CS11001/CS11002
Programming and Data Structures (PDS) (Theory: 3-0-0)

Class Teacher: Pralay Mitra

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

An Example: Random Number Generation

```
/* A programming example of
Randomized die-rolling */
#include <stdio.h>
#include <stdlib.h>

void main()
{
     int i;
     unsigned seed;

     printf("Enter seed: ");
     scanf("%u",&seed);
     srand(seed);
     for(i=1;i<=10;i++) {
          printf("%10d ",1+(rand()%6));
          if(i%5==0)
               printf("\n");
     }
}
```

Algorithm

1. Initialize seed
2. Input value for seed
   2.1 Use srand to change random sequence
   2.2 Define Loop
3. Generate and output random numbers

```
Enter seed: 293
      2       4       1       5       3
      3       1       1       2       6
```

```
Enter seed: 67
      6       4       4       6       4
      3       6       1       4       2
```

```
Enter seed: 867
      5       5       2       3       5
      4       2       2       3       4
```

Passing Arrays to a Function

- An array name can be used as an argument to a function.
  - Permits the entire array to be passed to the function.
  - Array name is passed as the parameter, which is effectively the address of the first element.
- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts.
  - The corresponding formal argument is written in the same manner.
    - Declared by writing the array name with a pair of empty brackets.
    - Dimension or required number of elements to be passed as a separate parameter.

Example 1: Minimum of a set of numbers

We can also write

int x[100];

But the way the function is written makes it general; it works with arrays of any size.

```
#include  <stdio.h>

void main()
{
   int  a[100], i, n;

   scanf (“%d”, &n);
   for  (i=0; i<n; i++)
      scanf (“%d”, &a[i]);

   printf (“\n Minimum is %d”,minimum(a,n));
}
```

```
int  minimum (int x[], int size)
{
   int  i, min = 99999;

   for  (i=0; i<size; i++)
      if  (min > x[i])
         min = x[i];

   return (min);
}
```

Parameter Passing mechanism

- When an array is passed to a function, the values of the array elements are *not passed* to the function.
  - The array name is interpreted as the *address* of the first array element.
  - The formal argument therefore becomes a *pointer* to the first array element.
  - When an array element is accessed inside the function, the address is calculated using the formula stated before.
  - Changes made inside the function are thus also reflected in the calling program.

Parameter Passing mechanism

- Passing parameters in this way is called

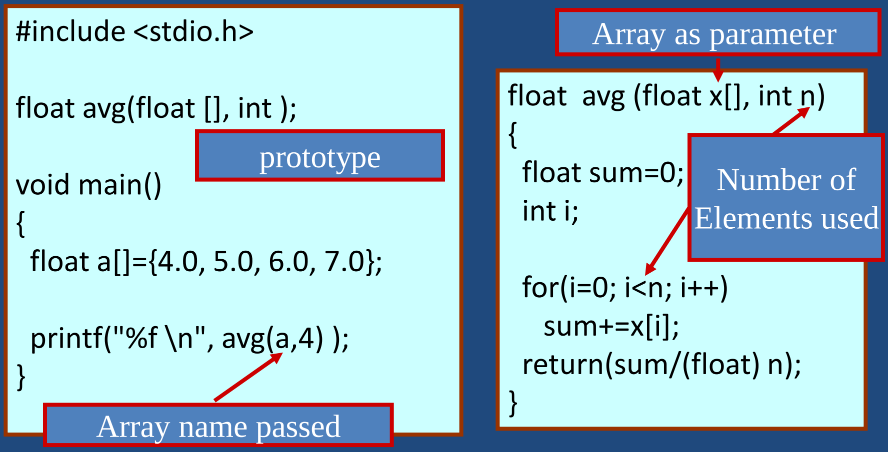
  call-by-reference.

- Normally parameters are passed in C using

  call-by-value.

- Basically what it means?
  - If a function changes the values of array elements, then these changes will be made to the original array that is passed to the function.
  - This does not apply when an individual element is passed on as argument.

Example: Average of numbers



```
#include <stdio.h>

float avg(float [], int );

void main()
{
  float a[]={4.0, 5.0, 6.0, 7.0};

  printf("%f \n", avg(a,4) );
}
```

prototype

Array name passed

Array as parameter

```
float  avg (float x[], int n)
{
  float sum=0;
  int i;

  for(i=0; i<n; i++)
    sum+=x[i];
  return(sum/(float) n);
}
```

Number of Elements used

Call by Value and Call by Reference

- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions

Example: Max Min function

```
/* Find maximum and minimum from a list of 10 integers */
#include <stdio.h>

void getmaxmin(int array[],int size,int maxmin[]);

void main()
{
        int a[20],i,maxmin[2];

        printf("Enter 10 integer values: ");
        for(i=0;i<10;i++)
                scanf("%d",&a[i]);
        getmaxmin(a,10,maxmin);
        printf("Maximum=%d, Minimum=%d\n",
                maxmin[0],maxmin[1]);
}
```

Still it can return value(s).

Return type of any function may be void

```
void getmaxmin(int array[],int size,int maxmin[])
{
        int i,max=-99999,min=99999;

        for(i=0;i<size;i++) {
                if(max<array[i])
                        max=array[i];
                if(min>array[i])
                        min=array[i];
        }
        maxmin[0]=max;
        maxmin[1]=min;
}
```

Returning multiple values from a function.

