USING THE FORCE: HOW STAR WARS CAN HELP YOU TEACH RECURSION

Jennifer S. Kay
Department of Computer Science
Rowan University
201 Mullica Hill Road
Glassboro, NJ 08028-1701
(856) 256-4593
kay@rowan.edu

ABSTRACT

Most students in a CS1 or CS2 class find the concept of recursion unnatural and difficult. This paper presents a unique approach to the task. Students are taught a method of writing recursive functions that is more algorithmic than the traditional approach. By breaking the writing of a recursive function into a sequence of sub-tasks, problems do not seem quite as intimidating. The approach incorporates a concept from a series of major motion pictures that makes it moderately entertaining for the students, while helping them with the process.

In addition to describing this approach to writing recursive programs, this paper also suggests a sequence of examples that build upon each other to help students practice using this method. These examples include some of the "classic" recursion examples found in today's textbooks, as well as some non-traditional examples that have proven successful in the classroom.

1. INTRODUCTION

Teaching recursion in a CS1 or CS2 class can be a real challenge. For most students this is a subject that does not seem natural or intuitive. The traditional methods of using a series of (often "classic") examples are not always successful. Students may feel moderately comfortable with one function (e.g. a power function), and yet be unable to solve what seems like a very similar problem (e.g. factorial).

This paper presents a way to introduce students to recursion using a four-step approach:

Present the students with a single "classic" example of recursion.

JCSC 15, 5 (May 2000)
© 2000 by the Consortium for Computing in Small Colleges

Teach the students to write a recursive function using a specific sequence of rules that must always be followed. Begin with very simple examples and graduate slowly to the classics.

Once they are comfortable with following the rules to write recursive functions, go through one of the simple examples to demonstrate how recursion works.

Present the students with one final "classic" recursion problem. Have the students solve the problem, and then demonstrate how their solution works.

The key to the success of this approach, as well as its uniqueness, is step 2. By presenting the students with a specific sequence of steps to follow when writing a recursive function, their discomfort about the use of recursion is reduced. In addition, the sequence of simple examples presented below seems to be a particularly effective, and pain-free, route to follow. The examples presented in this paper are in C++, but are appropriate to most other languages used for CS1 and CS2.

2. CURRENT TEXTBOOK "CLASSICS"

Having made the claim that the traditional approach is to use "the classics," it is worth taking some time to review what is currently being presented in our textbooks.

A survey of several C++ textbooks [1] [2] [3] [4] [5] [6] will convince most people that the classics are still in vogue. The exact definition of a classic is elusive, but the fact that these should all be instantly recognizable given only a brief title seems sufficient to argue that these are indeed classics.

The following examples were described in the text (not exercises) of at least two textbooks:

Factorial [3] [4] [6]
Power [1] [5] [6]
Binary Search [1] [6]
Fibonacci [4] [6]
Towers of Hanoi [2] [4]

Several of the examples that appeared only in a single textbook are still "classics"

Eight Queens [4]
Palindrome Checker [6]
Quicksort [1]

In addition several textbooks presented quite advanced examples which require more fuller explanations:

Look ahead in game trees [4]
Fractals and Mazes [5]

Finally, one textbook presented a unique example which is moderately simple and worth mentioning:

Printing out the digits of an integer vertically [5]

3. TEACHING RECURSION USING THE FOUR-STEP APPROACH

The four-step approach that is presented below has been used in CS1 and CS2 Undergraduate classes, as well as in a Masters-Level “bridge” course designed as a quick introduction for students who wish to pursue Masters degrees in Computer Science but have an insufficient undergraduate background. It works well, and the addition of the Star Wars theme even helps to motivate (albeit briefly) the night owls in 9 a.m. classes.

3.1 Step 1: Start with an Obvious Classic

Begin your lecture in the traditional way. Review the definition of factorial, and then ask the students, “suppose I wanted to compute the factorial of 27, and I already knew the factorial of 26, would that help me?” “Suppose I wanted to compute the factorial of 54, but knew the factorial of 53, would that help me?”

