ALGORITHM VISUALISATION

Department of Electrical & Computer Engineering
Part IV Research Project Report
2010
Title: Algorithm Visualisation
Project Number: 79
Author: Nathan Pitman
Project Partner: David Olsen
Primary Supervisor: Michael Dinneen
Secondary Supervisor: George Coghill

Declaration of Originality:

This report is my own unaided work and was not copied from, nor written in collaboration with any other person.

Name: Nathan Pitman

Final Report - Algorithm Visualisation Project

Nathan D. Pitman

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

Abstract

This final report discusses the Algorithm Visualisation Project spearheaded by Nathan Pitman and David Olsen. Algorithm visualisations have already proven themselves to be useful as tools for computer science education and visual debugging. The problem with existing algorithm visualisation tools is that they are algorithm-specific, offer relatively low modifiability and are functionally restricted. This report discusses in detail the algorithmic visualisation framework, Algovis, which has been developed for this project. Algovis provides algorithm visualisation for supported languages in a generic fashion, such that the user need only link it to their code in one place, in order to be presented with a visualisation during execution of their code. The background, requirements, design, implementation, evaluation and scope for future work are discussed for Algovis. This project represents a very promising start to Algovis.

1. Introduction

Software engineers in their final undergraduate year at the University of Auckland, are required to engage in a year-long project with a fellow pupil. This report details the Algorithm Visualisation project undertaken by David Olsen and Nathan Pitman.

1.1. Aim

The purpose of this project was to develop a tool which generically and automatically provides an animation of program code in execution. Specifically, the goal was to develop an algorithm visualisation framework which, with minimal effort from the user, provides a visualisation of the data structures and dataflow involved in their code.

It was intended that this tool be primary utilised for the visualisation of algorithms, but its potential as a general-purpose visual debugger was also considered.

1.2. Background / Motivation

Algorithm visualisation is useful in many fields of computing where visual abstraction aids understanding of code in execution. It plays a vital role in computer science education, and is especially popular with lecturers who desire to provide their students with visual representations of algorithms they are teaching. It is used to a lesser degree by experienced programmers who develop novel algorithms.

Existing digital algorithm visualisations used in computer science education generally take the form of animated GIF pictures (often used to visually depict sorting algorithms) or small applications (usually Java applets). The inherent problem with visualisations like these is that they are algorithm-specific and new visualisations must be specially developed for each new algorithm which is formulated, as they have a low level of flexibility and modifiability. They are restricted in that they allow their users only a small degree of control over resulting animations; users generally only have control over a few parameters to algorithms being visualised. Because these visualisations are created using different tools and languages, there is much work which is unnecessarily repeated between them.

The aforementioned animated GIF pictures and small interactive programs are mostly restricted to the realm of computer science education, as their low flexibility and modifiability hinder their usage for the visualisation of anything but the most established algorithms. As such, lesser-known, experimental or business specific production-code algorithms are completely neglected in this respect, whether they are incomplete or complete. However, debuggers are instrumental in the development of novel algorithms, and a subset of these debuggers can be construed as ‘low-level algorithm visualisers’. Indeed, the popularity of the Microsoft Visual Studio debugger can be partially attributed to the fact that it provides a more visual snapshot of program data than other debuggers. While debuggers provide a generic visualisation of program state which is available for any code, they are still deficient for the purpose of algorithm visualisation for the following reasons:

- They show as much of the program state as possible, rather than emphasizing the data structures which are crucial to understanding an algorithm.
- The view they provide is often more low-level than is desired and this makes them unsuitable for visualising algorithms which use complex

non-primitive structures, such as graph algorithms.
- They do not allow the user to backtrack through the algorithm during execution.
- They present static program state at discrete points in time and do not emphasize the transitions between states, the context of states, or the lifecycle of data.

There exist visual debuggers which improve on some of these points, but often at the cost of other factors including genericity and ease-of-use. Clearly, there is a need for a tool which provides automatic algorithm/code visualisation on a generic basis, offers a useful level of abstraction, is easy to use, and gives the user control over resulting visualisations. This project was centered on the creation of an algorithm visualisation framework to satisfy these requirements. While this project has ultimately aimed at providing visualisations of algorithms for anyone wanting to learn about them, it is not restricted strictly to algorithms as such, and can be considered as a general code visualiser.

1.3. Report Structure

The first section of this report is an introduction to the background and motivation behind this project and it lays out any preliminary knowledge which is required to understand the remainder of this report.

