Programming Languages -1 (Introduction to C)

functions

Instructor: M.Fatih AMASYALI

E-mail:mfatih@ce.yildiz.edu.tr

A little bit about Functions

- Should perform a well-defined task
- Why?
- Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.
- Code can be re-used, not just within one program but in others.
- Recursion easier to do

More on functions

- Function definition format

```
return-value-type function-name( parameter-list )
  {
     declarations and statements
  }
```

  - e.g. *int factorial(int n)*
  - e.g. *void execute_loop(char c, float f)*
- Call as:
  - *i=factorial(3);*
  - *execute_loop(townInitial, distance);*
- Defining a function inside another function is a syntax error.
- Functions with void type return can not be used in expressions. T=f( ); if (f( )<3) are not valid.

Example: A simple function

```
#include <stdio.h>
#include <conio.h>

int max( int a, int b );

int main()
{
  int i = 8, j = 17;
  printf( "Maximum of %d and %d is %d\n", i, j, max(i, j));
  getch();
  return(0);
}

int max( int a, int b )
{
  if( a > b ) return a;
  else return b;
}
```

Functions with input parameters

- Parameters are by default input parameters and passed by value.

```
Output:
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

```c
#include <stdio.h>

void calculate_triangular_number(int n)
{
   int i, triangular_number = 0;

   for (i = 1; i <= n; i++)
      triangular_number+=i;

   printf("Triangular number %d is %d\n", n, triangular_number);
}

int main(void)
{
   calculate_triangular_number(10);
   calculate_triangular_number(20);
   calculate_triangular_number(50);
   return 0;
}
```

Calculating a triangular number, that is summing integers 1 to n.

Functions can return a *single* result

- Which one is better? The previous one or this one?

```
#include <stdio.h>

int calculate_triangular_number(int n)
{
   int i, triangular_number = 0;

   for (i = 1; i <= n; i++)
      triangular_number+=i;

   return triangular_number;
}

int main(void)
{
   printf(“Triangular number %d is %d\n”, 10, calculate_triangular_number(10));
   printf(“Triangular number %d is %d\n”, 20, calculate_triangular_number(20));
   printf(“Triangular number %d is %d\n”, 50, calculate_triangular_number(50));
   return 0;
}
```

```
Output:
Triangular number 10 is 55
Triangular number 20 is 210
Triangular number 50 is 1275
```

Example 2 (Call by value)

```
#include <stdio.h>

void printDouble( int x )
{
   printf(“Double of %d”, x);
   x *= 2;
   printf(“is %d\n”, x);
}

int main()
{
   int i = 13;

   printDouble(i);
   printf(“i=%d\n”, i);
}
```

- What does it print?
- A parameter of the function can be a constant, expression, variable etc. (anything that has a value!)
- Only the value is passed (not variable!)

Example: badswap.c

```
/* Swap the values of two
   variables. */

void badSwap ( int a, int b )
{
  int temp;

  temp = a;
  a = b;
  b = temp;

  printf("%d %d\n", a, b);
}
```

```
int main(void)
{
  int a = 3, b = 5;

  printf("%d %d\n",a,b);
  badSwap ( a, b );
  printf("%d %d\n",a,b);

  return 0;
}
```

Output:

```
3  5
5  3
3  5
```

Calling Functions: Call by Value

```
int addone(int x)
{
  return ++x;
}

int main ()
      {
            int i = 5, j;
            j=addone(i);
            printf(“%d %d\n”,i, j);  5    6
      }
```

What if
return ++x changes to
return x++

Calling Functions: Call by Reference (Passing Address)

```
int addone(int *x)
{
 return ++(*x);
}

int main ()
    {
        int i = 5, j;
        j=addone(&i);
        printf(“%d %d\n”,i, j); 6    6
    }
```

Function OverLoading

```
int byap (int a,int b)
{
    return a+b;
}
int byap(int a)
{
    return 2*a;
}
int main(int argc, char*argv[])
{
    printf("%d\n",byap(1,2));
    printf("%d\n",byap(3));
    getch();
    return 0;
}
```

Prototyping of Functions

- Must declare functions before use (like variables)
- Declaration is called a “prototype”
- Specifies the name, parameters and return type of the function, but not the code

Example: isNegative.c

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");
   }

   return 0;
}
```

```
int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return result;
}
```

