# 6. Repetition

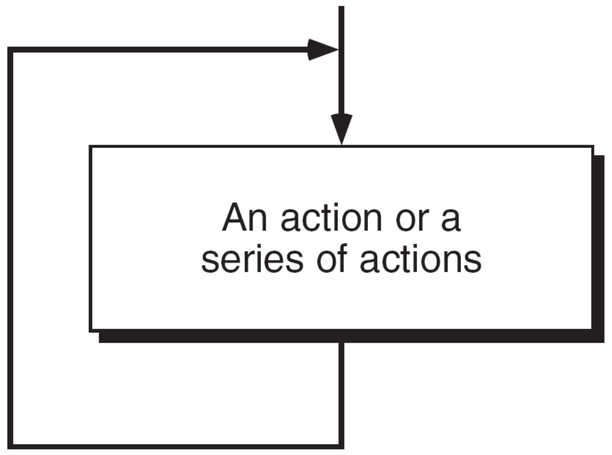
C Programming

## Agenda

- Concept of a loop
- Loops in C
- Loop Examples
- Other Statements Related to Looping
- Looping Applications
- Recursion

## Concept of Loop

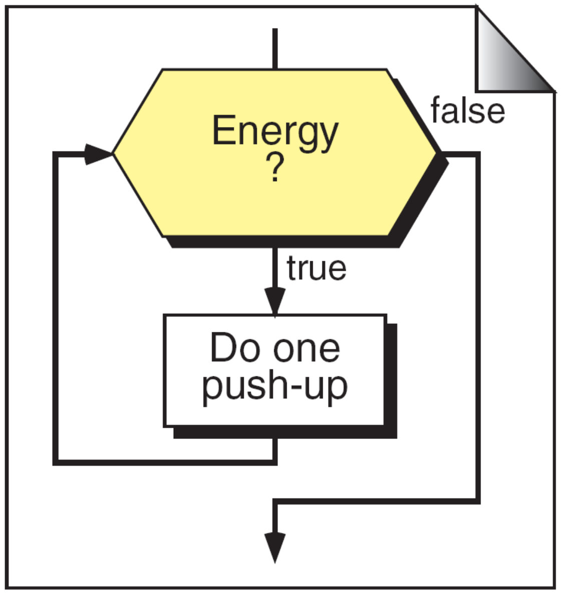
- Loop: repeating action over and over again
  - Repetition can be controlled by loop control expression



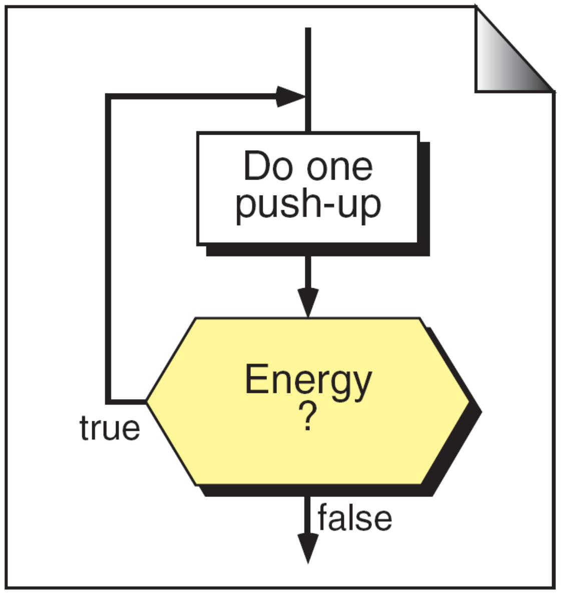
cf. Real power of computers is in their ability to repeat an operation or a series of operations many times

## Example

- Daily exercises – Push-up as many as possible
  - Test: Do I have enough energy?
  - Action: Push-up



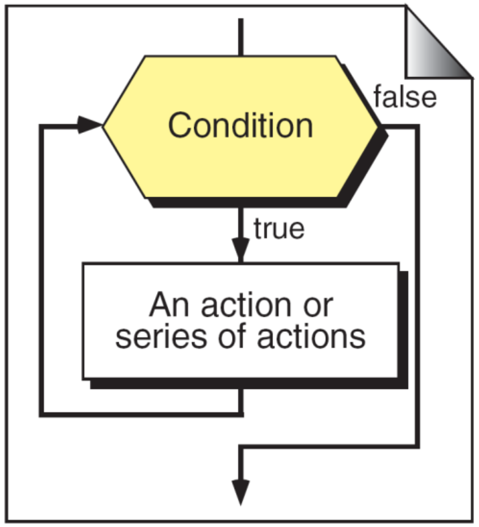
(a) Pretest Loop



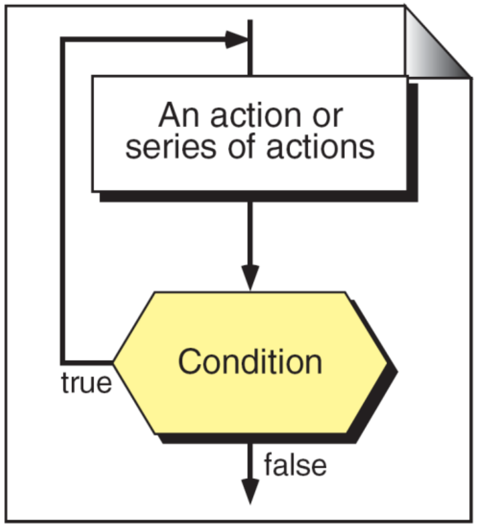
(b) Post-test Loop

## Pretest and Posttest Loops

- Loop control expression
  - Before or after each iteration, a condition is evaluated
    - If the condition is true, loop repeats one more time
    - Otherwise, loop terminates



