An Introduction to Programming though C++

Abhiram G. Ranade

Lecture 5.2

Ch. 10: Recursive functions

Recursion

- Many physical and abstract objects have the following property:
  - The object has parts which are similar to the object itself.
  - Such an object is said to be recursive, or possess recursive structure.
- Computation may also possess recursive structure:
  - While computing the GCD of m, n, we find the GCD of n, m%n.
  - So it might seem that a function that finds GCD of m, n should call itself with arguments n, m%n.
  - This idea works beautifully, and such recursive functions are vey useful.
  - Recursive functions are also useful for processing recursive objects.
- We see all this next!

Euclid’s theorem on GCD

THEOREM: If m % n == 0, then GCD(m, n) = n, else GCD(m,n) = GCD(n, m % n).

The theorem looks like a program!

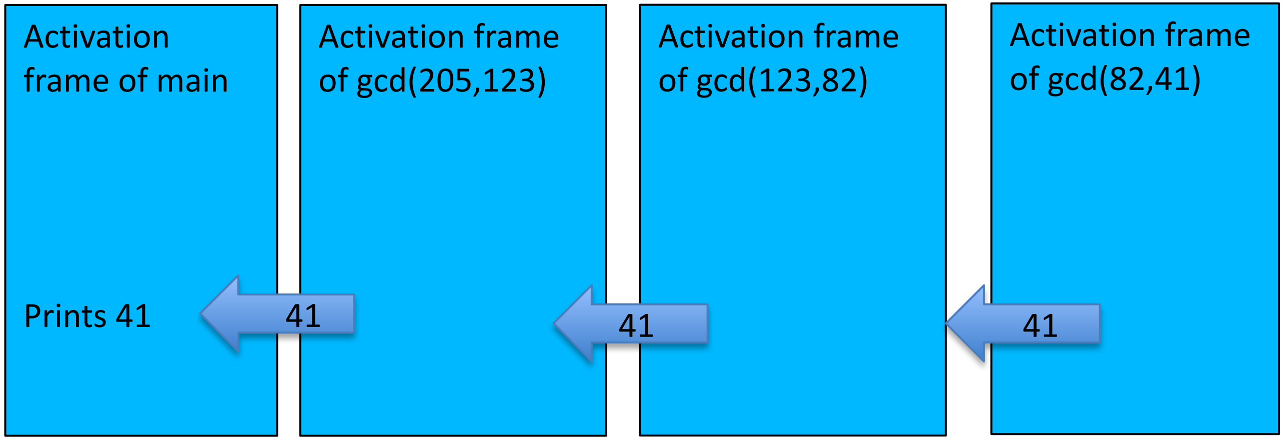
```
int gcd(int m, int n){
  if (m % n == 0) return n;
  else return gcd(n, m % n);
}
```

Will this work?

Execution

```
int gcd(int m, int n){
  if(m % n == 0) return n;
  else return gcd(n, m%n);
}
```

```
main_program{
  cout << gcd(205,123)
       << endl;
}
```



Demo

recursiveGcd.cpp

Recursion

- Recursion = The phenomenon of a function calling itself
  - Seems like we are defining the function in terms of itself
  - But no circularity if the arguments to the new call are different from the arguments in the original call.
- Each call executes in its own activation frame.
- Some call must return without another recursive call
  - Otherwise infinite recursion (error!)
- In the body of gcd there was just one recursive call. We can have several calls if we wish. Examples soon.

Comparison of recursive and non-recursive gcd

```
int gcd(int m, int n){
   if (m % n == 0) return n;
   else return gcd(n, m % n);
}
int gcd(int m, int n){
  while(m % n != 0){
       int r = m%n;
       m = n;
       n = r;
  }
  return n;
}
```

Recursive calls in gcd(205,123):

- gcd(123,82)
- gcd(82,41)

- Values of m,n in consecutive iterations of gcd(205,123):
- 205, 123
- 123, 82,
- 82, 41

- The two programs are ”really” doing the same calculations!
- But on the surface they look very different.

