The Memory Grid: A Glass Box View of Data Representation

T. Grandon Gill
IS & DS Department, CIS1040
University of South Florida
Tampa, FL 33620, USA
ggill@coba.usf.edu

ABSTRACT

Educational research has found that learning is often enhanced when the concrete is mixed with the abstract. One method of achieving this, in the context of teaching computer programming, is to provide students with a model of the activities that occur within the computer when programs are loaded and program steps are executed. Such a model is sometimes referred to as a glass box—differentiating it from a black box approach, where program activities are treated as being purely abstract in nature. The paper describes a glass box exercise developed by the author that requires students to match abstract data declarations to their concrete representation in primary storage. The exercise is used to help programming students better understand the nature of variables, arrays, and structures. Upon completing the exercise, which has proven to be popular with students in an introductory programming course, the instructor has found students are better able to apply the elegant (but initially mystifying) notations used for pointer, array and structure operations in C/C++. The paper also describes GridGen, a C++ based tool for creating such exercises and for generating online tests that can be delivered in course management environments such as Blackboard. The paper concludes with a discussion of the methodology and results that were used to evaluate the effectiveness of memory grid exercises.

Keywords: Introductory programming, CS1, Glass box models, C/C++, Data representation, Memory

1. INTRODUCTION

One of the key differences between an expert programmer and a novice is a deep understanding of how the code that he or she writes translates to underlying activities on a computer. In a computer science curriculum, separate courses on logical architecture and compiler design help to teach such concepts. In a typical MIS program offered within a business school, however, such topics receive little or no attention. Furthermore, the time allocated to programming courses tend to be very limited in such programs (Reichgelt, *et al.,* 2004), even though some programming is nearly universally offered (Gill and Hu, 1998). As a consequence, to assist those students wanting to develop programming expertise, teaching techniques are needed that provide students with a mental model of computing sufficiently sophisticated for the purposes of programming that do not require too much instructional time.

A class of techniques that have long been used to foster deeper understanding of computing is referred to as "glass box" modeling, although other terms such as "white box" and "clear box" are also sometimes used. Glass box models, in the context of teaching computer programming, are intended to provide students with a realistic understanding of the activities that occur within the computer when programs are loaded and program steps are executed. Such models may present a complete view of the various subsystems of the computer (e.g., processor, primary and secondary storage, I/O, etc.) or may focus on a particular system. The present paper reviews some examples of these models. It then presents an exercise, called the memory grid, developed for an introductory programming course in an MIS program. Both the details of the technique and a tool—GridGen, used both to create grids and automate the process of creating online grid exercises and examinations—are described. Some empirical results of using the grid for two years in an introductory programming course are then presented. The paper concludes with some general "lessons learned" based upon the author's experiences using the grid and assessing its effectiveness.

2. GLASS BOX MODELS

Considerable difference of opinion exists regarding whether or not it makes pedagogical sense to teach computer architecture in parallel with programming. The debate is sometimes framed as the choice between a "black box"-whereby a computer is treated solely in terms of inputs and outputs - and a "glass box" - where students are presented with a model of what is occurring within the computer while they are learning to program. The arguments for taking a black box approach generally center around the time it takes to introduce computer concepts while students are concurrently studying programming and the relative lack of hard empirical evidence supporting the value of doing so (Yehezkel *et al.*, 2001). Psychological research, on the other hand, has long argued that the early use of a glass box approach may encourage learning processes that move the student more rapidly from novice to expert and enhance their ability to evolve into creative problem-solvers (Mayer, 1981). The glass box approach is also consistent with the intuition of many instructors regarding how best to teach programming (Yehezkel *et al.*, 2001). As a consequence, a number of glass box techniques have been developed over the past decades.

Glass box techniques generally fall into two categories: simulators and pencil-and-paper exercises. Simulators involve creating a program using a simplified computer that is operated either mechanically or as a virtual machine running on an actual computer. The earliest reported use of the technique is around 1965, with Madnick and Donovan's "Little Man Computer", which involved simulating a computer as a postroom, with each mailbox representing a memory location whose contents were written on a piece of paper (Osborne, 2001). More recently, the trend has been to use virtual machines, running on PCs or other hardware, as the basis for the simulation. Dozens of these simulators have been developed (Wolffe, *et al.*, 2002). In some cases, the tools simulate an actual computer architecture, such as the IBM 360/370 series (Donovan, 1976), AT&T's 3B2 computer, and the IBM PC (EasyCPU, Yehezkel, *et al.*, 2001; PCAS, Gill, 2005b). Other simulators are designed to run simplified computer architectures that were never commercial products, such as variations of the Little Man Computer (Yehezkel, *et al.*, 2001; Osborne, 2001), Computer-1 (Miller, 1983) and RTLSim (Yehezkel, *et al.*, 2001).

A number of educational benefits have been reported from the use of glass box simulators in teaching programming. The visualization of the computer that they offer provides students with insights into computer structures (Yehezkel, *et al.*, 2001). They help students to learn the details of computer organization at multiple levels of abstraction (Wolffe, *et al.*, 2002). They also may improve student motivation (Yehezkel, *et al.*, 2001). In addition, simulators allow instructors to decouple the technologies they present to students from the ever-changing capabilities of computers as they evolve (Wolffe, *et al.*, 2002).

While simulators continue to be widely used in the computer science field, advances in computer technology have rendered them less attractive for technical MIS programs. As recently as the early 1990s, the prevailing operating system (DOS) and commonly used instructional development tools (e.g., Turbo C, Turbo PASCAL, MS QuickC) were sufficiently simple so that the lower level operation of a PC had a practical relevance to programmers. Comprehending the different programming memory models, for example, required a clear understanding of how registers in the Intel 8086 processor family were organized into segments and offsets. Similarly, multi-language programming nearly always required knowledge of assembly language and how arguments were stored on the stack. By the early 1990s, however, complex operating systems such as MS Windows 3.x and integrated development tools such as MS Visual BASIC, MS Visual C++ and, finally, MS Visual Studio, limited the practical relevance of machine-level PC operation for anyone not intending to develop operating systems, development tools or device drivers. With the introduction of Java and the .NET family of languages, the use of software emulators (e.g., the Java virtual machine) and intermediate languages (e.g., the MSIL used for .NET) further increased the distance between the actual PC processor and the languages being used to write programs. Thus—given the shortage of hours available for technical content in typical MIS programs and the decline in the direct benefits of learning lower level programming concepts—it makes sense that, during the 1990s, MIS programs experienced a dramatic decline in coverage of lower level systems concepts, such as assembly language, hardware and operating systems (Gill and Hu, 1998). The utility of full-fledged simulators declined accordingly.