(a) Pretest Loop



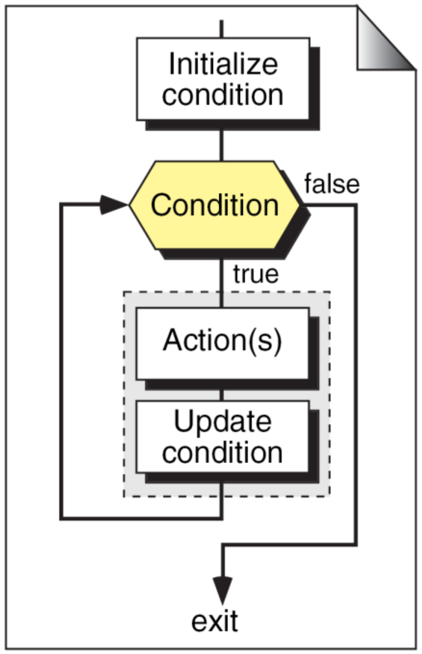
(b) Post-test Loop

## Event-Controlled Loops

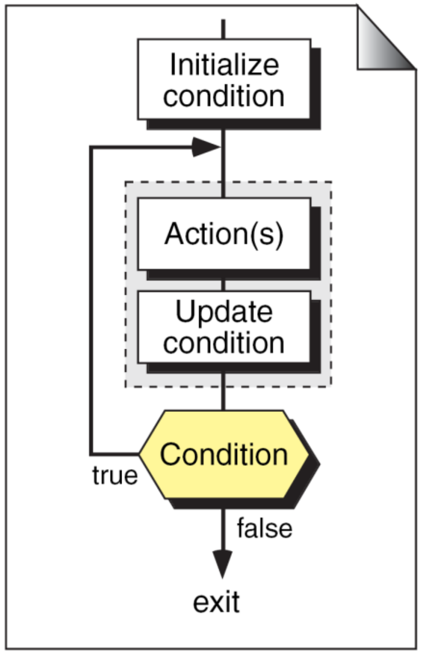
- Event-controlled loop: a loop, in which event changes the control expression from true to false

  Ex) Reading data: reaching the end

  Finding a piece of information: finding the target data



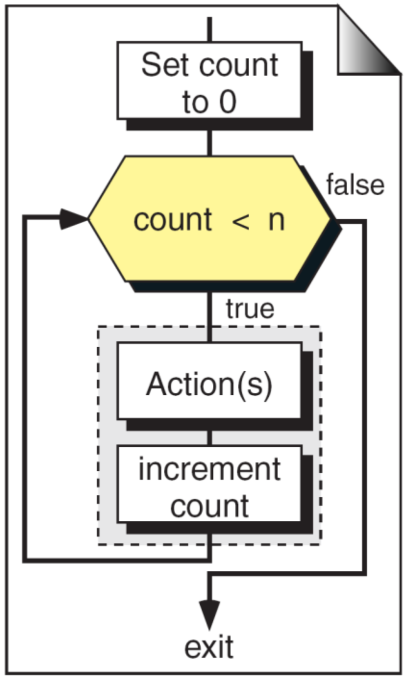
(a) Pretest Loop



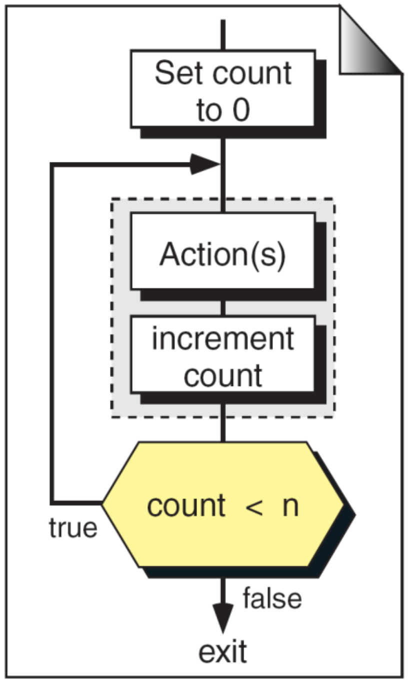
(b) Post-test Loop

# Counter-Controlled Loops

- Counter-controlled loop: a loop that is repeated a fixed number of times
  - Requires counter variable
  - Must be initialized, updated, tested



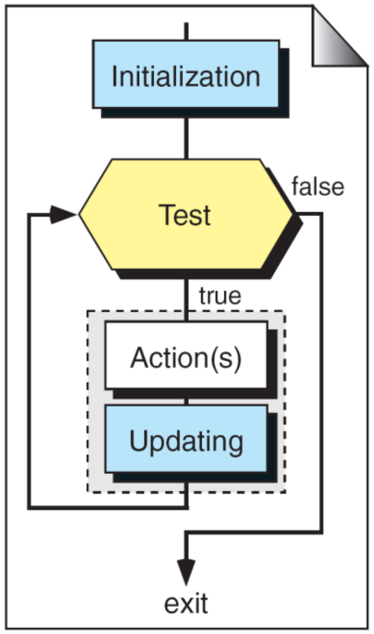
(a) Pretest Loop



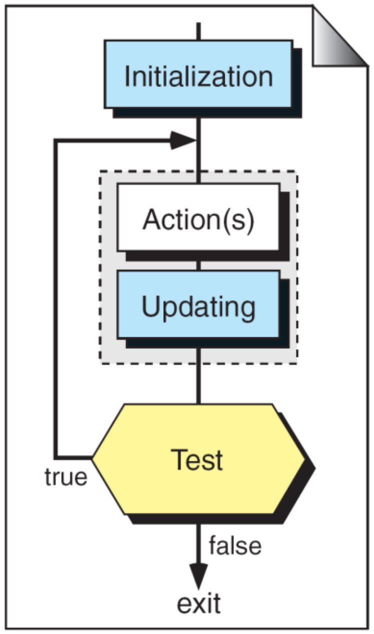
(b) Post-test Loop

# Initialization and Updating

- Initialization: preparation to start a loop
  Ex) setting counter by zero
- Update: change required to terminate loop properly
  Ex) losing energy to stop push-up
  increasing/decreasing counter



