Lecture Notes 4 - Recursion
CSS 501 – Data Structures and Object-Oriented Programming – Professor Clark F. Olson

Reading for this lecture: Carrano, Chapter 2

To be covered in this lecture:
- Classes with dynamic memory
- Writing recursive functions
- The Towers of Hanoi
- The 8 Queens Problem
- Drawing snowflakes

Copy constructor, destructor, operator=
The compiler will provide a default copy constructor, destructor, and operator= if you don’t define them. This is fine as long as there are no pointer variables in the private data that use dynamic memory allocation. The image library that I have made available uses dynamic memory allocation. So, when you are writing your new image class, you will need to implement these operations. The copy constructor and destructor should be straightforward. (You can use the functions in ImageLib!) The only tricky part is in operator=, it is important that you check for self-assignment. Consider this example for an arbitrary class:

```cpp
const someClass & operator=(const someClass &input) {
    if (this != &input) { // necessary since deallocation will
        // destroy input if they are the same!
        // deallocate memory for “this” here
        // then copy input into “this”
    }
    return *this;
}
```

Note that the copy constructor must also take a const & parameter, since otherwise the copy constructor would be needed to pass the parameter by value! (The copy constructor is used for three cases, call by value, return by value, and declaration with initialization, such as Object a = b.)

Usually, these three methods should be written first (if you are using dynamic memory), since they are called automatically in some cases. Otherwise, this may cause your code to crash.

Recursion
You should have been introduced to recursion in your previous courses. Here is a quick recap:
Recursion is a fundamental technique for writing programs. The basic idea is break the problem into easier problems of the same general type. Note that there are two distinct parts to recursive functions:
1. The base case. This part does not call itself. It handles a simple case that we know how to do without breaking the problem down into a simpler problem. In this case, it handles the case where there are no numbers to add.
2. The recursive case. This part breaks the problem into a simpler version of the original problem. This part makes (at least) one recursive call to itself. In this case, it adds all of the numbers, except the last one.

The factorial function
The factorial function is a common function that can illustrate recursion. The factorial function is:

\[ F(n) = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1 \quad \text{(for any integer n > 0)} \]

\[ F(1) = 1 \]

We can write a simple (non-recursive) function to compute factorial:
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result = result * i;
    return result;
}

Note that we can also write: \( F(n) = n \times F(n-1) \) (for \( n > 1 \)). Since this defines the function in terms of a simpler version of the problem, we can use recursion to compute factorial:

int factorial(int n) {
    if (n <= 1) return 1;
    else return n * factorial(n - 1);
}

Let’s compare the two versions:

- The iterative version has two local variables; the recursive version has none.
- The iterative version requires more lines of code than the iterative version.
- The iterative version is more difficult to understand (arguably).

The recursive version is simpler, because the computer is doing more of the work. The recursive version may require more memory (if the compiler is not smart enough to optimize the code) and it may take slightly longer (owing to function calling overhead), but these are not overriding factors. If you write your code using an efficient algorithm and, after it is working properly, find that the speed is not acceptable, then you can worry about optimization.

Ensuring that recursion works

Recursion is similar to induction in that you assume that simpler problem (the recursive call) is solved correctly and then you use this assumption to solve the complete problem. One of the most difficult aspects of recursion is accepting that the recursive call will do the right thing. Here is a checklist of conditions that must be true for the recursion to work. If they are all true, then you should feel confident that your function will work correctly.

1. A recursive function must have at least one base case and at least one recursive call (it’s OK to have more than one).
2. The test for the base case must execute before the recursive function call.
3. The problem must be broken down so that the recursive function call is closer to the base case than the original function call. (In theory, it is possible to get closer to the base case at each step, but never to reach it. This is quite rare in practice.)
4. The recursive call must not skip over the base case.
5. The non-recursive portions of the function must operate correctly (assuming that the recursive call operates correctly).

Let’s go back to our recursive factorial function and see if it meets these criteria.

1. There is a base case (\( n <= 1 \)) and a recursive case (\( n > 1 \)).
2. If we reach the recursive call, then we must have evaluated whether (\( n <= 1 \)), so we always test for the base case.
3. The recursive call is base case. If \( n > 1 \), then \( n - 1 \) is closer to the base case (\( n <= 1 \)). Otherwise, we are in the base case. Note that if our base case was (\( n == 1 \)), the recursive call would not necessarily get closer to the base case. The function will not work properly for negative numbers. Usually, you should use a base case that catches problems such as this.
4. Since \( n \) is an integer, and the recursive call reduces \( n \) by one, it is not possible to skip over the recursive case.
5. Assuming the recursive call works correctly, does the rest of the function operate correctly? Compare the code with the definition of factorial that says \( n! = n \times (n-1)! \) for \( n > 0 \) and \( n! = 1 \) for \( n = 1 \). When \( n \) is 1,
the code correctly returns 1. When n is greater than zero, the code returns n * factorial(n – 1), which is just what the definition says. The non-recursive portions of the function thus behave correctly.