Example: isNegative.c

Function Prototype,
don’t forget semicolon

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");
   }

   return 0;
}
```

```
int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return result;
}
```

Example: isNegative.c

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
      printf("Negative\n"
   }
   else
   {
                                   ;
   }

   return 0;
}
```

Function Definition

```
int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
      result=1;
   }
   else
   {
      result = 0;
   }
   return result;
}
```

Example: isNegative.c

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {


   }

   ret
}
```

```
int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return result;
}
```

Function Call
(*Must* be after prototype, but *can* be before definition)

Example: isNegative.c

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
      printf("Negative\n");
   }
   else
   {
      printf("Positive\n");
   }

   return 0;
}
```

Header files (filename**.h**) contain function prototypes and global variable declarations

```
                          )



      if ( n<0 )
      {
         result=1;
      }
      else
      {
         result = 0;
      }
      return result;
   }
```

Example: isNegative.c

```
#include <stdio.h>

int isNegative (int);

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");
   }

   return 0;
}
```

stdio.h contains function prototypes for **`printf(),`** **`scanf(),`** and other I/O functions

```
                                )

    if ( n<0 )
    {
        result=1;
    }
    else
    {
        result = 0;
    }
    return result;
  }
```

Header files

- You can make your own header files with prototypes of frequently used functions:

  ```
  #include "myFunctions.h"
  ```

- Put the functions in a corresponding C file, and include those too:

  ```
  #include "myFunctions.c"
  ```

Example: isNegative.c

```
#include <stdio.h>
#include "myFunctions.h"
#include "myFunctions.c"

int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");
   }

   return 0;
}
```

Note:

- " " around file name for user-defined files
- < > for standard system files

`isNegative()` is declared in **`myFunctions.h`** and defined in **`myFunctions.c`**, *not* in this file!

Example: myFunctions.c

```
int
isNegative ( int n )
{
  int result;
  if ( n<0 )
  {
      result=1;
  }
  else
  {
      result = 0;
  }
  return result;
}
```

Example: myFunctions.h

```
int isNegative ( int );
```

Scope: Local Variables

- Variables declared in a function body: only accessible whilst function executing
- In fact, this is true of every block in a program

Example: isNegative.c

```
#include <stdio.h>

int
isNegative ( int n )
{
  int result;
  if (number<0)
  {
      result=1;
  }
  else
  {
      result = 0;
  }
  return result;
}
```

```
int main (void)
{
  int number;

  printf ("Enter an integer: ");
  scanf ("%d",&number);

  if (isNegative(number))
  {
      printf("Negative\n");
  }
  else
  {
      printf("Positive\n");
  }

  return 0;
}
```

Example: isNegative.c

```
#include <stdio.h>

int
isNegative ( int n )
{
   int result;
   if (number<0)
   {
       resu t=1;
   }
   else
   {
       r      = 0;
   }
```

```
int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");

     rn 0;
```

ERROR! Number is local to the **`main`** function, not accessible here

Example: isNegative.c

```
#include <stdio.h>

int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return resu
}
```

```
int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else

           'Positive\n");


   return 0;
}
```

Use the parameter ***n*** which is local to the function **`isNegative()`**

Example: isNegative.c

```c
#include <stdio.h>

int
isNegative ( int n )
{
   int result;
   if ( n<0 )
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return result;
}
```

```c
int main (void)
{
   int number;

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative(number))
   {
       printf("Negative\n");
   }
   else
   {
       printf("Positive\n");
   }
```

result & n: local to ***isNegative()***

number: local to ***main()***

Scope: Global Variables

- Global variables are accessible in any function **after** their declaration to the end of that source file
- They're useful, but risky
  - if any and every function can modify them, it can be difficult to keep track of their value
- Better to use local variables and parameter passing if possible

Example: isNegativeGlobal.c

```
#include <stdio.h>

int number;

int
isNegative ( void )
{
   int result;
   if ( number <0 )
   {
        result=1;
   }
   else
   {
        result = 0;
   }
   return result;
}
```

```
int main (void)
{

   printf ("Enter an integer: ");
   scanf ("%d",&number);

   if (isNegative())
   {
        printf("Negative\n");
   }
   else
   {
        printf("Positive\n");
   }

   return 0;
}
```

