A VISUAL C++ ANALYSIS PROGRAM FOR EDUCATIONAL USE

RAVI B. WOODS

SUPERVISED BY DR THOMAS J. W. CLARKE

INTRODUCTION

Often when novices come to programming, they are used to interacting with computers through well-designed visual interfaces, and so they find it difficult to understand how to interface to a computer without something inherently visual.

For example, when declaring variables in C and C++, novices can often become confused with the complex syntax. In the past, techniques have been proposed to help programmers more easily understand these declarations. One example is 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. So, this project aims to develop an intuitive visualisation of these declarations.

SCOPE

This project concluded with a desktop application housing a code editor and a declaration visualiser, so that users can write code, and select declarations to visualise.

However, an important piece of future work is to migrate the declaration visualiser into a plugin for an Integrated Development Environment, so the project can be used without needing a dedicated application.

As for coverage, basic datatypes, as well as pointers, arrays and functions, were identified as the most important types needing visualisation. In addition, the const keyword was found to be important.

Of course, combinations of these are also allowed, which the visualisations needed to deal with, such as function pointers.

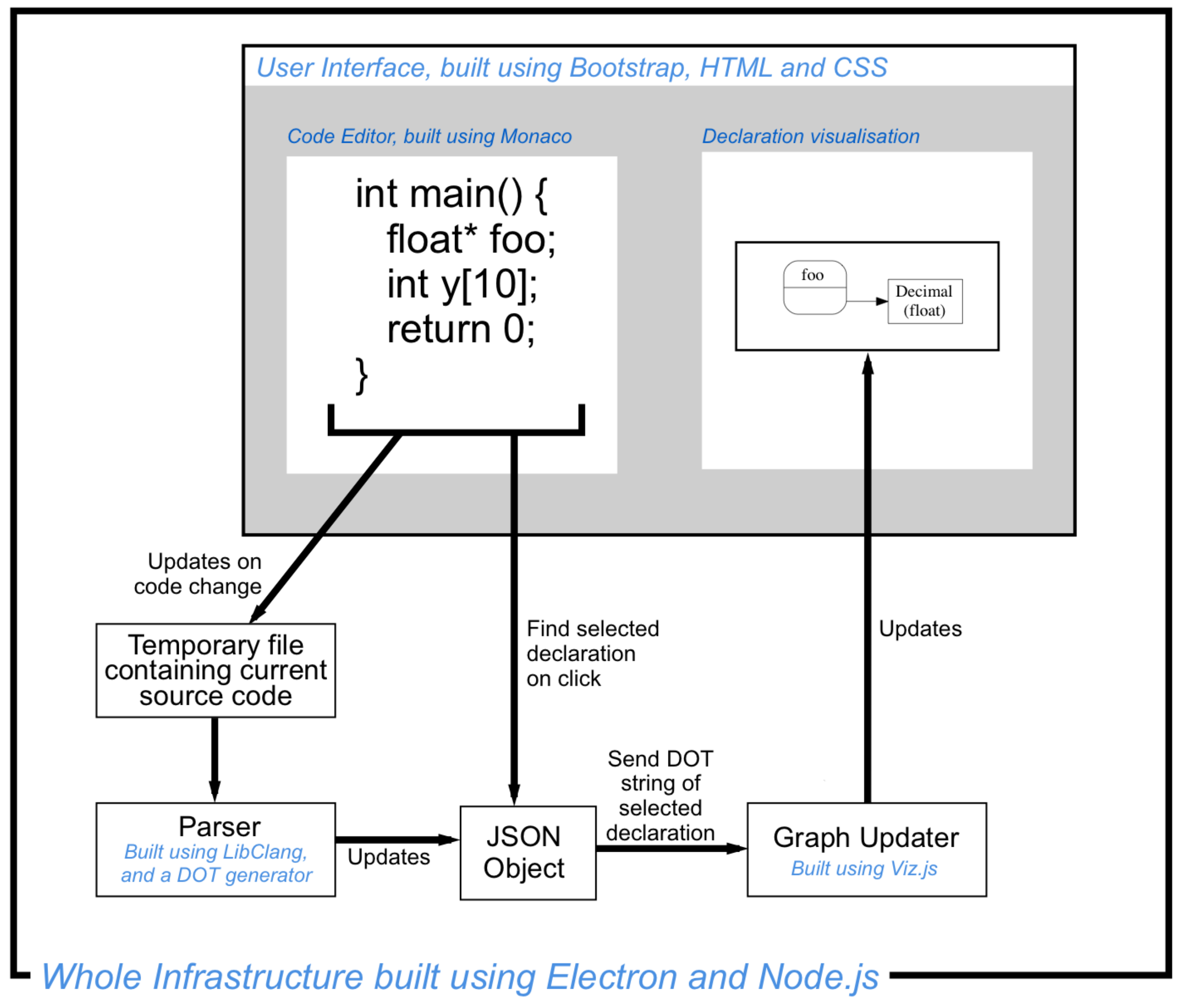
BACKEND

Figure 1 shows an overview of the backend infrastructure. The app was built using Node.js and Electron. Node.js is a JavaScript environment that gives access to 475,000 pre-written modules - these provided important modules in the backend (npm Inc., n.d.). Electron is a wrapper that creates desktop applications from Node.js.

The code editor was built using Monaco - the open source editor behind Visual Studio Code.

When the user changed the code, it was parsed using LibClang – a high level interface to the C-family compiler, Clang (The Clang Team, n.d.).

