Reinforcing a Generic Computer Model for Novice Programmers

Philip A. Smith, Geoffrey I. Webb
Deakin University
psmith@deakin.edu.au
webb@deakin.edu.au

Abstract

Novices often find learning their first programming language to be a frustrating and difficult process. They have difficulties in developing and debugging their programs. One of their problems is that their mental model of how the computer works is inadequate.

In this paper we discuss a programming assistant, called Bradman, which we are currently developing. It is aimed at novice programmers and designed to reinforce a concrete mental model of how the computer works as a program is executed. It shows explicitly how program states change as statements in the procedural language C are executed. It does this by means of graphical display together with contextualised verbal explanations of each statement.

Keywords

novices programming, debugging, programming assistant, mental models

1. Introduction

It is well documented that novices can find learning to program an extremely difficult process (Bonar and Soloway, 1983; Spohrer and Solway, 1985; Johnson, 1990). Both expert and novice programmers experience times when, during the development of a program, they find it does not exhibit the behaviour they expect. When this happens a novice programmer is often at a loss as to how to proceed. She often will not know why the program produced such output even after carefully examining the program code. On the other hand an expert programmer possesses skills and knowledge which enable her to hypothesise about the possible cause of the error, track it down and rectify it. What are the skills and knowledge an expert possesses that a novice lacks?

First, experts think about programming problems in a more abstract manner. They are ‘able to see the commonalities and differences among various problems and programs...’ (Ehrlich and Soloway, 1985). An expert might think about a programming problem in terms of certain high-level algorithms or data structures whereas a novice is more likely to think in low-level terms such as individual statements.

Second, novices lack an adequate mental model of how the computer works (Du Boulay, O’shea and Monk, 1981). Often they will try to compensate by using models drawn from their real-world experience. These models are inappropriate because they do not address the constraints of the computer world (Eisenstadt and Breuker, 1992). We will be concentrating on the second issue in our discussion.

2. Mental Models

When a programmer wishes to write a program to solve a problem she has three different views that she must reconcile. She must have a goal that she wishes to achieve. This is normally given as some sort of program specification. She must have knowledge of what the computer is capable of achieving, that is some sort of mental model of how the computer works. Finally, because a computer is such a powerful instrument, she must have some process of utilising the computer to realise her goal. This process involves everything from high level abstraction of the programming problem to its final realisation in the code.

Obviously, if the programmer has a faulty mental model of how the computer works then she will have great difficulties in creating correct programs to solve her problems. Hence one of the first learning tasks a novice programmer must undertake is to obtain an adequate mental model. This is not always an easy task. One problem is that novices have is trying to apply analogical models from their real world experience to computing. Bonar and Soloway (1985) call this ‘preprogramming knowledge’ and show that it can cause a lot of the misconceptions suffered by novices. This problem can be so pervasive that novices can fail to see that a programming problem is a problem at all (Eisenstadt and Breuker, 1992).

What sort of model should be reinforced by the debugging tool? A model that is too low-level would not be of any use. For example, a model that displayed a program in terms of binary code is unlikely to be of assistance to a novice programmer.

In an procedural language such as C a program is a sequential process. As it does this it undergoes various changes of state. A program state can be defined as the value of each of the program variables together with the execution location of the program. In an procedural language like C program states change after the execution of expressions. If an expression is executed which does not cause a change in program state then we can safely say the program is in an infinite loop.

This dynamic aspect of program execution is often shown in a static way by instructors using blackboards, etc.(Rajan, 1992). Common debugging tools that create an environment in which the user can more easily view the program by providing facilities to view the value of critical variables at critical execution points are designed for use by experts and make no explicit attempt to reinforce a mental model. We need to give novices a model that integrates this aspect of computing.

3. Bradman

Bradman is a window-based programming tool specifically aimed at novice programmers learning C as their first programming language. It is designed to emphasise a model of a program being a dynamic process which performs its function by altering program states. For ease of implementation Bradman works at the statement level rather that the expression level. We plan to extend Bradman to work at the expression level at a later date. It explicitly displays to the user how the program states change as each individual program statement is executed. As well as the graphical model it also gives textual explanations of each statement’s function as it is executed. The explanations are based on the semantics of the language and are contextualised for each individual statement.

