CS1010E Lecture 7
Modular Programming with Functions (Part 2)

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
www.comp.nus.edu.sg/~joxan
cs1101c@comp.nus.edu.sg

Semester II, 2016/2017

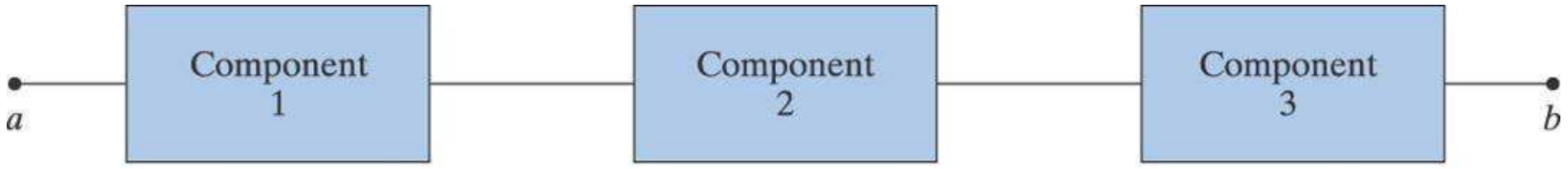
Lecture Outline

- A Fundamental Technique: Simulation
- Macros
- Recursion

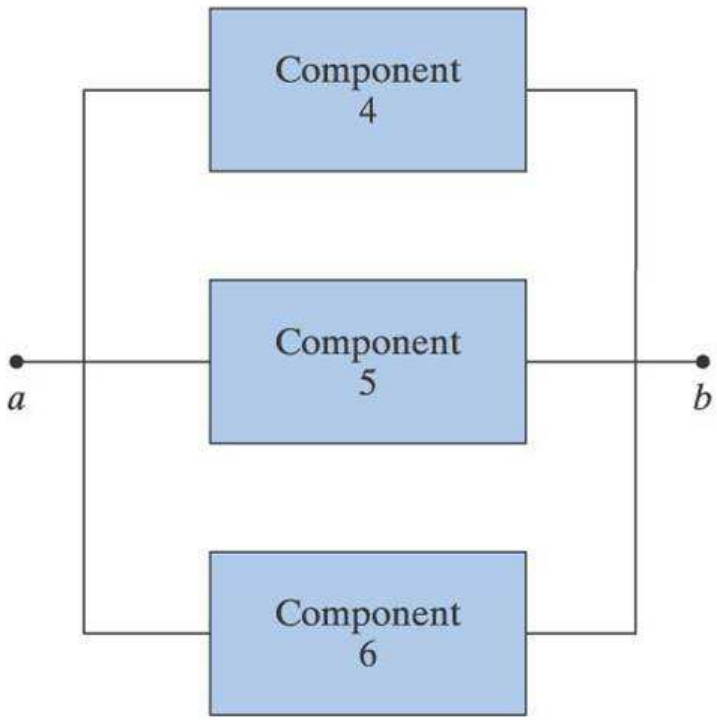
Instrumentation Reliability

- Equations for analyzing the reliability of instrumentation can be developed from the study of statistics and probability.
- **Reliability** is the proportion of the time that the component works properly.
- A component with a reliability of 0.8 works properly 80% of the time.
- Reliability of combinations of components can be determined if the individual component reliabilities are known.

Series / Parallel Configurations



(a) Series design.



(b) Parallel design.

Instrumentation Reliability

- In order for information to flow from point *a* to point *b* in the series design, all three components must work properly.
- In the parallel design, only one of the three components must work properly.
- The reliability of a specific combination of components can be determined in two ways:
  - Analytical reliability can be computed using theorems and results from probability and statistics.
  - Computer simulation can be developed to give an estimate of the reliability.

Series Configuration

- For the series configuration, if r is the reliability of a component, and if all three components have the same reliability, then the reliability of the series configuration is r^3.
- If the reliability of each component is 0.8, then the analytical reliability of the series configuration is $(0.8)^3$, or 0.512.
- The series configuration should work properly 51.2% of the time.

Parallel Configuration

- For the parallel configuration, if r is the reliability of a component, and if all three components have the same reliability, then the reliability of the parallel configuration is $3r - 3r^2 + r^3$.

- If the reliability of each component is 0.8, then the analytical reliability of the series configuration is $3(0.8) - 3(0.8)^2 + (0.8)^3$, or 0.992.

- The parallel configuration should work properly 99.2% of the time.

Computer Simulation

- We can also estimate the reliability of these two designs using random numbers from a **computer simulation**.
- First, we need to simulate the performance of a single component.
- If the reliability of a component is 0.8, then it works properly 80% of the time.
- To simulate this performance, we generate a random value between 0 and 1.
- If the value is between 0 and 0.8, the component worked properly; else, it failed.