Remarks

- Recursion often produces compact, elegant programs.
  - Recursive programs might be slightly slower because they need to create activation frames etc.
- Recursion is also a way to discover algorithms.

  Euclid quite possibly thought to himself:
  - “Instead of doing laborious computation to find the gcd of 205 and 123, can I find two smaller numbers whose gcd is the same as that of 205 and 123?”
  - This is recursive thinking! It is common in mathematics.
  - We will see more examples soon.

Exercise

The factorial of n, written as n!, is defined as follows.

- 0! = 1
- For n > 0, n! = n * (n-1)!

Example: 4! = 4*3! = 4*3*2! = 4*3*2*1! = 4*3*2*1*0! = 4*3*2*1 =24

Write a recursive function that computes n! for any non-negative integer n.

How many activation fames would a call factorial(5) create? Draw them out. Show the value returned by each.

What we discussed

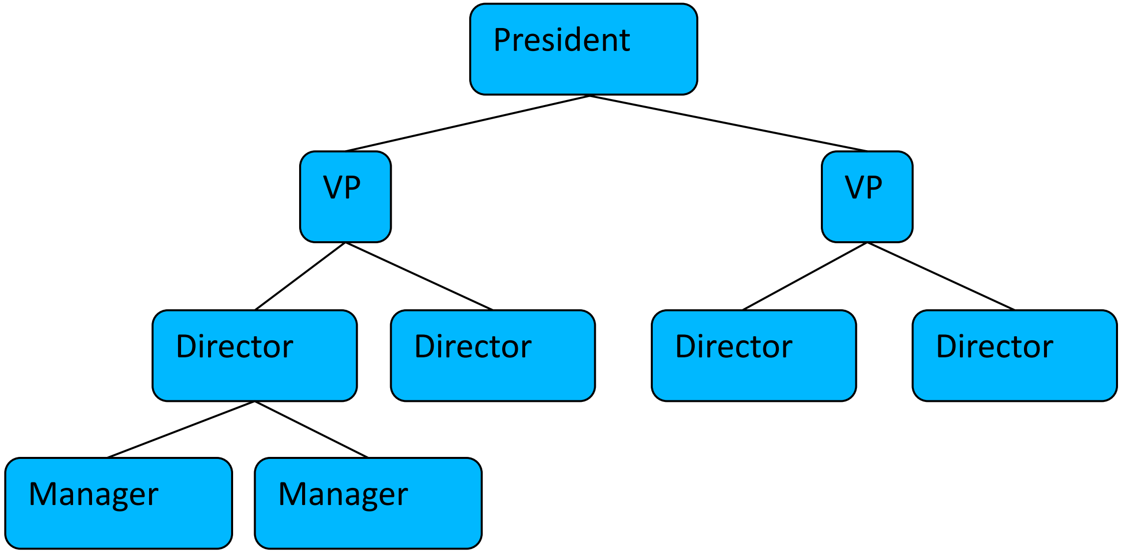
- Recursion = a function calling itself during execution.
- A recursive function to find the GCD.
- Comparison between recursive and non recursive gcd.
  - Recursive gcd is more compact and elegant, though both do the same computation