Bradman takes a syntactically correct C program as input. Correction of syntactic errors lies outside the current range of the system. When Bradman is invoked it presents four windows each of which displays a different aspect of the program. We will now discuss each of these windows individually and show how they assist in developing an adequate mental model of the computer and support novice programming.

4. Code Window

The code window is essentially the same as that provided by any state-of-the-art visibility debugger. It displays the program code and shows explicitly the current point of execution. The point at which execution has arrived is explicitly shown by the spider on the side. Scrollbars enable the user to see different parts of longer programs. Below is an example of the code window.

The code window in figure 1 shows a simple program which reads in three floating point numbers, sums them and prints out their total. The line numbers are shown at the far left. In the next column are square into which a user can designate breakpoints. This is done by means of a mouse click within the appropriate box. Breakpoints can be turned off the same way. The spider always sits next to the statement about to be executed. In this case the spider is sitting next to the statement at line 9. The printf statement at line 7 has been executed and so has the scanf statement at line 8. On the far right hand side is the scrollbar, which can be used if the user program is too big to fit on the screen. At the top of the window are various buttons that perform various functions when clicked.

The contribution of the code window to the model is to show explicitly how the part of the program state concerned with control location changes after the execution of a statement. As the program executes the spider moves to new locations within the program. This can be especially useful to novices at times where the new control location is one which she did not expect.

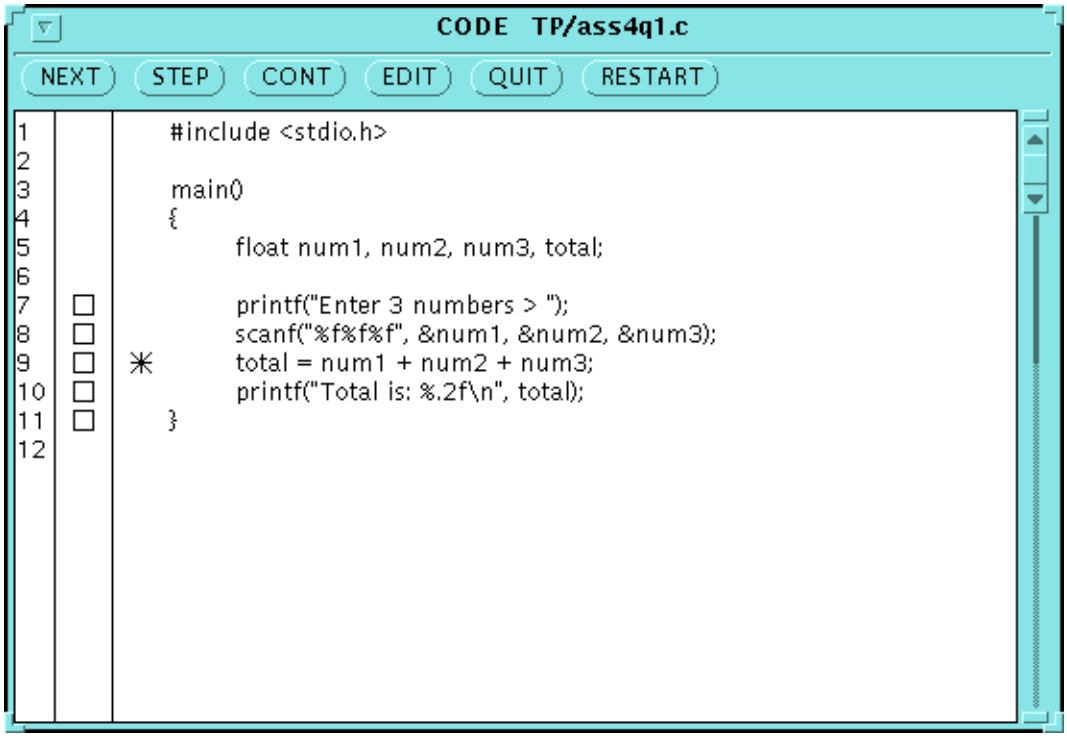

Figure 1. The Code Window.

