CSE / ENGR 142 Programming I

Recursion

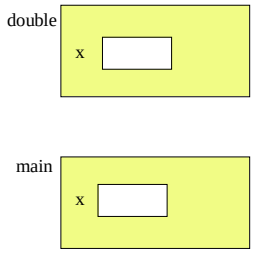
Chapter 10: Recursion

- 10.1 Nature of Recursion (skip "Mississippi" example 10.2)
- 10.2 Tracing: *Read!*
- 10.3 Recursive Math (but skip gcd example 10.6)
- 10.4 Skip
- 10.5 Skip
- 10.6 Towers of Hanoi: A Classic, but optional
- 10.7 Common Errors: *Read!*

Review: Function Calling

```
int double(int x) {
  return (2*x)
}

int main(void) {
  int x=3;
  x=double(5);
}
```



Each time a function is called, memory is allocated to store value of local variables (including parameters)

What is Recursion?

- Defn: A function is recursive if it calls itself

```
int foo(int x) {
  …
  y = foo(…);
  …
}
```

- Questions:
  - How can recursion possibly work?
  - Why would I want to write a recursive function?

Program *vs.* Process

- Program = a set of instructions
  - akin to a recipe
  - akin to the blueprint for an information factory
- Process = activity performed by computer when obeying the instructions
  - akin to the activity of cooking
  - akin to operation of a working factory
    - NO NEED FOR CEMENT OR STEEL
      - Instead, just need to allocate memory for local variables!!
    - CAN CREATE MULTIPLE FACTORIES AS NEEDED
      - *FROM THE SAME BLUEPRINTS!!!*

Factorial Function Revisited

```
0! is 1
1! is 1
2! is 1 * 2
3! is 1 * 2 * 3
…
```

```
int factorial ( int n ) {
  int product, i ;
  product = 1 ;
  for ( i = n ; i > 1 ; i = i - 1 ) {
    product = product * i ;
  }
  return (product) ;
}
```

function name

formal parameter

local variables

return type & value

Factorial via Recursion

```
/*  0! = 1! = 1;  for n > 1, n! = n(n-1)!  */

int factorial(int n)
{
   int t;

   if (n <= 1)
      t = 1;
   else
      t = n * factorial(n - 1);

   return t;
}
```

0! is 1
1! is 1
n! is n * (n-1)!, for n>1

E.g.: 3! =3 * 2!
= 3 * 2 * 1!
= 3 * 2 * 1

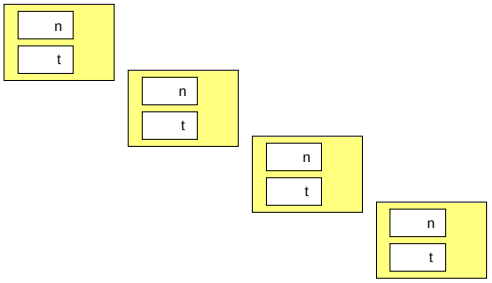
Review: Function Basics

- *Tracing recursive functions is no sweat if you remember the basics about functions:*
  - Formal parameters and variables declared in a function are local to it
    - Allocated (created) on function entry.
    - De-allocated (destroyed) on function return.
  - Formal parameters initialized by copying value of actual parameter.

Factorial

```
factorial(4) =
   4 * factorial(3) =
      4 * 3 * factorial(2) =
         4 * 3 * 2 * factorial(1) =
            4 * 3 * 2 * 1 = 24
```


Factorial Trace



Insist on 'y' or 'n'

```
char yes_or_no (void) {
   char answer = 'X';
   while (answer != 'y' && answer != 'n') {
      printf ("Please enter 'y' or 'n':");
      scanf (" %c", &answer);
      }
   return answer;
}
```


Insisting without Looping

```
char yes_or_no (void) {
   char answer;
   printf ("Please enter 'y' or 'n':");
   scanf (" %c", &answer);
   if  (answer != 'y' && answer != 'n')
       answer = yes_or_no ( );
   return answer;
}
```

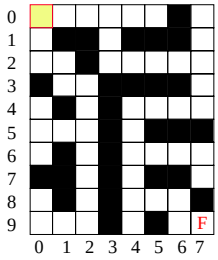
Iteration *vs.* Recursion

- Turns out ***any*** iterative algorithm can be reworked to use recursion instead (and vice versa).
- There are programming languages where recursion is the only choice!
- Some algorithms are more naturally written with recursion
  - But *naïve* applications of recursion can be inefficient

When to use Recursion?

