Lecture 4A: Recursion as a Problem Solving Technique - Part 1

ECE 2574

Data Structures & Algorithms

Spring 2015

Agenda

- Call stack
- General properties of recursive solutions
- Examples of recursive solutions
  - Factorial of N
  - Multiplying rabbits
  - Fibonacci sequence

Recursive Solutions

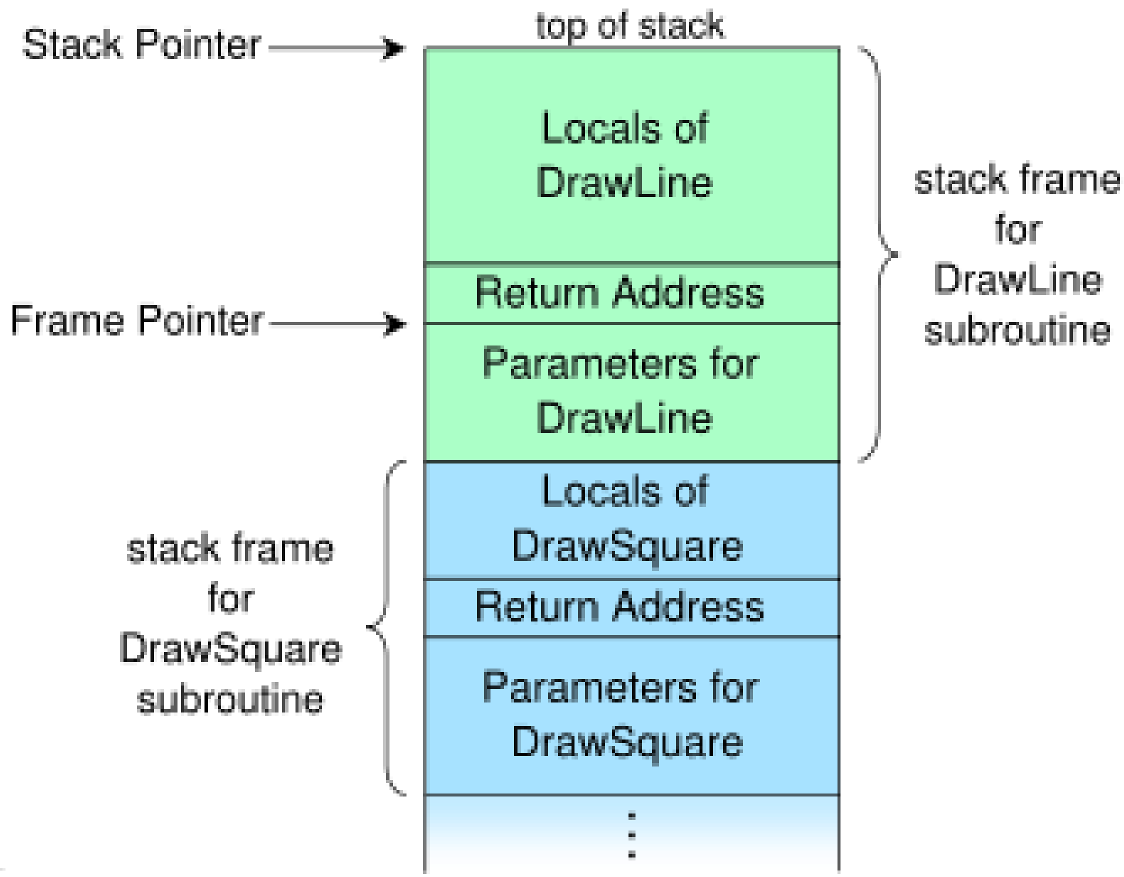
- Recursion is an extremely powerful problem-solving technique
  - Breaks a problem into smaller identical problems
    - Finally reaches the terminating cases
  - An alternative to iteration, which involves loops
    - The involving function calls itself recursively
  - Recursion is powerful idea-wise, but not programming-wise
    - The code is not very readable
    - Large call stack usage

What is a Call Stack?