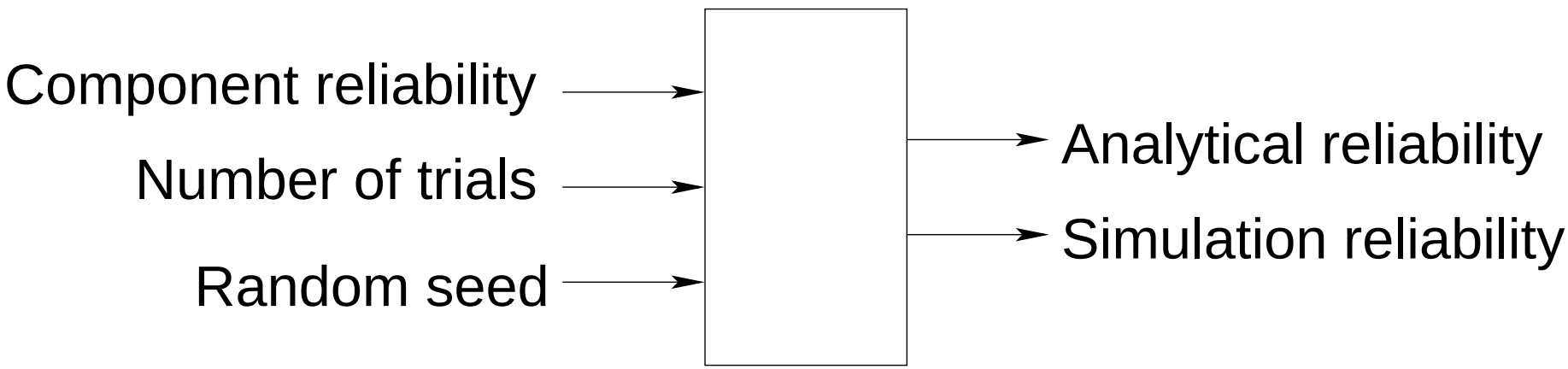
Computer Simulation

- We could also have used the values 0 to 0.2 for a failure and 0.2 to 1.0 for a component that worked properly.
- To simulate the series design with three components, we would generate three floating-point random numbers between 0 and 1.
- If all three numbers are less than or equal to 0.8, then the design works for this one trial.
- If any one of the three numbers is greater than 0.8, then the design does not work for this one trial.
- If we run many manytrials, we can compute the proportion of the time that the overall design works. This estimates the analytically computed reliability.

Computer Simulation

- To simulate the reliability of the parallel design with a component reliability of 0.8, we again generate three floating-point random numbers between 0 and 1.
- If any of the three numbers is less than or equal to 0.8, the design works for this trial.
- If all of the three numbers are greater than 0.8, the design does not work for this one trial.
- To estimate the reliability determined by the simulation, we divide the number of trials for which the design works by the total number of trials performed.

Input / Output Diagram



Hand Example

- For the hand example, we use a component reliability of 0.8 and three trials.

- Assume that the first nine random numbers generated are the following (generated from the `rand_float` function using a seed of 47 for values between 0 and 1):
  - First set of three random values:
    `0.005860 0.652303 0.271187`
  - Second set of three random values:
    `0.589007 0.064699 0.992248`
  - Third set of three random values:
    `0.719565 0.786615 0.001923`

Hand Example

- For the first group of three random numbers, all the values are less than 0.8; so both the series and parallel configurations would work properly.
- One of the values in the second set of numbers is greater than 0.8, so only the parallel configuration would work properly.
- Both configurations work properly with the third set of numbers.

Hand Example

- The analytical results and the simulation results for three trials are the following:
  - ```
    Analytical Reliability:
    Series:  0.512 Parallel:  0.992
    Simulation for 3 trials
    Series:  0.667 Parallel:  1.000
    ```

- As we increase the number of trials, the simulation results should approach the analytical results.

- If we change the random number seed, the simulation results may also change even with only three trials.
    ```

Algorithm Development

- The decomposition outline is:
  - 1. Read component reliability, number of trials, and random number seed.
  - 2. Compute analytical reliabilities.
  - 3. Compute simulation reliabilities.
  - 4. Print comparison of reliabilities.

- Step 1 prompts the user to enter the necessary information and then read it.

- Step 2 uses the equations given earlier to compute the analytical reliabilities. The computations are easy so we compute them in the `main` function.

Algorithm Development

- Step 3 involves a loop to generate the random numbers and to determine whether the configurations would perform properly for each trial. The `rand_float` function is used to compute the random numbers in the loop.

- Step 4 prints the results of the computations.

- The structure chart for this solution is shown in the next slide.

C Program Code

```
/*-------------------------------------------------*/
/* Program chapter4_6                              */
/*                                                 */
/* Estimate the reliability of a series and a      */
/* parallel configuration using a computer simulation */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*  Declare function prototype.  */
double rand_float(double a,double b);

int main(void)
{
   unsigned int seed;
   int n, k;
   double component_reliability, a_series, a_parallel,
          series_success=0, parallel_success=0, num1, num2, num3;
```

Code on Next 2 Slides

```
   /* Exit program. */ return 0; }
```

Body code for Program chapter4_6 (Part 1/2)

```
/*  Get information for the simulation.  */
printf("Enter individual component reliability: \n");
scanf("%lf",&component_reliability);
printf("Enter number of trials: \n");
scanf("%i",&n);
printf("Enter unsigned integer seed: \n");
scanf("%u",&seed);
srand(seed);
printf("\n");

/*  Compute analytical reliabilities.  */
a_series = pow(component_reliability,3);
a_parallel = 3*component_reliability
             - 3*pow(component_reliability,2)
             + pow(component_reliability,3);
```

Body code for Program chapter4_6 (Part 2/2)