- Problem has 1 or more simple cases
  - These have a straightforward nonrecursive soln
- Other cases can be redefined in terms of problems that are closer to simple cases
  - By applying this redefn process repeatedly one gets to one of the simple cases

Example: Path planning



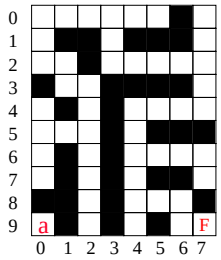

```
/* 'F' means finished!
   'X' means blocked
   ' ' means ok to move */
char maze[MAXX][MAXY];
int x =0, y=0;    /* start in yellow */
```

Unless blocked, can move up, down, left, right

Objective: determine if there is a path?

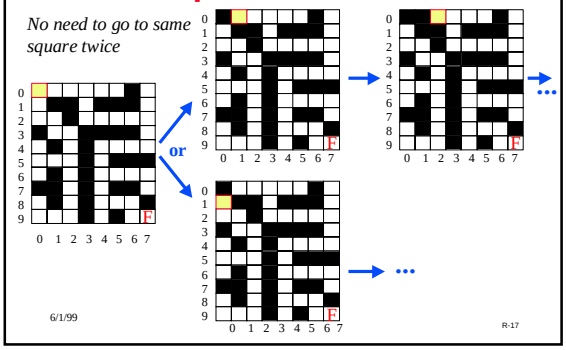
Simple Cases

- Suppose at x,y
- If maze[x][y]=='F'
  - Then "yes!"
- If no place to go
  - Then "no!"



Redefining a hard problem to several simpler ones

No need to go to same square twice



Helper function

```
/* Returns true if <x,y> is a legal move
   given the maze, otherwise returns false */
int legal_mv (char m[MAXX ][MAXY],
                                int x, int y) {
  return(x>=0 && x<=MAXX &&
         y>=0 && y<= MAXY &&
          m [x][y]!='X');
}
```

Elegant Solution

```
/* Returns true if there is a path from <x,y> to an element of maze
   containing 'F' otherwise returns false */
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
     return(TRUE);
  else {
     m[x][y] = 'X';
     return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
            (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
            (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
            (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
}}
```

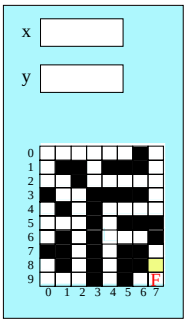

Example

is_path(maze, 7, 8)

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```

x

y



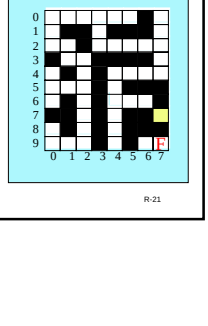
Example Cont

is_path(maze, 7, 7)

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```

x

y



Example Cont

is_path(maze, 7, 9)

```
int is_path(char m[MAXX][MAXY ], int x, int y) {
  if (m [x][y] == 'F')
    return(TRUE);
  else {
    m[x][y] = 'X';
    return((legal_mv(m,x+1,y) && is_path(m,x+1,y)) ||
           (legal_mv(m,x-1,y) && is_path(m,x-1,y)) ||
           (legal_mv(m,x,y-1) && is_path(m,x,y-1)) ||
           (legal_mv(m,x,y+1) && is_path(m,x,y+1)))
```

x

y