# Lecture 3-2

## Separate Compilation

You can place all the classes and main into one file. So why separate them?

- Abstraction and Encapsulation – provide libraries for other people – More important if you are developing an ADT
- Easier to split the job in a large project
- Easier for version control
- Easier debugging
- Change only one file and just compile that one

## Building a program

- Technically, a C++ program is a set of code components, compiled through to objects, and then linked together with the implementing libraries and a main() method.
- Thus, we can compile all of our individual files separately and then link them together, with a driver, to make a program.

## Interface file:

Exposed behaviors, with their usage comments

- Usually has a name <something>.h
- Contains private members of the class as well, to keep the entire class declaration in one place.
- We have to do this because C++ won't allow the class declaration across two files.

## Implementation file

– Usually has the name <something>.cpp

# Lecture 3-3

## Factorial

• Find n!

• Lets do it for some small n – 0!=1! =1 , 2!= 2 , 3!= 6

– 4!= 4.(what we just had! i.e. 6)= 24

– 5!= 5.(4!)

– Generally n!=n.(n-1)!

• Recursive approach:

– We know about a simple case (1!=1)

– Write a function that returns n.(the result for n-1)

– In order to find the result for n-1 it should call itself.

• Any other approach? – Of course! Iterative approach!

## Code in c++ for recursive factorial

```
int factorial(int n) {
        if (n < 0){
                cout << "ERROR!!!! Negative input\n";
                exit(1);
        }
        if (n == 0){
                return 1;
        }
        return n * factorial(n-1);
}
```

### Recursion versus Iteration

- Anything you can do recursively can be done iteratively.
  – Some languages don't even allow recursion.
  – Remember that everything, ultimately, gets turned into machine code

- Recursive functions are almost always slower and less efficient
  – but much easier to understand.

Recursion and iteration both repeatedly executes the set of instructions.
Recursion is when a statement in a function calls itself repeatedly.
The iteration is when a loop repeatedly executes until the controlling condition becomes false.

# Lecture 4-1

## Checklist for Recursion

• Three properties to be checked for a recursive algorithm:

– There is no infinite recursion

– Each stopping case performs the correct action

– For all recursive cases: IF all recursive calls perform correctly,

THEN the entire case performs correctly.

• To use recursion, break the problem into subproblems.

– at least one sub-problem needs to be the same problem as the main problem, but with a smaller size (simpler).

## Checking n!

• Infinite recursion?

– The relationship n! = n * ( (n-1)! ) has n getting smaller and heading towards 0. Provided we have a base case for 0, no infinite recursion.

• Is the returned value for base correct?

– Base case: n = 0, we return 1. This is correct.

• Is the relationship for recursion correct?

– If (n-1)! returns the correct result, then n! = n * (n-1)! will return the correct result.

• Therefore, this is correct!

## Problem Solving with Recursion

• Practice makes perfect!

• Step 1. Consider various ways to simplify inputs

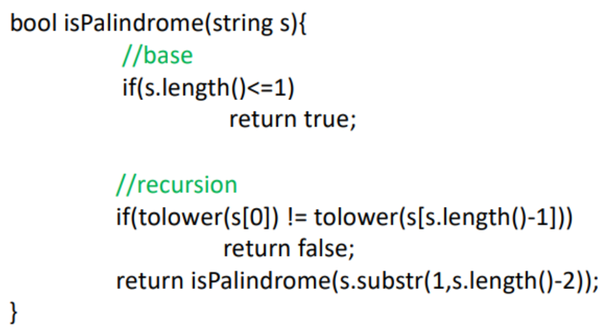
– Find sub-problems that perform the same task as the original problem, but with a simpler (or smaller) input.

• Step 2. Combine solutions with simpler inputs into a solution of the original problem.

• Step 3. Find solutions to the simplest cases.

• Step 4. Implement the solution by combining the simple cases and the reduction step.