Given the potential psychological value of glass box techniques in the early stages of learning to program (Mayer, 1981), one alternative to a full fledged simulation is to simulate only a portion of the computer. One computer subsystem particularly appropriate for such a simulation is primary storage (i.e., RAM). Such a simulation can be used to introduce students to important issues such as the underlying representation of different data types, the organization of primitive data elements into objects, the mechanics of linking objects together (e.g., pointers) and the fundamental differences between algebraic expressions (e.g., X=3Y+4) and equivalent assignment statements.

A number of different approaches to presenting a glass box view of primary storage have been developed. Some approaches present a very abstracted view of memory (e.g., Holliday and Luginbuhl, 2004; Salvage, 2001), making extensive use of arrows and boxes. Other approaches (e.g., Hair and Mahalakshmi, 2004; Gill, 2005b) focus more closely on actual memory contents, making use of tables to represent byte values. Although most of the support for the pedagogical value of these techniques appears to be anecdotal, some empirical evidence has been presented that demonstrates a high correlation between level of memory diagram understanding and overall understanding of programming concepts (Holliday and Luginbuhl, 2004). The memory grid, the subject of the current paper, is an example of a pencil-and-paper exercise that uses a glass box view of memory contents.

3. THE COURSE

The memory grid was developed as part of a course that introduces programming to undergraduate MIS majors at a large state university using the structured portion of the C++ programming language (to be referred to as C/C++). Over the period described in the paper, the course—required for all majors—had initial enrollments of 70-90 during spring and fall semesters, and 30-40 over the summers. By the end of each semester, roughly 20% of students had withdrawn from the class. Course surveys found that about 50% of the enrolled students had never taken a programming course before, 25% had taken a single prior course and 25% had taken more than one prior course.

To accommodate the large diversity in student backgrounds, the course was taught in a self-paced format. Under this format, a student's entire grade was based on assignment performance (Gill, 2005a). Various validation techniques, including online examinations, oral examinations (mirroring code walkthroughs) and recreation of assignments in a lab environment, were used to ensure the rigor of grading. Using this unique format, nearly all variation in student grades tended to result from percentage of course assignments completed, as scores on individual assignments normally clustered in the 80-100% range.

The choice of C/C++ as an introductory language for the course was motivated primarily by the non-trivial fraction of majors who would ultimately end up working in a programming capacity (end-of-semester surveys suggested that 15-20% thought it moderately likely or very likely that they would be employed as programmers within 10 years of graduation). Since the undergraduate MIS major did not provide for any courses in computer architecture or operating systems, the introductory programming course served as the principal opportunity for acquainting students with the inner workings of a computer. C/C++'s origins as a low-level language for the construction of operating systems were a good fit in this respect. Indeed, students attempting to make sense of various C/C++ language topics, such as arrays, structures, pointers, memory management and file input-output, must, out of necessity, develop a sound conceptual model of how a computer functions.

In using C/C++ as an introductory language, three conceptual areas seemed to be particularly difficult for students to grasp: 1) pointers and dereferencing, 2) the relationship between pointers and arrays, and 3) the syntax of structures (especially when used in conjunction with pointers and when declared as arrays). What these areas have in common is that a mental model of a computer more sophisticated than that of a simple "variable holds value in some undetermined way" is required if students are to apply them effectively. They also represent the aspects of the language that most directly relate to the logical organization of a computer. As a consequence, if students are going to master the pointer/array syntax of C/C++, they need to develop both a realistic model of how data is organized in a computer and they need to be able to apply that model in the context of C/C++ syntax. To address this pedagogical challenge of helping students acquire these higher order thinking skills, the memory grid was developed.

4. THE MEMORY GRID

Memory grids help students to internalize the nature of pointers and structures—and how to use them to access data—by requiring them to map their program representation (i.e., how they are declared within a program) into the way they will be placed in memory. Contrasted with other glass box pencil and paper models (e.g., Holliday and Luginbuhl, 2004; Salvage, 2001; Hair and Mahalakshmi, 2004), memory grids provide a uniquely direct mapping between raw memory and what it represents, forcing students to understand fully the conceptual relationship between declarations and their underlying representation in primary storage.

The format of the grid is similar to that used in a binary file viewer (with bytes presented in hexadecimal, and their ASCII equivalents—if displayable—placed on the right). Beneath the grid is C/C++ code that defines any structures used, followed by a hypothetical layout of memory elements such as variables, arrays and structure objects. An example of such a grid is presented in Figure 1, which is part of an actual assignment that was used in fall semester 2003.

Having been provided with a memory grid setup, students are then given expressions to evaluate. For example, **c1** evaluates to 0x77 in the example, **hdrV1.modified.nday** evaluates to 0x0F, **pvars[1]** evaluates to the address 0x000010B0 and **fld+2** evaluates to the address 0x00001090. Using the tool in this way, students learn to distinguish between:

- Expressions that return addresses vs. expressions that return values
- Alternative notations for accessing the same value (e.g., -> and. for structures, * and [] for addresses).
- Legal and illegal expressions

Practicing with a grid over a week or two can simulate notational complexities that would take years to emerge in actual program settings. In addition, working with the grid also develops expertise that can be used in interpreting and debugging information in binary files. Further details on the use of memory grids can be found in the course textbook (Gill, 2005b).

5. TEACHING USING THE MEMORY GRID

In the introductory programming course where the memory grid was used, the grid assignment was paired with a debugging assignment that required students to examine variables in the Visual Studio .NET debugger. The paired assignment immediately followed the first major programming assignment in the course. Students working on the assignment were concurrently introduced to pointers and structures through the course's textbook and lectures. Upon completing the exercises—done individually or in groups—students were required to take (individually) proctored validation exams. These exams were delivered using the Blackboard course management system and, to have their grades counted, students had to achieve a score consistent with their assignment grade.