(a) Pretest Loop

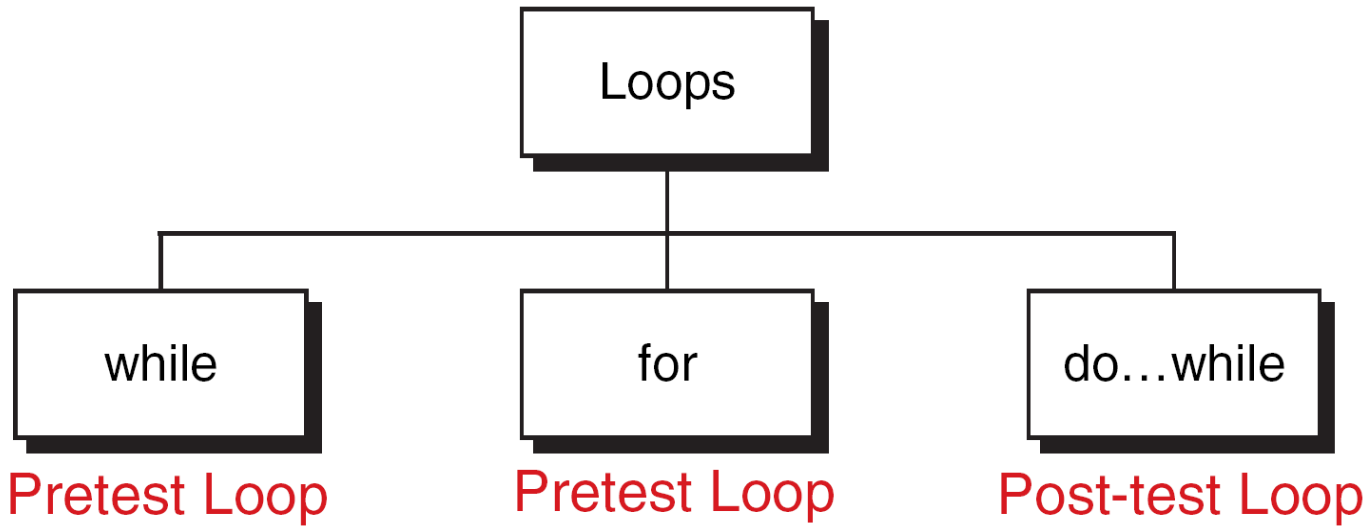


(b) Post-test Loop

## Agenda

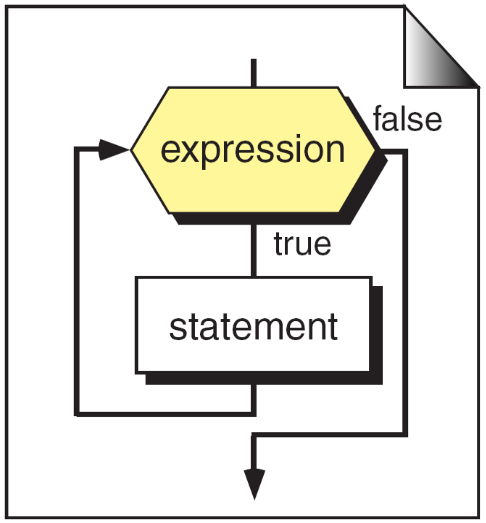
- Concept of a loop
- <u>Loops in C</u>
- Loop Examples
- Other Statements Related to Looping
- Looping Applications
- Recursion

## Loops in C

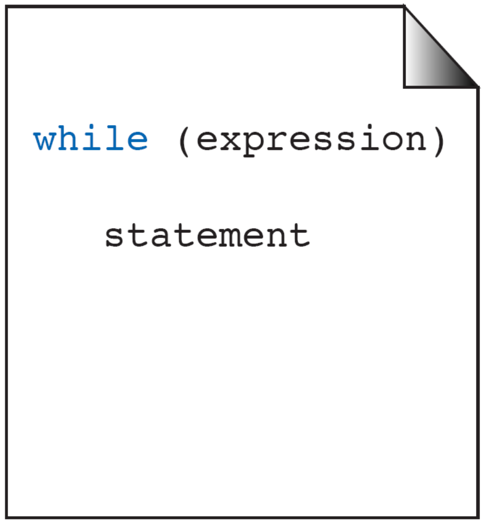

Loops
while
for
do…while
Pretest Loop
Pretest Loop
Post-test Loop

## while Loop

- *while* loop: a pretest loop
  - Condition expression is tested before every iteration



(a) Flowchart



```
while (expression)

    statement
```

(b) Sample Code

## Example

- Process control loop

```
while (true)
   {
    temp = getTemperature();
    if (temp < 68)
        turnOnHeater();
    else if (temp > 78)
        turnOnAirCond();
    else
       {
        turnOffHeater();
        turnOffAirCond();
       } // else
   } // while true
```

## Example

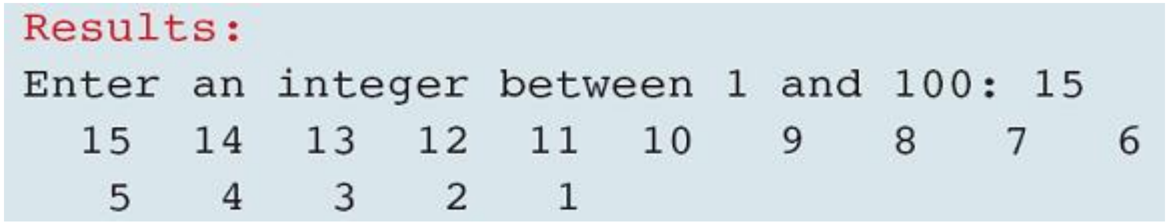
- factorial example

```
int num, fact=1, i=1;

printf (“Enter an integer between 1 and 100: “);
scanf (“%d”, &fact);          // Initialization

while (i <= num){
   fact = fact * i;
   i++;
}

printf(“% factorial is %d ₩n”, num, fact);
```

## Example

- Printing a series of numbers in descending order

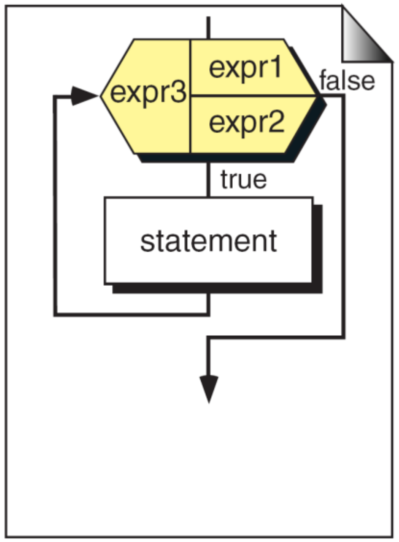
```
#include <stdio.h>

int main (void)
{
   int num;
   int lineCount = 0;

   printf ("Enter an integer between 1 and 100: ");
   scanf  ("%d", &num);          // Initialization

   if (num > 100)        // Test number
      num = 100;

   while (num > 0){
      if (lineCount < 10)
         lineCount++;
      else {
         printf("₩n");
         lineCount = 1;
      } // else
      printf("%4d", num--);        // num-- updates loop
   } // while

   return 0;
}              // main
```



