Working Paper Series
ISSN 1170-487X

Visible-C
A Simple Visualisation For C Data Structures

by: William J. Rogers

Working Paper 96/10

April 1996

VISIBLE-C
A SIMPLE VISUALISATION SYSTEM FOR C DATA STRUCTURES

William J. Rogers
Department of Computer Science,
University of Waikato, Hamilton, New Zealand.
w.rogers@cs.waikato.ac.nz

Introduction

I teach half of a second year "Programming with Data Structures" course in which students learn to program with dynamic memory and build linked data structures. In this paper I describe a piece of software, VisibleC, which I have developed to assist in teaching these concepts.

The VisibleC system reads C++ declarations and displays declared variables graphically, showing record, array, and pointer structures. Students may enter C++ assignment and allocation statements and see the effect of their execution. Statements can be read from files, typed interactively, or built by pointing and clicking on the displayed structures.

The paper is organized as follows: Section one describes the teaching context. In section two I describe an earlier piece of software I have used for teaching pointer concepts, explain where this work fits in the software visualisation field, and discuss some of my design considerations. Section three covers the VisibleC program, and has a little on its implementation. In section 4 I comment on the results I have had in using the system, and give results from a small survey of student reactions. In section 5 I present my conclusions and intentions for further development.

1. Teaching Context

In the Waikato University Computer Science Department we teach introductory programming to our first year students in two half courses. The half courses are strands in our first semester "Introduction to Computing", and second semester "Introduction to Computer Science" courses respectively. We use C++ as the principle programming language, although it might be more accurate to say that we teach Pascal, using C syntax (stealing a little from C++), and a C++ compiler. By the end of their first year students have learned the equivalent of the Pascal language without fully covering pointers. The idea of a pointer and of a list structure has been explained in lectures, but there have been no programming exercises. Students have also worked on algorithm and some data structure material. That is covered in the non-programming strands of the first year courses [Holmes et al 96].

The current first year arrangement was first introduced in 1995. It was in the change made at that time that we moved to using C++ as our introductory programming language. The reasons for using C++ are worthy of some comment.

During the 80's, like nearly everyone else, we taught students to program using Pascal. By the end of the decade we found ourselves wanting to change programming language. The main motivation for change was the desire to use a language which supported data abstraction and promoted good organisation of programs which were too large to handle as a single file. This was important to allow access to libraries of interesting software in the introductory course and to pave the way for the development of larger pieces of software in later years. With two colleagues I wrote an introductory text [Hopper et al 91] on programming in Modula-2. The book places particular emphasis on program structure and organisation, and took advantage of the module facility in the language. We wrote a library of software to accompany the book, which allowed students access to graphics from their first practical sessions, and later let them experiment with windows and mouse input.

The Modula-2 book was used with good results for four years. When we first began writing the book, Modula-2 had seemed a promising candidate for becoming one of the world's principal programming languages. A major standardisation effort was underway, and it was in use in New Zealand for embedded system work. By 1994, however, it had become obvious that history had consigned Modula-2 to the list of also-rans. Whilst we were still satisfied with it as an introductory programming language it began to cause difficulties later in the curriculum. Senior students programmed on UNIX systems, for which we could not obtain a satisfactory Modula-2 compiler. Staff wanted them to use C, and started to devote time in their courses to teaching it. Our four year degree program places strong emphasis on an honour's project in the fourth year of study. We found that most of the programming projects undertaken required C or C++, either because they involved developing and extending some existing software, or because they interfaced to some C/C++ library code.