The second section of this report defines the requirements for the Algovis Framework, most of which have been objectives from the project inception.

The third section of this report details the architecture and implementation of the framework, as well as any design decisions involved.

The fourth section of this report provides an evaluation of the Algovis Framework in its current incarnation, as well as an evaluation of the software development process throughout this project.

Sections Five and Six discuss the conclusions of this project as well as the scope for future development of Algovis.

1.4. Preliminaries - Terms Used

- **Algovis:** The name of the algorithm visualisation framework.
- **Supported Language / Data Structures:** If a language is described as a supported language, it means that Algovis is capable of visualising a subset of data structures defined in that language. These data structures are referred to as “supported data structures”.
- **Primary User:** A user who writes code to be visualised by Algovis; i.e. a lecturer or general developer.
- **Secondary User:** A user who views an animation created by a primary user; i.e. a student.
- **User Code:** Code written by a primary user.
- **Introspection:** In the context of Algovis, introspection is the discovery of the state of a data structure present in user code, as well as the detection of any operations performed on these data structures and parameters to those operations
- **Data Source:** The component of the Algovis architecture which is responsible for introspecting user code. Generally written in the same language as the user code.
- **View:** The View is the component of the Algovis architecture to which a Data Source reports all introspected data. The View is responsible for displaying and animating this data and providing a user-interface.

2. Requirements

This section lays out requirements for the Algovis framework.

1.) The framework should not be biased towards particular algorithms. It should offer a suitable level of abstraction for all algorithms, provided that they are written using supported data structures.
2.) It should be as simple as possible for a user to have their code visualised, such that they need not annotate their existing code with much extra visualisation code. But if a user should desire finer-grained control over a visualisation, they should be able to specify this, either in code or through a user-interface.
3.) In addition to the above point, there is a fidelity requirement that the act of supporting an existing data structure in Algovis, should not dramatically alter its interface or size. Furthermore, the functionality of a supported type should not be altered by Algovis.
4.) Instead of displaying all available state as a debugger does, the tool should selectively display data structures such that screen space is prudently conserved, users are not subjected to visual overload, and the most important data structures are highlighted.
5.) The tool should provide animations which depict the transitions between the states of data structures.
6.) The tool should be flexibly designed, such that it can easily be ported to other languages, in

order to visualise code written in those languages.

3. Design

This section describes the architecture of the Algovis framework as well as various design decisions and implementation details.

3.1. System Overview:

At the inception of the project, it was realised that there would be two major tasks involved in the development of the Algovis framework. The first would be the introspection of supported data structures in user code, and the second would be the display and animation of data obtained from introspection. To this end, the software architecture was partitioned into a View component and a number of introspection components called Data Sources, as depicted in Figure 1 below. This modular design and separation of concerns means that a new Data Source can be developed for each language which is to be supported by Algovis, yet the singular View is generic enough to be used for all Data Sources, and modifications to the Data Source do not ripple across the entire system.

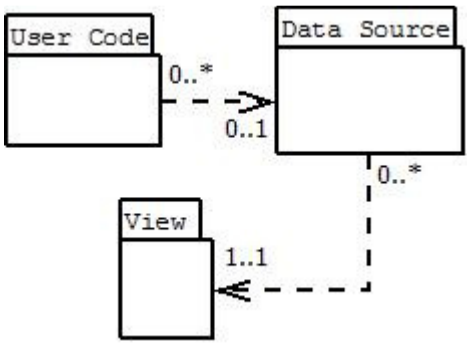


Figure 1: Algovis Architecture

The purpose of a Data Source is to perform runtime introspection on all data structures written in a particular language, by any means possible. Data Sources report the creation, destruction and any other operations performed on supported data structures to the View, which is responsible for visualising them. The View is a generic viewing component which allows for the display and animation of a set of generic data structures such as Array, Matrix, Single Printable, Custom Type etc. It also provides a user-interface for directly manipulating and controlling animations.

In order for a primary user to have their existing code visualised by Algovis, they need simply link it against a Data Source which supports introspection of code written in that language. Thereafter, when the user executes their code, all operations performed on data structures supported by that Data Source will be animated by the View.

