Developing Mental Models of Computer Programming Interactively Via the Web

Michael P. Bruce-Lockhart and Theodore S. Norvell
Computer Engineering Research Labs, Faculty of Engineering and Applied Science,
Memorial University of Newfoundland, St. John's, NL, A1B 3X5.
mpbl@engr.mun.ca theo@engr.mun.ca.

***Abstract* - A website featuring interactive examples used to teach introductory programming to both on-campus and, recently, distance students is described. On-line notes are created using a pair of tools, Web-Writer++, an authoring system for programming instructors, and the Teaching Machine, a program animator which is used to interactively display the examples. The tools allow the instructor in the classroom or the student on their own to step through computer programs written in C++ or Java. The animations that unfold are designed to build a deep understanding of how computers process programs, consonant with the kind of mental models we believe professional programmers hold. In-class versus distance experience will be discussed.**

Index Terms – The Teaching Machine. Program animation. Program visualization. Interactive course notes. Web-based course notes.

INTRODUCTION

The Faculty of Engineering at Memorial University in St. John's, Newfoundland is a leading Canadian co-operative engineering education institution offering five different degree programs in a co-operative engineering context. All first year students take a common curriculum which includes Structured Programming, (basically CS1). Students choose their major at the beginning of second year. During that second year both Electrical and Computer Engineering students take Advanced Programming followed by Data Structures.

Starting in 1999 we have introduced new teaching methods aimed at resolving what we consider to be a rising problem: modern students have more difficulty with traditional approaches to computer programming than did their counterparts of even a decade earlier. We believe a large part of the problem is that most of them come to university with very little appreciation of how a computer really works. Paradoxically, while today's student has grown up with computers, most young people are really more net- than computer- literate. On the one hand the computer is dwindling from an object of inherent interest into a mere gateway appliance; on the other hand that appliance, layered as it is in a wedding cake of menus, wizards and windows, is vastly more complex than the machines of only a decade ago. Manipulating today's machines has become so complicated that help is generally only available as a series of goal-based recipes. It's no wonder that students have little in the way of a mental model of computing.

THE MODELING PROBLEM

We have talked about the modeling problem more formally in a previous paper [1] using an approach based on the work of Norman [2] and Yehezkel [3]. The essence is that given a system *T*, a mental model of *T* can be defined as *M[T]*. Norman's work, however, was based on a very well defined *T* (a simple calculator). In teaching high-level (as opposed to machine language) programming it is much harder to define *T*.

As we struggled to impart to our students that each instruction they wrote was meaningful, we had an important insight. The machine (or system) *T* we were programming (and which we wanted the students to understand), was not really a computer, at least in the classic, hardware, sense.

Consider the following simple C code:

```
int x=5;
int y = 12;
int z;
z = y/5 + 3.1;
```

In the language of programming, we say, there are four *instructions* to be *executed*. Instructions to what and to be executed by what? *T* of course, but *T* is certainly not the CPU. The first three "instructions" are actually to the compiler. We view them as request for allocation of memory, in the stack, if they are internal declarations, in the static store if external. The fourth is a minefield. There's a truncation and two automatic type conversions. If you really want students to understand it they need to be able to interpret the expression and see the conversions, but *these are normally done by the compiler*. CPU operations include fetching the value for `y` (whether in a register or memory), carrying out the separate calculations, one