- A call stack is a dynamic stack data structure that stores information about the active subroutines of a program
  - A.k.a. execution stack, control stack, function stack, or run-time stack
- Purposes of the call stack
  - Storing the return address
  - Local data storage
  - Parameter passing

Example Call Stack

- A subroutine named **`DrawLine`** is currently running, having just been called by a subroutine **`DrawSquare`** (the stack is growing towards the top)

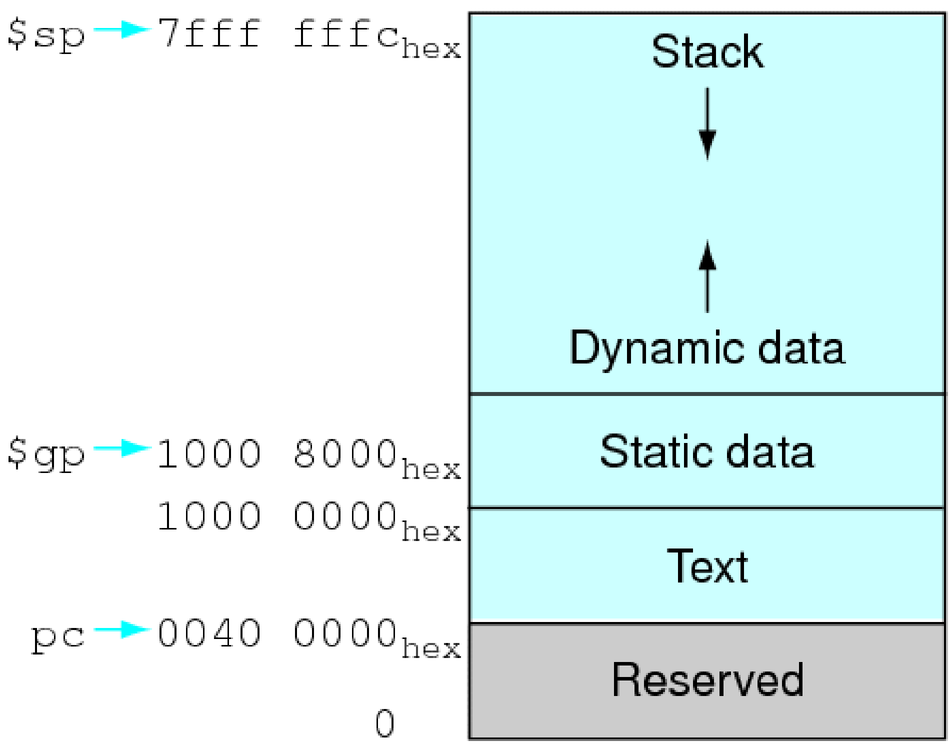


Elements of the Call Stack

- A call stack is composed of stack frames
  - It includes space for the local variables of the routine, return address back to the routine's caller, and the parameter values passed into the routine
- Stack is often accessed via a register called the stack pointer
  - Indicates the current top of the stack
- Memory within a frame may be accessed via a separate register, called the frame pointer
  - Points to some fixed point in the frame structure, such as the location for the return address

Looking at the Big Picture: Memory Allocation

- Text: Stores MIPS machine code
- Static data: Stores constants & other static variables
- Dynamic data (heap memory): Used for dynamic memory allocation



Memory allocation for ***MIPS*** (Microprocessor without Interlocked Pipeline Stages). It is a RISC microprocessor architecture developed by MIPS Technologies.

