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

(Likely 1-2 pages)

- Explain motivation for the project, from personal experience, and the previous attempts for "simple" solutions (Anderson, 1994)
- Explain report as mixture between the software side, as well as the important of the design side to aide understanding.
- Structure of Report. Analysis & Design explains choices behind the project's infrastructure. Implementation is then how I put that infrastructure together, and how I designed the interface.

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 visualizes 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. Visualization 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. So, any libraries used must be small enough to be stored with the source code and the whole package size must not exceed 500MB. In addition, the

package 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 Visualization 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 Visual Studio.

CHAPTER 3

Background Research

3.1 Analysis of C++

Introduction needed, explaining that we need to establish the importance of static analysis, and the theory behind declarations (so we can build implementation on solid groundwork).

3.1.1 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 3.1). 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 the 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

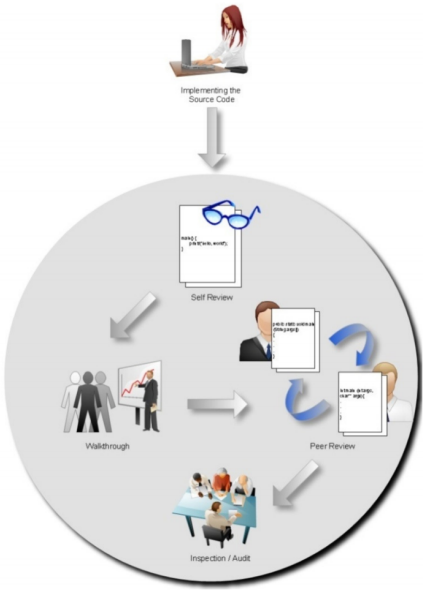

FIGURE 3.1. Types of formal manual code review. Figure reproduced from Gomes et al., 2008.

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 practises in the software the developer writes.

3.1.2 Parsing C++

A more theory based second paragraph.

1. Parsing C++ is undecidable http://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html.

This is not an issue, but does mean that the parser may not be perfect the whole time.

2. However, C declarations do have a grammar (see Figure 3.2). Explain the theory behind this grammar, is sections; keywords, arrays, functions, pointers. In addition, why we're leaving out structs.

3. Putting it into practise. From this compilers textbook (http://bit.ly/2rYLxLh), explain how declarations can be parsed.

How many	Name in C	How it looks in C

zero or more	pointers	one of the following alternatives:
		`* const volatile`
		`* volatile`
		`*`
		`* const`
		`* volatile const`

exactly one	direct_declarator	*identifier* or *identifier* [*optional_size*] ... or *identifier* (*args...*) or (*declarator*)
zero or one	initializer	= *initial_value*

FIGURE 3.2.

3.2 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 this 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.

3.2.1 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 where, and what, action can take place on the system. Krug says that, for example, clickable elements must be obviously so (Figure 3.3). Everyone 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.

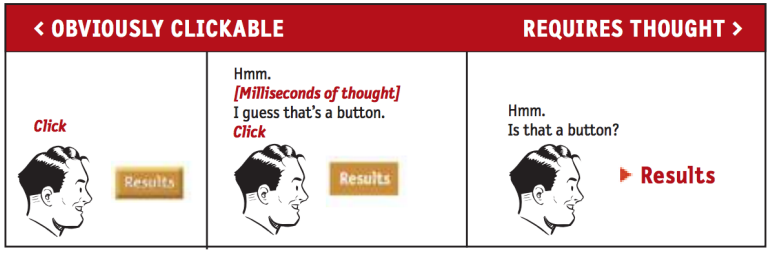

FIGURE 3.3. Obvious action in clickable elements. Figure reproduced from Krug, 2014.

3.2.2 User Interfaces in Programming Software

Second Paragraph concerning User Interfacing in Programming Software, with research from (Pane and Myers, n.d.)

1. Support Immediate Feedback. Restating some of what was said before, but more specific to the field. Graphical debuggers aid in program comprehension even if there are no bugs, but they must be executed immediately with visible feedback.

2. Choose an Appropriate Metaphor, and have a Consistency with the Metaphor. In the section on Programming Pedagogy, I talk about the characteristics of good mental models when learning programming. Here, I talk on the actual physical metaphors and visual elements used. Novices frequently encounter difficulty with the limits of the metaphor or analogy: mistakes arise out of attempting to extract too much from it. For example, when computers are anthropomorphized, novices tend to expect computers to be non-rigid like humans, and thus are not precise in describing a task. Conversely, good metaphors should be based on, and conceptually close to, a concrete real-world system that is widely known by the user audience.

