AAB University

Faculty of Computer Sciences

Object Oriented Programming

Week 6:

Functions and Introduction to Recursion

Asst. Prof. Dr. **Mentor Hamiti**

mentor.hamiti@universitetiaab.com

Last Time?!

- **Control Structures**
- **if** Selection Statement
- **if … else** Double-Selection Statement
- **while** Repetition Statement
- **for** Repetition Statement
- **do … while** Repetition Statement
- **switch** Multiple-Selection Statement
- **break** and **continue** Statements

Today

- **Standard Library Functions**
  - Math Library Functions
- **User Defined Functions**
  - Standard Functions
  - Inline Functions
  - Macro Functions
- **More on Functions**
  - References and Reference Parameters
  - Defaults Arguments
  - Unary Scope Resolution Operator
  - Function Overloading
- **Recursions**

Introduction

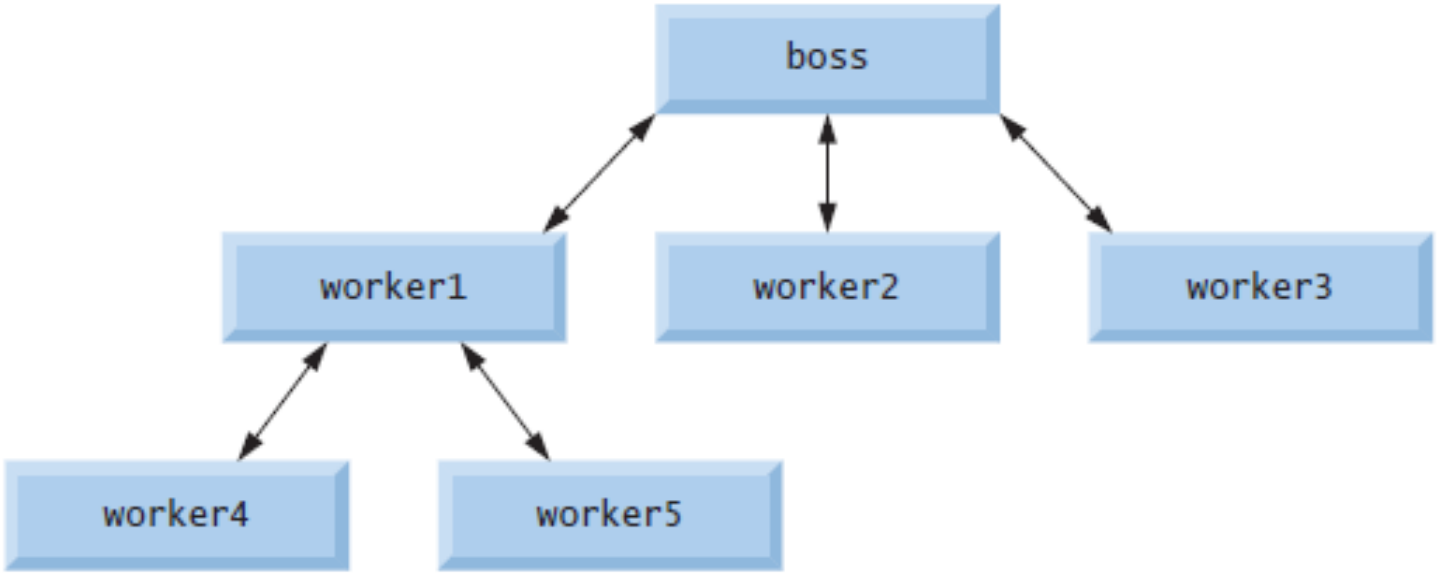
- Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or components
  - This technique is called divide and conquer
- C++ programs are typically written by combining new functions and classes you write with "prepackaged" functions and classes available in the C++ Standard Library

Program Components in C++

- The C++ Standard Library provides a rich collection of functions
- Functions allow you to **modularize** a program by separating its tasks into self-contained units
- Functions you write are referred to as **user-defined functions**
- The statements in function bodies are written only once, are **reused** from perhaps several locations in a program and are hidden from other functions