```
/*  Determine simulation reliability estimates.  */
for (k=1; k<=n; k++)  {
   num1 = rand_float(0,1);
   num2 = rand_float(0,1);
   num3 = rand_float(0,1);
   if (((num1<=component_reliability) &&
       (num2<=component_reliability)) &&
       (num3<=component_reliability))
          series_success++;
   if (((num1<=component_reliability) ||
       (num2<=component_reliability)) ||
       (num3<=component_reliability))
          parallel_success++;
}
/*  Print results.  */
printf("Analytical Reliability \n");
printf("Series: %.3f  Parallel: %.3f \n",a_series,a_parallel);
printf("Simulation Reliability, %i trials \n",n);
printf("Series: %.3f  Parallel: %.3f \n",
        (double)series_success/n,
        (double)parallel_success/n);
```

Code for Function `rand_float`

```
/*-------------------------------------------------------*/
/*  This function generates a random                     */
/*  double value between a and b (a<b).                  */

double rand_float(double a,double b)
{
   return ((double)rand()/RAND_MAX)*(b-a) + a;
}
/*-------------------------------------------------------*/
```

Testing

- Using the hand example gives the following interaction:
  - ```
    Enter individual component reliability:
    0.8
    Enter number of trials:
    3
    Enter unsigned integer seed:
    47

    Analytical Reliability:
    Series:  0.512 Parallel:  0.992
    Simulation for 3 trials
    Series:  0.667 Parallel:  1.000
    ```
    ```

Testing

- As the number of trials increase...
  - ```
    Enter individual component reliability:
    0.8
    Enter number of trials:
    100
    Enter unsigned integer seed:
    123

    Analytical Reliability:
    Series:  0.512 Parallel:  0.992
    Simulation for 100 trials
    Series:  0.470 Parallel:  1.000
    ```
    ```

Testing

- Simulation results approach analytical results:
  - ```
    Enter individual component reliability:
    0.8
    Enter number of trials:
    1000
    Enter unsigned integer seed:
    3535

    Analytical Reliability:
    Series:  0.512 Parallel:  0.992
    Simulation for 1000 trials
    Series:  0.530 Parallel:  0.990
    ```
    ```

Computer Simulation - Summary

- We can use computer simulations to provide a validation for the analytical results because the simulated reliability should approach the analytically computed reliability as the number of trials increases.
- There are also cases in which it is very difficult to analytically compute the reliability of a piece of instrumentation.
- In these cases, a computer simulation can be used to provide a good estimate of the reliability.

Macros

- Before compiling a program, the preprocessor performs any actions specified by preprocessing directives, such as the inclusion of header files or the symbolic definition of constants.

- A simple operation can also be specified by a preprocessing directive called a **macro**, which has the following general form:
  - `#define` macro_name(parameters) macro_text

- The macro_text replaces references to the macro_name in the program.

Macros

- If the macro does not have parameters, then it is essentially a symbolic constant.
- If a macro has parameters, then it can represent a simple function.
- If the description of the macro takes more than one line, a backslash (\) must be used at the end of each line (except the last line) to indicate that the line is continued on the next line.

Macros

- An advantage of using a macro instead of a function is that the macro does not need to be defined in a separate module.
- Thus, the compilation and linking/loading process is simplified, and the execution time is reduced.
- During preprocessing, each reference to the macro is replaced with the macro text.
- Consider the following simple program that converts degrees Fahrenheit to degrees Centigrade.

C Program Code

```
/*-----------------------------------------------*/
/*  Program chapter4_8                           */
/*                                               */
/*  This program converts a temperature in       */
/*  Fahrenheit to Centigrade.                    */

#include <stdio.h>
#define degrees_C(x) (((x) - 32)*(5.0/9.0))

int main(void)
{
   double temp;

   /*  Get temperature in Fahrenheit.  */
   printf("Enter temperature in degrees "
          "Fahrenheit: \n");
   scanf("%lf",&temp);

   /*  Convert and print temperature in Centigrade.  */
   printf("%f degrees Centigrade \n",degrees_C(temp));

   /*  Exit program.  */
   return 0; }
```

Macros

- When the `printf` statement from the program is compiled, the macro text replaces the macro reference, giving the following:
  - ```
    printf("%f degrees Centigrade \n",
            (((temp) - 32)*(5.0/9.0)));
    ```
- When this statement is executed, the value in `temp` is correctly converted from degrees Fahrenheit to degrees Centigrade before it is printed.
    ```

Parentheses

- It is important to include parentheses around each individual argument and around the complete macro_text in the macro definition so that the macro will work properly when it is referenced with an expression as an actual parameter.

- Consider these macros that convert temperatures in degrees Centigrade to degrees Fahrenheit:
  - `#define degrees1_F(x) ((x)*(9.0/5.0) + 32)`
  - `#define degrees2_F(x) x*(9.0/5.0) + 32`

Parentheses

- When these macros are used with a variable as an actual parameter, they both work properly.

- The statements:
  - `max_temp1 = degrees1_F(temp);`
  - `max_temp2 = degrees2_F(temp);`

- are correctly compiled as the following equivalent computations:
  - `max_temp1 = ((temp)*(9.0/5.0) + 32);`
  - `max_temp2 = temp*(9.0/5.0) + 32;`

Parentheses

- However, the statements:
  - `max_temp1 = degrees1_F(temp+10);`
  - `max_temp2 = degrees2_F(temp+10);`

- are compiled as the following statements, which do not yield the same values:
  - `max_temp1 = ((temp+10)*(9.0/5.0) + 32);`
  - `max_temp2 = temp+10*(9.0/5.0) + 32;`

- Therefore, the parentheses around the macro arguments and around the macro_text are necessary to ensure correct calculations.

Area of a Triangle

- The following macro computes the area of a triangle with a specified base and height:

```
#define area_tri(base,height)(0.5*(base)*(height))
```

- Note that parentheses are included around each argument and around the complete macro_text in the macro definition.

Macros vs Functions

A macro is simpler to execute because there is no action of transfering to and returning from a function. In particular, actual and formal parameters need not be passed. But macros have disadvantages:

- Program Size
  Macros are expanded at compile-time, and hence its expression is replaced in its entirety wherever it is used. In particular, if it is used several times, then several copies of the macro body are replaced in the program. This can lead to a significant increase in the size of the program.
  Each function is only defined once, even if each function is called several times.

- Local variables
  A function can have local variables whose effects can be isolated from the main program (or from another calling function).

Macro Example

```
#define abss(x) ((x) > 0 ? (x) : -(x))
#define maxx(x, y) ((x) > (y) ? (x) : (y))
```

Then the expression `abs(max(2, 3))` is expanded into

(((2) > (3) ? (2) : (3)) > 0 ? ((2) > (3) ? (2) : (3)) : -((2) > (3) ? (2) : (3)))

Note that the subexpression ((2) > (3) ? (2) : (3)) is repeated three times.

Function Example

```
int abss(int x) {
   return (x > 0 ?  x :  -x);
}

int maxx(int x, int y) {
   return (x > y ?  x :  y);
}
```

Then the function calll `abs(max(2, 3))` leads to the following computation steps:

```
abss(maxx(2, 3)) ⟶
   abss((2 > 3 ?  2 :  3)) ⟶
      abss(3) ⟶
         (3 > 0 ?  3 :  0) ⟶
            3
```

Note that the subexpression ((2) > (3) ? (2) : (3)) is executed only once.

Recursion

- A function that invokes itself (or calls itself) is a **recursive function**.
- Recursion can be a powerful tool for solving certain classes of problems in which the solution can be defined in terms of a similar but smaller problem.
- Then the smaller problem can be defined in terms of a similar but still smaller problem.
- This redefinition of the problem into smaller problems continues until the smaller problem has a unique solution that is then used to determine the overall solution.
- There are system-dependent limitations to the number of times that a recursive function can call itself as it is continually redefining a problem into smaller and smaller problems.

Recursion

- We illustrate problems that can be solved with a recursive algorithm.
- In the examples, note the two parts to a recursive solution.
- First, the solution has to be redefined in terms of a similar, but smaller, problem.
- Second, the smaller problems must reach a point at which there is a unique solution.

Factorial Computation

- A simple example of recursion can be shown using the **factorial** computation.

- Recall that $k!$ (read as k factorial) is defined as:

$$k! = (k)(k-1)(k-2)\cdots(3)(2)(1)$$

  where k is a nonnegative integer, and where $0! = 1$.

- Thus,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

Factorial Computation

- We could also compute 5! using the following steps:

$$5! = 5 \cdot 4!$$
$$4! = 4 \cdot 3!$$
$$3! = 3 \cdot 2!$$
$$2! = 2 \cdot 1!$$
$$1! = 1 \cdot 0!$$
$$0! = 1.$$

Factorial Computation

- We have defined a factorial in terms of a product that involves smaller factorials.
- The smallest factorial is continually redefined until we reach 0!.
- We substitute the value of 0! in the last equation, and then begin going back up the list of equations, substituting values for the factorials.

Factorial Computation

- This is shown below:

$$
\begin{aligned}
0! &= 1 \\
1! &= 1 \cdot 0! = 1 \\
2! &= 2 \cdot 1! = 2 \\
3! &= 3 \cdot 2! = 6 \\
4! &= 4 \cdot 3! = 24 \\
5! &= 5 \cdot 4! = 120.
\end{aligned}
$$

- We have now developed a recursive algorithm for computing a factorial.

Factorial Computation

- We present a program with two functions to compute a factorial.
- The first function is a nonrecursive (iterative) function.
- The second function is a recursive function.
- A factorial value becomes large quickly, so we use long integers for the factorial value.
- Note that both functions are referenced similarly in the `main` function.

C Program Code

```
/*-------------------------------------------*/
/*  Program chapter4_9                       */
/*                                           */
/* Compare a recursive function and a nonrecursive */
/* function for computing factorials.        */

#include <stdio.h>

/*  Declare function prototypes.  */
long factorial(int k);
long factorial_r(int k);

int main(void)
{
   int n;
   /*  Get user input.  */
   printf("Enter positive integer: \n");
   scanf("%i",&n);
   /*  Compute and print factorials.  */
   printf("Nonrecursive: %i! = %li \n",n,factorial(n));
   printf("Recursive: %i! = %li \n",n,factorial_r(n));
   /*  Exit program.  */
   return 0; }
```

Code for Nonrecursive Function `factorial`

```
/*-------------------------------------------------*/
/*  This function computes a factorial with a loop.  */

long factorial(int k)
{
   int j;
   long term;

   /*  Compute factorial with multiplication.  */
   term = 1;
   for (j=2; j<=k; j++)
       term *= j;

   /*  Return factorial value.  */
   return term;
}
```

Code for Recursive Function

factorial