Example: isNegativeGlobal.c

```
#include <stdio.h>

int number;

int
isNegative ( vo
{
   int result;
   if ( number <0
   {
       result=1;
   }
   else
   {
       result = 0;
   }
   return result;
}
```

```
int main (void)
{

                 eger: ");





                 ");

   else
   {
       printf("Positive\n");
   }

   return 0;
}
```

number is now GLOBAL - declared *outside* any function, accessible in all functions (*after* the declaration)

Scope: Functions

- Functions are also accessible in any function **after** their declaration to the end of that source file

Recursive functions

- Functions that call themselves either directly or indirectly (through another function) is called a *recursive function*.
- If a function is called with a simple case, the function actually knows how to solve the simplest case(s) - “base case” and simply returns a result.
- If a function is called with a complex case - “recursive case” , it invokes itself again with simpler actual parameters.
- Always specify the base case; otherwise, indefinite recursive will occur and cause “stack-overflow” error.

Recursive functions

1. must resembles original problem but slightly simplified
2. function will call itself (recursion step or recursive call) to solve slightly simplified problem
3. process continues until the last recursive call is with the base case
4. that call returns the result of the base case
5. return to previous calling functions
6. finally reaches main.c.

Recursion vs. Iteration

```
int main(int argc, char* argv[])
{
int N,top=0;
scanf("%d",&N);
for (int i=1 ;i<=N;i++)
 top+=i;
printf("top=%d",top);
return 0;
}
```

```
int top(int x)
{
        if (x==1) return 1;
        else return x+top(x-1);
}
int main(int argc, char* argv[])
{
        int N;
        scanf("%d",&N);
        printf("top=%d",top(N));
        return 0;
}
```

Recursion vs. Iteration

Recursion:

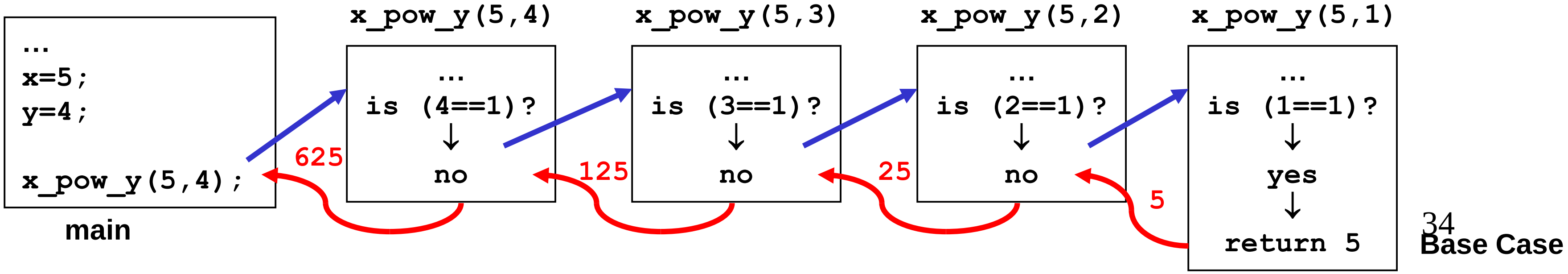
```
#include <stdio.h>
int x_pow_y(int, int);

main()
{
  printf(“enter x and y: \n”);
  scanf(“%d, %d”, x, y);
  z = x_pow_y(x,y);
  printf(“z: %d\n”, z);
}

x_pow_y(int a, int b)
{
  if (b==1)
     return a;
  else
     return (a * x_pow_y(a, b-1))
}
```

Iteration:

```
x_pow_y = 1;
  for (i = y; i >=1; i--)
    x_pow_y*=x;
```

If $x = 5$, $y = 4$

```
x_pow_y = 1, x = 5, y = 4, i = 4;
1.)  x_pow_y = 5*1 = 5, i = 3;
2.)  x_pow_y = 5*5 = 25, i = 2;
3.)  x_pow_y = 25*5 = 125, i = 1;
4.)  x_pow_y = 125*5 = 625, i = 0;
```