A number of practical challenges had to be addressed in order to use memory grids for teaching. First, they could be hard to set up. The original approach had been to take a screen shot from a binary file viewer and then overlay an imaginary collection of data elements (e.g., structures,

```
Address 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F          ASCII
1000    03 65 06 0F 21 00 00 00 22 02 8F 05 4E 6F 74 20         .e..!..."...Not
1010    55 73 65 64 00 CE A5 6A 6E DE C8 4B 2B BF 1B AC         Used...jn..K+...
1020    70 10 00 00 B0 10 00 00 50 10 00 00 90 10 00 00         p.......P.......
1030    57 6F 72 6B 73 20 66 6F 72 20 6D 65 00 45 B5 9C         Works for me.E..
1040    77 A7 7F A9 D0 10 00 00 BA FF 69 04 0B 65 0C 02         w.........i..e..
1050    4C 4E 41 4D 45 00 7D DD A6 FF 89 43 1E 00 01 00         LNAME.}....C....
1060    17 19 83 A3 1A 5C 12 31 04 29 4A E6 06 FD 8A 3C         .....\.1.)J....<
1070    46 4E 41 4D 45 00 09 25 B1 1E 98 43 14 00 1F 00         FNAME..%...C....
1080    1C 29 E4 FA 14 25 85 9E D3 26 08 BD C0 C4 3C 3A         .)...%...&....<:
1090    41 44 44 52 45 53 53 00 3E AB A9 43 28 00 33 00         ADDRESS.>..C(.3.
10a0    CD 38 E4 C6 60 FD 56 6D B9 4F 8C 8B 5A B6 B8 1C         .8..`.Vm.O..Z...
10b0    50 41 59 00 A5 1B F8 57 44 91 D9 4E 0A 02 5B 00         PAY....WD..N..[.
10c0    25 C8 76 1A D8 90 BB B2 1A ED 80 13 23 83 E4 B1         %.v.........#...
10d0    44 4F 48 00 30 F3 1B BE 30 B7 C1 44 08 00 65 00         DOH.0...0..D..e.
10e0    38 C0 4D 6D 26 59 97 8A 1F C8 82 62 0B DF E6 7C         8.Mm&Y.....b...|
```

The declarations that follow map to the start of the above grid. You may assume that you are currently stopped in a position in your program where they have been initialized.

```
struct field {
        char szname[11];
        char ctype;
        unsigned char flen;
        unsigned char fprec;
        short fpos;
        unsigned char reserved[16];
};
struct date {
        unsigned char nyear;
        unsigned char nmonth;
        unsigned char nday;
};
struct header {
        unsigned char cver;
        struct date modified;
        int nrec;
        short reclen;
        short nstart;
        unsigned char junk[20];
};
```

```
struct header hdrV1;
struct field *pvars[4];
char buf[16];
unsigned char c1;
char arc[3];
char *c2;
short v1;
struct date d1[2];
struct field fld[5];
```

Figure 1: Memory Grid Presentation

variables and arrays) over the bytes. Using this approach,however, contents of memory were often inconsistent with the nature of the imaginary data overlay (e.g., a date structure might end up with a month value of 0x45, string pointers might accidentally point to blocks of memory consisting entirely of non-printing characters). Furthermore, using imaginary overlays it was easy - even for experienced instructors - to make careless mistakes in designing and grading them. For example, the author once accidentally defined a 30-byte structure where a 32 byte structure (two even rows) had been intended. The resulting examination proved to be excruciating in its difficulty because of the tedious challenge of figuring out the starting and ending locations of structure elements.

Another problem associated with the use of memory grids as a pedagogical device was that of providing students with a sufficient number of problems to achieve the depth of understanding desired. Because C/C++ allows memory to be accessed in so many different notations, it is no trivial task to come up with a set of questions that systematically presents students with all the different ways in which data in memory can be accessed. The same problem, of course, manifested itself in developing appropriate tests. As a result of these problems, when the grid was first introduced as an exercise, students often chose to ignore the assignment—handing it in as part of a group but never completing grid-related questions on the course's final examination. The assignment was also unpopular, typically ranking 6th or 7th (out of 7 assignments) in perceived value in end of semester surveys.
Counterbalancing the lackluster overall reception to the grid pedagogy was an extremely strong positive attitude towards the assignment among some students—particularly within that group of students most likely to go into programming. For example, whenever the instructor polled the course teaching assistants (all of whom had formerly taken the course) about abandoning the assignment, the universal response was that the assignment should be retained. In fact, several informed the instructor that the assignment had been the most valuable in the course. As a consequence, the instructor decided the only suitable alternative to discarding the assignment was to develop resources that would make the pedagogy more accessible to the typical student, since the conceptual value of the assignment was recognized only by those who had mastered it. The resources developed included:

- Online multimedia lectures that walk students through sample grid exercises, available on the course web site.
- A grid-specific exam that students would have to complete individually if they were to get credit for the assignment, thereby controlling for "passive" group participants.
- A bank of practice questions, accessible on Blackboard at any time, to allow students to assess their individual performance in solving grid problems.

Developing the first of these was relatively straightforward, and took only a few hours. The nature of the grid assignment, however, made the remaining tasks considerably more daunting. The instructor anticipated that it would take dozens of hours of mind-numbing work to create a comprehensive bank of questions and answers that could be used for practice and testing purposes. Furthermore, he anticipated that either: 1) a huge bank of questions would need to be created, or 2) a new grid would have to be generated each semester to avoid problems of answers to test questions being passed on from semester to semester. To address these challenges, the instructor developed the GridGen tool.