```
/*-------------------------------------------------*/
/*  This function computes a factorial recursively.  */

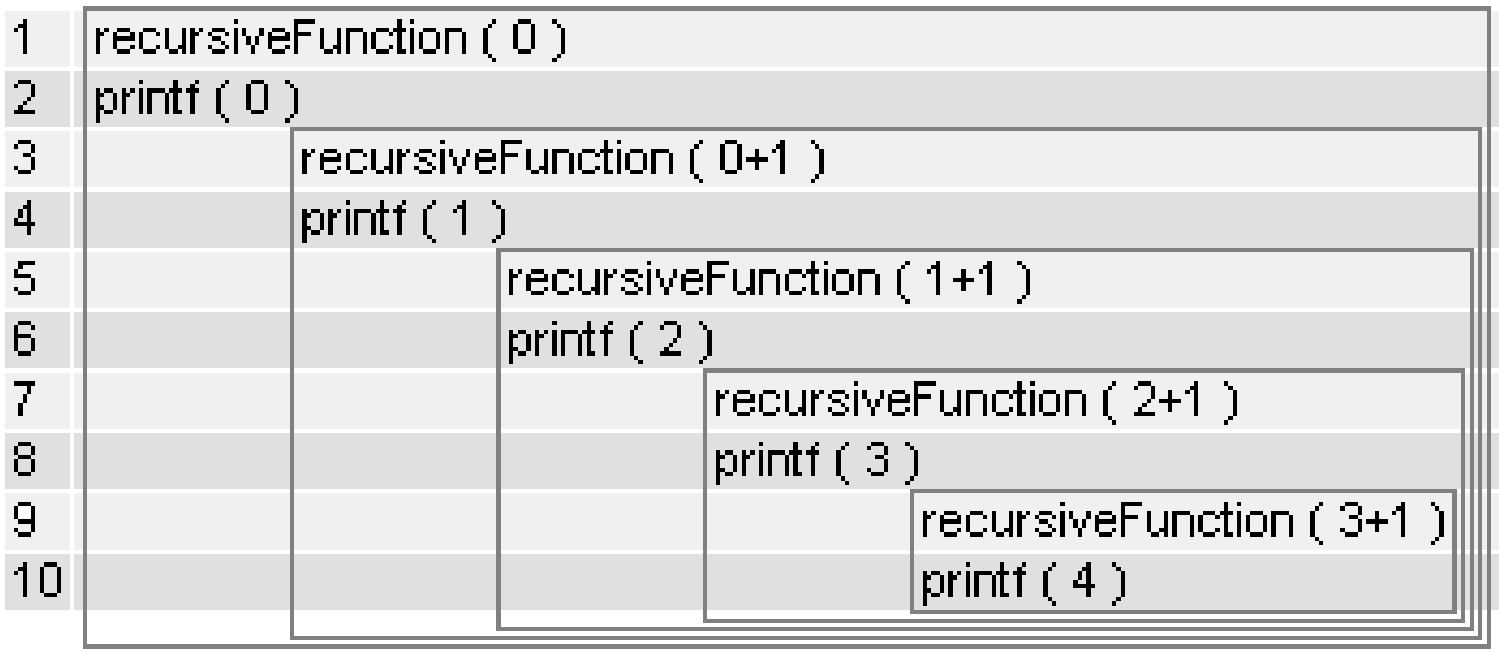
long factorial_r(int k)
{
   /*  Recursive reference until k is equal to 0.  */
   if (k == 0)
      return 1;
   else
      return k*factorial_r(k-1);
}
/*-------------------------------------------------*/
```

Factorial Computation

- The condition `k == 0` keeps the recursive routine from becoming an infinite loop.
- This routine calls itself recursively with an argument that is continually being decremented by 1, until the argument reaches zero.
- For large values of k, the value of $k!$ can exceed even long integers. In these cases, the computations should be done using `double` or `long double` values.
- An interesting approximation to $k!$ will also be presented in the tutorial questions.

Visualizing Recursive Functions

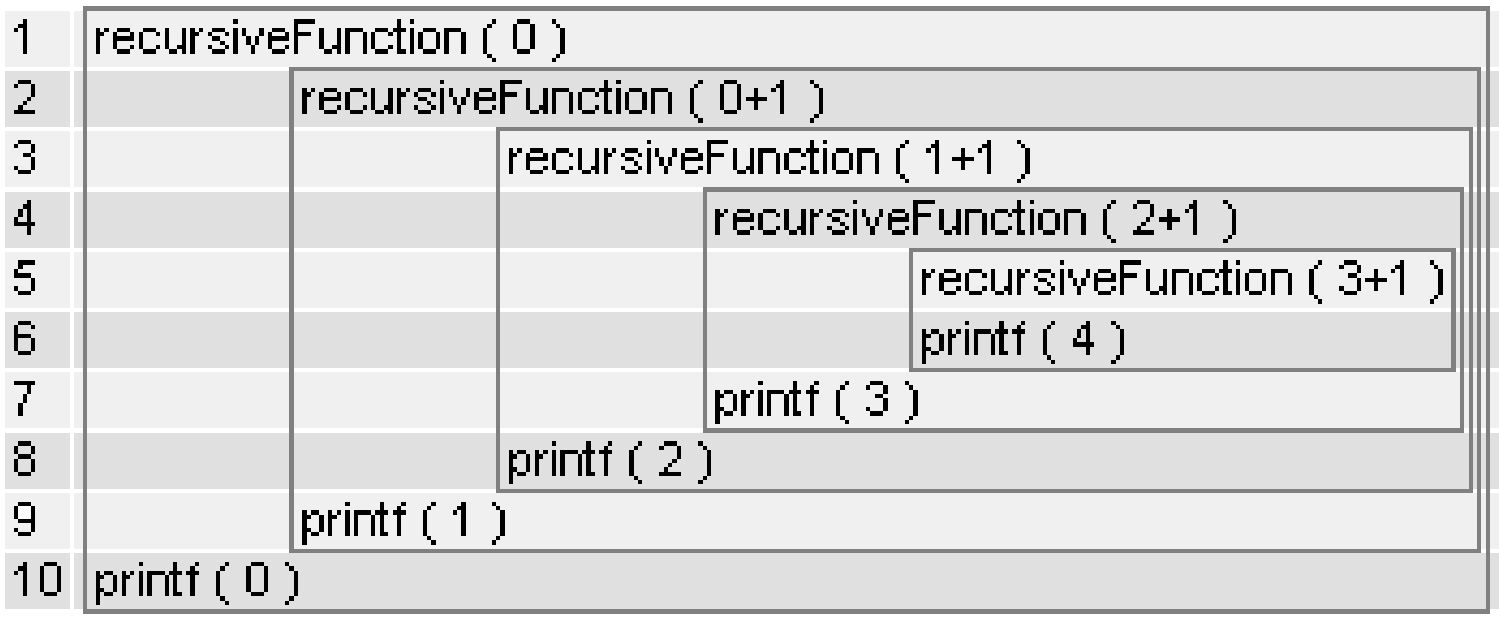
```
void recursiveFunction(int num) {
    if (num < 5) {
        printf("%d ", num);
        recursiveFunction(num + 1);
    }
}
```



Output: `0 1 2 3 4`

Visualizing Recursive Functions