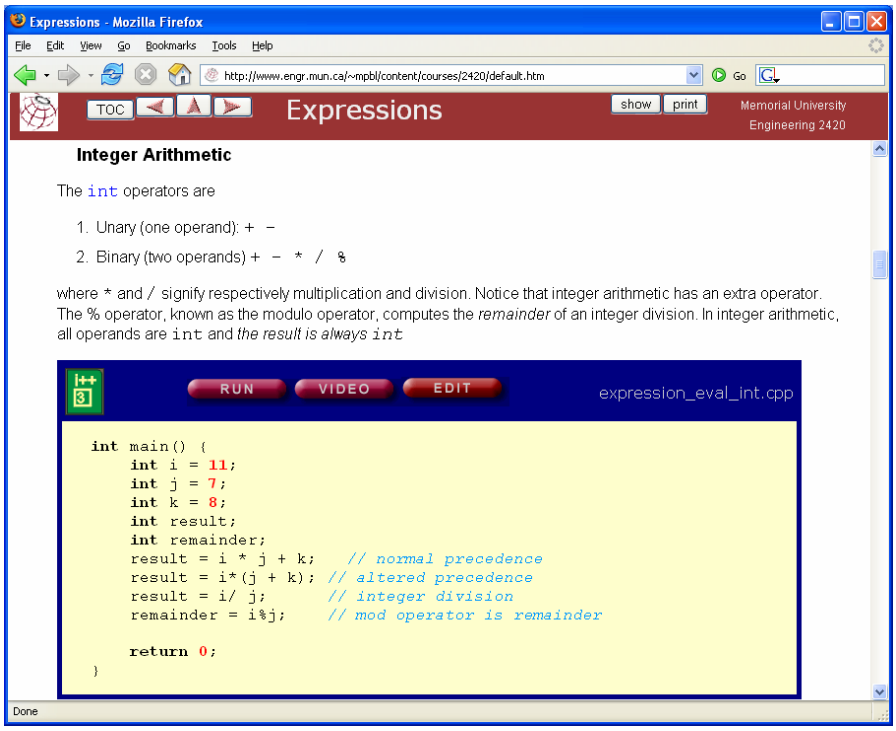

Figure 1. A page generated by WebWriter++.

in the integer arithmetic unit and one in the floating-point processor, and writing the final value back to z.

We define *T* to be *the system to which we are giving instructions*. That is, *T* is at least partly defined by the language. In the case of C++ and Java languages, *T* is an abstraction combining aspects of the computer, the compiler and the memory management scheme. Our *T* is not nearly as "knowable" as Norman's. That does not relieve us of the responsibility of at least trying to define it. We developed the Teaching Machine to provide students with a visual representation of the *T* that we believe approximates the one most professional programmers program to.

INTEGRATED TEACHING PAGES

I. WebWriter++

Originally we used the TM in the classroom as an adjunct to lecture notes written on the chalkboard or presented as overhead transparencies. As the TM is written in Java and can be run as an applet, it seemed natural to embed it directly into web pages and to use these web pages as the lecture notes. However, to do this efficiently and effectively, more than HTML was required. Thus was born a second tool.

WebWriter++ (WW++) is an authoring system written in JavaScript that allows instructors to easily create web pages for programming courses. Its most important feature is that it allows C++ and Java source files to be retrieved from the server and displayed on a web page. The examples appear as they would in a program editor, with keywords, comments, and constants all marked

Figure 1 shows a portion of a lecture on expressions taken from the Engineering 2420 lecture notes. The exact same notes are used

- As lecture notes in a class of 200+
- As study notes via the web (or they may be downloaded)
- As course notes for the distance version

The only difference between versions is that different style sheets are used as larger fonts are required for lecture projection than for self-study on an individual computer screen.

The print button on the title bar creates another version that is optimized for printing.

In Figure 1 (which shows the smaller fonts) the instructor has created a simple example using a standard Integrated Development Environment (IDE) and embedded it directly into the lecture notes. The notes themselves were prepared on a conventional HTML editor but the insertion of the example is handled by WW++. The appearance of the code within the code container is governed by a style sheet allowing instructors to match its look to that of the code in the IDE students use for their assignments.

WW++ provides the code container with up to three buttons. Of most importance is the run button which activates the TM in a new window, as shown in Figure 2.

