Dynamic Data Structure Visualisation [1]
IMAT3451 – Final Report

Christian Manning – p0928544x
De Montfort University
Computer Science

April 2012

[1]With thanks to De Montfort University

Abstract

This project aims to create an application which provides a graphical user interface containing a dynamic visualisation of variables, pointers and structures for the purpose of teaching these concepts to beginner programmers. To accomplish this it has provided an interpreter for a language designed specifically for this project as a subset of the C programming language, with some intentional omissions and additions. The result is a cross-platform application which simply visualises, and allows modification of, stack data from the interpreter, showing things such as the link between a pointer and its intended target memory address, primitive variables and `struct` variables. The project has been largely successful, though with some limitations on the range of data structures able to be defined, specifically a lack of recursive data structures. As such, a more accurate or appropriate title for this project may be "Dynamic Pointer and Variable Visualisation".

Contents

List of Figures

1 Introduction

1.1 Motivation

For any programmer, a reference is a fundamental concept, and is employed in many forms by programming languages. A reference is a data type whose value is referential to another data type at another location. A common implementation of a reference is a *pointer*. A pointer's value is simply the memory address of another piece of data, of which the location may not be known by any other means, and also may itself be a pointer. To access the data value being pointed to, an operation known as *dereferencing* must be performed on the pointer. This is similar to a street address, which refers to a particular house; a pointer's value would be the street address and the house a specific location in memory. When dereferenced, the house is then free to have tasks performed upon it.

While this may seem a simple concept, it is often asked *why* these data types are used, and not always access data as value-typed variables. This is a valid question that may be asked by many students not necessarily due to a lack of understanding, but a lack of examples of uses of pointers in programs. Figure 1 shows a common introductory example of pointers, which performs a swap operation on two `int` variables. It demonstrates how the value semantics of a simple `int` data type are *copied* into the parameters of the swap function, so that the operation isn't performed as expected, i.e. the copies are modified, not the originals. The second example shows pointer types as parameters for the function, allowing it to perform as expected by dereferencing the pointer variables so that it modifies the original variables.

```c
//Without pointers
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
//With pointers
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Figure 1: C swap function

This may be a potential use for pointers, however, it is perhaps overly simple and doesn't demonstrate the power of pointers with their main use case: data structures. Through the use of dynamic memory allocation, data does not need to be declared with a name, merely to a pointer variable. What this means is that allocations will occur when initiated directly or indirectly by the programmer. The amount of memory to be allocated is determined at runtime, making it a dynamic process. This is an important technique to avoid the copying of data, like in figure 1, which not only make it harder to modify data, but excessive memory allocations are also costly. Dynamic memory becomes useful in data structures whose memory requirements are determined by what data they are to store, how many pieces of data and in what order.

```
struct node {
  double data;
  node* next;
};
node* cur = malloc(sizeof(node));
cur->next = cur;
void insert(double x) {
  node* tmp = malloc(sizeof(node));
  tmp->data = x;
  tmp->next = cur->next;
  cur->next = tmp;
}
void next() {
  cur->next = cur->next->next;
}
```

Figure 2: C/C++ dynamic memory allocation with a singly linked list

One such data structure is a linked list (figure 2). When a new piece of data is to be entered into the list, the `insert` function allocates the necessary amount of memory and adds a pointer to this memory to the list. The `next` function will move through the list. This is an entirely dynamic process that can be triggered from user interaction (or otherwise), so the memory required by the application increases as needed.

A singly linked list is a simple example, but the principle remains the same in other data structures. The problem that motivates this project is the difficulty that comes from trying to visualise these structures, especially dynamically, for the purpose of teaching, or even just to gain insight into the layout of a particular data structure.

1.2 Aims and Objectives

This project aims to create an application which will provide a dynamic visualisation of pointers and, by extension, data structures. The means to achieve this are outlined in the following objectives.

- Gather requirements to produce a requirements specification.
- Devise a plan for the project.
- Learn about the chosen tools to be used by creating some basic examples.
- Design and create graphics providing an abstract representation of memory objects.
- Create a user interface for the purpose of displaying and manipulating the aforementioned graphics.

- Learn about programming language grammars and from this design a small programming language.
- Learn about compiler and interpreter concepts and implement the mentioned language as an interactive interpreter.
- Integrate the user interface with this interpreter such that it can interact with the interpreter to manipulate its data.
- Design and execute a suitable test plan.

2 Related Work

The literature review undertaken as part of the initial hand-in for this project contained some research into several areas relevant to this project. Through this process it was discovered that there exists some research into similar goals, some proposing software products attempting to achieve them. It should be mentioned that none of these applications are providing a pointer visualisation as this project aims, merely algorithm animation/visualisation. An early example of this is Balsa (M. Brown and Sedgewick, 1984) and its successor Balsa-II (M. Brown, 1988). These applications offered little in the way of interactivity as they were aimed more toward instructors providing a set animation of an algorithm by writing some specialised code in the Pascal language, with added annotations. It is also possible to create "scripts" which record a users actions during a session for playback at another time.

The Balsa applications were succeeded by Zeus (M. Brown, 1991) which introduced some new features such as the ability to step through an algorithm, stopping and starting it along the way. This was a significant increase in interactivity over Balsa and Balsa-II, though it still required the algorithm animations to be written specifically for the application and had little in interactivity with regard to the data. Another application in the vein of Balsa, Balsa-II and Zeus is Swan (Yang et al., 1996). Swan differs from the others in that it takes C and C++ code defining the data structures and algorithms which is then annotated to define the animations. This is unlike the previously mentioned solutions which require code to be specifically written for visualising.

The researched solution found to be most like the goals set out by this project is proposed by Chen and Sobh (2001). This tool provides a Java-like language for defining data structures and algorithms using some predefined models as the building blocks. This is different from the goals of this project in that the building blocks for the data structures are structs, pointers and variables, so everything is built from the ground up. Chen and Sobh's tools also lacks interactivity, like the previous examples, as each data structure or algorithm has to be translated into Java then compiled so that it can be run.

While all of the mention software products have merit, they generally lack interactivity and dynamism, instead focussing on defined animations. This is the void the Dynamic Data Structure Visualisation project intends to fill.

3 Design Rationale

Since the initial design document several decisions have been modified or excluded for the final application design. The final design and usage is located in the user documentation (refer to appendix A), and the key features are discussed below.

The original design focused on the manipulation of graphics of variables and pointers. Throughout the project, this focus has shifted more towards the interpreter. What this means is that the graphics have become mere abstract representations of the underlying data structures of the interpreter. This approach is potentially very powerful, but an interpreter is a very complex system, even for a small language such as that designed for this project (formal specification in appendix A). As a result of this complexity, some functionality intended to be included with the user interface has been excluded from the implementation.

Two such excluded features are queueing and a history, whose presence was determined to be non-essential for the original aims to be achieved. Queueing is a feature which can mostly be emulated using external applications, for example by using a text editor to keep code statements which can be moved to the application when they are required to be executed. This however, is not a solution to the problem, but a workaround should the user want this functionality. The history feature would have been closely related to the queue, but with the added ability for the user to step back through previous operations. While this would have enabled an increase in usability, it also was deemed non-essential for the project to be a success.

Another feature that has not been implemented is the ability to show the graphical elements in a particular order, or a layout. A layout would have enabled the user to view their data structures in a particular way, such that the visual representation is similar to what is taught in traditional means, for example, a binary tree is drawn top to bottom with a node's children below it. This feature could have been a big boost in the project's educational utility, though it is not a core feature.

Despite these features being removed, the design for this project is suitable for its intended aim to be achieved: to provide a means of visualising variables and pointers for the purpose of teaching.

3.1 Interpreter

A common early language taught to or learnt by students who are learning programming is C. C is quite a simple language which provides lower levels of abstraction than modern languages, making it an important tool in the learning of key concepts that would otherwise be hidden from the user. One of these concepts is the pointer.

By creating data structures in C, the memory layout created is often not dissimilar from the code written, but how is this *seen*? An interpreter for C with a visualisation interface is an approach utilised by this project. This means that for code written by the user, a graphical representation will displayed, showing variables, pointers, the data of the former and the destination of the latter. What this would allow the user to do is to write their data structures in C or a C-like language, then directly visualise how this will appear in memory.