Scope of a variable

```
#include<stdio.h>
void print(int a)
{
        printf("3.1 in function value of a: %d\n",a);
        a+=23;
        printf("3.2 in function value of a: %d\n",a);
}
void main()
{
        int a=10,i=0;
        printf("1. value of a: %d\n",a);
        while(i<1) {
                int a;
                a=20;
                printf("2. value of a: %d\n",a);
                i++;
        }
        printf("3. value of a: %d\n",a);
        print(a);
        printf("4. VALUE of a: %d\n",a);
}
```

```
3.1 in function value of a: 10
3.2 in function value of a: 33
1. value of a: 10
2. value of a: 20
3. value of a: 10
4. value of a: 10
```

Storage Class of Variables

What is Storage Class?

- It refers to the permanence of a variable, and its *scope* within a program.

- Four storage class specifications in C:
  - Automatic: auto
  - External: extern
  - Static: static
  - Register: register

Automatic Variables

- These are always declared within a function and are local to the function in which they are declared.
  - Scope is confined to that function.
- This is the default storage class specification.
  - All variables are considered as **`auto`** unless explicitly specified otherwise.
  - The keyword **`auto`** is optional.
  - An automatic variable does not retain its value once control is transferred out of its defining function.

auto: Example

```
#include <stdio.h>

int factorial(int m)
{
  auto int i;
  auto int temp=1;
  for (i=1; i<=m; i++)
     temp = temp * i;
  return (temp);
}
```

```
void main()
{
  auto int  n;
  for (n=1; n<=10; n++)
    printf (“%d! = %d \n”,
           n, factorial (n));
}
```

Static Variables

- Static variables are defined within individual functions and have the same scope as automatic variables.
- Unlike automatic variables, static variables retain their values throughout the life of the program.
    - If a function is exited and re-entered at a later time, the static variables defined within that function will retain their previous values.
    - Initial values can be included in the static variable declaration.
        - Will be initialized only once.
- An example of using static variable:
    - Count number of times a function is called.

static: Example

```
#include <stdio.h>
void print()
{
    static int count=0;
    printf("Hello World!! ");
    count++;
    printf("is printing %d times.\n",count);
}
int main()
{
    int i=0;
    while(i<10) {
        print();
        i++;
    }
    return 0;
}
```

Output

```
Hello World!! is printing 1 times.
Hello World!! is printing 2 times.
Hello World!! is printing 3 times.
Hello World!! is printing 4 times.
Hello World!! is printing 5 times.
Hello World!! is printing 6 times.
Hello World!! is printing 7 times.
Hello World!! is printing 8 times.
Hello World!! is printing 9 times.
Hello World!! is printing 10 times.
```

External Variables

- They are not confined to single functions.
- Their scope extends from the point of definition through the remainder of the program.
  - They may span more than one functions.
  - Also called global variables.
- Alternate way of declaring global variables.
  - Declare them outside the function, at the beginning.

global: Example

```
#include <stdio.h>
int count=0;
void print()
{
      printf("Hello World!! ");
      count++;
}
int main()
{
      int i=0;

      while(i<10) {
            print();
            i++;
            printf("is printing %d times.\n",count);
      }
      return 0;
}
```

Output

```
Hello World!! is printing 1 times.
Hello World!! is printing 2 times.
Hello World!! is printing 3 times.
Hello World!! is printing 4 times.
Hello World!! is printing 5 times.
Hello World!! is printing 6 times.
Hello World!! is printing 7 times.
Hello World!! is printing 8 times.
Hello World!! is printing 9 times.
Hello World!! is printing 10 times.
```

static vs global

```c
#include <stdio.h>
void print()
{
	static int count=0;
	printf("Hello World!! ");
	count++;
	printf("is printing %d times.\n",count);
}
int main()
{
	int i=0;
	while(i<10) {
		print();
		i++;
	}
	return 0;
}
```

```c
#include <stdio.h>
int count=0;
void print()
{
	printf("Hello World!! ");
	count++;
}
int main()
{
	int i=0;

	while(i<10) {
		print();
		i++;
		printf("is printing %d times.\n",count);
	}
	return 0;
}
```

