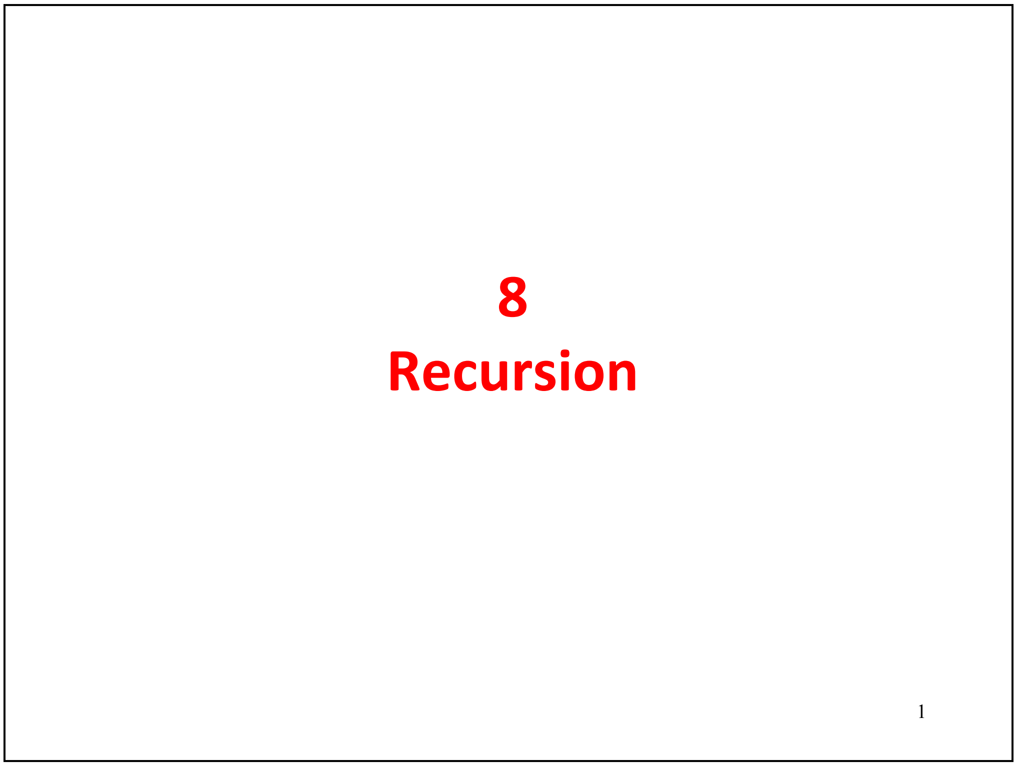
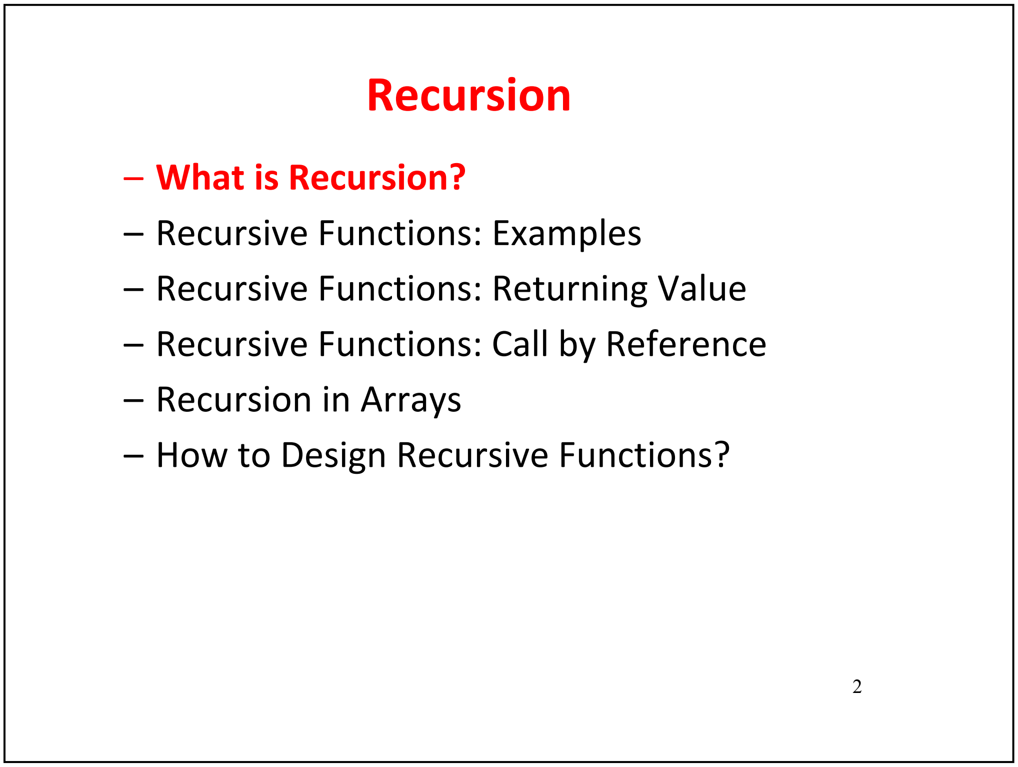


8
Recursion

1

Why Learning Recursion

1. One of the most interesting features of C is the ability of a function to call itself. This is known as recursion.
2. The use of recursion is particularly convenient for those problems that can be defined in naturally recursive terms, though such problems can also be programmed using non-recursive techniques.
3. In this lecture, we will discuss the concepts of recursion and give a few examples on recursive functions.



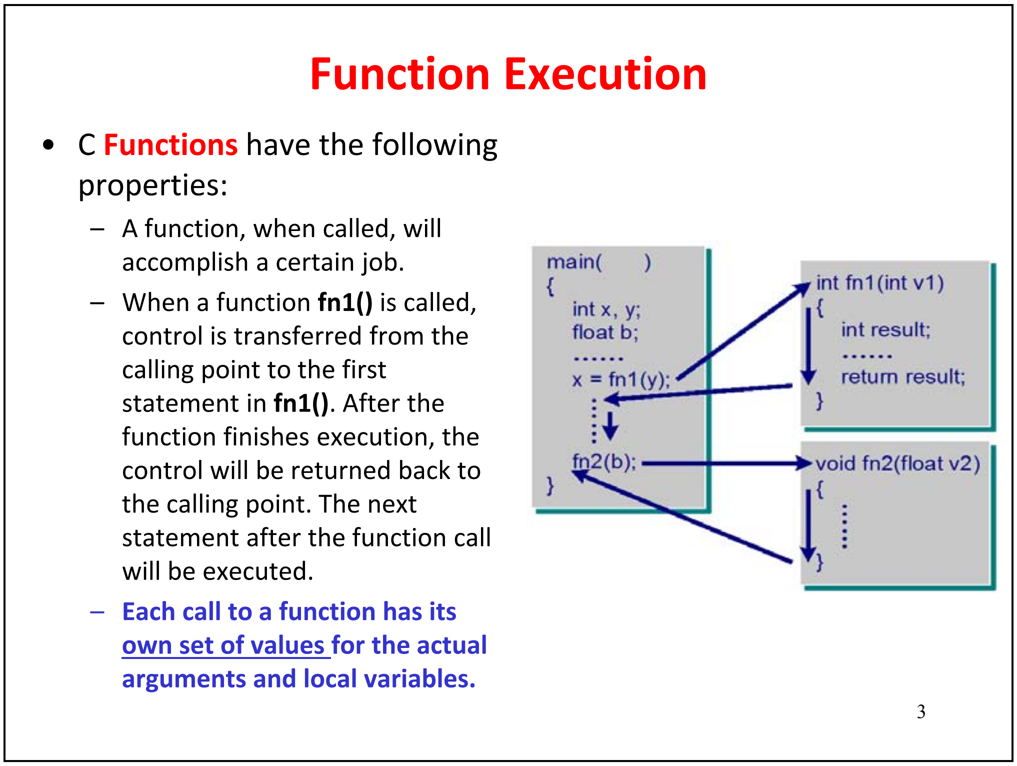
Recursion

- What is Recursion?
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

2

Recursion

1. Here, we first by discussing what is recursion.



Function Execution

- C **Functions** have the following properties:
  - A function, when called, will accomplish a certain job.
  - When a function **fn1()** is called, control is transferred from the calling point to the first statement in **fn1()**. After the function finishes execution, the control will be returned back to the calling point. The next statement after the function call will be executed.
  - **Each call to a function has its own set of values for the actual arguments and local variables.**

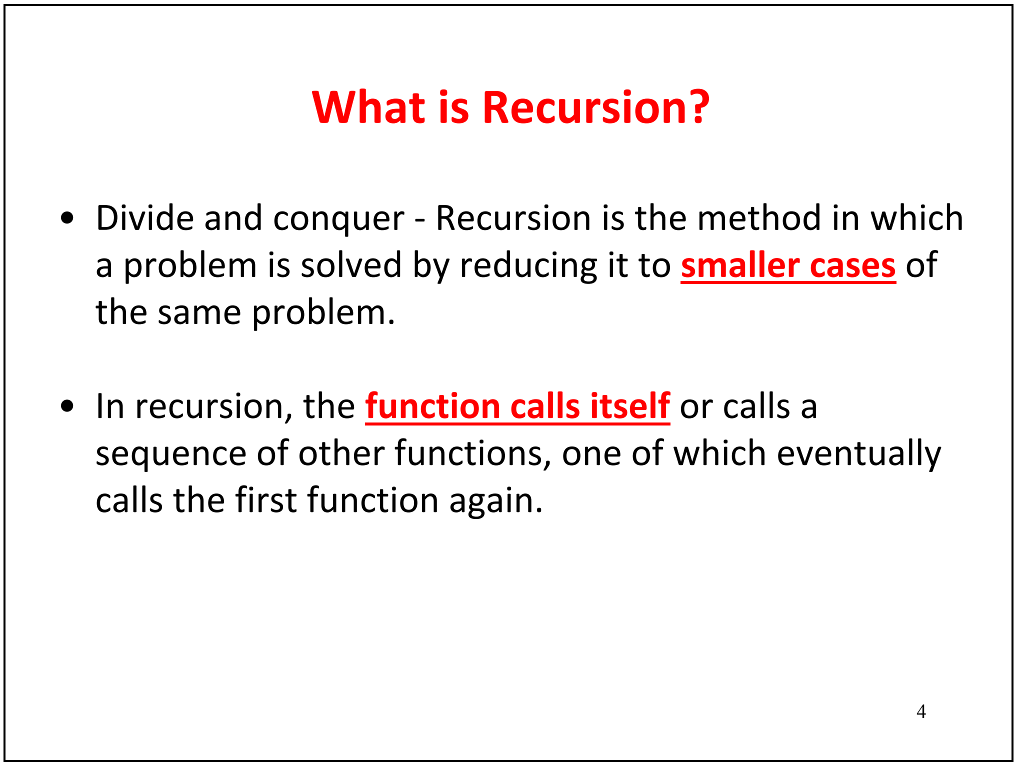
```
main(      )
{
   int x, y;
   float b;
   ......
   x = fn1(y);
   :
   fn2(b);
}
```

```
int fn1(int v1)
{
   int result;
   ......
   return result;
}
```

```
void fn2(float v2)
{
   :
}
```

What is Recursion?

1. In C, functions have the following properties:
   a) A function, when called, will accomplish a certain job.
   b) When a function **F()** is called, control is transferred from the calling point to the first statement in **F()**. After the function finishes execution, the control will return back to the calling point. The next statement after the function call will be executed.
   c) Each call to a function has its own set of values for the actual arguments and local variables.
2. These properties are important for the understanding of *recursive functions*.

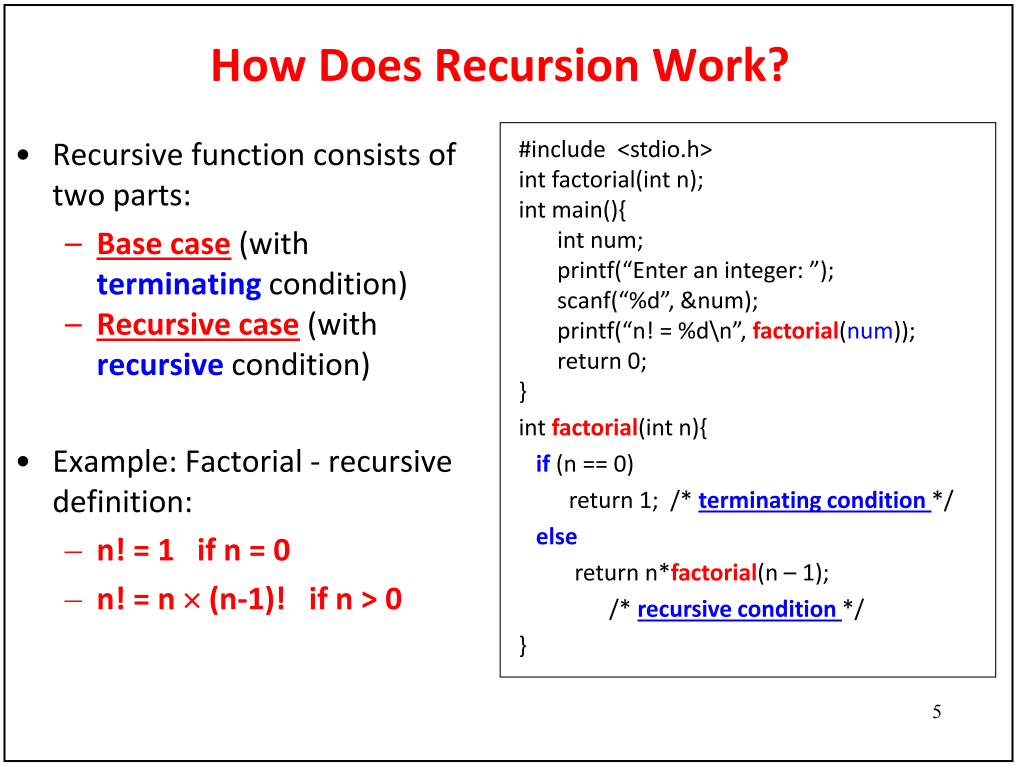


What is Recursion?

- Divide and conquer - Recursion is the method in which a problem is solved by reducing it to **smaller cases** of the same problem.
- In recursion, the **function calls itself** or calls a sequence of other functions, one of which eventually calls the first function again.

What is Recursion?

1. Recursion is a divide and conquer method in which a problem is solved by reducing it to smaller cases of the same problem.
2. In recursion, the function calls itself or calls a sequence of other functions, one of which eventually calls the first function again.



How Does Recursion Work?