I was tempted to quote the adage that "Students taught using a good language will be better programmers ..." and recommend that we continue with Modula-2. The only difficulty was that I didn't believe it. I knew perfectly well that even quite capable students seemed to find the step from Pascal or Modula-2 to C difficult. In particular, students found it very difficult to learn and use C/C++ during their honour's project. So although we engaged in some debate, as befits a decision usually beset by feelings of almost religious fervour, no-ones heart was really in it, and we quickly settled on teaching C++ from the outset. We do not claim that it is an especially good decision. C++ is not an ideal first programming language—just a pragmatic choice, giving in to the fact that C/C++ is now the language most commonly used for software development and the language we need students to know for advanced courses. A cost of the decision was that extra teaching time would be needed help students cope with details of the language.

We had observed the difficulty our senior students most often had with C was to do with memory allocation. C provides the programmer with considerable freedom in the declaration of pointers, allocation of storage and manipulation of addresses. It seemed that we had protected students too much when teaching Modula-2. They had not developed sufficient awareness of the way in which memory was allocated and accessed. They tended to rely on dynamic checking of array bounds, and not reason carefully about their memory allocation when writing programs. Rather than burden the first year courses, we carefully restricted first year teaching to a 'safe' subset of C, avoiding any use of pointers by using the C++ syntax for reference procedure parameters, and for text input. Laboratory demonstrators are aware that students are likely to have programs crash in odd ways due to array bound overflows, and assist as necessary. We then planned to put in additional material at second year help students develop sufficient understanding and awareness of memory allocation and access to cope with the full C/C++ language.

Accordingly the second year courses have been revised for 1996, to follow on from the 1995 first year arrangements. Students now take two programming papers. The first semester course "Programming with Data Structures" completes the introduction to dynamic memory use, and covers a restricted 'data structures' syllabus. It also introduces students to a functional programming language (we use Gofer, a Haskel subset). The second "Program Structure and Organisation", covers file structures and object oriented programming.

The "Programming with Data Structures" paper sets out to address the difficulties of programming in C/C++ in two ways. The first is the direct and obvious one. We are allocating quite a lot of time to memory, allocation, de-allocation, and the building of linked structures. The second is an attempt to enforce a 'decoupling' of data structure design from implementation. Students are asked to design and prototype data structures in the functional language Gofer, which is high level enough to hide issues of memory allocation and access. They then implement in C/C++.

The final innovation in our second year program is a negative one. Having the Gofer material and the additional C/C++ material in the course has squeezed out a good deal of the traditional content of a second year data structures course. Students will still see lists, trees and hash tables, but we will not discuss balancing binary trees, or B-tree structures, etc. until our third year algorithms course. There was some opposition to this change, but I take the view that it may be beneficial in its own right. We have observed that our honours students have been quite reluctant to build linked data structures in their programs, or at least think that it is a significantly adventurous thing to do. I suspect that the reason is that by teaching about linked structures in the context of algorithms like tree balancing, we have given them the impression that such algorithms are always required. They had not had the chance to develop confidence with simple linked structures first.

This then is my teaching task and context. I have students who have met the idea of a pointer, but not programmed using the idea. My role is to teach them about the use, design and implementation of elementary data structures. I have a curriculum in which I have time in which to slowly and carefully cover the issues involved in reliably building linked structures in the C/C++ language. I am not under pressure to teach a complete traditional data structures course.

2. Background

When designing teaching support software it is easy to concentrate on the expected gains, and neglect the intellectual overhead of learning to use and understand the tools themselves. We found this to be a difficulty, for example, with the libraries we provided with our Modula-2 system. They did make it possible for students to use graphics at the outset, but they also constituted an overhead. Students needed to learn the specifics of those libraries and we needed to use lecture time in explaining them. In writing software to demonstrate pointers and their use I have been very conscious of this pitfall and tried to write programs with very minimal user interfaces and only the needed functionality.

The VisibleC program evolved from a simple demonstration program (Pointers) I wrote some time ago for use in teaching the small introductory section on pointers we included in our first year course at that time. Pointers was first written for Pascal and later modified to use Modula-2 syntax. It assumed the declaration of a simple record structure for holding names and phone numbers.