Register Variables

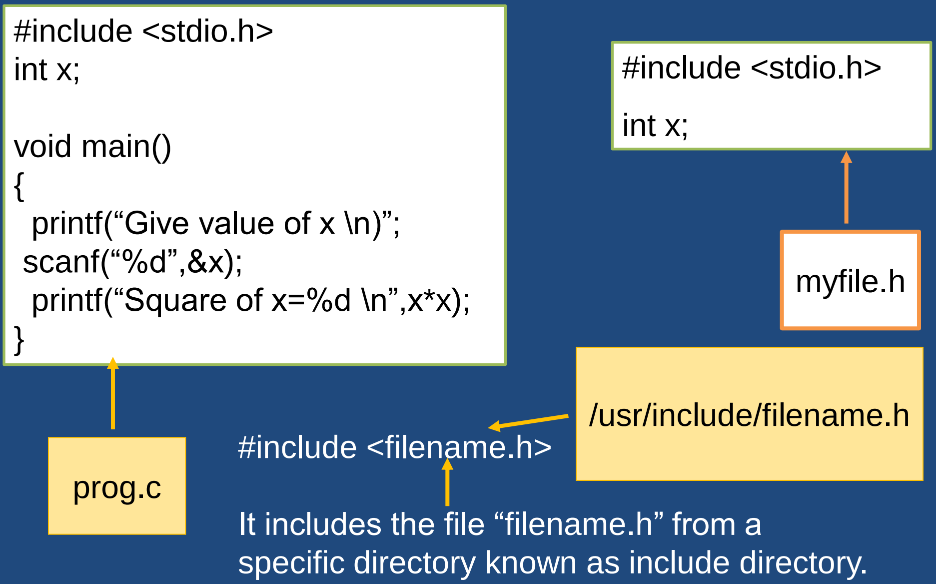
- These variables are stored in high-speed registers within the CPU.
  - Commonly used variables like loop variables/counters may be declared as register variables.
  - Results in increase in execution speed.
  - User can suggest, but the allocation is done by the compiler.

```c
#include<stdio.h>
int main()
{
   int sum;
   register int count;
   for(count=0;count<20;count++)
            sum=sum+count;
   printf("\nSum of Numbers:%d", sum);
   return(0);
}
```

#include: Revisited

- Preprocessor statement in the following form
  `#include “filename”`
- Filename could be specified with complete path.
  `#include “/home/pralay/C-header/myfile.h”`
- The content of the corresponding file will be included in the present file before compilation and the compiler will compile thereafter considering the content as it is.

#include: Revisited



```
#include <stdio.h>
int x;

void main()
{
  printf(“Give value of x \n)”;
 scanf(“%d”,&x);
  printf(“Square of x=%d \n”,x*x);
}
```

prog.c

```
#include <stdio.h>

int x;
```

myfile.h

`#include <filename.h>`

/usr/include/filename.h

It includes the file “filename.h” from a specific directory known as include directory.

Variable number of arguments

- General syntax:

  scanf (control string, arg1, arg2, …, argn);
  printf (control string, arg1, arg2, …, argn);

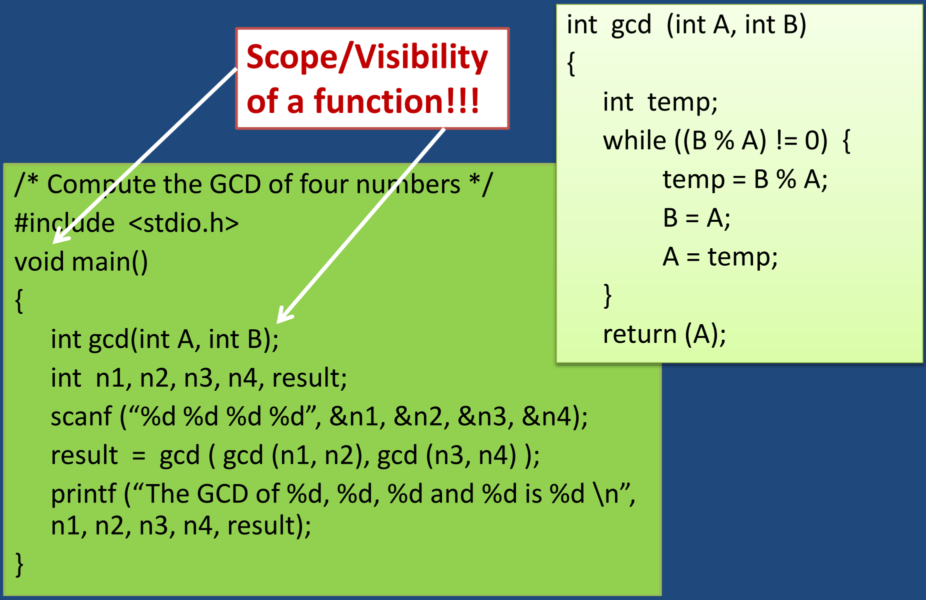
How is it possible?

Example: GCD calculation

```
/* Compute the GCD of four numbers */
#include  <stdio.h>
int gcd(int A, int B);
void main()
{
    int  n1, n2, n3, n4, result;
    scanf (“%d %d %d %d”, &n1, &n2, &n3, &n4);
    result  =  gcd ( gcd (n1, n2), gcd (n3, n4) );
    printf (“The GCD of %d, %d, %d and %d is %d \n”,
    n1, n2, n3, n4, result);
}
```

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
            temp = B % A;
            B = A;
            A = temp;
    }
    return (A);
```

Example: GCD calculation



Scope/Visibility of a function!!!