Program Components in C++

- A function is invoked by a function call, and when the called function completes its task, it either returns a result or simply returns control to the caller
- An analogy to this program structure is the hierarchical form of management ☺



Standard Library Functions

- Math Library Functions

Math Library Functions

- Sometimes functions are not members of a class
  - Called **global functions**
  - Function prototypes for global functions are placed in header files, so that the global functions can be reused in any program that includes the header file and that can link to the function's object code
- The **<cmath>** header file provides a collection of functions that enable you to perform common **mathematical calculations**
  - All functions in the **<cmath>** header file are global functions—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments

Math Library Functions

Function	Description	Example
`ceil( x )`	rounds x to the smallest integer not less than x	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`cos( x )`	trigonometric cosine of x (x in radians)	`cos( 0.0 ) is 1.0`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.718282` `exp( 2.0 ) is 7.389056`

`fabs( x )`	absolute value of x	`fabs( 5.1 ) is 5.1`
		`fabs( 0.0 ) is 0.0`
		`fabs( -8.76 ) is 8.76`

`floor( x )`	rounds x to the largest integer not greater than x	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`fmod( x, y )`	remainder of x/y as a floating-point number	`fmod( 2.6, 1.2 ) is 0.2`

Math Library Functions

Function	Description	Example
`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( 10.0 ) is 1.0` `log10( 100.0 ) is 2.0`
`pow( x, y )`	x raised to power y (x^y)	`pow( 2, 7 ) is 128` `pow( 9, .5 ) is 3`
`sin( x )`	trigonometric sine of x (x in radians)	`sin( 0.0 ) is 0`
`sqrt( x )`	square root of x (where x is a nonnegative value)	`sqrt( 9.0 ) is 3.0`
`tan( x )`	trigonometric tangent of x (x in radians)	`tan( 0.0 ) is 0`

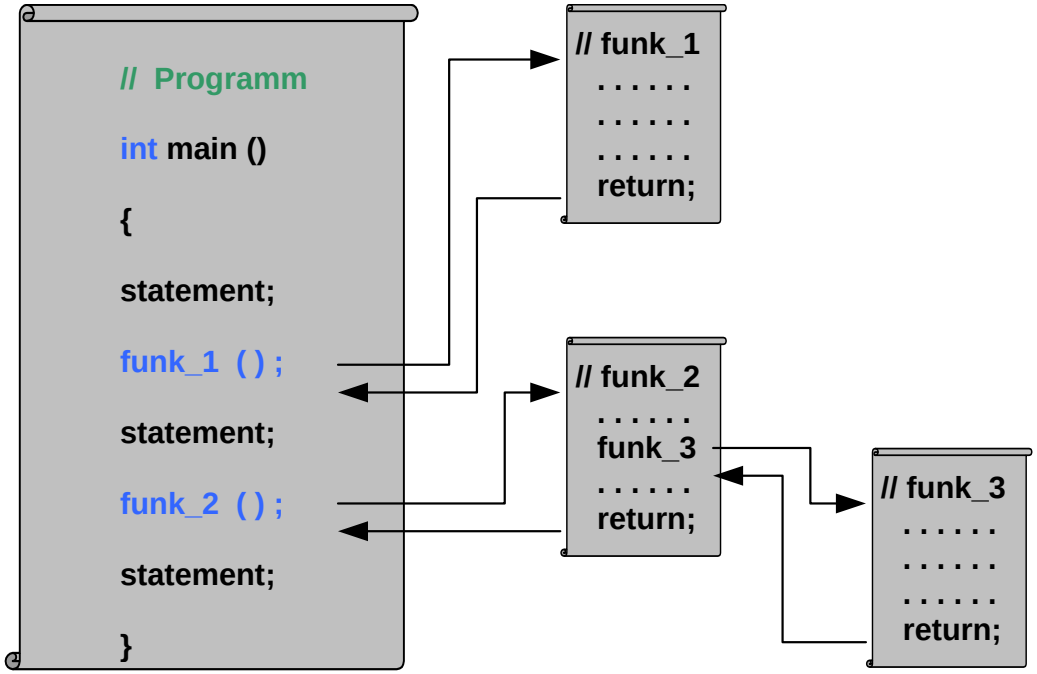
- Standard Library Functions
  - Math Library Functions
- User Defined Functions
  - Standard Functions
  - Inline Functions
  - Macro Functions