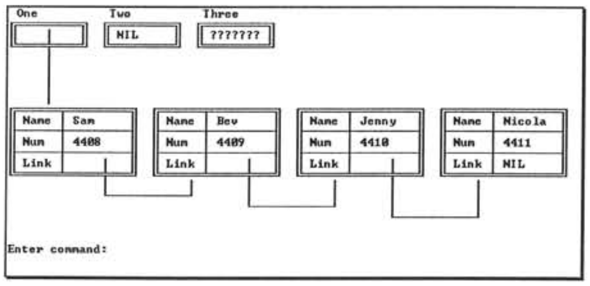
```
TYPE nptr = POINTER TO node;

     node = RECORD
       Name: ARRAY [1..7] OF CHAR;
       Num:  CARDINAL;
       Link: nptr
       END;
```

and three variables

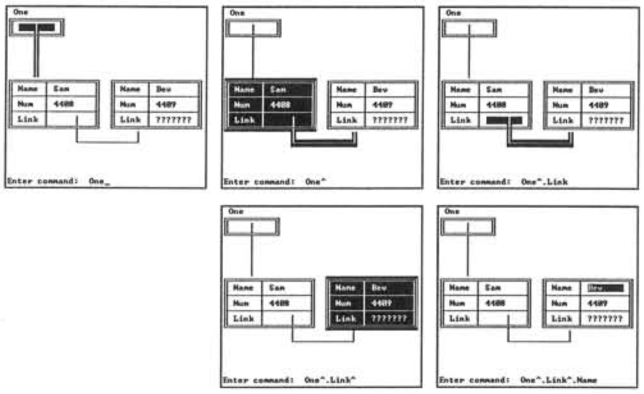
```
VAR One, Two, Three: nptr;
```

The user of the program could enter 'NEW', 'DISPOSE', and assignment statements interactively at the keyboard. Correct statements were immediately executed, updating the semi-graphic display. In the practical session which used this program students were asked to enter statements to allocate, link, and fill nodes so as to build a linked list of four elements.



```
Enter command:
```

The educational objective was to allow students to learn and become familiar with the syntax for accessing data structure components. It was therefore appropriate that they should type statements. To minimise the kind of frustration the novice programmer experiences in the typical edit/compile cycle the program used a hand coded *instant* parser—which detected errors as soon as a single incorrect character was typed. The parser was written rather like a recursive descent parser, except that the parsing algorithm worked down to the character level. Input strings were reparsed at every input keystroke. Incomplete tokens at the end of the current statement were ignored.



```
Enter command:  One_
Enter command:  One^
Enter command:  One^.Link
Enter command:  One^.Link^
Enter command:  One^.Link^.Name
```

After each parse, the diagram of the data structure was updated to highlight the part of the structure currently being referenced. For example, in typing `One^.Link^.Name` the highlighting goes through the sequence shown.

The very restricted programming context made it possible to issue highly specific error messages as soon as an incorrect character was entered. Thirty five distinct error conditions were detected and reported—providing good guidance to the student operating the program.

The fact that the program displayed at most four nodes provided an unexpected bonus. If a node was 'orphaned', by losing its pointer, there was no way that access to it could ever be regained, and no way that the 'memory' could ever be recovered. The demonstrated quite forcibly the need to keep careful track of allocated memory.

From the teaching viewpoint I think that this program worked well within its limited objectives. It did not do anything without explicit instruction by the learner, and it responded in some way to each action. The only special command to learn was 'quit'. Otherwise it accepted standard Modula-2 statements. Tutors and lab demonstrators reported that students responded favourably to the program and appeared to understand the concepts involved. As we did not ask students to write programs using pointers in that course, we never determined whether the skills gained with the graphic system generalised to ordinary programming activity. However, the narrow goal of teaching how to access data structure components seemed to have been achieved. Students performed well on such exam questions. The graphic interface was very helpful in the laboratory. It is certainly much easier to discuss a programming problem with a student when there is a graphic display of the results of their actions in from of you both. It is also very helpful to be able to conduct experiments quickly.

