (Krug, 2010, pp. 40-42). In our project, this is true. Of course, if I test someone adept at programming in C, in fact it may have the opposite effect, since the tool will likely not be needed. On the flip side, it may be difficult to be able to test an absolute beginner without teaching them to some degree. But, in many ways, neither of these things should matter. C declarations can be incredibly complicated, even for expert programmers. In addition, since the project is aimed at novice programmers, there should be no issue with testing the software on those who have never learnt programming. When it comes to questions, there are two types (Krug, 2010, p144). Firstly, "Get it" testing, where the user is shown an interface, and is asked whether they understand the unique selling point of the product, and whether they understand generally how it works without much guidance. Secondly,

"Key task" testing, where users are asked to perform specific and key tasks on the system, to see how well the system performs (Krug, 2014, p144). To find these tasks, we first consider all possible tasks on the system. For this situation, this may be different sets of declaration to visualise. From this set of tasks, we then must consider the critical tasks; both by working out ones that worry us as the developer, and by finding tasks that users had issues with in previous tests (Krug, 2010, p5-155). In addition, we must consider the behaviour of the test conductor, so they can best ascertain information from the participant. The general guideline is to try to get them to externalise their thoughts about the system, while never affecting their behaviour or thoughts. This includes asking questions when the participant is confused or bewildered, clarifying their position when it seems confusing, and answering a question to a question (which allows the user to have their say, and stops the conductor from influencing any decisions).

3.2.3 Conclusion

From this research, we can build up a picture of testing for the rest of the process. Firstly, to balance an iterative design with a short timescale, two rounds of testing are expected to be performed. The first, earlier round will be to see the usefulness of the tools, and how they can be changed to be more useful ("Get it" testing), as well as some "Key task" testing to refine the design of the visualisations themselves. In order to speed up the process, we will not be concerned in these tests with the state of prototypes, nor with the exact programming ability of the user. Of course, however, we will need at least one participant to actual understand the issues, in order to perform proper "Get it" testing. The second round of testing will be more "Key task" testing, to evaluate a more finalised version of the user interface. This will look at whether users can find how to show a visualisation once they've written code, as well as testing if they can understand shown visualisations without prompting. In addition, in order to aid evaluation, it will be worth having metrics for these tests, such as the time to understand a declaration, in order to test the product against competitors, and against the control.

3.3 Competing Products

Before undertaking this project, a set of competitors were found, who may have implemented something similar to what this project seeks to undertake. While Linters were initially looked at, many are distinct from the work of this project. Originally, Lint was produced as "a command which examines C source programs, detecting a number of bugs and obscurities" (Johnson, 1978), and most Linters for C++ stick to this original idea. However, in our case, the work is not focused on providing information at times when code may be buggy, instead allowing users to have a better understanding of what the code is doing, even if the code has no errors. So then, two

competitors were found that both seek to help users to better understand C code in a general sense. The reason for doing this is twofold. Firstly, it allows us to make sure that this project still uniquely fulfils the original project aims. While other products may be similar, it is helpful to know that this project has a unique selling point in some respect. Secondly, examining competitors allows us to, if their codebase is open-sourced, find possible modules to build on top of.

3.3.1 CDecl

CDecl, advertised as a "C gibberish translator" (ridiculousfish and Conrad, n.d.), does two things. Firstly, it allows users to generate complicated C declarations through an English-like syntax. In addition, it does the reverse, generating a human-readable sentence from a complex C declaration (ridiculousfish and Conrad, 2016). Pointers and keywords work how you'd expect. So a complex declaration, like `int const **const foo`, generates an easy to read sentence - *"declare foo as const pointer to pointer to const int"* (ridiculousfish and Conrad, n.d.). Arrays are slightly more difficult to read. For example, `float (*foo)[10]`, generates the clunky (but still readable) sentence *"declare foo as pointer to array 10 of float"*. Functions, and function pointers, become somewhat difficult, especially for beginners. With the declaration `char *(*fp)(float *)`, the sentence *"declare fp as pointer to function (pointer to float) returning pointer to char"* is generated. The difficulty arises since beginners would not necessarily make the inference that *"function (pointer to float) returning ..."* means the functions input is a pointer to a float. While these minor problems exist, it would initially seem that cdecl fulfils the requirements of this project. However, there are two issues with cdecl, that help our project to continue to be unique. The first is that cdecl is not visual. The visual output of our project is an advantage to the text output of cdecl, since programming concepts that may be new to beginners are not explained in cdecl, but can be naturally explained by elements in the visualisations. For example, concepts like pointers or arrays can be explained visually, where they are not explained by cdecl. Also, the issues in the sentences that cdecl provides, like the confusion in function inputs, can be easily solved by a well-designed visual output. The second issue is that cdecl fails on some types of function prototype. In C, it is best practice to include a function prototype, before the code for the actual function. This can also be found in a C header. In these prototypes, the actual names of the parameter values can be left in or out. However, with cdecl, the parser shows "syntax error" if the names are left in. So, for this declaration, found in stdlib.h - `long strtol(const char *__nptr, char **__endptr, int __base)` - cdecl throws an error.

While cdecl may not be perfect, it could make sense to use the underlying infrastructure, and build the visualization engine on top of it. The command line version of cdecl generates an executable file. It is then possible to run this using Node.js (Node.js Foundation, n.d.). However,

the codebase would likely need to be changed so that, instead of generating human-readable text given a C declaration, it generates a data structure containing a representation of the declaration. This is not trivial. The project itself it built on top of Flex & Bison - open source UNIX tools for building a Look-Ahead-Left-Right parser - along with a C source file. While the tokeniser is relatively self-explanatory, the parser and C source are 1000 lines of code each (ridiculousfish and Conrad, 2016). So, as was found through testing, changing just small elements of the output is not simple. In addition, cdecl is only designed for single declarations, so a first parse using another parser would be needed, obtaining the declarations as strings, which is then passed to the cdecl to parse. Finally, as we have seen previously, cdecl fails with some types of function prototype, showing that the grammar isn't perfect in all cases. From this, we can see that it is simply easier to create a single solution that parses a whole source file.

3.3.2 Jeliot-C

Jeliot 2000 visualises method calls and variables in Java, for novice programming students, developed as a desktop application rather than an IDE plugin (Ronit, Mordechai, and Pekka, 2003). The GUI of the application utilises animations to visualize expression evaluation during runtime. The visualisations were broad, including areas for Methods, Expression Evaluation and Constants (Figure 3.3). An empirical evaluation of Jeliot 2000 was undertaken (Ronit, Mordechai, and Pekka, 2003, p10-11) between two classrooms; one without any code visualization - the other utilising Jeliot. The results were that the strongest students did not seem to benefit from the tool, while the weakest students were overwhelmed by it. However, despite this, there seemed to be a marked improvement in the performance of most students when using the tool. Developed from Jeliot, Jeliot-C was created - a similar visualization program for C (Kirby, Toland, and Deegan, n.d., p1). The aim for Jeliot-C was to produce a broad set of visualisations, similar to its predecessor, making its aim overlap with the one of this project. The visualisation tool used for Jeliot-C was Visual InterPreter - a visualization of a subset of C++, called C–. The issue found by Kirby, Toland, and Deegan was that the animation engine of Jeliot had been designed around Java. Thus, it could not represent C concepts such as global variables or pointers. However, they developed Jeliot-C to add these capabilities. Thus, it would seem much of the work supercedes this project. However, while the authors say that "a fully working prototype of [Jeliot-C] has been developed and will be trialled in the Autumn semester of 2010 [at the Institute of Technology Blanchardstown]" (Kirby, Toland, and Deegan, n.d., p4), there has been no known work on Jeliot-C after the prototype and trial. Kirby, Toland, and Deegan were contacted regarding progress into the project, but no comment was given. Thus, since Jeliot-C is either aiding only a small class at this organisation, or none at all, we can safely assume that, by making this project open-source, we can still seek to uniquely fulfil the original project aims.

3.3.3 Conclusion

From this study of competitors, we unfortunately did not find any APIs or modules to base an implementation on top of. This means that we will still need to find an API to a C++ parser, to build an implementation on. However, we did make sure that an open-source module that visualises C declarations still does not exist, and so we can be sure that our project aims still stand. However, in order to further differentiate ourselves from competitors, it would be helpful to do two things. Firstly, as the project aims state, it would be helpful to make sure our tool is open-sourced and pointed towards use within an IDE so that, unlike Jeliot-C, the project can actually be used by novice programmers. Secondly, it would be helpful to make sure that, unlike cdecl, this project does parse function prototypes with parameter names. Doing these things helps us to keep a strong unique selling point.

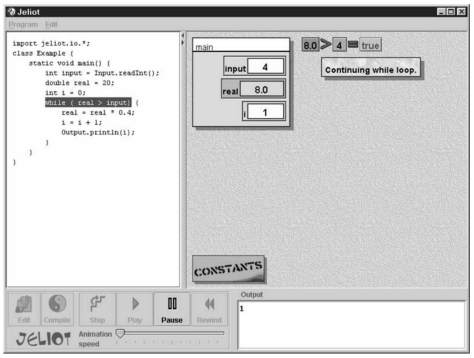


FIGURE 3.3. Example of Java visualisation with Jeliot-2000. Figure reproduced from Ronit, Mordechai, and Pekka, 2003.

3.4 Static Analysis

Both novice and professional programmers often make small mistakes all the time. Most of the time, these are small syntax errors, so the compiler highlights the error, the error is fixed, and development continues. However, this quick feedback cycle normally does not apply to deeper vulnerabilities. So then, within industry, the promise of static analysis is to identify many common coding problems automatically, without attempting to execute the program itself. Gomes et al. (2008) examined static analysis in industry, comparing human-led manual review, tools for static analysis and tools for dynamic analysis. Traditionally, the manual review has four stages between implementation and release, one being a sole effort, one needing a pair, and two needing a team (Figure 3.4). In these stages, everything produced is reviewed, including documentation and requirements, since there is the possibility of errors at all stages. So then, when we look at automated analysis tools, it is clear that the amount of human effort needed to complete it is much smaller. This, in turn, allows analysis of the code during the development cycle, not just at set checkpoints. There are issues with these tools, however. The first is that false positives and false negatives can occur. This is not an issue with the manual review, due to multiple distinct

stages, but is distinctly possible with analysis tools since no software tool is perfect all of the time. Gomes et al. concluded that false negatives are more dangerous, since they can let vulnerabilities slip into production code. So, in general, they should be avoided over false positives (Gomes et al., 2008, p6). In addition, unlike manual review, software analysis does not undergo a full review of all requirements and documentation. In this regard, automated tools cannot be used in isolation. Someone will need to consider the effect of errors they thrown up, and the wider context of the software as a whole. This needs human review at some point down the line.