5. Variables Window

The variables window contributes to the model by providing an explicit model of how each statement alters the states of the program variables. If user-defined functions are called during execution then the variables associated with the function are displayed underneath those of the calling function for the duration of its life.

The display consists of two columns of values. The left hand column contains the values of the variables before the statement in question was executed. The right hand column contains the values of the variables after the statement was executed. The statement itself appears in a box between the two columns. Values in the before column that are referenced in the statement are highlighted and have a line resembling lightning drawn from their box to the statement box. Variables that are referenced by the statement have their values highlighted in the after column and a straight line drawn from the program statement to their boxes. This provides an image of the statement as an active agent acting on the values of variables in its pre-state and altering the values of variables in its post-state.

In C a variable needs to be assigned a value explicitly before it can be safely used in an expression. The initial value of a variable cannot be assumed to be zero (or any other number). An attempt to reference an uninitialised variable will usually result in totally unexpected behaviour from the program. If a variable is uninitialised the variables window will not display a garbage value but will simply mark its box as yet unset. Any attempt to reference an uninitialised variable will result in an explicit error message.

One of the things that may not be immediately obvious to a novice programmer is that the after values of one statement will be the before values of the next statement. Bradman shows this explicitly by having the after column migrate across the window to the before column when the user steps through to the next statement. This animation can be switched on and off by the user by means of the animation button at the top of the window.

The variables window in figure 2 shows the state of the variables when the program shown in the code window has executed the scanf statement at line 8 and is waiting to executed the assignment statement at line 9. We will assume that the user has entered 2 3 4 into the input / output window which is described later. The display shows the values of the variables before and after the execution of the scanf statement. None of the variables had been set before the scanf statement. After the scanf statement the variables *num1*, *num2* and *num3* have received the values 2, 3 and 4 respectively. The highlighting and the lines connecting the statement box with the individual value boxes show which of the variables have been assigned a new value by the statement. While use of both lines and highlighting is strictly redundant we use both as a means of reinforcing what has occurred.

The variables window in figure 3 shows the state of the variables after the user has clicked the step button and executed the assignment statement. The variables window has changed in various ways. The new *before* column is identical to the previous *after* column. The values of *num1*, *num2* and *num3* are 2, 3 and 4 in the before column. The statement box now contains the statement *total = num1 + num2 + num3;* Because *num1, num2* and *num3* have all been referenced (their values have been used) by the statement their values are all highlighted in the before column and lightning goes from each of their value boxes to the statement box. The variable *total* is assigned the value 9 by this statement. Hence its value is highlighted in the *after* column and a line is drawn from its box to the statement box.