Once your students are comfortable with thinking recursively about factorial, present them with the standard code for factorial:

```
int fact (int num)
{
    if (num == 1)
        return (1);
    else
        return (num * fact(num-1));
}
```

Ask the students, “Pretend for a minute that a function could call itself, and so this code is legal and would compile. (I know it seems weird, but just bear with me) If you assume that a function can call itself, do you believe that this would work?”

Most students are willing to agree, given the condition that they are not required to believe that the technique works. It can be interesting at this point to have the class vote as to whether they believe the code would actually compile.

Finally, tell the students that it really does work, give them a simple definition of a recursive function (e.g. “a function that calls itself”) and give them a roadmap of how you will proceed for the rest of this topic:

1. “We will begin by learning some rules that will enable us to write recursive functions that work. As we write these, your job is to simply follow the rules. If you follow the rules, you will write recursive programs that will work. Don’t worry about understanding *why or how* it works, just learn how to write these functions.”

2. “We are going to start out by writing some very simple functions. Functions that you could write much more easily without recursion. But try and forget about that, and just follow the rules to see how you could write that function recursively.”
3. “Once you get used to writing recursive functions using these rules, we’ll take a look at how this actually works on the computer.

3.2 Step 2: The Rules for Writing Recursive Functions (Using Star Wars)

If you choose to use the Star Wars analogy, now is the time to ensure that all of your students have seen at least one of the Star Wars movies, or are at least familiar with the concept of “The Force”. To date I have only had one student who had never seen any of the movies, so explanations are typically unnecessary.

Here is a summary of the key points: “In Star Wars, the heroes often refer to a magical power called ‘The Force.’ The Force can help you achieve monumental tasks, but only if you believe in it, are at one with it, and allow it to help you. If you resist, it will not help you. You must simply believe, and go with the flow.”

Now proceed with writing the rules on the board (and even better, supplementing with a handout) as follows:

1. Specify the prototype for your function.
2. Write a careful comment, including pre-conditions, post-conditions, and any value returned by the function.
3. Code the base case (i.e. the simplest case) in (one or more) if statements.
4. Think of a concrete example as well as a second example that is a bit closer to the base case than your concrete example.
5. USE THE FORCE: believe that your comment is true for the second example and specify what the function will do for the second example.
6. Given your answer to step 5, how would you solve the problem for step 4
7. Code the recursive step in an else statement. USE THE FORCE. Believe that your comment is true for any case closer to the base case. Write the general way to solve the problem.

For the following examples, you provide the class with an oral description of the problem, and have them follow the steps to the solution.

3.2.1 Example 1: count_down

“In our first example, we want to write a function that takes a positive integer, and prints out all of the numbers between that integer and 1, separated by spaces. For example, if num is 6 our function will print: 6 5 4 3 2 1. Of course, it would be much easier to write this without recursion, how would we do that? Now let’s try and do it recursively.”

In the example below, the steps are specified as an aid to the reader. When presenting this to students, write the code as a single unit, and read the steps out loud as you go. Step 4 should be done off to the side so that it does not interfere with the function.

Specify the prototype:

```
void count_down (int num)
```

Write a careful comment:

```
/* count_down prints the integers between num and 1, separated
  by spaces.

        Preconditions: num > 0
   Postconditions: the integers between num and 1 will be
printed in order separated by spaces. E.g.
     count_down(6) will print 6 5 4 3 2 1
   Returns: void
*/
```

Code the base case:

```
{
    if (num == 1)
        cout << num;
```

Think of a concrete example: (8)
What example is closer to the base case? (7)

Use the Force: Believe the comment is true. If I call count_down(7) what happens?
(It prints 7 6 5 4 3 2 1)

Use the Force: how can I solve count_down on 8 using count_down on 7?
(Print the 8, then print a space, then run count_down(7)).