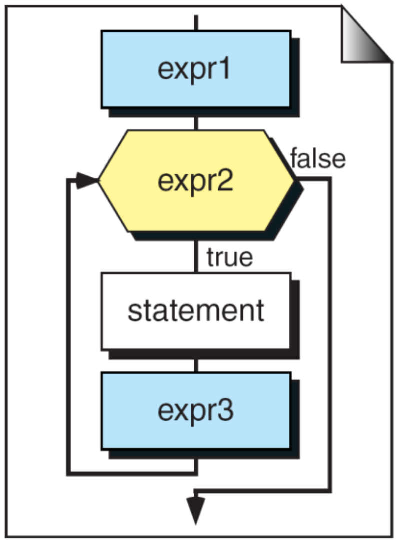
```
Results:
Enter an integer between 1 and 100: 15
   15   14   13   12   11   10    9    8    7    6
    5    4    3    2    1
```

# for Loop

- *for* loop: a pretest loop with three expressions
  - Initialization statements
  - Limit-test expression
  - Updating expression



(a) Flowchart



(b) Expanded Flowchart

```
for (expr1; expr2; expr3)
     statement
```

## for Loop

- A *for* loop is used when a loop is to be executed a known number of times.
  - We can do the same thing with a while loop, but the for loop is easier to read and more natural for counting loops.

Ex) printing '#' mark n times

```
int i = 0;
for(i = 0; i < n; i++)
   printf("#");
printf("₩n");
```

## Example

- Print number series

```
#include <stdio.h>
int main (void)
{
   int limit = 0;
   int i = 0;

   printf ("₩nPlease enter the limit: ");
   scanf ("%d", &limit);
   for (i = 1; i <= limit; i++)
      printf("₩t%d₩n", i);

   return 0;
}             // main
```

## for Loop

- Repeating actions n iterations
  - for(i = 0; i < n; i++)  -> most frequently used
    - 0, 1, 2, … n – 1
  - for(i = 1; i <= n; i++)
    - 1, 2, 3, … n

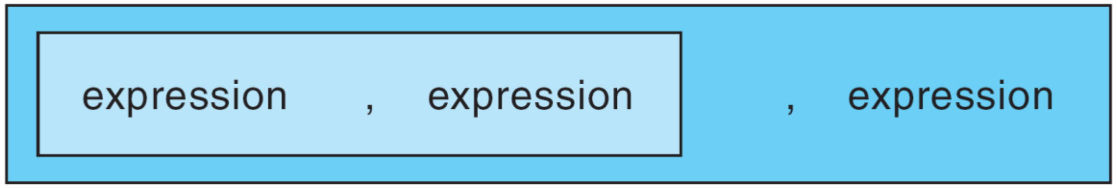
## Nested for Loop

- More than one *for* loops can be nested

```
Ex) for(i = 1; i <= 9; i++){
        for(j = 1; j <= 9; j++)
            printf("%2d * %2d = %2d₩n", i, j, i * j);
        printf("₩n");
    }
```

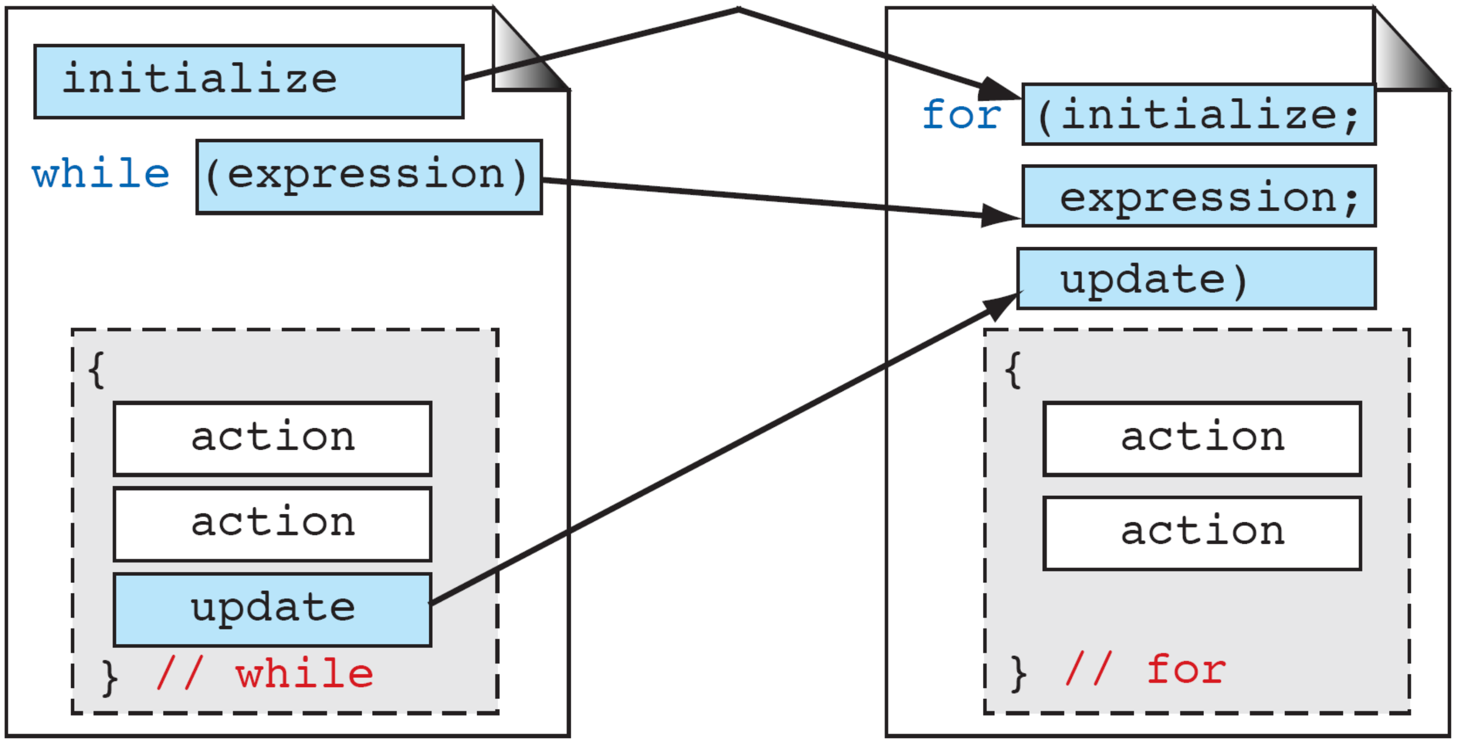
## Comma Expression

- Comma expression: complex expression made up of two (or more) expressions separated by a comma
  - Most often used in *for* statements



