# OOP/COMPUTER PROGRAMMING

Instructor: Dr. Danish Shehzad

# QUIZ - 01

- Write a program that generates the following output

| | |
|---|---|
| 1900 | 2135 |
| 1950 | 2235 |
| 2000 | 2335 |
| 2050 | 2435 |

- Using **static** two-dimensional arrays for input/output
- Using **dynamic** two-dimensional arrays for input/output

# FUNCTIONS

- What is a function?
- How many ways to pass arguments to a function?

# TODAYS TOPICS

- Functions Overloading
- Recursion
- Inline Functions

# FUNCTIONS

- **A function** is a group of statements that together perform a task.
- Here are all the parts of a function −
  - **Return Type** − A function may/maynot return a value.
  - **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
  - **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
  - **Function Body** − The function body contains a collection of statements that define what the function does.

# CALLING FUNCTIONS

- 3 ways to pass arguments to function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- Arguments passed to function using reference arguments
  - Modify original values of arguments

# WHAT IS FUNCTION OVERLOADING?

- Function overloading is the important characteristic of OOP.

- Two or more functions are said to be overloaded if they have the same name but different number of arguments.

- Also two or more functions are said to be overloaded if they have the same name, same number of arguments but the type of arguments are different.

- The C++ compiler can easily differentiate between such functions depending upon the number or type or order of arguments.

# EXAMPLE-01

**To understand the idea of function overloading consider the following example.**

```
void draw()
{
        for(int i=0;i<45;i++)
                    cout<<'+';
            cout<<endl;
}
void draw(char ch)
{
            for(int i=0;i<45;i++)
                    cout<<ch;
            cout<<endl;
}
void draw(char ch, int n)
{
            for(int i=0;i<n; i++)
                    cout<<ch;
            cout<<endl;
```

```
void draw()
void draw(char ch)
void draw(char ch, int n)
…
{`
        draw();
        draw('=');
        draw('*' , 30);
}
```

No argument

One argument

Two arguments

# RECURSION

Is used to solve a problem, where the solution to a problem depends on solutions to smaller instances of the same problem

# RECURSION

$$x^4 = x*x*x*x = x*( x*x*x ) = x*x^3$$

$$x^5 = x*x*x*x*x = x*(x*x*x*x ) = x*x^4$$

$$x^6 = x*x*x*x*x*x = x*(x*x*x*x*x ) = x*x^5$$

In general

$$\boxed{x^n = x*x^{n-1}}$$

# RECURSION

Unfortunately, this function is still not quite complete.

power 2(3)

→ 2 * power 2 (3-1) = 2 * power 2(2)

→ 2 * 2 * power 2 (1)

→ 2 * 2 * power 2 (0)

→ 2 * 2 * 2 * power (2 -1)

→ …

When does it stop ?

# RECURSION

**Remember that power should only be defined for *$n \geq 0$.***

**Therefore we need to stop at**

**power 2 0**

**This is called the *base case*.**

# RECURSION

**New power function:**

```
power x n
    | n == 0          = 1
    | otherwise       = x * power x (n-1)
```

**The function definition is said to be *recursive*, since it calls itself.**

# RECURSION

**To build *all* recursive functions:**

Sub-problems solved as a recursive call to the same function

1. Define the base case(s)
2. Define the recursive case(s)
   a) Divide the problem into smaller sub-problems
   b) Solve the sub-problems
   c) Combine results to get answer

# RECURSION

**Note:**

***the sub-problems must be "smaller" than the original problem otherwise the recursion never terminates.***

```
-- loop function
loop x = 1 + loop x
```

# RECURSION

**Trace:**

loop 5

→ 1 + loop 5

→ 1 + loop 5

→ 1 + loop 5

→ ...

infinite loop – no termination

# RECURSION

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

# EXAMPLE 2: FACTORIAL FUNCTION

In general, we can express the factorial function as follows:

```
n! = n * (n-1)!
```

Is this correct? Well… almost.

