

# Recursion

Week 10

Gaddis:19.1-19.5

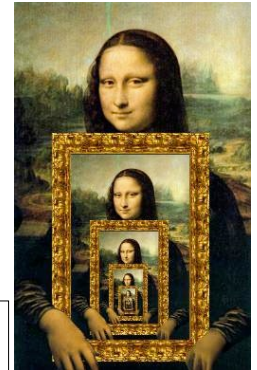
CS 5301  
Spring 2017

Jill Seaman

1

# What is recursion?

- Generally, when something contains a reference to itself
- Math: defining a function in terms of itself
- Computer science: when a function calls itself:



```
void message() {  
    cout << "This is a recursive function.\n";  
    message();  
}  
int main() {  
    message();  
}
```

What happens when this is executed?

2

# How can a function call itself?

- Infinite Recursion:

```
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
This is a recursive function.  
...
```

Note: If you encounter infinite recursion in Lab, be sure to STOP your program BEFORE running it again!!!

3

# Recursive message() modified

- How about this one?

```
void message(int n) {  
    if (n > 0) {  
        cout << "This is a recursive function.\n";  
        message(n-1);  
    }  
}  
int main() {  
    message(5);  
}
```

4

## Tracing the calls

- 6 nested calls to message:

```
message(5):
  outputs "This is a recursive function"
  calls message(4):
    outputs "This is a recursive function"
    calls message(3):
      outputs "This is a recursive function"
      calls message(2):
        outputs "This is a recursive function"
        calls message(1):
          outputs "This is a recursive function"
          calls message(0):
            does nothing, just returns
```

- depth of recursion (#times it calls itself) = 5<sup>5</sup>

## How to write recursive functions

- Branching is required (If or switch)
- Find a base case
  - one (or more) values for which the result of the function is **known** (no repetition required to solve it)
  - no recursive call is allowed here
- Develop the recursive case
  - For a given argument (say n), assume the function works for a smaller value (n-1).
  - Use the result of calling the function on n-1 to form a solution for n

6

## Recursive function example factorial

- Mathematical definition of n! (factorial of n)

```
if n=0 then    n! = 1
if n>0 then    n! = 1 x 2 x 3 x ... x n
```

- What is the base case?
  - n=0 (the result is 1)
- Recursive case: If we assume (n-1)! can be computed, how can we get n! from that?
  - $n! = n * (n-1)!$

7

## Recursive function example factorial

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

int main() {
    int number;
    cout << "Enter a number ";
    cin >> number;
    cout << "The factorial of " << number << " is "
         << factorial(number) << endl;
}
```

8

## Tracing the calls

- Calls to factorial:

```

factorial(4):
  return 4 * factorial(3);   =4 * 6 = 24
  calls factorial(3):
    return 3 * factorial(2); =3 * 2 = 6
    calls factorial(2):
      return 2 * factorial(1); =2 * 1 = 2
      calls factorial(1):
        return 1 * factorial(0); =1 * 1 = 1
        calls factorial(0):
          return 1;

```

- Every call except the last makes a recursive call
- Each call makes the argument smaller

9

## Recursive functions: ints and lists

- Recursive functions over integers follow this pattern:

```

type f(int n) {
  if (n==0)
    //do the base case
  else
    // ... f(n-1) ...
}

```

- Recursive functions over lists (arrays, linked lists, strings) use the length of the list in place of n
  - base case: if (length==0) ... // empty list
  - recursive case: assume f works for list of length n-1, compute the answer for a list with one more element.

## Recursive function example sum of the list

- Recursive function to compute sum of a list of numbers
- What is the base case?
  - length=0 (empty list) => sum = 0
- If we assume we can sum the first n-1 items in the list, how can we get the sum of the whole list from that?
  - $\text{sum}(\text{list}) = \text{sum}(\text{list}[0].. \text{list}[n-2]) + \text{list}[n-1]$

↑  
Assume I am given the answer to this

11

## Recursive function example sum of a list (array)

```

int sum(int a[], int size) { //size is number of elems
  if (size==0)
    return 0;
  else
    return sum(a,size-1) + a[size-1];
}

```

↑
↑  
 call sum on first n-1 elements    The last element

For a list with size = 4:  $\text{sum}(a,4) =$

$$\begin{aligned}
 & \text{sum}(a,3) + a[3] = \\
 & (\text{sum}(a,2) + a[2]) + a[3] = \\
 & ((\text{sum}(a,1) + a[1]) + a[2]) + a[3] = \\
 & (((\text{sum}(a,0) + a[0]) + a[1]) + a[2]) + a[3] = \\
 & 0 + a[0] + a[1] + a[2] + a[3]
 \end{aligned}$$

12

## Recursive function example

count character occurrences in a string

- Write a recursive function to count the number of times a **specific** character appears in a string
- We will use the string member function `substr` to make a smaller string
  - `string str.substr (int pos, int length);`
  - Returns a newly constructed string object containing the portion of `str` that starts at character position `pos` and spans `len` characters (or until the end of the string, whichever comes first).

```
string x = "hello there";  
cout << x.substr(0,10) << endl;  
cout << x.substr(1,10) << endl;  
cout << x[4] << endl;
```

Output:

```
hello ther  
ello there  
o
```

13

## Recursive function example

count character occurrences in a string

- This example is different from how the book does it.
- I use `substr` to remove the first character to make the recursive call on a shorter string.