Code the recursive step (keeping in mind our concrete example from steps 4, 5, and 6).

```
 else
 {
    cout << num << " ";
    count_down (num – 1);
 }
}
```

The most common problem that students have with this is remembering the space in steps 4 and 6. When working on 4 on the board, be sure to write the 8 and 7 without a space if the students do not explicitly tell you to write it, i.e. 87 6 5 4 3 2 1.

3.2.2 Example 2: count_down_up

"Now let's write a function that takes a positive integer, and prints out all of the numbers between that integer and 1 and then back up to num, separated by spaces. For example, if num is 6 our function will print: 6 5 4 3 2 1 2 3 4 5 6. Again, it would be much easier to write this without recursion, how would we do that? Now let's try and do it recursively."

Specify the prototype:

void count_down_up (int num)

Write a careful comment:

```
/* count_down_up prints the integers between num and 1, and back up to num
separated by spaces.

        Preconditions: num > 0
 Postconditions: the integers between num and 1 and then
   back up to num will be printed in order separated by
   spaces. E.g. count_down_up(6) will print
   6 5 4 3 2 1 2 3 4 5 6
 Returns: void
*/
```

Code the base case:

```
{
      if (num == 1)
            cout << num;
```

Think of a concrete example: (4)
What example is closer to the base case? (3)

Use the Force: Believe the comment is true. If I call count_down_up(3) what happens? (It prints 3 2 1 2 3)

Use the Force: how can I solve count_down_up on 4 using count_down_up on 3? (Print the 4, then print a space, then run count_down_up(3), then print a space, then print the 4).

Code the recursive step (keeping in mind our concrete example from steps 4, 5, and 6).

```
 else
 {
      cout << num << " ";
      count_down_up (num – 1);
      cout << " " << num;
 }
}
```

count_down_up is a particularly interesting example because the code is so similar to that of **count_down**. It's worth pointing out this fact, and explaining that sometimes the slightest mistake in writing a recursive function can produce unexpected results. It is worth mentioning that when starting to use recursion, it is often easier to start writing a program from scratch than to try and debug a bad function.

3.2.3 Example 3: count_up_to_100

At this point there is the concern that students may equate "closer to the base case" with "smaller." It's time to clear up that misconception.

"Now let's write a function that takes an integer that is less than or equal to 100, and prints out all of the numbers between that integer and 100, separated by spaces. For example, if num is 96 our function will print: 96 97 98 99 100. Again, it would be much easier to write this without recursion, how would we do that? Now let's try and do it recursively."

Specify the prototype:

```
void count_up_to_100 (int num)
```

Write a careful comment:

```
/* count_up_to_100 prints the integers between num and 100, separated by
spaces.

        Preconditions: num <= 100
    Postconditions: the integers between num and 100 will be printed in order
separated by spaces. E.g. count_up_to_100(96) will print
    96 97 98 99 100
 Returns: void
*/
```

Code the base case:

```
{
      if (num == 100)
            cout << num;
```

Think of a concrete example: (94)

*What example is **closer** to the base case?* (95)

Use the Force: Believe the comment is true. If I call count_up_to_100(95) what happens? (It prints 95 96 97 98 99 100)

Use the Force: how can I solve count_up_to_100 on 94 using count_up_to_100 on 95?
(Print the 94, then print a space, then run count_up_to_100(95)

Code the recursive step (keeping in mind our concrete example from steps 4, 5, and 6).

```
 else
 {
      cout << num << " ";
      count_up_to_100 (num + 1);
 }
}
```

3.2.4 Classic 1: Power

All of the functions we have written so far do not return a value. The power function is a useful one to demonstrate returning a value. Follow the same procedure to create a recursive function with the following prototype and description:

```
double power (double base, int exponent)
/* precondition: exponent > 0
   postcondition: none
   returns: base raised to the exponent power
*/
```