The parser created files for GraphViz - open source graph visualisation software (Bilgin et al., n.d.). When a declaration was selected, this GraphViz file was outputted as a visualisation.



```
int main() {
   float* foo;
   int y[10];
   return 0;
}
```

User Interface, built using Bootstrap, HTML and CSS
Code Editor, built using Monaco
Declaration visualisation
foo
Decimal (float)
Updates on code change
Temporary file containing current source code
Find selected declaration on click
Updates
Parser
Built using LibClang, and a DOT generator
Updates
JSON Object
Send DOT string of selected declaration
Graph Updater
Built using Viz.js
Whole Infrastructure built using Electron and Node.js

Figure 1: Overview of the backend

VISUALISATION

Initially, prototype drawings of the visualisations were made. They were recursive, to allow for arbitrary visualisation complexity, and had different distinct elements.

After testing these drawings with two users, problem areas were identified and corrected. These included changing the direction of the overall graph, and changing the portrayal of the const keyword.

The final designs were created in GraphViz, since it has easy to construct descriptions as inputs, and lots of functionality in visualisation (Bilgin et al., n.d.). There is delibrate distinction between pointers (as black arrows) and function inputs/outputs (as grey dotted lines), as seen in Figure 2.

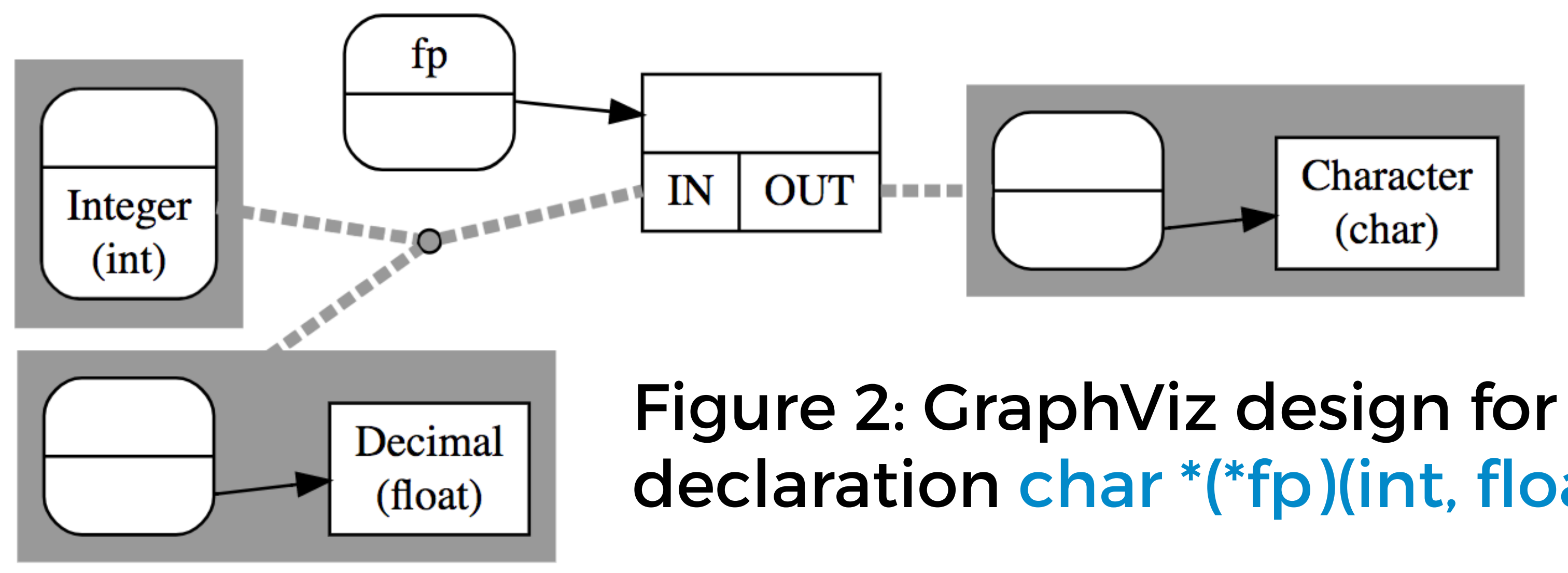


Figure 2: GraphViz design for the declaration char *(*fp)(int, float*)

TESTING

Testing the software with users was undertaken throughout the process. At the end of the project, three users were tested in depth, and both qualitative and quantitative data sets were created.

With the qualitative data, all three users agreed that the visualisations of this project were more helpful than a word-based competitor. In addition, the testers highlighted some issues in the software.

With the quantitative data, the time taken by each user to understand a declaration was measured; either with no software, with a word-based competitor, or with visualisations from this project.

The results in Figure 3 show that both pieces of software are better than the control. The times were similar for both, so this testing does not conclude which software is better.

	Beginner 1	Beginner 2	Advanced 1	Mean
No software (Median)	23.00	18.00	15.50	18.83
Competitor software (Median)	12.00	10.00	9.00	10.33
This Project (Median)	13.00	8.50	10.50	10.67

Figure 3: Graph of the median time for each user, with each method

REFERENCES

Bilgin, Arif et al. (n.d.). Welcome to Graphviz. URL: graphviz.org
Anderson, David (1994). The Clockwise/Spiral Rule. URL: c-faq.com/decl/spiral. anderson.html
The Clang Team (n.d.). Clang 5 documentation. URL: clang.llvm.org/docs/tooling.html
npm, Inc. (n.d.). Node Package Manager. URL: npmjs.com