Function ***Prototypes*** *and* ***Argument***

- A function prototype (also called a **function declaration**) tells the compiler:
  - The name of a function
  - The type of data returned by the function
  - The number of parameters the function expects to receive
  - The types of those parameters and
  - The order in which the parameters of those types are expected

Function Prototypes and Argument

- Functions often require more than one piece of information to perform their tasks
  - Such functions have multiple parameters
- There are also functions without parameters



Function Prototypes and Argument

- There are several ways to return control to the point at which a function was invoked
  - If the function does return a result, the statement "return expression;" evaluates expression and returns the value of expression to the caller
  - If the function does not return a result (i.e., it has a **void** return type), control returns when the program reaches the function-ending right brace, or by execution of the statement "return;"

Functions with Empty Parameter Lists

- In C++, an empty parameter list is specified by writing either **void** or nothing at all in parentheses

Functions with Empty Parameter Lists

- Example 1:

```
//Star Line
# include <iostream>
# include <iomanip>
using namespace std;

void starLine ()
{
    cout<<"\n"<<setw(79)<<setfill('*')<<'*'<<endl;
};

int main()
{
    starLine();

    cout<<"\n\tUniversiteti AAB"<<endl;

    starLine();

    cin.get();
    return 0;
}
```

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

        Universiteti AAB

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

Functions with Multiple Parameters

- Example 2:

Function that return a value

```
//Maximum Number x, y, z
# include <iostream>
using namespace std;

int maximumNumber (int x, int y, int z)
{
     int maximumValue = x;
     if (y > maximumValue)
          maximumValue = y;
     if (z > maximumValue)
          maximumValue = z;
     return maximumValue;
};

int main()
{
     int Number;
     int a, b, c;
     cout<<"Enter Three Integer Numbers:"<<endl;
     cin>>a>>b>>c;
     Number = maximumNumber (a, b, c);
     cout<<"\nMaximum Number is:"<<Number<<endl;
     cin.get(); cin.get();
     return 0;
}
```

```
Enter Three Integer Numbers:
1
7
6

Maximum Number is:7
```

Declaring function parameters of the same type as `double x, y` instead of `double x, double y` is a syntax error—a type is required for each parameter in the parameter list.

Functions with Multiple Parameters

- Example 3:

Function that does not return a value

```
//Maximum Number x, y, z
# include <iostream>
using namespace std;

void maximumNumber (int x, int y, int z)
{
     int maximumValue = x;
     if (y > maximumValue)
          maximumValue = y;
     if (z > maximumValue)
          maximumValue = z;
     cout<<maximumValue;
};

int main()
{
     int a, b, c;
     cout<<"Enter Three Integer Numbers:"<<endl;
     cin>>a>>b>>c;
     cout<<"\nMaximum Number is:";
     maximumNumber (a, b, c);
     cin.get(); cin.get();
     return 0;
}
```

```
Enter Three Integer Numbers:
1
7
6

Maximum Number is:7
```

Inline Functions

- C++ provides inline functions to help reduce function call overhead

- Placing the qualifier **in-line** before a function's return type in the function definition advises the compiler to generate a copy of the function's code in every place where the function is called (when appropriate) to avoid a function call
  - This often makes the program larger

- Reusable inline functions are typically placed in headers, so that their definitions can be included in each source file that uses them