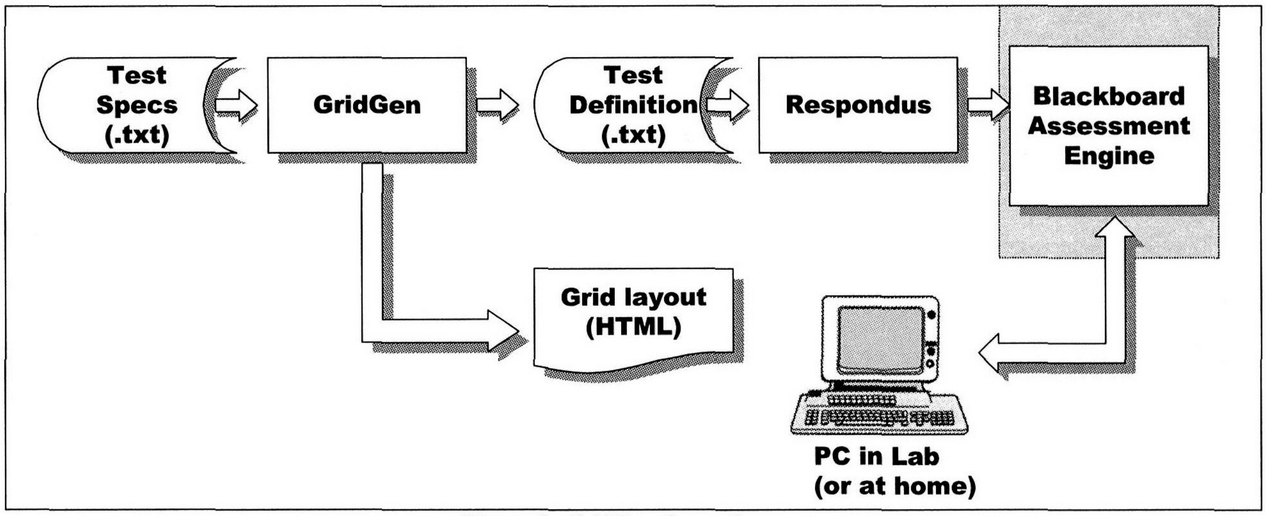


Figure 2: GridGen Operation

6. THE GRIDGEN TOOL

The GridGen tool was developed in spring 2003. It provided the instructor with a number of useful capabilities, including the ability to generate:

1. A customized memory grid, in HTML format, that could be used for assignment purposes
2. C/C++ definitions and declarations precisely matched to the grid contents.
3. A virtually limitless number of questions and answers relating to the specified grid, in a format suitable for upload to most widely used course delivery systems.

Taken together, these capabilities alleviated many of the practical difficulties associated with employing the memory grid pedagogy.

The operation of the GridGen is illustrated in Figure 2. The instructor starts with a grid definition file, which includes various parameters (e.g., how many questions to generate), HTML code for question wording and setup, and grid layout. This file becomes the input to the GridGen program, which then produces two outputs: an HTML file describing the layout (e.g., the file used to create Figure 1) and a text file containing a specified number of fill-in-the-blank questions and answers. The text file can, in turn, be input into a test generation program (Respondus) that can be used to generate paper and online tests.

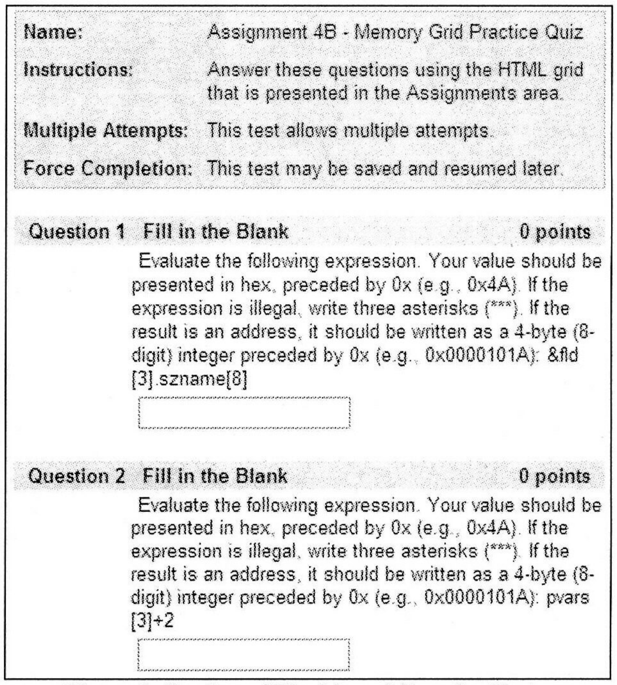
The heart of the GridGen tool is the data layout definition—identifying structure definitions (with initialization instructions), data layouts (with initializations) and a large number of parameters (e.g., what byte value should be used to initialize memory not used, if a given character arrays will be used to hold strings, if a given structure element is included for alignment purposes and is not to be displayed). These definitions allow for the creation of realistic data patterns. For example, the layout of the "date" structure (used in the Figure 1 example) is as follows:

```
DEFINE MSTRUCTURE date 12
unsigned char nyear R 100:105
unsigned char nmonth = 1,2,3,4,5,6,7,8,9,10,11,12
unsigned char nday R 1:31,1:28,1:31,1:30,
                1:31,1:30,1:31,1:31,1:30,1:31,1:30,1:31
END MSTRUCTURE
```

The first line states that 12 possible patterns for any date object are possible (logically, these correspond to months). The second line states that "nyear" values should be randomly chosen from between 100 and 105. The next line states that nmonth is fixed to a value depending upon which pattern is chosen. The final line specifies the range of nday values to be used for random selection for each month. Because nmonth and nday values are always taken from the same pattern number, consistent month and day values are always produced.

GridGen test generation works by using the test specification to create an image of memory, keeping track of the expression name for each byte (e.g., "arc[2]"). It generates fill-in-the-blank questions using a variety of techniques. To come up with a typical byte value question, for example, the program might randomly select a byte from memory and looks up its name. Based on the nature of the name, it then—probabilistically—may choose to transform it. For example, the expression fld[1].szname[3] might be rewritten as *((fld+1)->szname+3) or *(fld[1].szname+3) once it has been transformed.

Using GridGen reduces the incremental cost associated with creating memory grid exercises to nearly zero (especially when an old test specification file is available and can be modified). This allows exercises to be posted on Blackboard for practice purposes (see Figure 3), as well as for examinations.



Name:	Assignment 4B - Memory Grid Practice Quiz
Instructions:	Answer these questions using the HTML grid that is presented in the Assignments area.
Multiple Attempts:	This test allows multiple attempts.
Force Completion:	This test may be saved and resumed later.