II. The Teaching Machine

The TM uses the metaphor of a debugger; in Figure 2 it has already been stepped through a number of instructions. Each of the declarations has been

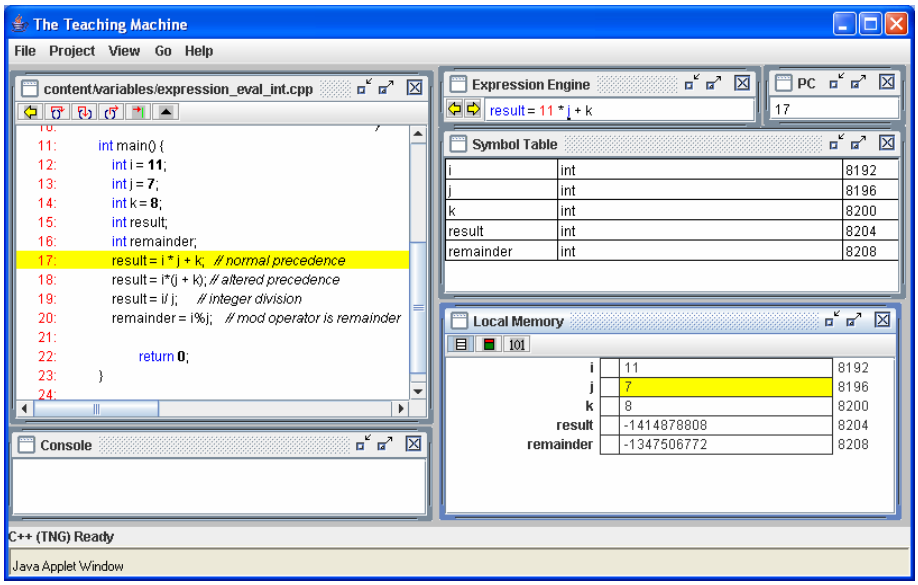

Figure 2. The Teaching Machine

treated as a line of active code. The execution of a declaration results in space being set aside in memory for the variable declared. If it is initialized the initial value appears in memory.

The next line to be executed has been highlighted in the code window. As it is in the form of an assignment expression, the expression has been loaded into what we call the Expression Engine (EE) shown in the top middle of Figure 2. The arrow buttons in the EE allow us to step through the evaluation of the expression. In Figure 2 the expression has been partly stepped through. The value of `i` has already been evaluated to 11 and the value of `j` is in the process of being looked up. Its address has been found in the symbol table as 8196 and address 8196 is highlighted in memory to show that the next step is to fetch the value of 7.

The fetch operation models the physical computer directly, while the symbol table is an artifact of the C++ compiler. The TM is a hybrid in that it models both the actions of the CPU —carrying out calculations— and the compiler —parsing the expression and looking in the symbol table. In order to properly understand how to write even very simple C++ or Java programs, students have to have a grasp of all these mechanisms as an integrated whole; that is they need an integrated mental model; we believe it is best to present these parts as if they were a single integrated machine.

Similar program animation systems include Jelliot [4] and VIP [5].

III. Usage

In presenting the material in lecture, we generally step through the examples as we go. Originally, we went through a similar process by hand, drawing and redrawing memory contents by hand as the program evolved. The TM has freed us from this labour, giving us far more time to interact with students and answer questions. Moreover, by integrating the demos directly into the notes, students are able to re-run the same examples for themselves. To aid in this process, WW++ provides a facility for allowing instructors to write separate demo notes that are taken out of the lecture flow but can be revealed later by the student to assist them in how to run the demo.

Distance students don't get to see the demos run in class, so videos have been created of some of the most crucial ones. These simply show the TM being run by an instructor with an audio commentary as well as labelled annotations. Where such videos are available, the instructor can include a video button in the example container (see Figure 1).

The edit button allows an example to be edited and rerun in the TM to answer 'what if' questions.

