fit@hcmus

PROGRAMMING TECHNIQUES

Week 6: Recursion (1)

Nguyễn Hải Minh - 03/2024

Content

- The Nature of Recursion
- Tracing a Recursive Function
- Work through Examples of Recursion
- Recursion vs Iteration

THE NATURE OF RECURSION

- Recursion – The mirrors
- Why Recursion?
- How to write a Recursive algorithm.

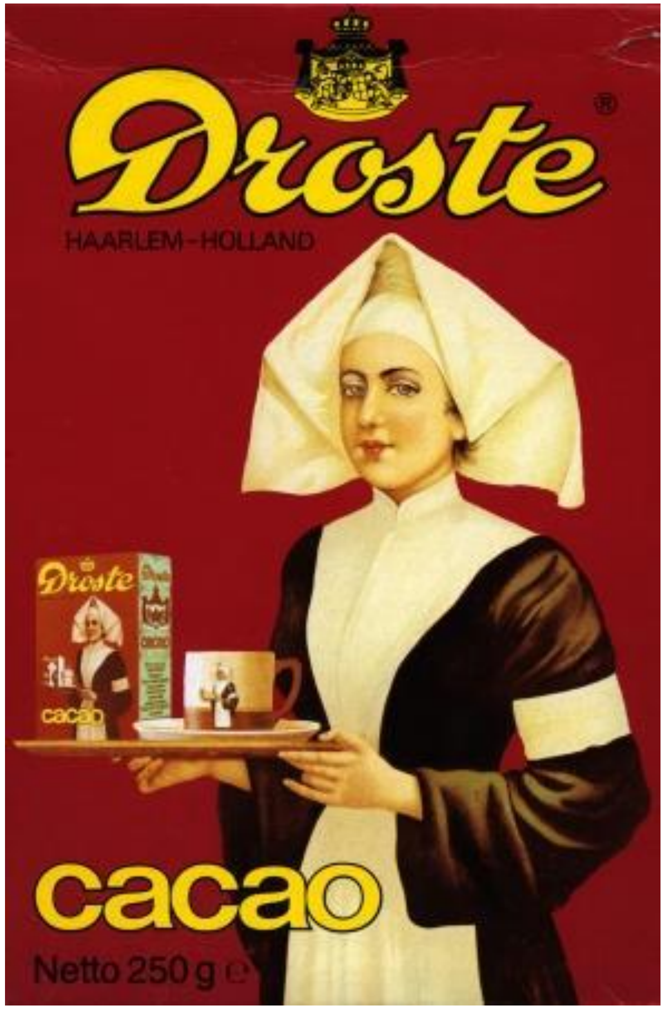
Recursion – The mirrors

- Recursion is an **extremely powerful** problem-solving technique.
- Problems that at first appear to be quite difficult often have simple recursive solutions.
- Idea:
  - Break a problem into several small problems
  - These smaller problem is **exactly the same** type as the original problem
    - Break smaller problem into smaller, and smaller, and smaller…
    - Until the answer is obvious or known.
    - → *Mirror images*

Recursion – The mirrors

A visual form of recursion known as the Droste effect.

(*Advertisement for Droste cocoa, c. 1900*)



Recursion – The mirrors

- Two mirrors face each other

Recursion – The mirrors

- Matryoshka doll

Recursion – The mirrors

- Fractional:

$$0! = 1$$
$$n! = n(n-1)!$$

- Set of natural numbers:

$$0 \in \mathbb{N}$$
$$\text{If } n \in \mathbb{N} \text{ then } n + 1 \in \mathbb{N}$$

- Fibonacci numbers:

$$F(0) = 0, F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Recursive Function

- So far, we have learned about control structures that allow C++ to iterate a set of statements a number of times.
- In addition to iteration, C++ can repeat an action by having a function call itself.
  - This is called recursion.
  - In some case it is more suitable than iteration.

```
int Func(…){
    …
    return Func();
}
```

Recursion – Definition

- Recursion is repetition (by self-reference)
  - It is caused when a function calls/invokes itself.
  - Such a process will repeat forever unless terminated by some control structure.

Why Recursion?

Great Style

Powerful Tool

Master of Control Flow

Recursion in Real Life

- How many students are there in a line?
- Rules:
  - You can only see the people directly in front and behind you.
  - You can ask questions of the people in front and behind you.

How can we solve this problem recursively?

Recursion in Real Life

- Answer:
  1. The first person looks behind him/her and sees if there is a person there. If not, he/she responds “0”
  2. If there is one person, repeat step 1, and wait for a response.
  3. Once a person receives a response, he/she adds 1 for the person behind him/her, and they respond to the person asked them.

How can we solve this problem recursively?