I felt that my second year class would also benefit from using a program which allowed them to visualise their data structures. The educational objectives were extended though. It was important to reinforce the mechanics of data structure access, but now necessary to show just how C declarations allocate memory, and how a variety of data structures could be assembled from linked components. It was important that the new program accept and process a variety of data structures, so that students could experiment with their own examples.

There is a large literature in the area of software visualisation—using graphics (and even sound) to convey to the programmer a mental image of a program and/or its behaviour. In 1994 it was reported [Price et al 94] that over 100 software visualisation prototypes had been built. They vary from 'pretty-printing' programs [Hueras et al 75] [Baecker et al 90], to a system which attempts to infer and display abstract interpretations of data structures [Henry et al 90]. A great deal of this work has been directed towards program/algorithm animation—systems which attempt at various levels of abstraction to display the dynamic behaviour of programs.

A taxonomy of the area was first proposed by Myers [Myers 86&90] and refined by Price, Baecker, and Small [Price et al 94]. Systems are distinguished by their: *Scope* the range of programs and systems they will run on; *Content* the aspects of the program that they exhibit; *Form* the nature of their visualisation; *Specification* extra commands, if any, required to describe the visualisation; and *Navigation* the means and nature of the interaction they permit.

Stated in terms of the taxonomy my requirements are as follows. Scope: a small number of declarations (less than 100 lines), but a need to cope with data structures whose display might not comfortably fit on one screen; Content: data structures only. Form: graphic presentation of data structures as understood by a C programmer—I did not need details like address and size of memory areas used, nor were any abstract views required. Specification: should require no commands additional to C declarations and statements. Navigation: the user would need to pan over large displays, and scale down or up to allow broad or detailed viewing.

Of the systems described in the literature, the one closest to my requirements was one of the earliest—Myers' *Incense* prototype for displaying Pascal data structures [Myers 83]. Incense is effectively a debugger which allows a programmer to graphically explore the data structures of a running program. A 'display' command is used to request a drawing of a variable and its linked structures. The default display form was a direct rendering of Pascal basic values, records and arrays using nested and stacked boxes, although provision was made for the programmer to supply alternative display functions to obtain more abstract views. Considerable efforts were made to make the system lay-out linked structures in a visually sensible manner, including the use of curved lines to depict pointers. The basic display forms of Incense were just what I required. However, Incense did not allow entry of statements or changes to data. I wanted students to be able to enter statements interactively and see their effect immediately.

Incense was developed into a production system called Amethyst [Myers et al 88] and later integrated into a commercial Pascal system known as Pascal Genie [Changhok et al 91]. Pascal Genie supports a structure editor and the graphic visualisation facility as parts of an integrated environment. It has been used in educational settings with good success [Goldenson 89].

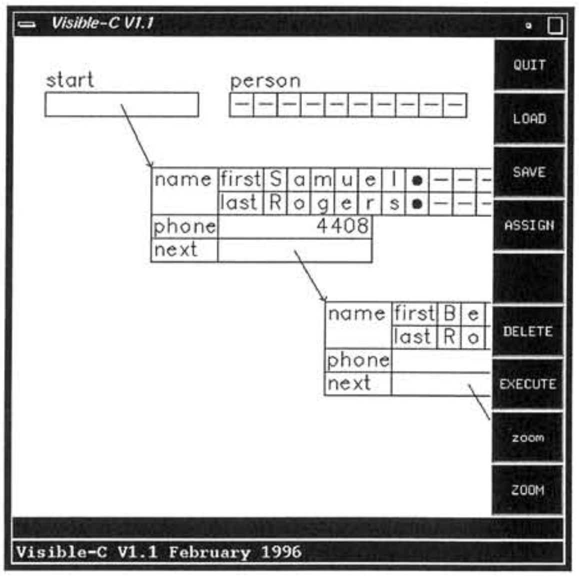
In comparison to these systems, my requirements are very modest. I do not discount the advantages of having students work with advanced environments, but I am wary of introducing systems where the intellectual overhead of learning the system is high. Such overhead could only be justified if the students could go on using the systems in later years. At present my task is to teach students to be competent in using a conventional text only programming system. Our decision has been to try to teach the more abstract view of data structures using Gofer, and to concentrate our C/C++ teaching on concrete implementation details.

