Turtles All the Way Down

CS 211

Adam Sweeney

December 2, 2017

Wichita State University, EECS

Introduction

- This lecture presents recursion
- It has the potential to be a powerful tool, if used correctly

Agenda

- Chapter 14, Sections 14.1 & 14.2
- A case study
- A closer look at recursion
- Recursion vs. Iteration
- Recursion that returns a value

A Case Study

A Case Study

- Let us consider a program that takes an integer and prints it vertically to the screen
- For example, if given the value 2049, the output is:

```
2
0
4
9
```

- On the surface, it doesn't seem so bad
- It's only a little more complicated than it seems
- Before looking at a recursive solution, we'll look at a "traditional" solution first

Building a Solution

- Let us first consider the simplest case, a single digit number
- If the number is strictly less than 10, we can simply print it
- We can then divide the larger task into smaller tasks, keeping this simplest case in mind
- If we want to print the number '2049', one way of dividing the task is to say we can print the first 3 numbers vertically, then print the last number
    - Printing the last number is the application of our simplest case

Building a Solution (cont.)

- In order to solve this problem, we need a way of figuring out the size of the number
- We will use the fact that these numbers are base 10 (decimal) to our advantage
- A number that is greater than or equal to 10, when integer divided by 10, will result in a non-zero value
- So, this means that we can create a copy of our number, and count every time we divide by ten and get a non-zero result
- This will give us the size of our number

Getting the Size of the Number

- Here is a chunk of code getting the size (length) of our number
- The variable `n` is the original copy of our number

```
int numTens = 1;
int copy = n;
while (copy > 9) {
    copy /= 10;
    numTens *= 10;
}
// numTens is a power of 10 with the same number
// of digits as n
```

Putting it Together

- Now that we've determined the length of the number, we can go about printing it vertically

```
void writeVertical(int n)
{
    int numTens = 1;
    int copy = n;
    while (copy > 9) {
        copy /= 10;
        numTens *= 10;
    }

    for (int length = numTens; length > 0; length /= 10) {
        cout << (n / length) << endl;
        n %= length;
    }
}
```

This is Programming, After All

- Is there another way to implementing this algorithm?
- Almost always
- Here's a different implementation

```
void WriteVetical(int n)
{
    if (n < 10) {
        cout << n << endl;
    }
    else {
        writeVertical(n / 10);
        cout << (n % 10);
    }
}
```

- Off the bat, we observe that the function is at least much shorter

Is That Meta?

- The interesting line in this new version of the function is that it contains a call to itself
- This is recursion, where a function invokes itself
- Recursion is an interesting concept in C++ programming
  - It is never required
  - If you think a recursive solution applies, an iterative (using a loop) solution can also be used
- Even though it's never required, being able to wrap your head around recursion and understanding its principles will make solving certain problems easier
- Specifically, any problem where the sub-solution is the same as the general solution simply applied to a smaller data set