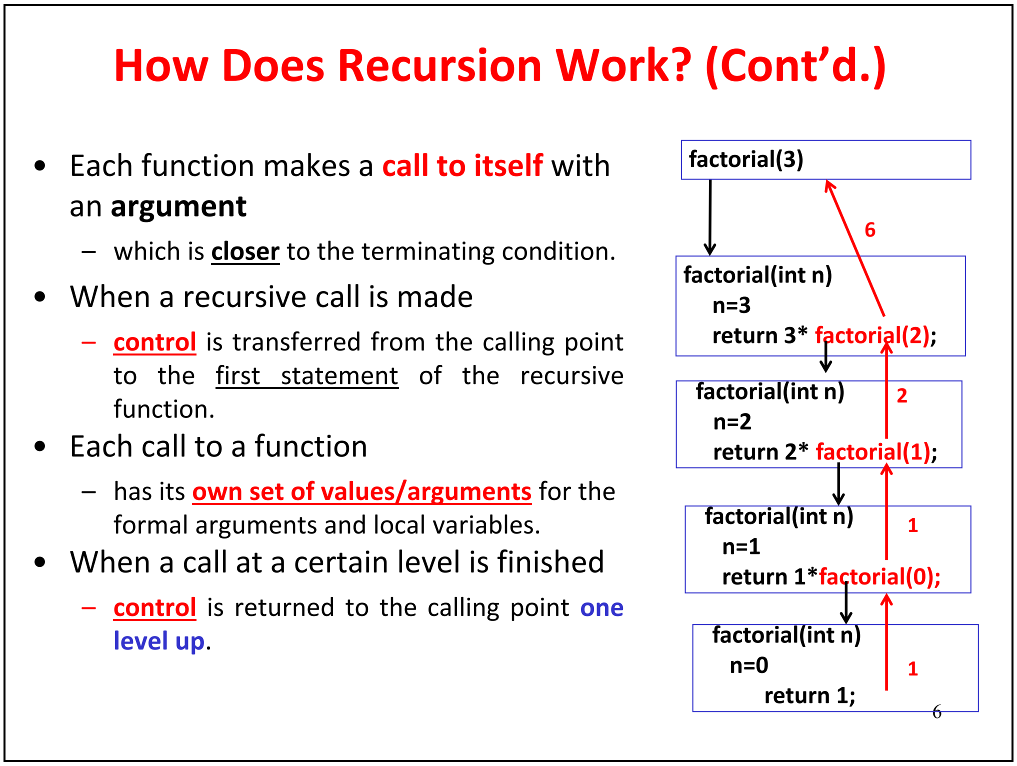
- Recursive function consists of two parts:
  - **Base case** (with **terminating** condition)
  - **Recursive case** (with **recursive** condition)

- Example: Factorial - recursive definition:
  - **n! = 1 if n = 0**
  - **n! = n × (n-1)! if n > 0**

```
#include  <stdio.h>
int factorial(int n);
int main(){
      int num;
      printf("Enter an integer: ");
      scanf("%d", &num);
      printf("n! = %d\n", factorial(num));
      return 0;
}
int factorial(int n){
   if (n == 0)
        return 1;  /* terminating condition */
   else
         return n*factorial(n – 1);
               /* recursive condition */
}
```

How Does Recursion Work?

1. Generally, a recursive function consists of two parts: base case for terminating condition and recursive case for recursive condition.
2. In the example, we define a recursive function for factorial:
   - n! = 1   if n = 0   [base case with terminating condition]
   - n! = n × (n-1)!   if n > 0   [recursive case with recursive condition]
3. Based on the recursive definition, the recursive code is implemented with the two cases accordingly.



How Does Recursion Work? (Cont'd.)

- Each function makes a **call to itself** with an **argument**
  - which is **closer** to the terminating condition.
- When a recursive call is made
  - **control** is transferred from the calling point to the first statement of the recursive function.
- Each call to a function
  - has its **own set of values/arguments** for the formal arguments and local variables.
- When a call at a certain level is finished
  - **control** is returned to the calling point **one level up**.

factorial(3)

factorial(int n)
n=3
return 3* factorial(2);

factorial(int n)
n=2
return 2* factorial(1);

factorial(int n)
n=1
return 1*factorial(0);

factorial(int n)
n=0
return 1;

6

2

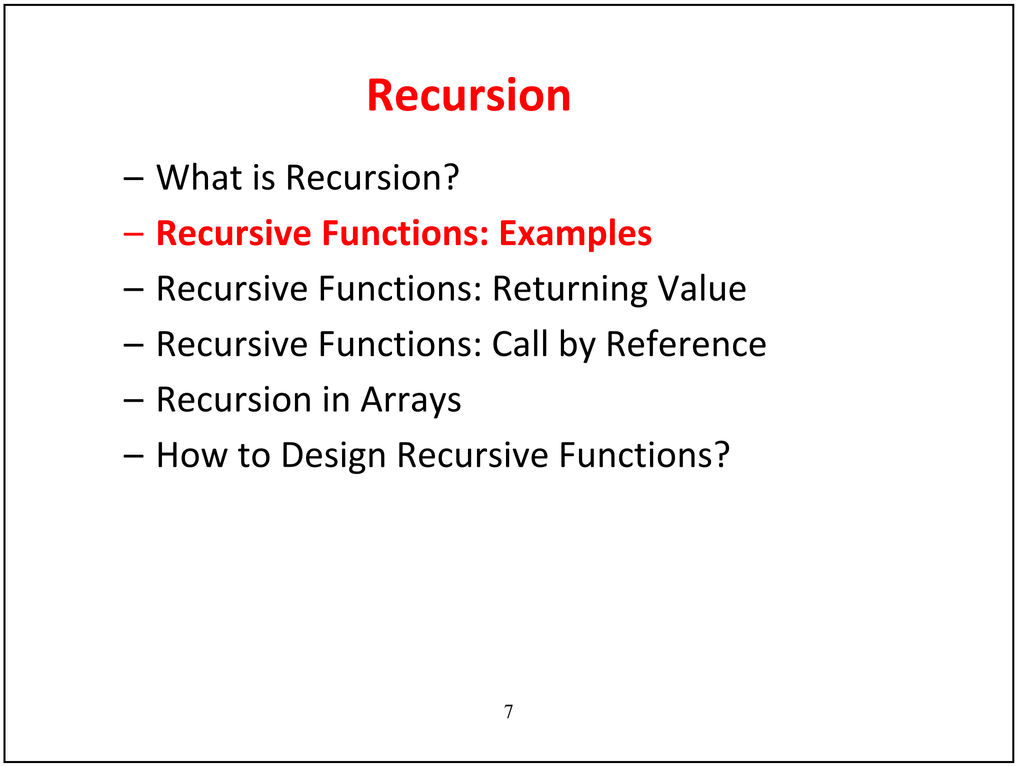
1

1

6

How Does Recursion Work?

1. A recursive function is a function that calls itself.
2. When a function calls itself, the execution of the current function is suspended, the information that it needs to continue execution is saved, and the recursive call is then evaluated.
3. Each function makes a call to itself with an argument which is closer to the terminating condition.
4. When a recursive call is made, control is transferred from the calling point to the first statement of the recursive function.
5. Each call to a function has its own set of values/arguments for the formal arguments and local variables, which is independent from the variables that are created from the previous calls to the function.
6. When a call at a certain level is finished, the control is returned to the calling point one level up. The evaluation result is passed back to the previous call, until the process is completed.

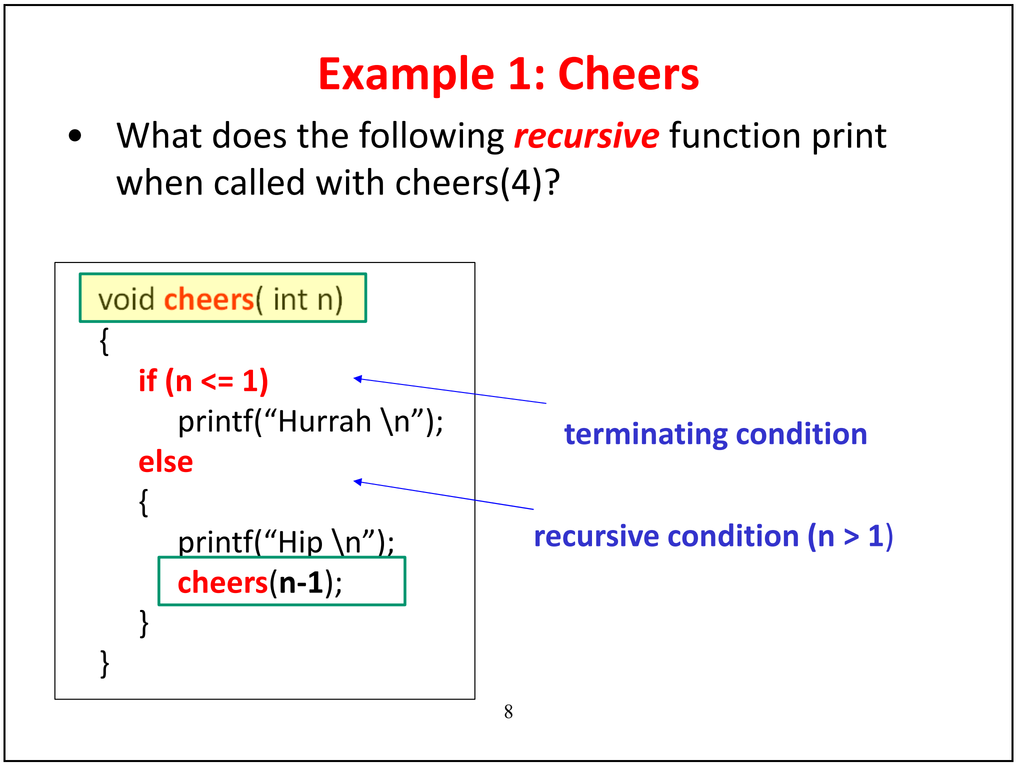


Recursion

- What is Recursion?
- **Recursive Functions: Examples**
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

Recursion

1. Here, we discuss some simple examples of recursive functions.



Example 1: Cheers

- What does the following ***recursive*** function print when called with cheers(4)?

```
void cheers( int n)
{
    if (n <= 1)
        printf(“Hurrah \n”);
    else
    {
        printf(“Hip \n”);
        cheers(n-1);
    }
}
```

terminating condition

recursive condition (n > 1)

8

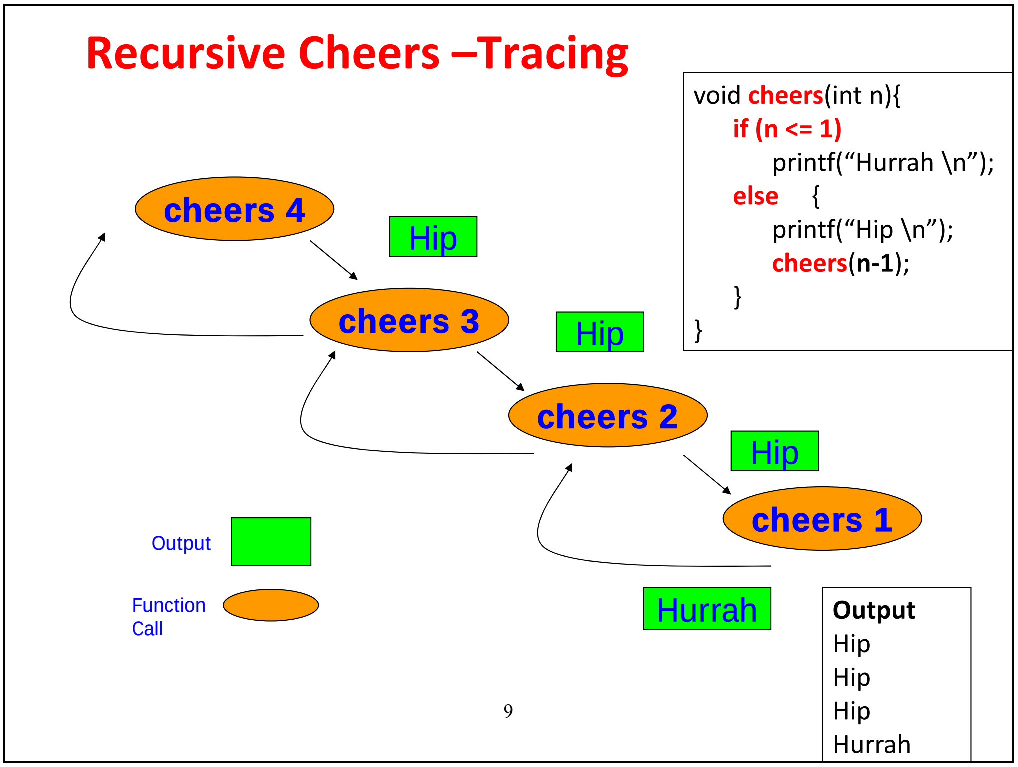
Example 1: Cheers

1. The recursive function called **cheers()** is implemented as follows:

```
void cheers(int n)
{
    if (n <= 1)                    /* terminating condition */
        printf("Hurrah \n");
    else                           /* recursive condition */
    {
        printf("Hip \n");
        cheers(n-1);
    }
}
```

2. The function **cheers()** accepts an integer parameter **n**. There are two parts of coding in the function. The first part is the code that handles the case when the terminating condition occurs. This will happen when the parameter **n <= 1**. In this case, the character string **"Hurrah"** will be printed on the screen. The second part is the code that handles the case when the recursive condition occurs. This will happen when **n** is greater than 1. In this case, the character string **"Hip"** will be printed on the screen and the function **cheers()** will be called again with the parameter **n** reduced by 1.

3. The recursive condition is to ensure the recursive function will be called again with updated parameter getting closer to the terminating condition.



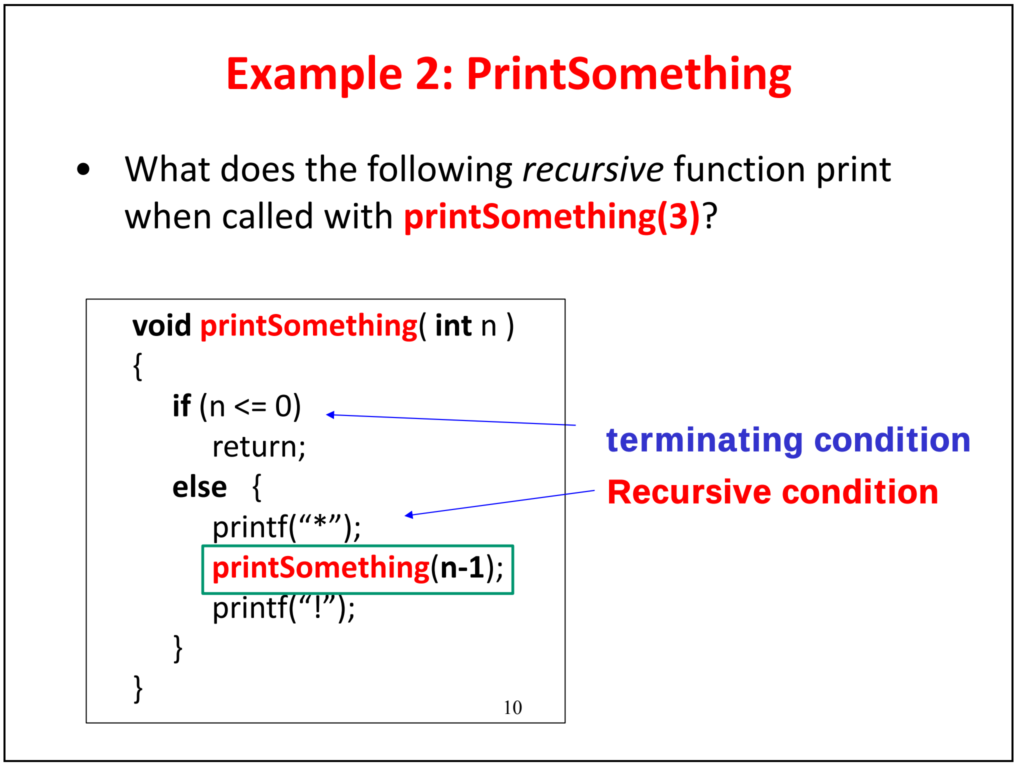
Recursive Cheers –Tracing

```
void cheers(int n){
     if (n <= 1)
          printf(“Hurrah \n”);
     else    {
          printf(“Hip \n”);
          cheers(n-1);
     }
}
```

```
Output
Hip
Hip
Hip
Hurrah
```

Recursive Cheers: Tracing

1. The program tracing for **cheers(4)** is illustrated here.
2. When the parameter **n**=4, the code for recursive condition is then executed. The string **"Hip"** will be printed on the screen, and the function **cheers(3)** will be called with **n**=3. This will be repeated to print the string **"Hip"** for two more times until the function **cheers(1)** is called with **n** is reduced to 1. In this case, the string **"Hurrah"** will be printed on the screen. When **cheers(1)** has completed execution. The control is returned to the previous function call executing **cheers(2)**. It will in turn complete the execution, and the control returns to the previous calling function **cheers(3)**, etc.
3. The program stops execution when **cheers(4)** has completed the execution.