In examining other systems I did decide however that there was a benefit in including a facility for navigation over data structures, allowing construction of statements by point and click, would be a useful addition. The idea here was to help students to map between graphic and textual presentations, by allowing statements to be entered either way, and having the system construct the alternative form. The Pointers program accepted text and showed by highlighting the graphic interpretation. I wanted the new program to work both ways.

In summary then I wanted my new system to extend my old Pointers program to accept arbitrary C declarations, to provide a capability for view navigation over a larger display, and to allow point and click statement construction. The features I wished to retain were the absence of a command structure beyond the C language itself and the high level of interactivity.

3. VisibleC

VisibleC is written in C, using X windows and the Athena 3D widget set. The computers used in the lab are Pentium PC's running Linux. They have sufficient processing power available to permit redrawing of the entire screen on any change. As a result the graphics programming was quite straightforward.



VisibleC is usually started from a command shell, with the name of a file holding type and variable declarations, initialisations and statements. On start-up it displays a window with command buttons and a view of the data.

The command buttons are: *QUIT* to exit the program; *LOAD* to load a file of declarations or statements; *SAVE* was intended to save the current 'machine' state but has not yet been implemented; and *ZOOM* & *zoom* for magnifying and shrinking the display. The display can be panned by 'grabbing' the view window background and dragging it. Panning and zooming can also be done with the arrow keys and +/- keys on the numeric keypad respectively. The remaining buttons are used when building statements.

To begin an assignment statement the user must press the *ASSIGN* button. To begin a `delete` statement the user must press *DELETE*. In either case they may then click on the data view to build an object reference. As components are clicked a C statement is built and displayed near the bottom of the window. The data structure is also highlighted to show which component is currently being referenced. The assignment statement may be completed with a new clause or a second reference. Literals are entered using the keyboard. All keyboard entry is captured by the program line widget, and at any stage the user has the option of continuing their statement using the keyboard. Pressing the *EXECUTE* button runs the current statement.

There is a single parser used for both file and keyboard input, and it can accept arbitrarily ordered declarations and statements, providing only that a declaration before use convention is followed. Declarations accepted are typedef's or variable declarations. Modifiers (static, unsigned, etc.) are not supported. The C++ style of structure declarations is used as that is what our students have been taught. I.e.: the structure name always becomes a type name. The basic types recognised are `char`, `int`, and `float`. Array, pointer and structure declarations are supported with standard C syntax. The program does not provide union or enumeration declarations. The full C syntax for initialisation of variables is supported, including nested structures and arrays, with the exception that basic items must be literals. The system cannot evaluate expressions.

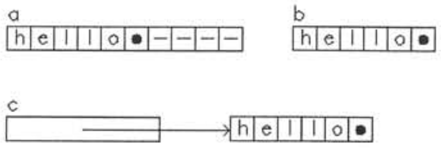
The statements supported are assign and delete. The right hand side of an assignment can be a `new` clause, a variable reference or a literal. In choosing `delete` for deallocation and `new` for allocation we followed the C++ syntax, rather than C usage. Again the reason was because this was what our students had been taught. Assignment is treated as a statement, not an operator, and there is no support for expressions. Array indices must be integer literals.

Type compatibility was generally treated rather liberally. It is possible to assign string literals to character arrays regardless of size. This provides convenience for interactive use. Array and record assignment is permitted. Pointer compatibility rules, however, are stricter than in a proper C implementation. A pointer cannot point to a component of something, only to an entire variable or allocated object. In this respect VisibleC behaves more like Pascal than C.