Question 1 Fill in the Blank 0 points

Evaluate the following expression. Your value should be presented in hex, preceded by 0x (e.g., 0x4A). If the expression is illegal, write three asterisks (***). If the result is an address, it should be written as a 4-byte (8-digit) integer preceded by 0x (e.g., 0x0000101A): &fld [3].szname[8]

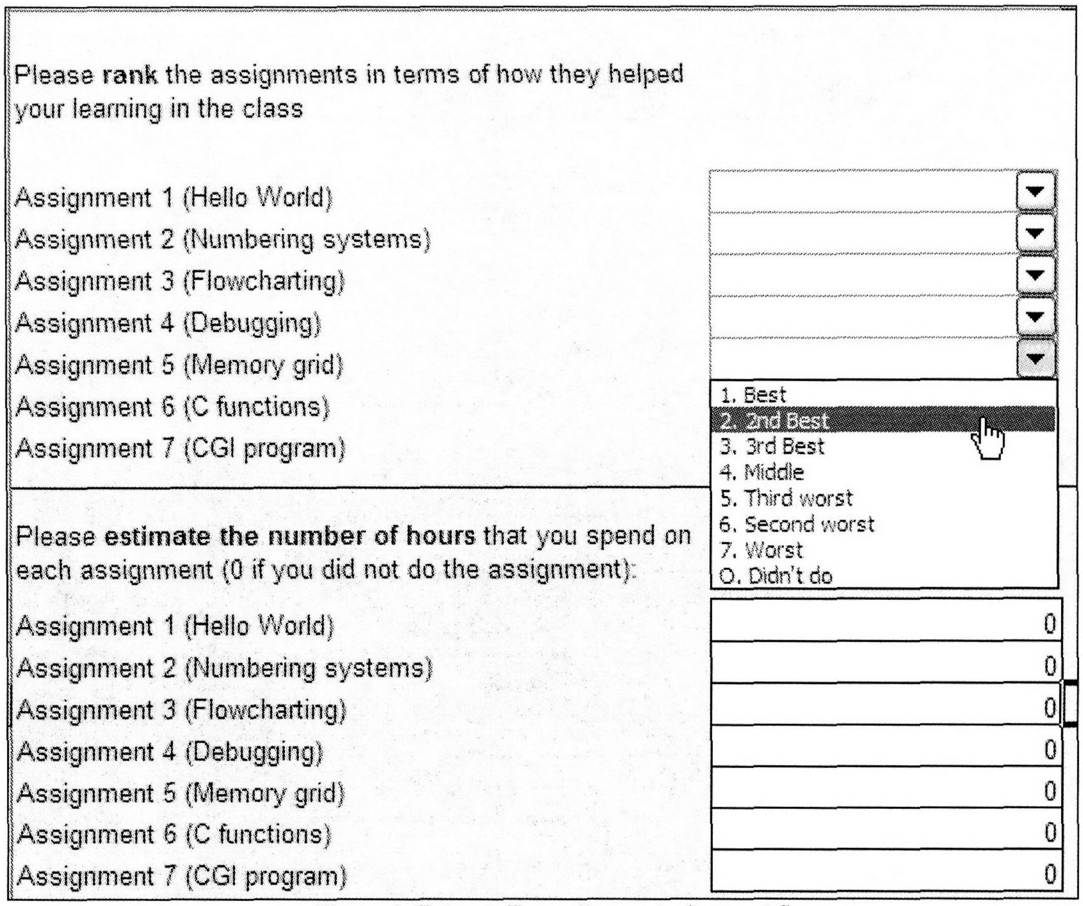
Question 2 Fill in the Blank 0 points

Evaluate the following expression. Your value should be presented in hex, preceded by 0x (e.g., 0x4A). If the expression is illegal, write three asterisks (***). If the result is an address, it should be written as a 4-byte (8-digit) integer preceded by 0x (e.g., 0x0000101A): pvars [3]+2

Figure 3: Portion of Blackboard Practice Test

7. METHODOLOGY

The effectiveness of the memory grid exercise was assessed principally through the use of a comprehensive survey implemented to track all aspects of the course. The survey instrument, developed by the instructor and administered at the end of each semester since spring 2003, consisted of nearly 300 questions. Students completing the voluntary survey, made available in the form of a downloadable spreadsheet that was returned to the instructor's department as an email attachment, received a + added to their course letter grade (e.g., C became C+, B became B+, A became A+). As a consequence of this incentive, response rates of roughly 70% of active students were normally achieved. The survey was not anonymous but completed surveys were directed to an administrative mailbox that the instructor



Please **rank** the assignments in terms of how they helped your learning in the class

Assignment 1 (Hello World)	
Assignment 2 (Numbering systems)	
Assignment 3 (Flowcharting)	
Assignment 4 (Debugging)	
Assignment 5 (Memory grid)	
Assignment 6 (C functions)	
Assignment 7 (CGI program)	

Please **estimate the number of hours** that you spend on each assignment (0 if you did not do the assignment):

Assignment 1 (Hello World)	0
Assignment 2 (Numbering systems)	0
Assignment 3 (Flowcharting)	0
Assignment 4 (Debugging)	0
Assignment 5 (Memory grid)	0
Assignment 6 (C functions)	0
Assignment 7 (CGI program)	0

Figure 4: Extracts From Course Assignment Survey

could not access. Immediately prior to submission of course grades, the departmental secretary provided the instructor with a list of students who had responded, so grades could be adjusted. Copies of the completed surveys, on a CD, were made available to the instructor only after course grades had been submitted to the registrar. The same procedure was used for the 7 question university course evaluations, since the course's self-paced format and flexible schedule meant that traditional end-of-semester class meetings (used primarily for filling out course evaluations) did not take place.

The instrument was designed to elicit information on student background (e.g., programming experience, work experience, career aspirations), reaction to individual assignments, satisfaction with course design and perceived learning gains. It was developed as a composite of three NSF-recommended evaluation instruments, the Student Opinion Survey, the Computer Programming Survey and the Student Assessment of Learning Gains, with additional questions added to assess student background and experience. (Links to source surveys are provided in "Instrument References" section at the end of the article). Selected portions of the survey relating to assignments are presented in Figure 4.

8. RESULTS

Prior to the introduction of online testing and practice exams, the relatively low weight of memory grid exercises in the entire course (~5% of total points) and the challenge presented by learning the material led most students to ignore the assignment. While many students handed it in, typically as part of a group, very few actually completed final examination questions relating to the grid—suggesting they had not fully understood what they were handing in. This conclusion of the instructor seemed to be confirmed by the very low popularity of the assignment, with most students ranking it at or near the bottom in end of semester surveys. Even the introduction of online practice tests, in spring 2003, did little to change this.