…
x=5;
y=4;

x_pow_y(5,4);

main

x_pow_y(5,4)
…
is (4==1)?
↓
no

x_pow_y(5,3)
…
is (3==1)?
↓
no

x_pow_y(5,2)
…
is (2==1)?
↓
no

x_pow_y(5,1)
…
is (1==1)?
↓
yes
↓
return 5

Base Case

Example Using Recursion: Factorial

Example: factorials: **`5! = 5 * 4 * 3 * 2 * 1`**

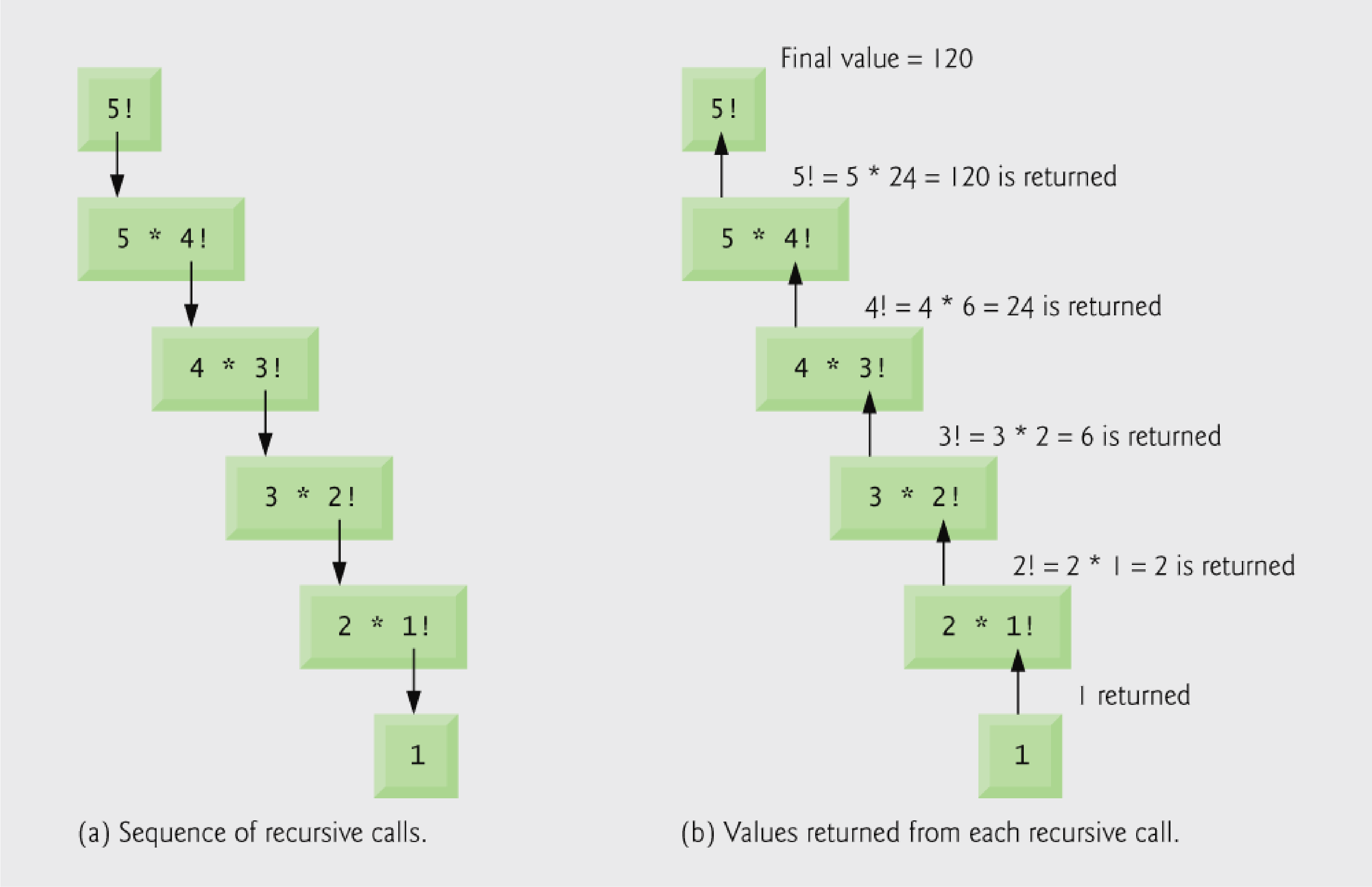
- Notice that
  - **`5! = 5 * 4!`**
  - **`4! = 4 * 3!`** …