```c
//pointer to int
int* a;
int b = 123;
//assign address of b
a = &b;
//dereference and assign
*a = 321;
```

Figure 3: C pointers

An alternative to using an interpreter could be to take a similar approach to a debugger. A debugger uses information provided by a compiler to step through a program, step by step. Visual debuggers such as those incorporated into an integrated development environment (IDE) such as Visual Studio, Eclipse or Qt Creator, can show the data used by the program at each step. This data could be manipulated and displayed graphically, although this approach would require that only data needed to be displayed be made known else a large amount of unnecessary data will be shown, displacing the necessary. To accomplish this, either executables will need to be modified or create a means to annotate code such that only needed code will be able to be debugged. Modifying executables is a dangerous process with potentially drastic effects, for example, corrupting data. Enforcing the annotation of code can be beneficial in that very little will need to change from the data structure implementation. However, this method would need significant work in

Before:

Address	**Variable**	**Value**
0x00	`int* a`	0x01
0x01	`int b`	123

After:

0x00	`int* a`	0x01
0x01	`int b`	321

Figure 4: Stack showing the results of figure 3

the areas of executable file formats and debugging formats, these both being different on each platform, and even compilers; this is beyond the scope of a final year project.

An advantage of the interpreter approach is that programming exercises can become better understood by students due to being able to see the effects of these structures, as defined in code, and their operations as soon as they are entered. Often these structures are not simple to draw or generally visualise, certainly when operations are performed on them as they will need to be redrawn or rearranged to accommodate for the changes, which are tedious tasks and may not be clear to students.

The main disadvantage of this feature is the complex theory and extensive implementation required. A compiler or interpreter is a very elaborate piece of software comprised of several parts, themselves being complicated. Not only this, but it also needs to be accessible through the use of a graphics user interface, requiring strong integration. However, this is still a feasible objective that can be accomplished within the bounds of this project.

4 Development Tools Rationale

4.1 C++

The C++ programming language was chosen for this project because of its wealth of resources available: libraries, frameworks and learning materials can all be freely acquired. C++ compilers are very widespread and available for all of the target platforms. Through the use of the C++ standard library, standard template library (STL) and certain libraries and frameworks, the resulting application will be as portable as possible.

Features from the latest C++ standard, C++11 (ISO, 2012), have been used where appropriate, which has been very advantageous with regard to code readability and maintainability. The most useful things that C++11 brings are type inference, initialiser lists and the range-based for-loop, the latter only previously being available through a library solution, and the others not available at all in C++.

4.2 Boost

Boost (*Boost C++ Libraries* 2012) is a collection of high quality, peer-reviewed and portable libraries providing many different features and functionality, contributed to by a variety of people. The main purpose of using boost in this project is for the spirit library. Spirit is a library which, among other features, exposes a Domain-Specific Embedded Language (DSEL) for defining a language grammar. This DSEL is an almost exact replica of Extended Backus–Naur Form (EBNF), a common notation for expressing context-free grammars, though confined within the limitations of C++ operators. This is accomplished using a metaprogramming technique known as expression templates and results in a recursive-descent parser (Davie and Morrison, 1982).

A common approach to creating an interpreter or compiler is to use an external parser generator such as the popular Bison(or Yacc)/lex (Mason and D. Brown, 1990; Lesk and Schmidt, 1975; Johnson, 1978) combination. The downsides to this approach are for one, requiring an extra step in the build process, but also only providing a C interface. The extra build step is for generating C code from the grammar defined in EBNF, and would be acceptable if it were generating C++. With the generated C code, a C++ interface would have to be grafted on top so as to integrate with the rest of the application. C and C++ code, while compatible, are not interchangeable, so what may be idiomatic C is not idiomatic C++ and does not integrate well with either the standard library or other libraries such as Boost or Qt.

Boost Spirit requires only a single compilation stage and provides a modern C++ interface which has made integration with other C++ code a non-issue. Though it too, has its negatives. A problem with heavy use of metaprogramming is the memory usage, executable size and time taken during compilation, exceeding 6GB in some instances of compiling this project. Unfortunately the solution to this problem is to disable debugging information produced by the compiler, which is not satisfactory during development. The release build does not suffer from this problem though, where debugging information is kept to a minimum.

C++ compiler template errors are also somewhat of a burden on development as they are often difficult to decipher, and are produced in great quantities as a result of the slightest error in a Boost Spirit grammar. This has resulted in a vast amount of time being devoted to tracking down these errors. Microsoft's Visual Studio C++ compiler outputs some slightly more human-readable errors, though this compiler does not support platforms other than Microsoft Windows, and GNU/Linux is the primary development platform.

Other Boost libraries have been used extensively as many integrate directly with Spirit, when constructing the abstract syntax tree (AST) in particular, which uses Boost Variant. A variant, also known as a tagged or discriminated union, is a data structure that can store any one of a predefined set of types. This is used by Spirit directly when using the *or* operator (|) in the grammar definition. Variant is a very convenient and type-safe way to represent attributes created from rules where this operator is used, and is just as easily processed during the semantic analysis phase of compilation.

Overall the usage of Boost in this project has been of great benefit. The expressiveness of Boost Spirit's embedded language vastly improved understanding of the code so that when changes needed to be made, the implications were clear on the rest of the code base. It has helped to reduce code complexity, with the only major disadvantage being significantly increased compile times, which is not a major issue. The portability of the Boost libraries has also been helpful as it has enabled the same code to compile and run on multiple platforms with minimal, if any, changes.

4.3 Qt

Qt (Nokia, 2012) is a cross-platform application development framework, mostly used as a user-friendly way of creating a graphical user interface (GUI) that looks native on all supported platforms. This is made possible by utilising each platforms' native drawing facilities and emulating their look and feel, making Qt a fine choice for cross-platform development. In combination with Boost, Qt has made cross-platform development very simple, especially in its speciality area of user interface building, a feat that is often very difficult to achieve without sacrificing desktop integration.

Along with Qt, Nokia also provides the integrated development environment (IDE) Qt Creator. This IDE provides first class support for C++ and Qt, with a graphical form designer interface for UI design and development. A major benefit of using this IDE during development of this project is due to the fact that it is based around the Qt framework: it works exactly the same on every supported platform. This has made cross-platform development a very easy task that would otherwise be a chore.

4.4 Git

Git is a distributed source code management (SCM) system with revision control capabilities. The usage of this tool has allowed the development of this project to be tracked, version by version. This became useful during development when a regression was discovered while testing changes, as the exact revision where this regression occurred can be identified. Each revision in git is called a *commit*, with each one carrying a message summarising its changes. When a regression is found, these messages are useful for showing the intent of the original commit, which can then be fixed appropriately.

The log of commits produced by git for this project can be found in appendix D, provided as evidence of coding progress.

5 Implementation Rationale

As discussed in the initial design documentation this projects takes an approach similar to a model-view-controller design pattern, though without a distinct controller. What follows is an overview of the implemented application.

The model in the case of this project is contained within the interpreter and consists of the program stack, variable table and struct table. The only way to modify this model is through the interpreter, by passing it a string of code. This string is then parsed by the Boost Spirit parser, which produces a variety of data structures, dependant on its input, which form an abstract syntax tree (AST). The AST is heavily reliant on Boost Variant, described above, to allow the AST to differentiate between different structures that can each be valid at a particular time, but not all at once. This AST then processed to create entries in the variable table for newly created variables, in the struct table for newly defined struct types and to evaluate and verify the input, with some basic type checking. During this processing, known as semantic analysis (Aho et al., 2007), an intermediate representation is generated from certain codes, one for each kind of operation (i.e. add, subtract, load, store, etc.), known as "op codes". Once this has been completed for all input, these op codes can then be passed on to the virtual machine (VM). The VM takes the op codes and modifies the stack based on their contents. The stack, implemented as a "vector" or dynamic array, is where the values of variables are stored and can be seen as the actual program data. The stack data is then used to update the visualisation.

Should an error occur during either stage it is displayed using a dialog window, with some information on what went wrong and where in the input. This is accomplished by "tagging" each successfully parsed rule's resultant data structure. This tag, along with the position in the input string is stored in the error handler, so that this position can be accessed by using the tag as its identifier. The error reporting provided by this system is a great help as it has the ability to tell where in the input code the error is and point out what is wrong with it, as best it can.

When a new struct is defined by the user, the user interface is notified of this change through use of a construct know as signals and slots. Signal and slots is provided by Qt and implements the observer design pattern (Blanchette and Summerfield, 2008). This enables, through very little code, an object to notify other objects of changes so that they can make the necessary adjustments. In this case that means adding the new struct and its members to the struct tree widget so that a user can view all defined structs and their member names and types.

Also implemented is dynamic memory through the use of the `new` operator. Dynamic memory is used with pointers to reserve a space on the stack for a particular type. This is analogous to the same keyword in C++ or a combination of `malloc` and `sizeof` in C.

Each variable, or allocated memory space, contained in the stack is displayed on screen along with its value and type. This is updated after every interpreter run. Pointers are displayed just as variables, but with a line drawn to the variable that it points to. Dynamically allocated memory is displayed much like a normal variable, except that their name is generated from their type and stack address as they are unnamed variables, referred to only by pointers.

The implementation started out from an example included with Boost Spirit, named "conjure", which demonstrated a compiler for a very small subset of the C programming language, smaller than that of this project. Because of its limitations this example has had to be almost entirely rewritten to support more advanced language features in the parser, semantic analysis and code generation stages, such as pointers, structs, dynamic memory allocations and a type system, increasing the code base to twice its size. While this was a long process, it has allowed for a hands-on approach to learning about a compiler's internals, which is a very complex system. If this was written from scratch it would not have been feasible for a final year project as the time and knowledge needed would be much longer and wider, respectively.

The language specification provided in the user documentation (appendix A) is an amalgamation and simplification of a variety of sources (**Kernighan:1988:CPL:576122**; Sitaker, 1999; Degenern, 2012). These sources are known to be correct specifications of the grammar of the C programming language. Using Sitaker, 1999 as the predominant source, the grammar was translated into Boost Spirit, which was an almost direct translation. This particular source was chosen because of not only being in Backus–Naur Form (BNF), but being a translation of the specification provided by **Kernighan:1988:CPL:576122** which is the American National Standards Institute (ANSI) C language specification. The reason for using sources such as these was to ensure the validity of the grammar and to support the more advanced features not provided by the "conjure" example in a known correct way.

There are also some notable limitations to the implementation, which are areas of potential improvements to the application to better achieve the goals set at the beginning of the project. One such flaw is the inability to define recursive structures, a cornerstone of data structures. A recursive data structure is one which contains an instance of or a reference to an item of the same type. This ability can be added with some work to the semantic analysis phase, without affecting other areas, but still is not present at the time of writing. A similar yet distinct limitation is that the graphical visualisation does not represent a pointer with a line when the pointer is a member of a structure type. Again this can be implemented in the graphics code without affecting other areas and is also a candidate for further improvement of the application.

Functions are another part of the interpreter that is missing, which are a very useful concept in programming languages. In the context of this project, functions could have been a useful tool aiding in automation and the reduction of code needed to operate on data structures. However, with the limitation of non-recursive data structures only, the lack of functions is not as sought after as they otherwise would. The implementation of functions will require much work to the semantic analysis phase of the interpreter as well as some possible modifications to internal data structures and the virtual machine. This extension in functionality would be of great benefit to the application should recursive data structures also get implemented.

Another area of the final application that could be improved is the visualisation as a whole. It excels is displaying the necessary information to the user, but perhaps does not accomplish this in the most aesthetically pleasing of ways. An advancement in this could improve usability by making the displayed information clearer and more accessible. The visualisation could also use some other usability enhancements such as context menus on the graphical items that provide a means to perform operations in a more convenient way.

6 Plan

The plan for this project has had to be revised as the project went forward. As noted in the previous section, an interpreter is an incredibly complex system that involved extensive studying of compiler internals from material such as Aho et al., 2007.

When the plan for this project was devised, the time needed for the implementation was severely underestimated in several regards. Firstly the language grammar design had to go through several iterations until it provided all the needed features in an unambiguous fashion, so that certain grammar rules were not arbitrarily chosen over the intended targets and thus fail to match. This was an ongoing process throughout the implementation of the interpreter, which may have resulted in a better language overall, but the time taken has resulted in the loss of several desirable features as discussed in the previous section.

Due to this significant underestimation during the original planning, user testing was unable to be scheduled because of a lack of time, potentially making the testing of the application inadequate. Despite this, normal testing procedures have been undertaken which is discussed in the next section.

Other than the removal of user testing, the plan has only been slightly modified by extending the time-scales of the implementation stages and the slight delaying of the writing of this report.

6.1 Methodology

There are many software development methodologies, each catering to their own environment, whether a large team in a corporate setting, a small independent team or an individual. Due to its individualism, this project has limited options with regard to development methodologies. Those appropriate are outlined below with the final choice described.

The waterfall model is a sequential development methodology where each of its phases must be completed in order to continue. These phases are: system requirements, software requirements, analysis, design, coding, testing, and operations and maintenance (Royce, 1970). Although the model is often criticised as flawed and prone to failure, it is in widespread use among software development projects in industry and governments (Laplante and Neill, 2004). The reason for this criticism is aimed at its requirements to produce formal documentation to signify the end of each phase, when it is often the case that several phases have the need for some overlap or even iterations (Boehm, 1988). This rigidity can be the downfall of many development projects as a problem identified late in the process (i.e. in the testing phase) can derail a project entirely. Because of this many modifications are generally made to the model such as multiple passes, or allowing for backtracking. Another problem with a sequential model is that early processes, such as a fully detailed design, may not be able to be fully realised until later on in the life cycle. The features of the waterfall model such as its sequential nature and lack of flexibility make it unsuitable for this project due to the multiple hand-ins, and high risk of failure if a defect is found too late.