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

```
n! = 1              (if n is equal to 1)
n! = n * (n-1)!     (if n is larger than 1)
```

# FACTORIAL FUNCTION

The C++ equivalent of this definition:

```
int fac(int numb){
   if(numb<=1)
      return 1;
   else
      return numb * fac(numb-1);
}
```

***recursion* means that a function calls itself**

# FACTORIAL FUNCTION

- Assume the number typed is 3, that is, numb=3.

```
fac(3) :
 3 <= 1 ?                    No.
 fac(3) = 3 * fac(2)
   fac(2) :
      2 <= 1 ?               No.
      fac(2) = 2 * fac(1)
          fac(1) :
             1 <= 1 ?        Yes.
             return 1
      fac(2) = 2 * 1 = 2
      return fac(2)
 fac(3) = 3 * 2 = 6
 return fac(3)
 fac(3)  has the value 6
```

```
int fac(int numb){
  if(numb<=1)
     return 1;
  else
     return numb * fac(numb-1);
}
```

# FACTORIAL FUNCTION

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive solution

```
int fac(int numb){
  if(numb<=1)
    return 1;
  else
    return numb*fac(numb-1);
}
```

Iterative solution

```
int fac(int numb){
    int product=1;
    while(numb>1){
      product *= numb;
      numb--;
    }
    return product;
}
```

# RECURSION

We have to pay a price for recursion:

- calling a function consumes more time and memory than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)

# RECURSION

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
    ...
```

Oops!

```
int result = 1;
while(result >0){
    ...
    result++;
}
```

Oops!

# RECURSION

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
    return numb * fac(numb-1);
}
```

Or:

```
int fac(int numb){
    if (numb<=1)
        return 1;
    else
        return numb * fac(numb+1);
}
```

Oops!
No termination condition

Oops!

# RECURSION

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

# DIRECT COMPUTATION METHOD

- Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - **`F(0) = 0;`**
  - **`F(1) = 1;`**
  - **`F(number) = F(number-1)+ F(number-2);`**

# RECURSION

**Recursive Examples**

1. ***Factorial function***

**(Revision)** **0! = 1**

**1! = 1**

**2! = 2 * 1 = 2**

**3! = 3 * 2 * 1 = 6**

**4! = 4 * 3 * 2 * 1 = 24**

The Fibonacci Numbers and Golden section in Nature - 1 - Netscape

File Edit View Go Communicator Help

Back Forward Reload Home Search Netscape Print Security Stop

Bookmarks Location: http://www.ee.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibnat.html What's Related

# Fibonacci Numbers and Nature

This page has been split into TWO PARTS.

This, the **first**, looks at the Fibonacci numbers and why they appear in various "family trees" and patterns of spirals of leaves and seeds.

The second page then examines why the golden section is used by nature in some detail, including animations of growing plants.

## Contents of this Page

The line means there is a Things to do investigation at the end of the section.