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

- 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;`**

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

Example Using Recursion: Factorial


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

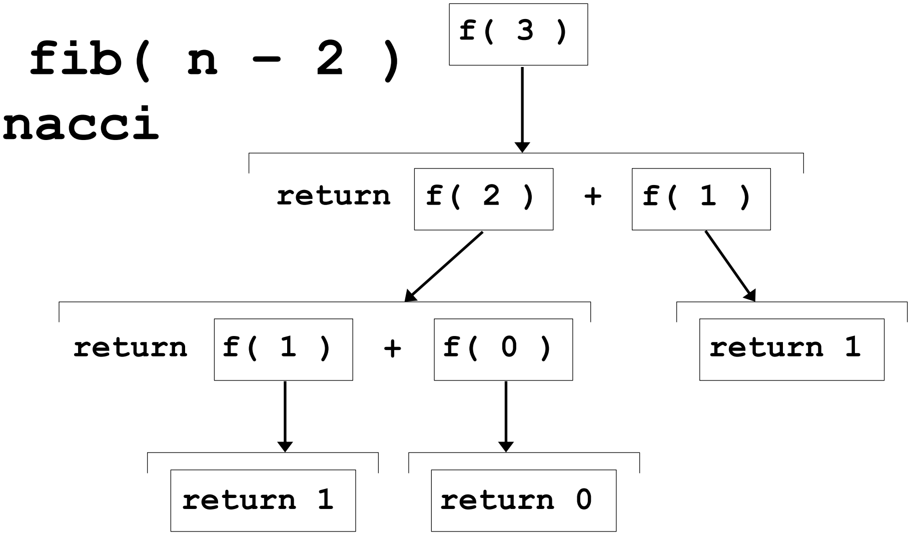
Example Using Recursion: The Fibonacci Series

Example: Fibonacci series:

$F_k = F_{k-1} + F_{k-2}$, $F_0 = 1$, $F_1 = 1$ ex: 0, 1, 1, 2, 3, 5, 8…

- Each number is the sum of the previous two
- Can be solved recursively:

  `fib( n ) = fib( n - 1 ) + fib( n – 2 )`
- Set of recursive calls to function **`fibonacci`**
- Code for the **`fibonacci`** function



```
long fibonacci(long n)
{
  if (n <= 1)
    return n;
  else;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

```
#include <stdio.h>
#include <conio.h>

long fibonacci (long);

long fibonacci (long x)
{
          if (x==1 || x==0)
          return x;
          else
   return fibonacci(x-2)+fibonacci(x-1);
}

int main (void)
{
          long number,fib;
          printf ("Enter an integer: ");
          scanf ("%ld",&number);
          fib=fibonacci(number);
          printf ("Fibonacci(%ld) = %ld",number,fib);
          getch();
          return 0;
}
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1
Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Recursion vs. Iteration

- Both are based on the control structures
  - Repetition (Iteration): explicitly uses repetition (loop).
  - Selection (Recursion): implicitly use repetition by successive function calls
- Both involve termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can lead infinite loops
  - Loop termination is not met
  - Base case is not reached
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)
- A reason to choose recursion is that an iterative solution may not apparent

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
- Format for calling functions
  - `FunctionName(` *`argument`* `);`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

Function	Description	Example

`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282`
		`exp( 2.0 )` is `7.389056`

`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 1.0 )` is `0.0`
		`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`fabs( x )`	absolute value of *x*	`fabs( 5.0 )` is `5.0`
		`fabs( 0.0 )` is `0.0`
		`fabs( -5.0 )` is `5.0`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

Function	Description	Example

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128.0`
		`pow( 9, .5 )` is `3.0`

`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Random Number Generation

- `rand` function
  - Load `<stdlib.h>`
  - Returns "random" number between `0` and `RAND_MAX` (at least `32767`)