An alternative methodology to the waterfall is the V-model (Graham et al., 2008). The V-model is similar to the waterfall model, and is sometimes considered an extension to it. Its main difference is its focus on testing, which happens throughout the development cycle rather than near the end as with the waterfall model. There are four stages of testing which take place during the V-model: component testing, integration testing, system testing and acceptance testing. Component testing involves the testing of specific parts of the system that are able to be separately tested. This could things such as functions or classes. Integration testing tests how well each of the components interface with one another. System testing tests the software product as a whole, verifying against the requirements. The last kind of testing employed by the V-model is acceptance testing, otherwise known as user testing. User testing is used to validate the requirements of the system, which if the previous testing phase was successful, the software should meet. The V-model, despite being more flexible than the waterfall model, is still sequential in nature, making the prototyping required by this project difficult, if not impossible.

What's needed for this project is a methodology following an iterative and incremental life cycle. There are many iterative and incremental development methodologies that have been used throughout the software development world through various time periods (Larman and Basili, 2003). An iterative development model follows several compartmented stages, each resulting in a particular software component. Each of these stages have their own life cycle which can take a number of forms. This is largely what separates the different iterative models.

6.2 Life Cycle

The actual development methodology constituting the life cycle of this project could be considered an iterative and incremental version of the waterfall model. This is because of the structure and deliverables expected of a final year project. A final year project has two deadlines, the initial hand-in and final. This suggests that the life cycle must take this into account with two "releases" or iterations. The first release is required to produce many documents that describe the design and plan the project throughout the year. This is a similar approach to what happens at the start of a waterfall life cycle, though these documents are able to be modified in this case.

A look at the plan (appendix F) will demonstrate its incremental nature with some of the tasks directly showing different stages of an iteration.

7 Testing

7.1 Verification

Verification testing is the process of determining whether a system meets the requirements set out nearer the beginning of a project. This has been accomplished in this project through a set of test cases that were initially defined for the first hand-in. These test cases have however, been modified to account for changes in design and therefore specification. This modified test plan can be found in appendix B.

The plan consists of two separate parts, the user interface testing and interpreter testing. These set of tests together form the system testing, as the tight integration of the two parts makes it difficult to test them separately, so the separation outlined in the test cases is merely a convenience. The tests were designed to verify that the planned objectives have been achieved to the standard outlined by the specification and design documents.

The results of these test can be found in the Testing Documentation (appendix C), which shows that the final application passes all of the devised test cases. This means that the software should conform to the specifications and designs. The test results are also an ideal way to show the abilities of the application, such as the features of pointers and structs.

7.2 Validation

Validation testing tests the appropriateness of the specification such that the objectives of the project have been achieved to satisfactory degree. User testing is a potential method of validation, however as mentioned above, it was not possible to carry this out due to time constraints. However, the question of validity can still be answered objectively, with respect to the aims and objectives as outlined near the beginning of this document and in the project contract.

The final result of the project matches all of the criteria outlined in the aims and the objectives have all been met, with an application as a result of the work towards these. How well these criteria have been met is not a question that can be answered without external user involvement through user testing, the lack of which being a flaw in this project's execution.

8 Conclusions

The Dynamic Data Structure Visualisation project had lofty goals from the start, so it may be of no surprise that some of the more complex aspects were given priority over those less so. The interpreter and associated language account for around 75% of the application code, with more integrating the interpreter with the user interface. This is a significant portion of a reasonably large code base, totalling 4300 lines of un-preprocessed C++ and Qt code.

Because of the complexity of this substantial segment of the code base, some features were removed, such as functions, and some were just not able to be completed on time, such as recursive data structures. The lack of the ability to define recursive data structures has hindered hindered the project somewhat as they are a key aspect of data structures, so being unable to define these has led some uncertainty as to whether this project meets its aim. The aim was to provide a visualisation for pointers and data structures, the former of which has been met, but the latter is less clear. It could be seen that a data structure is simply a structure which holds data, in which case the aim has been met as `struct`s are able to be defined and used as expected, bar recursion. However, many data structures used in real-world applications use recursion as their basis and so these generally cannot be defined using this project's application.

Despite this, the resultant application serves the purpose of demonstrating pointers and their effects, which is the more fundamental concept, being vital in many aspects of software development, and should therefore not be seen as anything but a success.

8.1 Further Work

The are several areas of the application the could use some further work, as mentioned already, which would increase its viability as a teaching tool. Should this project continue, first and foremost, the feature that needs to be implemented is recursive data structure definitions, allowing the most common data structures to be visualised as normal `struct`s, variables and pointers are currently. Another important potential improvement lies in the visualisation, as currently even if recursive data structures are implemented in the interpreter, they will not be displayed correctly in the visualisation. Many other improvements could be made to the application including better graphical representation of data in the visualisation and general user interface improvements to improve presentation and usability.

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman (2007). *Compilers: Principles, Techniques, and Tools.* 2nd. Boston, MA, USA: Addison-Wesley. ISBN: 0-321-49169-6.

Blanchette, Jasmin and Mark Summerfield (2008). *C++ GUI Programming with Qt 4.* 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 9780137143979.

Boehm, B. W. (May 1988). "A spiral model of software development and enhancement". In: *Computer* 21.5, pp. 61–72. ISSN: 0018-9162. DOI: 10.1109/2.59.

Boost C++ Libraries (2012). URL: http://boost.org (visited on 04/22/2012).

Brown, M.H. (May 1988). "Exploring algorithms using Balsa-II". In: *Computer* 21.5, pp. 14–36. ISSN: 0018-9162. DOI: 10.1109/2.56.

Brown, M.H. (Oct. 1991). “Zeus: a system for algorithm animation and multi-view editing”. In: *Visual Languages, 1991., Proceedings. 1991 IEEE Workshop on*, pp. 4–9. DOI: 10.1109/WVL.1991.238857.

Brown, M.H. and Robert Sedgewick (Jan. 1984). “A system for algorithm animation”. In: *SIGGRAPH Comput. Graph.* 18 (3), pp. 177–186. ISSN: 0097-8930. DOI: 10.1145/964965.808596.

Chen, Tao and T. Sobh (2001). “A tool for data structure visualization and user-defined algorithm animation”. In: *Frontiers in Education Conference, 2001. 31st Annual.* Vol. 1, pages. DOI: 10.1109/FIE.2001.963845.

Davie, J. T. and R. Morrison (1982). *Recursive Descent Compiling.* John Wiley & Sons. ISBN: 0470273615.

Degenern, Jutta (2012). *ANSI C Grammar (Yacc).* URL: http://www.quut.com/c/ANSI-C-grammar-y.html (visited on 04/22/2012).

Graham, Dorothy et al. (2008). *Foundations of Software Testing: ISTQB Certification.* Intl Thomson Business Pr. ISBN: 9781844809899.

ISO (Feb. 28, 2012). *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

Johnson, S.C. (1978). *Yacc: yet another compiler-compiler.* Computing science technical report. Bell Laboratories.

Laplante, Phillip A. and Colin J. Neill (Feb. 2004). “The Demise of the Waterfall Model Is Imminent”. In: *Queue* 1.10, pp. 10–15. ISSN: 1542-7730. DOI: 10.1145/971564.971573.

Larman, C. and V.R. Basili (June 2003). “Iterative and incremental developments. a brief history”. In: *Computer* 36.6, pp. 47–56. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1204375.

Lesk, M. E. and E. Schmidt (July 1975). *Lex - A Lexical Analyzer Generator.* Tech. rep. Bell Laboratories.

Mason, Tony and Doug Brown (1990). *Lex & yacc.* Sebastopol, CA, USA: O’Reilly & Associates, Inc. ISBN: 0-937175-49-8.

Nokia (2012). *Qt.* URL: http://qt.nokia.com (visited on 04/22/2012).

Royce, W W (1970). “Managing the development of large software systems”. In: *Electronics* 26.August. Ed. by Sterling MEditor McMurrin, pp. 1–9.

Sitaker, Kragen (Oct. 1999). *C BNF Grammar.* URL: http://lists.canonical.org/pipermail/kragen-hacks/1999-October/000201.html (visited on 04/22/2012).

Yang, Jun, Clifford Shaffer, and Lenwood Heath (1996). "Swan: A data structure visualization system". In: *Graph Drawing*. Ed. by Franz Brandenburg. Vol. 1027. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 520–523. ISBN: 978-3-540-60723-6. DOI: 10.1007/BFb0021837.

Appendices

A User Documentation

IMAT3451 – Final Year Project
Dynamic Data Structure Visualisation
User Documentation

Christian Manning – p0928544x
De Montfort University

April 2012

Contents

List of Figures

1 Introduction

The Dynamic Data Structure Visualisation (DDSV) application aims to provide help to students who are learning to program. The focus of this application is on the concept of pointers; specifically, how they are used to build data structures.

It does this by utilising a graphical user interface (GUI) to provide a visualisation containing graphical representations of abstract data types and pointers to them. This facilitates the visual construction of a data structure, which can then be manipulated to simulate algorithmic operations.

Another interface to this application utilises the expressiveness of a programming language from the GUI in the form of an interpreter. The language accepted by this interpreter, is a small subset of the C programming language which has quite a low level model and so pointers are exposed to the user more readily. With this interpreter, C code can then be visualised graphically, allowing the user to learn

2 Interpreted Language

2.1 Specification

This is the formal grammar specification of the interpreted language using an Extended Backus-Naur Form (EBNF)-like notation, with some brief descriptions. Each entry is referred to as a grammar *rule*.

Key:

Symbols	**Description**
`<a>`	`a` is a non-terminal
`a` ::= `b`	`a` is defined as a rule with `b` as its singular constituent
`a b`	`a` followed by `b`
(`a b`)	Groups rules `a` and `b` together
`a`	Repeat `a` zero, one or more times
[`a`]	`a` is to be matched zero or one times. (Optional)
`a` $\mid$ `b`	`a` *or* `b`
`-a`	Do not match `a`
‘`c`’	Character string `c`
%`a`	Comma separated list of `a`
;	End of rule

assign_op ::= ‘=’ ;

logical_or_op ::= ‘||’ ;

logical_and_op ::= ‘&&’ ;

equality_op ::= ‘==’ | ‘! =’ ;

relational_op ::= ‘<’ | ‘≤’ | ‘>’ | ‘≥’ ;

additive_op ::= ‘+’ | ‘−’ ;

multiplicative_op ::= ‘∗’ | ‘/’ ;

Listed above are the *binary operators*, meaning they require two *operands.* These are arithmetic and boolean operators.

unary_op ::= ‘+’ | ‘−’ | ‘!’ | ‘∗’ | ‘&’ ;

Unary operators require only one *operand.*

struct_op ::= ‘− >’ | ‘.’

Struct operators are used to select members of a struct variable.

memory_op ::= ‘`new`’

The *new* operator is used for dynamically allocating memory.

keywords ::= ‘`true`’ | ‘`false`’ | ‘`if`’ | ‘`while`’ | ‘`struct`’ | ‘`return`’ | ‘`new`’ ;

These are the *keywords* which are reserved words and cannot be used outside of their required context.

types ::= ‘`void`’ | ‘`int`’ | ‘`bool`’ ;

These are the predefined primitive *types* that are used for declaring variables, etc.

$\langle$*identifier*$\rangle$ ::= -(keywords | types) , alpha | ‘_’ , { alpha | digit | ‘_’ } ;

An *identifier* must not be a *keyword*, a primitive type or begin with a digit. It must start with an alphabetic character or an underscore (_) to be optionally followed by zero or more alphabetic characters, digits or underscores.

$\langle$*assignment_expression*$\rangle$ ::= $\langle$*logical_OR_expression*$\rangle$ [$\langle$*unary_assign*$\rangle$];

$\langle$*allocation_expression*$\rangle$ ::= $\langle$*memory_op*$\rangle$ $\langle$*type_specifier*$\rangle$;

$\langle$*unary_assign*$\rangle$ ::= $\langle$*assign_op*$\rangle$
($\langle$*allocation_expression*$\rangle$
| $\langle$*logical_OR_expression*$\rangle$);

$\langle$*logical_OR_expression*$\rangle$ ::= $\langle$*logical_AND_expression*$\rangle$
{$\langle$*logical_or_op*$\rangle$ $\langle$*logical_AND_expression*$\rangle$};

$\langle$*logical_AND_expression*$\rangle$ ::= $\langle$*equality_expression*$\rangle$
{$\langle$*logical_and_op*$\rangle$ $\langle$*equality_expression*$\rangle$};

$\langle$*equality_expression*$\rangle$ ::= $\langle$*relational_expression*$\rangle$
{$\langle$*equality_op*$\rangle$ $\langle$*relational_expression*$\rangle$};

$\langle$*relational_expression*$\rangle$ ::= $\langle$*additive_expression*$\rangle$
{$\langle$*relational_op*$\rangle$ $\langle$*additive_expression*$\rangle$};

$\langle$*additive_expression*$\rangle$ ::= $\langle$*multiplicative_expression*$\rangle$
{$\langle$*additive_op*$\rangle$ $\langle$*multiplicative_expression*$\rangle$};

$\langle$*multiplicative_expression*$\rangle$ ::= $\langle$*unary_expression*$\rangle$
{$\langle$*multiplicative_op*$\rangle$ $\langle$*unary_expression*$\rangle$};

$\langle$*unary_expression*$\rangle$ ::= [$\langle$*unary_op*$\rangle$] $\langle$*postfix_expression*$\rangle$;

$\langle$*struct_expr*$\rangle$ ::= $\langle$*struct_op*$\rangle$ $\langle$*identifier*$\rangle$;

$\langle$*postfix_expression*$\rangle$ ::= $\langle$*primary_expression*$\rangle$
{$\langle$*struct_expr*$\rangle$
| $\langle$*postfix_op*$\rangle$} ;

⟨*primary_expression*⟩ ::= int
  | ⟨*identifier*⟩
  | bool
  | '(' ⟨*logical_OR_expression*⟩ ')' ;

Assignment expression is the catch-all expression rule. This rule uses recursion so that it also takes into account operator precedence without additional algorithms. Each of its constituent rules have a specific set of operators, each with their own precedence.

As can be seen from the top rule only one variable can be assigned at a time, whereas the rest can be composed of any of the others, allowing complex expressions to be formed. Available primitive types for constants and variables are integer and boolean only.

An *allocation expression* is one which allocates memory of a specified type to be assigned to a pointer.

⟨*type_specifier*⟩ ::= ⟨*types*⟩
  | ⟨*struct_specifier*⟩ ;

⟨*declarator*⟩ ::= ['`*`'] ⟨*identifier*⟩;

⟨*declaration*⟩ ::= ⟨*type_specifier*⟩ [⟨*init_declarator*⟩] '`;`';

⟨*init_declarator*⟩ ::= ⟨*declarator*⟩ ['`=`' ⟨*allocation_expression*⟩
  | ⟨*logical_OR_expression*⟩];

⟨*struct_member_declaration*⟩ ::= ⟨*type_specifier*⟩ ⟨*declarator*⟩ '`;`';

⟨*struct_specifier*⟩ ::= '`struct`' ⟨*identifier*⟩ ['`{`' {⟨*struct_member_declaration*⟩} '`}`';

The above lists the rule to match a `struct` definition, which may contain one or more member declarations. A struct member declaration must not be initialised in any form. Variable declarations may have a single initialisation from an expression. The optional '*' denotes a pointer type.

⟨*statement_list*⟩ ::= {⟨*statement*⟩} ;

⟨*statement*⟩ ::= ⟨*declaration*⟩
  | ⟨*assignment_expression*⟩ '`;`'
  | ⟨*if_statement*⟩
  | ⟨*while_statement*⟩
  | ⟨*return_statement*⟩
  | ⟨*compound_statement*⟩ ;

Any of the above listed *statements* can take the place of a $\langle statement \rangle$ instance.

$\langle if\_statement \rangle$ ::= '`if`' '`(`' $\langle logical\_OR\_expression \rangle$ '`)`' $\langle statement \rangle$;

An *if statement* is a conditional which will only execute its encompassed statement if a condition is evaluated as `true`. This condition can be anything that can be represented by a *logical or expression*, which is any expression other than an assignment.

$\langle while\_statement \rangle$ ::= '`while`' '(' $\langle logical\_OR\_expression \rangle$ '`)`' $\langle statement \rangle$;

The *while statement* is a looping statement which will execute its enclosed statement for as long as its conditional *logical or expression* evaluates to `true`.

$\langle compound\_statement \rangle$ ::= '{' [$\langle statement\_list \rangle$] '}' ;

A *compound statement* is an list of zero, one or many *statements* enclosed in braces ('{' & '}').

$\langle return\_statement \rangle$ ::= '`return`' [$\langle logical\_OR\_expression \rangle$] '`;`' ;

$\langle argument \rangle$ ::= $\langle type\_specifier \rangle$ $\langle init\_declarator \rangle$;

$\langle function\_definition \rangle$ ::= $\langle type\_specifier \rangle$ $\langle declarator \rangle$
'`(`' [%$\langle argument \rangle$] '`)`'
$\langle compound\_statement \rangle$;

$\langle translation\_unit \rangle$ ::= { $\langle statement \rangle$
| $\langle function\_definition \rangle$ } ;

The *translation unit* consists of a list of both *function* declarations and *statement lists.* This is root of all that can be passed to the interpreter, i.e. everything must be entered in this form.

2.2 Operator Precedence

The following table shows the operator precedence of the interpreted language, from lowest to highest.

Precedence	Operator	Description
1	$=$	Assignment
2	$\|$	Logical OR
3	&&	Logical AND
4	$==$	Equal
	$!=$	Not Equal
5	$<$	Less than
	$\leq$	Less than or equal to
	$>$	More than
	$\geq$	More than or equal to
6	$+$ (binary)	Addition
	$-$ (binary)	Subtraction
7	$*$	Multiplication
	$/$	Division
8	$+$ (unary)	Plus
	$-$ (unary)	Minus
	!	Not
	&	Address of
	$*$ (unary)	Dereference
9	$->$	Select element through pointer
	.	Select element

The `new` operator does not have a particular precedence as it can only be used in assignments and declarations, effectively giving it a precedence higher than that of '=', though it is considered a special operator.

2.3 Notes

This language may be similar to C but there are some notable exceptions and omissions:

- Only primitive types are `int` and `bool`
- No arrays.
- No increment, decrement or arithmetic assignment (`+=`, `*=`, etc.) operators.
- No `for` loop.

- No `else` to accompany an `if` statement. A similar effect could be accomplished by negating the condition of the `if` statement.
- No recursive struct data types.
- Functions are currently parsed but not processed.
- Struct variables occupy an additional stack entry at its beginning.

Should the parser fail (i.e. fail to match any of the above rules), an error message is displayed in a dialog. The information it provides will refer to the above rules informing of what rule was to be expected and also where it was expected (see figure 1). A similar

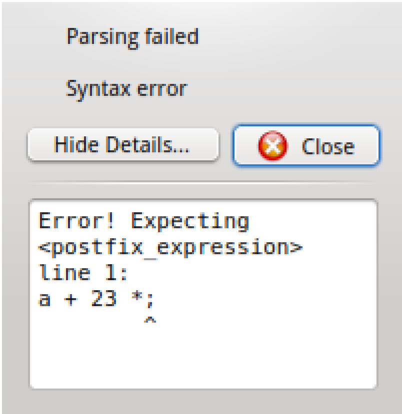


Figure 1: Syntax error.

error dialog will appear if an error occurs during processing after parsing (semantic analysis), though it attempts to be slightly more informative without needing to have the language specification handy.

2.4 Examples

2.4.1 Variable Declaration