In fall 2003, course requirements were changed so that online testing (using a GridGen created test bank) was required in order to get assignment credit. Paradoxically, one impact of this new requirement was to lead to significant increases in assignment completion rate—to nearly 90% of those students who completed the course in fall 2003. Even more noteworthy, however, was a change in how student perceived the value of the assignment when compared with other assignments. As shown in Table 1, the implementation

	Spring 2003 & Summer 2003 (pre-validation requirement)	Fall 2003 & Spring 2004 (post-validation requirement)
Number of students surveyed	65	73
Mean ranking (s.d.) of assignment, out of 7 assignments	5.1 (1.8)	2.7 (1.4)
Rank score compared to other assignments	Last (worst)	Top (best)
Percent ranking assignment as the top (best) assignment	3%	25%
Percent ranking assignment in the top 3 (of 7) assignments	18%	69%
Percent ranking assignment in the bottom 3 (of 7) assignments	58%	3%

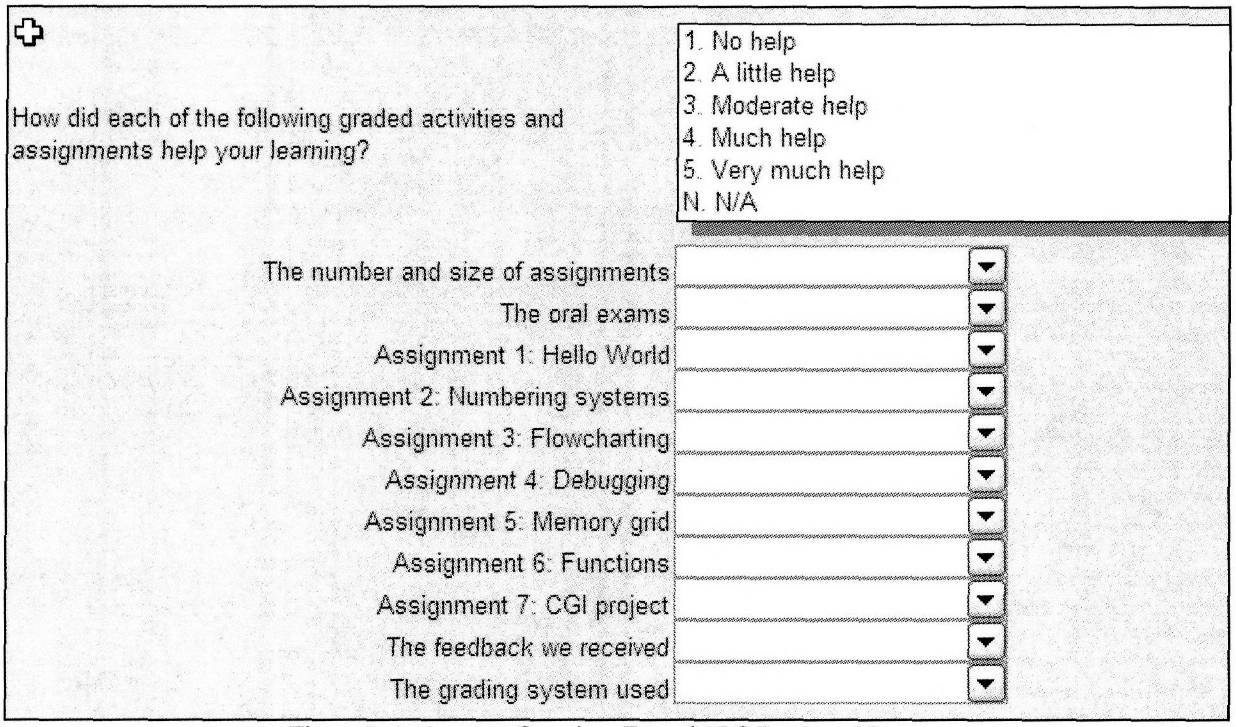
Table 1: Outcomes of GridGen Implementation

of validation testing was accompanied by a dramatic reshuffling of perceptions regarding the assignment. Once students were required to individually validate the assignment—making passive participation in a group impossible—their perception of the value of the assignment changed. The mean ranking (2.7) made the assignment the most highly ranked of the course's 7 assignments in fall 2003/spring 2004, as compared with being the lowest ranking assignment (5.1) in spring/summer 2003, prior to the initiation of GridGen-enabled online validation. With respect to the distribution of perceptions, in fall 2003/spring 2004 25% of students ranked it the best assignment and more than 69% ranked it in the top 3—as compared with 3% and 18% in spring/summer 2003.

Further confirmation of these findings could be found from questions derived from the Student Assessment of Learning Gains (SALG) section of the survey. In this section, students were asked to rate how various aspects of the course impacted their learning on a No Help (1) to Very Much Help (5) scale. As shown in Figure 5, one section of the SALG portion of the survey specifically addressed individual assignments.

Both before and after the GridGen implementation, the mean score for all completed assignments was approximately 3.3 (3.27 pre-adoption and post-adoption 3.36). Pre-adoption, the assignment's value of 3.32 was indistinguishable from the mean of all assignments. For the two semesters following adoption, however, the assignment's rating of 3.66 was significantly different from the all-assignment mean of 3.36, as illustrated in Table 2, and comparable to that of the two most popular programming assignments (Assignments 3 and 5).

Correlation analysis of SALG data over the entire period of measurement (7 semesters, N=216) also suggested the particular area where the memory grid/debugging assignment made its greatest contribution. In another area of the SALG survey section, students were asked to rate the


How did each of the following graded activities and assignments help your learning?