```
/* Compute the GCD of four numbers */
#include  <stdio.h>
void main()
{
    int gcd(int A, int B);
    int  n1, n2, n3, n4, result;
    scanf (“%d %d %d %d”, &n1, &n2, &n3, &n4);
    result  =  gcd ( gcd (n1, n2), gcd (n3, n4) );
    printf (“The GCD of %d, %d, %d and %d is %d \n”,
    n1, n2, n3, n4, result);
}
```

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
          temp = B % A;
          B = A;
          A = temp;
    }
    return (A);
```

Recursion

- A process by which a function calls itself repeatedly.
  - Either directly.
    - X calls X.
  - Or cyclically in a chain.
    - X calls Y, and Y calls X.
- Used for repetitive computations in which each action is stated in terms of a previous result.
  - fact(n) = n * fact (n-1)

Recursion

- For a problem to be written in recursive form, two conditions are to be satisfied:
  - It should be possible to express the problem in recursive form.
  - The problem statement must include a stopping condition

    fact(n) = 1, if n = 0
    = n * fact(n-1), if n > 0

Recursion

- Examples:
  - Factorial:

    fact(0) = 1
    fact(n) = n * fact(n-1), if n > 0

  - GCD:

    gcd (m, m) = m
    gcd (m, n) = gcd (m-n, n), if m > n
    gcd (m, n) = gcd (n, n-m), if m < n

  - Fibonacci series (1,1,2,3,5,8,13,21,….)

    fib (0) = 1
    fib (1) = 1
    fib (n) = fib (n-1) + fib (n-2), if n > 1

Example 1 :: Factorial

```
#include <stdio.h>

int fact(int n)
{
        if (n == 0)
                return 1;
        else
                return (n * fact(n-1));
}
void main()
{
         int i=6;
         printf (“Factorial of 6 is: %d \n”, fact(i));
}
```

Mechanism of Execution

- When a recursive program is executed, the recursive function calls are not executed immediately.
  - They are kept aside (on a stack) until the stopping condition is encountered.
  - The function calls are then executed in reverse order.

Advantage of Recursion :: Calculating fact(5)

– First, the function calls will be processed:

fact(5) = 5 * fact(4)
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
fact(1) = 1 * fact(0)

– The actual values return in the reverse order:

fact(0) = 1
fact(1) = 1 * 1 = 1
fact(2) = 2 * 1 = 2
fact(3) = 3 * 2 = 6
fact(4) = 4 * 6 = 24
Fact(5) = 5 * 24 = 120

Facts on fact

- `5! = 5 * 4 * 3 * 2 * 1`
- Notice that
  - `5! = 5 * 4!`
  - `4! = 4 * 3! ...`
- Can compute factorials recursively
- Solve base case (`1! = 0! = 1`) then plug in
  - `2! = 2 * 1! = 2 * 1 = 2;`
  - `3! = 3 * 2! = 3 * 2 = 6;`

Example 2 :: Fibonacci series

```
#include <stdio.h>

int fib(int n)
{
        if (n < 2)
            return n;
        else
            return (fib(n-1) + fib(n-2));
}
void main()
{
         int i=4;
         printf ("%d \n", fib(i));
}
```

Execution of Fibonacci number

- Fibonacci number fib(n) can be defined as:

  fib(0) = 0

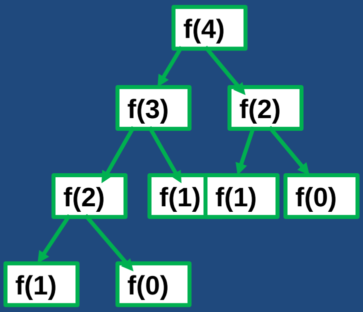
  fib(1) = 1

  fib(n) = fib(n-1) + fib(n-2), if n > 1

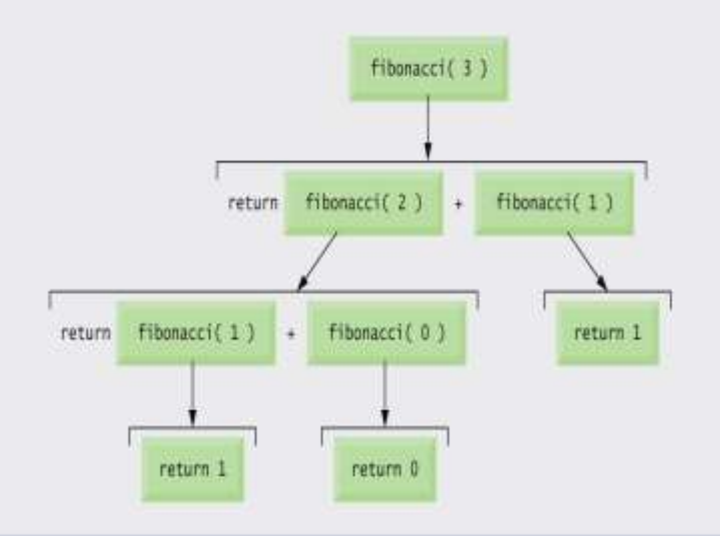
  - The successive Fibonacci numbers are:

    0, 1, 1, 2, 3, 5, 8, 13, 21, …..

```
int  fib(int n)
{
   if  (n  < 2)
          return (n);
   else
          return (fib(n-1) + fib(n-2));
}
```

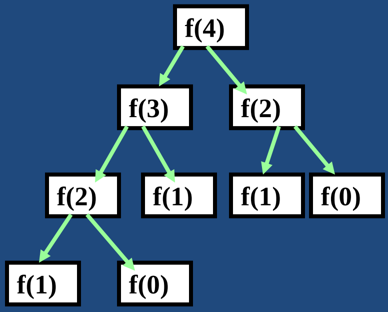


Set of recursive calls for `fibonacci(3)`



Inefficiency of Recursion

- How many times the function is called when evaluating f(4) ?