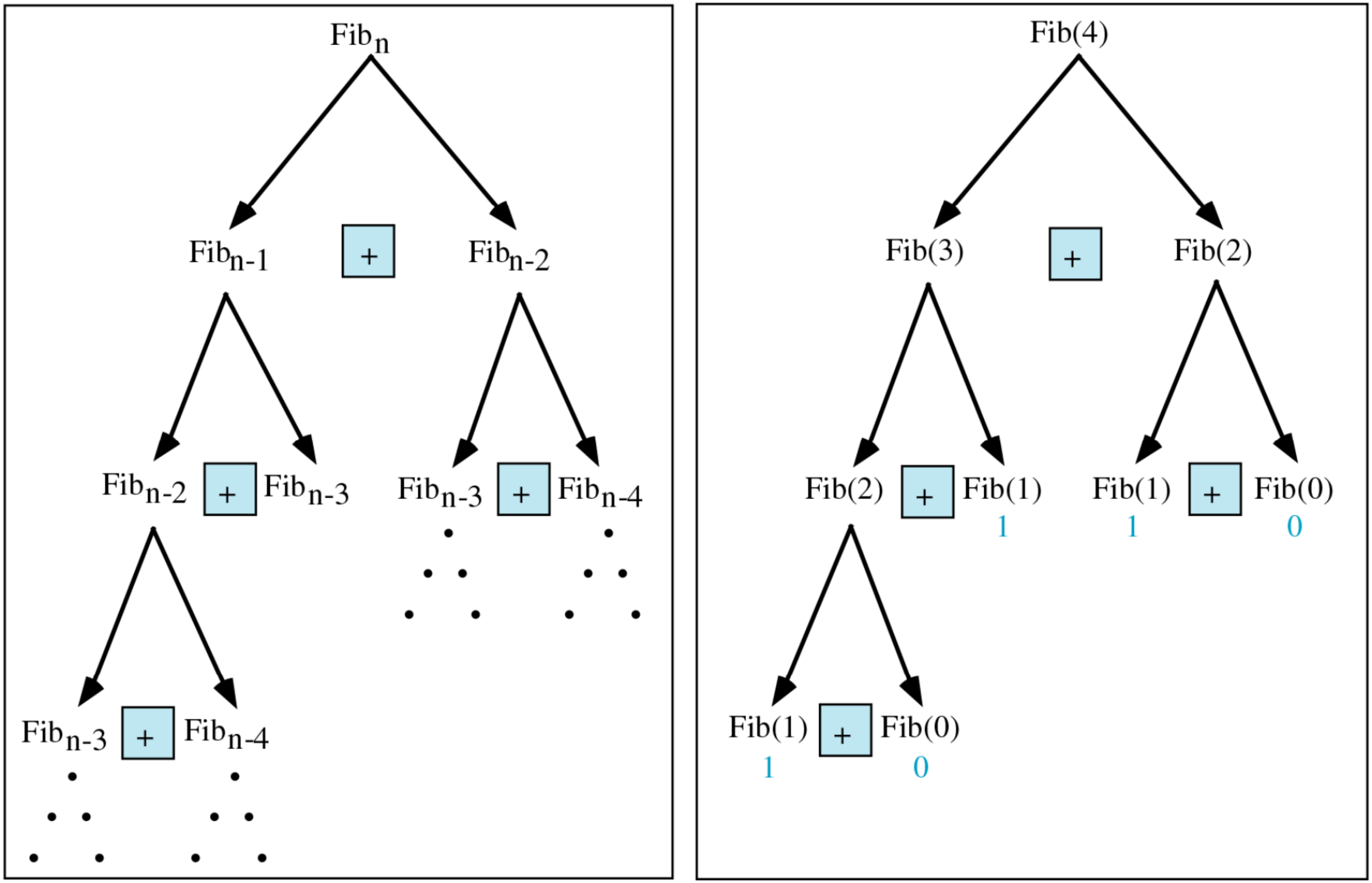


Document: Done

# EXAMPLE 3: FIBONACCI NUMBERS

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well
int fib(int number)
{
  if (number == 0) return 0;
  if (number == 1) return 1;
  return (fib(number-1) + fib(number-2));
}