Looking further at the tools themselves, we have either static or dynamic versions. Dynamic tools, on the whole, test the functional requirements of a system, by analyzing the code during run-time. In contrast, Static tools look for a certain set of software vulnerabilities without running the program. While, on the whole, static analysis is slower, its results are independent of the program environment. In addition, while neither type can review software requirements, "one of the advantages of the static analysis approach during development is that the code is forcefully directed in a way as to be reliable, readable and less prone to errors on future tests." (Gomes et al., 2008, p7). While this is not a replacement for human reviews, a set of design guidelines can be enforced or encouraging through static analysis. The conclusion of the examination, then, is that automated static code analysis is important in the software industry if it is, firstly, easy to use and, secondly, encourages or enforces helpful best practices in the software the developer writes.

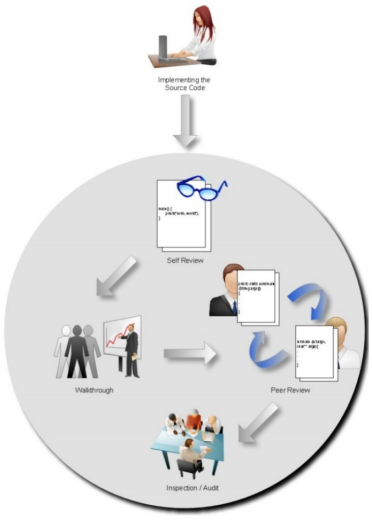


FIGURE 3.4. Types of formal manual code review. Figure reproduced from Gomes et al., 2008.

3.5 Programming Pedagogy

As with any taught skill, there is much academic study behind the teaching of programming. Since the sole aim of this project is to aid those teaching C++, an overview of this study needs to be undertaken. In this overview, we first look at the difficulties novices often find when first encountering C++. This does two things. Firstly, it proves that the project is well-founded on a base of research. Secondly, it allows for targeting tools towards areas that students find particularly

challenging. We then move on to look at the characteristics of models and visualisations that best help students to overcome these initial challenges. This can help in the design of the implementation.

3.5.1 Difficulties in learning C++

Overall, it does not seem that C++ is necessarily an easy language to learn. A 2005 study of 559 students determined that C++ is more of a challenge to novice programmers than Java. While they ascertained particular syntactic constructs that may cause this challenge, including arrays and pointers, these results were marginal, with p=0.5 (Lahtinen, Ala-Mutka, and Jarvinen, 2005). Thus, we must look further afield for insights. A 2002 paper conducted two surveys; the first with 66 computing students from the University of Dundee, and the second with members of the higher education network LTSN - the Learning and Teaching Support Network (Milne and Rowe, 2002). Each respondent ranked the difficulty of different programming concepts on a 7 point scale. While neither survey would be conclusive by itself, the strong positive correlation between them ($R^2 = 0.7704$) adds to the credibility. The surveys seem to agree that C++ is harder to learn than Java. When looking at pointers specifically, they found that the teaching of pointers was easier in C++ after students had experience with Java, rather than the other way around. The conclusion was made that, since pointers are not explicit in Java (all objects are accessed by reference), students were used to the concept when coming to C++, without the syntactical difficulties that C++ has, for example with '*' and '&' notation (Milne and Rowe, 2002, p60).

Another important discussion is between programming paradigms. Large portions of research in programming pedagogy regard the paradigm that best aids students. With C++, we have the choice of both procedural and object-orientated paradigms. Milne and Rowe found that, in both surveys, simple object-orientated constructs - namely Objects and Classes - were not found to be difficult (4.6 out of 7 for LTSN, and 3 out of 7 for students). However, the object-orientated constructs that needed a good grasp of memory and pointers, such as Virtual Functions and Copy Constructors, scored much higher. This is backed up by the fact that Pointers themselves were found to be particularly difficult to master - 6.054 out of 7 for LTSN, and 4.154 out of 7 for students. Of the 28 topics rated, the LTSN respondents rated Pointers the most difficult. From this, the conclusion was that a good understanding of memory storage is necessary for a fuller understanding of object orientation (Milne and Rowe, 2002, p60).

3.5.2 Mental models in programming

At the end of the study by Milne and Rowe, they said this:

> We believe that the results show that the most difficult topics are so ranked because of the lack of understanding by the students of what happens in memory as their programs execute. Therefore, the students will struggle in their understanding until they gain a clear mental model of how their program is 'working' - that is, how it is stored in memory, and how the objects in memory relate to one another (Milne and Rowe, 2002, p63).

From this conclusion, we will look at the characteristics of mental models that help students to understand the inner workings of the program further. As we have already seen, mental models are the models a user has of a system, which are usually implied by the system's user interface. More specifically to programming pedagogy, Du Boulay (1986) talked of the "notional machine". This is the user's abstract model of a computer, which is implied by constructs in the specific programming language they are using - essentially a mental model of a computer when the interface is the programming language itself (Du Boulay, 1986, p431). The suggestion of Du Boulay was that this notional machine should not be a black box, and instead should have a set of tools to observe it - a glass box, so to speak. A study by Wiedenbeck, Fix, and Scholtz (1993) , gives insights into good notional machines. When comparing experts against novices, they found five characteristics experts had more often in their representations (Wiedenbeck, Fix, and Scholtz, 1993, p795-797):

1. **Hierarchical structure:** If a program is split into goals to be achieved, experts had layered representations; a goal at one level is split into subgoals at the next, to an arbitrary depth.

2. **Links between layers:** It is not enough to just have layers - experts could link actual implementation (lower layers), to the task of a specification (a higher layer).

3. **Recurring basic patterns:** The theory here is that, on the whole, programming knowledge is stored as "plans" for handling stereotypical programming problems, which serve as the building blocks of writing programs. As such, experts were found to create plan-like structures in their representations.

4. **Well connected representation:** Here it was found that experts would pay particular attention to interfaces between program structures.

5. **Well grounded in the code:** While representations may be abstract, it was found that experts could easily find where operations occurred, since their representation mapped to specific code elements.

From this, we can begin to see characteristics that should be present in the implementation. Since the tools of this project would be similar to that described by Du Boulay, which gives an insight into the notional machine, it must have the same characteristics of experts' representations, as described by Wiedenbeck, Fix, and Scholtz. Firstly, it must be multi-layered, with an obvious understanding of how the layers intersect. Secondly, it must be made of basic structures, and careful consideration should be made to the intersections between structures. Finally, it should clearly map to the code in some way.

3.5.3 Summary

We have hopefully managed to do two things in this overview of the research. Firstly, we have managed to better target the visualisation tools into certain areas of programming knowledge. Memory seems to be both a building block in learning programming, and also seems to be particularly difficult to understand when learning C++ specifically. So, targeting memory constructs would likely be helpful. Secondly, we have managed to better understand how experts build models of programs. This knowledge can then be mapped into specific visual elements of the design and interface of the project, through use of layout, colour, text, shape and so on. We already saw ways of making these visual elements clear and engaging, in the User Interface section, and can refine our designs through testing.

CHAPTER 4

ANALYSIS & DESIGN

4.1 C++ Parser

Choosing a parser is one of the most important choices regarding the infrastructure of the project. With regards to Technical Requirements, the choice affects the range of declarations that can be parsed, as well as the ease of packaging the final deliverable into a self-contained, portable application. If, for example, the parser relies on a certain operating system, the application immediately lacks portability. With regards to actual graph visualisation, the choice of parser changes how easy it is to visualise code, as different parsers give different amounts of information about the underlying code.

For the purposes of this project, the output of the parser needs to be an abstract syntax tree, which the application has the capability of referring to. It can do this in two ways. The first, more sophisticated way, is through specific bindings to the AST, most likely through the Node Package Manager, or possibly as a Javascript API. The second way would be to turn the complex C++ source into a more manageable data format (such as XML or JSON). This could be undertaken either using a Node.js module, or instead could be distinct from the rest of the infrastructure. Then the XML or JSON representation could be parsed, to ascertain information about the source code.

In addition, the AST itself would preferably be quite close to the original C++ code. For example, if the AST preserves the original variable names, it becomes easier to refer back to these names when giving feedback to the user. Similarly, line numbers would help to map elements of the code to the parsed output.

Using these two rough guidelines, as well as constraints identified in the Requirements Capture, possible solutions can now be explored. To compare solutions, four pieces of source code were parsed, seen in Appendix A.

4.1.1 CastXML

Initially, CastXML seems like an obvious choice. Described as a "C-family abstract syntax tree XML output tool" (King, 2017), the open source project is built on top of the Clang Compiler. On class declarations, the AST shows the structure of the classes well, and doesn't mangle variable names. For example, in Listing 4.1, the variable names *static_method* and *method*, the field names *static_field* and *field*, and the class name *MyClass*, can be referenced in the final XML representation.