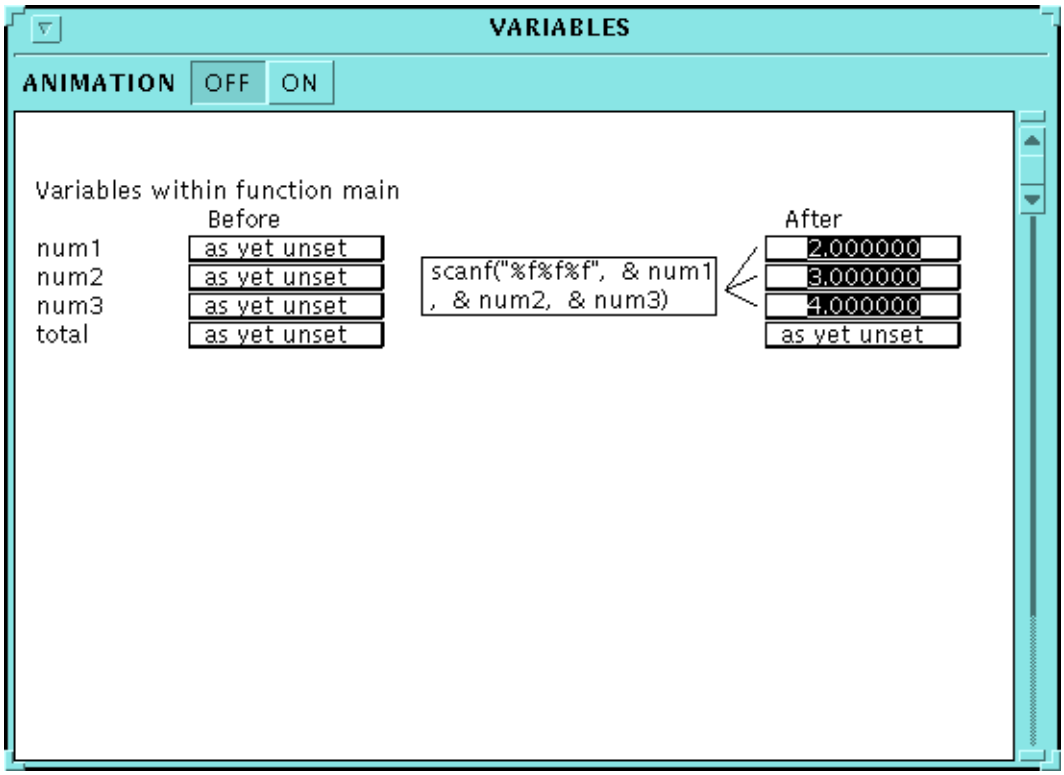

Figure 2. The Variables Window after Execution of the SCANF Statement

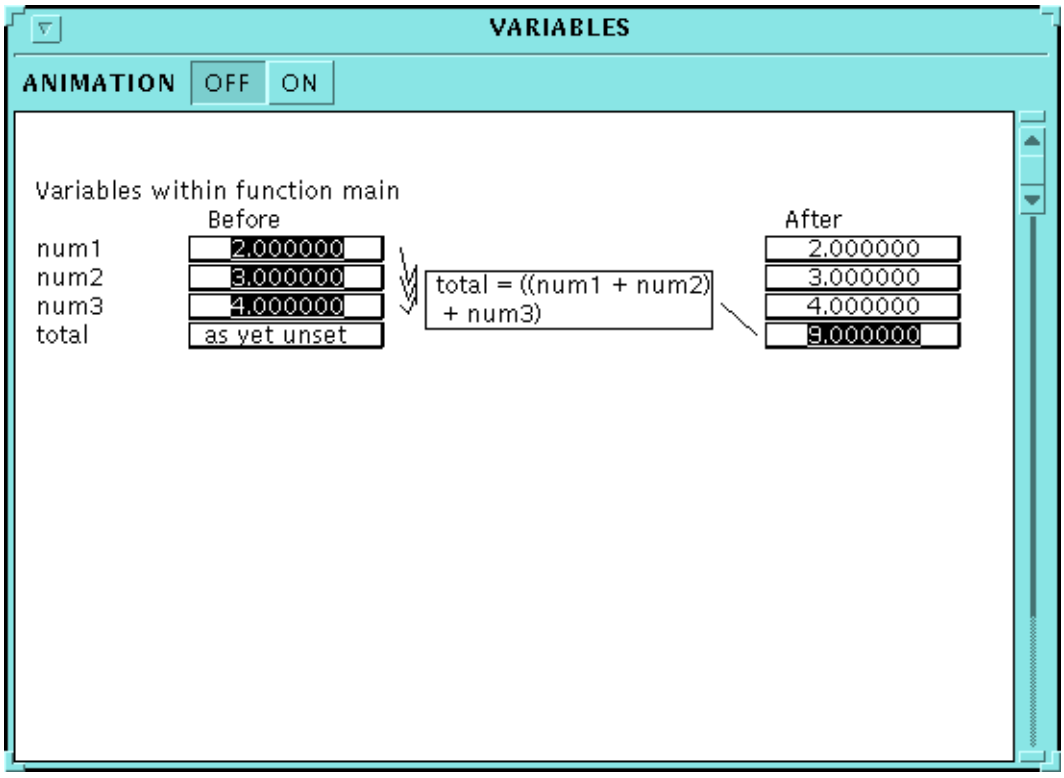

Figure 3. The Variables Window after Execution of the Assignment Statement

6. Explanations window

The code window and the variables window shows the programmer which changes are effected in the program state by the execution of a statement. However, it does not tell the user how the statement caused these changes. The explanations window gives contextualised information explaining how each statement works.