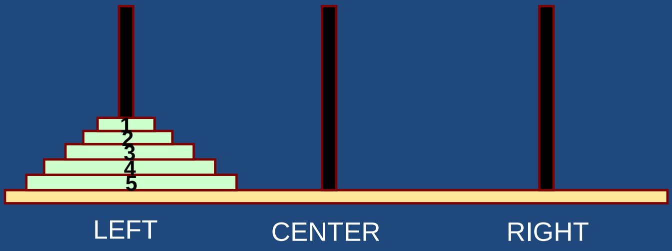

f(4)
f(3)
f(2)
f(2)
f(1)
f(1)
f(0)
f(1)
f(0)

- Same thing is computed several times.

Performance Tip

- Avoid Fibonacci-style recursive programs which result in an exponential "explosion" of calls.

Example 3: Towers of Hanoi Problem



- The problem statement:
  - Initially all the disks are stacked on the LEFT pole.
  - Required to transfer all the disks to the RIGHT pole.
    - Only one disk can be moved at a time.
    - A larger disk cannot be placed on a smaller disk.

Recursion is implicit

- General problem of n disks.
  - Step 1:
    - Move the top (n-1) disks from LEFT to CENTER.
  - Step 2:
    - Move the largest disk from LEFT to RIGHT.
  - Step 3:
    - Move the (n-1) disks from CENTER to RIGHT.

Recursive C code: Towers of Hanoi

```
#include  <stdio.h>

void  transfer (int n, char from, char to, char temp);

int main()
{
     int  n;  /* Number of disks */
     scanf (“%d”, &n);
     transfer (n, ‘L’, ‘R’, ‘C’);
     return 0;
}

void  transfer (int n, char from, char to, char temp)
{
     if  (n > 0)  {
              transfer (n-1, from, temp,to);
              printf (“Move disk %d from %c to %c \n”, n, from, to);
              transfer (n-1, temp, to, from);
     }
     return;
}
```

Towers of Hanoi: Example Run

```
3
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

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

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

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

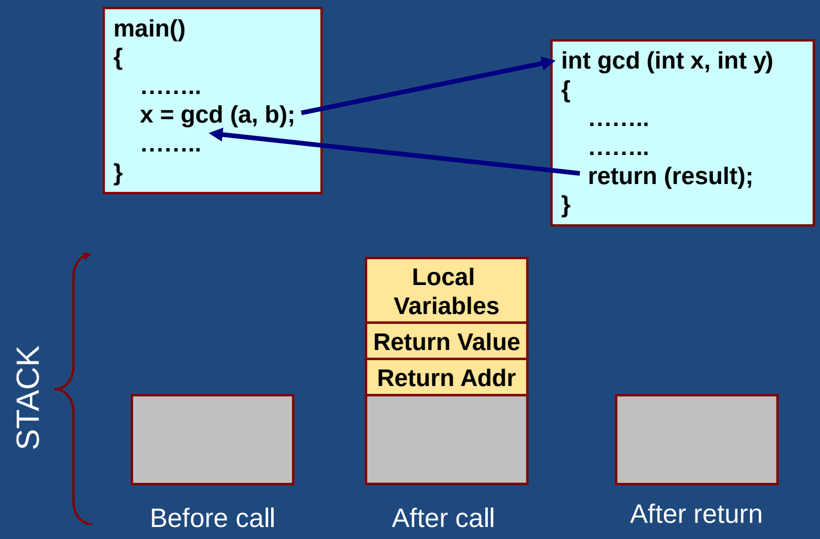
Performance Tip

- Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

How are function calls implemented?

- In general, during program execution
  - The system maintains a *stack* in memory.
    - *Stack* is a *last-in first-out* structure.
    - Two operations on stack, *push* and *pop*.
  - Whenever there is a function call, the *activation record* gets *pushed* into the stack.
    - Activation record consists of the *return address* in the calling program, the *return value* from the function, and the *local variables* inside the function.
    - At the end of function call, the corresponding *activation record* gets *popped* out of the stack.

At the system



```
main()
{
    ……..
    x = gcd (a, b);
    ……..
}
```

```
int gcd (int x, int y)
{
    ……..
    ……..
    return (result);
}
```

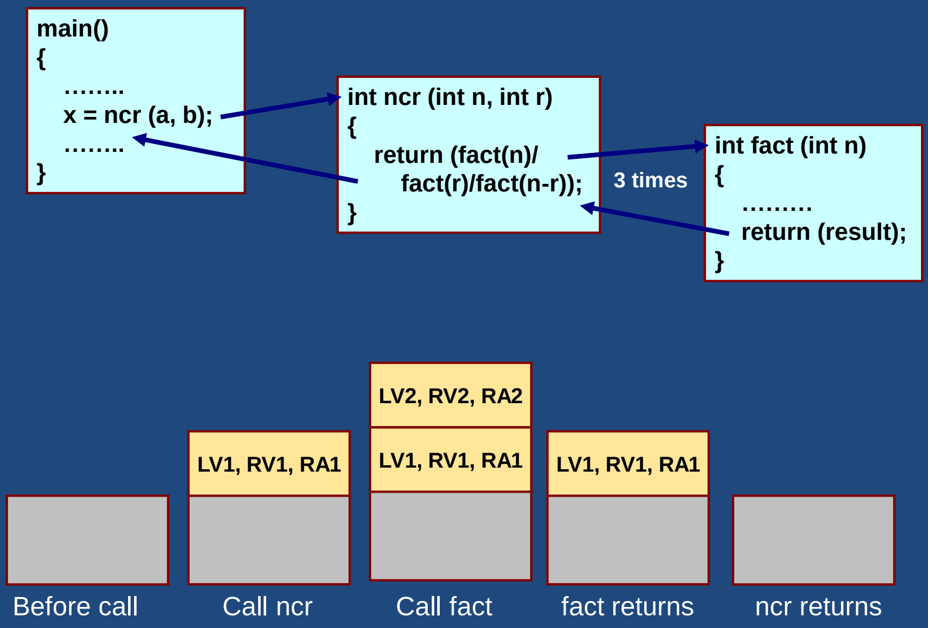
STACK

Local Variables
Return Value
Return Addr

Before call

After call

After return