The current incarnation of Algovis consists of a C++ Data Source and the View component (as discussed in 3.2-3.4). Originally, the user code and C++ Data Source were deployed in one process while the View was deployed in another process. The Data Source was to report introspected data to the View process via IPC (Inter-Process Communication). This communication model would have required the time-consuming development of a method for serialising functions calls and their parameters into a string format suitable for IPC. Instead, it was decided that the View should be deployed in its own DLL such that it would operate in the same process as user code, but have its implementation hidden behind a DLL interface and isolated from the rest of the system. Likewise, the C++ Data Source is also implemented as a DLL. Deploying components as dynamic libraries (DLLs in Microsoft Windows and shared libraries in Linux) is advantageous for several reasons:

- Libraries provide the modularity required in this system.
- Multiple dynamic libraries can be used in the same process and function calls can be made across library boundaries as if all code was in the same project. Remote procedure calls through IPC are considerably more unwieldy due to serialisation requirements.
- Any changes made to the implementation of the Data Sources or the View do not necessitate recompilation or relinking of user code. The user need simply overwrite the existing library file with the newer version.[1]
- Static libraries increase the size of binaries which utilise them whereas dynamic libraries are self-contained.[1]

3.2. C++ Data Source Implementation:

This section details the C++ Data Source which is used in the current incarnation of Algovis, and has been developed throughout the duration of this project.

3.2.1. How It Works

The mechanics of the C++ Data Source are best explained by the architectural sequence diagram in Appendix A, which depicts user code which is written in what appears to be standard C++ accompanied by the STL (Standard Template Library). But invisibly to the user, the standard int and vector types have been redefined as IntWrapper and VectorWrapper classes

respectively, which are implemented in the Data Source. Essentially, this C++ Data Source works by redefining each type which is to be supported, as a wrapper type which contains the actual type. Whenever an operation is performed on a wrapper, it is performed on the actual wrapped data structure, and in addition, the operation and any pertinent data are reported to the View for animation. Appendix A demonstrates the sequence of events resulting from the insertion of an integer at the beginning of a vector, which actually involves shifting all existing elements to the right. Because of this, a vector insertion can involve extra copy-construction and copy-assignment operations, all of which are detected by the omniscient Data Source. However, the Data Source only reports a single high-level atomic insertion operation to the View for abstraction purposes.

As partially depicted in Figure 2 below, there is an implicit mapping from supported types, to wrappers to generic data structures in the View (referred to as Viewables). All wrappers for C++ primitive types are mapped to a generic Single Printable type in the View as they can all be drawn in the same fashion with their values represented by strings. Wrappers for both the standard vector and double-ended queue types currently map to the View's array type, as both can be drawn as generic arrays.

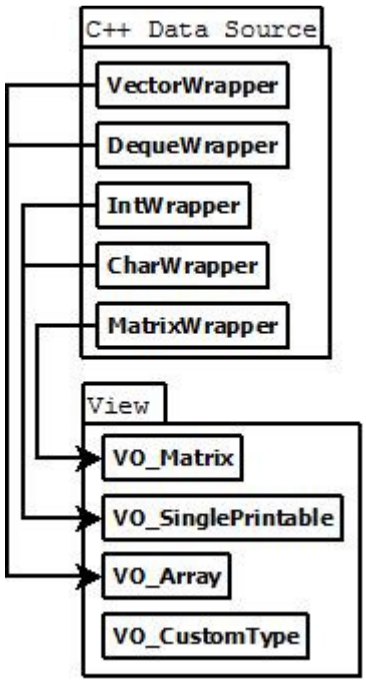


Figure 2*: Mapping from C++ Wrappers to Viewables*

The Data Source maintains information about every wrapper which is instantiated during execution and allocates each one an ID. This affords a higher level of abstraction than communicating about wrappers in terms of memory addresses as it allows for special cloning behavior where wrappers can be 'transplanted' to new memory addresses, yet be treated as the same wrappers by the View. This type of behavior is desirable as the View mostly isn't concerned with lower-level implementation details such as elements of a dynamic array taking on new addresses when it is reallocated (possibly due to an insertion). The View is only concerned with high-level operations performed upon wrappers, not with any temporary variables created in between, which are invisible to the user.

3.2.2. Evaluation of the C++ Wrappers Solution

The wrappers solution for the C++ Data Source has worked fairly well up until this point. In accordance with fidelity requirements (Requirement #3), the wrappers appear almost identical to the supported types that they wrap as they implement the same interface and are identically sized.