int main(){        // driver function
   int inp_number;
   cout << "Please enter an integer: ";
   cin >> inp_number;
   cout << "The Fibonacci number for "<< inp_number
         << " is "<< fib(inp_number)<<endl;
  return 0;
```



$Fib_n$
$Fib_{n-1}$
+
$Fib_{n-2}$
$Fib_{n-2}$
+
$Fib_{n-3}$
$Fib_{n-3}$
+
$Fib_{n-4}$
$Fib_{n-3}$
+
$Fib_{n-4}$
Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0


(a) Fib(n)

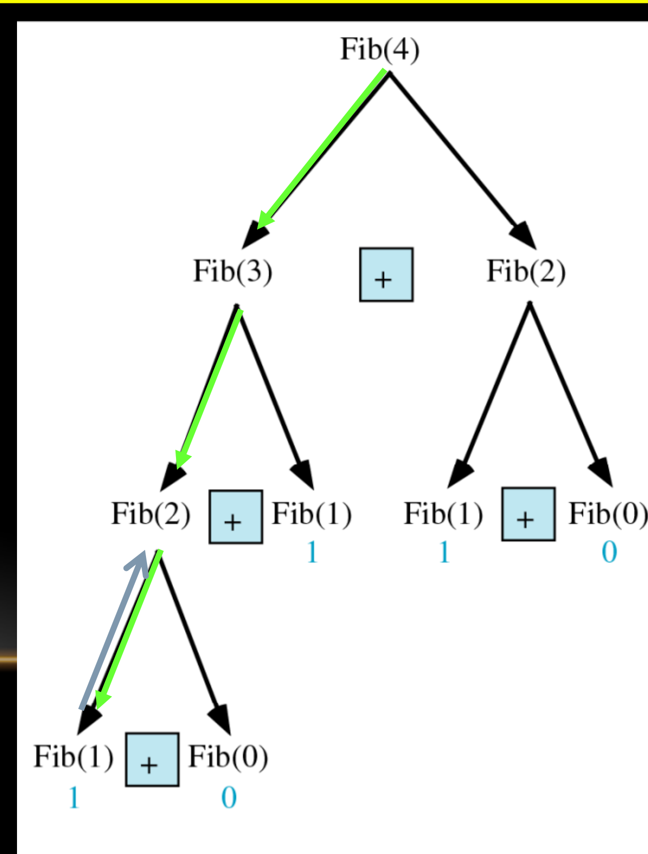
(b) Fib(4)

# TRACE A FIBONACCI NUMBER

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```

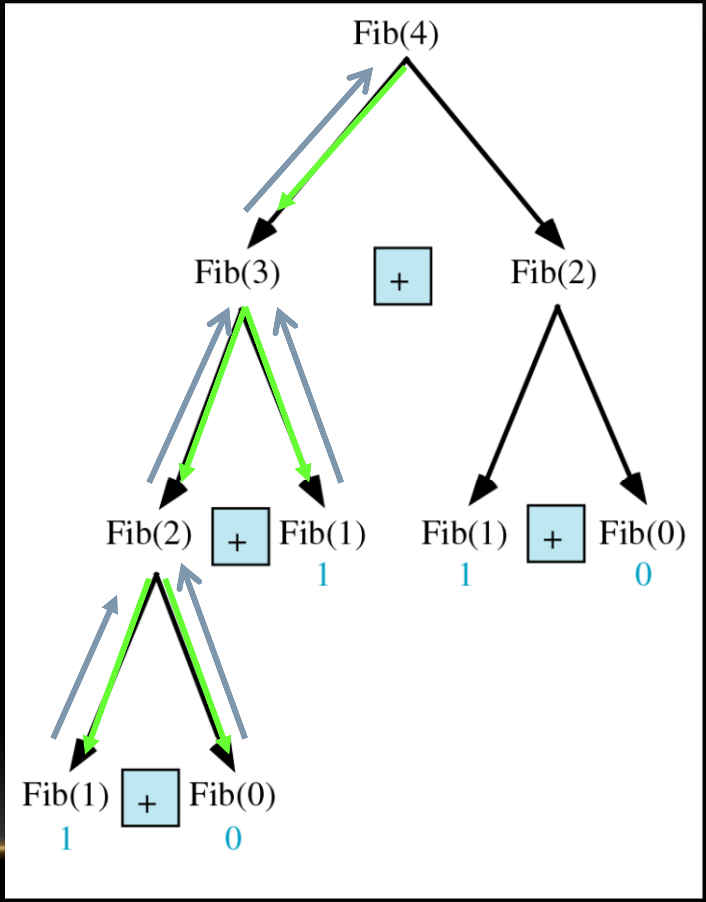
- Assume the input number is 4, that is, num=4:

```
fib(4):
    4 == 0 ? No;    4 == 1? No.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? No; 3 == 1? No.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? No; 2==1? No.
            fib(2) = fib(1)+fib(0)
            fib(1):
                1== 0 ? No; 1 == 1? Yes.
                fib(1) = 1;
                return fib(1);
```



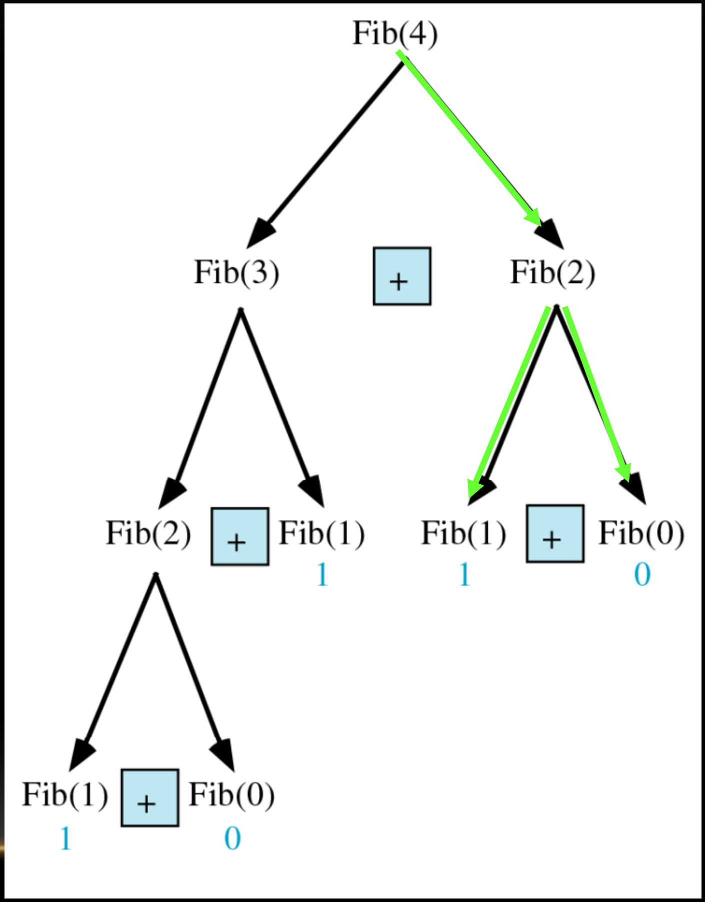
# TRACE A FIBONACCI NUMBER

```
        fib(0):
            0 == 0 ?  Yes.
            fib(0) = 0;
            return fib(0);
    fib(2) = 1 + 0 = 1;
    return fib(2);
fib(3) = 1 + fib(1)
    fib(1):
    1 == 0 ? No; 1 == 1? Yes
    fib(1) = 1;
     return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```



# TRACE A FIBONACCI NUMBER

```
fib(2):
    2 == 0 ? No; 2 == 1?   No.
    fib(2) = fib(1) + fib(0)
    fib(1):
       1== 0 ? No; 1 == 1?   Yes.
        fib(1) = 1;
        return fib(1);
     fib(0):
         0 == 0 ?    Yes.
         fib(0) = 0;
        return fib(0);
     fib(2) = 1 + 0 = 1;
     return fib(2);
  fib(4) = fib(3) + fib(2)
         = 2 + 1 = 3;
```



# HOMEWORK

- Tower of Hannoi through recursion

**Obeying the following rules:**

1. Only one disk can be transfer at a time.
2. Each move consists of taking the upper disk from one of the peg and placing it on the top of another peg i.e. a disk can only be moved if it is the uppermost disk of the peg.
3. Never a larger disk is placed on a smaller disk during the transfer.