    ```
    i = rand();
    ```
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between `1` and `n`

    ```
    1 + ( rand() % n )
    ```
    - `rand() % n` returns a number between `0` and `n - 1`
    - Add `1` to make random number between `1` and `n`

      ```
      1 + ( rand() % 6)
      ```
      - number between `1` and `6`

```c
/* Fig. 5.7: fig05_07.c
   Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
   int i; /* counter */

   /* loop 20 times */
   for ( i = 1; i <= 20; i++ ) {

      /* pick random number from 1 to 6 and output it */
      printf( "%10d", 1 + ( rand() % 6 ) );

      /* if counter is divisible by 5, begin new line of output */
      if ( i % 5 == 0 ) {
         printf( "\n" );
      } /* end if */

   } /* end for */

   return 0; /* indicates successful termination */

} /* end main */
```

Generates a random number between 1 and 6

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

```c
/* Fig. 5.8: fig05_08.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
   int frequency1 = 0; /* rolled 1 counter */
   int frequency2 = 0; /* rolled 2 counter */
   int frequency3 = 0; /* rolled 3 counter */
   int frequency4 = 0; /* rolled 4 counter */
   int frequency5 = 0; /* rolled 5 counter */
   int frequency6 = 0; /* rolled 6 counter */

   int roll; /* roll counter, value 1 to 6000 */
   int face; /* represents one roll of the die, value 1 to 6 */

   /* loop 6000 times and summarize results */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6; /* random number from 1 to 6 */

      /* determine face value and increment appropriate counter */
      switch ( face ) {

         case 1: /* rolled 1 */
            ++frequency1;
            break;

         case 2: /* rolled 2 */
            ++frequency2;
            break;

         case 3: /* rolled 3 */
            ++frequency3;
            break;

         case 4: /* rolled 4 */
            ++frequency4;
            break;

         case 5: /* rolled 5 */
            ++frequency5;
            break;

         case 6: /* rolled 6 */
            ++frequency6;
            break; /* optional */
      } /* end switch */

   } /* end for */

```

```c
   /* display results in tabular format */
   printf( "%s%13s\n", "Face", "Frequency" );
   printf( "   1%13d\n", frequency1 );
   printf( "   2%13d\n", frequency2 );
   printf( "   3%13d\n", frequency3 );
   printf( "   4%13d\n", frequency4 );
   printf( "   5%13d\n", frequency5 );
   printf( "   6%13d\n", frequency6 );

   return 0; /* indicates successful termination */

} /* end main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

Random Number Generation

- `srand` function
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**
  - `srand( time( NULL ) );/*load <time.h>*/`
  - `time( NULL )`
    - Returns the number of seconds that have passed since January 1, 1970
    - “Randomizes" the seed

Real Random Number Generation

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
  int i,j;
  srand( (unsigned int)time( NULL ) );
   for (i=1;i<=10;i++)
    {
     j=rand();
     printf("%d ",j);
    }
getch();
return 0;
}
```

Time

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i,j;
    time_t t1,t2;
    /* t1 degiskeninin adresi fonksiyona gonderiliyor.
  o adrese o anki zaman yaziliyor*/
    time(&t1);
    for (i=1;i<=300;i++)
       for (j=1;j<=300;j++)
       printf("%d %d %d\n",i, i*i, i*i*i);
    time(&t2);
    printf("n Time to do 300 squares and cubes= %d
  secondsn",  t2-t1);
getch();
return 0;
}
```

Symbolic Constants

```
#define LOWER 0
#define UPPER 100
#define <name> <replacement text>
```

- Replaces each occurrence of <name> with <replacement text>

e.g. `fahr = LOWER`

becomes `fahr = 0`

Can also define: `#define SQ(X) X*X`

Symbolic Constants

```
#define REPL 5+1
#define SQ(X) X*X
#include <stdio.h>

int main()
{
   printf("%d\n", SQ(REPL));
}
```

Output = 36? NO! It’s 11!!!

Why? SQ(REPL) becomes REPL*REPL which is 5+1*5+1

Referance

- Ioannis A. Vetsikas, Lecture notes
- Dale Roberts, Lecture notes