The Microsoft Visual Studio and GCC compilers do not object to the redefinition of primitive POD types (Plain Old Datatype) as complex class types (i.e. redefining int as IntWrapper). The exception is when primitive types are explicitly required by the C++ standard. One example of this is non-type template parameters, for which the type must be primitive. Another example is union types, which may only house primitive types. These exceptional cases generally only arise in external C++ libraries (such as the Boost C++ Library [2]). There are various workarounds for these cases such as altering the include order of the Algovis and external library header files, or explicitly using the REALINT type which is a type alias for an actual primitive integer type (not IntWrapper). Another limitation is that native C arrays cannot be fully introspected, as they do not present a single token 'type' which can be redefined in the same way as regular primitive types.

C++ was chosen as a substrate for introspection by the first Algovis Data Source as it was decided that its power and relative lack of restraint would render it more highly amenable to introspection.

3.3. Data Source Alternatives:

When developing a Data Source for a language, it is important to consider its characteristics and how they could potentially lend themselves to introspection. The C++ Wrappers solution is only one approach to introspection, and some other approaches are outlined below:

3.3.1. The Valgrind Approach

Valgrind is a framework which is used for writing program analysis tools such as memory leak checkers

and cache profilers. It works by translating user code into IR code (Intermediate Representation) and performing dynamic binary instrumentation on it (i.e. injecting code at runtime).

Early on in the project, the feasibility of the Valgrind approach was explored. This involved developing a Valgrind tool (a tool which uses the Valgrind framework) in order to introspect supported data structures at runtime, at a similar level of abstraction to a debugger. The advantage of the Valgrind approach is that it is theoretically language independent, so it could be used for any languages which are not amenable to other forms of introspection. The major disadvantage is that the IR code tends to be too low-level to be useful; it would be difficult to reconstruct a high-level picture from the data it yields. Furthermore, Valgrind tools cannot be linked against the C standard library and this limits their use.

3.3.2. The Polymorphism/Inheritance Approach

The Polymorphism/Inheritance approach is similar to the Wrappers approach, but instead of defining a wrapper type which encapsulates a type to be supported, it involves defining a type which inherits from the type to be supported. The inheriting subclass type would override every operation (public method) offered by the supported superclass type with methods which call their superclass equivalents before reporting details of the operation to the View. Due to polymorphism, an object of the subclass type can substitute for an object of the parent class type in any user code.

This approach is restricted to OOP (Object-Oriented Programming) languages which represent all types as classes, and allow all methods in supported types to be overridden.

3.3.3. The Reflection/Hooking Approach

C++ does not offer full runtime reflection of data, but it could be leveraged for developing Data Sources in other languages which do support it. But it is not sufficient to merely use structural reflection to discover the state of an object of a supported type. The Data Source must be aware of all operations which are performed on supported types, at the time they occur. This can be achieved through hooking into functions (intercepting their invocations). This option has not been fully explored, but it is understood that operating systems, virtual machines (e.g. the Java VM) and various interpreters support hooking and code instrumentation.

3.4. The View Component:

The View component, as depicted in Appendix B, is responsible for displaying and animating data structures in a meaningful way, according to the introspected data received from a Data Source. The basic implementation details of the View are discussed in 3.4.1, and the dataflow analysis and ActionBuffer components of the View are discussed in 3.4.2 and 3.4.3. Finally, the user-interface functionality provided by the View is discussed in 3.4.4.

3.4.1. Basic View Implementation

By default, the View displays all data structures except for primitives types. The rationale behind this is that primitives often tend to be unimportant temporary variables (of which there are many), and therefore, do not tend to play a big enough part in an algorithm to be considered crucial. Of course, this is not always the case, and if the user should desire to suppress the display of a non-primitive or enforce the display of a primitive, they can specify this in their code. The View is deployed as a DLL written in C++ using the QT library for GUI components.

3.4.2. Dataflow Analysis

In order to animate data moving in between data structures, it is necessary for a HistoryManager to maintain a history of each Viewable so that dataflow analysis can be performed. For example, Figure 3 below depicts the addition of three elements in one array to obtain a result which is assigned to an element in a different array. Many temporary variables are involved in this operation, but are not displayed by default as explained in 3.4.1. In order to be able to show an animation where the values 1, 9 and 4 contribute to the new value replacing 7, it must be possible to track through multiple operations documented in the history of temporary variables which are not even displayed. The HistoryManager plays this role in the View.