Display Linked Lists Recursively (review)

- Recursive strategy to display a list
  - Write the first node of the list
  - Write the list minus its first node

```
struct Node {
  char item;
  Node *next;
};
Node *stringPtr;

void writeString(Node *stringPtr)
{ if( stringPtr!=NULL )
  { cout << stringPtr->item;
    writeString( stringPtr->next );
  }
}
```

Recursive Solutions

- Example: look up a word in a dictionary
  - A sequential search is iterative
    - Starts at the beginning of the dictionary
    - Looks at every word one by one in alphabetical order
    - Works well if the word is in the beginning, but not well if the word is at the end
  - A binary search is recursive
    - Repeatedly halves the collection and determines which half could contain the word
    - Uses a divide and conquer strategy
- "Divide & Conquer" is a very common Computer Science algorithm
  - Sorting algorithm: QuickSort, MergeSort, etc.
  - Binary search
  - Many more...

Recursive Solutions

- Facts about a recursive solution
  - A recursive method calls itself
  - Each recursive call solves an identical, but smaller, problem
    - passing different function parameters
  - A test for the base case enables the recursive calls to stop
    - Base case: a known case in a recursive definition
  - Eventually, one of the smaller problems must be the base case
    - Multiple base cases are possible for a problem

Recursive Solutions

- Four questions for constructing recursive solutions
  - How can you define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case?
  - As the problem size diminishes, will you reach this base case?

Example: The Factorial of N

- By definition, for an integer $n > 0$,

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

  For $n = 0$, we define $0! = 1$

- For $n > 0$, $n! = n \times (n - 1) \times (n - 2) \times \ldots\ldots \times 2 \times 1$

  By substitution,

  $(n - 1)! = (n - 1) \times (n - 2) \times \ldots\ldots \times 2 \times 1$

  Therefore:

  $n! = n \times (n - 1)!$ (for $n > 0$)

  $n! = 1$ (for $n = 0$)

  $n! = \textit{undefined}$ (for $n < 0$)

A Recursive Valued Method

- A recursive definition of factorial(n)

$$
\begin{aligned}
\textit{factorial}\,(n) &= 1 && \text{if } n = 0 \\
&= n * \textit{factorial}\,(n\text{-}1) && \text{if } n > 0
\end{aligned}
$$

Write a C++ Function to Compute *n!*

- There are 2 basic approaches
  - The *iterative* method:

```
int factorial (int n)
{       . . .
        for (int i=n; i>0; i--)

        . . .
}
```

  - The *recursive* method:

```
int factorial (int n)
{       . . .
        return (n*factorial(n-1));
}
```

An Iterative Solution

```
int factorial (int n)
{
    int  i, fact;

    if (n < 0)
        PrintError("n must be nonnegative");

    fact = 1;
    for (i = 2; i <= n; i++)
       fact = fact * i;

    return (fact);
}
```

A Recursive Solution

```
int factorial (int n)
{
    if (n < 0)
        PrintError ("n must be nonnegative");
    if ( n == 0 )
        return (1);
    else
        return (n * factorial (n-1));
}
```

This is the “base case”

The Rest of the Program: **main()**

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int factorial (int);
void PrintError (char *);
void Usage (void);

int main(void)
{ int n;
  // get value of n from standard input
  cin >> n;
  // compute and print factorial(n)
  cout << n << "! = " << factorial(n) << endl;
  return (EXIT_SUCCESS);
}
```

Get *n* from the Command Line

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int factorial (int);
void PrintError (char *);
void Usage (void);

int main(int argc, char *argv[])
{  int n;
   // get value of n from command line
   if (argc == 2) n = atoi (argv[1]);
   else Usage();
   cout << n << "! = " << factorial(n) << endl;
   return (EXIT_SUCCESS);
}
```

The Rest of the Program (cont'd)

```
// Send an error message and exit.
void PrintError (char *str)
{
    cerr << "ERROR: " << str << endl;
    exit (EXIT_FAILURE);
}

// Explain how to run the program, and exit.
void Usage (void)
{
    cerr << "USAGE: factorial n" << endl;
    exit (EXIT_FAILURE);
}
```