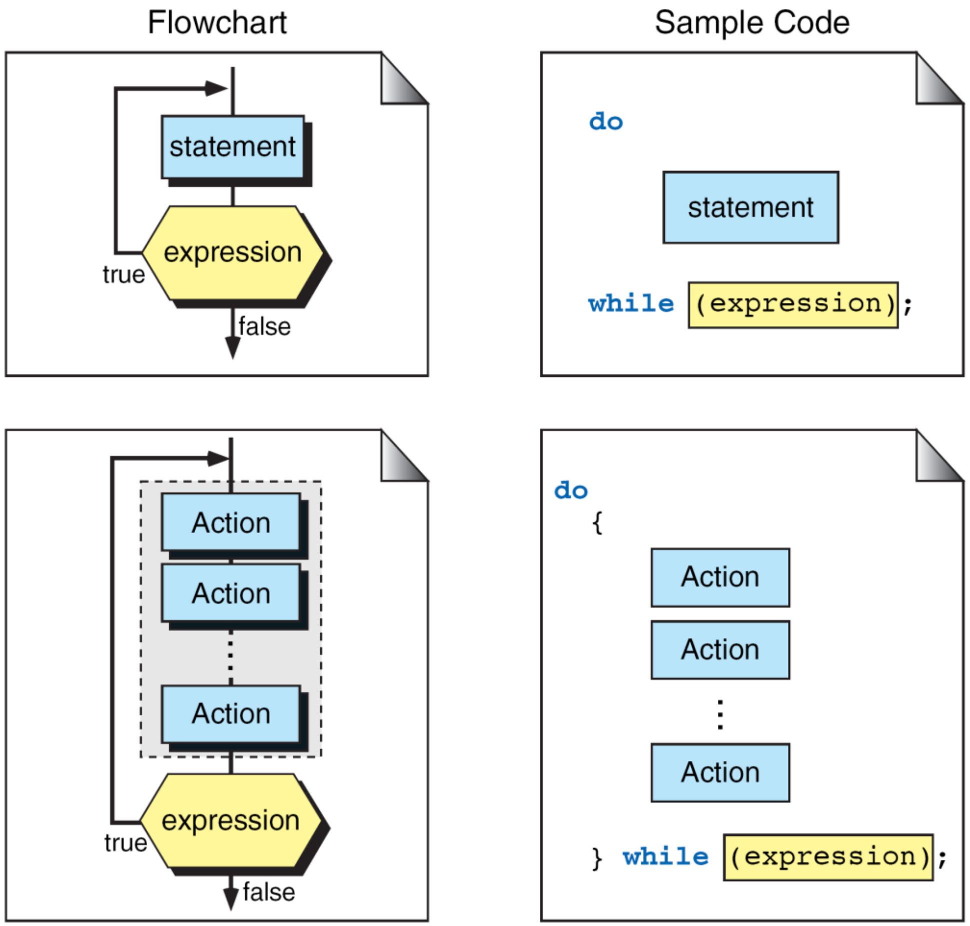
```
Ex) for(i = 0, j = 0; i < 10; i++, j+=2)
        printf("%d – %d₩n", i, j);
```

## Comparing while/for Loops



## do … while Loop

- *do … while* loop: a posttest loop



```
do

        statement

while (expression);
```

```
do
  {
        Action
        Action
          ⋮
        Action
  } while (expression);
```

## Example

- Add a list of integers from keyboard

  Ex) Enter your numbers: <EOF> to stop.

  10 15 20 25 <CTRL-d> or <CTRL-z>

  Total: 70

```
#include <stdio.h>

int main()
{
   int sum = 0, x;
   int testEOF = 0;

   printf(“Enter your numbers <EOF> to stop.₩n”);
   do {
      testEOF = scanf(“%d”, &x);
      if(testEOF != EOF)
         sum += x;
   } while(testEOF != EOF);
   printf(”Total: %d₩n”, sum);
   return 0;
}
```

## Pretest vs. Posttest

```
#include <stdio.h>

int main()
{
    int n = 0, i = 0;

    printf("This is a pretest loop₩n");
    printf("Enter a number : ");
    scanf("%d", &n);

    while(i < n){
        printf("i = %d₩n", i);
        i++;
    }

    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int n = 0, i = 0;

    printf("This is a posttest loop₩n");
    printf("Enter a number : ");
    scanf("%d", &n);

    do {
        printf("i = %d₩n", i);
        i++;
    } while(i < n);

    return 0;
}
```

## Agenda

- Concept of a loop
- Loops in C
- <u>Loop Examples</u>
- Other Statements Related to Looping
- Looping Applications
- Recursion

## Loop Examples

- Left triangle, whose width is a given integer *x*

```
1
12
123
1234
12345
```

- Right triangle, whose width is a given integer *x*

```
*****
 ****
  ***
   **
    *
```

## Loop Examples

- Print calendar of a month
  - void printMonth(int startDay, int days);

Ex) printMonth(2, 29)

Result:

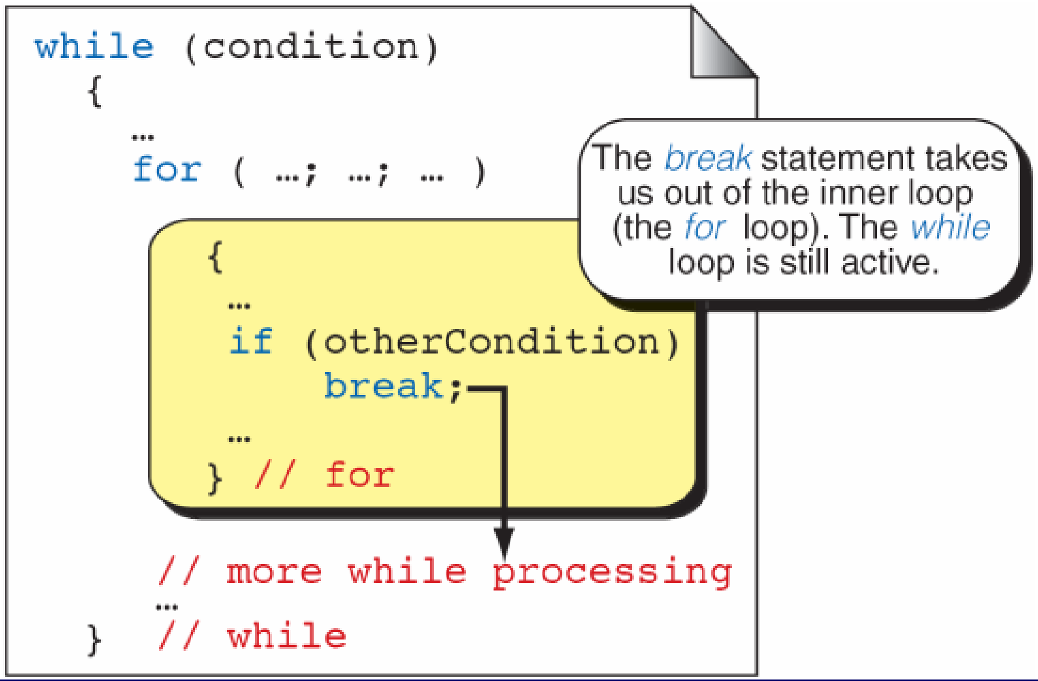
```
Sun Mon Tue Wed Thu Fri Sat
--- --- --- --- --- --- ---
          1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29
--- --- --- --- --- --- ---
```

## Agenda

- Concept of a loop
- Loops in C
- Loop Examples
- <u>Other Statements Related to Looping</u>
- Looping Applications
- Recursion

## break Statement

- *break* statement: terminates loop
    - In nested loop, *break* terminates only one inner loop
    - Major use: specifying alternative termination condition



```
while (condition)
  {
    …
    for ( …; …; … )
      {
        …
        if (otherCondition)
             break;
        …
      } // for

      // more while processing
      …
  }   // while
```

The *break* statement takes us out of the inner loop (the *for* loop). The *while* loop is still active.

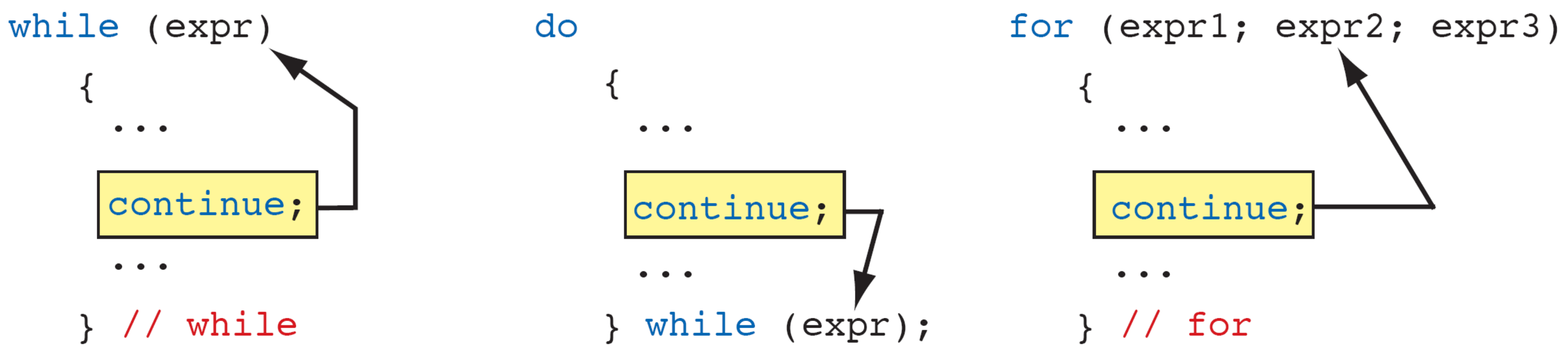
## break Statement

```
#include <stdio.h>

int main()
{
   int sum = 0;
   int testEOF = 0;

   printf("Enter numbers <EOF> to
     stop.₩n");
   do {
      testEOF = scanf("%d", &x);
      if(testEOF != EOF)
         sum += x;
   } while(testEOF != EOF);
   printf("Total: %d₩n", sum);
   return 0;
}
```

```
#include <stdio.h>

int main()
{
   int sum = 0;
   int testEOF = 0;

   printf("Enter numbers <EOF> to
     stop.₩n");
   do {
      testEOF = scanf("%d", &x);
      if(testEOF == EOF)
              break;
      sum += x;
   } while(testEOF != EOF);
   printf("Total: %d₩n", sum);
   return 0;
}
```

## continue Statement

- *continue* statement: not terminating the loop, transfers to the testing expression



```
while (expr)
    {
      ...
     continue;
      ...
    } // while
```

```
do
    {
      ...
     continue;
      ...
    } while (expr);
```

```
for (expr1; expr2; expr3)
    {
      ...
     continue;
      ...
    } // for
```

## Example

- Get average of a integer list from standard input

```c
float readAverage (void)
{
// Local Declarations
int    count = 0;

int    n;
float sum = 0;

// Statements
while(scanf("%d",&n)
        != EOF)
    {
     if (n == 0)
        continue;
     sum += n;
     count++;
    }   // while

return (sum / count);
} // readAverage
```

```c
float readAverage (void)
{
// Local Declarations
int    count = 0;

int    n;
float sum = 0;

// Statements
while(scanf("%d",&n)
        != EOF)
    {
     if (n != 0)
        {
         sum += n;
         count++;
        } // if
    } // while
return (sum / count);
} // readAverage
```

## Recommendation

- Don’t use *break* and *continue* too often
  - It’s not desirable to make multiple condition check for a loop
  - Use of break and continue can be minimized by proper use of if-statement

## Find the Largest and Smallest

- Write a program minmax.c that read a series of integers and prints the minimum and maximum.