In the explanations window in figure 4 the user is told that the right hand side of the statement is evaluated giving the value 9 and that this value is stored in the variable *total*. The user is also shown exactly how the calculations are made. That is, that *num1* and *num2* are summed first to give the value 5 and this value is added to *num3* to give the value 9.

The explanations window uses a very simple process by which information is added to it as the program is executed. Even so, we have elsewhere demonstrated (Smith Webb 95) that novice users who had access to the explanations window of Bradman felt that the system was of more assistance to them than students who had a version of Bradman that was identical except that the explanation facility was missing. We plan to work on a more sophisticated version of the explanations in the near future which we expect will be of even more benefit to novices.

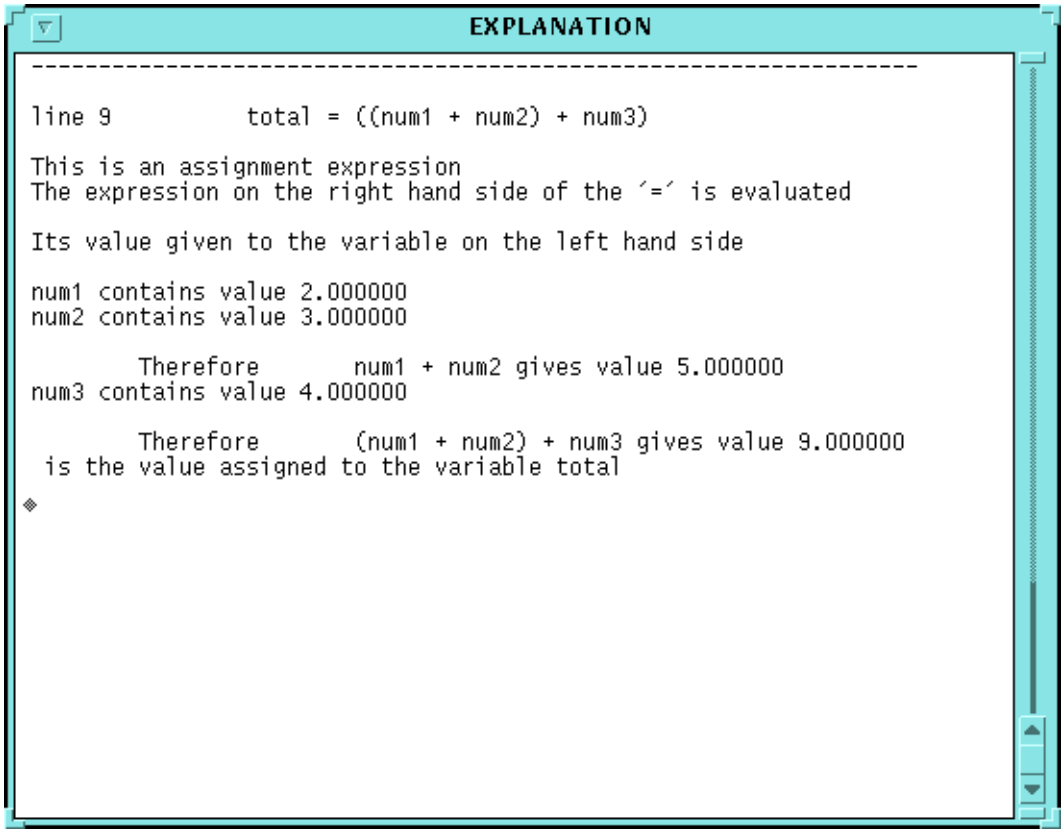

Figure 4. The Explanations Window

7. The Input / output window

This window provides a mechanism by which the user can communicate necessary input and output to her program. While this is mainly straightforward, this window makes visible two aspects of input that otherwise are invisible to the user. First, when a user running her program normally enters more standard input than the program is ready for, the excess input is stored in a buffer and used if more input statements requiring standard input are executed. This buffer is normally invisible and the