```
main()
{
    ……..
    x = ncr (a, b);
    ……..
}
```

```
int ncr (int n, int r)
{
    return (fact(n)/
        fact(r)/fact(n-r));
}
```

3 times

```
int fact (int n)
{
    ………
    return (result);
}
```

LV2, RV2, RA2
LV1, RV1, RA1

Before call

Call ncr

Call fact

fact returns

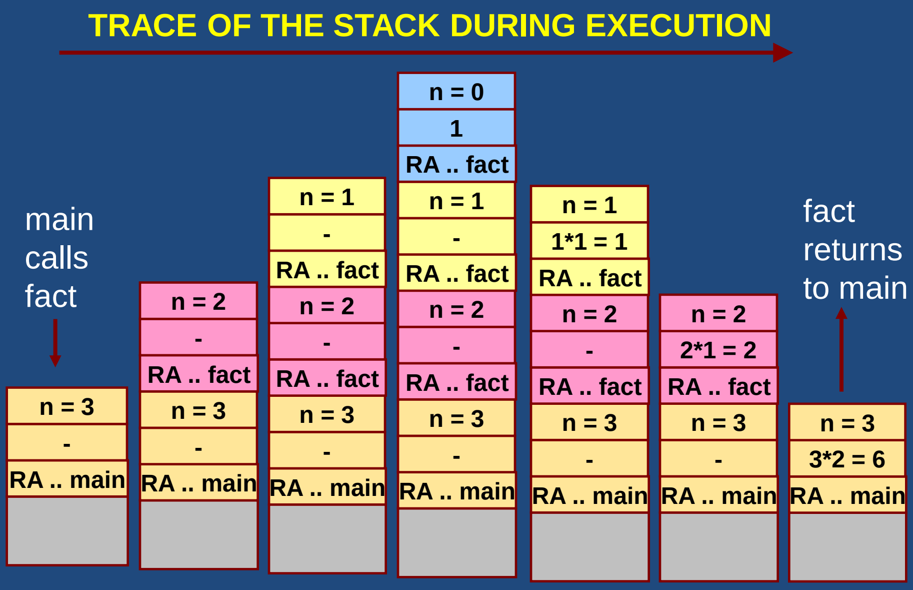
ncr returns

Example:: main() calls fact(3)