Ex) minmax.exe

```
Enter a series of numbers (<EOF> to stop)
100
200
-500
1000
^Z                              // <EOF> on Windows
min = -500
max = 1000
```

## Agenda

- Concept of a loop
- Loops in C
- Loop Examples
- Other Statements Related to Looping
- Looping Applications
- <u>Recursion</u>

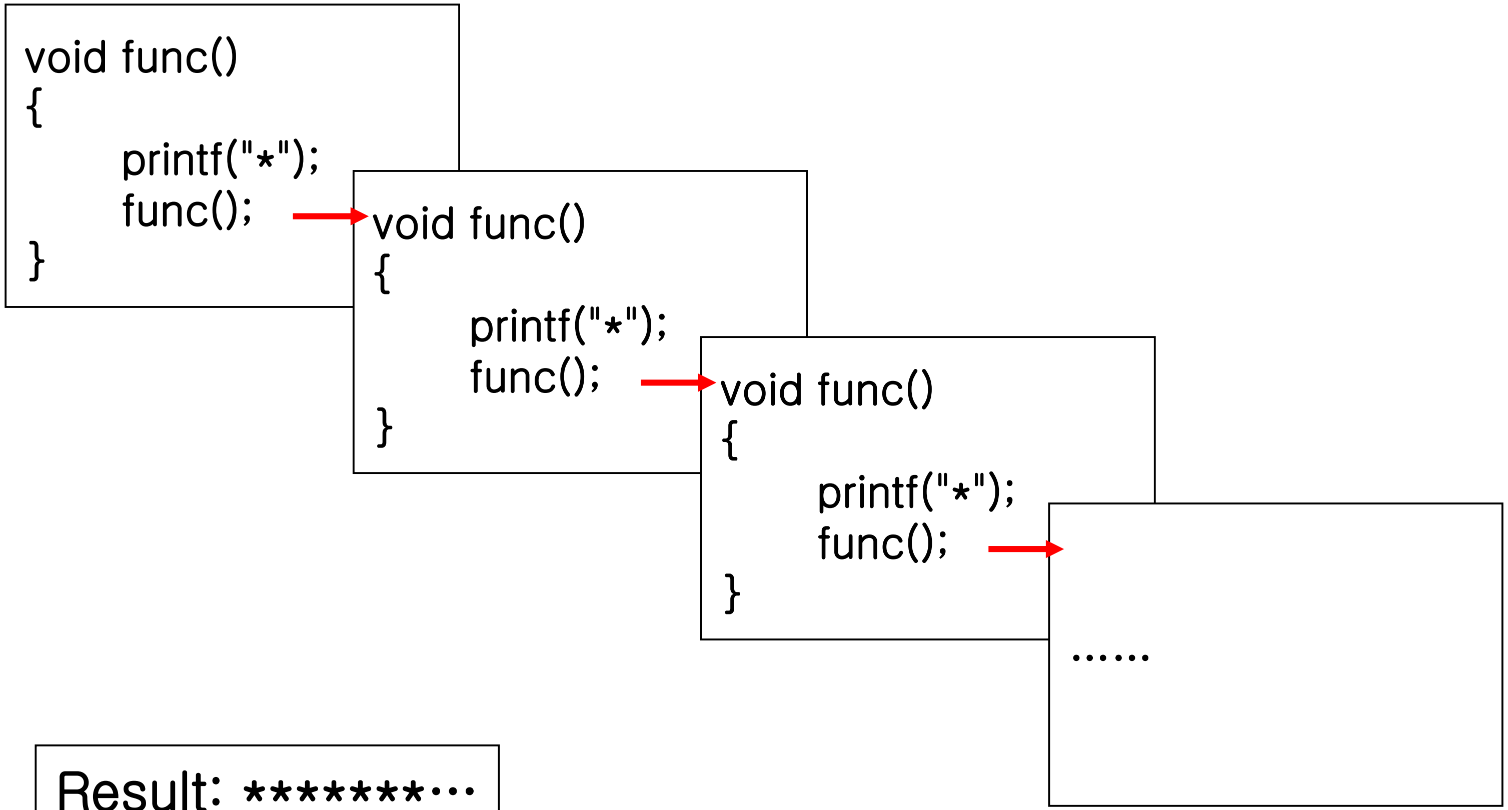
## Recursion

- Recursion: function call to itself
  - Action is repeated by calling itself
  - Termination condition is implemented using function arguments

- Recursion is an alternative repetition
  - All iterative algorithms (loops) can be transformed into recursive algorithm and vice versa.
  - However, one can be much simpler than the other

## Example

- A stupid recursive function



```
void func()
{
        printf("*");
        func();
}
```

```
void func()
{
        printf("*");
        func();
}
```

```
void func()
{
        printf("*");
        func();
}
```

```
……
```

Result: *******…

## Example: Factorial

- Iterative definition of factorial
  - factorial(n) = 1 // if n == 0
  - factorial(n) = n*(n-1)*(n-2)*⋯*2*1 // if n > 0

- Iterative solution

```
long factorial(int n)
{
   long facN = 1;
   int i = 0;

   for(i = 1; i <= n; i++)
      factN *= i;

   return factN;
}
```