Listing 4.1: Part of the CastXML representation of code found in Appendix A.4

```
<Field id="_13" name="field" type="_27" context="_7" access="public"
  location="f1:3" file="f1" line="3" offset="64"/>
<Method id="_14" name="method" returns="_28" context="_7" access="public"
  location="f1:4" file="f1" line="4" const="1" virtual="1" pure_virtual="1"
  mangled="_ZNK7MyClass6methodEv"/>
<Variable id="_15" name="static_field" type="_27c" context="_7" access="public"
  location="f1:6" file="f1" line="6" static="1"
  mangled="_ZN7MyClass12static_fieldE"/>
<Method id="_16" name="static_method" returns="_27" context="_7" access="public"
  location="f1:7" file="f1" line="7" static="1"
  mangled="_ZN7MyClass13static_methodEv"/>
<OperatorMethod id="_17" name="=" returns="_29" context="_7" access="public"
  location="f1:1" file="f1" line="1" inline="1" artificial="1" throw=""
  mangled="_ZN7MyClassaSERKS_">
  <Argument type="_30" location="f1:1" file="f1" line="1"/>
</OperatorMethod>
<Destructor id="_18" name="MyClass" context="_7" access="public" location="f1:1"
  file="f1" line="1" inline="1" artificial="1" throw=""/>
<Constructor id="_19" name="MyClass" context="_7" access="public" location="f1:1"
  file="f1" line="1" inline="1" artificial="1" throw=""/>
<Constructor id="_20" name="MyClass" context="_7" access="public" location="f1:1"
  file="f1" line="1" inline="1" artificial="1" throw="">
  <Argument type="_30" location="f1:1" file="f1" line="1"/>
</Constructor>
```

However, the main issue with CastXML is that the representation is high level, with function

bodies not being shown in the XML at all. For some static analysis, for example one which only concerns control flow between functions, this would cause no issue, however much of the analysis relevant for this project needs an AST representation of function bodies. So, CastXML is not a suitable parser.

4.1.2 SWIG

SWIG (Simplified Wrapper and Interface Generator) is designed to interface code from C/C++ to several high-level programming languages (SWIG, n.d.[a]). Typically, the use case is that SWIG provides an interface, so that the target language can call into the C/C++ code. However, since SWIG creates a parse tree of the C/C++ code to generate this interface, it "can also export its parse tree in the form of XML and Lisp s-expressions" (SWIG, n.d.[a]). This is useful for this project. However, since the purpose of SWIG is to provide an interface to the underlying code, three issues were found.

The first is seen in the SWIG manual, where they say that certain advanced C++ constructs are not fully supported. This is due to SWIG's unrestricted parsing of C++ datatypes (SWIG, n.d.[b], p35). However, since this use case is for beginners, who are unlikely to use advanced features of C++, this does not pose a problem.

The second issue is more pressing; SWIG only creates an interface for globally defined data. So, given code for a pointer declaration in the *main* function, found in Appendix A.2, the parser only copies the code found in *main*, as an attribute of the XML. This is undesirable for this project, since much of the parsing is not on the global scope. However, since the original code exists in the final parser, it could be possible to extract and reparse this.

This leads us onto a third issue. If the declaration from Appendix A.2 is now defined globally, the relevant lines of XML (found in Listing 4.2) need further interpretation, and are not easy to work with. While each input parameter is a separate XML construct, the output of the function, and the fact that the variable is a function pointer, need to be ascertained from an unwieldy construct - `p.f(int,p.float).p.` .

So, while SWIG *could* be used as the parser for this project, a better solution would be desirable.

Listing 4.2: Part of the SWIG representation of code found in Appendix A.2, with the declaration taken out of *main*

```
<attributelist id="178" addr="0x1066c1c80" >
   <attribute name="sym_name" value="fp" id="179" addr="0x1066c2620" />
    <attribute name="name" value="fp" id="180" addr="0x1066c2620" />
    <attribute name="decl" value="p.f(int,p.float).p." id="181" addr="0x1066c2620" />
```

```
    <parmlist id="182" addr="0x1066c1b60" >
        <parm id="183">
            <attributelist id="184" addr="0x1066c1b60" >
                <attribute name="type" value="int" id="185" addr="0x1066c2620" />
                <attribute name="compactdefargs" value="1" id="186"
                  addr="0x1066c2620" />
            </attributelist >
        </parm >
        <parm id="187">
            <attributelist id="188" addr="0x1066c1c20" >
                <attribute name="type" value="p.float" id="189" addr="0x1066c2620" />
            </attributelist >
        </parm >
    </parmlist >
    <attribute name="kind" value="variable" id="190" addr="0x1066c2620" />
    <attribute name="type" value="char" id="191" addr="0x1066c2620" />
    <attribute name="sym_symtab" value="0x1066ae8a0" id="192" addr="0x1066ae8a0" />
    <attribute name="sym_overname" value="__SWIG_0" id="193" addr="0x1066c2620" />
</attributelist >
```

4.1.3 LibClang

Clang, the front-end to LLVM's C/C++ compiler, provides tools to interact with the AST produced by the compiler. Previously, it was possible to get an XML representation of the AST, but this has since been removed. Clang provides three main interfaces to the AST - LibClang, Clang Plugins, and LibTooling. Both LibTooling and Clang Plugins are designed to give you full control over the AST, while LibClang does not. However here, we are indifferent, since only an overview of the structure of the code is needed. Clang Plugins, however, are useful for providing "special lint-style warnings or errors for your project", which could be useful for parts of the final design (The Clang Team, n.d.[b]). However, the reason LibClang is more useful here is that there are bindings between it and other languages, making it easy to ascertain facts about the given code in the target language. Bindings provided by the Clang developers are for Python and, when using Python and LibClang on the sample code in Appendix A, lots of information could be found.

The code itself is traversed using a *CXCursor*, representing an element in the AST. In addition, LibClang also exposes the type of elements as a struct, named *CXType* (The Clang Team, n.d.[a]). The information that can be obtained for certain types (found using the full description of the API (The Clang Team, n.d.[c])) are as follows:

1. **Pointers**: The *CXType* of a pointer type is *CXType_Pointer*. However, a function is provided, *clang_getPointeeType*, to get the *CXType* of what is pointed to.

2. **Arrays**: Given a constant array, LibClang gives functions to find the size (*clang_getArraySize*) and the type of its elements (*clang_getArrayElementType*).

3. **Function**: Given a function, LibClang allows recursion into the function arguments, by providing a function (*clang_Cursor_getArgument*) that returns a *CXCursor* to each function argument. In addition, the return type can be found using the function *clang_getResultType*.

So, we have seen that a set of information can be found for C++ declarations using LibClang. In addition, the location (column and line number) are given for each declaration.

One issue is that, up until now, LibClang's Python bindings have been used, rather than using Node.js. However, multiple libraries exist to provide Node.js bindings to LibClang. Initially however, it seemed that none would run on any recent versions of Node.JS. In the end, Timothy Fontaine's library (Fontaine, 2013), interfaced to LibClang correctly. Since the last update to the library was in 2013, the example code had to be changed considerably to work with the current version of Node.js, but the API can now be exposed. The second issue was that, while some of the API was exposed in the given library, the library will need to be changes, to open up more of the API's functions where needed.

4.1.4 Summary

From examining these three parsers, we see that they all have issues which could hinder progress when wanting to fully parse C++ declarations. CastXML was found to be wholly unsuitable for the purpose, due to its high level nature. SWIG was, in many ways, similarly high-level, but with considerable work, enough information could be exposed from the given C++ code. However, while LibClang was not without it's problems, the fact that it exposed a simple API to grab considerable information from a C++ source file aids with clarity when visualising, and with extensivity when parsing. In addition, while portability considerations are not fully resolved, it being a Node.js module will help further along in the process. When looking back at the Requirements Document, then, this makes it the most suited to the job.

4.2 Graphing Frameworks

The graphing framework is at the heart of the implementation of this project. While we may be able to come up with an excellent visualisation method on paper, this is useless for our purposes without a way of implementing it. There are three broad requirements for the framework. Firstly,

it must have a bed of features suitable for our needs. This includes varied node types, such as a table-like node for array visualisation. Secondly, it must be able to layout graphs in a sensible way, without the need to manually specify the exact position of every node. Finally, it must be able to be somewhat interactive. An important example of this is with zooming in and out, so the words written in complex visualisations can actually be seen. From these three broad requirements, we will go through the process taken to decide on a final framework to use.

4.2.1 Cytoscape

Cytoscape (Franz et al., n.d.[b]) is a Javascript library designed as an open source graph theory library, allowing the developer to display interactive graphs on the web, in both desktop and mobile browsers. Part of this interactivity is a large number of gestures. This is useful for our purpose. The Requirements Capture specifies zoom capability if text is too small to read, and these gestures allow zooming and panning as standard. However, the use case here is not just traditional graph theory. For example, there is a need for nodes inside nodes with function declaration visualisation. In addition, when visualising arrays, a table is needed instead of a traditional node. Luckily, Cytoscape's support of a variety of use cases helps here. Specifically, Cytoscape's support for compound nodes (nodes within nodes), should help with the problem with function visualisation. An example of this can be seen in Figure 4.1. With regards to a table-like node, there is a possibility of using a set of rectangular nodes within another node (using compound nodes) to achieve this. So, hopefully the complexities of the interface can be produced in Cytoscape. In addition, since the library itself is open source, changes can be made if necessary, however of course due to the complexity of the library, this is to be avoided.

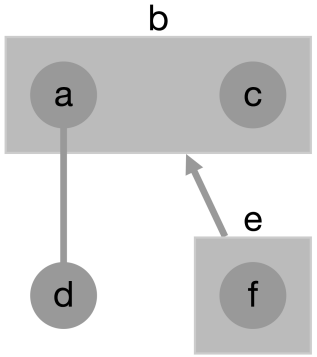


FIGURE 4.1. An example of compound nodes in Cytoscape. Figure reproduced from Franz et al., n.d.(a).

Another issue with Cytoscape may be the actual laying out of nodes. With the library, several preset layouts exist, each with a different algorithm for setting out nodes. There are two useful possible layouts for us. The first, called "preset", is simple - a given pixel coordinate determines the position of each node. While this would add a layer of complexity to the project (since the co-ordinate of each node would need to be worked out individually), it does give full control over the overall graph. The second, called "grid", puts each node on a point in a well-spaced grid. To determine the position of each node in this grid, a sorting function must be given. So,

while exact pixel values would not have to be calculated, a (possibly complex) sorting function would need to be produced. A further option is to write a layout engine. As mentioned on the website, "Layouts are extensions of Cytoscape.js such that it is possible for anyone to write a layout without modifying the library itself" (Franz et al., n.d.[b]) So, if neither of these options suffice, this would be an, albeit difficult, way of determining node layout. From this, we can see that Cytoscape would give enough functionality to produce visualisations, and would give an easy way of interacting with the produced graphs. However, layout may be an issue we need to contend with.