```
void main()
{
   int  n;
   n = 4;
   printf ("%d \n", fact(n) );
}
```

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

TRACE OF THE STACK DURING EXECUTION



main calls fact

n = 3
-
RA .. main

n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 1
-
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 0
1
RA .. fact
n = 1
-
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 1
1*1 = 1
RA .. fact
n = 2
-
RA .. fact
n = 3
-
RA .. main

n = 2
2*1 = 2
RA .. fact
n = 3
-
RA .. main

n = 3
3*2 = 6
RA .. main

fact returns to main

Homework

Trace of Execution for Fibonacci Series

Sorting: the basic problem

- Given an array

  x[0], x[1], ... , x[size-1]

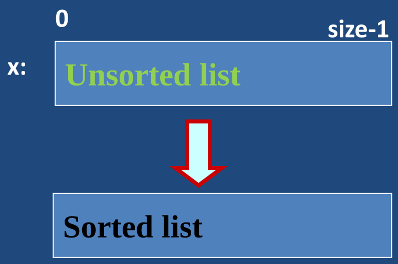
  reorder entries so that

  x[0] <= x[1] <= . . . <= x[size-1]

  - List is in non-decreasing order.
- We can also sort a list of elements in non-increasing order.

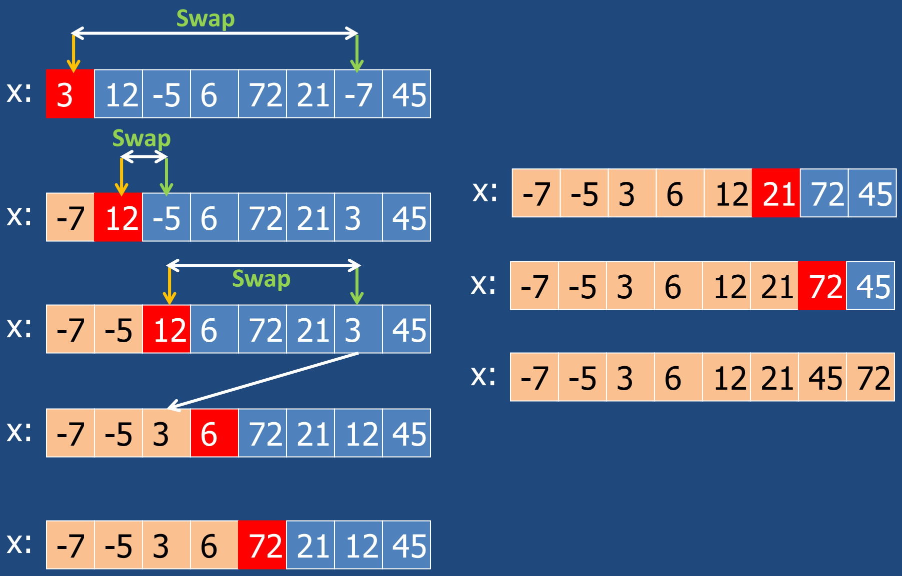
Sorting Problem

- What we want : Data sorted in order
- Input: A list of elements
- Output: A list of elements in sorted (non-increasing/non-decreasing) order



- Original list:
  - 10, 30, 20, 80, 70, 10, 60, 40, 70
- Sorted in non-decreasing order:
  - 10, 10, 20, 30, 40, 60, 70, 70, 80
- Sorted in non-increasing order:
  - 80, 70, 70, 60, 40, 30, 20, 10, 10

Example



x: 3 12 -5 6 72 21 -7 45

x: -7 12 -5 6 72 21 3 45

x: -7 -5 12 6 72 21 3 45

x: -7 -5 3 6 72 21 12 45

x: -7 -5 3 6 72 21 12 45

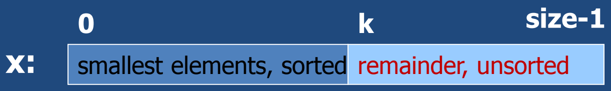
x: -7 -5 3 6 12 21 72 45

x: -7 -5 3 6 12 21 72 45

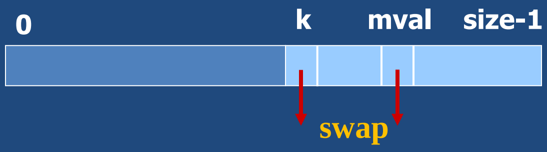
x: -7 -5 3 6 12 21 45 72

Selection Sort

- General situation :



- **Steps :**
  - **Find smallest element, mval, in x[k+1..size-1]**
  - **Swap smallest element with x[k-1],**
  - **Increase k.**



Subproblem: Find smallest element

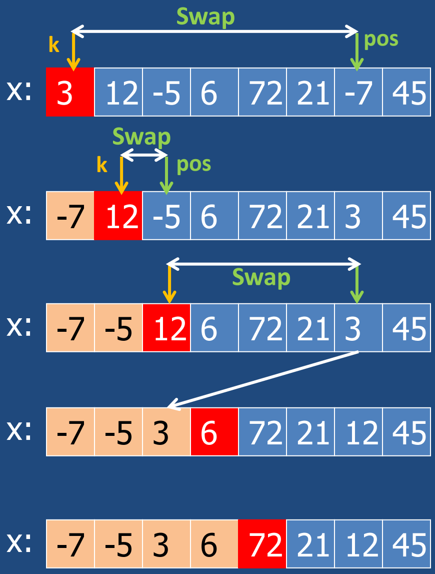
```
/* Yield location of smallest element in x[k .. size-1] and store in pos*/

int j, pos;
pos = k;            /* assume first element is the smallest element */
for (j=k+1; j<size; j++) {
          if (x[j] < x[pos]) {   /* x[pos] is the smallest element as of now */
                    pos = j;
          }
}
printf("%d",pos);
```

Subproblem: Swap with smallest element

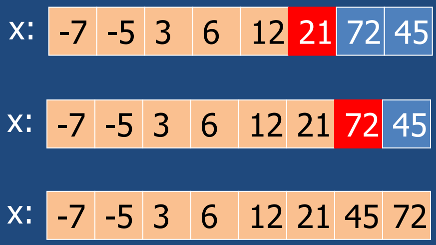
```
/* Yield location of smallest element in x[k .. size-1] and store in pos*/

int j, pos;
pos = k;              /* assume first element is the smallest element */
for (j=k+1; j<size; j++) {
          if (x[j] < x[pos]) {   /* x[pos] is the smallest element as of now */
                    pos = j;
          }
}
~~printf(“%d”,pos);~~
if (x[pos] < x[k]) {
    temp = x[k];                /* swap content of x[k] and x[pos] */
    x[k] = x[pos];
    x[pos] = temp;
}
```

```
#include <stdio.h>                         /* Sort x[0..size-1] in non-decreasing order */
int main()
{
    int k,j,pos,x[100],size,temp;
    printf("Enter the number of elements: ");
    scanf("%d",&size);
    printf("Enter the elements: ");
    for(k=0;k<size;k++)
        scanf("%d",&x[k]);
    for (k=0; k<size-1; k++)  {
        pos = k;                           /* assume first element is the smallest element */
        for (j=k+1; j<size; j++) {
            if (x[j] < x[pos])              /* x[pos] is the smallest element as of now */
                pos = j;
        }
        if (x[pos] < x[k]) {
            temp = x[k];                              /* swap content of x[k] and x[pos] */
            x[k] = x[pos];
            x[pos] = temp;
        }
    }
    for(k=0;k<size;k++)                    /* print the sorted (non-decreasing) list */
        printf("%d ",x[k]);
    return 0;
}
```

Example