Recursive Function

- The structure of recursive functions is typically like the following:

```
RecursiveFunction(){
   if(test for simple case){
      Compute the solution without recursion
   }
   else{
      Break the problem into subproblems of the same form
      Call RecursiveFunction() on each subproblem
      Reassamble the results of the subproblems
   }
}
```

Recursive Algorithm

- Every recursive algorithm requires at least two cases:
  - Base case: The simple case, the occurrence that can be answered directly (stopping condition)
  - Recursive case: a more complex occurrence of the problem that cannot be answered directly; but can be described in terms of smaller occurrences of the same problem.

Recursive Function

- The function for the count number of student problem:

```
NumberOfStudentBehind(student curr){
  if(No one is behind curr){
     return 0;
  }
  else{
     student behind = StudentBehind(curr);
     return NumberOfStudentBehind(behind) + 1;
  }
}
```

Base case

Recursive case

Recursive call

Recursive Algorithm – How to write

- 3 “must have” of a recursive algorithm:
  1. Your code must have a case for **all valid inputs**.
  2. You must have a **base case** with no recursive calls.
  3. When you make a **recursive case**, it should be to a simpler instance and make forward progress towards the base case.

Recursion – Example

- Let's look at another simple example;
  - In this case we can see that by using recursion we can make some difficult problems very trivial...
  - Many of these problems would be very difficult to solve if you only were able to use iteration.

Recursion – Example

```
void strange();
int main(){
   cout << "Pls input numbers(0-stop)"
   <<endl;
   strange();
   cout << endl;
   return 0;
}
```

```
void strange() {
   int t;
   cin >> t;
   if (t != 0){
      strange();
      cout << t << “ ”;
   }
}
```

What is the purpose of this function?

Recursion – Example

- This program writes the reverse of what was entered at the keyboard, *no matter how many numbers were entered!*
  - Try to write an equally simple program just using the iterative statements we know about;

    → *it would be difficult to make it behave the same without limiting the number of numbers that can be entered or using up a lot of memory with a huge array of numbers!*
  - Notice, with recursion, we didn't have to even use an array or list to store data...

Recursion – Example

- What happens to this “power” if we had swapped the `cout` statement with the recursive call in the previous example?
  - It would have simply read and echoed what was typed in.
  - Recursion would be overkill; iteration should be used instead.

TRACING A RECURSIVE FUNCTION

What happens when a function is called?

```
int func1(int a1)
{
   return a1*2;
}
```

```
int func2(int a2)
{
  int x, z;
  x = 5;
  z = func1(a2)+x;
  return z;
}
```

An activation record is stored into a function call stack (run-time stack)

1. The computer stops executing funct2 and starts executing func1
2. Since it needs to come back to func2 later, it needs to store everything about func2 that is going to need (a2, x, z and the place to start executing upon return)
3. Then, a2 from func2 is bounded to a1 from func1
4. Control is transferred to func1

What happens when a function is called?

```
int func1(int a1)
{
    return a1*2;
}
```

```
int func2(int a2)
{
   int x, z;
   x = 5;
   z = func1(a2)+x;
   return z;
}
```

An activation record is stored into a function call stack (run-time stack)

5. After func1 is executed, the activation record is popped out of the run-time stack
6. All the old values of the parameters and variables in func2 are restored and the return value of func1 replaces func1(a2) in the assignment statement

Different between recursive and non-recursive calls?

→ **NO**

What happens when a function is called?

- When a recursive call is encountered, execution of the current function is temporarily stopped.