4.2.2 Graphviz and Viz.js

Graphviz (Bilgin et al., n.d.[c]) is an open source framework for graph visualization, aimed at reproducing information in graph diagrams. The input to Graphviz is text written in DOT - a simple graph description language - which Graphviz parses before displaying a diagram. This diagram can either be in a usual image format, or as an SVG. On the whole, Graphviz is very versatile, and certain parts of this versatility are useful for us.

The first feature, useful for array layouts, are record-based nodes. These are, in essence, boxed nodes, with subfields as rows or columns within the main node. An example can be seen in Figure 4.2. Of course, this naturally maps to use in drawing array structures, with a single column, multi-row node producing something that resembles an array. The second feature, useful for function declarations, are node clusters. These are rectangular clusters of nodes, used to delineate different groups of nodes. An example can be seen in Figure 4.3. The use case here is slightly more complex. For this project, a cluster can be used to represent the input to a function or the output of a function. This is helpful since these inputs and outputs can be arbitrarily complex, so a cluster can represent the whole of this input or output.

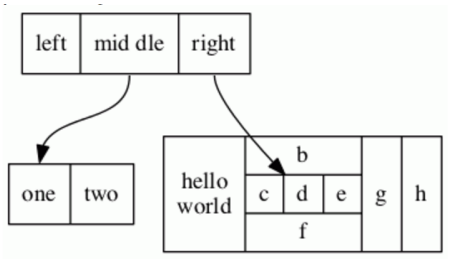


FIGURE 4.2. An example of record-based nodes in Graphviz. Figure reproduced from Bilgin et al., n.d.(b).

The next step is working out how well Graphviz can layout the visualisations of this project. Like Cytoscape, Graphviz has different layout algorithms. The best one for our purposes is the dot algorithm. It is best for hierarchical layouts and directed graphs, which is similar in many ways to our purpose, due to the way that C/C++ memory is pointer based. One issue is that fine-tuning of graphs made using the algorithm is possible, but often very unstable. So, while we have access to a number of settings, it may not be possible to get the exact visualisation we

desire. The final issue is how do we go from Graphviz to an element in HTML. The obvious option for this is Viz.js (Daines, 2017) - a simple, but useful project, which wraps the main Graphviz library with an API to display graphs within the browser. However, as with Graphviz, the graphs produced are static images, and not interactive. So then, we have seen that Graphviz is a robust, feature rich library for producing graph diagrams. However, there may be issues when wanting to undertake complex interactions or produce specific layouts with the engine.

4.2.3 Choosing a framework

Initially, Cytoscape was chosen as the graphing framework for the system, due to it's better interactive elements. The first task was to see if is possible to produce an array-like node with the system. Using the technique described earlier, with compound nodes, a suitable solution was found, which can be seen in Figure 4.4. From here, the next step was to produce these types of notes automatically in code just, for example, given a JSON representation of the declaration. The main issue here was with the layouts of nodes. We previously discussed use of a grid system, which initially worked well. However, when compound nodes were introduced, it was found that two grids would be needed - one for the overall set of nodes, and the other for within the compound node itself. The issue is that Cytoscape did not natively support this. From here, three courses of action could be taken; firstly, to continue to use and adapt the grid layout, secondly, to use another layout system with Cytoscape, and the third to use Viz.js. The decision was made to, initially, prototype with Viz.js, and compare ease of implementation between it and Cytoscape.

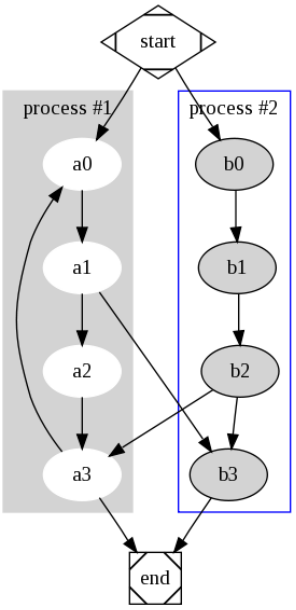


FIGURE 4.3. An example of clusters in Graphviz. Figure reproduced from Bilgin et al., n.d.(a).

When prototyping with Graphviz, however, the richness of features (especially when laying out graphs) helped immensely. While there was no way to fix the exact position of any node, Graphviz allows graphs and subgraphs to specify the direction of the graph layout (left to right, or top to bottom). This, along with the previously mentioned record-based nodes, meant implementation was far quicker with Graphviz. However, when comparing to Cytoscape, there was still the issue of interacting with the graph - especially being able to zoom out or in to the graph itself. In prototyping, though, it was noted that Viz.js produces a native SVG element within the HTML of the system. This allows for simple CSS styles, such as a maximum width (so graphs do not run off the page), and even allows for the use of other Javascript libraries. For

example, the SVG Pan/Zoom library (Riutta, 2017) gives functionality to add interactivity into SVGs, namely panning and zooming into the element itself. So, since Viz.js superseded Cytoscape in most ways at this point, it made more sense to switch to using this instead.

4.2.4 Summary

Here we have seen two viable graphing libraries for use within the project. Initially, it made most sense to use Cytoscape, since it was native to within a web-browser, through Javascript, and thus was more easily interactive. However, the layout of nodes was not easy, so algorithmically producing a well laid out input to Cytoscape was a slow process. When moving onto Graphviz, with Viz.js, the development process was much quicker, due to a richer bed of features and a more sophisticated layout engine. In addition, the loss in interactivity was minimised, as Viz.js embeds graphs straight into the HTML as an easily customisable element. So, from here, Graphviz was chosen for the final product.

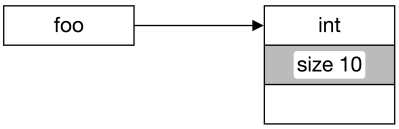


FIGURE 4.4. A prototype for array-like nodes in Cytoscape.

CHAPTER 5

IMPLEMENTATION

5.1 Infrastructure Design

Without a well designed backend, the final deliverable will be unlikely to meet the requirements set out in chapter 2. One requirement is cross-platform portability, which is affected by the backend frameworks used, especially if they are platform specific. In addition, the feel of the final solution, include live updates while typing, depends on how sophisticated the interactions are between the code editor and the parser. Finally, how much is parsed, and the range of graphs displayed, is affected by the design of the parser. Naturally, the parser was the most complex module of the system, due to an often confusing API to LibClang. However, by designing the backend methodically, a solution can be created that meets a large portion of the project's requirements.

5.1.1 Infrastructure Overview

Initially, the core framework of the infrastructure needed to be decided on. The choice of this is relatively straightforward. When choosing the parser (Section 4.1), a simple, extendable API was found from Node.js to LibClang. Then, similarly, when choosing the visualisation engine (Section 4.2), it was concluded that Viz.js is the easiest way to create these graphs. Again, Viz.js is built as a Node.js module. While there are other frameworks to make API calls in, Node's package manager offers 475,000 packages (npm, n.d.), so Node.js will invariably have a pre-existing module to build on top of. Finally, an app has to be built from the final codebase. There are two benefits in using Electron to do this. Firstly, Electron helps to meet the requirement for

portability to both MacOS and Windows (found in Section 2.1). Electron is designed "for building cross-platform desktop applications" (Github, n.d.), by combining Node.js and Chromium (the open source web browser behind Google's Chrome). While this means that apps are large in size, they can be packaged for Mac, Windows and Linux. Secondly, Electron helps the project to be pointed toward use in an IDE, since both Visual Studio Code (an open source IDE developed by Microsoft) and Atom (a customizable code editor developed by GitHub) are built using Electron. This will help for smoother transfer in later stages. For example, a package has already been developed to display Viz.js graphs within Atom (Verweij, 2017), which can be built on top of during the transfer.

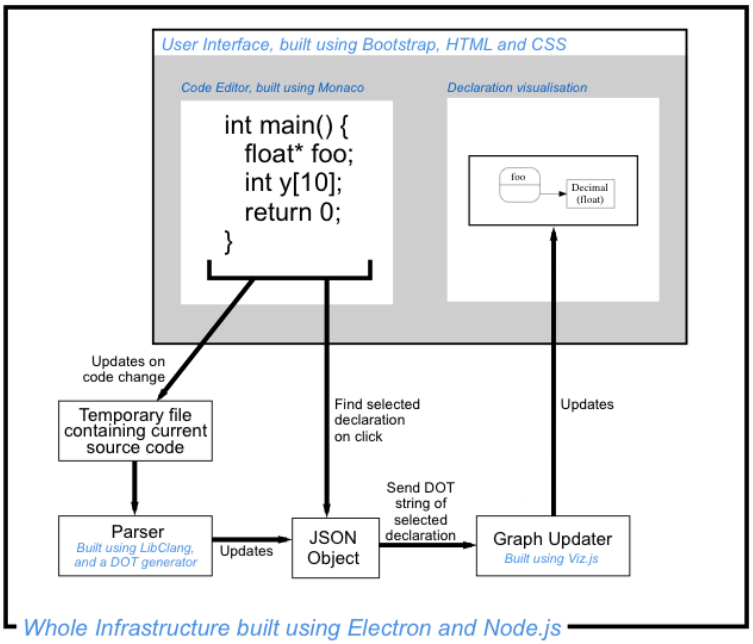


FIGURE 5.1. Overview of the backend processes

Having chosen Electron and Node.js as the backbone of the infrastructure, a framework was needed to allow a user to type code, and then click a declaration to update the visualisation window. A diagram, showing how this is done, can be seen in Figure 5.1. As can be seen, the browser based code editor Monaco (Microsoft, n.d.) was used as the code editor for the interface of the system. The reason is that Monaco has a rich API, which is helpful for our application. Using *onDidChangeModelContent* from the API, the parser can reparse the text in the code editor whenever the code changes. The output of the parser is a JSON object, as seen in Figure 5.1, with each declaration given a line number, a start column and an end column, as well as a DOT string representing the declaration. So, using *onMouseDown* from the API, when the user clicks, the

correct DOT string can be found by checking the line number and column number of the click.