```
int numChars(char target, string str) {  
    if (str.empty()) {  
        return 0;  
    } else { //make recursive call, then modify the results:  
        int result = numChars(target, str.substr(1,str.size()-1));  
        if (str[0]==target)  
            return 1+result;  
        else  
            return result;  
    }  
}  
  
int main() {  
    string a = "hello";  
    cout << a << " " << numChars('l',a) << endl;  
}
```

14

## Recursive function example

greatest common divisor

- Greatest common divisor of two non-zero ints is the largest positive integer that divides the numbers evenly (without a remainder)
- This is a variant of Euclid's algorithm:
  - $\text{gcd}(x,y) = y$  if  $x/y$  has no remainder otherwise:  
 $\text{gcd}(x,y) = \text{gcd}(y, \text{remainder of } x/y)$
- It's a recursive definition, correctness is proven elsewhere.

15

## Recursive function example

greatest common divisor

- Code:

```
int gcd(int x, int y) {  
    if (x % y == 0) {  
        return y;  
    } else {  
        return gcd(y, x % y);  
    }  
}  
  
int main() {  
    cout << "GCD(9,1): " << gcd(9,1) << endl;  
    cout << "GCD(1,9): " << gcd(1,9) << endl;  
    cout << "GCD(9,2): " << gcd(9,2) << endl;  
    cout << "GCD(70,25): " << gcd(70,25) << endl;  
    cout << "GCD(25,70): " << gcd(25,70) << endl;  
}
```

16

## Recursive function example

### Fibonacci numbers

- Series of Fibonacci numbers:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- Starts with 0, 1. Then each number is the sum of the two previous numbers

$F_0 = 0$   
 $F_1 = 1$   
 $F_i = F_{i-1} + F_{i-2}$  (for  $i > 1$ )

- It's a recursive definition

```
int fib(int x) {
    if (x==0 || x==1)
        return x;
    else
        return fib(x-1) + fib(x-2);
}
```

17

## Recursive function example

### Fibonacci numbers

- Note: the recursive fibonacci functions works as written, but it is VERY inefficient.
- Counting the recursive calls to fib:

The first 40 fibonacci numbers:

```
fib (0)= 0 # of recursive calls to fib = 1
fib (1)= 1 # of recursive calls to fib = 1
fib (2)= 1 # of recursive calls to fib = 3
fib (3)= 2 # of recursive calls to fib = 5
fib (4)= 3 # of recursive calls to fib = 9
fib (5)= 5 # of recursive calls to fib = 15
fib (6)= 8 # of recursive calls to fib = 25
fib (7)= 13 # of recursive calls to fib = 41
fib (8)= 21 # of recursive calls to fib = 67
fib (9)= 34 # of recursive calls to fib = 109
...
fib (40)= 102,334,155 # of recursive calls to fib = 331,160,28118
```

## Recursive functions over linked lists

- Member functions of a linked list class can be defined recursively.
  - These functions take a pointer to a node in the list as a parameter
  - They compute the function for the list starting at the node p points to.
- The pattern:
  - base case: empty list, when p is NULL
  - recursive case: assume f works for list starting at p->next, what is the answer for the list starting at p? (it has one more element).

19

## Recursive function example

### count the number of nodes in a list

```
class NumberList
{
private:
    struct ListNode {
        double value;
        struct ListNode *next;
    };
    ListNode *head;
    int countNodes(ListNode *); //private version, recursive

public:
    NumberList();
    NumberList(const NumberList & src);
    ~NumberList();
    void appendNode(double);
    void insertNode(double);
    void deleteNode(double);
    void displayList();
    int countNodes(); //public version, calls private
};
```

20

## Recursive function example

count the number of nodes in a list

```
// the private version, has a pointer parameter
// How many nodes are in the list starting at the pointer?
int NumberList::countNodes(ListNode *p) {
    if (p == NULL)
        return 0;
    else
        return 1 + countNodes(p->next);
}

// the public version, no arguments (Nodes are private)
// calls the recursive function starting at head
int NumberList::countNodes() {
    return countNodes(head);
}
```

Note that this function is overloaded

21

## Recursive function example

display the node values in reverse order

```
// the private version, needs a pointer parameter
void NumberList::reverseDisplay(ListNode *p) {
    if (p == NULL) {
        //do nothing
    } else {
        //display the "rest" of the list in reverse order
        reverseDisplay(p->next);
        cout << p->value << " ";
    }
}

// the public version, no arguments
void NumberList::reverseDisplay() {
    reverseDisplay(head);
    cout << endl;
}
```

22

## Linked List example:

- Append x to the end of a singly linked list:
  - Pass the node pointer by reference
  - Recursive

```
void List::append (double x) { Public function
    append(x, head);
}

void List::append (double x, Node *&p) { Private recursive function
    if (p == NULL) {
        p = new Node;
        p->data = x;
        p->next = NULL;
    }
    else
        append (x, p->next);
}
```

23

## Why use recursion?

- It is true that recursion is never **required** to solve a problem
  - Any problem that can be solved with recursion can also be solved using iteration.
- Recursion requires extra overhead: function call + return mechanism uses extra resources

However:

- Some repetitive problems are more easily and naturally solved with recursion
  - the recursive solution is often shorter, more elegant, easier to read and debug.

24