IV. Other WebWriter++ Features

It is often undesirable to show an entire program when discussing a particular point. Using a simple markup system embedded in comments in the code, the instructor can select only a portion of the source file for display and can select different portions for different points. The student can see the entire program by activating the TM. Of course, this can't be done when studying from printed notes so, when the notes are printed, full versions of the code are shown at the end of each chapter.

The authoring system includes a number of other features. For example one can roll the mouse pointer over a variable and see its scope illuminate in the code.

WebWriter's primary thrust is to let courseware authors focus on content rather than technology.

MORE ADVANCED MODELS

In Advanced Programming, our second course, students need a more sophisticated model for computer memory. The "Local Memory" window of Figure 2 actually represents the stack and is shown as such in more advanced courses. In addition, the heap and static store are shown as separate windows. Again while most computers have stack management built directly into the hardware, heap management is implemented in software in a language dependent way. Thus the TM includes aspects of memory management as well as of the compiler and the hardware.

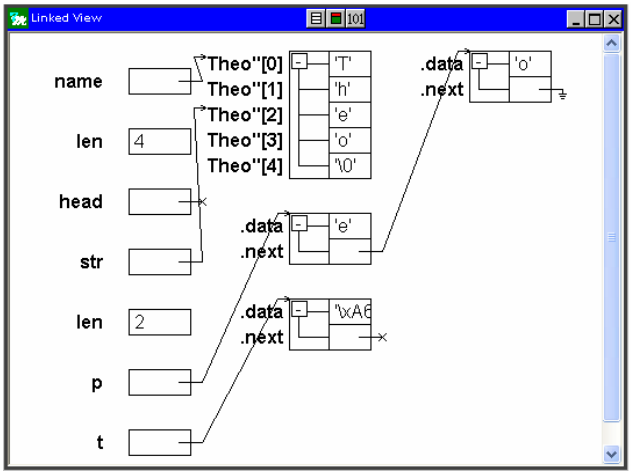

Figure 3. The linked view.

I. Linked View

In the standard views of memory, pointers are shown as addresses (in decimal) and values of C++ references are presented as the name of the variable being referred to. An alternative view of memory is shown in Figure 3. The linked view shows stack data on the left and heap data to the right. Pointers and references are shown as arrows pointing to the box representing the data item they point to. Typically the linked view is used in more advanced courses, such as Data Structures.

II. Program animation and debugging

The resemblance of the TM bears to a debugger is quite deliberate. However the intended audience and purpose are different. A debugger is intended for a professional software engineer, who already has a good understanding of programming. The TM is intended to help learners to build effective mental models of programming. The linked view gives an example of this. In a debugging context, the linked view would be hopelessly space consuming and would require support for navigation through structures too big to fit on the screen. In the TM, the linked view is only intended for small examples and works quite well on them.

I. Language Coverage

The current version of the TM, known as Teaching Machine 2, supports most of the features of C++ used in our first three programming courses (eventually to be extended to all remaining such features with the exception of templates). In addition we have a reasonably complete Java implementation (enough that we use it in a grad course) with plans to implement the entire language, except for concurrency and generics.

IN-CLASS EXPERIENCE

I. Context

As of this writing all course notes in the fundamental three course programming stream have been rewritten using WW++ with extensive use of embedded, interactive TM examples. All three courses use C++ as a teaching language. However, we have far more experience with Advanced Programming (AP) and Data Structures (DS) than with Structured Programming (SP, our version of CS1).

II. Advanced Courses

The TM was first used in AP in 1999 as a standalone. It was something of an interruption to the usual flow of lecturing from overhead transparencies or the chalkboard and so fully integrated notes built with WW++ were started in 2003.

In 2003, the notes were written while the course was being taught. Students were guardedly positive during that process but embraced the concept in 2004 when the instructor was able to concentrate on teaching instead of writing (and when printed notes were available at the outset).