5.1.2 Parser Module

As can be seen in Figure 5.1, the parser outputs a JSON file after parsing the current code editor text. This JSON file contains an object for each declaration. Each object contains the Line Number, Start Column and End Column of the declaration, along with the declarations name, and type (Array, Function, Pointer, Int, Char, Double or Float), as well as a DOT file for visualising the declaration. To create this DOT file, a module was found that generates a graph through function calls, with the final graph outputted as a valid DOT string. Having found this, the necessary control flow was mapped out. LibClang, the parser for the system, goes through each element in the code. If it is a declaration, the parse function is called. A flowchart for this function can be seen in Figure 5.2. It specifies when to create a node, edge or subgraph, as well as when recursion is necessary. Implementing this parser was relatively self-explanatory, apart from one main issue. As we saw in section 4.1, some objects in the declaration (including the declaration as a whole, and function parameters) are of type *CXCursor*, while some (pointers, array elements and function results) are of type *CXType*. While the *CXType* of a *CXCursor* can be found, this is not true the other way around. The workaround for this is to have three inputs into the parse function - the graph object, along with both a *CXType* and *CXCursor* object, with the *CXCursor* object being null where necessary. This allows use of the *CXType* when evaluating an arbitrary input, but when specific features of the *CXCursor* are needed, these can be used.

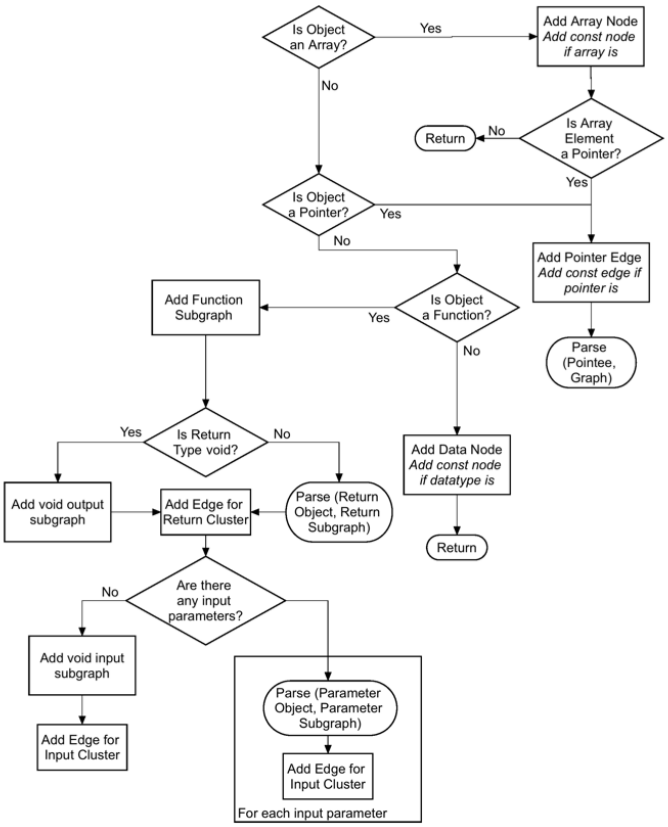


FIGURE 5.2. Flowchart of `Parse(Object, Graph)`; Object is the current LibClang cursor being parsed, and the graph is the current graph/subgraph being implemented.

5.1.3 Summary

Specifying the infrastructure in the backend did two things; firstly, it made sure that requirements are more likely to be met by the end of the process, and secondly, it made the production of each module more directed, since inputs and outputs of the system were already known. So, having built a strategy for the backend, difficult modules (such as the parser) could be created within a wider framework. The parser itself was built last. This allowed testing of the current parser state, by visualising a test set of different declarations.

5.2 Design of Visualisations

As we saw in Section 3.2, testing users early on a design is incredibly important to make sure that potential issues can be found. So, when producing designs for the visualisations, this approach

was used. For these designs, two factors were identified to be of the utmost importance. Firstly, the designs must be scalable; they must cater for more than just the most basic cases, with scope for even the most complex declaration. Since we saw in the Requirements Capture (Chapter 2) that a successful implementation must be able to recurse to multiple levels, so the designs must have the scope to do this also. Secondly, the design must differentiate between different constructs in a clear way, to avoid confusion. This can be done through varying styles and colours in different elements. While this was considered when developing drawings, ultimately the successfulness of the design, and whether a design is confusing, can only fully be ascertained during testing. So, then, these factors were considered when developing designs, with testing highlighting problems along the way.

5.2.1 Initial Drawn Designs

Initially, different designs were prototyped on paper. This allowed for many quick design iterations, with focus on the overall approach of the design, free from any implementation constraints. With regards to scalability, the most important design decision was to make each datatype (including arrays and functions), into a box or set of boxes. Scalability becomes most tricky with pointers, since you can have an arbitrary number of layers in the code, but by homogenising all datatypes, pointers could become as simple as a set of arrows between boxes. So, with the complex example `char (*(*x)[10])(int)` - a pointer to an array of function pointers - the design becomes relatively simple, as can be seen in Figure 5.3 .

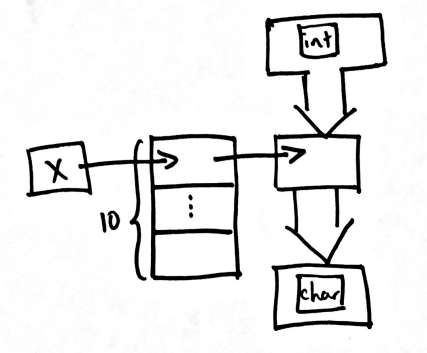


FIGURE 5.3. Drawing of the design for the declaration `char (*(*x)[10])(int)`

The other important design decision regards distinctive design elements. The main worry here is to sufficiently differentiate between arrows; namely arrows denoting function inputs/outputs, and arrows denoting pointers. This is done in two ways. Firstly, the two sets of arrows are always perpendicular, with functions always heading downwards, and pointers always heading to the right. Secondly, the arrow types are distinct. The thinner arrow type is commonly used in programming classes to denote pointers and, due to the background of this project, this then seems like an obvious way to denote them. A thicker arrow type was then used for functions. So then, when looking at Figure 5.4, we can see the distinction in action, even with a complex declaration.

5.2.2 Early Stage Testing

Having completed the full set of drawings, two users were asked to describe different declarations before and after showing them the accompanying drawing. The two users were chosen to have different skill sets; one, a novice, and the other much more advanced. The tests themselves, while not conclusive, did seem to help the users. This was especially true with the novice user, who could fully understand the complex declaration `char *(*fp)(int, float*)`, after being given both a basic explanation of datatypes in C, and the relevant diagram (Figure 5.4). However, two problems were found. Firstly, the visualisations of the "const" keyword were ineffective. In the initial design, the ∞ sign was used, since the keyword denotes eternality. This seemed, in fact, to have the opposite effect on the user; they thought that the ∞ sign meant it could change. The second issue was with the directions of the declarations. While the vertical/horizontal split did not seem to effect the novice, the advanced programmer found this somewhat confusing, since he is used to reading declarations left to right. So, when it came to explaining `void (*signal(int, void (*fp)(int)))(int)`, given the diagram in Figure 5.5 he could not give a full answer, citing the direction issue as a possible cause.

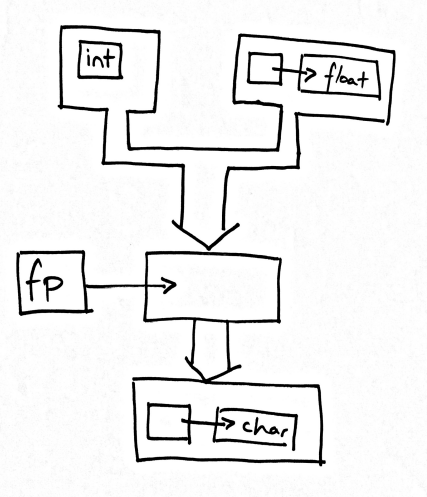


FIGURE 5.4. Drawing of the design for the declaration `char *(*fp)(int, float*)`

5.2.3 Second Iteration

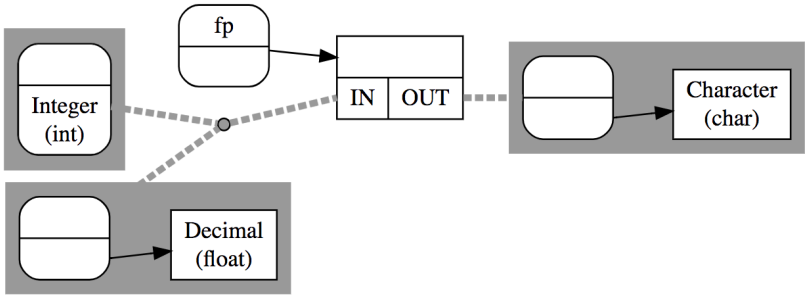


FIGURE 5.6. Second iteration of the design for the declaration `char *(*fp)(int, float*)`

Having completed these tests, the designs were refined into more helpful visualisations. In addition, in this second iteration, the designs were moved from drawings to GraphViz visualisations. While before, drawings were more helpful as they were devoid of implementation constraints, implementing this second iteration in GraphViz ensures the implementation constraints are still taken into account. In this second iteration, it was most important to work on solving the problems highlighted in testing. Regarding the issue with the "const" keyword, the change was relatively simple. Instead of an ∞ sign, a "lock" symbol was used. Often in software, a lock indicates that something cannot be changed. For example, on the iPhone, a lock symbol is used to indicate that the rotation orientation of the phone cannot be changed. So, it was felt this would be a more obvious symbol to use. The issue with the direction of function declarations proved more difficult. While edges could be set from left to right, there was no way of creating a thicker arrow type, like that in Figure 5.4. The workaround was to, rather than using an arrow, use a thick dotted line to denote function inputs. Then, the keywords "IN" and "OUT" denote whether the cluster is a function input, or an output. This can be seen in the second iteration of this declaration, in Figure 5.6.