The data view uses colour to distinguish objects of different kinds. Variables are cyan, objects allocated in the heap are grey, and string literals are blue. The effects of subtly different declarations can be observed. For example the declarations

```
char a[10] = "hello";
char b[] = "hello";
char (*c)[] = "hello";
```

generate the following display.



The algorithm for arranging objects on the screen is simple. When a new object is to be placed, VisibleC searches for gap to insert it into. The search is performed in a left to right, bottom to top order over a rectangular search area. When the program first starts, the rectangle is that displayed in the program's window. Whenever a search for space fails the rectangle is enlarged by 50% to the right and bottom. When a `new` is executed, the search for a place to put the newly allocated heap object begins with a rectangle whose top left corner is on the pointer cell receiving the new pointer. For simple examples the result of this strategy is to place variables across the top of the display and to draw lists downwards to the right.

Automatic placement of objects is not always successful. The system allows the user to rearrange the display to their own satisfaction. Objects may be grabbed and dragged about with the right mouse button.

Point and click statement construction must be done in the same series of operations as are required in the C text. This was deliberate. It would have been possible to allow the user to click on any component of a data structure and have the entire C access text generated. However, that would have taught students very little about the C language. For example, to generate the phrase

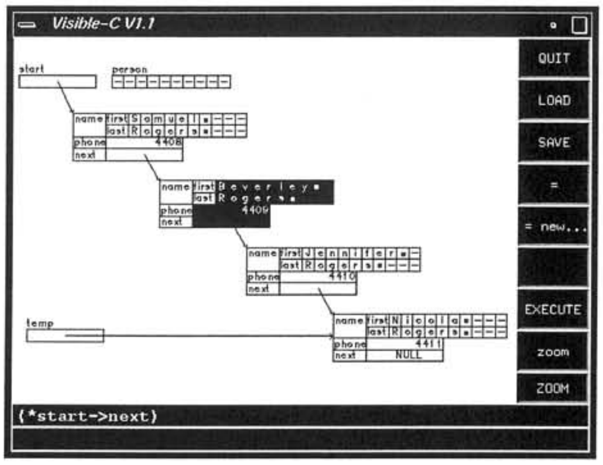
```
start->next->name.first[2]
```

the student must complete seven steps.

1. Click on `start`: highlights the contents of the variable including the arrow representing the pointer.

2. Click on the arrow to access the target of the pointer. The whole target record is highlighted. At this stage the generated code is `*(start)`.

3. Click on the 'next' field of the highlighted record. Now only that field (and outgoing arrow) is highlighted and the generated code is changed to `start->next`.
4. Click on the arrow.



```
{*start->next}
```

5, 6, 7. Click on `name` field, `first` field, and finally the third element of the character array to select the character cell holding the 'v'.

The syntax of C is unhelpful in this example. I wanted to emphasise the step of dereferencing the pointer by having students click on the arrow. It is a pity that the C statement does not grow linearly with the access path. In complex accesses the jumping between the `*()` and `->` notations is quite irritating. One possibility for improving the interface would be to allow direct clicking on a pointer field, corresponding to the `->` operator. It would still be necessary to click on the arrow (explicitly dereference the pointer) in the case of pointers to objects other than records.

VisibleC does very strict run time checking. Variables which have not been initialised are displayed with question marks in their value fields. Attempts to dereference an undefined pointer are trapped, as are attempts to use a NULL pointer. Copying an undefined value is however permitted. Array bounds are also checked.

The current implementation is not complete in some respects. I have not managed to provide such detailed and useful error messages as for the Pointers program. The parser was constructed using BISON (GNU YACC clone) and is not suitable for use on partial statements. This means that it cannot be used in 'instant' mode like the parser in Pointers. As a result the display does not highlight while a statement is being typed. I felt that the point and click statement entry provided a sufficient alternative for those unsure of C.