Sample Output for **fact.cpp**

```
Input an integer: 6
Called factorial ( 6 )
Called factorial ( 5 )
Called factorial ( 4 )
Called factorial ( 3 )
Called factorial ( 2 )
Called factorial ( 1 )
Called factorial ( 0 )
Recursive version: 6! = 720
Iterative version: 6! = 720

Input an integer: -1
Called factorial ( -1 )
ERROR: n must be nonnegative
```

A Recursive Valued Method

- Box trace (see p. 54-57 of textbook)
  - A systematic way to trace the actions of a recursive method
  - Each box roughly corresponds to an activation record (or “stack frame”)
  - Contains a function’s local environment at the time of and as a result of the call to the method

```
n = 3
A: fact(n-1) = ?
return ?
```

A Recursive Valued Method (cont'd)

- A function's local environment includes:
  - The function's local variables
  - A copy of the actual value arguments
  - A return address in the calling routine
  - The value of the function itself

Potential Pitfalls

- With <u>iteration</u> we can have *infinite loops*:

```
for (i=1; i>0; i++)
{
        . . .
}
```

Condition always satisfied
(at least until **`i`** overflows)

- With <u>recursion</u> we can have *infinite regresses*
  - The function calls itself again, again, and again..., never encountering a base case

- Infinite regresses result in (call) stack overflow.

Example with an Infinite Regress

```
int factorial (int n)
{
    return (n * factorial (n-1));
}
```

Result is:

System.StackOverflowException

Why does this code result in an infinite regress?

Recursive Problem Example: Multiplying Rabbits

- "Assumptions" about rabbits
  - » Rabbits never die
  - » A rabbit reaches sexual maturity exactly two months after birth
    - At the beginning of its third month of life
  - » At the beginning of every month, each mature male-female pair gives birth to exactly one male-female pair
- If we start with a single new-born male-female pair, how many pairs would there be in month **n**, counting the births that took place at the beginning of month **n**?

Counting Rabbits

- The number of alive rabbit pairs depends on numbers in previous months
- Let’s count a simple case **n = 6**

Month 1: 1 pair, the original rabbits

Month 2: 1 pair still, since the rabbits are not yet sexually mature.

Month 3: 2 pairs; the original pair has reached sexual maturity and has given birth to a second pair.

Month 4: 3 pairs; the original pair gives birth again, but the new-born at month 3 are not mature yet

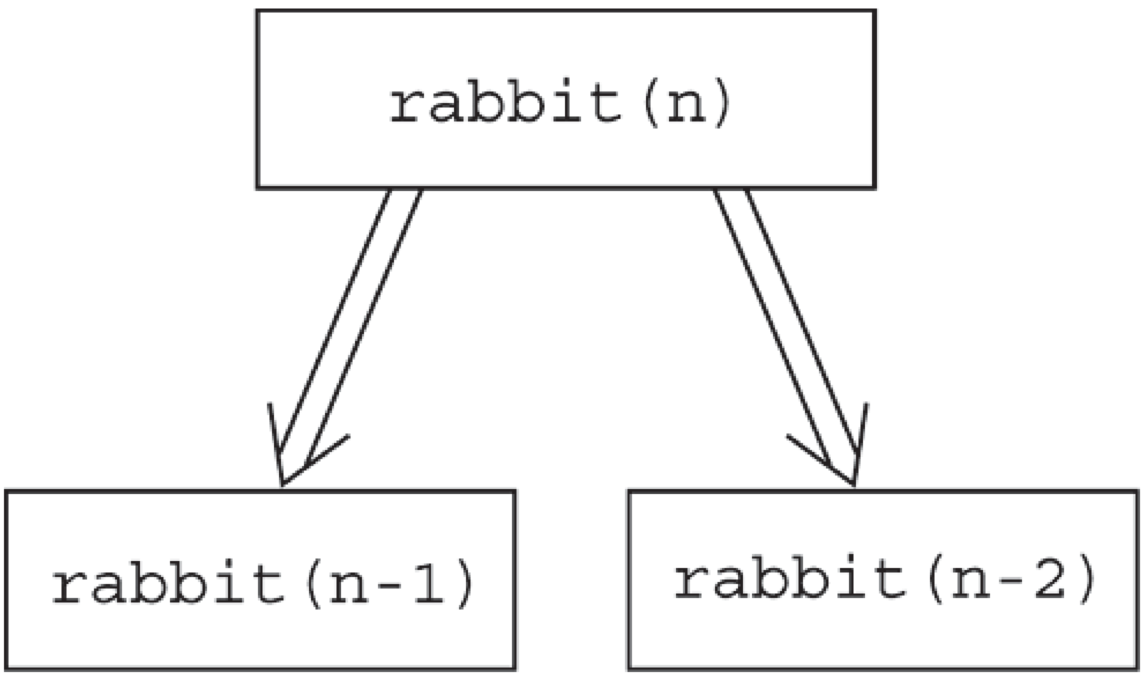
Month 5: 5 pairs; all rabbits alive in month 3 (2 pairs) are now sexually mature. Add their offspring to those pairs alive in month 4 (3 pairs)