```
//declare an integer variable named "a"
int a;
//assignment on declaration
int b = 3;
//assignment from an expression
int c = b + 2;
```

```
//declare a pointer and assign it the address of a
int * p = &a;
*p = 51; // a == 51

//declare a pointer variable and allocate it some memory
int * x = new int;
//dereference and assign
*x = 123;
```

2.4.2 Function Declaration

```
int factorial(int n) {
  if(n < 1)
    return 1;
  else
    return n * factorial(n-1);
}
```

2.4.3 Struct Declaration

```
//Define a struct
struct point {
  int x;
  int y;
};

//Define and declare a struct variable in one statement
struct point {
  int x;
  int y;
} p1;

struct point * ptr = new struct point;
ptr->x = 42;
ptr->y = 24;
```

2.4.4 Struct Instantiation

```
struct point p1;
p1.x = 4;
p1.y = 6;
```

2.4.5 If Statement

```
int a = 123;

// performs the statement(s) only if the
// expression enclosed in () evaluates to true
if(a < 200) {
  a = a * 2;
}
// a == 246
```

2.4.6 While Statement

```
int a = 123;

// performs the statement(s) repeatedly until the
// expression enclosed in () evaluates to false
while(a < 200) {
  a = a + 1;
}
// a == 200
```

3 User Interface

The user interface provides several ways to control and visualise program data. Figure 2 shows how the main UI appears when it is first opens, with nothing other than the basic UI elements displayed. The large empty space here is the visualisation area. This is where data items are displayed graphically once defined with the interpreter text input or added with the UI. In the upper-right of this screen-shot is a tabbed widget containing three tabs: Variables, Structs and Stack. These contain informational (read-only) data

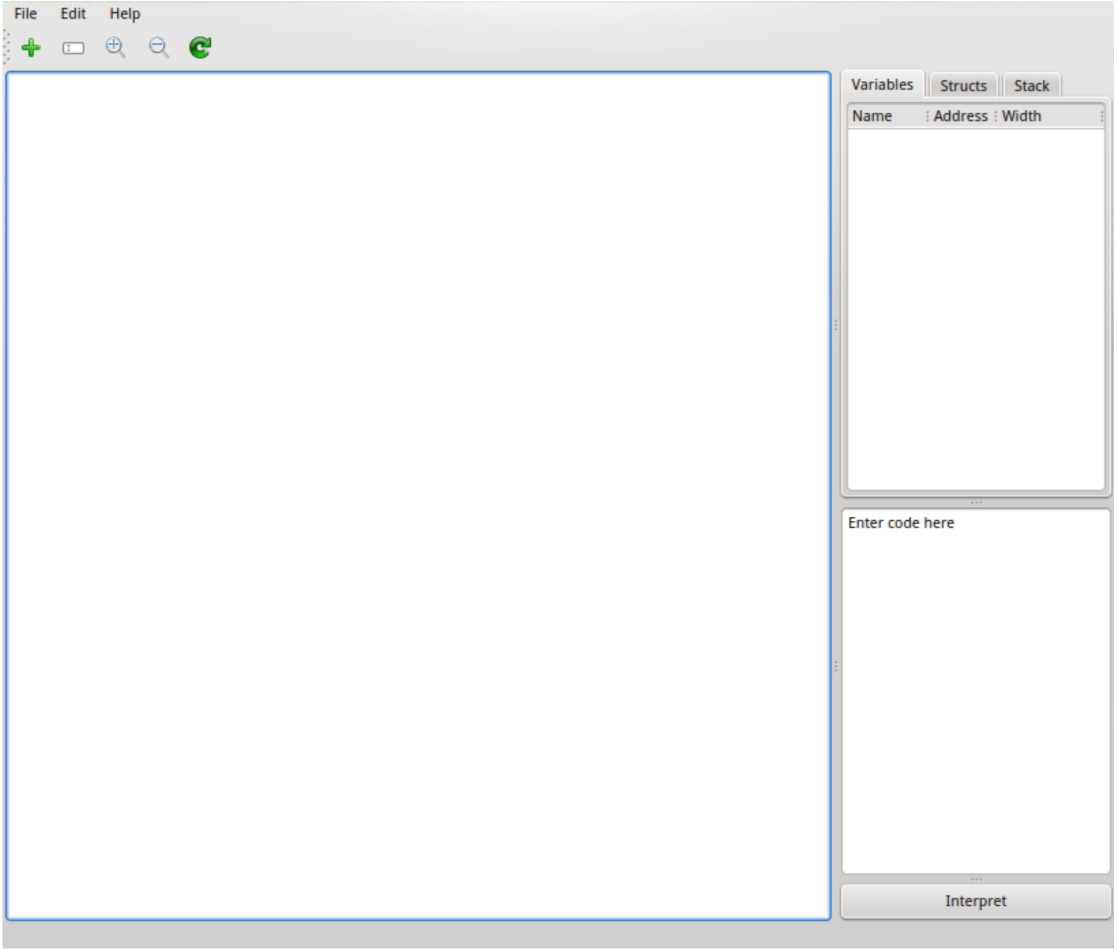


Figure 2: Empty main window.

concerning the underlying data structures and also to indicate the available struct types and their members.

Below the tabs is a text box, in which code for the C-like language defined above can be entered and then interpreted by using the wide push-button.

The tool bar pictured shows five icons which perform separate operations: add item, edit item, zoom in, zoom out and refresh view. The mouse wheel also performs zooming functions, i.e. scroll up to zoom in and vice versa. Once in a zoomed in state the visualisation area can be moved by clicking and dragging an empty space, or by using the scrollbars which will appear when applicable.

3.1 Adding Variables

Creating a variable instance to add to the visualisation can be accomplished in two ways, through the programming language interpreter (see previous examples) or by using the "Add Item" dialog. This is activated by clicking either the '+' tool-bar button or the "Add Item" entry in the "Edit" menu. Figure 3 shows this dialog with values entered for the declaration of a integer variable. If the entered values are at all invalid, the

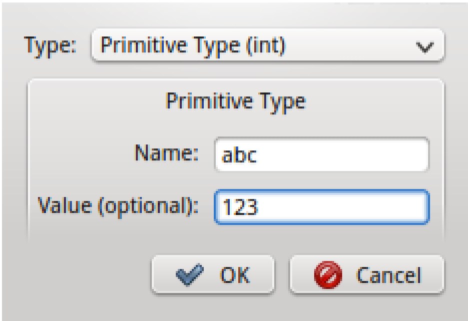


Figure 3: Add item dialog.

same dialog windows as those produced by entering text into the interpreter directly are utilised.

3.2 Editing Variables

Editing the values of variables can be achieved through the use of assignment expression statements using the interpreter, or the "Edit Item" dialog. The graphic representing the variable intended to be edited must be selected, then either click the edit tool-bar button or the "Edit Item" entry in the "Edit" menu.

3.3 Stack Table

The stack table provides information on the stack data structure which is operated on by the interpreter (see figures 5a and 5b). This information is useful in determining the memory layout of variables and struct data, and also to determine what value occupies at a particular stack address (the values in the left side headers are the stack addresses).

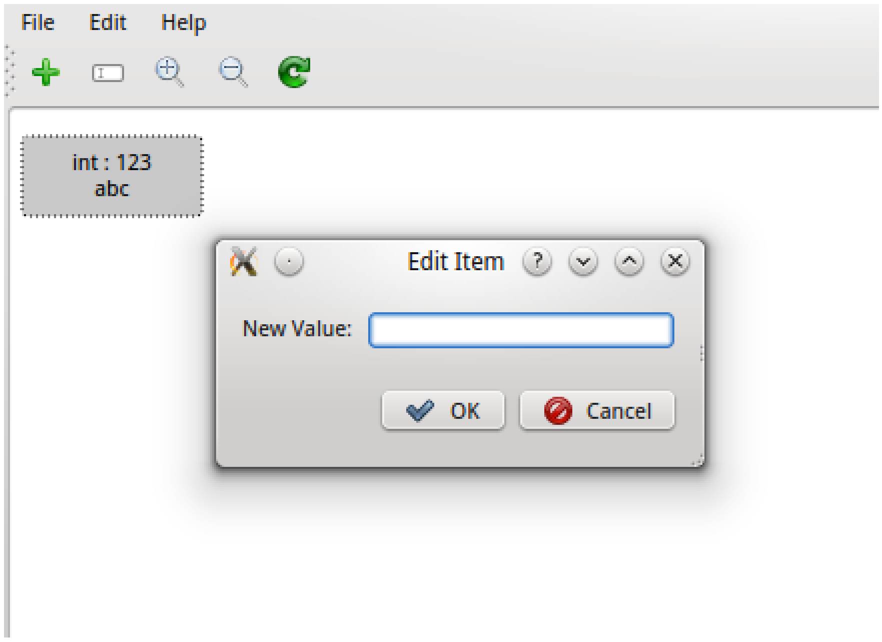
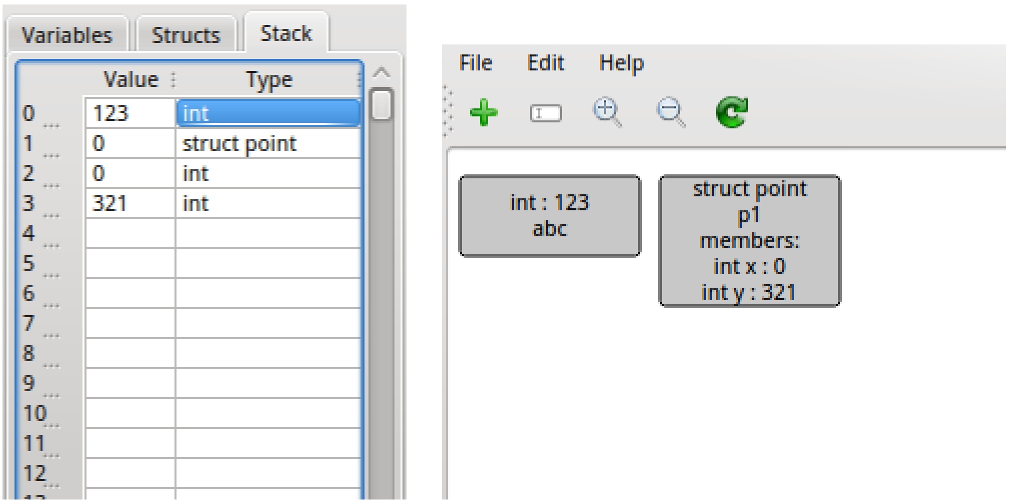


Figure 4: Edit item dialog.



(a) Stack table

(b) Items

Figure 5: Stack table and items.

3.4 Variable Table

Figure 6 shows the variable table for the previous example. It shows each declared (or dynamically allocated) variable's name, stack address and the number of stack entries

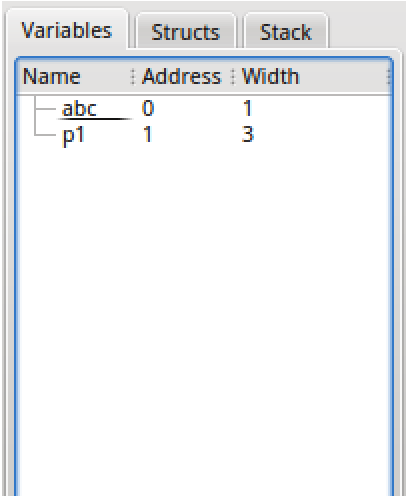


Figure 6: Variable table.

it occupies. This, in combination with the stack table and struct tree described below, provides some important information with regard to diagnosing problems caused by pointers, much like a visual debugging interface.

3.5 Struct Tree

The struct tree, as shown in figure 7, displays each defined struct data type, with each of its members and their types as its children. This can be used with the stack table

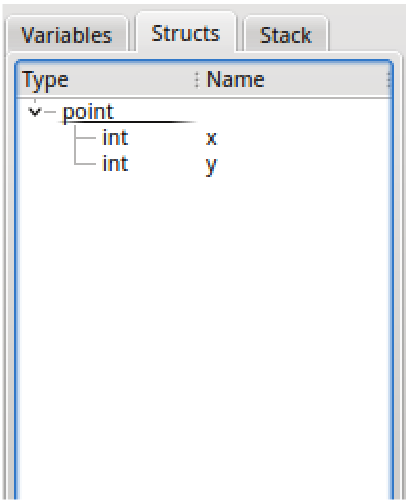


Figure 7: Struct tree.

to determine which stack entries are for which member variable by both the order they appear in, and their type.

B Modified Test Plan

Test Plan
for
Dynamic Data Structure Visualisation
Final Year Project

Christian Manning – p0928544x
De Montfort University

December 2011
Modified: April 2012

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to specify a plan for testing of the Dynamic Data Structure Visualisation (DDSV).

1.2 Related Documents

- Software Requirements Specification (SRS)
- Software Design Description (SDD)

1.3 Test Approach

1.3.1 Program Logic

The DDSV project utilises an iterative development methodology which means that testing is a continuous effort throughout the development life cycle. An ideal way to test program logic is with unit testing. Unit testing is not applicable in all cases, however, for the majority of the program logic, unit testing will be very beneficial.

1.3.2 User Interface

The User Interface (UI) can also be tested via unit tests using the QTestLib framework, which is part of Qt [1]. However, due to the nature of the UI requiring user interaction, not everything will be covered by unit tests. Therefore, testing the UI will require hand testing by myself to make sure the functionality works as I intend.

User testing removed April 2012

1.3.3 Interpreter

The interpreter will be using a small C-like language which, mostly, has well defined behaviour. This is beneficial as it allows tests to be crafted around well-known behaviour to ensure the correct output is produced. The parser, semantic analysis and virtual machine shall be tested as a singular unit as they are highly dependent on one another.

1.3.4 Integration

The above components are tested separately due to their different natures. But, their integration with each other also needs to be tested so that the system can work as a whole. This will be executed using unit tests.

1.3.5 Cross-platform Compatibility

Although Qt [1] makes cross-platform development such that the same code works on all supported platforms, there are some situations where the UI will behave slightly differently. One example of this is that Mac OS X often draws things in a different way to Windows and Linux. For this reason, DDSV's UI should be tested on all three of the target platforms (Windows, Linux, Mac OS X).

2 Test Cases

Test numbers (#) added April 2012

2.1 User Interface

#	Test	Input	Expected Result
UI1	Add item dialog	User selects "Add Item" from Edit menu or toolbar	Dialog prompts user for type
UI2	Add basic type	User selects "Primitive type (int)" from "Add item" and enters a name and optionally a value, then selects "OK" dialog	Graphical representation of an integer gets added to visualisation. Variable table and stack updated.
UI3	Add struct type	User selects "Struct Type (User defined)" from "Add item" dialog, enters a name, optionally enters values for the members, then selects "OK"	Graphical representation of user-defined type gets added to visualisation. Variable table and stack updated.