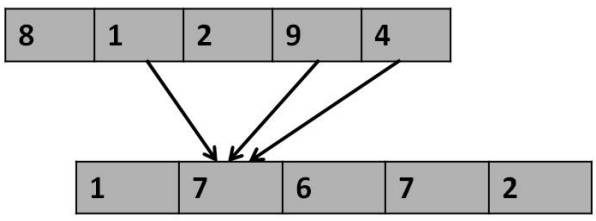


Figure 3*: Addition of Three Array Elements*

3.4.3. *Actions and the ActionBuffer*

The View's unit of work is an action. Most operations on supported data structures are reported to the View by the Data Source and result in the creation of an Action, whose role is to update the relevant Viewables and possibly animate the operation. This is an asynchronous process as Actions are not necessarily performed immediately; they are added to an ActionBuffer for reasons discussed below. If the ActionBuffer is large enough, the user code could technically finish executing before a single Action is animated. Examples of Actions include the creation and destruction of data structures, assignment of the value from one primitive to another, the highlighting of operands in a comparison operation, the insertion of an element in an array etc.

Displayed in Figure 4 is user code which swaps two array elements using the temporary variable swap algorithm [3], and the resulting Actions in the View. If the View were to display the temporary variable, it would be fairly obvious to the user that a swap was occurring, based on the three circular assignment animations which would be performed. However, since temporary variables are not displayed, the resulting animation(s) should involve only the two array elements. In this case, the first assignment would be muted, the second assignment would be animated as normal, but a conflict would be detected in the animation of the third assignment. The HistoryManager would track the temp variable back to array[1] and observe that its value had been altered since the first assignment, and therefore, it is fresher than the temp value used in the third assignment. Therefore, displaying data moving from array[1] to array[0] would be erroneous, but displaying both assignment actions simultaneously (from array[0] to array[1] and array[1] to array[0]) would be correct and meaningful. The purpose of the ActionBuffer's existence is to buffer Actions prior to animation, so that these dependency conflicts can be detected and conflicting actions can be combined and animated simultaneously.

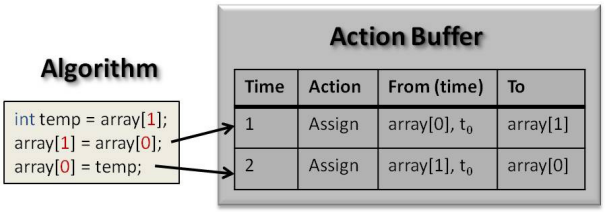


Figure 4*: Addition of Three Array Elements*

3.4.4. *The View User-Interface*

The View presents a user-interface allowing a user to perform the following operations:

- Moving Viewables around on screen, labelling them, suppressing their display and zooming in and out on them
- Pausing, resuming, toggling and skipping through actions animations
- Rewinding back through previous actions.

4. Product/Process Evaluation

This section discusses evaluation of the Algovis framework as well as the software development process which created it.

4.1.1. *Algovis Evaluation*

Formal evaluation has not been performed on Algovis due to time constraints and the relative scarcity of undergraduate students knowledgeable enough in C++ to write STL algorithms. Various classmates have informally evaluated Algovis demos as secondary users and positive feedback has been received from all of them. There is certainly scope in the future for formal evaluation of Algovis, perhaps when it is more mature.

A series of demos has been developed to evaluate the capabilities of Algovis. Some of these test established algorithms including recursive merge, merge-sort, bubble sort, and other basic array algorithms such as summation, Fibonacci calculation, cumulative addition and Moving Average.

4.1.2. *Evaluation of Software Development Process*

This project has not strictly conformed to a software development process, but the workflow has leaned towards an iterative prototype-based process. Owing to the modular architecture of Algovis, it has been fairly simple to divide the workload between two people. Despite comprehensive debugging facilities, due to the complexity of long sequences of events and the difficulty of debugging asynchronous actions, it has sometimes been challenging to track down bugs at various stages of Algovis development. In order to mitigate this, unit-testing and the non-automated execution of a series of demos has been used to reveal as soon as possible any faults which have been introduced by changes to the code.

5. Future Work

There is wide scope for future work on Algovis and possible improvements include:

- Automatic Detection of Names for Data Structures:

  Viewables are currently labeled with their memory addresses by default as C++ lacks the reflection capability to extract the name of a variable. While,

the user is given the power to manually set the label of a Viewable, Data Sources for other languages might be capable of automatically introspecting variable names.

- Animation Recording:

  In the future, the View could offer a feature to save an algorithm animation as a video file.