```
for (k=0; k<size-1; k++)  {
        pos = k;
        for (j=k+1; j<size; j++) {
            if (x[j] < x[pos])
                pos = j;
        }
        temp = x[k];
        x[k] = x[pos];
        x[pos] = temp;
}
```



Analysis

How many steps are required?

Let us assume there are n elements (size=n).
Each statement executes in constant time.

To read

for loop will take of the **order of n** time

Sorting

When k=0, in worst case

(n-1) comparisons to find minimum.

When k=1, in worst case

(n-2) comparisons to find minimum.

…..

(n-1)+(n-2)+……+1= n(n-1)/2

To print

for loop will take of the **order of n** time

```
for(k=0;k<size;k++)
        scanf("%d",&x[k]);
for (k=0; k<size-1; k++)  {
        pos = k;
        for (j=k+1; j<size; j++) {
            if (x[j] < x[pos])
                pos = j;
        }
        temp = x[k];
        x[k] = x[pos];
        x[pos] = temp;
}
 for(k=0;k<size;k++)
        printf("%d ",x[k]);
```

Total time= 2 × order of n + order of n^2 = order of n^2

Analysis

How many steps are required?

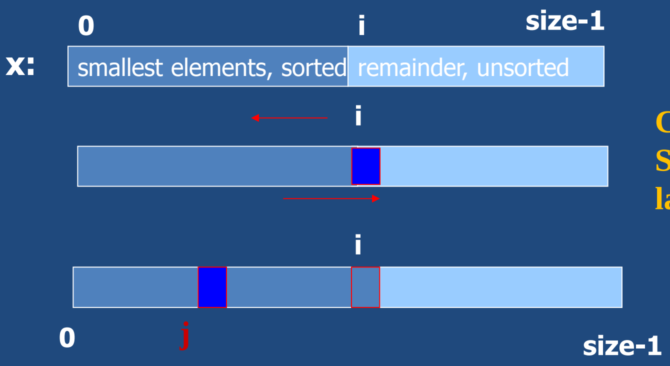
Let us assume there are n elements (size=n).
Each statement executes in constant time.

Best Case?
Worst Case?
Average Case?

```
for(k=0;k<size;k++)
        scanf("%d",&x[k]);
for (k=0; k<size-1; k++)  {
        pos = k;
        for (j=k+1; j<size; j++) {
            if (x[j] < x[pos])
                pos = j;
        }
        temp = x[k];
        x[k] = x[pos];
        x[pos] = temp;
}
 for(k=0;k<size;k++)
        printf("%d ",x[k]);
```

Insertion Sort

- General situation :



Compare and Shift till x[i] is larger.

Insertion Sorting

```
int list[100], size;
___________________;

for (i=1; i<size; i++) {
    item = list[i] ;
    for (j=i-1; (j>=0)&& (list[j] > item); j--)
            list[j+1] = list[j];
    list[j+1] = item ;
}
___________________;
```

Complete Insertion Sort

```
#include <stdio.h>                              /* Sort x[0..size-1] in non-decreasing order */
#define SIZE 100
int main()
{
    int i,j,x[SIZE],size,temp;
    printf("Enter the number of elements: ");
    scanf("%d",&size);
    printf("Enter the elements: ");
    for(i=0;i<size;i++)
        scanf("%d",&x[i]);
    for (i=1; i<size; i++) {
          temp = x[i] ;
          for (j=i-1; (j>=0)&& (x[j] > temp); j--)
                    x[j+1] = x[j];
          x[j+1] = temp ;
    }
    for(i=0;i<size;i++)           /* print the sorted (non-decreasing) list */
        printf("%d ",x[i]);
    return 0;
}
```

Insertion Sort Example

```
for (i=1; i<size; i++) {
        temp = x[i] ;
        for (j=i-1; (j>=0)&& (x[j] > temp); j--)
                x[j+1] = x[j];
        x[j+1] = temp ;
}
```

Input:

3 12 -5 6 72 21 -7 45

3 **12** -5 6 72 21 -7 45

-5 3 12 6 72 21 -7 45

-5 3 **6** 12 72 21 -7 45

-5 3 6 12 **72** 21 -7 45

-5 3 6 12 **21** 72 -7 45

-7 -5 3 6 12 21 72 45

-7 -5 3 6 12 21 **45** 72

Output:

-5 3 6 12 **21** 72 -7 45

-7

-5 3 6 12 21 72 __ 45

-5 3 6 12 21 __ 72 45

-5 3 6 12 __ 21 72 45

-5 3 6 __ 12 21 72 45

-5 3 __ 6 12 21 72 45

-5 __ 3 6 12 21 72 45

__ -5 3 6 12 21 72 45

-7 -5 3 6 12 21 72 45

Time Complexity

- Number of comparisons and shifting:
  - Worst Case?

    1+2+3+ …… +(n-1) = n(n-1)/2
  - Best Case?

    1+1+…… (n-1) times = (n-1)