Example 2: PrintSomething

- What does the following *recursive* function print when called with **printSomething(3)**?

```
void printSomething( int n )
{
    if (n <= 0)
        return;
    else   {
        printf(“*”);
        printSomething(n-1);
        printf(“!”);
    }
}
```

terminating condition

Recursive condition

10

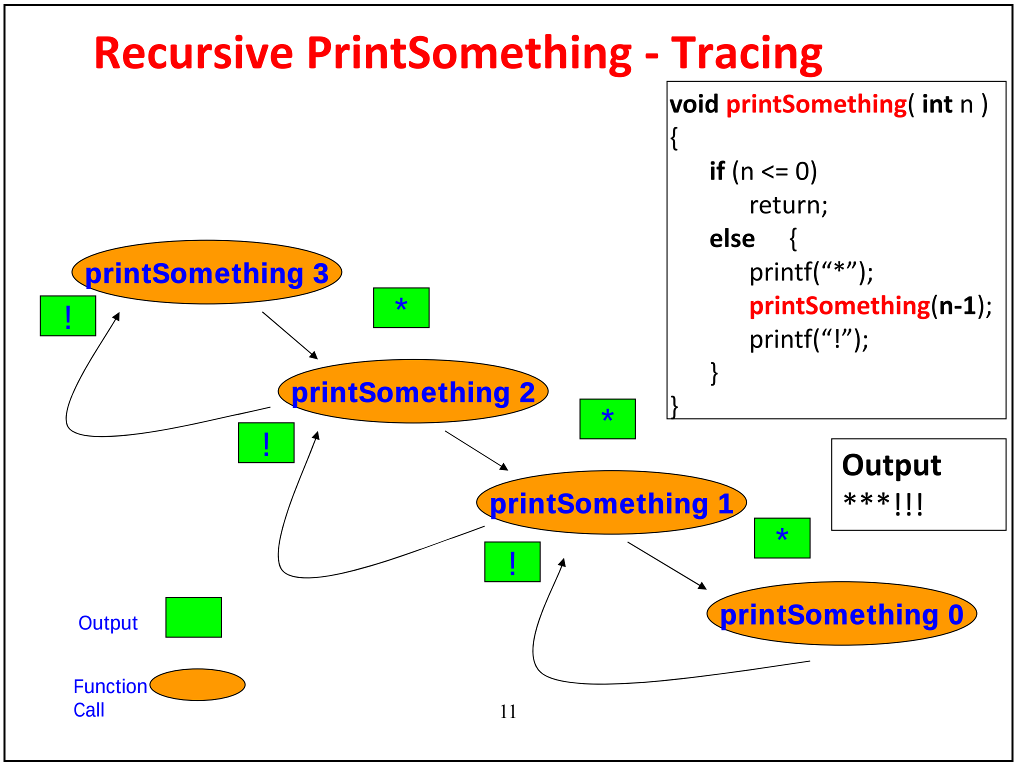
Example 2: printSomething

1. The recursive function called **printSomething()** is implemented as follows:

```
void printSomething(int n)
{
    if (n <= 0)                     /* terminating condition */
        return;
    else                            /* recursive condition */
    {
        printf("*");
        printSomething(n-1);
        printf("!");
    }
}
```

2. The function **printSomething()** accepts an integer parameter **n**. There are two parts of coding in the function. The first part is the code that handles the case when the terminating condition occurs. This will happen when the parameter **n <= 0**. In this case, the control is returned. The second part is the code that handles the case when the recursive condition occurs. This will happen when **n** is greater than 0. In this case, the character string **"*"** will be printed on the screen and the function **printSomething()** will be called again with the parameter **n** reduced by 1. After that, the character string **"!"** will be printed on the screen.

3. Note that when the function completes execution, the control will be returned to the calling function and continue execute the statement(s) after that function call.



Recursive PrintSomething - Tracing

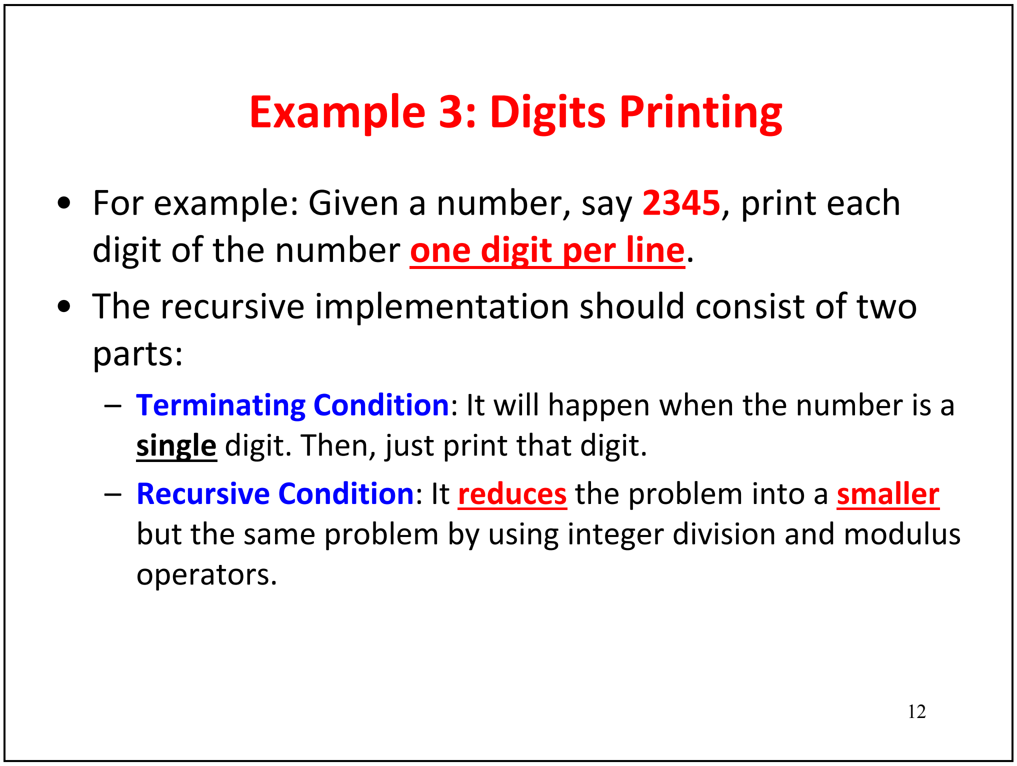
```
void printSomething( int n )
{
    if (n <= 0)
        return;
    else   {
        printf(“*”);
        printSomething(n-1);
        printf(“!”);
    }
}
```

Output

```
***!!!
```

Recursive Cheers: Tracing

1. The program tracing for **printSomething(3)** is illustrated here.
2. When the parameter **n**=3, the code for recursive condition is then executed. The string **"*"** will be printed on the screen, and the function **printSomething(2)** will be called with **n**=2. This will be repeated to print the **"*"** for two more times until the function **printSomething(0)** is called with **n** is reduced to 0. In this case, the **return** statement will return the control to the previous function call **printSomething(1)**. In **printSomething(1)**, it will continue its execution with the next statement **printf("!");** and the string **"!"** will be printed. After that, the function has completed execution and the control is returned to the previous function call executing **printSomething(2)**. It will in turn complete the execution and print the string **"!"**, and the control will be returned to the previous calling function **printSomething(3)**.
3. The function **printSomething(3)** prints the string **"!"** and returns the control to the calling **main()** function.



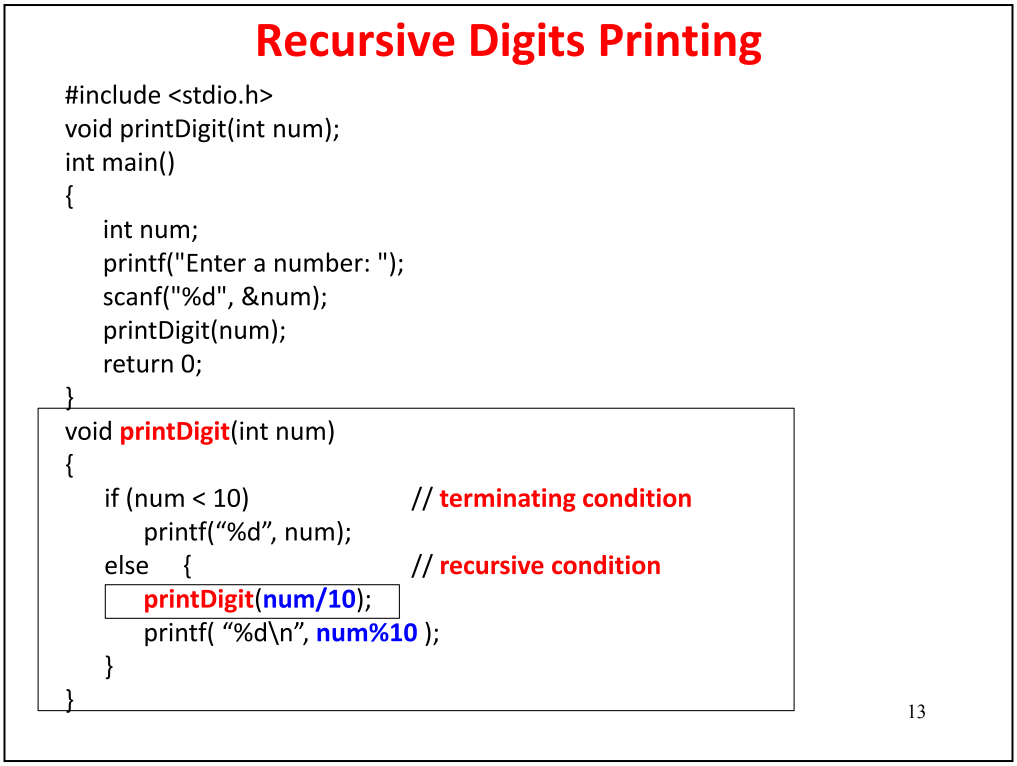
Example 3: Digits Printing

- For example: Given a number, say **2345**, print each digit of the number **one digit per line**.
- The recursive implementation should consist of two parts:
  - **Terminating Condition**: It will happen when the number is a **single** digit. Then, just print that digit.
  - **Recursive Condition**: It **reduces** the problem into a **smaller** but the same problem by using integer division and modulus operators.

12

Example 3: Printing Digits

1. In this example, we write a recursive function to print the digits of an integer number one digit per line. To write the code, we need to divide the code into two parts, one for the terminating condition, and the other for the recursive condition.
2. For the **terminating condition**, it will happen when the number is a single digit. In this case, the digit is just printed on the screen.
3. For the **recursive condition**, it will reduce the problem into a smaller but the same problem by calling the function itself again. Here, we will need to use the division operator and modulus operator in order to obtain the quotient and remainder of a **number**. The division operator (e.g. quotient = **number**/10) will give the quotient of the number, while the modulus operator (e.g. digit = **number**%10) will give the remainder digit of the number.



Recursive Digits Printing

```
#include <stdio.h>
void printDigit(int num);
int main()
{
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printDigit(num);
    return 0;
}
void printDigit(int num)
{
    if (num < 10)                  // terminating condition
        printf(“%d”, num);
    else   {                       // recursive condition
        printDigit(num/10);
        printf( “%d\n”, num%10 );
    }
}
```

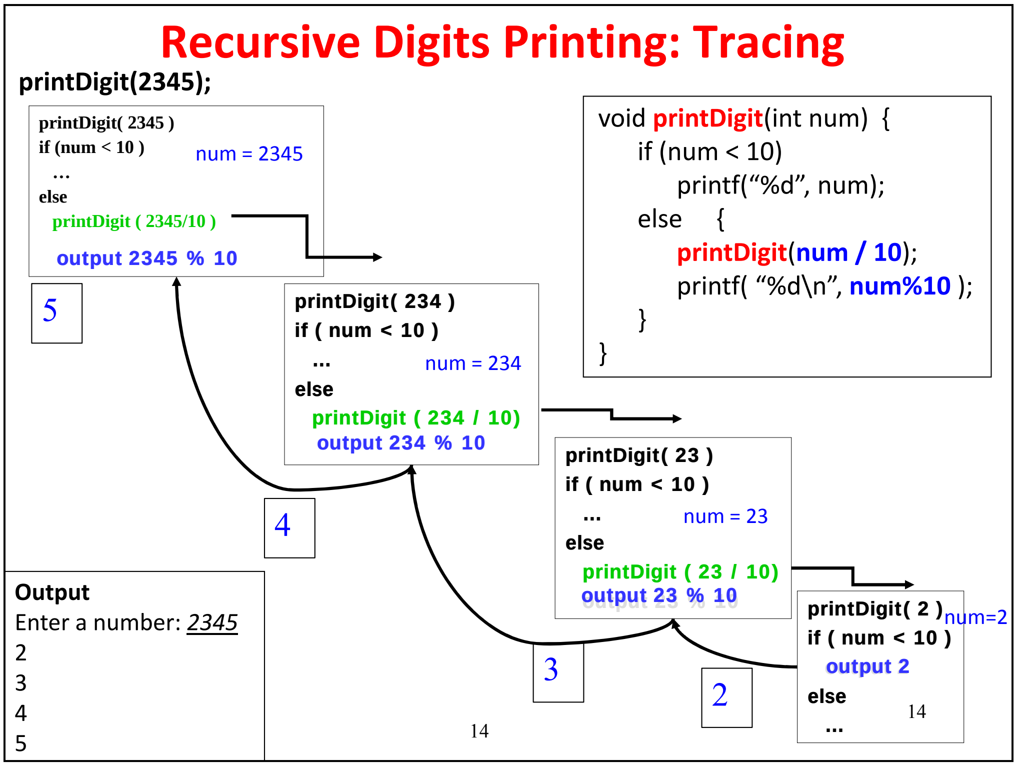
13

Recursive Digits Printing

1. The recursive function **printDigit()** is implemented as follows:

```
void printDigit(int num)
{
    if (num < 10)                 // terminating condition
        printf("%d", num);
    else     {                    // recursive condition
        printDigit(num/10);
        printf("%d\n", num%10);
    }
}
```

2. In the recursive function **printDigit()**, it receives an integer argument **num** from the calling function.
3. The terminating condition is defined when the argument **num** is a single digit.
4. The function calls itself for the recursive condition and each time the argument **num** will be updated to get the quotient of **num** by using the division operator **num/10**. The corresponding digit of the input number is obtained through **num%10** which will be printed on the screen after the recursive call.
5. The recursive condition will be called repeatedly until the terminating condition is reached. In which case, the final digit of the input number will be printed on the screen.