- Next: Recursive objects

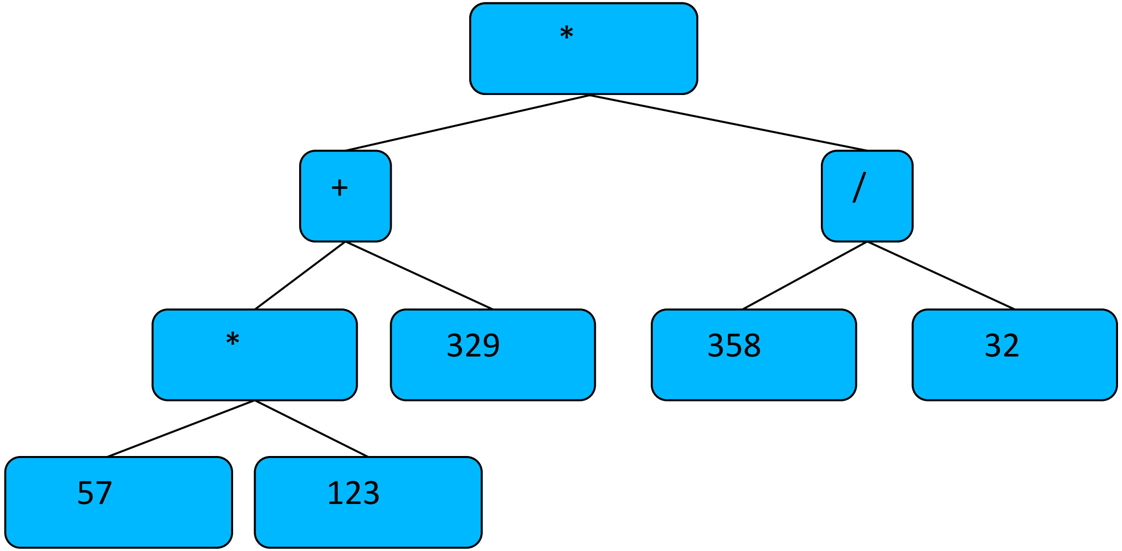
Outline

- Examples of recursive objects
- Example of processing recursive objects

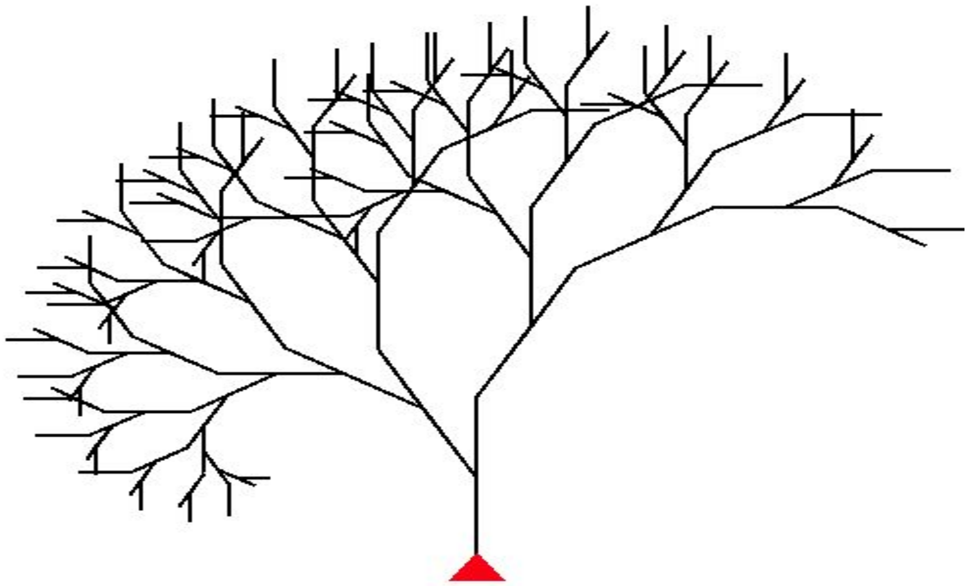
Organization tree
(typically “grows” downwards)



Tree representing ((57*123)+329)*(358/32))



An actual tree drawn using the turtle in simplecpp



Processing of recursive objects on a computer

Natural strategy:

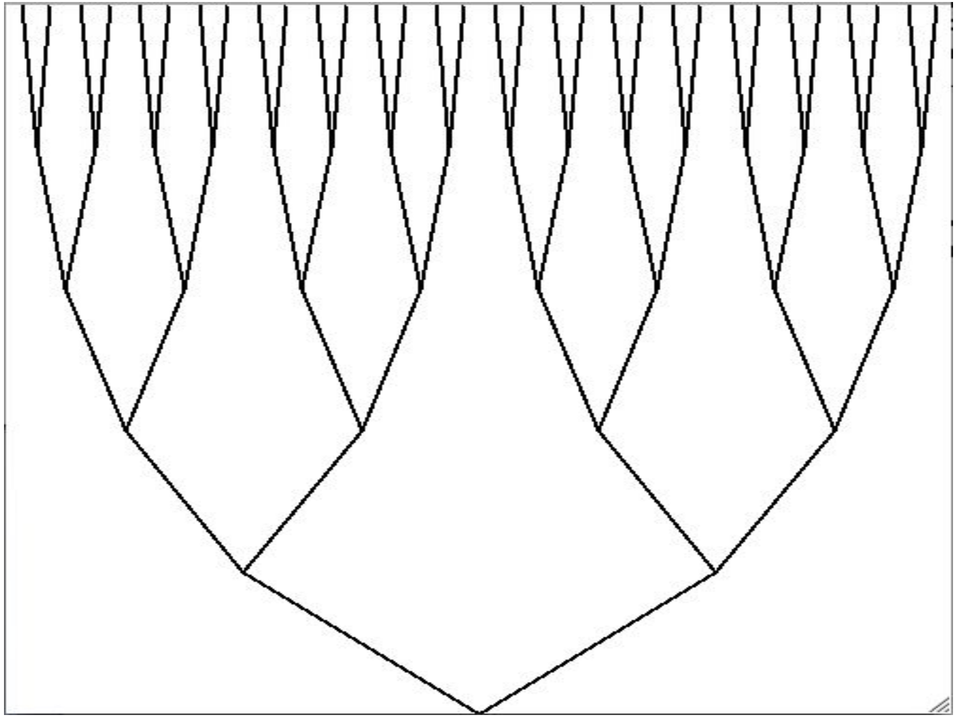
- Processing entire object = Processing all parts.
- Processing parts:
  - Use a recursive call if the part is similar to the entire object.

If you want to process organizations, or mathematical expressions, you must understand recursion!

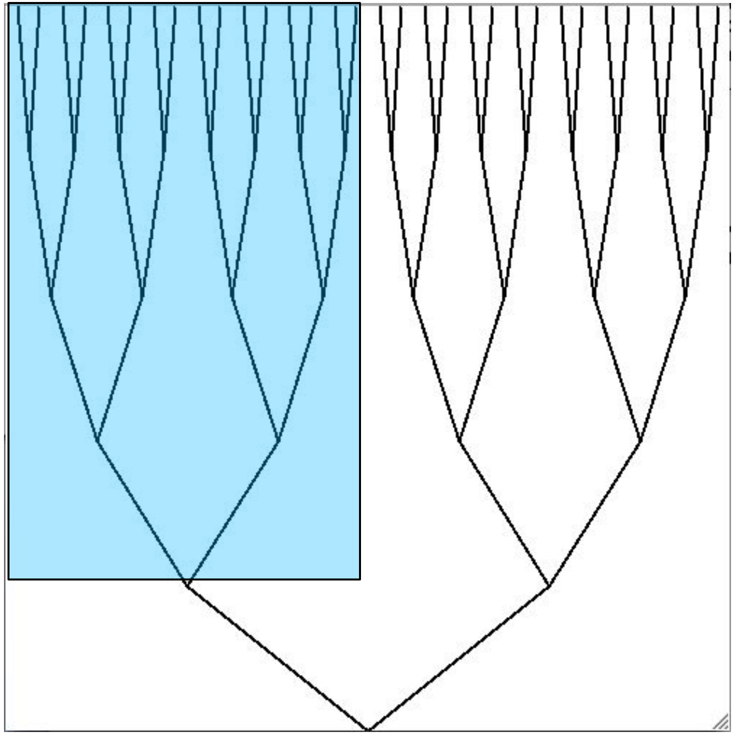
Next:

- Drawing a recursive object on the screen
- This will require us to employ the “natural strategy”.