**The towers of Hanoi**
There is a classic problem called “the towers of Hanoi” that can be solved very simply using recursion, even though it seems difficult. The problem can be stated as follows:
- There are three tall, thin towers: A, B, and C.
- There are n large disks with holes in the center, so that they can fit on towers.
- Each disk is a different size and can only be placed on top of a larger disk.
- All of the disks start on tower A.
- Only one disk can be moved at a time.
- The goal is to put all of the disks on tower C using tower B as a spare.

This problem seems difficult at first glance (and it **is difficult to solve without recursion**), but it can be solved very elegantly using recursion. First, let’s look at how the problem can be broken down into simpler problems of a similar nature. A simpler problem would be to move fewer disks than in the original problem. What if we moved half of the disks? What would be our next step? We have nowhere to go from there, because the new problem can’t be solved like the old problem (there is no spare tower to work with). How about moving all but one of the disks? At that point, we could **move the largest disk**. We can also move the other disks after moving the largest disk, since the pole with the largest disk can be used as a spare tower. This leads to the following procedure: move all but the largest disk onto the spare tower (using a recursive call), move the largest disk onto goal tower, and then move all of the other disks onto the goal tower (using another recursive call). Believe it or not, with a simple base case, this solves the entire problem!

To solve the problem, we will assume that we already have classes corresponding to towers and disks with the following interface:

- **disk**:
  - No methods needed.

- **tower**:
  - disk removeTopDisk()
  - bool addDiskToTop(disk diskAdded) // Returns false if new disk is too big to be added

  // Recursive algorithm that solves the towers of Hanoi problem.
  // Pre-conditions: Source has at least n disks on it.
  // // Destination and spare are either empty or they hold only
  // // disks that are larger than the top n disks on source.
  // // Post-conditions: The top n disks on source have been moved to destination.
  // // Disks that start on destination or spare are not moved.
  void hanoi(int n, tower &source, tower &destination, tower &spare)
  {
    if (n > 0)
    {
      hanoi(n – 1, source, spare, destination);
      disk diskToMove = source.removeTopDisk();
      destination.addDiskToTop(diskToMove);
      hanoi(n – 1, spare, destination, source);
    }
  }

Believe it or not, that solves the entire problem!

A java applet that demonstrates the solution can be found at:
http://www.cut-the-knot.org/recurrence/hanoi.shtml
Trace the algorithm for n = 4:

\[
\begin{align*}
&\text{hanoi}(4, A, B, C); \\
&\text{hanoi}(3, A, C, B); \\
&\quad \text{hanoi}(2, A, B, C) \\
&\quad \quad \text{hanoi}(1, A, C, B) \\
&\quad \quad \quad \text{Move disk from a to c} \\
&\quad \quad \quad \text{Move disk from a to b} \\
&\quad \quad \quad \text{hanoi}(1, C, B, A) \\
&\quad \quad \quad \quad \text{Move disk from c to b} \\
&\quad \quad \text{Move disk from a to c} \\
&\quad \text{hanoi}(2, B, C, A) \\
&\quad \quad \text{hanoi}(1, B, A, C) \\
&\quad \quad \quad \text{Move disk from b to a} \\
&\quad \quad \quad \text{Move disk from b to c} \\
&\quad \quad \quad \text{hanoi}(1, A, C, B) \\
&\quad \quad \quad \quad \text{Move disk from c to b} \\
&\quad \text{Move disk from a to b} \\
&\text{hanoi}(3, C, B, A); \\
&\text{hanoi}(2, C, A, B) \\
&\quad \text{hanoi}(1, C, B, A) \\
&\quad \quad \text{Move disk from c to b} \\
&\quad \quad \text{Move disk from c to a} \\
&\quad \quad \text{hanoi}(1, B, A, C) \\
&\quad \quad \quad \text{Move disk from b to a} \\
&\quad \text{Move disk from c to b} \\
&\text{hanoi}(2, A, B, C) \\
&\quad \text{hanoi}(1, A, C, B) \\
&\quad \quad \text{Move disk from a to c} \\
&\quad \quad \text{Move disk from a to b} \\
&\quad \quad \text{hanoi}(1, C, B, A) \\
&\quad \quad \quad \text{Move disk from c to b}
\end{align*}
\]

How do we know that the pre-conditions are never violated? We need induction to prove this, so you’ll have to trust me for now.