Recursive Digits Printing: Tracing

printDigit(2345);

```
void printDigit(int num) {
     if (num < 10)
          printf("%d", num);
     else     {
          printDigit(num / 10);
          printf( "%d\n", num%10 );
     }
}
```

```
printDigit( 2345 )
if (num < 10 )          num = 2345
  …
else
  printDigit ( 2345/10 )
  output 2345 % 10
```

```
printDigit( 234 )
if ( num < 10 )
  …              num = 234
else
  printDigit ( 234 / 10)
  output 234 % 10
```

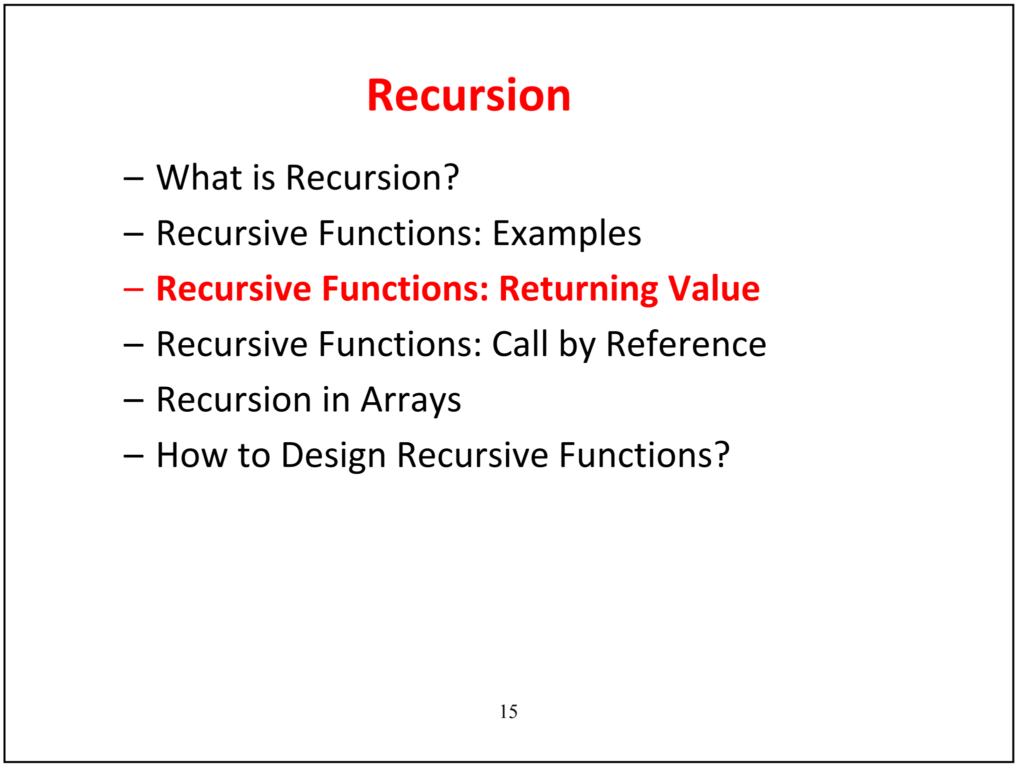
```
printDigit( 23 )
if ( num < 10 )
  …              num = 23
else
  printDigit ( 23 / 10)
  output 23 % 10
```

```
printDigit( 2 )num=2
if ( num < 10 )
  output 2
else
  …
```

```
Output
Enter a number: 2345
2
3
4
5
```

Recursive Digits Printing: Tracing

1. The program tracing for **printDigit(1234)** is illustrated here. When the parameter **num**=1234, the code for recursive condition is then executed. The function **printDigit(num/10)** will be called with the quotient **num**=2345/10=**234**. This will be repeated to call the function again with **printDigit(23)** with **num**=234/10=**23**.
2. Similarly, the function **printDigit(2)** with **num**=23/10=**2** is called again. When this function is called, it reaches the terminating condition with only a single digit. The value 2 is printed to the screen. The control is then returned to the previous calling function **printDigit(23)**. The function continues execution and prints the digit 3 (23%10 = 3) to the screen. Similarly, the digits 4 and 5 will then be printed when the control is returned to the previous calling functions.
3. In this program, it is important to observe that the execution order of the statements when the function has completed execution. The control will be returned to the previous calling function and continues execution.
4. It is also important to observe that the **printf()** statement is placed after the recursive function call in this program. And also observe that each call to a function has its own set of values for the actual arguments and local variables.



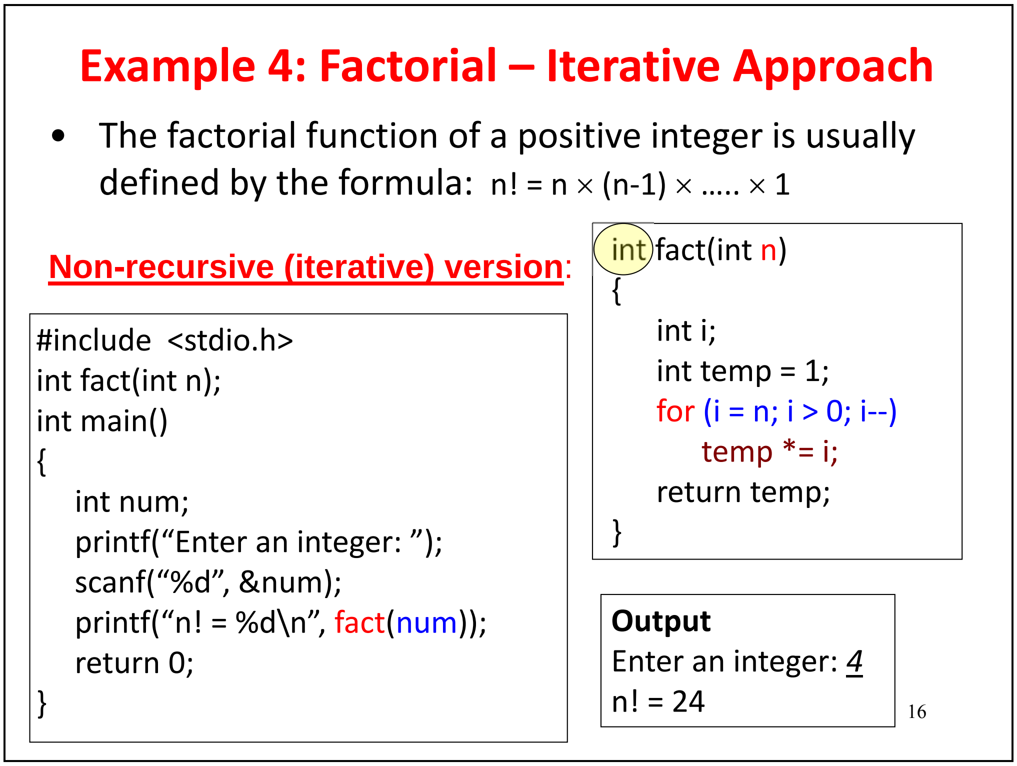
Recursion

- What is Recursion?
- Recursive Functions: Examples
- **Recursive Functions: Returning Value**
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

15

Recursion

1. Here, we discuss examples of recursive functions that return values.



Example 4: Factorial – Iterative Approach

- The factorial function of a positive integer is usually defined by the formula: n! = n × (n-1) × ..... × 1

Non-recursive (iterative) version:

```
#include  <stdio.h>
int fact(int n);
int main()
{
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    printf("n! = %d\n", fact(num));
    return 0;
}
```

```
int fact(int n)
{
    int i;
    int temp = 1;
    for (i = n; i > 0; i--)
        temp *= i;
    return temp;
}
```

Output

```
Enter an integer: 4
n! = 24
```

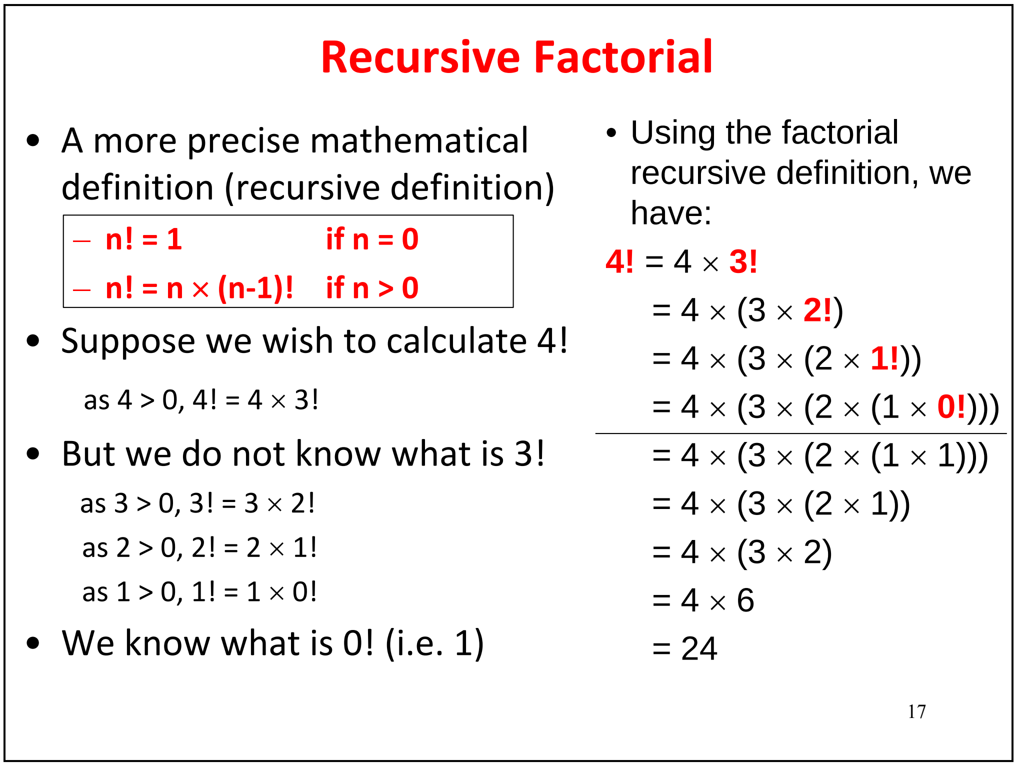
16

Example 4: Factorial – Iterative Approach

1. Consider the problem of finding the factorial of a number **n**: **n! = n * (n-1) * (n-2) * … * 1**
2. The program implements the factorial function using non-recursive approach with a **for** loop iteratively.

```
int fact(int n)
{
    int i;
    int temp = 1;
    for (i = n; i > 0; i--)
        temp *= i;
    return temp;
}
```

3. The function **fact()** receives the argument from the calling function. In the **for** loop, the factorial of the input argument is calculated. The result is then passed to the calling function. For **fact(4)**, it will return the value of 24 to the calling **main()** function.
4. It is quite straightforward to implement the factorial function using non-recursive approach.



Recursive Factorial

- A more precise mathematical definition (recursive definition)
  - n! = 1 if n = 0
  - n! = n × (n-1)! if n > 0
- Suppose we wish to calculate 4!

  as 4 > 0, 4! = 4 × 3!
- But we do not know what is 3!

  as 3 > 0, 3! = 3 × 2!

  as 2 > 0, 2! = 2 × 1!

  as 1 > 0, 1! = 1 × 0!
- We know what is 0! (i.e. 1)

- Using the factorial recursive definition, we have:

4! = 4 × 3!

= 4 × (3 × 2!)

= 4 × (3 × (2 × 1!))

= 4 × (3 × (2 × (1 × 0!)))

= 4 × (3 × (2 × (1 × 1)))

= 4 × (3 × (2 × 1))

= 4 × (3 × 2)

= 4 × 6

= 24

17

Recursive Factorial

1. Mathematically, a recursive definition of the factorial function can be defined as:

   0! = 1

   n! = n(n-1)!
2. That is, **n!** is defined in terms of **(n-1)!**.
3. An example tracing of the recursive factorial definition using 4! is illustrated below:

   4! = 4 × **3!**

   = 4 × (3 × **2!**)

   = 4 × (3 × (2 × **1!**))

   = 4 × (3 × (2 × (1 × **0!**)))

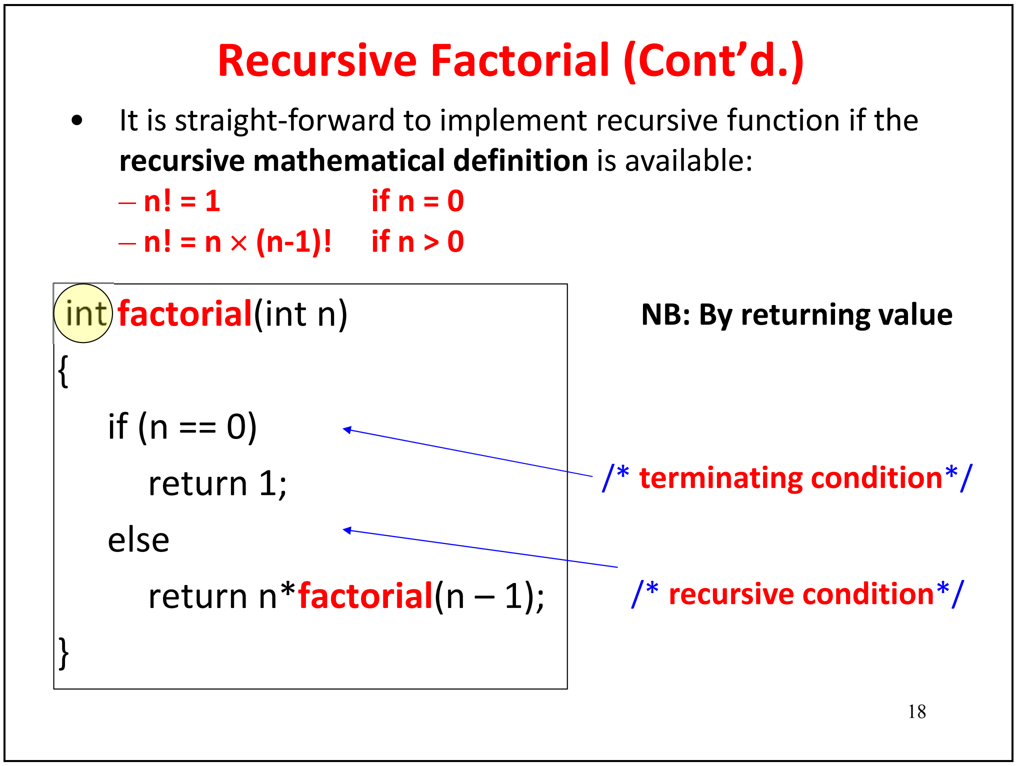
   = 4 × (3 × (2 × (1 × 1)))

   = 4 × (3 × (2 × 1))

   = 4 × (3 × 2)

   = 4 × 6

   = 24
4. Based on this recursive definition, we can write the recursive function.