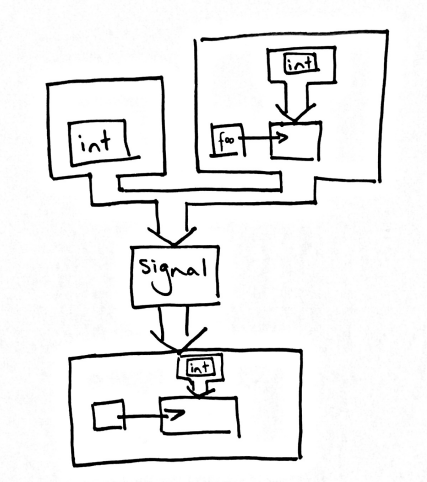


FIGURE 5.5. Drawing of the design for the declaration *void (*signal(int, void (*foo)(int)))(int)*

5.2.4 Summary

From this process, a set of GraphViz visualisations were created, that could then be implemented programmatically by the parser. In addition, by going through iterations in the design, testing along the way, there is better assurance that the system will perform well in the final evaluation. With more time, it would have been helpful to perform more tests, and more iterations. This would have helped ascertain whether the problems found in the first stage of testing were resolved. However, when looking back at the initial design requirements of scalability and distinctiveness in different design elements, both seem to have been met. With regards to scalability, function inputs and outputs can be arbitrarily complex through use of subgraphs in GraphViz. In addition, sets of blocks and arrows allow for pointer depth. With regards to distinctiveness, function edges and pointer edges are distinct, both in style and in colour. So then, we can be relatively sure that the final set of visualisations can be read by a novice.

5.3 Code Editor Design

Implementation of the code editor naturally occurred in stages. Initially, the research into user interfacing guided the production of a prototype. From this prototype, an actual implementation of the design was created. As we have seen, this design was built on top of the Monaco Code Editor. The implementation of this needed to be different to the design in some ways, both due to constraints in the technology, and slight changes in the domain knowledge of the project as a whole. Finally, from this implementation, small-scale user testing was undertaken, taking advice from previous research into user testing, to ensure the design was as user-centred as possible. From here, final changes were made to create a design which the visualisations can then be inserted into.

5.3.1 Initial UI Design

Initially, a prototype of the user interface of the code editor was designed. Here, we will go through two figures from the prototype, to show an overview of the design choices. In Figure 5.7, the situation is shown when the user hovers over a declaration. The code editor itself needs two halves - one for editing the code, and the other for showing a visualisation. The choice was made to split the interface into left & right (rather than top & bottom). This is because most style guides recommend a relatively low limit on the number of characters per line (often the recommendation is 80) and it seems that the left & right configuration of the interface would fit at least 80 characters. So, since the left & right configuration also allows for double the number of lines visible at any one time, this choice made the most sense.

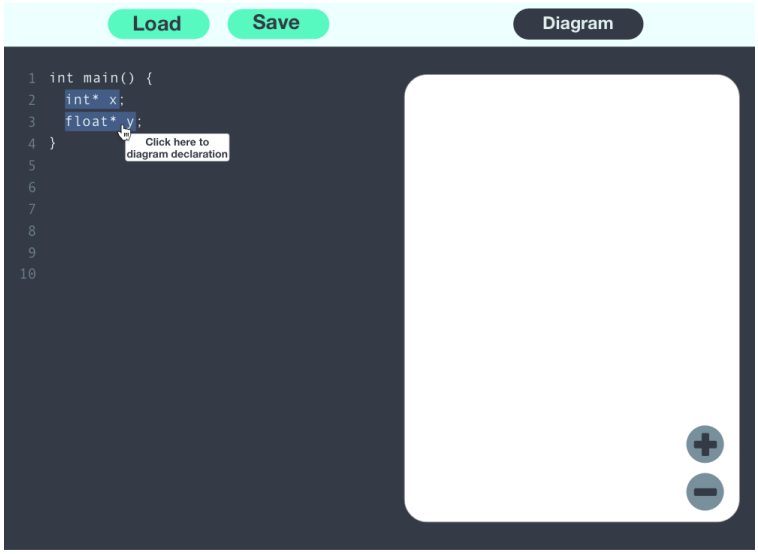


FIGURE 5.7. Design of the User Interface, when hovering over a declaration.

The next choices to be made were in highlighting the declarations themselves. The choice was made to highlight each declaration automatically , as the user types. The colour for this was chosen to be blue, since it clearly shows up in the interface, but is not distracting (unlike warmer colours, such as red, pink or orange). Then, in addition, when hovering over a declaration, a box is shown as an instruction to the user. If, however, it is found in testing that this is distracting or unhelpful, it can be taken away.

A further important element of the interface is the ability to zoom in & out of the editor. In the Requirements Capture, it was specified that "if elements or words are small, there must be the ability to zoom". For particularly complex declarations, it is likely that words would need to be small, in order to fit everything in. So, buttons to zoom in and out were identified as important elements of the interface.

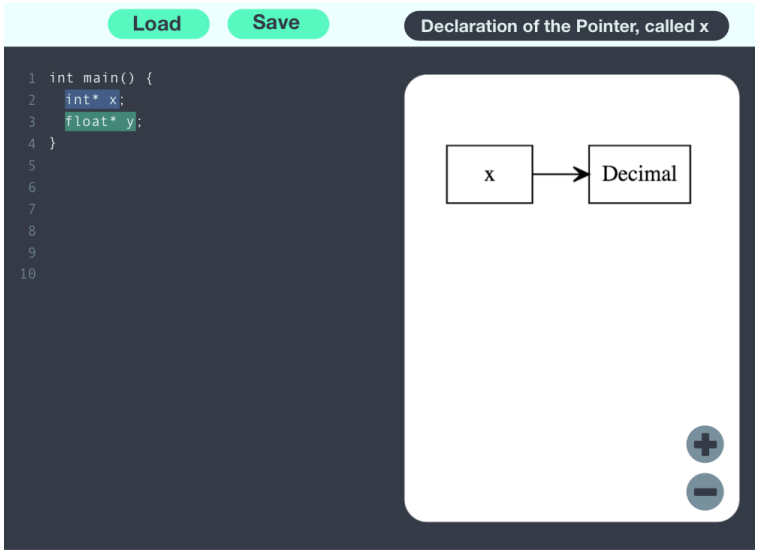


FIGURE 5.8. Design of the User Interface, once a declaration has been clicked on.

Three further elements were needed in the interface; a load button, to load a file into the editor, a save button, to save the current text of the editor, and a description of the declaration currently being visualised. These elements can be seen in Figure 5.8, where the user has clicked on a declaration. Since each of these elements needed a prominent position, a bar at the top was added to house them. The load and save buttons were placed above the code editor, since they map to a use within the code editor and the description was placed above the diagram container, since the description maps naturally to the diagram below. Since the load and save buttons are clickable, they were given a bright green background, while the description was given a darker background. However, all three used the same shape for continuity in the design. In addition, the decision was made to make the description as close to an English sentence as possible, rather than, for example, "A diagram of `float* y`" or something similar. This allows a novice to more smoothly read the diagram. The only issue is that it is not necessarily known what part of the code is actually being diagrammed. So, the relevant declaration is highlighted in green when being diagrammed.

5.3.2 Changes in Implementation

Then, the implementation was created from the design, primarily using the Monaco Code Editor. However, due to issues in basic use of the editor, and increased knowledge of the domain, changes between the user interface and the implementation were found to be necessary. The final design can be seen in Figure 5.9. A big change was in the highlighting of declarations. It was noted that it makes more sense to only highlight the variable name, rather than the whole declaration. For

example, the line `int a, b;` contains two declarations, so it makes more sense to highlight just the variable name. However, this naturally makes the size of the clickable area smaller and, with short variable names, too small. So, the text size of the code editor was increased. This has the consequence of having less characters per line, with the maximum now at 50.

Another necessary change was to the wording in the box shown on hover. In *Don't Make Me Think* (which we looked at in Section 3.1), Krug mentions that "when instructions are absolutely necessary, cut them back to the bare minimum" (Krug, 2014, p47). In this case, the necessity of this box will be identified during testing. However, the sentence in the box can be simplified. This is additionally important since the box could cover prime screen space - the contents of the user's code. So, "click here to diagram declaration" was reduced to "click to diagram". Additionally, the text size was reduced in this box.

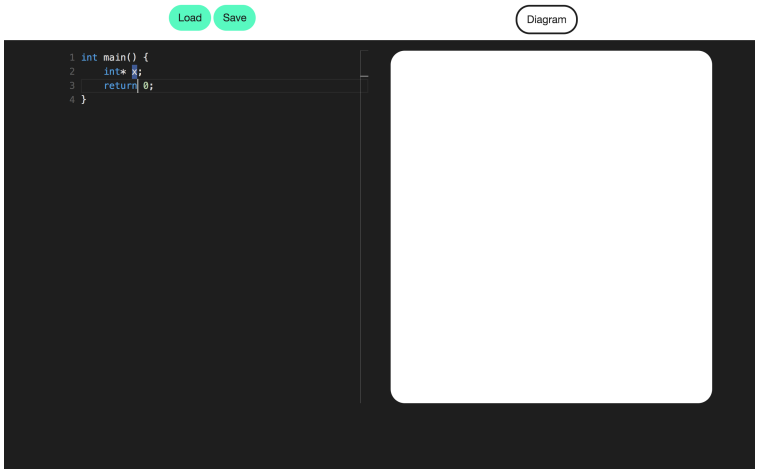


FIGURE 5.9. Design of the initial implementation of the code editor

5.3.3 Changes in Testing

Having produced a prototype of the implementation, small-scale testing was undertaken, in line with research in Section 3.2, on user testing. While only done on one user, it gave me two important insights. Initially, the user was asked to identify the points of the user interface they thought they could interact with. They initially went for the navigation bar, including the diagram description. This is of course a flaw with the design. In *Don't Make Me Think*, Krug said that a user "should never have to devote a millisecond of though to whether things are clickable or not" (Krug, 2014, p14). This is a problem here, since the user has to actively work out whether this element is clickable. Another issue with this description is that it is in a section separate from the actual visualisation, so there may not be a clear mapping between it and the visualisation. So, the decision was made to move this description directly above the visualisation

(away from the bar), and to make the description just text, thus less like a button. This made the top bar seem lopsided, so the "Save" and "Load" buttons were moved to be above the code editor, with the top bar deleted.