A Closer Look at Recursion

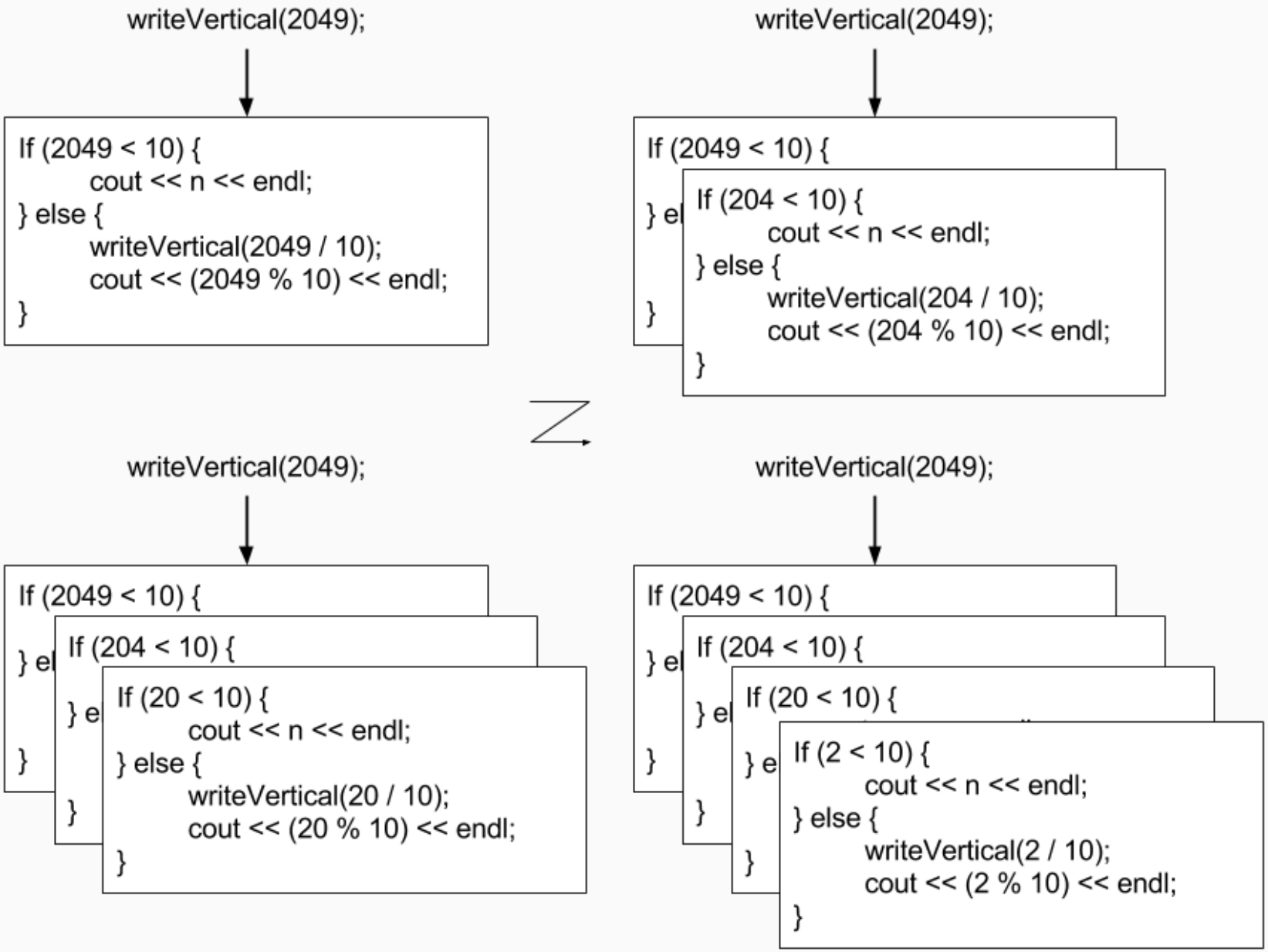
How Recursion Works

GIF LINK

How Recursion Really Works

- The easiest way to picture recursion involves a stack of cards
  - The recursive function is represented by a card
- When the original function call is made, a single card goes onto the stack
- As subsequent recursive calls are made, more cards are added to the stack
- Once a base case is encountered, the function would finish its execution, and that card would come off the stack
- The cards continue to fall off the stack until the original call finishes and the stack is left empty

Recursion, Visualized



About the Stack

- The stack is organized in a LIFO (Last In First Out) manner
- This means that we can only add to the top of the stack, and we can only remove from the top of the stack
- The stack is limited
- If a recursive function is infinite, or simply too long, there is a risk of causing a stack overflow
    - This means that the stack attempts to grow beyond its limit

Recursion vs. Iteration

Recursion vs. Iteration

- It has been stated that recursion is purely optional, and that is true
- If you see a problem where recursion could provide a solution, then there exists an iterative solution as well
- Generally, the iterative solution will also be more efficient!
- However, recursion allows the system to handle a lot of legwork for us, and can our code more compact / easier to read

So Why Are We Talking About Recursion Again?

- A recursive algorithm solves a specific type of problem
- One where the sub-task is just a smaller version of the main solution
- This is a type of problem that is fairly common
- Being able to derive these solutions requires exposure to this line of thinking
- At least knowing that recursion exists gives you another tool to work with

- We'll consider another example

Recursion That Returns a Value

Recursion That Returns a Value

- We will consider the factorial
- $5! = 120 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$

- The previous recursive function of printing vertical numbers did not have a return value
- We will now examine a recursive function that does need to return a value

Right Off the Bat, the Code

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

- A fairly straightforward function
- The only difference here is that we are dealing with returned values
- This means that in order for our total solution to be correct, every returned value must also be correct

Interaction of the Returned Values

- *NOTE*: If you decide to experiment with the factorial function, do not use a number greater than ~15
  - Factorials grow very rapidly, and we want to avoid attempting to calculate a number greater than the maximum for an integer
- The recursive call occurs in the return statement
- And a calculation is applied to the recursively returned value
- So, one mis-step, and the entire answer will be thrown off
- Luckily, we only need to check two cases

The First Case

- We need to check the base case, and any recursive call
- The base case:

```
if (n <= 1)
    return 1;
```

- What we need to ensure is if that is the correct value to return
  - It is
  - The factorials of both 0 and 1 is 1
  - While we are not able to take the factorial of a negative number, the base case catches those and returns a default value of 1

The Second Case

- Here, we will check the first recursive call

```
else
    return n * factorial(n - 1);
```

- Does this return the correct value?
  - If want to calculate the factorial of 5, the first recursive call will be `return 5 * factorial(5 - 1);`
    or $5 \cdot 4!$, which is correct

Not So Different

- If your recursive function returns a value or not, the checks that you make are not so different
- A proper action taken or a proper value returned still only needs to be checked twice, for the base case and for the recursive case