Recursive Factorial (Cont'd.)

- It is straight-forward to implement recursive function if the **recursive mathematical definition** is available:
  - **n! = 1** **if n = 0**
  - **n! = n × (n-1)!** **if n > 0**

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

NB: By returning value

/* **terminating condition***/

/* **recursive condition***/

18

<u>Recursive Factorial</u>

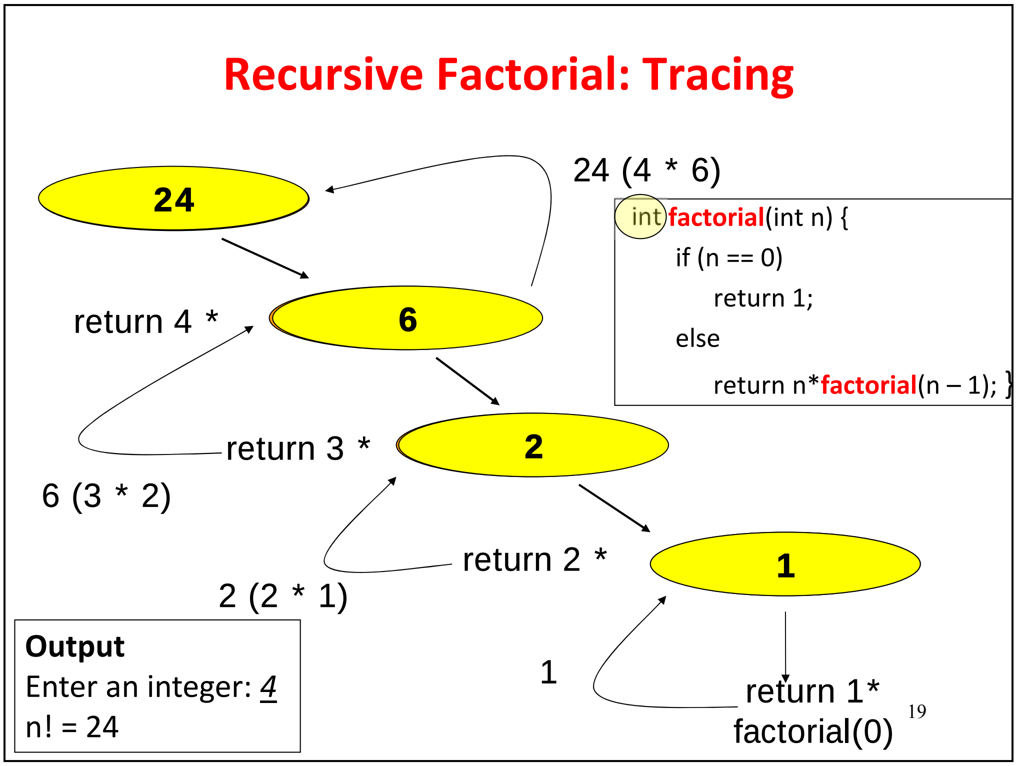
1. The recursive factorial function can be implemented according to the mathematical definition as below:

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

2. To evaluate the function **factorial()**, assume that the user enters 4 for the input value, the result should be 4 factorial (4!) which equals to 4 * 3 * 2 * 1, or 24. The **factorial()** function is recursive, as it calls itself in the **return** statement:

```
return n * factorial(n-1);
```

3. There are two **return** statements in the function:
   - The first **return 1** statement does not call the function **factorial()**. This is the terminating condition, where the recursion will stop.
   - The second return call is that for each new call to **factorial()** for recursive condition, the value **n** gets smaller by 1 each time, and eventually equals to 0. When **n** equals to 0, i.e. **0!**, the terminating condition is reached, and the function returns the value 1.



Recursive Factorial: Tracing

24 (4 * 6)

24

return 4 *

6

return 3 *

2

6 (3 * 2)

return 2 *

1

2 (2 * 1)

1

return 1*
factorial(0)

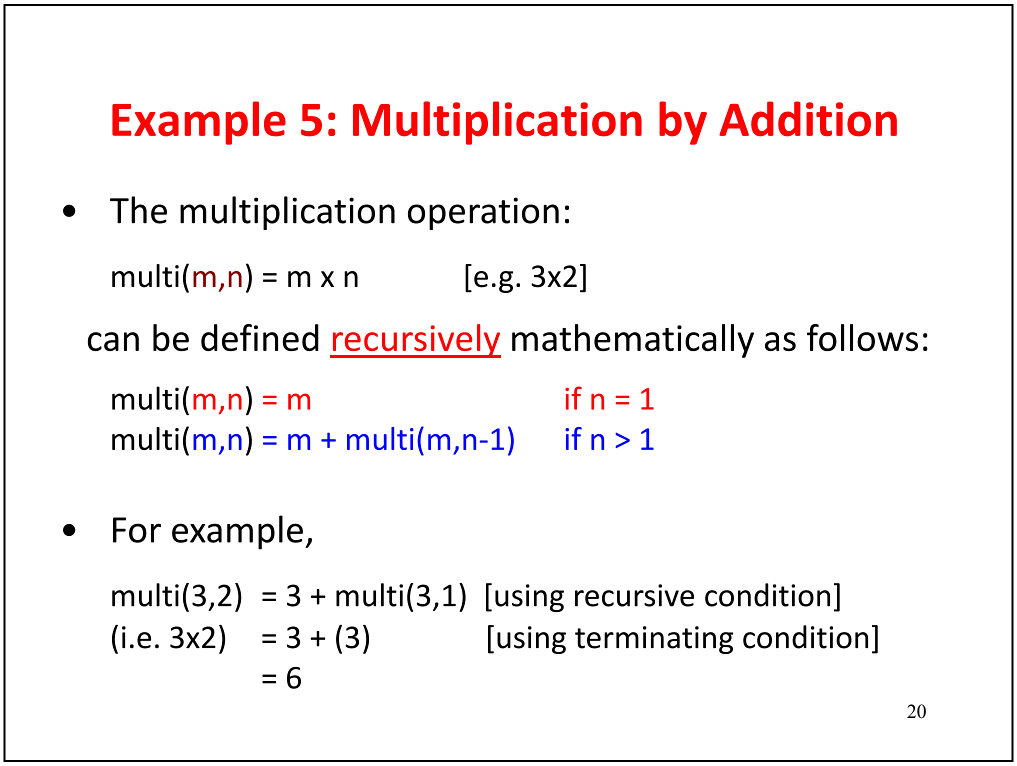
```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n*factorial(n – 1); }
```

Output
Enter an integer: _4_
n! = 24

19

Recursive Factorial: Tracing

1. The program tracing for the function call **factorial(4)** is illustrated.
2. The values associated with the parameter **n** and the **return** values for each level of recursion are also shown.
3. Note that each level of the recursive function has its own set of variables. For each level of recursion, a recursive call invokes **factorial()**. When each recursive **factorial()** function terminates, it returns a value to the previous level (from which it was called). When the last **factorial()** function is called, it returns the value 1, as **0!** equals to 1.



Example 5: Multiplication by Addition

- The multiplication operation:

  multi(m,n) = m x n [e.g. 3x2]

can be defined recursively mathematically as follows:

multi(m,n) = m if n = 1
multi(m,n) = m + multi(m,n-1) if n > 1

- For example,

  multi(3,2) = 3 + multi(3,1) [using recursive condition]
  (i.e. 3x2) = 3 + (3) [using terminating condition]
  = 6

20

Example 5: Recursive Integer Multiplication

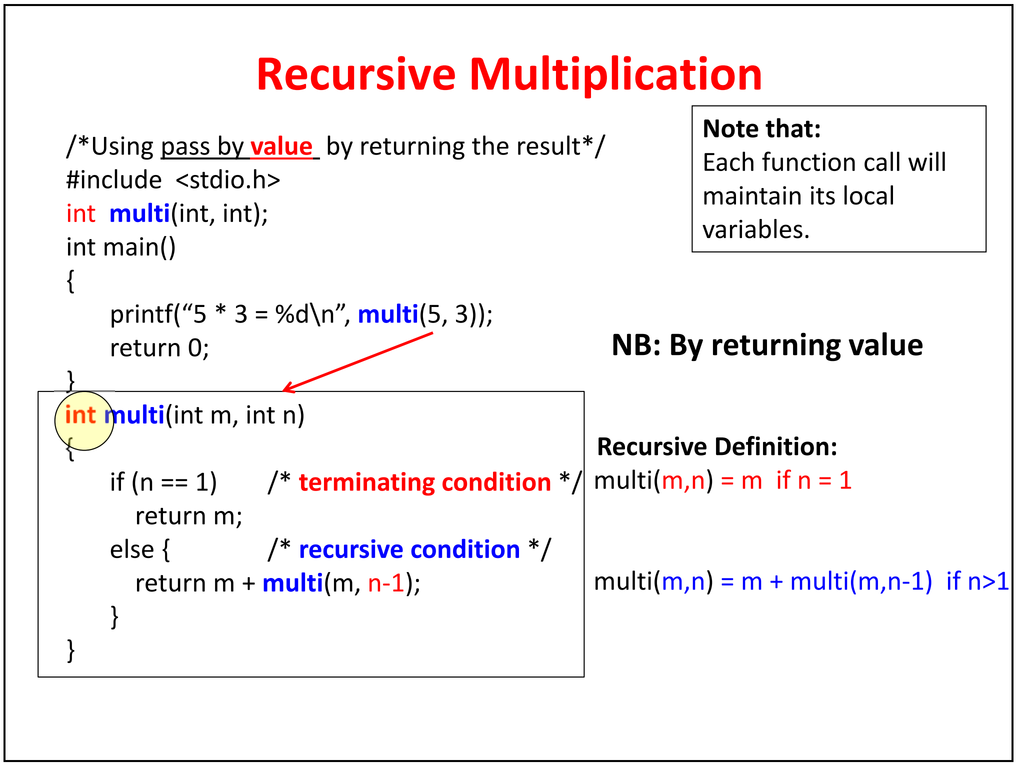
1. Consider another example that performs multiplication using addition operation. Assume **m** and **n** are positive integers, the multiplication function can be defined mathematically as:

   m * n = m if n=1

   m * n = m + m*(n-1) if n>1

2. Therefore, we can define the recursive function **multi()** as:
   multi(m,n) = m if n=1
   multi(m,n) = m + multi(m,n-1) if n>1

3. For example:
   multi(3,2) = 3 + **multi(3,1)** [using recursive condition]
   = 3 + (3) [using terminating condition]
   = 6



Recursive Multiplication

Note that:
Each function call will maintain its local variables.

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
      printf("5 * 3 = %d\n", multi(5, 3));
      return 0;
}
int multi(int m, int n)
{
      if (n == 1)       /* terminating condition */
         return m;
      else {            /* recursive condition */
         return m + multi(m, n-1);
      }
}
```

NB: By returning value

Recursive Definition:

multi(m,n) = m if n = 1

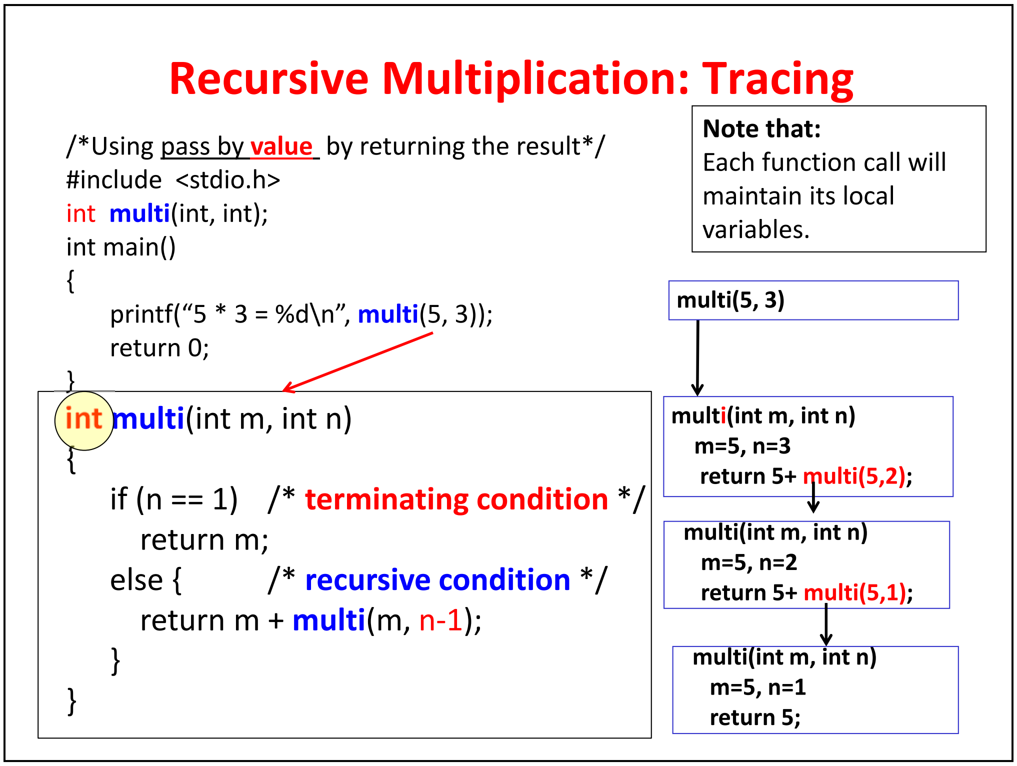
multi(m,n) = m + multi(m,n-1) if n>1

Recursive Multiplication: by Returning Value

1. In the program, it implements a recursive integer multiplication function using call by value. In the function **multi()**, it receives two integer arguments, **m** and **n**, from the calling function. The function calls itself with the following **return** statement:

   return m + multi(m, n-1);

2. For each new function call, a new set of parameters is created and used. The value **n** also gets smaller, and eventually equal to 1, where the terminating condition is reached.



Recursive Multiplication: Tracing

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
      printf("5 * 3 = %d\n", multi(5, 3));
      return 0;
}
int multi(int m, int n)
{
      if (n == 1)    /* terminating condition */
         return m;
      else {           /* recursive condition */
         return m + multi(m, n-1);
      }
}
```

Note that:
Each function call will maintain its local variables.

multi(5, 3)

