Tools for Creating Portable Demonstration Programs

Eric Roberts
Department of Computer Science
Stanford University
eroberts@cs.stanford.edu

Abstract

This paper describes a collection of software tools called the **csdemo** package, which is designed to support development of interactive programs that can be used both as lecture demonstrations and as hands-on tools for increasing student comprehension. Because the package is based on a graphics library that has been implemented for a variety of platforms, the programs generated by the **csdemo** package are highly portable and not restricted to a single computing environment.

1. Introduction

Over the last five years, technological improvements have made it much easier to incorporate graphics and algorithmic visualization into the teaching process. The computers used by students today are considerably more powerful than the previous generation of machines and typically have the processor speed and memory necessary to support color and animation. Many classrooms are equipped with high-quality projection systems that enable the use of interactive displays in lecture as well. The hardware technology, however, is only part of the problem that needs to be solved. Writing the software necessary to implement effective demonstration programs is a difficult task, particularly if implementers intend those programs to be used portably on a variety of platforms. This paper describes a set of library interfaces—collectively called the **csdemo** package—that facilitates the creation of portable, interactive demonstration programs for teaching computer science.

The structure of applications created using the **csdemo** package differs from that of slide shows generated by commercial software packages. Commercial tools such as PowerPoint™ make it easy to present a preconstructed series of example slides that illustrate a particular concept. By contrast, **csdemo** applications are developed by someone with computer science expertise—typically the instructor. These applications usually begin with the programs the students are studying and then extend them to include graphical animation. For example, if the students are learning about sorting algorithms, the instructor can insert **csdemo** calls into the sorting programs in such a way that the extended programs illustrate their own operation.

The advantage of the **csdemo** approach over the PowerPoint methodology is that the user has much greater control over the resulting program. For example, it is easy to change the input data, control the pace of the presentation, and focus on precisely those parts of the program that are the source of confusion. The **csdemo** package is therefore more closely aligned in structure with such algorithm-animation tools as Balsa [1] and Zeus [2]. The **csdemo** package, however, is more portable than these earlier algorithm-animation tools because it is layered on top of a multiplatform graphics library [4].

2. Examples of demonstration programs

During the past year, instructors in Stanford's introductory computer science course have used the **csdemo** package to create more than 40 distinct demonstration programs, including those listed in Figure 1. Using the demo in class allows the instructor to illustrate the material dynamically in a fraction of the time required to write examples on the

Figure 1. Demo programs used in CS1/CS2

- Expression evaluation
- Truth tables for Boolean operators
- Control statements and program flow
- The function-calling mechanism
- Euclid's algorithm
- The **graphics.h** interface
- Random number scaling
- String manipulation
- Internal operation of a token scanner
- Array operations
- File manipulation
- The relationship between pointers and arrays
- Data structure design
- Sorting and searching algorithms
- Simple recursion
- Stack implementation of the Towers of Hanoi
- Computational complexity
- Recursive maze solution
- Random maze generation
- Minimax strategies for games
- Memory models for linked lists
- ADTs for stacks, queues, sets, and graphs
- Hashing
- Binary search trees
- Parsing arithmetic expressions

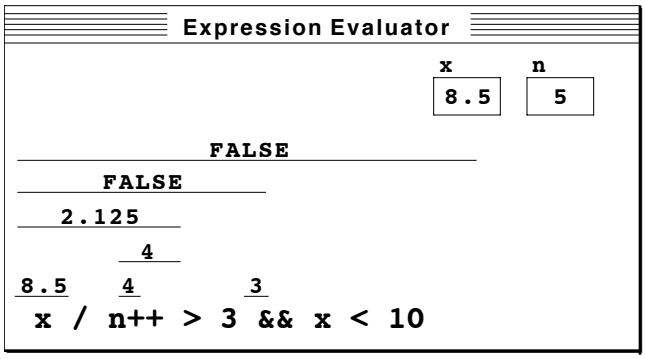
Integrating Tech. into C.S.E. 6/96 Barcelona, Spain
© 1996 ACM 0-89791-844-4/96/0009...$3.50

board. The same demo program can also be made available to students, who are then able to run it on their own. Because students control the operation and pace of the demo, the programs are quite responsive to the individual student's needs.