A second, more minor point was that the tester agreed that a box is needed when hovering over a declaration. This is backed up by the fact that it took her some time to recognise that the highlighted declarations could be clicked on, and so a box shown on hover may make this process smoother. From here, a final design of the editor was created, which can be seen in Figure 5.10.

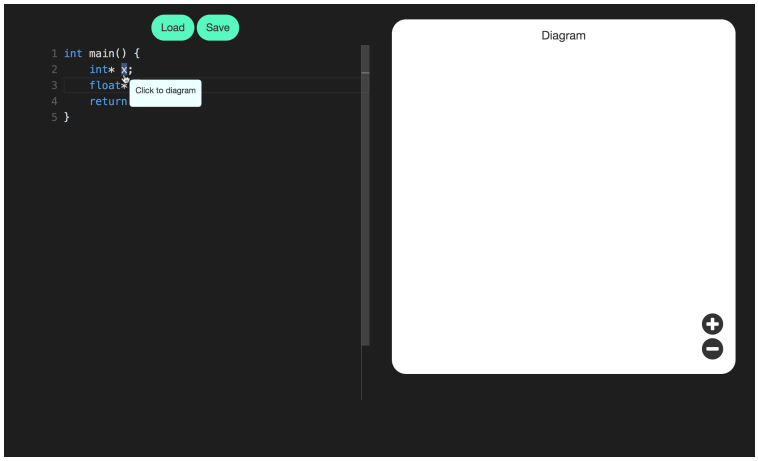


FIGURE 5.10. Design of the code editor interface, after small-scale testing

5.3.4 Conclusion

We have seen different iterations of the design of the code editor, from a basic prototype to a final, user-tested implementation. Throughout this process, previous research, knowledge of C++ as a whole, and common sense helped to inform the choices within the design. This has then hopefully produced a final design which is intuitive, both for experienced programmers and for novices. However, the design process as a whole is iterative, and since further evaluation is still to be undertaken, comments and issues there can continue to help the design develop.

CHAPTER 6

TESTING & EVALUATION

6.1 Interface Testing

6.1.1 Testing Methodology

In testing, it was decided to test a small number of users (3) thoroughly, rather than test a larger sample. In *Rocket Surgery Made Easy!* (Krug, 2010), which we looked at in Section 3.2, Krug agrees, saying that three is enough to be able to find the important issues in a system. He also says to recruit loosely, not having a specific idea of the "ideal" user. So, while I did have two beginners test the system, I also had one tester who is more advanced. In addition, by testing each user more thoroughly, we can have a more in-depth view of their understanding of declarations, and how the visualisations change that understanding .The testing itself had four parts. Firstly, I gave each tester a tutorial on reading C declaration, as well as an explanation of the Clockwise/Spiral rule (Anderson, 1994), in order to read declarations with an arbitrary complexity. In addition, I allowed them to ask questions during this process. This stage was to simulate the kind of teaching they would get during programming classes, since the final deliverable aims to be an accompaniment to conventional programming teaching. Secondly, I asked them questions as they encountered the User Interface for the first time, to see how intuitive the final app is. Thirdly, I showed them different declarations, and asked them to explain the declaration fully in words. These declarations were either given as-is, or given with the corresponding output from CDecl (See Section 3.3.1), or visualised in my system. This allowed comparison between the three types of output. Finally, I asked them a few questions about their experience, and asked them about issues they may have had.

The results of the testing came in two forms, from the recordings of each test. In test sections 2, 3 and 4, any comments, issues and complaints they had about the app were noted. From here, we get a set of qualitative data for each tester. While not conclusive, it helps to ascertain where the problem points are, especially if multiple testers had the same problem. In addition, in test section 2, I timed how long it took from showing the user a declaration, to them being able to fully explain it to me. In addition, if they were incorrect in their explanation, this was noted also. From this set of quantitative tests, we can begin to see how much improvement the system gives, along with a comparison between it and a competitor.

6.1.2 Results of Qualitative Testing

Having undertaken the tests, a set of comments and issues from each tester was collated. In each set, there were comments on both the user interface, and on the visualisations themselves. The full set of these can be see in Appendix B.1. Looking first at comments on the user interface, two comments were seen in multiple users. Firstly, two users commented that the box shown when hovering over a declaration was useful to understand how to use the system, and thus was necessary rather than being irritating. While this box was questioned previously, testing proved the importance of it. Secondly, two users did not initially realise that clicking a declaration did anything, with one user saying that it looked similar to the syntax highlighting of other parts of the code.

With the graph visualisations, one comment was stressed by all three users; that visualisations were more immediately clear than the output of CDecl. The reasons given for this is that, while CDecl's output is easy to read, it is not easy to understand, especially with complex declarations. Some had an immediate understanding of the different arrows, while others took a little bit of time, but all three testers concluded that the diagrams helped more than CDecl. Another conclusion found was that, unlike the previous iteration's testing in section 5.2.2, the use of the "lock" symbol seemed to be understood quickly in all testers. Issues in this area were twofold. Firstly, both beginners could not easily understand multiple levels of pointers, for example in the visualisation in Figure 6.1, due to the presence of the empty boxes. Part of the issue, cited by one, is that he did not fully understand how a pointer could point to a pointer, and an unhelpful diagram seemed to exacerbate this. Secondly, two users expressed concern regarding differentiating arrays, and arrays of pointers (Figure 6.2). While initial confusion went away after becoming acquainted with the visualisation

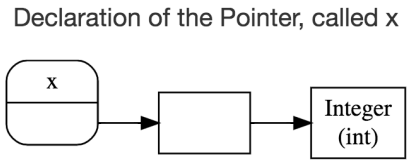


FIGURE 6.1. Visualisation of double layered pointer

structure, both said they would not understand the visualisation of an array of pointers if it were shown to them first.

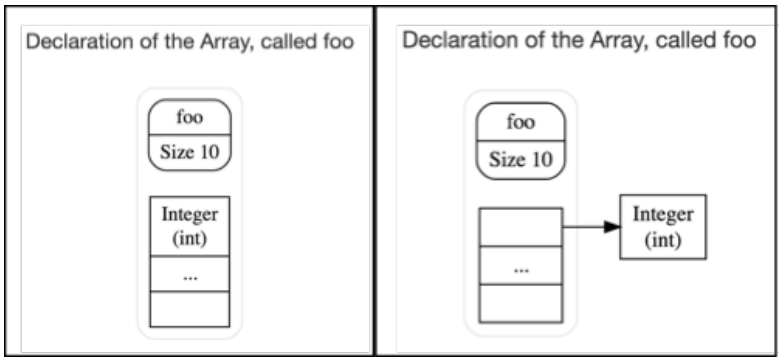


FIGURE 6.2. Left: Diagram of an Integer Array. Right: Diagram of a Pointer Array

6.1.3 Results of Quantitative Testing

Once quantitative data was collected for each user (see Appendix B.2), analysis of the results was undertaken to find patterns. Initially, it was found that there were large outliers in the data. This is because, while many declarations were simple to understand, and others could not be understood, the testers took a long time (over 60 seconds) before they could explain certain declarations. So then, it was decided that the median would be taken to analyse the trends in the data. However, to make sure these slower times were accounted for, a fail percentage was calculated for each method. This fail percentage included times when a user failed to understand, and times when full understanding took above a minute. For the raw declaration, this percentage was 26%. For CDecl declarations, no fails were found. For my visualisation, this percentage was 9%.

	Beginner 1	Beginner 2	Advanced 1	Mean
Median - Raw	23.00	18.00	15.50	18.83
Median - Cdecl	12.00	10.00	9.00	10.33
Median - Mine	13.00	8.50	10.50	10.67

FIGURE 6.3. Graph of the median time for each user, with each method

Then, we moved onto calculating the median time taken for each user for each method. This can be seen in Figure 6.3. While raw declarations scored consistently slower than both other methods, the difference between CDecl and this project was minimal. Finally, I classified each

declaration; either as a function declaration, a function pointer, an array, or a declaration using *const*. I then compared the medians of the raw declarations for each classification, with the same medians for this project. Here, it was found that the difference in understanding time was minimal for functions. However, the other three classifications were significantly faster with this system; on average roughly 35% the time compared to the raw declaration.

	Function	Function Pointer	Const Keyword	Array
Median - Raw	8.00	37.50	26.00	23.00
Median - Mine	10.00	14.00	10.00	7.00
Mine/Raw Percentage	125.00	37.33	38.46	30.43

FIGURE 6.4. Comparing times for different sections of declarations

6.2 Evaluation

In light of the results of user testing, we can evaluate the successfulness of the user interface against the requirements laid out in chapter 2. From the requirements, three important factors must be considered. Firstly, can the app be used without a manual? On the whole, it can be. While there were issues with some users not immediately recognising that declarations can be clicked, the app was intuitive, and responsive to user input. Secondly, are the layouts of the graphs easy to see? Small text did not pose an issue to any of our testers, even on complex declarations, so we can safely assume this is not an issue, and potential zoom buttons were taken out. Finally, was there clarity in the visualisations? At points, there was not, especially in two specific cases. In addition, some declarations were impossible or very difficult to understand, with a 9% fail percentage. So then, while all the testers praised the visualisations, there is room to improve some details of the graphs.

A second important metric we have not yet considered is the coverage of the system. Since LibClang is almost a complete C++ parser, it handles source code containing Object Orientated constructs without issue. In addition, when it comes to creating visualisations, it implements all declaration types specified in the requirements capture; basic types, arrays, functions, pointers, the *const* keyword, and mixtures of these. However, while the *volatile* keyword was a desirable feature, this was not implemented.

In addition, we must also evaluate how well the system performed against the main competitor, CDecl. With regards to coverage, the main issue with CDecl is the handling of parameter names

in function declarations, with this system handles. However, conversely, there are some things implemented by CDecl that aren't existent in this system. Firstly, 'by reference' function inputs are implemented by CDecl. This, however, could be possible future work. The other, key feature that is unique to CDecl is the ability to write declarations through typing English; essentially reversing the parser. However, as we saw with testers, visual representations of the declarations were unanimously easier to read than just text. So, while CDecl and this system are competitors, both have their own unique selling points.