novice programmer can become confused when her input variables are given values she did not intend for them. To clarify this for the novice, unused input is displayed at the bottom of the input / output window. Second, when a statement requiring input from standard input is executed and the input buffer is empty the program will wait until input is entered. This can confuse novices if they have not coded an appropriate prompt for input into their program causing them to think that there is something wrong with their program. When a program being run on Bradman is waiting for standard input a flashing message will appear at the bottom of the input / output window telling the user to enter input.

8. Pointers

Pointers are an important feature of C that can be a difficult concept for novices to grasp. Bradman uses the variables window to model how the pointers are used to point to other addresses in memory. The following is a simple example of how this concept is modelled. Consider the program in the code window shown in figure 5. This program simply assigns the address of variable *k* to the pointer *p*. It then assigns a value to *k* indirectly through *p*.

The variables window in figure 6 shows the state of the variables after the assignment statement *p = &k;* is executed. The *before* box of *k* is highlighted. Because the address of *k* is being referenced the whole box is highlighted not just its value. Also the variable does not need to have been assigned a value before its address is referenced and so as yet unset remains in the *before* box. The *after* box of *p* is also highlighted. This time only the content of *p* is highlighted not the whole box. The content of *p* is shown as an arrow to the *after* box of *k*. This is designed to show explicitly that *p* contains the address of *k* something that might not be obvious if the integer address of *k* was placed in *p*.

The variables window displayed in figure 7 shows the state of the variables after the next statement **p = i + j;* is executed. The before value of *i* and *j* are highlighted to show that they were referenced in the statement. The before value of *p* is also highlighted because its value is used to determine the address into which the new value should go. The after value if *k* now contains 11 and is highlighted to show that it is the variable that was given a new value. This model is designed to give the novice a concrete idea of the concept behind pointer variables.