```
void recursiveFunction(int num) {
    if (num < 5) {
        recursiveFunction(num + 1);
        printf("%d ", num);
    }
}
```



Output: `4 3 2 1 0`

Recursion - Divide and Conquer

Recall the *factorial* example:

```
int fact(int n) {
    return (n <= 1) ? 1 : n * fact(n - 1);
}
```

The formulation of this program was straightforward from the *definition* of the factorial function:

$$fact(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n * fact(n-1) & \text{if } n > 1 \end{cases}$$

Sometimes the formulation of a recursive function is not so straightforward.

Recursion - Divide and Conquer

- A perfect number is one which is the sum of its proper positive divisors. Eg 6 = 1 + 2 + 3 is the first perfect number. Next 28 = 1 + 2 + 4 + 7 + 14.

- Write a recursive function to detect a perfect number.

- The idea:
  a function `sumdiv(n, k)` sums up the divisors of n *up to k*.
  The base case is easy: `sumdiv(n, 1)` = 1.
  The recursive case:

$$sumdiv(n,k) = \begin{cases} sumdiv(n,k-1)+k & \text{if } k \text{ divides } n \\ sumdiv(n,k-1) & \text{otherwise} \end{cases}$$

Perfect Numbers

```
int int sumdiv(int n, int div);

int main() {
    int n = 2, i;
    do {
        i = sumdiv(n, n-1);
        if (n == i) printf("%d ", n);
    } while(++n < 8129);
}