## Example



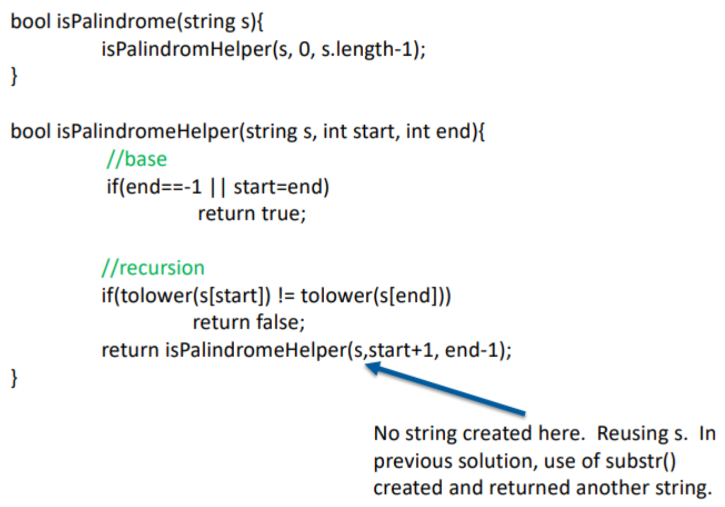
```
bool isPalindrome(string s){
        //base
        if(s.length()<=1)
                return true;

        //recursion
        if(tolower(s[0]) != tolower(s[s.length()-1]))
                return false;
        return isPalindrome(s.substr(1,s.length()-2));
}
```

This implementation of isPalindrome() is not efficient.

– It creates a new string for every recursive call

– What about checking whether a substring is a palindrome or not?

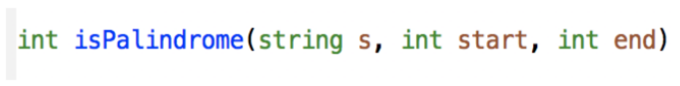
## Example



```
bool isPalindrome(string s){
        isPalindromHelper(s, 0, s.length-1);
}

bool isPalindromeHelper(string s, int start, int end){
        //base
        if(end==-1 || start=end)
                return true;

        //recursion
        if(tolower(s[start]) != tolower(s[end]))
                return false;
        return isPalindromeHelper(s,start+1, end-1);
}
```

No string created here. Reusing s. In previous solution, use of substr() created and returned another string.

- It is a common design technique in recursive programming to declare a second function that receives additional parameters.



```
int isPalindrome(string s, int start, int end)
```

### Heads and Tails

• Some compilers perform optimisation to reduce your call overhead.

• If possible, the compiler will remove stack heavy operations and replace them with lighter ones.

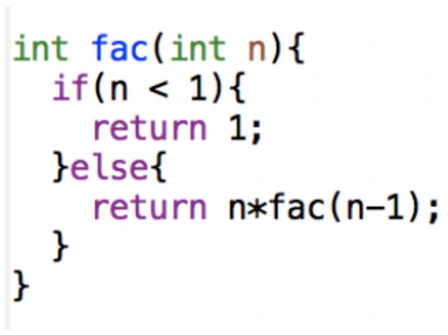
• It’s possible to do this, in recursion, using a technique called ‘tail recursion’.

• A recursive function is tail recursive when recursive call is the last thing executed by the function.

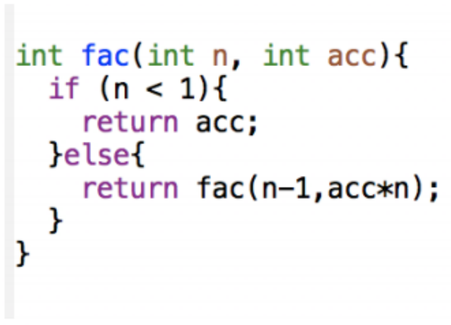
## Tail-recursive factorial

• How is stack used for this one?

• Here's a recursive factorial that a compiler can optimise to reduce stack overhead:



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



```
int fac(int n, int acc){
  if (n < 1){
    return acc;
  }else{
    return fac(n-1,acc*n);
  }
}
```