UI4	Add pointer	User selects "Pointer" from "Add item dialog". The type of the pointee is entered, primitive or struct. The desired name is entered. The value entered is an expression resulting in a stack address	Graphical representation of a pointer is added to the visualisation. It points to the item with the address specified. Variable table and stack updated.
	Remove an item or items	**Removed April 2012**	
UI5	Edit item dialog	User selects an item and clicks "Edit Item" from the Edit menu or the toolbar	"Edit Item" dialog prompts the user
UI6	Edit item	User enters new value or expression for item in "Edit Item" dialog and selects "OK"	The selected item is updated with the new value
	Binary Tree Layout	**Removed April 2012**	
	Generalised Tree Layout	**Removed April 2012**	
	Graph Layout	**Removed April 2012**	

2.2 Interpreter

New section April 2012

#	Test	Input	Expected Result
IN1	Declare a variable	`int a;`	Variable added to variable table. Stack table shows zero (0) entry with type `int`. Graphic added to visualisation.
IN2	Assign a value to variable	`a = 123;`	Stack entry shows value as 123. Visualisation updated.

IN3	Evaluate an expression, testing operator precedence	`a = 4 + 9 * 11 - 3;`	Stack entry shows value as 100. Visualisation updated.
IN4	Increment variable by 3	`a = a + 3;`	Stack entry shows value as 103. Visualisation updated.
IN5	Decrement variable by 9	`a = a - 9;`	Stack entry shows value as 94. Visualisation updated.

IN6	Define a struct type	`struct point {`
		`  int x;`
		`  int y;`
		`};`
			Struct tree shows the struct type name with its members as its children.

IN7	Declare struct variable	`struct point p1;`	Variable added to variable table. Stack has three new entries, with relevant types listed. Graphic added to visualisation.
IN8	Assign to a struct member variable	`p1.x = 42;`	Stack entry shows value as 42 for member variable. Visualisation updated.
IN9	Declare a pointer to a struct and assign it an address	`struct point *ptr = &p1;`	Graphic added to visualisation. Single stack entry added, with value equal to the stack address of `p1`.
IN10	Assign struct member through pointer	`ptr->y = 24;`	Stack entry shows value as 24 for member variable at correct address. Visualisation updated.

IN11	False if statement	`if(p1.x == 24) {`
		`  p1.y = p1.y - 1;`
		`}`
			Nothing happens.

IN12	True if statement	`if(p1.x == 42) {`
		`  p1.y = p1.y - 1;`
		`}`
			`p1.y` decreases by `1`. Stack entry updated. Visualisation updated.

IN13	Increment variable in a while loop	`while(p1.x < 100) {`
		`  p1.x = p1.x + 1;`
		`}`
			`p1.x` updated to 100. Stack entry updated. Visualisation updated.

IN14	Infinite loop	`while(true) {`
		`  123;`
		`}`
			Error dialog appears informing of stack overflow. Can safely resume.

IN15	Invalid input	`p1.4;`	Syntax error dialog appears describing the problem.
IN16	Incorrect input	`p1.c;`	Semantic analysis error dialog appears describing the problem.

2.3 History

Removed April 2012

3 References

[1] Nokia. *Qt*. URL: `http://qt.nokia.com` (visited on 12/2011).

C Testing Documentation

Testing Documentation
for
Dynamic Data Structure Visualisation
Final Year Project

Christian Manning – p0928544x
De Montfort University

April 2012

Contents

1 Introduction

This document contains the results of the tests defined in the test plan. These tests will be referred to by their test number (#) as defined in the plan throughout this document.

2 User Interface

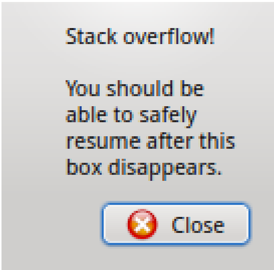
#	Test	Expected Result	Actual Result
UI1	Add item dialog	Dialog prompts user for type	Dialog prompts user for type
UI2	Add basic type	Graphical representation of an integer gets added to visualisation. Variable table and stack updated.	Graphical representation of an integer gets added to visualisation. Variable table and stack updated.
UI3	Add struct type	Graphical representation of user-defined type gets added to visualisation. Variable table and stack updated.	Graphical representation of user-defined type gets added to visualisation. Variable table and stack updated.
UI4	Add pointer	Graphical representation of a pointer is added to the visualisation. It points to the item with the address specified. Variable table and stack updated.	Graphical representation of a pointer is added to the visualisation. It points to the item with the address specified. Variable table and stack updated.
UI5	Edit item dialog	"Edit Item" dialog prompts the user.	"Edit Item" dialog prompts the user
UI6	Edit item	The selected item is updated with the new value.	The selected item is updated with the new value.

3 Interpreter

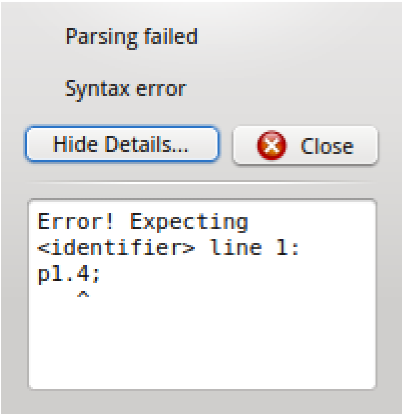
#	Test	Expected Result	Actual Result

IN1	Declare a variable	Variable added to variable table. Stack table shows zero (0) entry with type `int`. Graphic added to visualisation.	Variable added to variable table. Stack table shows zero (0) entry with type `int`. Graphic added to visualisation.
IN2	Assign a value to variable	Stack entry shows value as 123. Visualisation updated.	Stack entry shows value as 123. Visualisation updated.
IN3	Evaluate an expression, testing operator precedence	Stack entry shows value as 100. Visualisation updated.	Stack entry shows value as 100. Visualisation updated.
IN4	Increment variable by 3	Stack entry shows value as 103. Visualisation updated.	Stack entry shows value as 103. Visualisation updated.
IN5	Decrement variable by 9	Stack entry shows value as 94. Visualisation updated.	Stack entry shows value as 94. Visualisation updated.
IN6	Define a struct type	Struct tree shows the struct type name with its members as its children.	Struct tree shows the struct type name with its members as its children.
IN7	Declare struct variable	Variable added to variable table. Stack has three new entries, with relevant types listed. Graphic added to visualisation.	Variable added to variable table. Stack has three new entries, with relevant types listed. Graphic added to visualisation.
IN8	Assign to a struct member variable	Stack entry shows value as 42 for member variable. Visualisation updated.	Stack entry shows value as 42 for member variable. Visualisation updated.

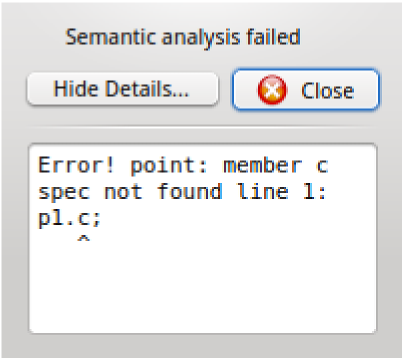
IN9	Declare a pointer to a struct and assign it an address	Graphic added to visualisation. Single stack entry added, with value equal to the stack address of `p1`.	Graphic added to visualisation. Single stack entry added, with value equal to the stack address of `p1`.
IN10	Assign struct member through pointer	Stack entry shows value as 24 for member variable at correct address. Visualisation updated.	Stack entry shows value as 24 for member variable at correct address. Visualisation updated.
IN11	False if statement	Nothing happens.	Nothing happens.
IN12	True if statement	`p1.y` decreases by `1`. Stack entry updated. Visualisation updated.	`p1.y` decreases by `1`. Stack entry updated. Visualisation updated.
IN13	Increment variable in a while loop	`p1.x` updated to 100. Stack entry updated. Visualisation updated.	`p1.x` updated to 100. Stack entry updated. Visualisation updated.
IN14	Infinite loop	Error dialog appears informing of stack overflow. Can safely resume.	Error dialog appears informing of stack overflow. Can safely resume. See figure 1a
IN15	Invalid input	Syntax error dialog appears describing the problem.	Syntax error dialog appears describing the problem. See figure 1b
IN16	Incorrect input	Semantic analysis error dialog appears describing the problem.	Semantic analysis error dialog appears describing the problem. See figure 1c



(a) Stack overflow



(b) Syntax error



(c) Semantic error

D Evidence of Coding – Git Log

```
commit 7cf95e9f9c1694a2fe0fe274b30915b08f9cd8ad
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Nov 21 15:54:42 2011 +0000

    Empty Qt project

commit f18186a6cfefaa78f647b06e33caeeddb4a9fb40
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Nov 23 14:30:48 2011 +0000