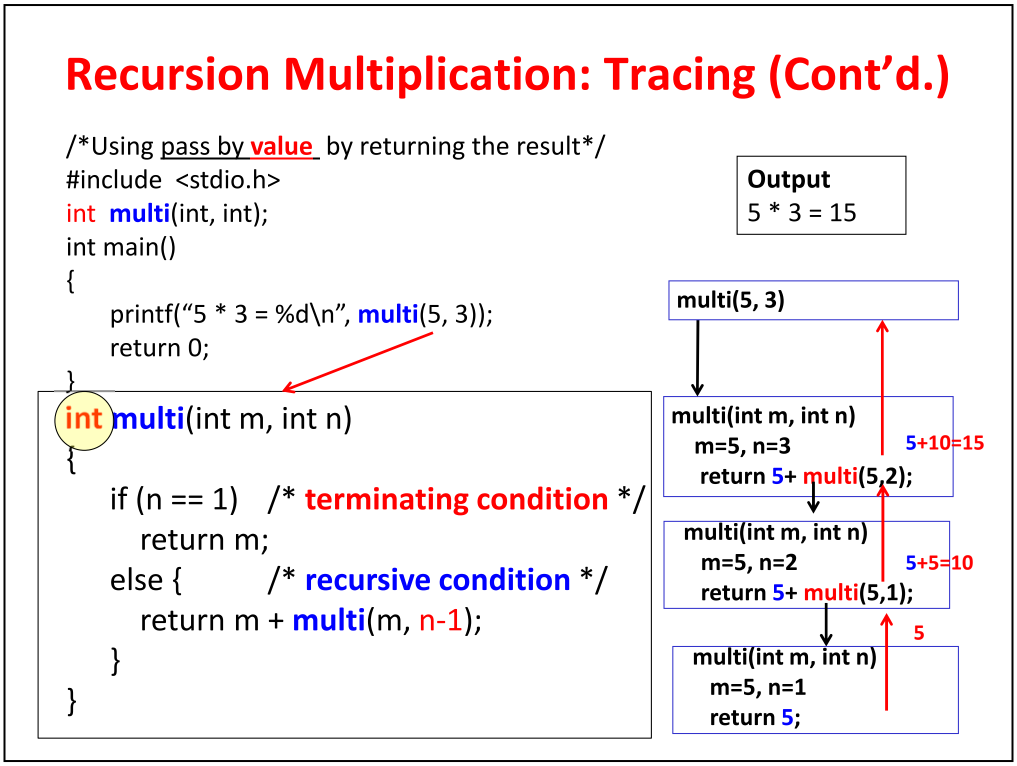
multi(int m, int n)
m=5, n=3
return 5+ multi(5,2);

multi(int m, int n)
m=5, n=2
return 5+ multi(5,1);

multi(int m, int n)
m=5, n=1
return 5;

Recursive Multiplication: Tracing

1. When the function **multi(5, 3)** is called from the **main()** function, the code under the recursive condition is executed as **n** is not equal to 1.
2. The statement **return m + multi(m, n-1);** will be executed and call the function **multi(5, 2)** with **n** reduced to 2.
3. This will in turn execute the function call **multi(5, 1)**. When the function **multi(5, 1)** is executed, the code under the terminating condition will be executed. The value 5 will be returned to the calling function.



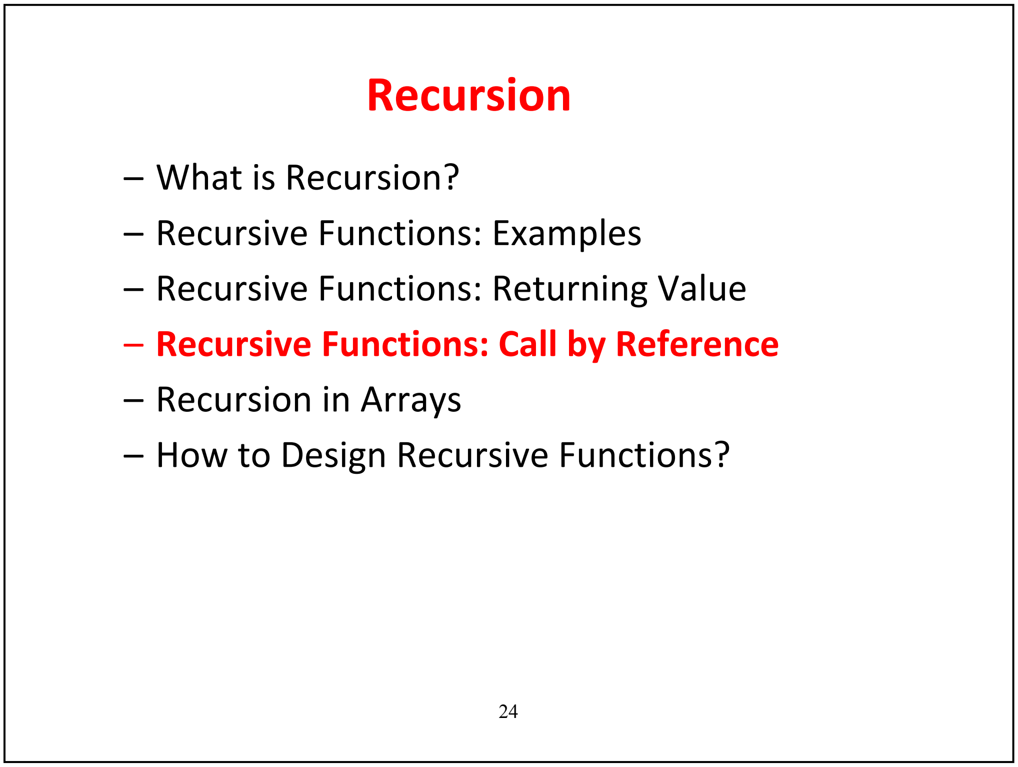
Recursion Multiplication: Tracing (Cont'd.)

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
     printf("5 * 3 = %d\n", multi(5, 3));
     return 0;
}
int multi(int m, int n)
{
     if (n == 1)    /* terminating condition */
        return m;
     else {         /* recursive condition */
        return m + multi(m, n-1);
     }
}
```

```
Output
5 * 3 = 15
```

Recursive Multiplication: Tracing

1. The previous calling function **multi(5, 2)** will receive the value 5 and add the returned value to 5 and return the sum of 10 to the previous calling function **multi(5, 3)**.
2. The function **multi(5, 3)** will add the returned value to 5 and return the sum of 15 to the calling function **main()**.
3. In the **main()** function, the returned value of 15 will then be printed to the screen.

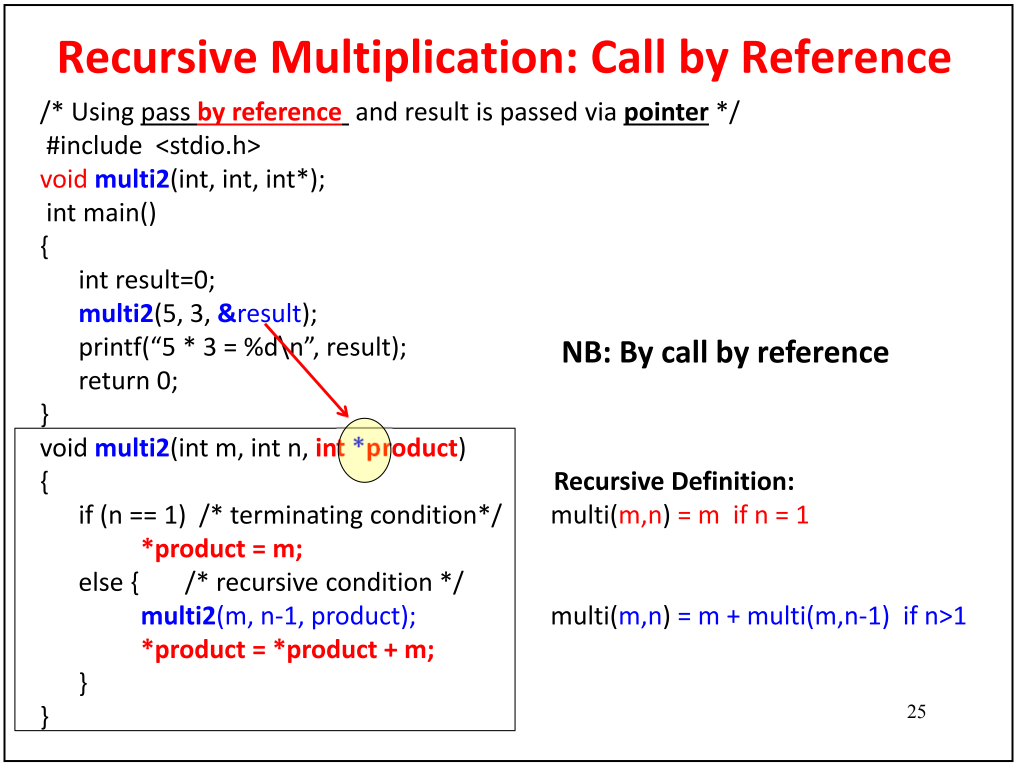


Recursion

- What is Recursion?
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- **Recursive Functions: Call by Reference**
- Recursion in Arrays
- How to Design Recursive Functions?

Recursion

1. Here, we discuss some examples of recursive functions that use call by reference.



Recursive Multiplication: Call by Reference

```
/* Using pass by reference  and result is passed via pointer */
 #include  <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result=0;
     multi2(5, 3, &result);
     printf(“5 * 3 = %d\n”, result);
     return 0;
}
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
          *product = m;
     else {      /* recursive condition */
          multi2(m, n-1, product);
          *product = *product + m;
     }
}
```

NB: By call by reference

Recursive Definition:

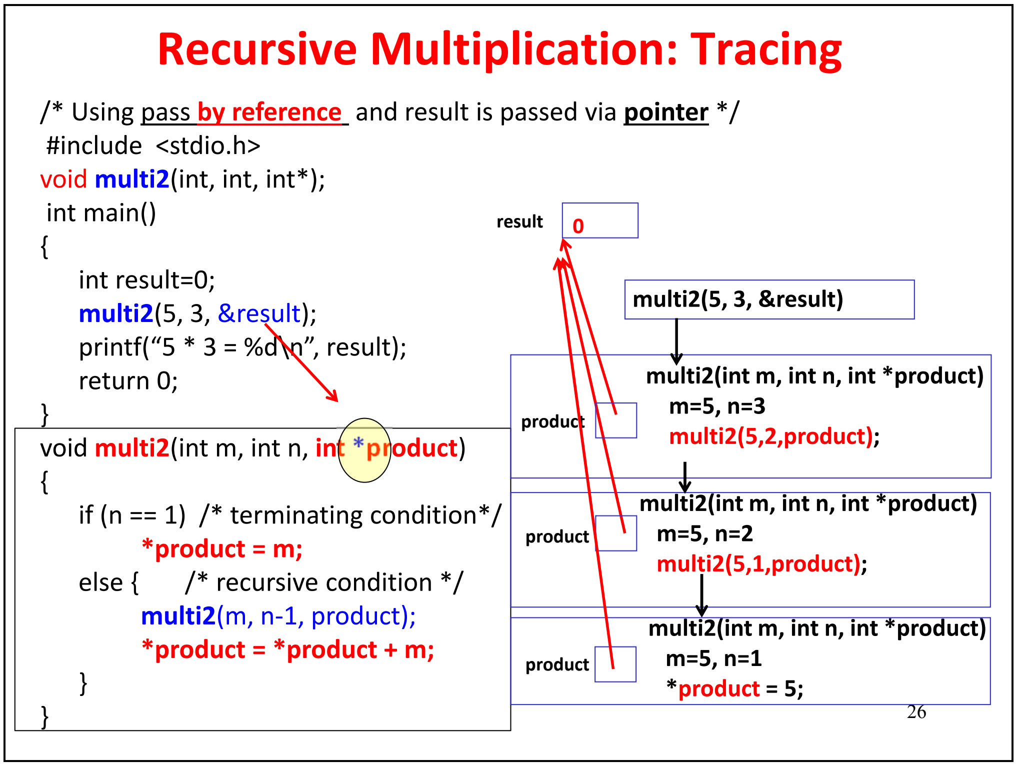
multi(m,n) = m if n = 1

multi(m,n) = m + multi(m,n-1) if n>1

25

Recursive Multiplication: Using Call by Reference

1. Instead of using call by value via returning value, we can also implement the recursive function for integer multiplication via call by reference.
2. In the recursive function **multi2()**, it uses call by reference for parameter passing. This function differs from the previous function **multi()** in that in **multi2()**, the function returns the result through **call by reference** instead of returning the result to the calling function directly through the **return** statement.
3. This is achieved through the use of the pointer parameter **product** of type **int**. The other two parameters, **m** and **n**, are specified of type **int** in the function. The terminating condition and the recursive call are the same as that in the **multi()** function.



Recursive Multiplication: Tracing

```
/* Using pass by reference  and result is passed via pointer */
 #include  <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result=0;
     multi2(5, 3, &result);
     printf(“5 * 3 = %d\n”, result);
     return 0;
}
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
          *product = m;
     else {      /* recursive condition */
          multi2(m, n-1, product);
          *product = *product + m;
     }
}
```

result 0

multi2(5, 3, &result)

product

multi2(int m, int n, int *product)
m=5, n=3
multi2(5,2,product);

product

multi2(int m, int n, int *product)
m=5, n=2
multi2(5,1,product);

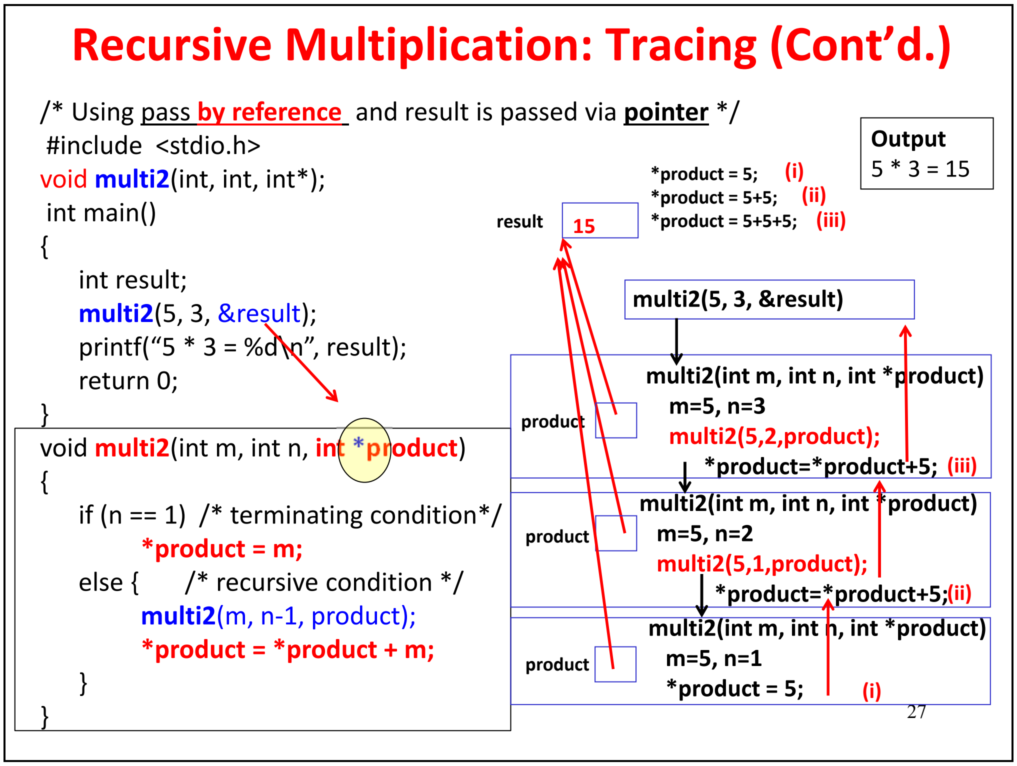
product

multi2(int m, int n, int *product)
m=5, n=1
*product = 5;

26

Recursive Multiplication: Tracing

1. In the **main()** function, the variable **result** is declared to hold the result of the multiplication operation. When it calls the function **multi2()**, it passes the address of the variable **result** via call by reference: **multi2(5, 3, &result);**
2. In the function **multi2()**, it declares a pointer parameter **product** of **int** type. By passing in the address of the variable **result** to **product**, the result will be processed and updated in the memory location of the variable **result**.



Recursive Multiplication: Tracing (Cont'd.)