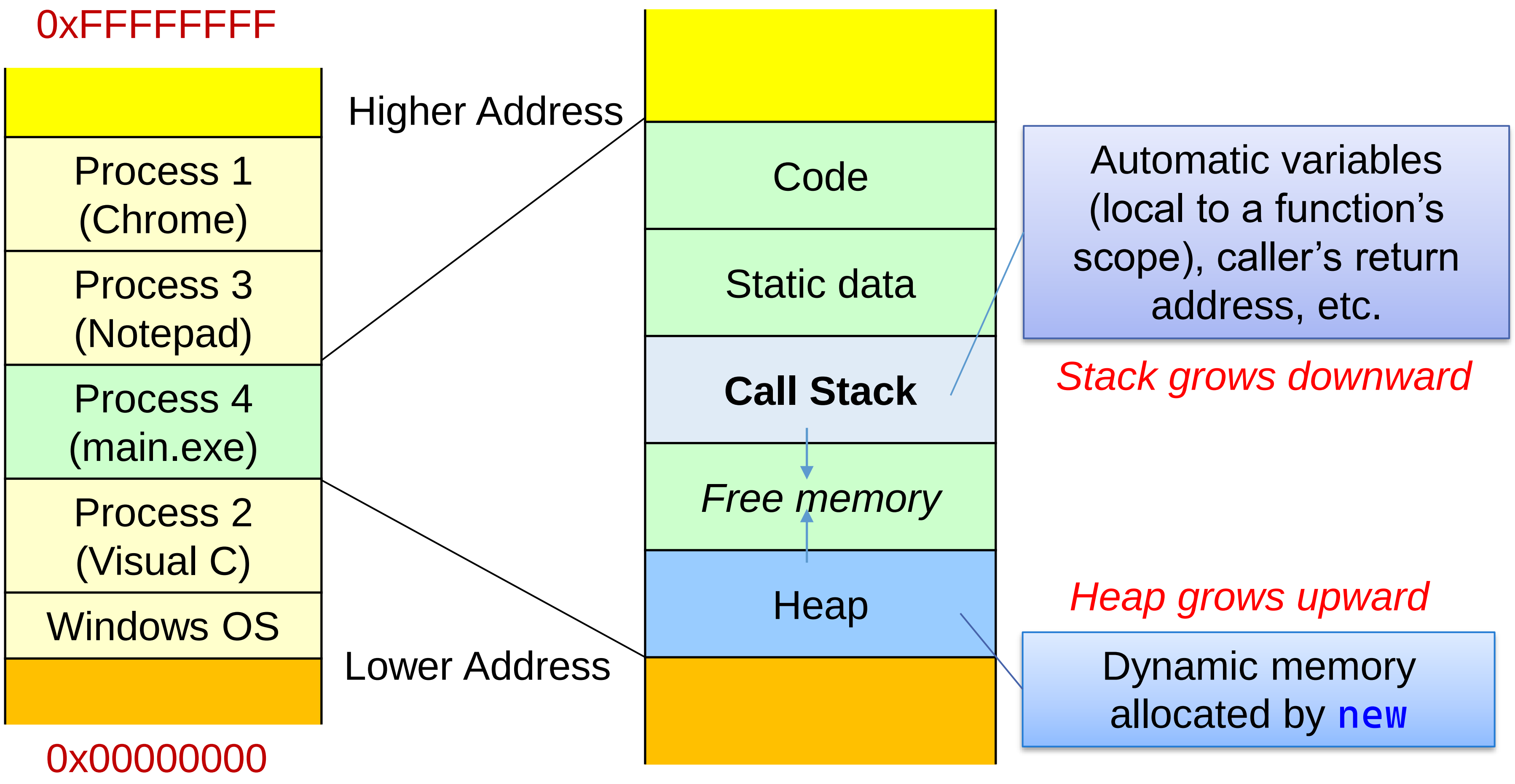
  → This is because the *result* of the recursive call *must be known* before it can proceed.

- So, it saves all of the information it needs in order to continue executing that function later

  → (i.e., all current values of all local variables and the location where it stopped).

- Then, when the recursive call is completed, the computer returns and completes execution of the function.