    Many changes:
    - Added some icons from the Oxygen project
    - Created a dialog for adding items
    - Created DataType class, which are the items added to the scene
    - QGraphicsScene now uses OpenGL by default when available (
        should be an option later)
    - Created a (currently non-functioning) zoomer for the graphics
        scene

commit a815143900c2117e4074a49a1493f3603b0c4732
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Nov 23 21:48:00 2011 +0000

    - Add types to combobox

    - Stop using OpenGL by default (uses lots of RAM)
    - Make drawing items dynamic to their text
    - Add item dialog now works for simple types
    - Non-working widgets added to dialog for user defined-types

commit 4f10f020cdea86738420846dbd0b9d3ef98c2872
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Nov 24 22:23:13 2011 +0000

    Make the add item dialog show only a specific group of options
        based on selected type.

commit a199448483c08e7b18dfadbb1bff82a3c48d645f
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Nov 26 16:34:31 2011 +0000

    Create a widget with a table for adding members to new types

commit 1dcfea4833d2e9db2c34eadd6c8a5ef1c1be495b
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Nov 28 11:02:37 2011 +0000
```

```
    Modify some widget minimum sizes to be more compatible with linux
        . Add
    c++11 support (qt-4.8+)

commit 28250a888a273fa7d42bebe6242fe27d24307faf
Author: Christian Manning &lt;cmanning999@gmail.com&gt;
Date:   Mon Nov 28 13:34:30 2011 +0000

    Add rudimentary zooming + columns for user defined types

commit 448261eaf3cb31d2be2ff0e4591d7c9a08cdf6d9
Author: Christian Manning &lt;cmanning999@gmail.com&gt;
Date:   Fri Dec 9 23:25:33 2011 +0000

    many changes:

    - remove c++11 until qt-4.8 is out
    - store graphics items in a hash table instead of linked list.
        allows access by name, or maybe by "address" in the future
    - create a "Pointer" type for graphically representing pointers
    - implement functionality to link a pointer type to a normal data
        type
    - reference link follows the pointer object and data object (
        quite inefficiently atm)

commit abb1a39c2ac8ca1789601d2716cf371029890ad7
Author: Christian Manning &lt;cmanning999@gmail.com&gt;
Date:   Tue Dec 13 05:20:12 2011 +0000

    Several changes:

    - Make only simple types work for the prototype
    - Add an edit item dialog
    - Resize window a bit

commit 241b43110b0ed20e6e6d328984c346dd996beb4c
Author: Christian Manning &lt;cmanning999@gmail.com&gt;
Date:   Mon Jan 2 17:28:15 2012 +0000

    Add Boost includes in preparation for interpreter.

commit df65502fe542c217af42026abb6eccc93ff23b43
Author: Chris Manning &lt;cmanning999@gmail.com&gt;
Date:   Fri Feb 17 16:39:41 2012 +0000

    Lots of big additions:
     - We have an interpreter! Based around the Boost Spirit conjure1
    example (refactored a bit).
```

```
 - What currently works:
   - variable declaration + assignment
   - function definition
   - function calls, but only as part of an expression
   - scope with functions, may break later
   - struct definitions are parsed but not interpreted yet
   - basic error handling
   - operator precedence
 - Some buttons added to the UI for debugging purposes + some
refactoring of UI

commit 8b9e47258062348ff99f0ef99c48664ed2a1a7cb
Author: Chris Manning <cmanning999@gmail.com>
Date:   Fri Feb 17 16:58:13 2012 +0000

    Add interpreter file to ddvs.pro for qmake building.
    Also use boost-1.48 over 1.47 (this may just use the system boost
        in the future)

commit 6d979246092b4f47d2c42a5d17485780c50f3155
Author: Chris Manning <cmanning999@gmail.com>
Date:   Fri Feb 17 16:59:45 2012 +0000

    Change the warnings flag to -Wextra to ignore all the warnings
       about missing parentheses in the boost spirit grammar

commit e8da8bd29cd26caf1be993f6abaaac1f6db29839
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Feb 24 12:51:32 2012 +0000

    Stop the noise

commit 462a235eb0ed03af87fec740221132982fe12b9d
Merge: e8da8bd 6d97924
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Feb 24 12:51:52 2012 +0000

    Merge branch 'master' of github.com:chrismanning/ddvs

commit 5bc867b902e7e368277bb5313c44402125b56263
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Feb 24 22:41:23 2012 +0000

    Comment unused variables in function declarations to get rid of
       warnings. Also change local includes to "" instead of <>

commit d6dd5ed87132f220169c187708161ddc8a71e8ad
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Feb 24 22:42:36 2012 +0000
```

```
    Add boost to includes for windows.

commit 8f06f1f96dbec5024c024e48b8d944af350e763c
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Feb 24 22:51:58 2012 +0000

    Some slight changes to make it work on windows. A parser instance
         has to be created each time the user requests it which may be
         inefficient.

    Stack offsets for functions are now be calculated (hopefully
        correctly).
    MSVC should also work, though it doesn't seem to with boost-1.48,
         it works fine with 1.47, so this may have to be downgraded as
         MSVC produces much smaller binaries (orders of magnitude).

commit b3f51489618938d8890340d0f5d9500879d023a5
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Feb 25 16:36:51 2012 +0000

    Fix function declarations + disable nested functions (not that
        they
    worked anyway)

commit cb7f8fc35d2e81dbc51a1f080ad0e1f08ecb24e5
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Feb 25 23:39:23 2012 +0000

    Add boost includes for Mac OS X.

    It builds and runs  OS X with no other changes!
    Also note that version 1.49 is explicitly specified

commit d5752052eaee181172095862aa5ff614e1ff9fe2
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Feb 25 23:48:22 2012 +0000

    Add some (extremely) rudimentary code just for adding variables
    originating from the interpreter to the graphics scene.

commit 3105f0b7c3a8832f85a7f2798365bb93976a4f18
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Feb 26 00:51:46 2012 +0000

    Update to boost-1.49 for windows which fixes building with MSVC

commit caad96e12f14a1c8df30e8df347526eb999c171d
Author: Christian Manning <cmanning999@gmail.com>
```

```
Date:   Mon Feb 27 11:34:47 2012 +0000

    Cleanup.
    Got rid of some commented code. Changed the grammer slightly

commit 7fae8bd0b420a2365b2b8cea6df7c9dd07343e17
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Feb 27 21:27:59 2012 +0000

    Add proper support for parsing pointer types.

commit 15f1ea3dc36aa6380c7db6fea153d9fdec9cd0e5
Author: Chris Manning <cmanning999@gmail.com>
Date:   Thu Mar 1 22:07:06 2012 +0000

    Start user docs in LaTeX

commit a6c45df83ba89cfd22f4c7d5c84e0917dd4cca6d
Author: Chris Manning <cmanning999@gmail.com>
Date:   Sun Mar 4 19:41:49 2012 +0000

    Finished the language grammar + short descriptions
    Added operator precedence table + some examples

commit fb52c4b19cca52176890319ffad9170108c21a16
Author: Chris Manning <cmanning999@gmail.com>
Date:   Sun Mar 4 19:45:52 2012 +0000

    Remove unneeded operators and keywords

commit 3344bfb5bd9d4ea4199dc8db86167c8e488a521d
Author: Chris Manning <cmanning999@gmail.com>
Date:   Sun Mar 4 19:50:26 2012 +0000

    Allow structs to have pointer and struct members

commit 635dff74dc29c463863159a6a65165ff81f9dfec
Author: Chris Manning <cmanning999@gmail.com>
Date:   Sun Mar 4 19:54:11 2012 +0000

    Remove the types QStringList as it's redundant, but the GUI still
        needs to know about the types somehow

commit c1eef769eda6029ac542d6fda5aa2f9d088b7d3e
Author: Chris Manning <cmanning999@gmail.com>
Date:   Sun Mar 4 19:55:36 2012 +0000

    Rework the UI using C++ rather than the form designer.
```

```
    It now expands properly on resize and the debugging buttons are
        only shown when built in debug mode.
    Need to add a queue back.

commit e110b91a1d27b10138c036f7f47ca188097e302b
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Mar 5 00:42:33 2012 +0000

    Add struct example

commit bbe434de604026961400a100bd48923c3b3425df
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Mar 29 20:17:15 2012 +0100

    -Go back to separate def & header compilation for speed ups
    -Also allow pointers as function return types

commit 201dd533cd6ef880190543aca2a8be2efaab9313
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Mar 29 23:10:28 2012 +0100

    Report errors only when necessary

commit 6ebee21ca7e7f0874a189e989e0432c89c92012a
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Mar 30 19:38:27 2012 +0100

    Improve error reporting + add struct debugging

commit 6243244a14e69d17895311e361100d74a5d61299
Author: Christian Manning <cmanning999@gmail.com>
Date:   Tue Apr 3 17:14:02 2012 +0100

    Work towards getting parser closer to the C standard BNF to fix
        stuff

commit 0d069e58f6dee034bde2254657d3fada7210545e
Author: Christian Manning <cmanning999@gmail.com>
Date:   Tue Apr 3 23:26:09 2012 +0100

    Continue fixing interpreter + some cleanup

commit 39b76f81d6c9dca5aa7e8762567708da121c1dcb
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 5 01:01:42 2012 +0100

    Bring expression parsing rules closer to C standard. Interpreter
    temporarily broken.
```

```
commit 1e6b51f8b6b1c088475fd27b9f180518a776131c
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 6 00:28:47 2012 +0100

    - Parser should now be fully functional
    - Interpreter still broken
    - Some clean up

commit 98e3c026e2f3623790d715cecb7ad6f7d536877c
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 12 23:03:51 2012 +0100

    Allow more attributes to be annotated for better errors, etc.
    + removal of some commented code

commit 1a2c539c46f13b5aa1809cabd61fad4da2ee695b
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 12 23:06:02 2012 +0100

    Fix parser grammar so that it's easier to work with

commit 5ab4c6c1a5d396a244beec4c60cb16b2b026759c
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 12 23:11:21 2012 +0100

    Use c++0x(11) flag on everything other than msvc where it's
        default. Ues boost-1.49 on linux.

commit 003c97d183e829824a11614ba3729feb5607f26b
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 12 23:14:52 2012 +0100

    Fixed interpreter! Structs also work now. Pointers to come.
    + lots of commented code deleted

commit d2f15839b3765e04ff9f9a0e8e82607362ea1ef2
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 00:14:30 2012 +0100

    Implement the new keyword for memory allocation + implement
        pointers (hopefully work)

commit a6928e27b5b75fac27ef0cffc364da5a60df4a02
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 00:15:46 2012 +0100

    Show a dialog with parser/interpreter errors

commit 7e2a8a9d04891560acb0237bd8449141f7063292
```

```
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 01:11:02 2012 +0100

    Remove an unneeded include. Fixes build on windows

commit 287dc7d72b5b4fc9646171dd57e51f2c8ac91f34
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 17:38:18 2012 +0100

    Get rid of BOOST_FOREACH in favour of range based for (C++0x/C
        ++11)

commit b52465ade47b3e6c937504905a877914523f661f
Merge: 287dc7d 7e2a8a9
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 17:38:56 2012 +0100

    Merge branch 'master' of github.com:chrismanning/ddvs

commit 977b987c511029c3ba380f09ac676c9dc7bcd71f
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 17:47:17 2012 +0100

    Remove some more unneeded includes

commit 78e9bef76a19979b562ea6941fa1d392d248c413
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 21:43:01 2012 +0100

    Switch remaining BOOST_FOREACH to range based for + reduce clang
    warnings

commit 228fd607b0a0deb2f86c8d82036bea330d4f5eb7
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 14 22:33:10 2012 +0100

    Remove the now unneeded explicit type conversion operators.
        Should now
    be MSVC compatible.

commit 44e8665f79c1d99e117631c6dc9e21ac6f5ea281
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Apr 16 14:01:16 2012 +0100

    Customise QGraphicsView for zooming

commit 0515c89fd83334a8b9edad175650d79b692ce1af
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Apr 16 19:07:40 2012 +0100
```

```
    Fix zomming with the wheel, enable dragging with the mouse and
        disable
    zoomwidget

commit c5d784b9ce3f427f0adeac6ed6fd537bef4fbb15
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Apr 16 19:37:30 2012 +0100

    Add an "About Qt" option in the Help menu which shows details
        about Qt

commit 9ef97c6647b78fcb689216858f4da1ce0aaef0c9
Author: Christian Manning <cmanning999@gmail.com>
Date:   Mon Apr 16 21:47:06 2012 +0100

    Remove unused variable

commit 08ea21daf07fe663128e9ec0d32690ca53bfa937
Author: Christian Manning <cmanning999@gmail.com>
Date:   Tue Apr 17 17:35:03 2012 +0100

    Fix structs + '&' addressing

commit 75a3645e7597070b45b5f5aa6ec615f9ece11135
Author: Christian Manning <cmanning999@gmail.com>
Date:   Tue Apr 17 18:36:02 2012 +0100

    Fix function definition grammar

commit d12778c45a7a87e0fde294c065cb5c374d049a85
Author: Christian Manning <cmanning999@gmail.com>
Date:   Tue Apr 17 18:36:55 2012 +0100