Introduction of the TM and integration into notes in DS followed a similar arc, with a one to two year delay. We have ample evidence of the success of the approach in these two courses, including student surveys and exit interviews with the department head [1]. Perhaps the most telling indicator of its success, however, is that in the late nineties AP was widely regarded by students as the most difficult course in the entire Electrical Engineering curriculum, so much so that we devoted four lectures a week to it instead of

three. It no longer has that reputation, despite reducing the number of lecture hours to three with no drop in content. Indeed, grades have become so high that it is clear we need to add new material to the course.

II. First Year Programming

Let us say at the outset that we have not yet been so successful with the first year course in Structured Programming (SP). In good part this has been because of confounding factors such as major changes to the course itself. But it has also been due to mistakes on our part, and therein lies a cautionary tale. We're learning and we believe we'll get it right but the mistakes are illuminating.

In 2005 the Faculty of Engineering committed to a full-scale introduction of the technology to SP. Following the pattern of the more advanced courses, new web-based notes were written as the course was taught and were moderately well received by the students. At that point we were at the guardedly positive stage and had every expectation that the same arc would be followed as occurred in AP and DS. However, in the background, major changes were in the offing.

Although Engineering has four academic years, our students have always entered only after an additional general year of university. Starting in 2009 students will enter Engineering directly from high school, as they do throughout the rest of Canada.

The brunt of this change is occurring in the first year. Ten engineering courses had to be cut back to four to make room for first year math, science and English. The first year programming course was kept on the grounds that it is important to expose first year engineers to as many different styles of thinking as possible.

Nevertheless, we had to agree to revise the course extensively. The changes were not required until the fall of 2008. However, that year is going to create a major transient in the curriculum. Of the four first year engineering courses in the new curriculum, two are composites of existing courses and so can only be introduced in 2008, while the other remains substantially unchanged. SP is the only course that could be changed in place, as it were. In the interests of reducing the 2008 transient we decided to implement the changes immediately.

Traditional (that is, non-objects-first) beginning courses spend a lot of time writing simple programs with just a main function. Functions are not introduced until about half way through and arrays generally don't get covered until the end of the semester. As SP is a terminal course for most of the class, the objects first approach was not an acceptable option. Instead, we decided to teach programming from the inside out, starting with functions and moving to full programs only at the end of the course.

In the winter of 2006, the notes were rewritten again. The course was being taught to the larger class size of 200+ for the first time. Not to put too fine a point on it the course was a disaster. Students were baffled.

The instructor (Bruce-Lockhart) had taught the course in 2005 with moderate success, and had been using the technology to teach AP for years. So it wasn't the instructor. Moreover, the same instructor had taught introductory circuits to the same class in the same (new) lecture theatre, developing new electronic notes as he went with great success. So it wasn't the new class size, the particular group of students or the qualities of the new room.

III. Suppression of main

Our traditional approach had always provided a very distorted view of programming. Students spend most of their time writing main functions whereas professional programmers very seldom write a main function. The TM represents nine years development, has over 500 classes and 5,000 functions, yet has only one main function (written nine years ago). At best it can be regarded as a moderate sized program. By getting students to write functions embedded in a larger system, we reasoned that we would be modeling modern programming more closely, while also opening the door to being able to do more interesting tasks (such as image processing).

In keeping with that philosophy, we suppressed main completely, which turned out to be a very bad idea. In retrospect, it's obvious: programs became mysterious and students had no idea where the data fed to their functions came from. Already unhappy about learning programming, bafflement quickly turned to resistance.

IV. Failure to Engage

The bafflement was compounded by a failure of the students to engage with the TM. One of the benefits of our approach is that students can rerun examples for themselves when they are studying. The benefits of such active engagement are well understood by the programming visualization community which has found that simply viewing a visualization provides inadequate learning [6]. We knew from surveys that our advanced students were using the TM actively on

their own. We found at the end of the course that most of the first year students had never run it on their own and consequently didn't really understand how either the TM, or the model it was based on, worked.