## Example: Factorial

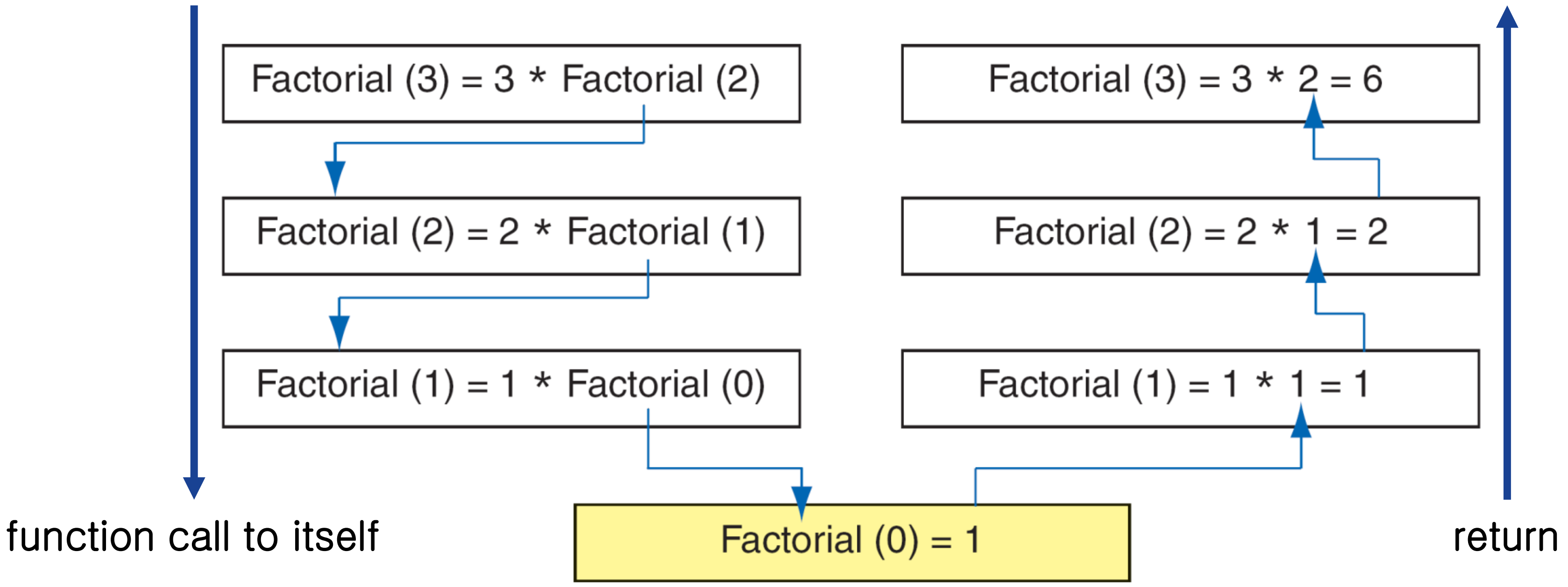
- Recursive definition of factorial
  - factorial(n) = 1                                    // if n == 0
  - factorial(n) = n * factorial(n-1)        // if n > 0

- Recursive solution

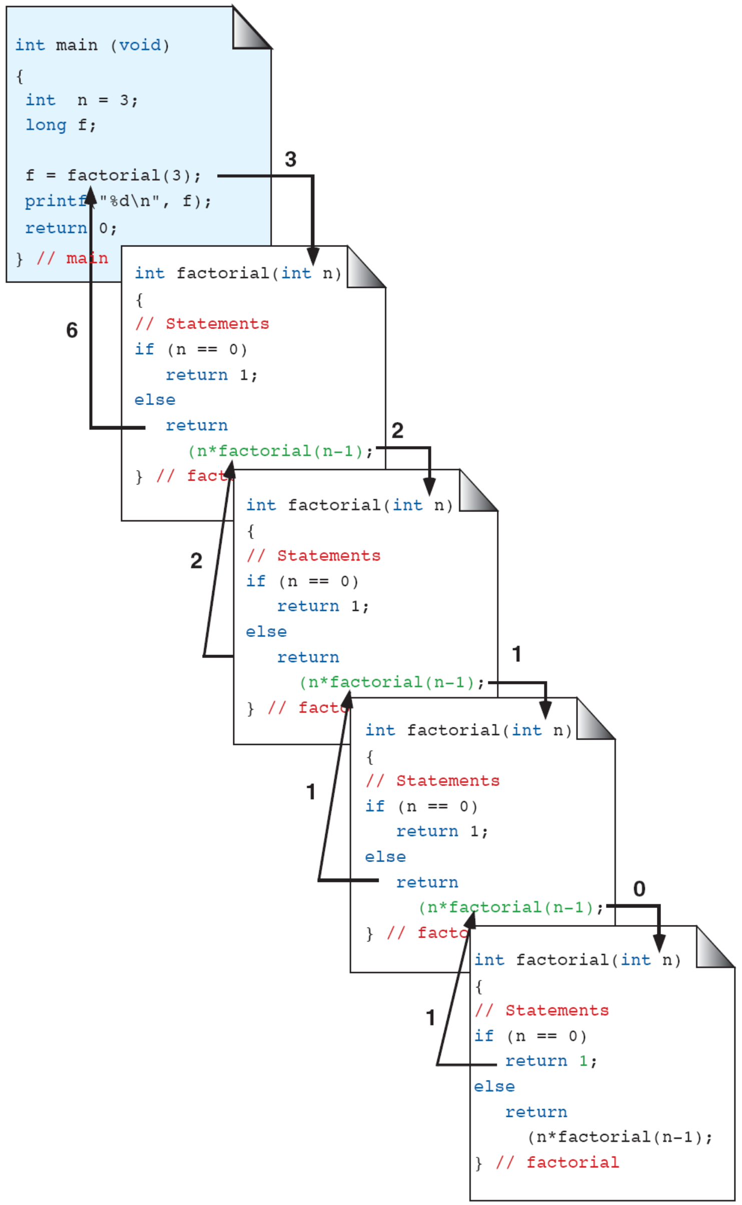
```
long factorial(int n)
{
   if(n == 0)
      return 1;
   else
      return n * factorial(n-1);    // Recursion
}
```

## Example: Factorial

- Recursion for factorial(3)


Factorial (3) = 3 * Factorial (2)
Factorial (2) = 2 * Factorial (1)
Factorial (1) = 1 * Factorial (0)
Factorial (0) = 1
Factorial (1) = 1 * 1 = 1
Factorial (2) = 2 * 1 = 2
Factorial (3) = 3 * 2 = 6
function call to itself
return

# Calling a Recursive Function



```
int main (void)
{
 int  n = 3;
 long f;

 f = factorial(3);
 printf("%d\n", f);
 return 0;
} // main
```

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

## Designing Recursive Function

- Two elements of recursive function
  - Every recursive call must either solve part of the problem or reduce the size of the problem (general case)
  - Recursive function should have a non-recursive solution for (base case)

```
long factorial(int n)
{
   if(n == 0)
      return 1;                         // base case
   else
      return n * factorial(n-1);    // Recursion
}
```

Function call argument was reduced

# Example: Fibonacci Numbers

- Fibonacci series
  - $\text{Fibonacci}_0 = 0$, $\text{Fibonacci}_1 = 1$
  - $\text{Fibonacci}_n = \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2}$

  Ex) 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

- Recursive solution

```
long Fibonacci(int n)
{
   if(n == 0)
      return 0;                                  // base case
   else if(n == 1)
      return 1;                                  // base case
   else
      return Fibonacci(n-1)+Fibonacci(n-2);      // general case
}
```