int sumdiv(int n, int div) {
    if (div <= 1) return 1;
    return (n % div == 0) ?
        div + sumdiv(n, div - 1) : sumdiv(n, div - 1);
}
```

Output: 6 28 496 8128

Fibonacci Sequence

- A **Fibonacci sequence** is a sequence of numbers $(f_0, f_1, f_2, f_3, \dots)$ in which the first two values (f_0 and f_1) are equal to 1, and each succeeding number is the sum of the previous two numbers.

- The first few values of the Fibonacci sequence are:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \quad \dots$$

- This sequence was first described in the year 1202, and it has applications that range from biology to electrical engineering.

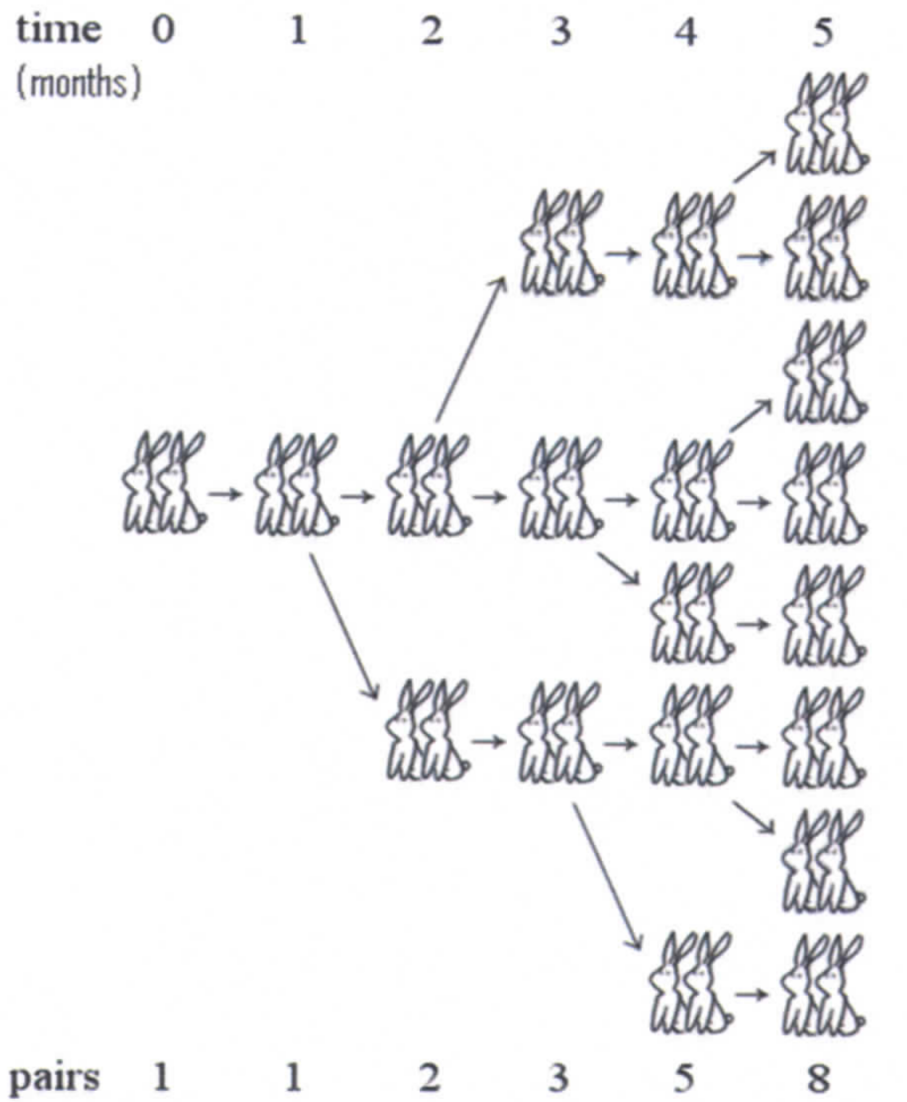
Fibonacci Sequence

- Fibonacci sequences are often used in studies of rabbit population growth.

- A function to compute the *k*th value in the Fibonacci sequence is a good candidate for a recursive function because each new value in the sequence is computed from the two previous values.

- Start with one (male/female) pair of rabbits.
  They can mate at the age of one month.
  At the end of second month, a pair can produce another pair of rabbits, so now there a two pairs.
  At the end of the third month, these two pairs make two more, so total is now four pairs.
  In general, how many pairs after *N* months?

Fibonacci Sequence (rabbits)



C Program Code

```
/*-----------------------------------------*/
/* Program chapter4_10                     */
/*                                         */
/* Compare a recursive and nonrecursive function */
/* for computing Fibonacci numbers.        */

#include <stdio.h>

/*  Declare function prototypes.  */
int fibonacci(int k);
int fibonacci_r(int k);

int main(void)
{
   int n;
   /*  Get user input.  */
   printf("Enter positive integer: \n");
   scanf("%i",&n);
   /*  Compute and print Fibonacci numbers.  */
   printf("Nonrecursive: Fibonacci number = %i \n",
          fibonacci(n));
   printf("Recursive:    Fibonacci number = %i \n",
          fibonacci_r(n));
   /*  Exit program.  */
   return 0;  }
```

Code for Function `fibonacci`

```
/*-------------------------------------------*/
/*  This function computes the kth Fibonacci */
/*  number using a nonrecursive algorithm.   */

int fibonacci(int k)
{
   /*  Declare variables. */
   int term, prev1, prev2, n;
   /*  Compute kth Fibonacci number with a loop.  */
   term = 1;
   if (k > 1)
   {
      prev1 = prev2 = 1;
      for (n=2; n<=k; n++)
      {
         term = prev1 + prev2;
         prev2 = prev1;
         prev1 = term;
      }
   }
   /*  Return kth Fibonacci number.  */
   return term;
}
```

Code for Function `fibonacci_r`

```
/*-------------------------------------------------*/
/*  This function computes the kth Fibonacci       */
/*  number using a recursive algorithm.            */

int fibonacci_r(int k)
{
   /*  Declare variables.  */
   int term;

   /*  Compute kth Fibonacci number recursively  */
   /*  until k == 1.                             */