    Make the error message boxes show code in monospace font

commit e8f9a67cd701cb5ef6a5d4a500f58233bb6d5f14
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 01:02:36 2012 +0100

    Show defined structs and their members in a tree in a tab in the
        UI
    + start to reimplement functions

commit 529a18edec4612bf019867953916c0613294840d
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 17:42:55 2012 +0100

    Fix bug of pointer type widths not being correct
```

```
commit 2dd0cf91f356dc812550ea356271109047071919
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 17:43:38 2012 +0100

    Move stack size to a #define macro

commit c92eae784bbe8204ca1946692056c3fb226020da
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 17:44:13 2012 +0100

    Template function to create a range from two iterators

commit 743f1b0419aa759ff36a246dc18bf4ba4781c524
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 17:45:25 2012 +0100

    Populate stack and variable lists after each interpreter run

commit 6c7e935d9c20d2e44e50d1599a718c9726d9c5b8
Author: Christian Manning <cmanning999@gmail.com>
Date:   Wed Apr 18 18:03:12 2012 +0100

    Remove ability to remove items from visualisation as it doesn`t
       really
    make sense

commit 839822118e962f00b4590c16ddd811b51d568145
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 19 02:38:03 2012 +0100

    Fix memory allocations, structs + weird variable bug

commit 37c5bda9509531b6006f8c180157fd07a91ba96d
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 19 02:40:30 2012 +0100

    Add variables to visualisation again

commit c479645af8ea69df6782e2116874f7982f5122df
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 19 02:42:00 2012 +0100

    Reverse scroll wheel zooming

commit 784657a298cd72c5909dfd48156baf2082c5b26b
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 19 18:31:58 2012 +0100
```

```
    Fix struct allocations properly

commit ec7a5ebc71f116302793735bc8af08f30af2ebb5
Author: Christian Manning <cmanning999@gmail.com>
Date:   Thu Apr 19 20:57:44 2012 +0100

    Draw structs, fix drawing pointers

commit 9fdbcb3dfc7f0b368fa629c955885ecc8e3901a6
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 00:23:53 2012 +0100

    Fix `if` and `while` statements + removed `else`

commit af3260a9dcd4fbe55628fb0537a6db447ba703c2
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 01:15:05 2012 +0100

    Set stack size to 8192, avoid stack overflows + remove else
        keyword

commit 359834b3ccb03270528f2942bc6e97fe15470629
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 03:51:32 2012 +0100

    Work on user docs some more

commit a736a9289a177e0a0001cf71544aa72757d4b1a7
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 19:04:12 2012 +0100

    Re-enable adding and editing items via the UI
    + remove the ability to create structs with the UI

commit 11fe8fee482ea0bf05bef236e130ae9ffc3504b6
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 19:10:18 2012 +0100

    Remove some unused menu items

commit 3e08347dbf113bd29628b4531b6df3b9246d5de5
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 19:22:46 2012 +0100

    Fix editing items from UI

commit 6a1808ee4290924cbfa649adcf015fe6ed93d839
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 20:52:20 2012 +0100
```

```
    Set some more initial column widths

commit d56195511a90b4df4d6c1364057b6a5739f5031e
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 22:28:12 2012 +0100

    Remove the remains of zoomwidget

commit 1dbf50e00c09911d831e4b36b213daeebd26b5d8
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 22:30:42 2012 +0100

    Add a simple "About" dialog

commit c61657b94d6ef5a78c2d6929fe51e33256149363
Merge: 1dbf50e d561955
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 22:31:05 2012 +0100

    Merge branch 'master' of github.com:chrismanning/ddvs

commit ec957200510b69820db1e06759a7b382705fc9d0
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 23:00:00 2012 +0100

    Make the pointer dialog behave correctly

commit 12a51888ca5b5270a2f95b19d4117d6f3e771b4a
Author: Christian Manning <cmanning999@gmail.com>
Date:   Fri Apr 20 23:07:23 2012 +0100

    Allow 6 items per row

commit 3fbd4a97ed626157f4379bcac297b8ed2fcb1a49
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 21 00:55:16 2012 +0100

    Change dialog window title

commit 89e8a51df3e2cf0863bebabeda878daa74bf1ed6
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 21 02:19:43 2012 +0100

    Add lots of things to the user docs. Hopefully finished them

commit 6db4037f2b83f5eaa80926d5fb5d90fd472cd10d
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 21 02:27:45 2012 +0100
```

```
    Remove old graphics items

commit fa35da22040d1bd1e027801d43120e0eabe9cf6e
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 21 19:35:10 2012 +0100

    Fix scoping of struct definitions

commit 101145b25122af048cc4f2b50dd4fc9e7fb1a998
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sat Apr 21 20:29:27 2012 +0100

    Remove informative text for non-parser error

commit b3ba23342d33cba15383deebb25bc33267421310
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 00:00:35 2012 +0100

    Make compatible with clang and probably MSVC

commit 25b5ecb049480900125f104874ed2182febcc492
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 00:19:20 2012 +0100

    Adjust row height for windows compatibility

commit ac0f53880c0bf37204f599ff67e1111d99446100
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 00:34:44 2012 +0100

    Fix struct rendering bug

commit b9a4d4fb42971fa1233798bfc76dda6d77664bce
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 02:17:39 2012 +0100

    Enable compiler warnings again and get rid of the parentheses
        ones

commit 14057b7d3362a6586b1239bdc53a4f3408298314
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 02:30:52 2012 +0100

    Remove some initialisation order warnings + some comment cleanup

commit 58f5a3ef8e190528b51a5bc0805a543cd3a87393
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 02:31:47 2012 +0100
```

```
    Fix error reporting on release builds

commit ec2f2432def04a7173bb98fc4f93c96b459fdc99
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 02:32:15 2012 +0100

    Use Courier on Windows as the monospace font

commit 422dfea8fb2ce682d85a431c83d92a12b7a8ff36
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 02:32:49 2012 +0100

    Disable boost spirit qi debugging

commit baab1813ce350f50dea9be03ea312b29d2993a43
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 14:17:51 2012 +0100

    Remove some unused parameters and comment out function to get rid
        of
    most warnings

commit d6f264618b1a3b8f2f175f1e3e8502c47545ba09
Author: Christian Manning <cmanning999@gmail.com>
Date:   Sun Apr 22 14:37:58 2012 +0100

    Properly fix errors
```

E Evidence of Coding – Samples

What follows are several samples of the C++ code from this project's implementation, highlighting some distinct features.

E.1 Expressions Grammar

```
//expression_def.h
// operators from lowest to highest precedence
assign_op.add
    ("="  , ast::op_assign)
    ;

logical_or_op.add
    ("||", ast::op_logical_or)
    ;
logical_and_op.add
    ("&&", ast::op_logical_and)
    ;

equality_op.add
    ("==", ast::op_equal)
    ("!=", ast::op_not_equal)
    ;

relational_op.add
    ("<"  , ast::op_less)
    ("<=", ast::op_less_equal)
    (">"  , ast::op_greater)
    (">=", ast::op_greater_equal)
    ;

additive_op.add
    ("+", ast::op_plus)
    ("-", ast::op_minus)
    ;

multiplicative_op.add
    ("*", ast::op_times)
    ("/", ast::op_divide)
    ;

unary_op.add
    ("+", ast::op_positive)
    ("-", ast::op_negative)
    ("!", ast::op_not)
    ("*", ast::op_indirection)
```

```
        ("&", ast::op_address)
        ;

    memory_op.add
        ("new", ast::op_new)
        ;

    struct_op.add
        ("->", ast::op_select_point)
        (".", ast::op_select_ref)
        ;

    postfix_op.add
        ("++", ast::op_post_inc)
        ("--", ast::op_post_dec)
        ;

    keywords.add
        ("true")
        ("false")
        ("if")
        ("while")
        ("struct")
        ("return")
        ("new")
        ("error")
        ;

    //expressions in reverse precedence
    assignment_expression = logical_OR_expression > -unary_assign;

    allocation_expression = memory_op > type_specifier;

    unary_assign = assign_op > (allocation_expression |
       logical_OR_expression);

    logical_OR_expression = logical_AND_expression > *(logical_or_op >
       logical_AND_expression);

    logical_AND_expression = equality_expression > *(logical_and_op >
       equality_expression);

    equality_expression = relational_expression > *(equality_op >
       relational_expression);

    relational_expression = additive_expression > *(relational_op >
       additive_expression);
```

```
additive_expression = multiplicative_expression > *(additive_op >
    multiplicative_expression);

multiplicative_expression = unary_expression > *(multiplicative_op >
    unary_expression);

unary_expression =
        -unary_op
    >   postfix_expression
    ;

struct_expr = struct_op > identifier;

postfix_expression =
        primary_expression
    >   *(struct_expr | postfix_op)
    ;

primary_expression =
        uint_
    |   identifier
    |   bool_
    |   ('(' > logical_OR_expression > ')')
    ;
//end expressions

type_specifier =
        types
    |   struct_specifier
    ;

declarator = matches['*'] > identifier;

struct_member_declaration = type_specifier > declarator > ';';

struct_specifier =
        lexeme["struct"] > type_id > -('{' > +
            struct_member_declaration > '}');

identifier =
        !(keywords | types)
    >>  raw[lexeme[(alpha | '_') >> *(alnum | '_')]]
    ;

type_id =
        !(keywords | types)
    >>  raw[lexeme[(alpha | '_') >> *(alnum | '_')]]
    ;
```

The above shows the grammar for expressions written using Boost Spirit which are declared in an object constructor. This sample highlights the expressiveness of Spirit and its similarity to EBNF notation. Due to C++'s operator limitations there are some notable differences to EBNF. `>` and `>>` are the *expectation* and *followed-by* operators, respectively, which are not required at all in standard (E)BNF. `-` also differs in that it is the *optional* operator here, but the *not* operator in EBNF. The *not* operator in Spirit is the same as that in regular C++, `!`.

Each of these rules has an attached *attribute*, making this an attribute grammar. When a rule is matched, its attribute is created with its various members containing values, strings and other information.

```
//ast.h
typedef boost::variant<
        nil
      , Bool_Value
      , Int_Value
      , identifier
      , boost::recursive_wrapper<logical_OR_expression>
  >
primary_expression;

struct logical_OR_op : Typed {
    optoken operator_;
    logical_AND_expression rhs;
};

struct logical_OR_expression : Typed {
    logical_AND_expression lhs;
    std::list<logical_OR_op)> rest;
};

BOOST_FUSION_ADAPT_STRUCT(
    logical_OR_op,
    (optoken, operator_)
    (logical_AND_expression, rhs)
)
BOOST_FUSION_ADAPT_STRUCT(
    logical_OR_expression,
    (logical_AND_expression, lhs)
    (std::list<logical_OR_op>, rest)
)
```

The above snippet demonstrates some *attribute* types that are associated with rules defined in the previous sample. Due to the way Boost Spirit interacts with objects, it needs to access the attribute members as a *tuple*. Boost Fusion contains a macro, BOOST_FUSION_ADAPT_STRUCT, which adapts struct types to act as tuples.

This sample also shows the use of Boost Variant, representing the attribute of a primary expression. A primary expression, also shown in the previous sample, is defined as being either an integer, boolean, identifier or logical_OR_expression; it can be any one of these, meaning it can only be stored in a specialised container like a variant.

A type-safe way to access the variant's contents is through using the apply_visitor function.

```
//interpreter.cpp line 778
ast::Type global::operator()(ast::type_specifier& t)
{
    qDebug() << "Processing: ast::type_specifier";
    type_resolver tr(error_, current_scope);
    return t.apply_visitor(tr);
}
//interpreter.h line 297
struct type_resolver : boost::static_visitor<ast::Type>
{
    type_resolver(error_handler& error_, scope* env)
        : env(env), error_(error_)
    {}

    ast::Type operator()(ast::Type& t)
    {
        return t;
    }

    ast::Type operator()(ast::struct_specifier& ss)
    {
        qDebug() << "Processing: ast::struct_specifier";
        auto t = env->lookup_struct_type(ss.type_name.name);
        //[...] snipped
        //process struct definition or declaration
        return t;
    }

    void error(int id, std::string const& what)
    {
        error_("Error! ", what, error_.iters[id]);
    }

    scope* env;
    error_handler& error_;
};
```

This sample shows apply_visitor in use in combination with a function object type_resolver to determine the type of an attribute. Function objects are also used extensively throughout the semantic analysis stage, enabling objects to be called like functions, while also maintaining state.

E.2 Error Handling

```
//parts of expression_def.h
typedef function<error_handler> error_handler_function;
typedef function<annotation> annotation_function;

on_error<fail>(assignment_expression,
    error_handler_function(error)(
        "Error! Expecting ", _4, _3));
on_success(assignment_expression,
    annotation_function(error.iters)(_val, _1));
```

The above is an example of how error handling is implemented, using Boost Spirit and Phoenix (utilising functional programming techniques). If an error occurs, the error_handler_function function object is called along with what it was expecting (_4) and where in the input this happened (_3). On a successful parse of a rule, the annotation_function is called with its attribute (_val) and its start position (_1), tagging the attribute with an id which is then associated with the position. Whenever an error occurs outside of parsing, the error handler object can be called, with the attribute's id to find its position. The items beginning with an underscore (_) are placeholders for Phoenix functors, and are used to access otherwise difficult to reach information.

E.3 C++11

```
//interpreter.cpp line 1035
bool global::operator()(ast::translation_unit& ast) {
    for(ast::statement_or_function& fs : ast) {
        if(!fs.apply_visitor(*this)) {
            return false;
        }
    }
    return true;
}
```

This section of code shows the entry point of all grammar attributes. The translation_unit is the "root" of the abstract syntax tree (AST), and can contain zero, one or many function definitions *or* statements, stored as a `std::list<boost::variant>`. Of course, as long as functions remain unimplemented it just be seen as a list of statements. The purpose of demonstrating this piece of code was to show the range-based `for` loop in action. This is a new language feature introduced in the new C++ standard (C++11), influenced by the similar features available in many modern programming languages. The range-base `for` reduces code complexity and greatly increases readability, though can be improved further by using another new language feature: type inference. The following snippet shows the modified loop.