There is not the extent of interaction between the keyboard and graphic statement construction that I would like. It was intended that statements begun on the keyboard could be extended by point and click. As the implementation currently stands, once keyboard entry starts, it must be used to finish the entire statement. This was also a consequence of difficulty with the parser. Finally the system does not properly check the type specified in a 'new' phrase.

4. Results

At the time of writing I have used the program for an introductory laboratory session, designed to review the mechanics of pointer access and to get students to the point of building their own first linked list program. They worked through a staged exercise, first trying accesses to a variety of data structures, and then experimenting with the code fragments they would use to add nodes to, and traverse, their linked list. All but 10 of the 135 students attempting the exercise managed to write the C program which built the linked list. My intention during the remainder of the course is to encourage students to test data declarations and experiment with code fragments on the system, prior to writing C programs.

As a preliminary evaluation I surveyed the students after their first exercise. Each student was asked whether they thought the VisibleC program had helped them, whether they found the user interface acceptable, and what changes or improvements they would like. Of the 74 students asked, 73 filled in the form.

Question 1: *In learning about pointers and linked lists I found the VisibleC program to:*

Be very helpful	45
Be helpful	25
Make no difference	3
Be unhelpful	0
Be very unhelpful	0

This is very encouraging. Nearly two thirds of the students found the program very helpful, and none thought that using it effected them adversely.

Question 2: *I found the VisibleC program to be (choose a point on the scale):*

Simple and intuitive to use	15
...	33
...	19
...	6
Clumsy and awkward to use	0

Whilst the responses are generally favourable, there is clearly room for improvement here. There were aspects of the operation of the program which were not satisfactory.

Question 3. *Are there any aspects of the VisibleC program which you feel could be improved or changed, or are there any additions you would like to see made?*

Of the 73 participants, 43 gave written comments to this question and one student asked for a meeting to explain his views. I found it very pleasing that so many took the time to make constructive suggestions. A brief summary of those comments follows.

4 comments requested correction of problems with C syntax: 2 wanted the type properly checked in a `new` clause; and 2 wanted semicolons to be compulsory on statements entered interactively.

22 of the comments requested corrections or minor improvements. Of these: 8 were about command entry, and the relationship between keyboard and graphic statement entry; 2 requested better file load command entry; 1 asked for the addition of a command to delete all data; 8 asked for more documentation and better error messages; 3 wanted improvements in visual appearance

2 people reported serious problems with the program: one found that learning to use the program took too long; and the other had their work plan disrupted by a fault in my installation of the program which meant that it hadn't worked on the day the assignment was first issued.

18 can be classified as requests for new capabilities: 3 of these were for some output trace of commands entered; 3 for a built-in editor for declaration files; and 12 for a fuller implementation of the C language to be interpreted by the system.

4 concerned point and click manipulations: 1 wanted pointer assignment by direct manipulation (i.e.: to be able to assign pointers) by dragging from pointer to target); 1 wanted to be able to shortcut the process of building data structure accesses by clicking immediately on the required component; and 1 explicitly objected to clicking on the arrow to dereference pointers.

5. Future Directions

There are a number of changes I would like to make to the VisualC program. The most important is to replace the LALR parser it is using. Faults in the current parser include the limited nature of its error messages and its inability to deal sensibly with incomplete sentences and tokens. Improvement in the parser should make it possible to better integrate keyboard and graphic entry of statements. Corrections to the parser would satisfy 16 of the requests made by students. Simple changes to the file loading system and the documentation of the program included in the laboratory assignment would probably satisfy 6 more.

I am also interested in experimenting with improvements to the display algorithms. There is room to improve the rendering of arrays. At present all arrays are shown as horizontally stacked boxes, and this leads to very long narrow objects. It should be possible to reformat arrays to display them more compactly. The other interesting area for improvement of the display is in the automatic layout of linked structures.