The fact that demonstration programs are useful to both lecturers and students is illustrated by the **ExpTrace** application, which reads ANSI C expressions from the user and diagrams their evaluation. For example, assuming that previous statements have initialized the floating-point variable **x** to 8.5 and the integer **n** to 4, entering the expression

```
x / n++ > 3 && x < 10
```

generates the following diagram on the **ExpTrace** screen:



The **ExpTrace** example illustrates many features of C's evaluation rules, including precedence, automatic type conversion, the effect of the postincrement **++** operator, and short-circuit evaluation of **&&**. Moreover, it presents the results of the conditional operators as the Boolean values **TRUE** and **FALSE** rather than as their underlying representation as integers, which is consistent with our pedagogical presentation. Using the **ExpTrace** program in lecture makes it easier to get these concepts across, but the real advantage comes from the fact that students can use the program to enhance their own understanding. In our experience over the past year, students seem to have fewer problems understanding the process of expression evaluation, although it is impossible to prove conclusively that the existence of the **ExpTrace** program is responsible for the increased level of comprehension.

3. Evolution of the csdemo package

The **ExpTrace** demo was one of the first programs to take advantage of the portable graphics library and was written before the **csdemo** package came into existence. After seeing how valuable it was as a pedagogical tool, I decided to create other demonstration programs that could illustrate additional aspects of the introductory CS curriculum, many of which are listed in Figure 1. In the process of putting those tools together, it quickly became clear that many of the demo programs had a common structure that would allow them to share a large fraction of the code. As a result, I set out to design a set of library interfaces that would facilitate the construction of demonstration programs suitable for classroom use.

The fundamental enabling technology behind the **csdemo** package is the portable graphics library described in the textbook developed for the class [3] and a 1995 SIGCSE paper [4]. The graphics library consists of two interfaces—**graphics.h** and **extgraph.h**—that together define a graphics abstraction that can be implemented easily on a wide variety of platforms. These implementations, which cover the most widely distributed platforms used by the educational community, provide a reasonable level of portability because it is possible to compile the same application code for many different environments. By building all higher-level functionality on top of this common abstraction layer, the entire **csdemo** package retains the portability provided by its base, because all higher-level code is independent of any specific platform.

The next level of the abstraction hierarchy consists of several interfaces that act as tools for the construction of the user interface, such as buttons and scrollable text areas. Although implementing these functions on top of the graphics library is less efficient than using the native facilities available on each platform, the increase in portability justifies the cost, particularly because the subjective impact on the user is small.

The **csdemo** package is the next layer in the abstraction hierarchy and consists of several interfaces to support the creation of demonstration programs. The current set of interfaces includes the following:

- **fntrace.h**—This interface supports interactive code tracing, which allows the user to follow the operation of a program as it proceeds from statement to statement or through nested function calls. The package draws the stack frames, keeps track of local variables, and highlights individual lines of code as they are executed. Section 4 illustrates the use of the **fntrace.h** interface in the context of a program to trace recursive functions.
- **memtrace.h**—This interface supports the creation of memory diagrams that track the low-level contents of memory. The client uses the functions provided by the package to define a memory map showing how memory is allocated. Once established, the abstraction itself keeps track of the graphical display, including drawing pointers from one memory cell to another so that new pointers don't obscure other links already displayed on the screen.
- **filetrace.h**—This interface animates the operation of the standard I/O file-management facility. The functions in this interface parallel those in **stdio.h** but also record their operation graphically on the screen.
- **stackadt.h**—This interface allows the client to define a stack that traces its own state on the screen as it responds to **Push** and **Pop** operations generated by the client. Similar interfaces exist for tracing queues and binary search trees.