- Seamless Integration into Large Projects:

  As mentioned earlier, the C++ Data Source clashes with various external libraries. As there is a high probability that large projects use these libraries, it is difficult to integrate Algovis into them. This hampers the potential usage of Algovis in production code. Other Data Sources are not expected to suffer from this problem.

- Improved Usability as a Debugger:

  Once Algovis can be seamlessly integrated into production code, its potential as a visual debugger can be fully realised. This could lead to new features and requirements for Algovis, such as a scoping feature whereby data structures which are out of scope can be suppressed, and the requirement for bidirectional communication between the Data Source and View, to allow users to manipulate data structures in their code, through the View.

- Statistics Module:

  The current View is only one possible view of the introspected data provided by Data Sources. Another possible view could take the form of a statistics module which records and displays pertinent statistics for each data structure. For example, when comparing sorting algorithms, a user may want details about the number of array reads/writes for each algorithm implementation. A statistics module could potentially predict the average case performance for an algorithm.

- Debugger Integration:

  It would be possible but difficult to integrate existing debuggers into our framework in order to obtain extra data to supplement the data introspected by Data Sources. This would be especially useful where a Data Source for a language is fairly comprehensive in its introspective capability, but lacks particular data which could be provided by a debugger. With regards to the C++ Data Source, a debugger would be able to obtain variable names and scoping information for data structures as required for Algovis to be used as a higher-level visual debugger itself. A debugger can also be used to intercept function calls, as is required for the Reflection/Hooking introspection solution described in 3.3.3. The drawback of relying on debug symbols is that the framework would become not only platform-specific, but compiler-specific.

6. Conclusions

The Algovis Framework represents a generic, flexible and innovative framework which can be used effectively in computer science education. The current incarnation of Algovis is capable of animating C++ code which uses primitive types, vectors, maps, deques and matrices. It would be possible to port the Data Source to another language which is more amenable to introspection without requiring changes to the view. This could see Algovis' potential as a visual debugger fully explored.

No other animation tool works for such a wide range of algorithms. A user is able to write any code that comes to their mind, and the Algovis tool will animate it in some fashion, while giving the user full control over the visualisation produced.

There is broad scope for future work on the project which could see it being ported to other languages, being ameliorated through added support for more data structures and their animations, and having its use-case as a visual debugger further investigated.

Acknowledgements

I would like to acknowledge my supervisors, Michael Dinneen and George Coghill, for their support and guidance throughout this project. And I would like to thank my project partner, David Olsen, without whom this project could not have been a success. Appendices A and B, and Figures 1, 2 and 4 are project artefacts, resulting from our joint collaboration prior to this report.

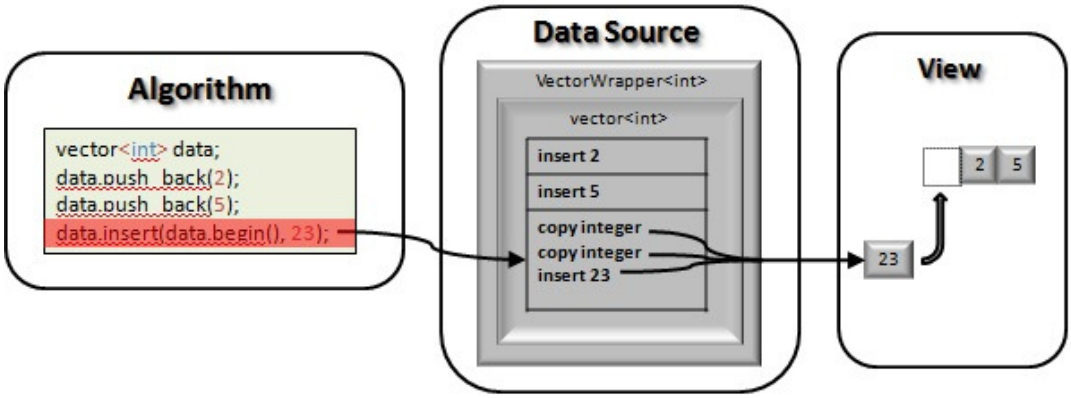
7. References

[1] Silberschatz, A. et al. (2009) Operating System Concepts. Eighth Edition. John Wiley & Sons, Inc. pages 321-322

[2] Boost C++ Libraries. Retrieved September 12 from http://www.boost.org/

[3] Swap (computer science). Retrieved September 12 from http://en.wikipedia.org/wiki/Swap_(computer_science)

Appendix A



Appendix B