Inline Functions

- Example 4:

```
// Using an inline function to calculate the volume of a cube.
#include <iostream>
using namespace std;

inline double cube(double side )
{
return side * side * side;      // calculate cube
}

int main()
{
    double sideValue; // stores value entered by user

    cout << "Enter the side length of your cube: ";
    cin >> sideValue; // read value from user

    cout << "Volume of cube with side "<< sideValue << " is " <<cube( sideValue )<< endl;

    cin.get(); return 0;
}
```

Inline Function

```
Enter the side length of your cube: 3
Volume of cube with side 3 is 27
```

Macro Functions

- Macro ar functions that are written in a single line
- Example: Which solution do you prefere?!

```
#define MAX (a,b)  (a > b)  ?  a : b
```

OR

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

Macro Functions

- Example 5:

```
//Macro Function
# include <iostream>
using namespace std;

# define cube(a) (a*a*a)

int main()
{

    int x;
    cout<<"Side =";
    cin>>x;

    //V = cube(a);

    cout << "Volume of cube is " << cube(x) << endl;

    cin.get(); return 0;
}
```

Macro Function

```
Side = 5
Volume of cube is 125
```

- Standard Library Functions
  - Math Library Functions
- User Defined Functions
  - Standard Functions
  - Inline Functions
  - Macro Functions
- More on Functions
  - References and Reference Parameters
  - Defaults Arguments
  - Unary Scope Resolution Operator
  - Function Overloading

References and Reference Parameters

- Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**

  - When an argument is passed by value, a copy of the argument's value is made and passed to the called function
  - Changes to the copy do not affect the original variable's value in the caller

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

References and Reference Parameters

- With **pass-by-reference**, the caller gives the called function the ability to access the caller's data directly, and to modify that data
- A reference parameter is an alias for its corresponding argument in a function call
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand "**&**"

References and Reference Parameters

- Example 6:

```
// Comparing pass-by-value and pass-by-reference with references.
# include <iostream>
using namespace std;

int squareByValue( int number)      // function prototype (value pass)
    { return number *= number; }    // caller's argument not modified

void squareByReference( int &numberRef)     // function prototype (reference pass)
{ numberRef *= numberRef; }         // caller's argument modified

int main()
{
    int x = 2; // value to square using squareByValue
    int z = 4; // value to square using squareByReference

    // demonstrate squareByValue
    cout << "x = " << x << " before squareByValue\n";
    cout << "Value returned by squareByValue: " << squareByValue( x ) << endl;
    cout << "x = " << x << " after squareByValue\n" << endl;
    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference ( z );
    cout << "z = " << z << " after squareByReference" << endl;
    cin.get(); return 0;
}
```

Reference Parameters

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Default Arguments

- It isn't uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter
  - Can specify that such a parameter has **a default argument**, i.e., a default value to be passed to that parameter
- When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument
- **Default arguments** must be the rightmost (trailing) arguments in a function's parameter list

Default Arguments

- Example 7:

Default Arguments

```
// Using default arguments.
# include <iostream>
using namespace std;

// function boxVolume calculates the volume of a box
int boxVolume( int length=1, int width=1, int height=1 )
{
return length * width * height;
}

int main()
{

    // no arguments--use default values for all dimensions
    cout << "The default box volume is: " << boxVolume();

    // specify length; default width and height
    cout << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume(10);

    // specify length and width; default height
    cout << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume(10, 5);

    cin.get();  return 0;
}
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50
```

Unary Scope Resolution Operator

- It's possible to declare **local** and **global** variables of the same name
- C++ provides the unary scope resolution operator "**::**" to access a global variable when a local variable of the same name is in scope
- Using the unary scope resolution operator "**::**" with a given variable name is optional when the only variable with that name is a global variable

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.

Unary Scope Resolution Operator

- Example 8:

```
// Using the unary scope resolution operator.
#include <iostream>
using namespace std;

int number = 7; // global variable named number

int main()
{
    double number = 10.5; // local variable named number

    // display values of local and global variables

    cout << "Local double value of number = " <<number
        << "\nGlobal int value of number = " <<::number<< endl;

    cin.get(); return 0;
}
```

Global vs Local Variables

```
Local double value of number = 10.5
Global int value of number = 7
```

Function Overloading

- C++ enables several functions of the **same name** to be defined, as long as they have different signatures
  - This is called function overloading
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types

Overloading functions that perform closely related tasks can make programs more readable and understandable.