   term = 1;
   if (k > 1)
      term = fibonacci_r(k-1) + fibonacci_r(k-2);
   /*  Return kth Fibonacci number.  */
   return term;
}
/*-------------------------------------------------*/
```

When to use Recursive Functions?

- Sometimes the problem itself is stated recursively (factorial)
- Sometimes it is succinct to state the solution recursively (fibonacci, perfect numbers)
- But sometimes a recursive function is much slower than an appropraite non-recursive function (again, fibonacci)
  WHY ?
- Sometimes, only a recursive solution is evident (next example)

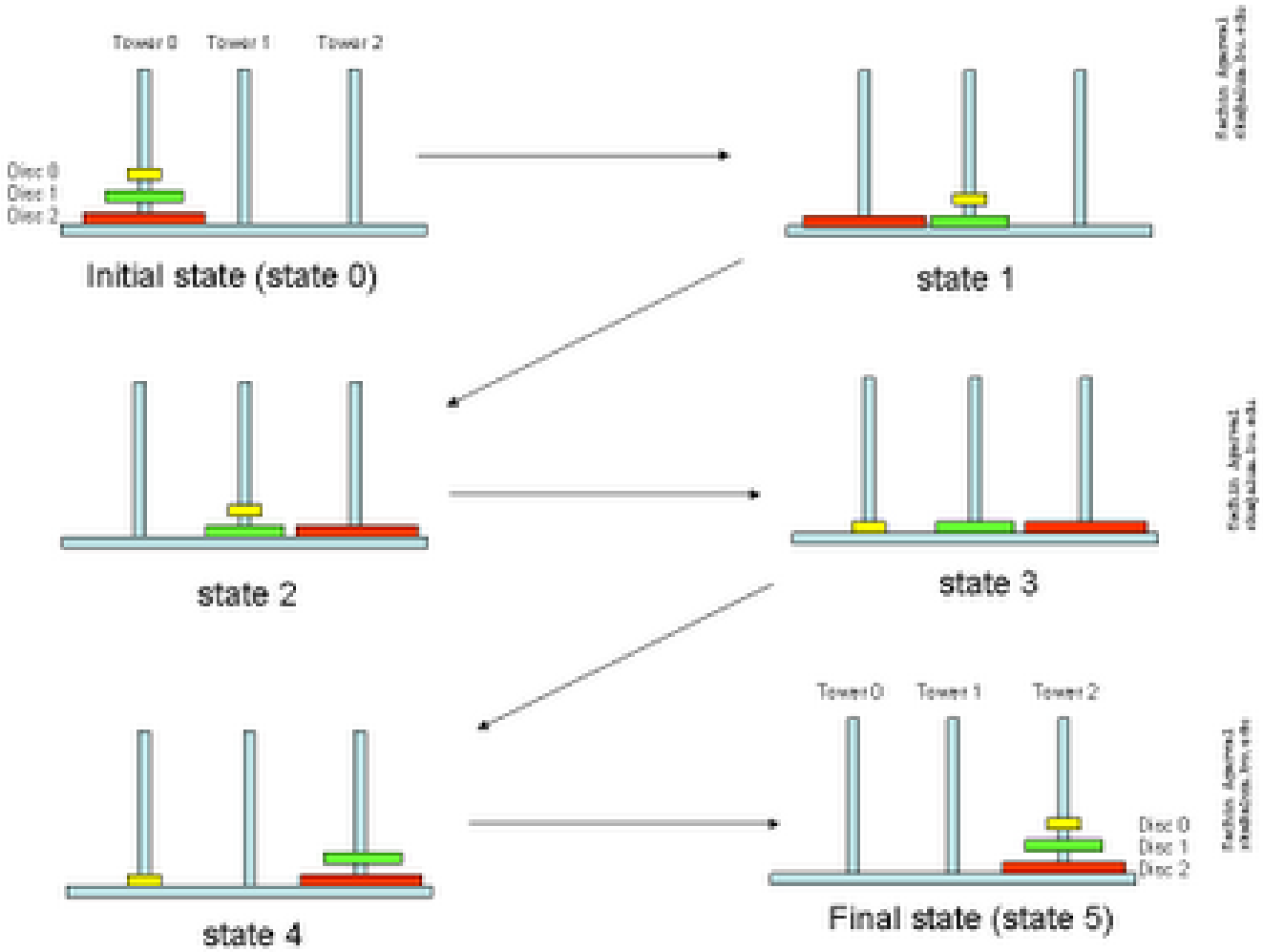
A Big Difference between Factorial and Fibonacci

- FACTORIAL(n):
  The number of *calls* to the procedure is proportional to n.

- FIBONACCI(n):
  The number of *calls* to the procedure is proportional to $fib(n)$
  Note that $fib(n)$ is *"much"* larger than n.
  (In fact, $fib(n)$ is *exponentially larger* than n.)

Towers of Hanoi

Given 3 pegs with a number of disks arranged in order of size (bottommost is biggest) on peg 1, move all to peg 3 one at a time but never having to place a disk on a smaller one.



Recursive Functions - Towers of Hanoi

```
void hanoi(int n, char LEFT, char RIGHT, char CENTER) {
    if( n > 0) {
        hanoi(n-1, LEFT, CENTER, RIGHT);
        printf("\ndisk %d from %c to %c", n, LEFT, RIGHT);
        hanoi(n-1, CENTER, RIGHT, LEFT);
    }
}

void main()
{
    int n;
    printf("\nEnter no. of disks: ");
    scanf("%d",&n);
    hanoi(n,'L','R','C');
}
```

See: http://www.mazeworks.com/hanoi/index.htm

```
Enter no. of disks: 5

disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 3 from L to R
disk 1 from C to L
disk 2 from C to R
disk 1 from L to R
disk 4 from L to C
disk 1 from R to C
disk 2 from R to L
disk 1 from C to L
disk 3 from R to C
disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 5 from L to R
disk 1 from C to L
disk 2 from C to R
disk 1 from L to R
disk 3 from C to L
disk 1 from R to C
disk 2 from R to L
disk 1 from C to L
disk 4 from C to R
disk 1 from L to R
disk 2 from L to C
disk 1 from R to C
disk 3 from L to R
disk 1 from C to L
disk 2 from C to R
```

References

Etter Sections 4.5 to 4.9

Next Lecture

Arrays and Matrices

(Part 1)