1. No help
2. A little help
3. Moderate help
4. Much help
5. Very much help
N. N/A

The number and size of assignments
The oral exams
Assignment 1: Hello World
Assignment 2: Numbering systems
Assignment 3: Flowcharting
Assignment 4: Debugging
Assignment 5: Memory grid
Assignment 6: Functions
Assignment 7: CGI project
The feedback we received
The grading system used

Figure 5: Assignment Questions From SALG Portion of Survey

Assignment	Responses	Mean of SALG Rating	T-statistic	Significance (2-tailed)
1. Hello World	64	2.92	-3.02	.00**
2. Numbering Systems	63	3.11	-2.00	.05*
3. Flowcharting	64	3.56	1.24	.22
4. Memory Grid and Debugging	**62**	**3.66**	**2.72**	**.01****
5. C/C++ Functions	45	3.67	1.68	.10
6. CGI Application	30	3.33	-.11	.92
Comparison of individual assignment means with weighted average mean of 3.36 (Significances: * p<0.05, ** p<0.01)				

Table 2: SALG Results for Memory Grid Assignment

degree to which the course as a whole was helpful in developing understanding in three areas: 1) flowcharting, 2) how computers are organized, and 3) the nature of computer programming. Although correlations existed between the perception that the grid portion of the assignment was helpful and all three skills (0.253, $p<0.001$ with flowcharting; 0.290, $p<0.001$ with computer organization; 0.182, $p<0.01$ with programming) that relationship was strongest for understanding computer organization—the design goal of the exercise.

9. DISCUSSION

In considering the results presented, there are two issues that warrant further discussion. The first relates to the degree to which observed improvements in memory grid ranking can be directly attributed to better understanding of the grid, versus other effects. The second relates to the potential applicability of the grid to languages outside of C/C++.

9.1 Process vs. Understanding Changes

One reasonable concern in interpreting these results is separating the impact of process changes (i.e., creating practice tests and requiring online validations) from that of improved understanding of assignment content (i.e., the memory grid itself) when explaining the change in rankings. In addressing this issue, a reasonable case can be made that the improved rankings of the grid exercise were much more directly related to student understanding of actual content than to delivery method. The basis for this conclusion relates to the fact that while the memory grid changes were being instituted, similar protocol changes were being put in place for another pencil and paper assignment dealing with numbering systems (e.g., base conversions, hexadecimal, twos complement). Specifically, the program NumGen (Gill 2005b) was also created in spring 2003 to generate numbering system exercise validation exams, with practice exams being made available in spring 2003 and required validation being instituted in fall 2004. Thus, to the extent that protocol changes were purely responsible for changes in rankings, similar effects should have been observed for both assignments.

As shown in Table 3, the ranking changes that accompanied the two concurrent course protocol changes were far more pronounced for the memory grid assignment than they were for the numbering system assignment. While a reasonable case can be made that some positive impact was observed for the numbering system assignment—its overall rank improved slightly despite the fact that the memory grid moved in front of it and its SALG rating improved from 2.96 to 3.09—neither change was anywhere near as dramatic as the corresponding change for the memory grid. The instructor's interpretation of these findings, heavily influenced by his experience grading numbering system and grid questions on final exams in previous semesters, was as follows:

- Prior to the GridGen/NumGen adoption, most students had mastered numbering systems by the end of the course, but only a few had really understood memory grids.
- While the new protocol made learning numbering systems a bit easier (leading to a slight ranking improvement), it had not led to any dramatic change in ultimate understanding.
- For the memory grid, however, ranking changes were primarily the result of a change in appreciation of the value of the assignment that accompanied greater levels of understanding.

While it is certainly possible that other factors impacted the change in rankings (e.g., summer 2003 was an accelerated 10 week semester, as opposed to a 15 week semester), the fact remains that the memory grid became the only non-coding assignment to achieve rankings comparable to that of programming exercises.

9.2 Generalizability

With respect to generalizability of the grid pedagogy beyond C/C++ language courses, there are obvious limitations. Others (e.g., Hair and Mahalakshmi, 2004) have noted that memory exercises seem to have special practical relevance to lower level languages such as C and C++. While such languages have recently been offered in most MIS programs (e.g., 77% of U.S. undergraduate programs surveyed in 1997; Gill and Hu, 1998), they also appear to have declined significantly in relative importance in recent years, replaced by Java or through the elimination of programming requirements altogether. Two responses can be made to these

	Spring 2003 Ranking (S.D.) Pre-validation	Summer 2003 Ranking (S.D.) Pre-validation	Fall 2003 Ranking (S.D.) Post-validation	Spring 2004 Ranking (S.D.) Post-validation
Assignment 2 (Numbering Systems)	4.39 (1.6)	4.12 (1.6)	**3.77 (1.6)**	**4.02 (1.7)**
Assignment 4/5 (Memory Grid)	5.14 (1.6)	4.91 (2.2)	**2.29 (1.4)**	**3.08 (1.4)**

Table 3: Grid vs. Numbering System Assignment Rankings (s.d.) Before and After Validation

concerns. First, while the potential relevance of the grid exercise in the U.S. may be decreasing, there is still strong demand for low level language instruction internationally, particularly in Asia (e.g., Hair and Mahalakshmi, 2004). Furthermore, even in the U.S., industry leaders (e.g., Gates and Klawe, 2005) have voiced concerns relating to learning to program exclusively in garbage-collected languages, which hide the underlying process by which chunks of memory are acquired from the operating system and returned to it once the chunks are no longer being used. Relying solely on these languages (e.g., Java and Visual Basic), it is feared, can lead to students who lack an understanding of underlying issues—such as memory management—that prove to be critical as program size and complexity reach "real world" scale. Such complaints, coming from industry, could motivate greater emphasis on computer architecture in U.S. curricula, perhaps leading a resurgence of interest in C/C++ (at least as an elective option).

A second possibility for generalizability would be to adapt GridGen to generate memory exercises in other languages, or in a language-neutral form. Such modifications would be relatively straightforward. Towards this end, the source code for the GridGen application has been made available to instructors (provided, for example, in the instructor's manual CD for Gill, 2005b). For some widely used languages, such as COBOL, FORTRAN, PL/1 and PASCAL, many aspects of the memory grid remain relevant since a fairly straightforward correspondence between memory and variable location exists for these languages.

A more substantial modification—whereby GridGen's objectives could be refocused towards abstract memory diagrams (e.g., box and arrow) more suitable for teaching languages such as C# or Java (e.g., Holliday and Luginbuhl, 2004)—can also be envisioned. In this scenario, the program would generate diagrams (as opposed to hexadecimal grids) corresponding to the layouts specified by the instructor, and students would be required to relate expressions to their locations in the diagrams. The feasibility and potential benefits of such a graphical approach have already, to a certain extent, been demonstrated by the earlier development and successful implementation of flowcharting software (Gill, 2004) in the same introductory course where the memory grid was developed.