Function Overloading

- Example 9:

Function Overloading

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

```
// Overloaded functions
#include <iostream>
using namespace std;

// function square for int values
int square( int x )
{
cout << "square of integer " << x << " is ";
return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
cout << "square of double " << y << " is ";
return y * y;
} // end function square with double argument

int main()
{
cout << square( 7 ); // calls int version
cout << endl;
cout << square( 7.5 ); // calls double version
cout << endl;

cin.get(); return 0;
}
```

- **Standard Library Functions**
  - Math Library Functions
- **User Defined Functions**
  - Standard Functions
  - Inline Functions
  - Macro Functions
- **More on Functions**
  - References and Reference Parameters
  - Defaults Arguments
  - Unary Scope Resolution Operator
  - Function Overloading
- **Recursions**

Recursion

- A **recursive function** is a function that calls itself, either directly, or indirectly (through another function)

- **Factorial**
  - The factorial of a nonnegative integer **n**, written **n!**, (1! = 1, and 0! = 1) is the product :

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

- The factorial of an integer, number, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) by using a loop:

```
factorial = 1;
for ( int counter = number; counter >= 1; --counter )
   factorial *= counter;
```

Recursion

- A **recursive** definition of the factorial function is arrived at by observing the following algebraic relationship:

$$n! = n \cdot (n-1)!$$

- Example:

5!

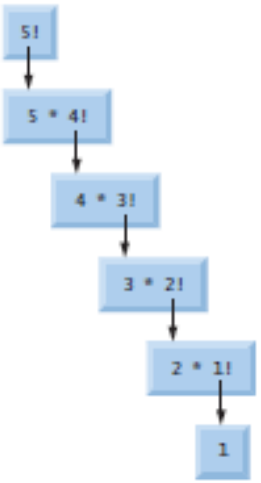
5 * 4!

5 * 4 * 3!

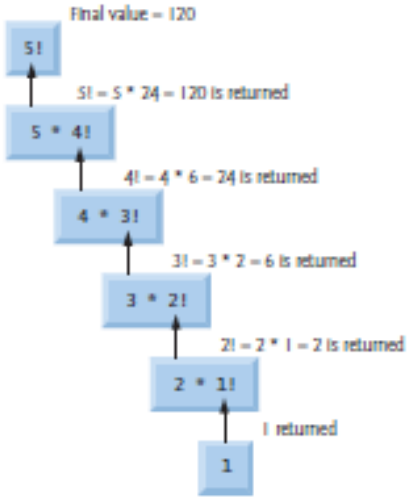
5 * 4 * 3 * 2!

5 * 4 * 3 * 2 * 1!