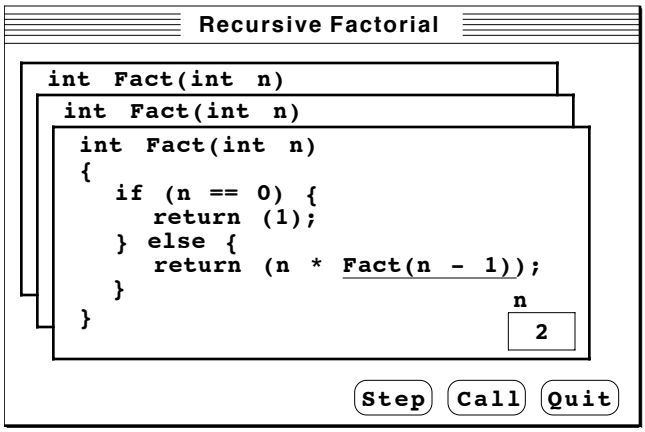
4. Creating a demonstration program

The best way to understand the **csdemo** package is to walk through the construction of a simple demo. Consider, for example, the task of creating a program that lets the student

trace the evaluation of the following recursive implementation of the factorial function:

```
int Fact(int n)
{
  if (n == 0) {
    return (1);
  } else {
    return(n * Fact(n - 1));
  }
}
```

The finished demo program should provide a complete graphical trace of the operation that allows the student to step through the program line by line as it enters each recursive invocation of the function. At each point in the execution, the display indicates the current state of the computation and allows the user to decide how to proceed. The following diagram, for example, shows the computation of **Fact(4)** after two recursive calls have been made, just as the program is about to make its next recursive call:



If the student clicks the **Step** button, the program will create the new stack frame and stop again at the entry to the next function level; clicking the **Call** button executes the entire function without stepping through it in detail.

With the **csdemo** package, generating a demo program of this sort is straightforward because the frame creation, program sequencing, and user interactivity are supported by the **fntrace.h** interface. The first step in the process is to modify the code for **Fact** so that it includes calls to trace its own execution. The extended code is shown in Figure 2. The calls to **MarkStatement** and **MarkExpression** specify the points at which control returns to the user and indicate what line or expression is highlighted. The other calls declare the parameter variable and manage the call and return logic. Because of the restrictions of C's type system, all arguments and results are declared as strings and must be converted internally to the proper type.

After extending the **Fact** function, the next step is to register **Fact** with the **fntrace** module, passing along a null-terminated array of strings containing the original code. The final step is to use the **Call** facility to invoke the **Fact** function on its initial arguments:

Figure 2. Fact function extended with tracing calls

```
static void Fact(void)
{
  varT vn;
  string f;
  int n;

  vn = DeclareParameter("n");
  n = GetIntegerValue(vn);
  MarkStatement(2);
  if (n == 0) {
    MarkStatement(3);
    SetIntegerResult(1);
  } else {
    MarkStatement(5);
    MarkExpression(5, "Fact(n - 1)");
    f = Call(Fact, IntegerToString(n - 1));
    SetIntegerResult(n * StringToInteger(f));
  }
  MarkStatement(7);
}
```

```
(void) Call(Fact, IntegerToString(4));
```

From this point, the operation of the demo is automatic, because the **fntrace** module takes care of the details.

5. Conclusions and future directions

The **csdemo** package has greatly simplified the process of creating demo programs for use in class. On many occasions over the past year, I've put together in an hour an effective demonstration program that would have taken several days to implement without these tools. More importantly, end-of-quarter evaluations indicate that students find these programs helpful, both in enhancing their in-class understanding and as tools for independent study.

On the basis of its success in the last year, we intend to continue using this package to create demo programs for the introductory course, and further plan to undertake the following activities:

- Expand the **csdemo** package by adding new capabilities
- Devise a mechanism for public distribution and support of the existing demonstration programs
- Explore various strategies for reimplementing these tools for distributed use with the World-Wide Web

For more information on the **csdemo** package and related pedagogical tools at Stanford, please consult our web site at **http://www-cs-education.stanford.edu**.

References

1. Brown, Marc H. and Robert Sedgewick. A system for algorithm animation. *Computer Graphics,* Vol. 18, No. 3, 1984.
2. Brown, Marc H. Zeus: a system for algorithm animation. *Proceedings of the 1991 Workshop on Visual Languages,* October 1991.
3. Roberts, Eric S. *The Art and Science of C: A Library-Based Approach,* Reading, MA: Addison-Wesley, 1995.
4. Roberts, Eric S. A C-based graphics library for CS1. *SIGCSE Bulletin,* Vol. 27, No. 1, March 1995.