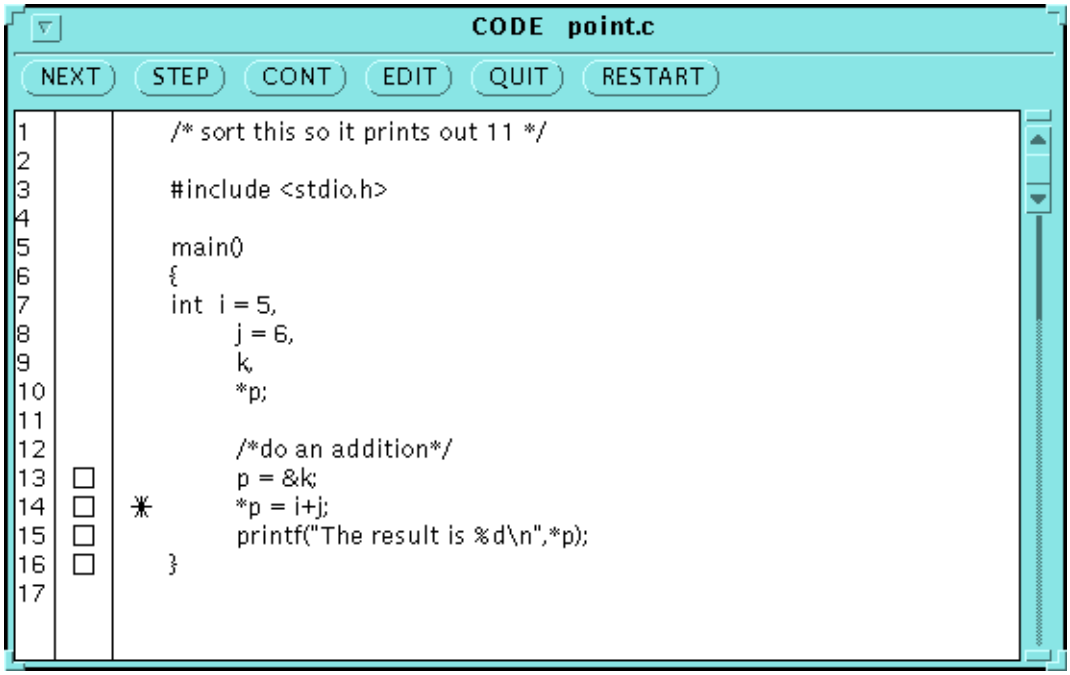

Figure 5. A Code Segment Using Pointers

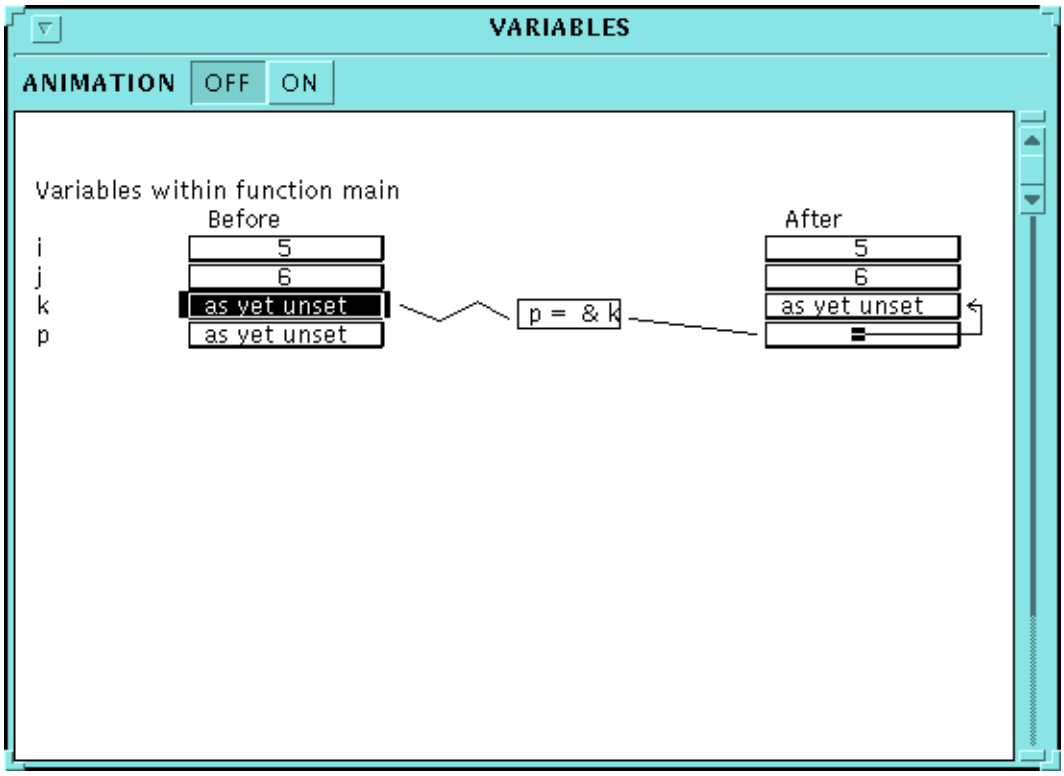

Figure 6. The State of the Variables after the Assignment Statement *p = &k;* is Executed