Conclusion, showing what we've learnt, and specifically how that applies to both the code editor and the visualisations.

3.3 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, 2014), 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.

3.3.1 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 3.4, 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.

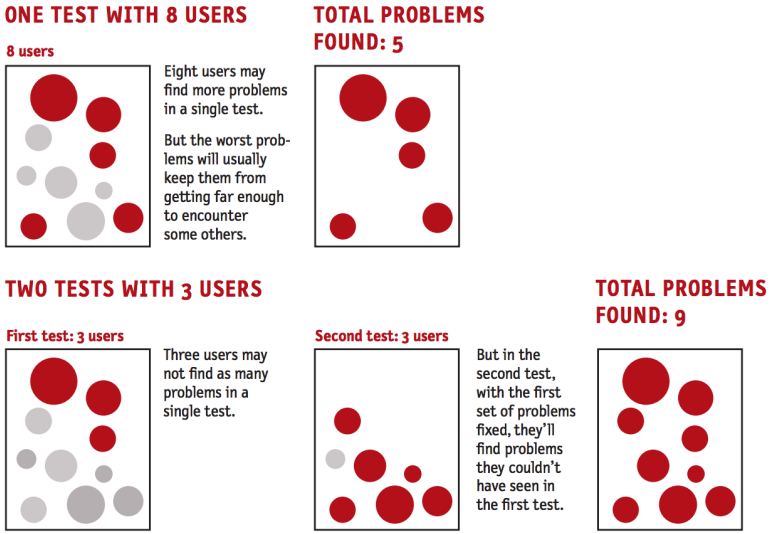

FIGURE 3.4. The case for multiple, smaller test iterations. Figure reproduced from Krug, 2014.

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.

3.3.2 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).

3.3.3 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.

3.4 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.

3.4.1 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 practise 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.

3.4.2 Jeliot-C

Jeliot 2000 visualises method calls and variable 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 3.5). 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.

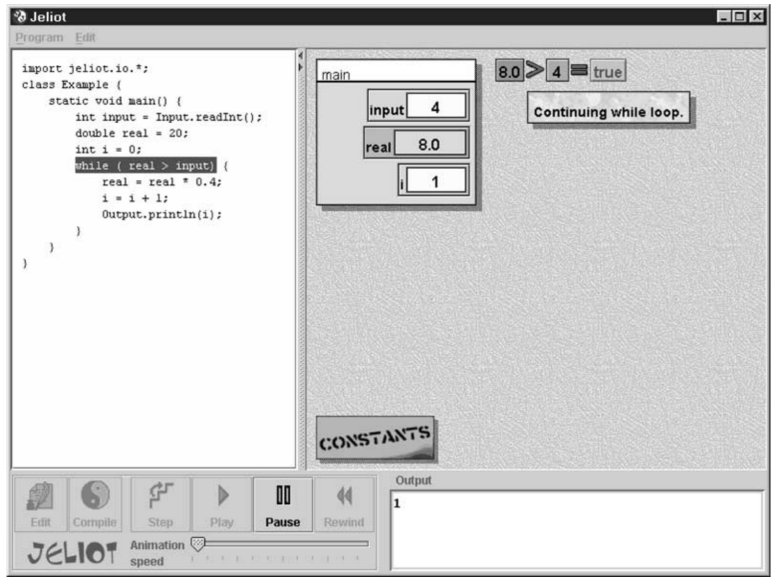

FIGURE 3.5. Example of Java visualisation with Jeliot-2000. Figure reproduced from Ronit, Mordechai, and Pekka, 2003.

3.4.3 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.

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 Conclusion

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 aren't 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. This does pose a problem for two stage pointers (for example `int** foo`), since there is no obvious way to ascertain that an integer is present in the declaration.

2. **Arrays**: Given a constant array, LibClang gives functions to find the size (*clang_getArraySize*) and the type of its elements (*clang_getArrayElementType*). This poses a similar problem to before, with seemingly no way to ascertain the integer type of an array of pointers to int.

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*. However, we have the same problem as before, that there seems to be no way to find the full return type if a pointer to an integer is returned.

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. However, there are some issues when working with pointer types in some cases. One solution to this is to use a tokeniser, which LibClang provides (The Clang Team, n.d.[a]), and manually find these types. While not ideal, it does full parsing of declarations

The final 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 Conclusion

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 it's 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. However, we still need to look at how recursive pointer cases can be handled, and other types of multi-layered declaration.