At the end of the winter 2006 semester, three tutorials in the computer lab were offered to (and taken by most) students. New examples were created for each of the three most difficult topics and students were taken through them on the TM, just as in lecture, only this time they were asked to follow on and run the examples themselves on the lab machines. Student response was overwhelmingly positive and the course was at least rescued.

DISTANCE VERSION

In the summer and fall of 2006, SP was offered by distance as a pilot project. In view of the experience reported above, the examples (and to some extent the notes) were rewritten to re-introduce main, albeit in a reduced role (students see main and have its workings explained but are not expected to write main code). In addition, to compensate for the fact that students could not see the instructor in class, videos of the most crucial examples were made. Very few students took the course, five in the summer and seven in the fall, so conclusions are necessarily sketchy.

One interesting point is that all students in the summer group had failed the winter on-campus course and were retaking SP by distance in preference to simply writing a re-exam in the fall. Normally, the pass rate on re-exam is about 50-60%; however all these students passed and a couple of them did quite well.

In contrast, the fall group were all neophytes. Performance was more mixed, although in the end everyone passed. One notable difference from the summer group was that there was a lot more difficulty getting student programs to compile. The summer students had the benefit of labs when they did the course on-campus and so at least knew how to set up a project on the compiler.

At this stage we have concluded in teaching first year programming by distance, that our approach was useful to at least some of the students, and that if we are to continue the experiment we must find a way to create an on-line equivalent of a lab, whereby an instructor can at least be able to view a student's screen and see why a program isn't compiling.

SUMMARY: MAKING THE IMPLICIT EXPLICIT

Students, especially early students, often complain that programming is 'too abstract'. To the teacher, used to dealing with the abstract algorithms and abstract mathematical models of computation, such comments seem puzzling —what could be more concrete than a computer program which spells out all the details for the compiler. But the student has a good point, from their perspective. First, that program only spells it out if you know all the ins and outs of the language. An identifier might identify any of a number of identically named identifiers. An expression might contain implicit conversions. A parameter might be copy constructed. Two pointers might be aliases. The TM strives to make the implicit explicit, visible, and tangible. Second, the model of computation is implicit. The instructor may talk about variables changing value, but this is not meaningful for most students if they can't see the values of variables and can't see them change. The implicit model of computation is made explicit by the TM.

We found that the TM was useful on its own, but imposed breaks in our lecture flow; moreover using the TM was a significant break from the student studying the notes out of class. WW++ allowed us to create interactive electronic course notes with little effort. Experience with the TM and WW++ together has suggested to us that the combination improves the experience of the course for students in lectures, studying on their own, and in distance education.

REFERENCES

[1] M. Bruce-Lockhart, T.S. Norvell & Y. Cotronis. Program and algorithm visualization in engineering and physics, *Proceedings of the Fourth Program Visualization Workshop (PVW 2006)* (Florence, Italy, 2006).

[2] D. A. Norman. Some Observations on Mental Models, in *Mental Models*, D. Gentner and A. L. Stevens eds., Lawrence Erlbaum Associates, 1983.

[3] C. Yehezkel, M. Ben-Ari, and T. Dreyfus, Inside the Computer: Visualization and Mental Models, *Proceedings of the Third Program Visualization Workshop, (PVW 2004)* (Warwick, U.K., 2004).

[4] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, Visualizing Programs with Jeliot 3, *Proceedings of the International Working Conference on Advanced Visual Interfaces 2004, (AVI 2004),* (Gallipoli, Italy, 2004).

[5] A. Virtanen, E. Lahtinen, and H.-M. Järvinen, VIP, a Visual Interpreter for Learning Introductory Programming with C++, *Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education*, 2005.

[6] T. Naps, G Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, J. A. Veláquez-Iturbide. Exploring the role of visualization and engagement in computer science education, *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*, pp. 132–152, ACM Press, 2002. Also published in *ACM SIGCSE Bulletin*, vol. 35, #2, June 2003