```
/* Using pass by reference  and result is passed via pointer */
 #include  <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result;
     multi2(5, 3, &result);
     printf("5 * 3 = %d\n", result);
     return 0;
}
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
          *product = m;
     else {      /* recursive condition */
          multi2(m, n-1, product);
          *product = *product + m;
     }
}
```

Output
5 * 3 = 15

*product = 5; (i)
*product = 5+5; (ii)
*product = 5+5+5; (iii)

result 15

multi2(5, 3, &result)

product
multi2(int m, int n, int *product)
m=5, n=3
multi2(5,2,product);
*product=*product+5; (iii)

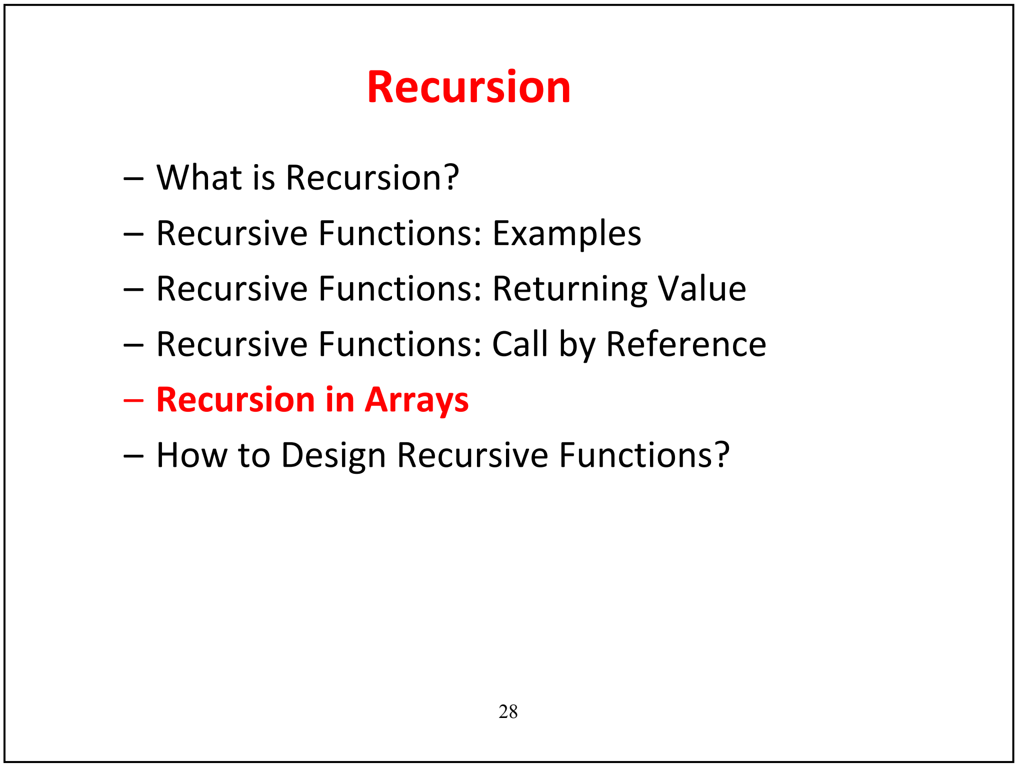
product
multi2(int m, int n, int *product)
m=5, n=2
multi2(5,1,product);
*product=*product+5;(ii)

product
multi2(int m, int n, int *product)
m=5, n=1
*product = 5; (i)

27

Recursive Multiplication: Tracing

1. The recursive operation of the function **multi2()** is similar to **multi()**. Instead of returning the result to the calling function via the **return** statement, **multi2()** will store the updated result via ***product** as follows: ***product = *product + m;**
2. That is, after each function call, the result will be updated directly via the memory location of the variable **result** in the **main()** function.



Recursion

– What is Recursion?

– Recursive Functions: Examples

– Recursive Functions: Returning Value

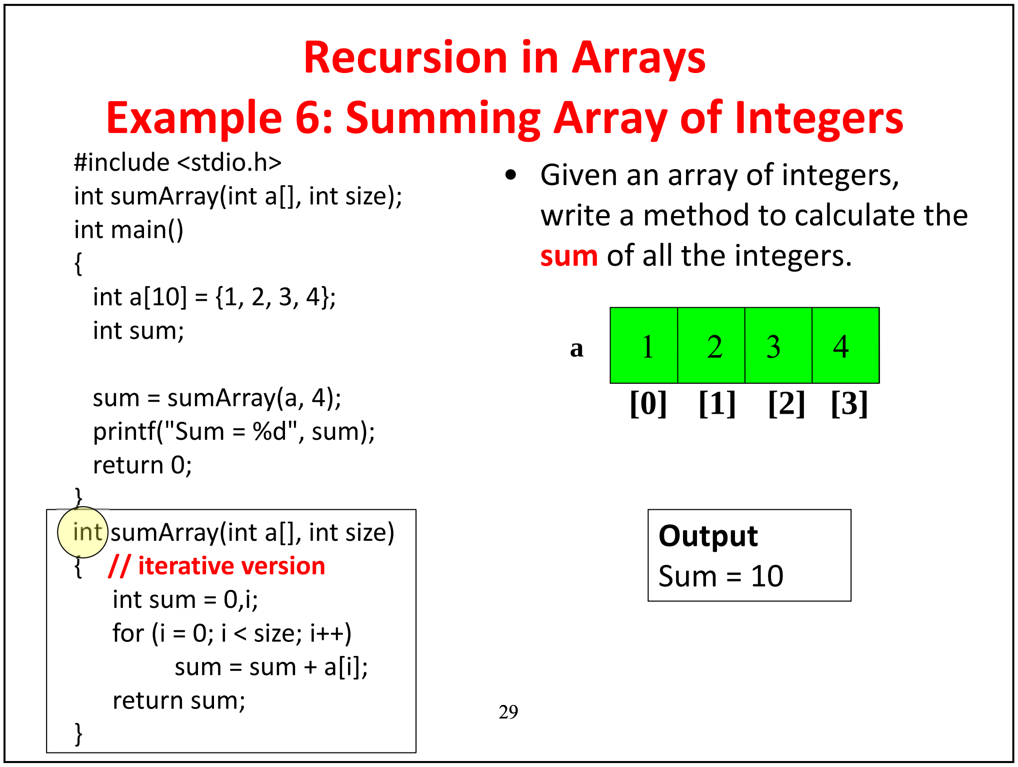
– Recursive Functions: Call by Reference

– **Recursion in Arrays**

– How to Design Recursive Functions?

Recursion

1. Here, we discuss some examples of recursive functions on arrays.



Recursion in Arrays
Example 6: Summing Array of Integers

```
#include <stdio.h>
int sumArray(int a[], int size);
int main()
{
  int a[10] = {1, 2, 3, 4};
  int sum;

  sum = sumArray(a, 4);
  printf("Sum = %d", sum);
  return 0;
}
int sumArray(int a[], int size)
{    // iterative version
     int sum = 0,i;
     for (i = 0; i < size; i++)
            sum = sum + a[i];
     return sum;
}
```

- Given an array of integers, write a method to calculate the **sum** of all the integers.

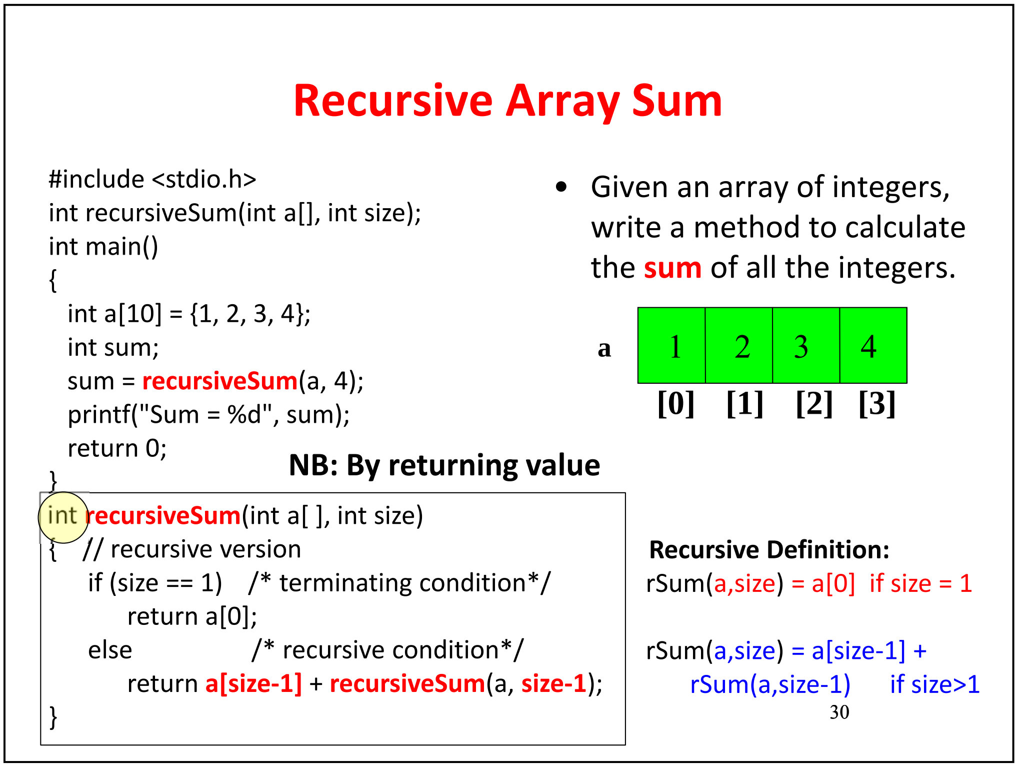
a	1	2	3	4
	[0]	[1]	[2]	[3]

Output
Sum = 10

29

Example 6: Summing Array of Integers

1. To implement a function **sumArray()** that computes the sum of all the elements in an array using non-recursive approach is quite straightforward.
2. A **for** loop can be used to traverse individual elements of array and add them up, and returns the result to the calling function.



Recursive Array Sum

```
#include <stdio.h>
int recursiveSum(int a[], int size);
int main()
{
  int a[10] = {1, 2, 3, 4};
  int sum;
  sum = recursiveSum(a, 4);
  printf("Sum = %d", sum);
  return 0;
}
```

NB: By returning value

```
int recursiveSum(int a[ ], int size)
{   // recursive version
    if (size == 1)   /* terminating condition*/
        return a[0];
    else             /* recursive condition*/
        return a[size-1] + recursiveSum(a, size-1);
}
```

- Given an array of integers, write a method to calculate the **sum** of all the integers.

a	1	2	3	4
	[0]	**[1]**	**[2]**	**[3]**

Recursive Definition:

rSum(a,size) = a[0] if size = 1

rSum(a,size) = a[size-1] + rSum(a,size-1) if size>1

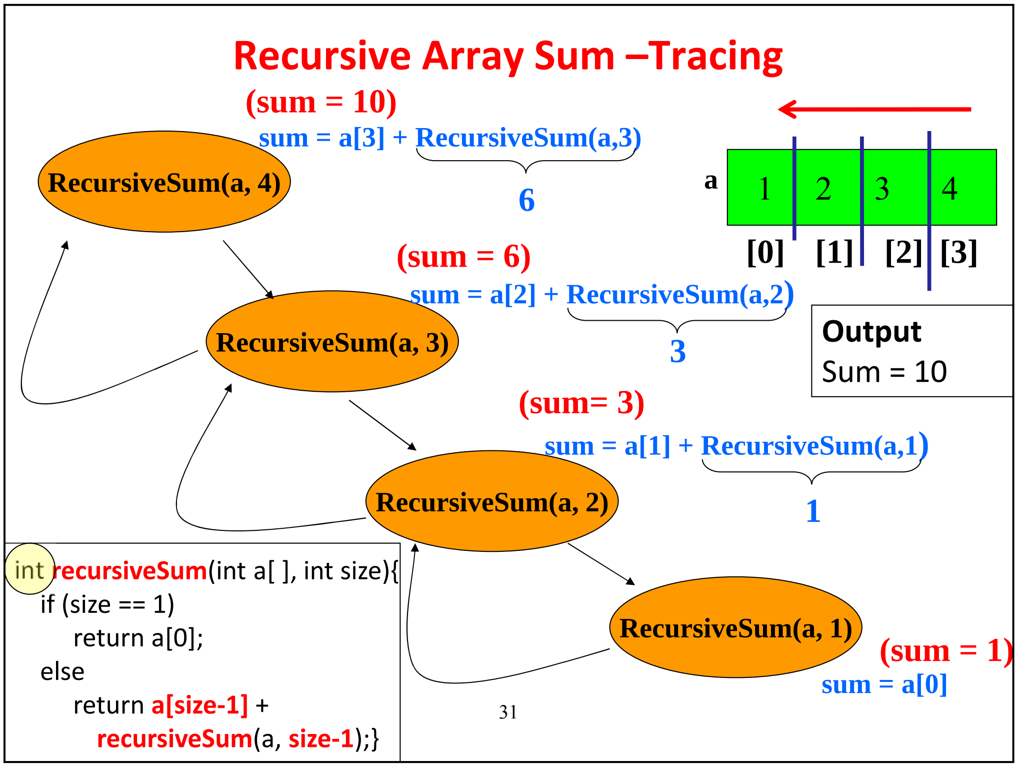
30

Recursive Array Sum

1. To implement the function that recursively computes the sum of elements in arrays is more challenging. The function **recursiveSum()** has two parameters: **a** is an array and **size** is the size of the array to be processed. The code for the function is given as shown below:

```
int recursiveSum(int a[ ], int size)
{
    if (size == 1)          /* terminating condition*/
        return a[0];
    else                    /* recursive condition*/
        return a[size-1] + recursiveSum(a, size-1);
}
```

2. In the function **recursiveSum()**, it computes the sum of all the elements in the array by recursively processes the array and reduces the size of the array by one element each time until the terminating condition is reached. In which case, the array will contain only one element.



Recursive Array Sum –Tracing

(sum = 10)
sum = a[3] + $\underbrace{\text{RecursiveSum(a,3)}}_{6}$

RecursiveSum(a, 4)

a

1	2	3	4
[0]	[1]	[2]	[3]

(sum = 6)
sum = a[2] + $\underbrace{\text{RecursiveSum(a,2)}}_{3}$

RecursiveSum(a, 3)

Output
Sum = 10

(sum= 3)
sum = a[1] + $\underbrace{\text{RecursiveSum(a,1)}}_{1}$

RecursiveSum(a, 2)

RecursiveSum(a, 1)

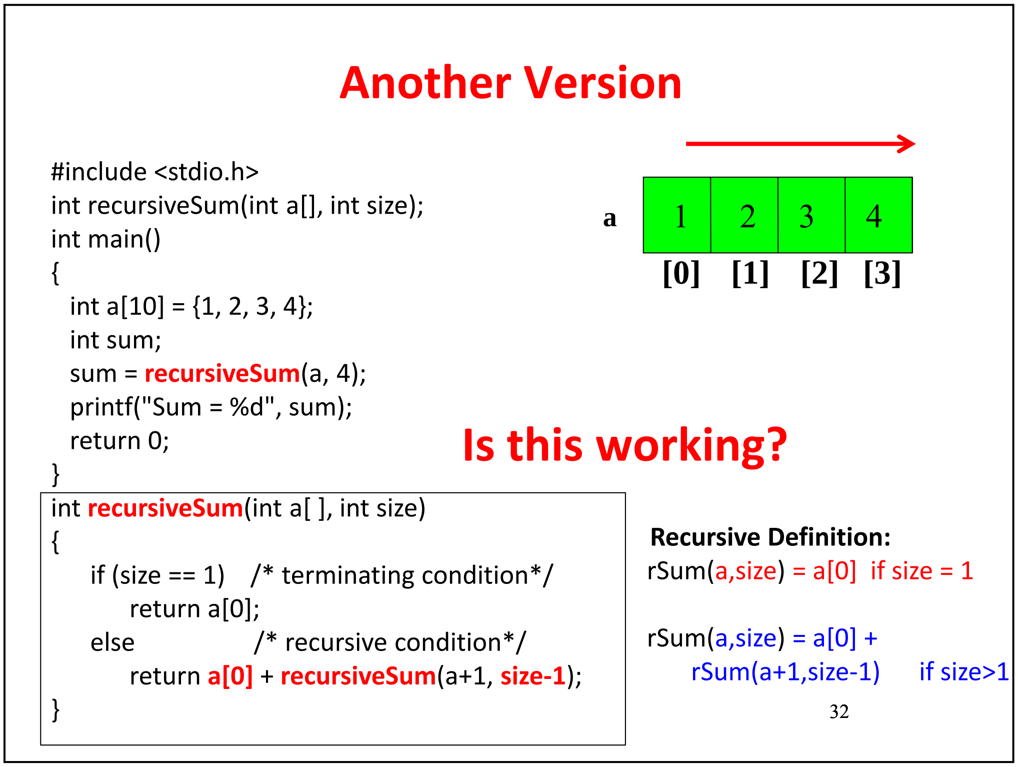
(sum = 1)
sum = a[0]

```
int recursiveSum(int a[ ], int size){
   if (size == 1)
      return a[0];
   else
      return a[size-1] +
         recursiveSum(a, size-1);}
```

31

Recursive Array Sum: Tracing

1. The tracing of the function **recursiveSum()** is illustrated. When the function **recursiveSum(a, 4)** is called from the **main()** function, the code under the recursive condition is executed as **size** is not equal to 1. The statement **return a[size-1] + recursiveSum(a, size-1);** will be executed and call the function **recursiveSum(a, 3)** with **size** reduced to 3.
2. This will in turn execute the function call **recursiveSum(a, 2)**. When the function **recursiveSum(a, 1)** is executed, the code under the terminating condition will be executed. The value of **a[0]** (i.e. 1) will be returned to the calling function. The previous calling function **recursiveSum(a, 2)** will receive the value 1 and add the returned value to **a[1]** and return the sum of 3 to the previous calling function **recursiveSum(a, 3)**. The function **recursiveSum(a, 3)** will add the returned value to **a[2]** and return the sum of 6 to the previous calling function **recursiveSum(a, 4)**. The function **recursiveSum(a, 4)** will add the returned value to **a[3]** and return the sum of 10 to the calling function **main()**.
3. In the **main()** function, the returned value of 10 will then be printed to the screen.



Another Version

```
#include <stdio.h>
int recursiveSum(int a[], int size);
int main()
{
  int a[10] = {1, 2, 3, 4};
  int sum;
  sum = recursiveSum(a, 4);
  printf("Sum = %d", sum);
  return 0;
}
```

a	1	2	3	4
	[0]	[1]	[2]	[3]

Is this working?

```
int recursiveSum(int a[ ], int size)
{
    if (size == 1)   /* terminating condition*/
        return a[0];
    else             /* recursive condition*/
        return a[0] + recursiveSum(a+1, size-1);
}
```

Recursive Definition:

rSum(a,size) = a[0] if size = 1

rSum(a,size) = a[0] +
 rSum(a+1,size-1) if size>1

32

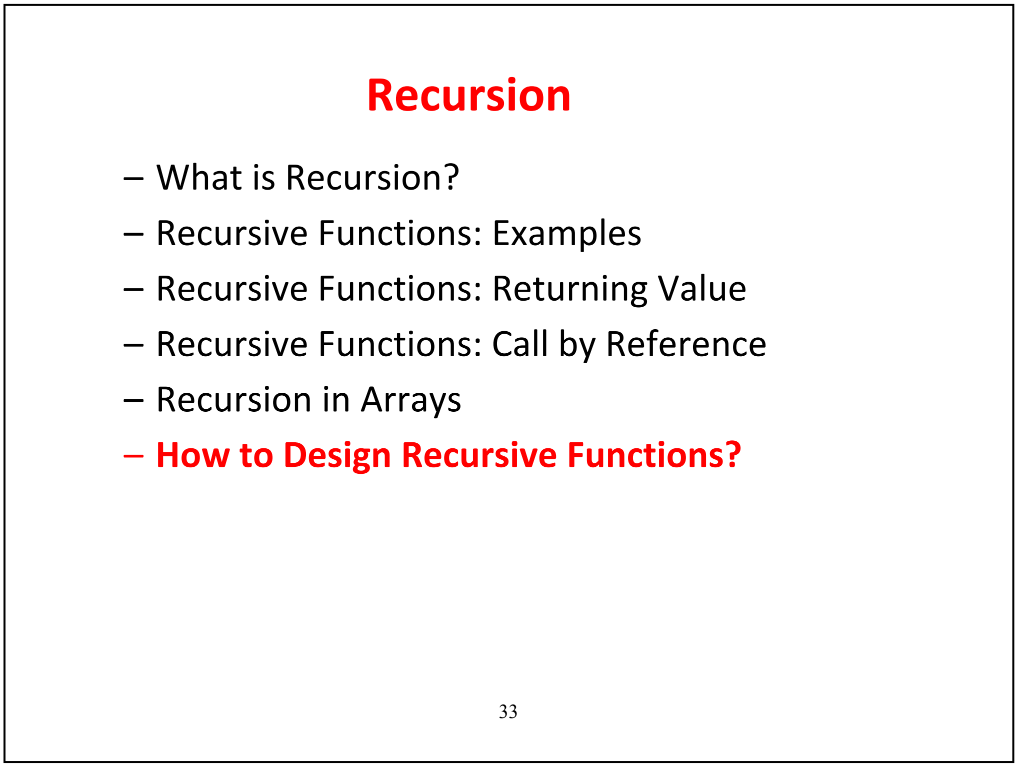
Recursive Array Sum: Another Version

1.Another version for the function is given as shown below:

```
int recursiveSum(int a[ ], int size)
{
    if (size == 1)          /* terminating condition*/
        return a[0];
    else                    /* recursive condition*/
        return a[0] + recursiveSum(a+1, size-1);
}
```

2.For the recursive condition, the following statement will recursively call the function by passing in the updated pointer **a+1** and **size-1** as parameters:

return a[0] + recursiveSum(a+1, size-1);



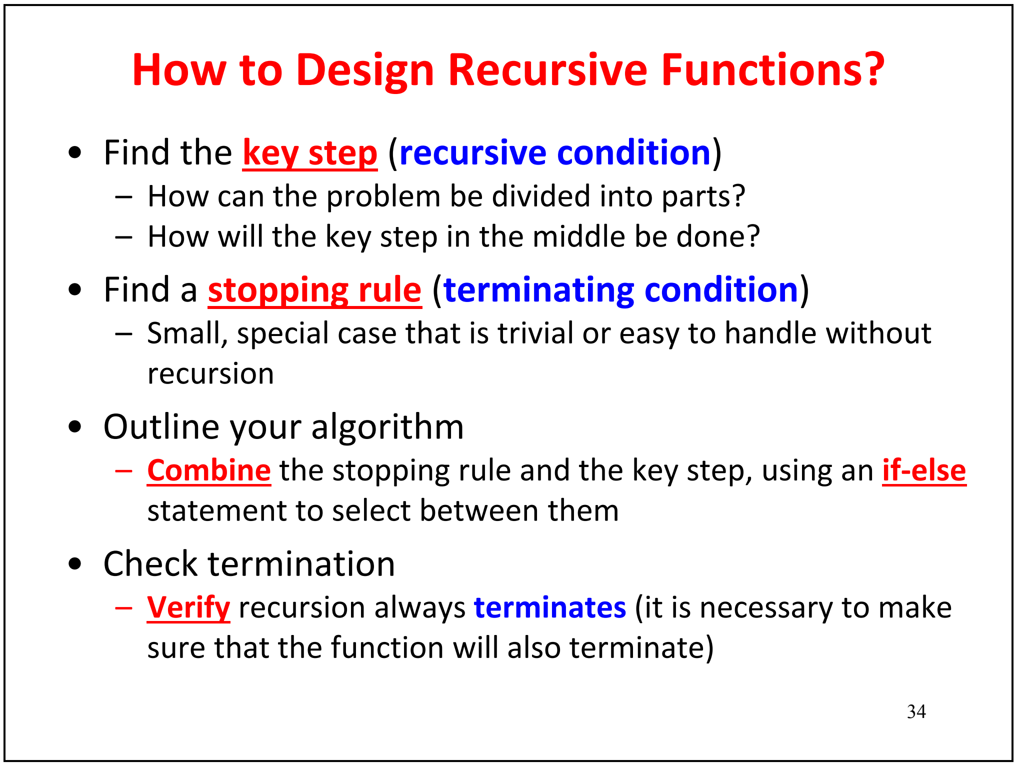
Recursion

- What is Recursion?
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- **How to Design Recursive Functions?**

33

Recursion

1. Here, we discuss the design of recursive functions.



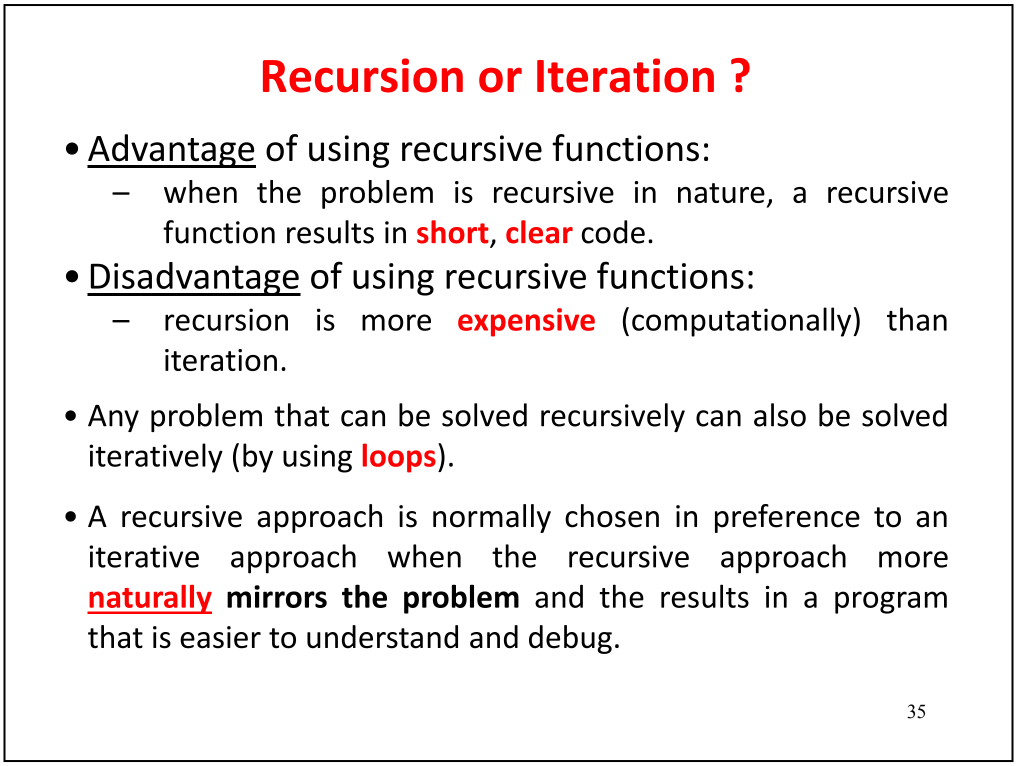
How to Design Recursive Functions?

- Find the key step (**recursive condition**)
  - How can the problem be divided into parts?
  - How will the key step in the middle be done?
- Find a stopping rule (**terminating condition**)
  - Small, special case that is trivial or easy to handle without recursion
- Outline your algorithm
  - Combine the stopping rule and the key step, using an if-else statement to select between them
- Check termination
  - Verify recursion always **terminates** (it is necessary to make sure that the function will also terminate)

34

How to Design Recursive Functions?

1. There are 4 steps involved when you design any recursive functions:
   a) Find the key step (recursive condition) – To decide how can the problem be divided into parts, and how will the key step in the middle be done.
   b) Find a stopping rule (terminating condition) – To decide the terminating condition which should be small and special case that is trivial or easy to handle without recursion.
   c) Outline your algorithm – To combine the stopping rule and the key step, and use an **if-else** statement to select between them.
   d) Check termination – To verify the recursion always terminates. It is necessary to make sure that the function will also terminate.



Recursion or Iteration ?

- Advantage of using recursive functions:
  - when the problem is recursive in nature, a recursive function results in **short**, **clear** code.
- Disadvantage of using recursive functions:
  - recursion is more **expensive** (computationally) than iteration.
- Any problem that can be solved recursively can also be solved iteratively (by using **loops**).
- A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more **naturally** **mirrors the problem** and the results in a program that is easier to understand and debug.

Recursion or Iteration?

1. How can one decide whether to use iteration or recursion approach for implementing a function?
2. The main advantage of using recursive functions is that when the problem is recursive in nature, a recursive function results in shorter and clearer code.
3. However, the main disadvantage of using recursive functions is that recursion is more expensive than iteration in terms of memory usage.
4. Any problems that can be solved recursively can also be solved iteratively (by using loops). A recursive approach is generally chosen over an iterative approach when the recursive approach can mirror the problem more naturally which results in a program that is easier to understand and debug.

In summary, we have observed the following properties of recursive functions:

- Each function has a terminating condition where recursive call will end.
- Each function makes a call to itself with an argument, which is closer to the terminating condition.
- Each level of the function call has its own arguments.
- When a recursive call is made, control is transferred from the calling point to the first statement of the recursive function. When a call at a certain level is finished, control returns to the calling point one level up.

Thank you !!!