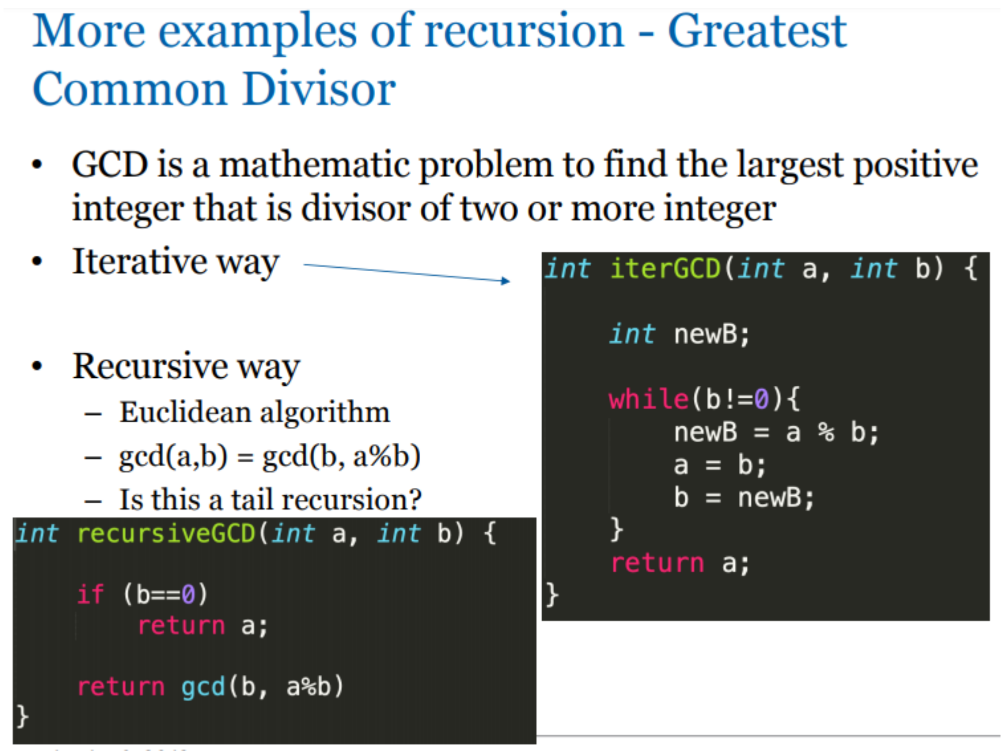
## Lecture 4-2

### Improving Efficiency with Memoisation - Trucks Revisited

Memoisation: store the obtained results of recursive function calls somewhere, so that you don't go through it again

For the call numTrucks(10, 2), how many times do we have to recursively call numTrucks(2, 2)?

Instead of making another call, we could store results and lookup in a table
– Assumes lookup takes less time than function call and calculation;



### More examples of recursion - Greatest Common Divisor

- GCD is a mathematic problem to find the largest positive integer that is divisor of two or more integer
- Iterative way

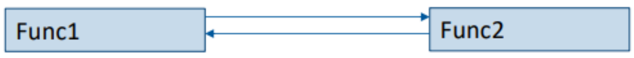
```
int iterGCD(int a, int b) {

    int newB;

    while(b!=0){
        newB = a % b;
        a = b;
        b = newB;
    }
    return a;
}
```

- Recursive way
  - Euclidean algorithm
  - gcd(a,b) = gcd(b, a%b)
  - Is this a tail recursion?

```
int recursiveGCD(int a, int b) {

    if (b==0)
        return a;

    return gcd(b, a%b)
}
```

### Indirect Recursion

- So far we've looked at direct recursion - a function calling itself.
- Indirect recursion occurs when a function calls itself through the intermediary of another function.
- For example a function Func1 calls a function Func2, which then calls Func1 again.



## Lecture 4-3

### Improving efficiency – Tabulation

- Similar to Memoization, but Tabulation calculates and stores *all* of the sub-results in advance, not just those produced in the recursion.
- What do we do in Tabulation?
    - Again, break the problem down to some smaller subproblems
    - solving each of them once
    - storing the solutions into some data structure (usually a table).
- Tabulation (bottom up) – useful if the sub values are all going to be used
- Memoization (top down) – useful if not all sub values are going to be used
- Both of these approaches are ways to implement ***Dynamic Programming*** (breaking the problem down into smaller problems and storing results)

Let's look back to the Fibonacci number

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

```
int fib(int n){
  int * fibTable= new int[n];

  fibTable[0] = fibTable[1] = 1;

  for(int i=2; i<n; i++){
    fibTable[i] = fibTable[i-1]+fibTable[i-2];
  }

  return fibTable[n-1];
}
```

### Dynamic Programming

- DP is just about filling table.
    - All in advance for tabulation
    - 'On the fly' (as generated) for memoisation
- DP is not always better!
    - Memory use for table
    - CPU time to create table and lookup values
    - If you don't reuse the saved values often, these costs will outweigh the benefit from saved recursive calls and perform worse!