5 * 4 * 3 * 2 * 1 = **120**

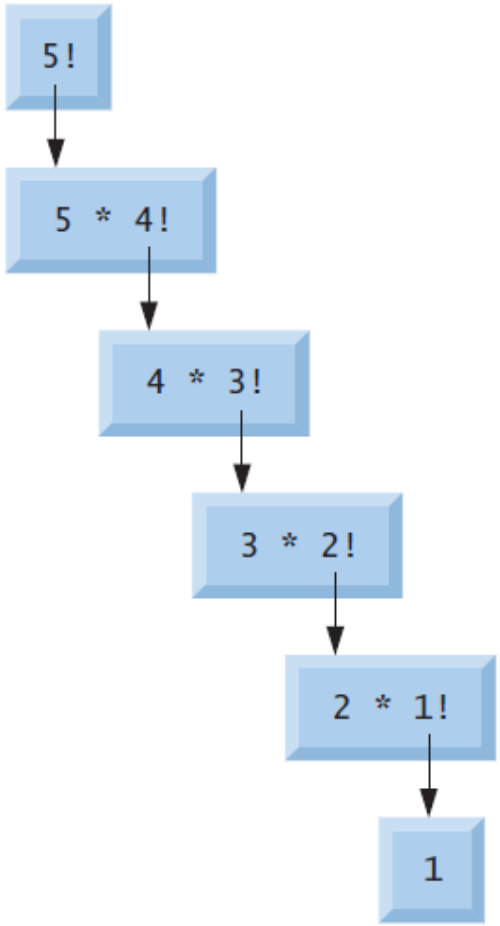


Procession of recursive calls

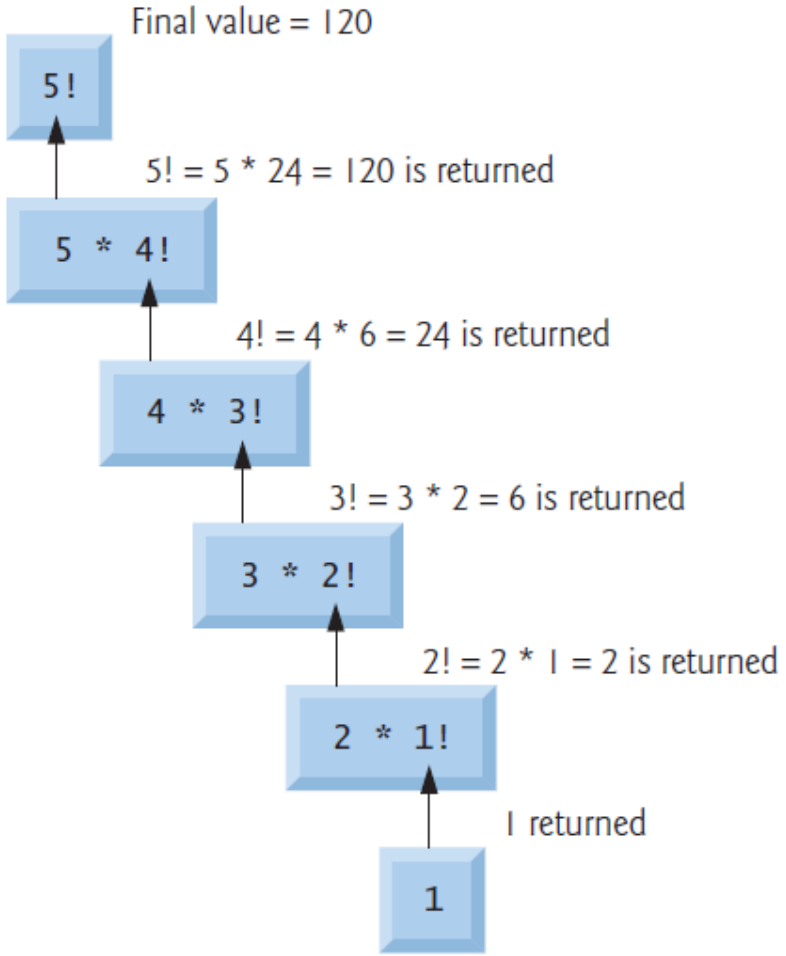


Values returned from each recursive call

- The evaluation of **5!** would proceed as follow:



Procession of recursive calls



Values returned from each recursive call

Recursion

- Example 10:

Recursive Factorial

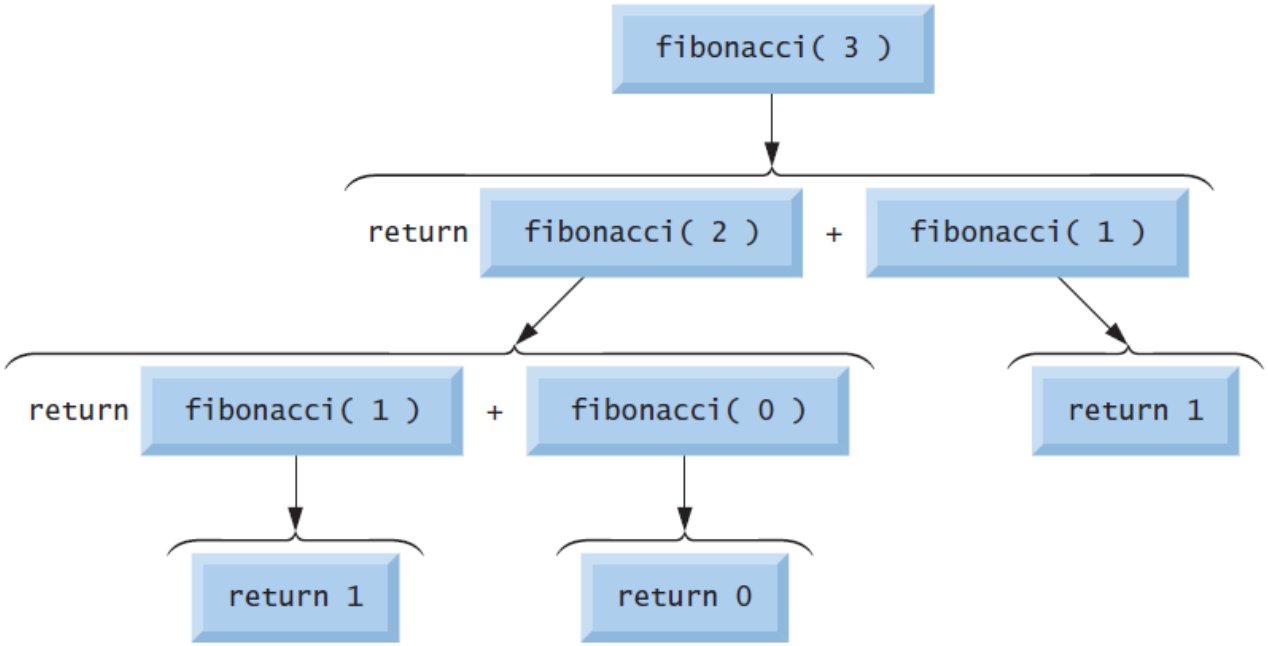
```
n = 5

      Factorial ( 5 ) = 120
```

```
#include <iostream>
using namespace std;

int Factorial (int n);

int main()
{

int n, Result;
cout<<"\n";
cout<<" n = ";  cin>>n;
cout<<"\n\n";

Result=Factorial(n);
cout<<"\n\tFactorial ( "<<n<<" ) = "<<Result;

    cin.get(); cin.get();
    return 0;
}

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

- The **Fibonacci Series:**

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers



Recursion

- Example 11:

Recursive Fibonacci

```
#include <iostream>
using namespace std;

int fib (int n);

int main()
{

cout<<"\nFibonacci:\n"<<endl;
cout<<"\t1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .\n"<<endl;
cout<<"..........................................................\n"<<endl;

int n, Result;
cout<<"n = ";   cin>>n;
cout<<"\n\n";

Result=fib(n);
cout<<"\n\tfib ( "<<n<<" ) = "<<static_cast<float>(Result);
//convert from int to float

    cin.get(); cin.get();
    return 0;
}

                    int fib (int n)
                        {
                        if (n<3)
                            return (1);
                        else
                            return (fib(n-2)+fib(n-1));
                        }
```

```
Fibonacci:
        1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

----------------------------------------
n = 10


        fib ( 10 ) = 55
```

Recursion vs. Iteration

- Both iteration and recursion are based on **a control statement**:
  - Iteration uses a repetition structure;
  - Recursion uses a selection structure
- Both iteration and recursion involve **repetition**:
  - Iteration explicitly uses a repetition structure;
  - Recursion achieves repetition through repeated function calls
- Iteration and recursion each involve a **termination test**:
  - Iteration terminates when the loop-continuation condition fails;
  - Recursion terminates when a base case is recognized

Recursion vs. Iteration

- Both iteration and recursion can occur **infinitely**:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false;
  - Infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base

- Questions?!

mentor.hamiti@universitetiaab.com