```
for(auto& fs : ast) {
    if(!fs.apply_visitor(*this)) {
        return false;
    }
}
```

Through the use of a small utility function template, `makeRange` shown below, a range can be made from any pair of iterators, enabling *slicing* for some type of iterators. An example follows.

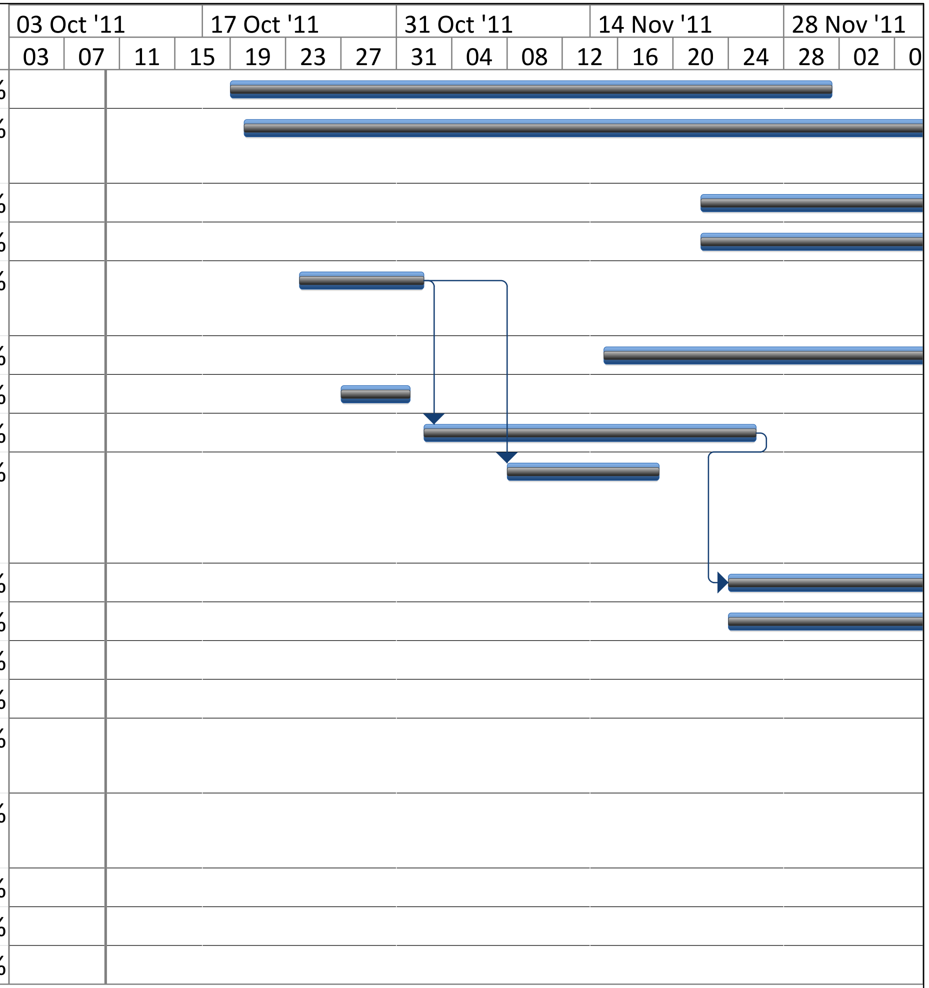
```
//interpreter.h line 602
template <typename Iterator>
struct Range
{
    Range(Iterator const& begin_, Iterator const& end_)
        : begin_(begin_), end_(end_) {}

    Iterator const& begin()
    {
        return begin_;
    }
    Iterator const& end()
    {
        return end_;
    }

    Iterator const& begin_;
    Iterator const& end_;
};

template <typename Iterator>
Range<Iterator> makeRange(Iterator const& begin, Iterator const& end)
{
    return Range<Iterator>(begin, end);
}
```

```
//mainwindow.cpp line 233
for(auto const& var : makeRange(interpreter.getStack().begin(),
    interpreter.getStackPos())) {
     QString p = "";
     if(var.type.pointer) {
          p += "*";
     }

     auto t1 = new QTableWidgetItem(QString::number(var.var));
     t1->setToolTip(t1->text());
     stackTableWidget->setItem(i, 0, t1);

     auto t2 = new QTableWidgetItem(QString::fromStdString(var.type.
         type_str)+p);
     t2->setToolTip(t2->text());
     stackTableWidget->setItem(i, 1, t2);
     ++i;
}
```

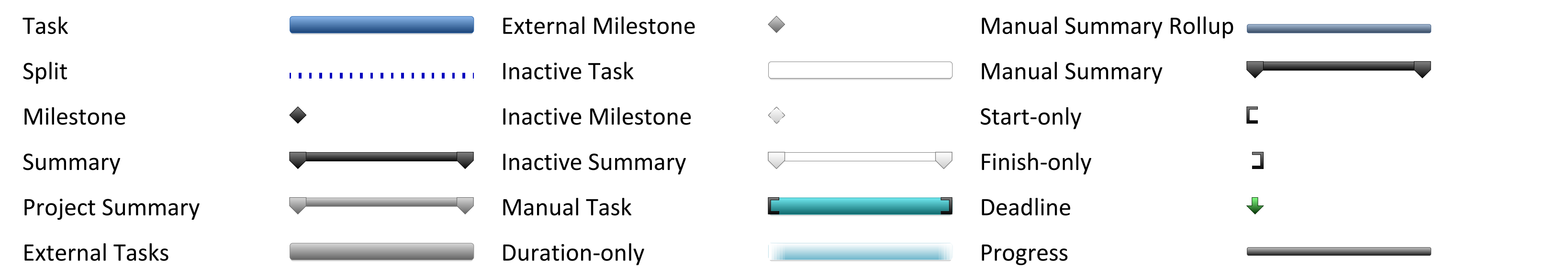
The example shown is the code that populates the stack table widget in the UI, which is executed at the end of every interpreter run. It shows how the `makeRange` function takes only up to what is represented by the second parameter, which in this case is the current stack offset. This means that this loop will only operate on the currently allocated memory (stack positions), else it would try to populate the table with thousands of non-existent stack entries.

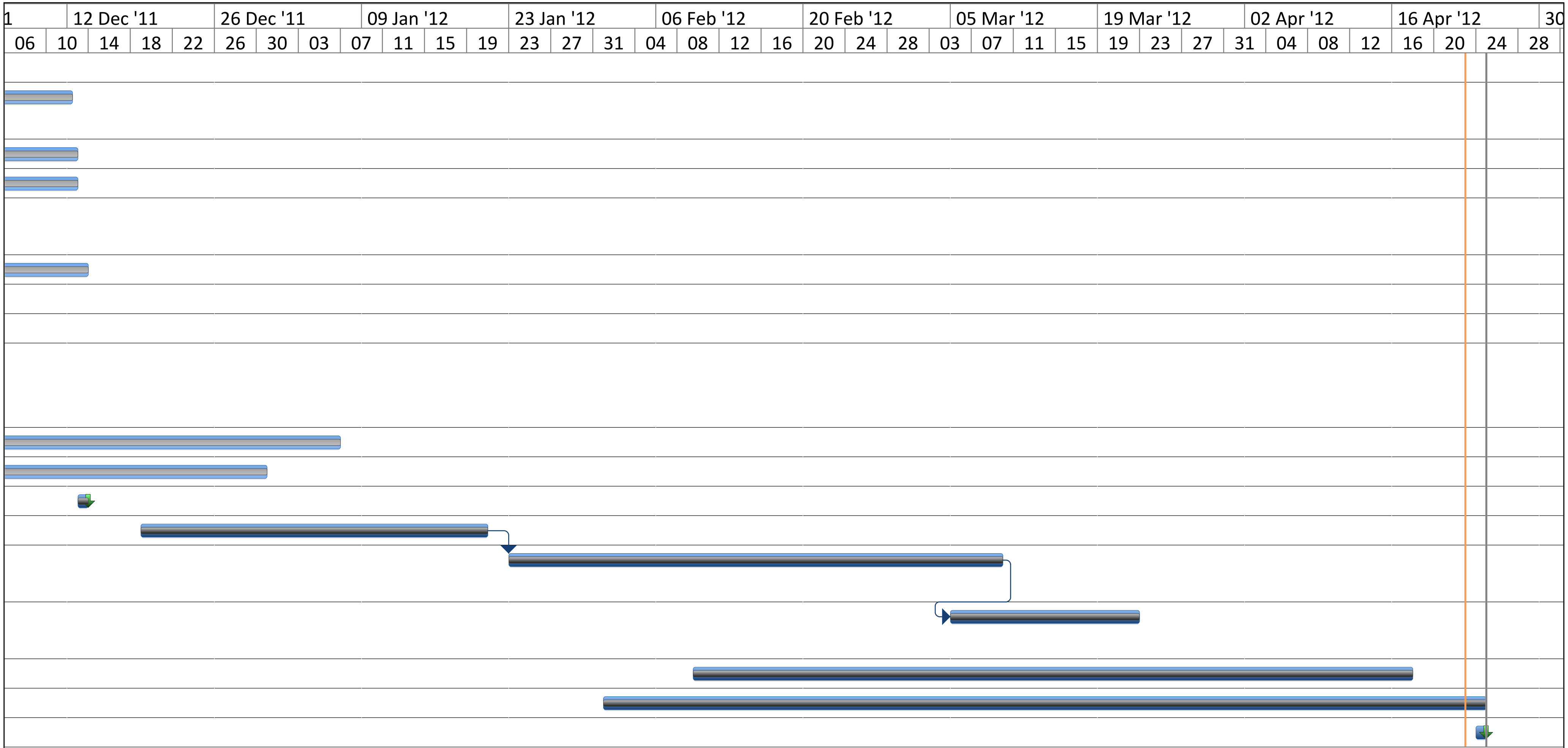
F Modified Project Plan

ID	Task Mode	Task Name	Duration	Start	Finish	% Comp.
1		Write Contract	31.5 days?	Wed 19/10/11	Thu 01/12/11	100%
2		Requirements Specification	37.5 days?	Thu 20/10/11	Mon 12/12/11	100%
3		Design Documentation	15 days	Tue 22/11/11	Mon 12/12/11	100%
4		Test Plan	15 days?	Tue 22/11/11	Mon 12/12/11	100%
5		Research HCI for pedagogic tools	7 days?	Mon 24/10/11	Tue 01/11/11	100%
6		Literature Review	21 days?	Tue 15/11/11	Tue 13/12/11	100%
7		Rudimentary UI	3 days?	Thu 27/10/11	Mon 31/10/11	100%
8		Design UI	18 days?	Wed 02/11/11	Fri 25/11/11	100%
9		Design graphics for representing pointers and data structures	9 days?	Tue 08/11/11	Fri 18/11/11	100%
10		Implement UI	32 days?	Thu 24/11/11	Fri 06/01/12	100%
11		Implement Data Model	27 days?	Thu 24/11/11	Fri 30/12/11	100%
12		First Hand-in	1 day	Tue 13/12/11	Tue 13/12/11	100%
13		Design small DSL	25 days?	Mon 19/12/11	Fri 20/01/12	100%
14		Implement interpreter for DSL	35 days?	Mon 23/01/12	Fri 09/03/12	100%
15		Integrate language with UI	14 days?	Mon 05/03/12	Thu 22/03/12	100%
16		User documentation	48.5 days?	Thu 09/02/12	Tue 17/04/12	100%
17		Final Report	60 days?	Wed 01/02/12	Tue 24/04/12	100%
18		Final Deadline	1 day	Tue 24/04/12	Tue 24/04/12	0%



Project: ProjectPlan
Date: Mon 23/04/12





Project: ProjectPlan
Date: Mon 23/04/12

Task	External Milestone	Manual Summary Rollup
Split	Inactive Task	Manual Summary
Milestone	Inactive Milestone	Start-only
Summary	Inactive Summary	Finish-only
Project Summary	Manual Task	Deadline
External Tasks	Duration-only	Progress

G Periodic Progress Reports

H Self Assessment – Spring Term