4.2 Graphing Libraries

(Likely 1-2 pages)

Go over each possibility

- Viz.js (found here - https://github.com/mdaines/viz.js/). Not the best API with regards to bugginess, since it In addition, viz.js, as with graphviz, is not dynamic or interactive in it's nature.
- d3.js (found here - https://d3js.org/). Very complicated, and used for far more than just graphing. Some libraries have been written for it, including this (https://github.com/cjrd/directed-graph-creator). Would need a fair amount of tweaking to work, however. The only advantage is that d3 is complex, so hackable if issues arise.
- treant.js (found here - http://fperucic.github.io/treant-js/). Opposite of d3 - very specific for tree structures. Useful features, but would probably be too simple.
- Cytoscape (found here - http://js.cytoscape.org/). Seemingly good mix - has enough features to hack with (will show an example or two here). In addition, allows compound graphs, which allows the whole of a function paramater to be in its own box, no matter how complicated it is. Slight issues with layout, however.

CHAPTER 5

IMPLEMENTATION

5.1 Infrastructure Design

(Likely 2-3 pages)

- Explaining (and diagramming) the backend infrastructure, with an overview of how I create JSON from LibClang, pass it to the layout function, which lays out nodes within the graph. Explain Electron and the use of Monaco as a Code Editor
- Cross-platform and App packaging: Explain more about how the app went from being just Javascript code, to HTML/CSS/Javascript within Electron, to an app that works cross-platform.

5.2 Interfacing with LibClang

(Likely 2-3 pages)

- Test Driven Development and QUnit Infrastructure. Explain the development process; firstly, create a unit test environment with QUnit, secondly, write requirements for the output JSON, thirdly, write pairs of C++ inputs and JSON outputs as tests and, finally, write JS which interfaces with LibClang and passes the tests
- JSON requirements. Go over the JSON requirements, and why they are the way they are
- JSON generation from LibClang. Go over the development process to meet the requirements (not necessary if the code is boring).

5.3 Iterative Design in Graphing UI

(Likely 4-5 pages)

- Initial drawings, with recursive layout patterns. Explanation of some individual drawings, especially how they can work in recursive areas, for example functions pointing to functions.
- Initial tests. Two tests conducted on these drawings. Explain the tests briefly, what was ascertained, and how the drawings changed.
- Changes from Drawings to Implementation. Explain what needed to change when we moved from drawings to software, due to constraints in the graphing library.
- Second stage of initial tests. Two test conducted on the initial code prototypes. Explain whether the transfer from drawings to software was successful, and the issues that arose.

5.4 Overall Layout and User Experience

(Likely 2-3 pages)

- Code Editor with Monaco. Explain the design of the Code Editor. Mostly simple and self explanatory. Small scale insanity tests were carried out. Details of these, and explain how issues were solved.
- Graphing Layout. Explain how we go from a JSON file to a full layout. Will partly take in ideas from the previous section.
- Code to Graph Mapping. Explain how we interface from the Code Editor and the visualisations themselves. This will include colour mappings and/or interactive elements, depending on what is implemented in time.

CHAPTER 6

TESTING

6.1 Coverage Testing

(Likely 2-3 pages)

- Source Files. Talk about how the source files were made, and how edge cases were found
- Failures/Issues in JSON generation. If any issues are present in the JSON code (ie, if QUnit still finds errors), talk about what they are and why this is.
- Failures/Issues in Graphs. If the graphs display any issues from what I expected (for example, parts can't be seen due to long variable names) talk about these. Include a full set of graphs in appendix

6.2 UI Testing

(Likely 3-5 pages) Specific testing details. Refers often back to previous section

- Interviewees. How many interviewees, how they were chosen, etc.
- Explain brainstorming of the variety of questions, and how they were whittled down to a manageable set.
- Overview of Results. Split results into sections, and talk specifically about one or two notable sections.

Chapter 7

Evaluation

7.1 Technological Evaluation

(Likely 1-2 pages)

- Extensibility and Portability: Discuss use as a standalone app, including simple cross-platform testing. Move on to discuss how well the system is set up to be moved towards use as a Visual Studio Code plugin.
- Coverage: Discuss coverage, possibility against competitors also. In addition, discuss issues with graphing layout.

7.2 User Interface Evaluation

(Likely 2-3 pages)

- Ease of Use: How well did users find it to use the Code Editor. Did they get it?
- Increased Understanding: Discuss the feelings of participants with regards to increased understanding, especially novices. Explain metrics, if taken (for example how much quicker they were at understanding with this product).

Chapter 8

Conclusion and Further Work

8.1 Conclusions

8.2 Further Work