What we will draw: A very stylized tree



Stylized tree = 2 small stylized trees + V

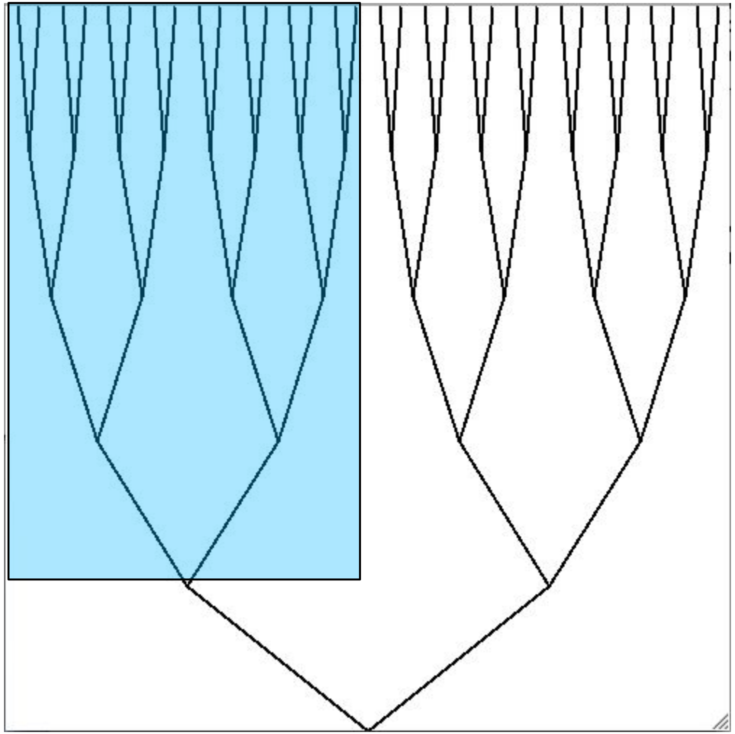


Parts:

- Root
- Left branch,   Left subtree
- Right branch, Right subtree

- Number of levels: number of times the tree has branched going from the root to any leaf.
- Number of levels in our tree = 5

Drawing the tree using coordinate based graphics



To draw an L level tree:

if L > 0{

Draw the left branch,

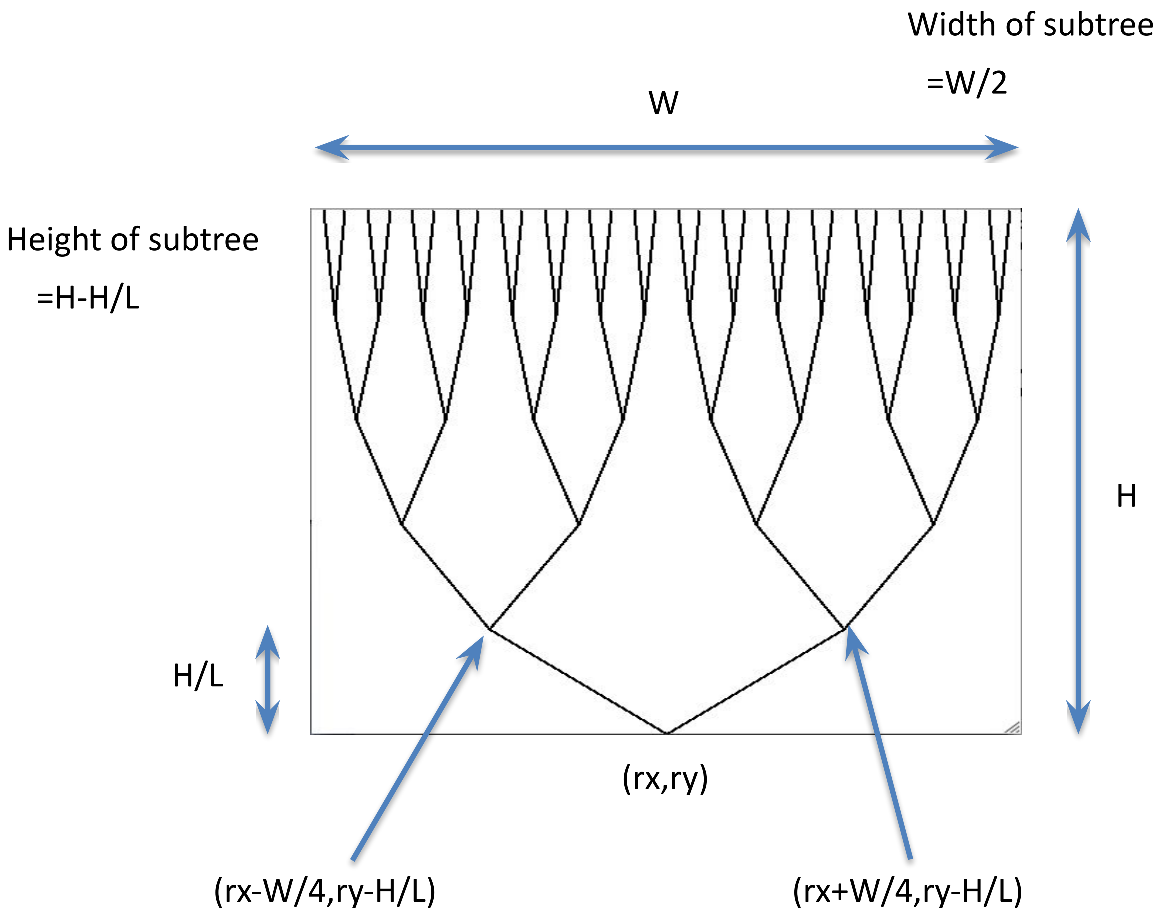
Draw a Level L-1 tree on top of it.

Draw the right branch,

Draw a Level L-1 tree on top of it.

}

- We need coordinates ...
  - Say root is to be drawn at (rx,ry)
  - Total height of drawing is H.
  - Total width of drawing is W.
- We should then figure out where the roots of the subtrees will be, and their width and height.



Width of subtree
=W/2
W
Height of subtree
=H-H/L
H
H/L
(rx,ry)
(rx-W/4,ry-H/L)
(rx+W/4,ry-H/L)

```
void tree(int L, double rx, double ry,
                double H, double W){
// L levels, Root at (rx,ry), Height H, Width W
  if(L>0){
    Line left(rx, ry, rx-W/4, ry-H/L);
    Line right(rx, ry, rx+W/4, ry-H/L);
    right.imprint();
    left.imprint();
    tree(L-1, rx-W/4, ry-H/L, H-H/L, W/2);
    tree(L-1, rx+W/4, ry-H/L, H-H/L, W/2);
  }
}
main_program{
  initCanvas(); tree(5, 250, 300, 300,500);
}
```

Demo

- Tree.cpp

Exercise

Draw the botanical tree using the turtle.

- Break it up into parts, i.e. trunk, left subtree, right subtree.
- Use the turtle to first draw the trunk.
- On top of it draw the left subtree.
  - After the drawing is finished, the turtle should come back to the original position and be facing in the same direction.
- The left and right subtrees are not exactly the same.
- You will need to play around a bit to get this. Start by trying to draw trees with few levels first.

What we discussed

- Recursive objects have parts similar to themselves.
- Many interesting objects have recursive structure.
- Processing recursive objects requires recursive functions.
- Drawing recursive objects is a good example of how you might "process" recursive objects.

Next: How to think about recursion. Conclusion