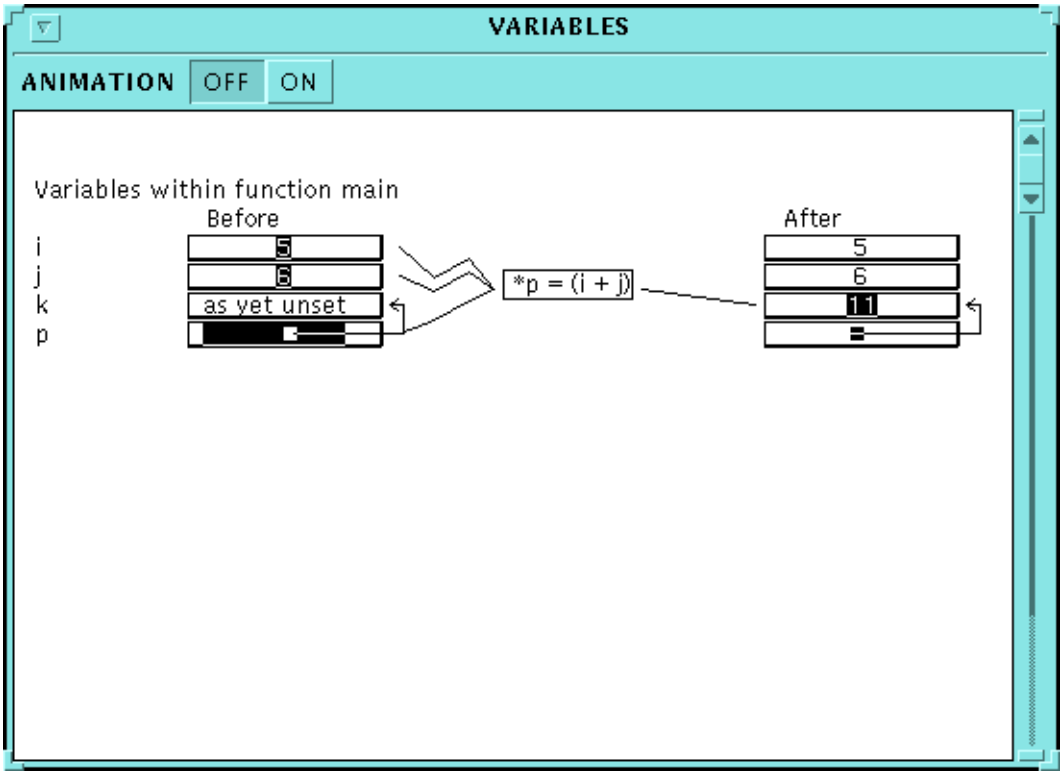

Figure 7. Shows the State of the Variables after the Statement **p = i + j;* is Executed

9. Conclusion

Bradman is a tool that is designed to assist novice programmers to learn C as their first programming language. It does this by reinforcing a model of how the computer produces output from input via a program. This is shown as a dynamic process which produces its results by constantly changing program states. A model of the way the program states change by the execution of program statements is augmented by explanations of how the statements effect these changes. We have already conducted an experiment which showed that novices can benefit from contextualised verbal explanations of how statements create changes in program states. We intend conducting experiments into the effectiveness of the graphical model early next year.

10. References

Bonar, J. and Soloway, E. (1983). Uncovering principles of novice programming. *ACM*, pp. 10-13.

Bonar, J. and Soloway, E. (1985). Preprogramming knowledge: a major source of misconceptions in novice programmers, *Human-Computer Interaction*, Vol. 1, pp. 133-161.

Du Boulay, B., O'Shea, T. and Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices, *International Journal of Man-Machine Studies*, Vol. 14, pp. 257-249.

Ehrlich, K. and Soloway, E. (1985). An empirical investigation of the tacit knowledge in programming, *Human Factors in Computing*, Norwood NJ: Ablex, pp. 113-133.

Eisenstadt, M. and Breuker, J. (1992). An account of the conceptualisations underlying buggy looping programs, *Novice Programming Environments: Explorations in Human Computer Interaction and Artificial Intelligence*, Lawrence Erlbaum Associates: London.

Johnson, W. L. (1990). Understanding and debugging novice programs, *Artificial Intelligence*, Vol. 42, pp. 51-97.

Rajan, T. (1992). Principles for the design of dynamic tracing environments for novice programmers, *Novice Programming Environments: Explorations in Human Computer Interaction and Artificial Intelligence*, Lawrence Erlbaum Associates: London.

Smith, P.A. and Webb, G.I. (1995). Transparency debugging with explanations for novice programmers, *Proceedings of the 2nd Workshop on Automated and Algorithmic Debugging*, St. Malo, France, 1995.

Spohrer, J.C. and Soloway, E. (1985). A goal / plan analysis of buggy Pascal programs, *Human-Computer Interaction*, Vol. 1, pp. 163-207.