**The cost of the towers of Hanoi**

An interesting question is: given n, how many moves does it take to solve the towers of Hanoi problem using the above algorithm. The number of moves is given by the following recurrence relation:

\[
\begin{align*}
moves(n) &= moves(n - 1) + 1 + moves(n - 1) = 2 \times moves(n - 1) + 1 \\
moves(1) &= 1
\end{align*}
\]

For example: \(moves(3) = 2 \times moves(2) + 1 = 2 \times (2 \times moves(1) + 1) + 1 = 2 \times (2 \times 1 + 1) + 1 = 7\)

It can be observed (and proven using induction) that \(moves(n) = 2^n - 1\).
The Eight Queens problem
A fun problem that can be solved using recursion is the Eight Queens Problem. The goal is to place eight queens on a chessboard so that none is in the same row, column, or diagonal of the board. We will assume that we have an ADT with the following operations for our board.

```cpp
class board {
    public:

        // Places a queen on the appropriate square
        // Pre-conditions: 1 <= row, column <= 8
        void setQueen(int row, int column);

        // Sets the appropriate square to empty (whether it has a queen or not)
        // Pre-conditions: 1 <= row, column <= 8
        void removeQueen(int row, int column);

        // Determines whether a queen can be placed on a square
        // Pre-conditions: 1 <= row, column <= 8
        bool isLegalPosition(int row, int column) const;
    }

    // Places queens in column col and subsequent columns. Returns true if successful, false otherwise
    // Pre-condition: queens have been placed legally in all columns from 1 to col-1 (inclusive)
    bool placeQueens(board &b, int col) {
        if (col > 8) return true;
        for (int row=1; row<=8; row++)
        {
            if (b.isLegalPosition(row, col))
            {
                b.setQueen(row, col);
                bool solved = placeQueens(b, col+1);
                if (!solved) b.removeQueen(row, col);
                else return true;
            }
        }
        return false;
    }
}
```

One solution:

This will apparently be the first solution found if it is rotated such that the bottom-left queen is in the upper-left corner.
Survivor (21 flags)
One season on an episode of Survivor, the immunity challenge involved an intelligence test where the teams played a game against each other called 21 flags. At the start of the game, there are 21 flags in a designated location. The two teams take turns removing 1, 2, or 3 flags from the game. The winning team is the team to remove the last flag. Interestingly, if the first team knows the optimal strategy, they can guarantee that they win the game. The solution can be seen using recursive (and logical) reasoning.

First, let’s think of the base case. At what position can you guarantee that you can win the game? If there are 1, 2, or 3 flags left, you win. Now, how can you put the other team in a position such that they must leave you with 1, 2, or 3 flags. The only case that guarantees this is if they have 4 flags during their turn. If you can leave them with exactly 4 flags, then you win. From here, we need to devise a way to leave the opponent with exactly four flags. The key is to see that this is exactly like trying to leave the opponent with 0 flags (winning the game). If they have exactly 8 flags on their turn, then you can leave them with exactly 4 flags on their next turn, and then win. This same situation applies at each step – you want to leave you opponent with a number of flags that is a multiple of 4 every time. If you go first, you remove 1 flag, leaving the opponent with 20. No matter how many flags your opponents takes, you leave them with 16, then 8, then 4, then 0, and you win no matter what they do.

Fractals
Fractals are (according to Wikipedia quoting Mandelbrot) “a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole.” Fractal patterns can be drawn relatively easily using recursion. An example is a ruler pattern (which is a simple fractal). The inches (on the bottom) are divided into halves, which have a smaller tick mark. The halves are recursively divided into quarters and then again into eighths, with successively smaller tick marks.

If we assume that we have a method to draw a segment somewhere, we can write a function to draw a snowflake fractal. Let’s assume that someone has given us a method:

```c
void drawSegment(x1, y1, x2, y2); // (x1, y1) and (x2, y2) are the coordinates of the endpoints.
```

Here is our snowflake method:

```c
void drawSnowflake(float centerx, float centery, float size) {
    if (size < MIN_SIZE) return;

    // draw “plus”
    drawSegment(centerx, centery – size/2, centerx, centery + size/2);
    drawSegment(centerx – size/2, centery, centerx + size/2, centery);

    // recursively draw smaller snowflakes
    drawSnowflake(centerx, centery – size/4, size/2);
    drawSnowflake(centerx, centery + size/4, size/2);
    drawSnowflake(centerx – size/4, centery, size/2);
    drawSnowflake(centerx + size/4, centery, size/2);
}
```

Notice that each snowflake is a copy of the largest snowflake, except that it is smaller, shifted, and has one less level of “flake.” You will be programming an image fractal in your next assignment. Here are some alternative patterns:
Practice problems (optional):
Carrano, Chapter 2:
4, 7, 9, 10, 11, 13, 18