Month 6: 8 pairs;

Framing the Rabbits Problem as a Recursive Problem

- Number of rabbit pairs is the number of pairs of last month plus the new-born pairs at the beginning of this month
- Recurrence relation

  rabbit(n) = rabbit(n-1) + rabbit(n-2)



Multiplying Rabbits (Cont'd)

- Recurrence relation

  `rabbit(n) = rabbit(n-1) + rabbit(n-2)`

- Base cases
  - **`rabbit(2), rabbit(1)`**
- Recursive definition

```
rabbit(n) = 1                          if n is 1 or 2
rabbit(n-1) + rabbit(n-2)              if n > 2
```

- Fibonacci sequence
  - The series of numbers **`rabbit(1), rabbit(2), rabbit(3),`** and so on
  - Models many naturally occurring phenomena

Iterative Version of Fibonacci: **`fib.cpp`**

```
// iterative solution...
int fibonacci2 (int n)
{  int i, lastmonth, last2month, thismonth = 1;

   if (n < 0)
      PrintError("n must be nonnegative");

   lastmonth = 1;  last2month = 1;
   for (i=2; i<n; i++)
    {  thismonth = lastmonth + last2month;
       last2month = lastmonth;
       lastmonth = thismonth;
    }

    return thismonth;
}
```

Summary

- Iteration and recursion are 2 different ways to do something over and again
- Recursion can be difficult to learn
- In writing recursive functions:
  - Try to write down the fundamental recurrence relation (express the problem in terms of itself, but "smaller")
  - Decide on the base case(s) (= stopping condition(s))
  - Write the function, checking for the base case (and possibly parameter validity) first

Summary (cont'd)

- Potential pitfalls of recursion
  - Infinite regresses
  - Numerical overflow
  - Stack overflow
    (More about this later. It means there were too many recursive function calls for the amount of memory in our machine.)
  - A recursive solution is often slower than a corresponding iterative approach, because all the recursive function calls take time

Summary (cont'd)

- Recursion solves a problem by solving a smaller problem of the same type
  - » Powerful idea-wise, but maybe troublesome programming-wise
- Four questions for designing a recursive algorithm:
  - » How can you define the problem in terms of a smaller problem of the same type?
  - » How does each recursive call diminish the size of the problem?
  - » What instance of the problem can serve as the base case?
  - » As the problem size diminishes, will you reach this base case?

Summary (cont'd)

- A recursive call's post-condition can be assumed to be true if its pre-condition is true
- The box trace can be used to trace the actions of a recursive method
- Recursion can be used to solve problems whose iterative solutions are difficult to conceptualize
- Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of method calls
- If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so

Reading Assignment

- Chapters 2 & 5 of the textbook (Carrano)