There are two drawbacks in the final app. Firstly, while the user interface was designed to be able to Load and Save source files (as per the Requirements Document), this was not implemented due to time constraints in the project. Another requirement was portability between MacOS and Windows. While packaging the app for MacOS was easy, due to a smooth interface between MacOS and Clang, packaging for Windows proved difficult. While it likely is still possible, these difficulties meant that this requirement is, for now, also not met.

CHAPTER 7

CONCLUSION AND FURTHER WORK

The final application delivered by this project should not really be, in any sense, a final product. There is clear scope for further work to be done in this specific area, along with the more broad area of visualising C, C++ and other programming languages. However, by thinking carefully over the visualisation design of the project, through research, testing and iterations of development, the application did achieve something that is both novel and helpful. Novel in the sense that an open-source, extendable application for visualising C/C++ declarations does not seem to exist until now. Helpful in the sense that, during the course of an hour, two beginners could begin to read declarations that most intermediate programmers would struggle with.

One major difficulty in this project was interfacing to arbitrary codebases, including Viz.js and LibClang. Confusing, and sometimes misleading, documentation meant that often the development process was slow. However, by making sure a high-level view of the system was undertaken, problems in individual elements could be mitigated.

When it comes to further work, the obvious next step would be to first package the application for Windows, and then migrating the application into a code editor or IDE, like Atom or Visual Studio Code. This would allow more users, and would encourage other developers to build on my prototype. Further enhancements could include more declarations, into more object orientated constructs, such as structs and classes in C++. This would likely need a set of more refined visualisations, however, through more iterations of design and testing.

To download the final app, follow the README at github.com/RaviWoods/cpp-analysis-for-education

APPENDIX A

CODE SAMPLES FOR C++ PARSER TESTING

A.1 Variable Printing

Simple Source file which declares and prints a variable

```
#include <iostream>
int main() {
    int x = 10;
    std::cout << x << std::endl;
    return 0;
}
```

A.2 Pointer Declaration

File to test a relatively complex pointer declaration

```
int main() {
    char (fp)( int, float );return 0;
```

A.3 Control Flow

File to test basic control flow with an if statement

```
#include <iostream>
int main() {
    int x = 10;
    int y = x5/2;if (y > 20) x = 7;return 0;
```

A.4 Classes

Testing capabilities with C++, through a class declaration

```
class MyClass {
  public:
    int field;
    virtual void method() const = 0;

    static const int static_field;
    static int static_method();
};
```

APPENDIX B

RESULTS FROM TESTING

B.1 Qualitative Data

Sets of comments made by the three testers

Beginner 1

Comments on User Interface

- Box shown on hover made it obvious what clicking does

Comments on Visualisation

- Lock signs initially confusing, clicked after a couple of seconds
- Often had a conflict between what the diagram said and what the declaration said
- Initially, a difficulty in the diagram regarding arrays and pointers
- Very intuitive
- Immediate understanding of two arrow types
- Didn't understand empty boxes in layered pointer declarations

Beginner 2

Comments on User Interface

- Didn't recognise highlighted declarations were clickable
- Highlighted declarations looked just like syntax highlighting.

- Liked how declarations were immediately highlighted

Comments on Visualisation

- Didn't understand blank pointer
- Found graphs much easier to understand
- Understanding of const clicked with the lock symbol
- Initially conflated int* x[10] and int x[10]

Advanced 1

Comments on User Interface

- Understood that highlighted declarations were clickable
- Box shown on hover was not annoying

Comments on Visualisation

- Basic cases seemed clear and easy to read
- Diagrams help in every way
- Can read the text, but visualization has much quicker understanding
- Difficulty with Arrays of pointers in diagrams
- Found complex declarations slow (but not impossible) to read on diagram
- Layout of function pointers is odd and not immediately obvious
- Locks definitely made sense with constants
- Would be better with a key/legend

B.2 Quantative Data

Time taken for each tester to understand each declaration

Method	Section	Beginner 1 (secs)	Beginner 2 (secs)	Advanced 1 (secs)
Raw	*Function*	37	21	
Raw	*Function*	8	7	
Raw	*Function*	8	6	
Raw	*Function*	100	39	6
Raw	*Function Pointer*			FAIL
Raw	*Function Pointer*	60	15	FAIL
Raw	*Const Keyword*	23	26	34
Raw	*Array*	FAIL	35	25
Raw	*Array*	23	13	5
Cdecl	*Function*	6	10	
Cdecl	*Function*	10	12	
Cdecl	*Function*	12	5	
Cdecl	*Function Pointer*			30
Cdecl	*Function Pointer*	21	12	9
Cdecl	*Const Keyword*	12	10	5
Mine	*Function*	10	7	
Mine	*Function*	10	7	9
Mine	*Function*	8	3	
Mine	*Function*	13	12	
Mine	*Function*	27	17	12
Mine	*Function Pointer*			90
Mine	*Function Pointer*	13	15	13
Mine	*Const Keyword*	72	10	8
Mine	*Array*	40	7	5

Bibliography

Anderson, David (1994). *The Clockwise/Spiral Rule*. URL: http://c-faq.com/decl/spiral.anderson.html.

Bilgin, Arif et al. (n.d.[a]). *Clusters*. URL: http://www.graphviz.org/Gallery/directed/cluster.html.

Bilgin, Arif et al. (n.d.[b]). *Node Shapes*. URL: http://www.graphviz.org/doc/info/shapes.html.

Bilgin, Arif et al. (n.d.[c]). *Welcome to Graphviz*. URL: http://www.graphviz.org/.

Daines, Mike (2017). *Viz.js*. URL: https://github.com/mdaines/viz.js.

Du Boulay, Benedict (1986). "Some Difficulties of Learning to Program". In: *Journal of Educational Computing Research* 2.1. DOI: 10.2190/3LFX-9RRF-67T8-UVK9.

Fontaine, Timothy J (2013). *Node-Libclang*. URL: https://github.com/tjfontaine/node-libclang.

Franz, Max et al. (n.d.[a]). *Compound Nodes*. URL: http://js.cytoscape.org/demos/compound-nodes/.

Franz, Max et al. (n.d.[b]). *Cytoscape.js*. URL: http://js.cytoscape.org/.

Github (n.d.). *About Electron*. URL: https://electron.atom.io/docs/tutorial/about/.

Gomes, Ivo et al. (2008). "An overview on the Static Code Analysis approach in Software Development". In: *An overview on the Static Code Analysis approach in Software Development*. URL: https://www.researchgate.net/publication/267302639_An_overview_on_the_Static_Code_Analysis_approach_in_Software_Development.

Johnson, S C (1978). "Lint, a C Program Checker". In: *COMP. SCI. TECH. REP*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841.

King, Brad (2017). *CastXML*. URL: https://github.com/CastXML/CastXML.

Kirby, Stephen, Benjamin Toland, and Catherine Deegan (n.d.). "Program Visualisation tool for teaching programming in C". In: URL: http://www.iiis.org/CDs2010/CD2010IMC/ICETI_2010/PapersPdf/EB134TP.pdf.

Krug, Steve (2010). *Rocket surgery made easy: the do-it-yourself guide to finding and fixing usability problems*. New Riders.

Krug, Steve (2014). *Don't Make Me Think!* 2nd ed. New Riders.

Lahtinen, Essi, Kirsti Ala-Mutka, and Hannu-Matti Jarvinen (2005). "A study of the Difficulties of Novice Programmers". In: *ACM SIGCSE Bulletin* 37.3, p. 14. DOI: 10.1145/1151954.1067453.

Microsoft (n.d.). *Monaco Editor*. URL: https://microsoft.github.io/monaco-editor/index.html.

Milne, Iain and Glenn Rowe (2002). "Difficulties in Learning and Teaching Programming - Views of Students and Tutors". In: *Education and Information Technologies* 7.1, pp. 55–66.

Node.js Foundation (n.d.). *Child Processes*. URL: https://nodejs.org/api/child_process.html.

Norman, Don (2001). *The Design of Everyday Things*. MIT.

npm, Inc. (n.d.). *Node Package Manager*. URL: https://www.npmjs.com/.

Pane, John and Brad Myers (n.d.). "Usability Issues in the Design of Novice Programming Systems". In: *Research Showcase @ CMU*.

ridiculousfish and David R Conrad (2016). *cdecl: Source Code*. URL: https://github.com/ridiculousfish/cdecl-blocks.

ridiculousfish and David R Conrad (n.d.). *cdecl: C gibberish to English*. URL: https://cdecl.org/.

Riutta, Anders (2017). *svg-pan-zoom*. URL: https://github.com/ariutta/svg-pan-zoom.

Ronit, Ben-Bassat Levy, Ben-Ari Mordechai, and Uronen Pekka (2003). "The Jeliot 2000 program animation system". In: *Computers Education* 40.1, pp. 1–15. DOI: 10.1016/S0360-1315(02)00076-3.

SWIG (n.d.[a]). *Simplified Wrapper and Interface Generator*. URL: http://www.swig.org/.

SWIG (n.d.[b]). *SWIG-3.0 Documentation*. URL: http://www.swig.org/Doc3.0/SWIGDocumentation.pdf.

The Clang Team (n.d.[a]). *C Interface to Clang*. URL: https://clang.llvm.org/doxygen/group__CINDEX.html.

The Clang Team (n.d.[b]). *Clang 5 documentation*. URL: https://clang.llvm.org/docs/Tooling.html.

The Clang Team (n.d.[c]). *Type information for CXCursor*. URL: https://clang.llvm.org/doxygen/group__CINDEX__TYPES.html.

Verweij, Sander (2017). *atom-graphviz-preview-plus*. URL: https://github.com/sverweij/atom-graphviz-preview-plus.

Wiedenbeck, Susan, Vikki Fix, and Jean Scholtz (1993). "Characteristics of the mental representations of novice and expert programmers: an empirical study". In: *International Journal of Man-Machine Studies* 39.5, pp. 793–812. DOI: 10.1006/imms.1993.1084.