3.2.5 Classic 2: Fibonacci

This is useful to demonstrate that a recursive function may have more than one base case (of course, Fibonacci can be written with a single if statement, but force them to initially write it with two, and then simplify to one if you wish).

```
int fib (int num)
/* precondition: num > 0
   postcondition: none
   returns: the num'th fibonacci number, where the first
      and second fibonacci numbers are 1, and after that
      the n'th fibonacci number is the n-1'th fibonacci
      number plus the n-2'th fibonacci number
*/
```

3.3 Step 3: Demonstrating How Recursion Works

At this point it is time to demonstrate how recursive functions actually work, using the count_down_up example. Different text books recommend different approaches to this. My approach is to augment my code with line numbers (or even better, letters) and step through the levels of recursion, crossing out a letter when I have completed that step. For example, suppose the code is written as follows:

```
        void count_down_up (int num)
        {
a)          if (num == 1)
b)              cout << num;
c)          else
            {
d)              cout << num << " ";
e)              count_down_up (num – 1);
f)              cout << " " << num;
            }
        }
```

A snapshot of the blackboard of the run of count_down_up just after I have printed the number 1 would look something like Figure 1:

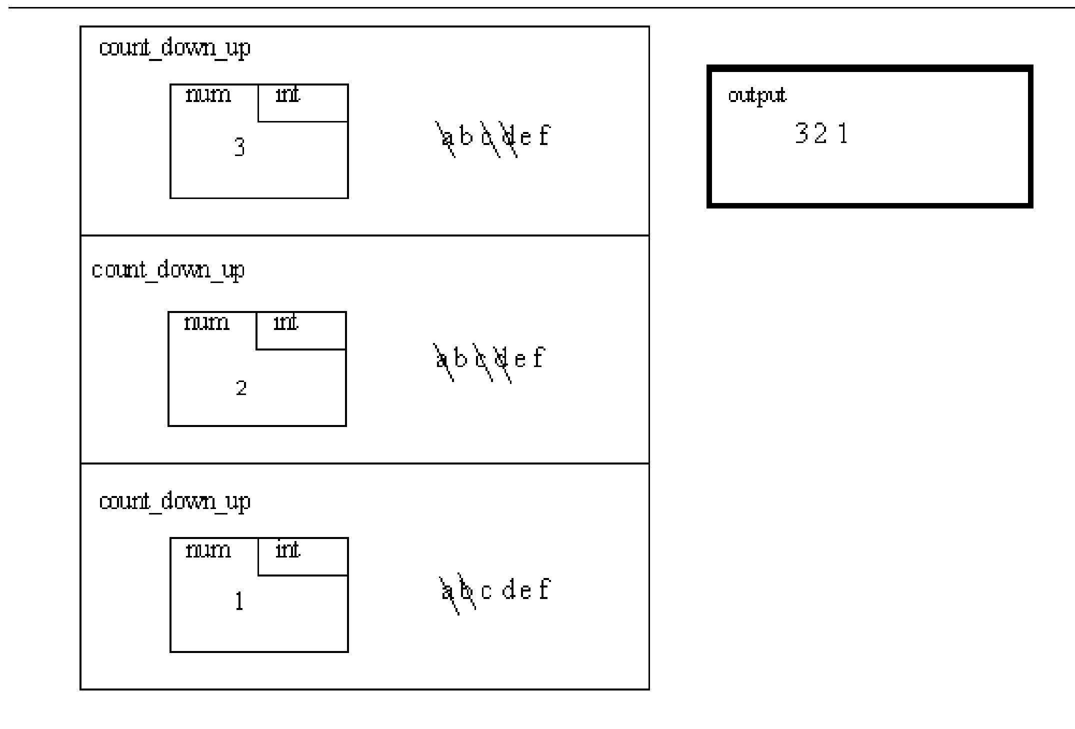

Figure 1. Blackboard Snapshot after printing the number 1

After printing a 2 for the second time, the blackboard would look like Figure 2:

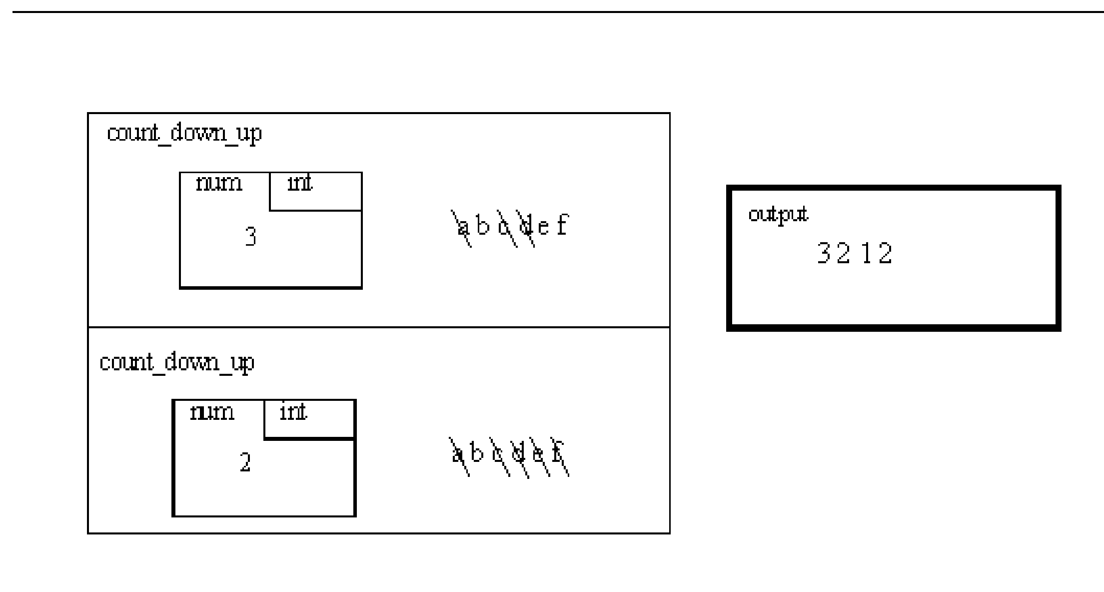

Figure 2. Blackboard Snapshot after printing the number 2

3.4 Step 4: Present One Final Classic

If you're feeling confident try Towers of Hanoi. This works best when you use the following prototype (from [4]) with a modification of their comment

```
void move (int count, int start, int finish, int temp)
/*
        precondition: There are at least count disks on the tower
          start. The top disk (if any) on each of towers temp and
          finish is larger than any of the top count disks on
          tower start

        postcondition: This prints the directions for playing the
                towers of Hanoi, as a series of lines:
                        move a disk from peg xxx to peg yyy
                for a game starting with count disks on start
                and ending with those disks on finish
*/
```

This problem is always very difficult. Encourage the students to follow the steps, and be sure that their postcondition specifies that a sequence of directions **will be printed**. If the students are stuck, suggest that they have lost their faith in The Force and must believe that they can do it.

4. CONCLUSIONS

Recursion can feel very unnatural to students as they first learn the subject. The above material typically takes at least a week of class time, but the resulting understanding is worth the effort. Use of this approach does not guarantee that students leave the class feeling that recursion is the most natural thing on earth. But it does provide them with a solid foundation and a set of techniques that they can use to approach recursive problems.

5. REFERENCES

Berman, A. Michael, Data Structures via C++: Objects by Evolution, Oxford University Press, 1996.

Budd, Timothy, Data Structures in C++ using the Standard Template Library, Addison Wesley, 1998.

Horstmann, Cay, Computing Concepts with C++ Essentials, Wiley, 1997.

Kruse, Robert L., and Ryba, Alexander J., Data Structures and Program Design in C++, Prentice Hall, 1999.

Main, Michael, and Savitch, Walter, Data Structures and Other Objects Using C++, Addison Wesley, 1998.

Nyhoff, Larry, C++: An Introduction to Data Structures, Prentice Hall, 1999.