10. CONCLUSIONS

Beyond the mechanics of its implementation, the example of the memory grid is intended convey three important lessons to readers. The first involves the potential learning benefits that can be realized from mixing concrete and abstract exercises when teaching programming. Unless MIS departments want to abandon programming altogether as a viable career option, we need to come up with tools—such as the memory grid—that teach our students the fundamental computer concepts they need to know in order to compete effectively with their computer science counterparts, both at home and abroad.

The second lesson relates to how the GridGen application enhanced the use of the assignment in two ways. First, it increased instructor productivity through reducing the time required for assignment creation and grading. Second, it enhanced instructional effectiveness by making it possible to provide huge banks of practice questions to students. Instructors, particularly those with a modicum of programming skill, should not be afraid to apply their craft creatively to develop exercises and testing materials. The combined development time (in spring 2003) of GridGen and NumGen was about 4 days. That setup cost now seems trivial when balanced against the time savings and educational benefits that accrued in the two years that followed. Even instructors for whom the memory grid does not seem to be directly relevant might benefit from examining the GridGen architecture and assessing its applicability to their own teaching challenges.

The final lesson is the potential value to us, as instructors, of continuously assessing the consequences of each innovation we introduce into a course. As noted previously, many of the studies of glass box assignments conducted in the past provided mainly anecdotal evidence of effectiveness. Such lack of more rigorous direct evidence is understandable since standardized university teaching evaluations are normally not specific enough to measure the consequences of changes to individual course components, particularly those lasting only a week or so in a semester-long course. Indeed, had these been the only source of information available for the present study, it would have been impossible to identify whether or not the memory grid exercise had been effective. Furthermore, we cannot usually make such assessments retroactively. It is therefore incumbent upon each instructor (who wants introduce innovations into the classroom) to establish baseline measurements that can be used as a basis for comparison as new techniques are implemented. Not only do such measures aid the instructor in judging effectiveness, they can prove to be indispensable in the process of communicating these innovations to others.

11. REFERENCES

Donovan, J.H. (1976) "Tools and Philosophy for Software Education". Communications of the ACM. Vol. 19, No. 8. August. pp. 430-436.

Gates, W. and Klawe, M. (2005) "Remarks by Bill Gates, Chairman and Chief Software Architect, Microsoft Corporation and Maria Klawe, Dean of Engineering and Applied Science, Princeton University, Microsoft Research Faculty Summit 2005". Accessed 8/7/2005 at: http://www.microsoft.com/billgates/speeches/2005/07-18FacultySummit.asp

Gill, T.G. (2004) "Teaching Flowcharting Using FlowC". Journal of IS Education. Vol. 15, No. 1. pp. 65-78.

Gill, T.G. (2005a) "Assignment-Centric Design: Testing the Assignments, Not the Lectures". Decision Sciences Journal of Innovative Education. Vol. 3, No. 2. pp. 339-346.

Gill, T.G. (2005b) Introduction to Programming Using Visual C++.NET. Hoboken, NJ: Wiley.

Gill, T. and Q. Hu. (1998) "Information Systems Education in the USA," Education and Information Technologies. Vol 3. pp. 119-136.

Holliday, M.A. and D. Luginbuhl. (2004) "CS1 Assessment Using Memory Diagrams". SIGCSE'04, March 3-7, Norfolk, Virginia, USA. pp. 200-204.

Mayer, R.E. (1981) "The Psychology of How Novices Learn Computer Programming". Computing Surveys. Vol. 18, No. 1. pp. 121-141.

Miller, D.S. (1983) "COMPUTER-l -- A Modern Simple Computer to Introduce Computer Organization and Assembler Language Programming". ACM SIGCSE Bulletin, Proceedings of the Fourteenth SIGCSE Technical Symposium On Computer Science Education. Vol 15, No. 1. February. pp. 271-277.

Osborne, H. (2001) "The Postroom Computer". Journal of Educational Resources in Computing. Vol. 1, No. 4, December, pp. 81–110.

Reichgelt, H., B. Lunt, T. Ashford, A. Phelps, E. Slazinski, and C. Willis. (2004) "A Comparison of Baccalaureate Programs in Information Technology with Baccalaureate Programs in Computer Science and Information Systems," Journal of IT Education. Vol. 3.

Salvage, J. (2001) C++ Coach: Essentials for Introductory Programming. Boston, MA: Addison Wesley.

Wolffe, G.S., W. Yurcik, H. Osborne and M.A. Holliday (2002) "Teaching computer organization/architecture with limited resources using simulators". ACM SIGCSE Bulletin, Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education. Vol. 34, No. 1. February. pp. 176-180.

Yehezkel, C., W. Yurcik, M. Pearson and D. Armstrong (2001) "Three Simulator Tools for Teaching Computer Architecture: EasyCPU, Little Man Computer, and RTLSim". ACM Journal of Educational Resources in Computing. Vol. 1, No. 4. December. pp. 60-80.

12. INSTRUMENTS REFERENCED

"Student Opinion Survey"
http://oerl.sri.com/instruments/cd/studcourse/instr16.html
accessed on 4/14/2003

"Computer Programming Survey"
http://oerl.sri.com/instruments/cd/studcourse/instr11.html
accessed on 4/14/2003

"Student Assessment of Learning Gains (SALG)"
http://www.wcer.wisc.edu/salgains/instructor/
accessed on 4/14/2003

Note: Instructors may acquire copies of the survey instrument used for the course from the author, as well as GridGen and operating instructions from the author. The latter are also included with the instructor's manual of Gill (2005b).

AUTHOR BIOGRAPHY

T. Grandon Gill is an Associate professor in Information Systems and Decision Sciences at the University of South Florida. He received his A.B., *cum laude,* from Harvard College, and both his MBA (*high distinction)* and DBA from Harvard Business School. He currently teaches courses at the undergraduate, masters and doctoral levels, as well as leading faculty seminars at USF's *Center for 21st Century Teaching Excellence.* His teaching focus includes courses in programming, databases, instructional technologies and case method techniques. His current research involves the impact of instructional information technologies on IS education, where he has published extensively in IS education-related journals, such as the *Journal of IS Education*, the *Decision Sciences Journal of Innovative Education*, and *Education and Information Technologies.* He has also published a textbook on C++ programming with *Wiley,* developed a complete line of laminated study guides for *Barcharts, Inc.* and has authored numerous published case studies.