At present VisibleC simply attempts to place newly allocated objects down and to the right of the location receiving their pointers. In the Incense system Myers uses an invisible layer of nested boxes to contain descendants of an object with pointers, allowing sublists to extend off in different directions. A similar effect could be achieved in VisibleC by varying the starting points for placement search with the position of the pointer field in the parent structure. The situation in VisibleC is however more complex than in Incense because the display must accommodate changes in data structures. It would be possible to move other objects about when a new item was installed, but that seems to me to spoil the visual metaphor to some extent. When a new object arrives, it shouldn't alter the 'position' in the heap of existing objects. A compromise solution might be to institute a 'redisplay' command button which could be used to reorganise a messy display. This would at least make it clear that rearrangement was purely for the benefit of the observer, and not an action of the C language.

The student responses in the laboratory and in the survey suggest that VisualC is helpful in teaching pointer concepts. Their responses to the usability question suggest that it does require some learning effort however, and offer a warning that new capabilities should be added with great care. The ideas for improvement in the preceding paragraphs do not increase the complexity of the program from the viewpoint of a user—they are intended to make perform better and interact more uniformly.

Moving to a fully visual programming environment with structure editor, debugging and animation facilities might be worthwhile in the future. The positive reaction of students to VisibleC is an encouragement to further experimentation with visual programming systems. Some of the suggestions made by the students are clearly requesting a move in this direction. In my context it is not appropriate to make such a move immediately. My task at present, is to teach students who must go on to program in a conventional textual environment. It is important that they do learn about conventional C programming from the systems they use.

A program like VisualC is helpful because it has a small learning overhead, and assists in developing mental pictures of data structures which students can continue to use, either in the abstract or with pen and paper.

References

Baeker, R.M. & Marcus, A. (1990), *Human Factors and Typography for More Readable Programs*, Reading, MA, Addison-Wesley.

Chandhok, R., Garlan, D., Meter, G., Miller, P., & Pane, J. (1991), *Pascal Genie (Version 1.0)*, Chariot Software Group, San Diego, CA.

Goldensen, D.R. (1989), *The Impact of Structured Editing on Introductory Computer Science Education: The Results So Far*, ACM SIGCSE Bulletin 21(3), Sept 89, pp 26-29.

Henry, R.R., WHaley, K.M., & Forstall, B. (1990), *The University of Washington Illustrating Compiler*, Proc ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp 223-233.

Holmes, G., Smith, T., Rogers, W.J. (1996), *Computer Concepts Without Computers: A First Course in Computer Science.*

Hopper, K., Holmes, G., Rogers W. (1991), *The Magic of Modula-2*, Addison Wesley.

Hueras J., & Ledgard, H. (1977), *An Automatic Formatting Program for Pascal*, ACM SIGPLAN Notices, 12(7), pp 82-84.

Myers, B.A. (1983), *Incense: A System for Displaying Data Structures*, Computer Graphics, 17(3), pp 115-125.

Myers, B.A. (1986), *Visual Programming, Programming by Example, and Program Visualization: A Taxonomy.* In M.Mantei & P.Orbeton(Ed.), Proc Human Factors in Computing Systems (CHI'86), New York, ACM Press, pp 59-66.

Myers, B.A., Changhok, R. & Sareen, A. (1988), *Automatic Data Visualization for Novice Pascal Programmers*, Proc. IEEE Workshop on Visual Languages, New York, IEEE Computer Society Press, pp 192-198.

Myers, B.A. (1990), *Taxonomies of Visual Programming and Program Visualization*, Journal of Visual Languages and Computing, 1(1), pp 97-123.

Price, B.A., Baecker, R.M. & Small, I.S. (1994), *A principled Taxonomy of Software Visualisation*, Journal of Visual Languages and Computing 4(3), pp 211-266.