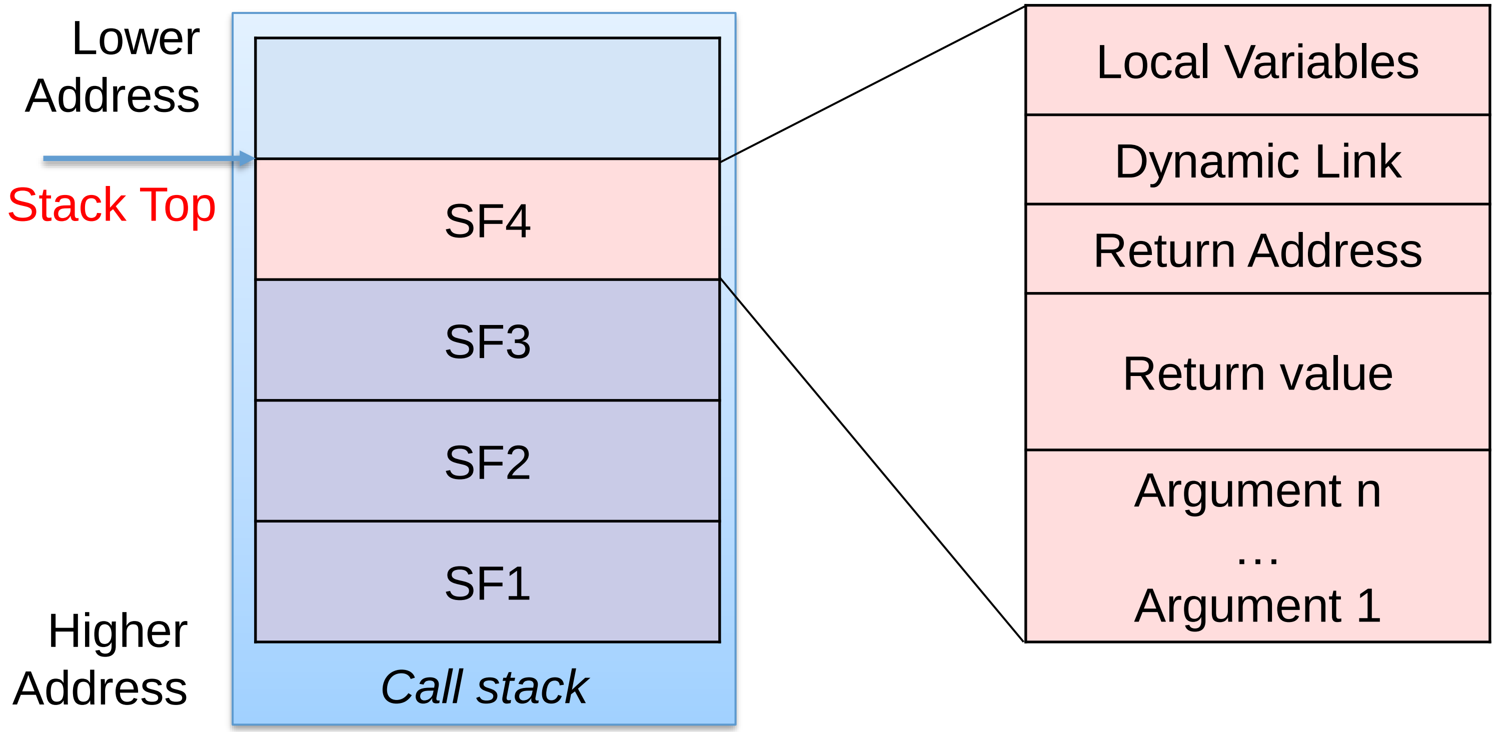
Memory Organization



However, the direction in which the memory grows is dependent on the operating system and the architecture of the computer.

Activation Record (Stack Frame)

- Whenever a function is invoked, the program creates an **activation record** (or **stack frame**) and places it on top of the call stack.



Box Trace – Step 1

- Use box trace to understand and debug recursive functions.
- The box illustrates how compilers implement recursion
- Each box represents an activation record of a function call.

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

A

1. Label each recursive call in the body of the recursive function

```
int main()
{
   cout << Fact(3);
}
```

Box Trace – Step 2

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

Argument Values

Value returned by recursive call from the current box

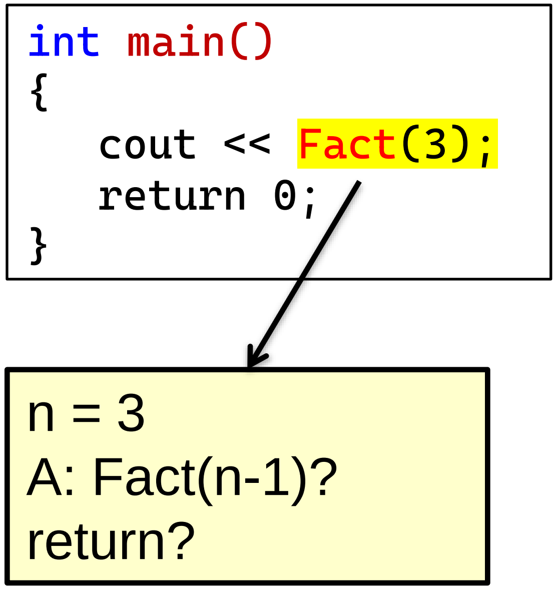
n = ?
A: Fact(n-1)?
return?

Function Values

2. Represent each call to the function by a new box in which you note the local environment of the function

Box Trace – Step 3

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



```
int main()
{
    cout << Fact(3);
    return 0;
}
```

n = 3
A: Fact(n-1)?
return?

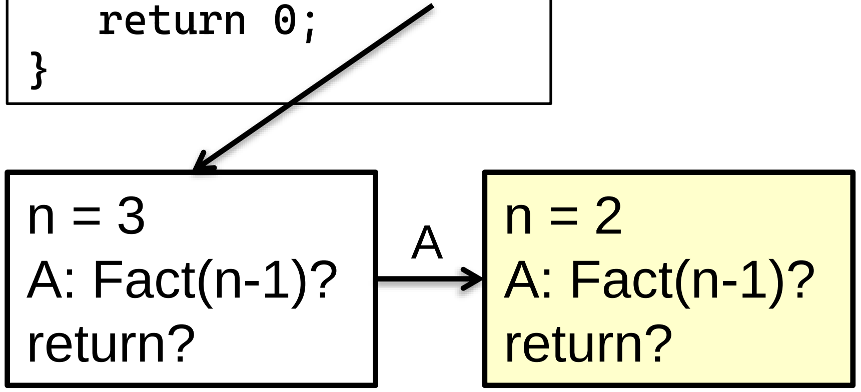
3. Draw an arrow from the statement that initiates the recursive process to the first box

Box Trace – Step 4

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

```
int main()
{
    cout << Fact(3);
    return 0;
}
```

4. Start executing the body of the function

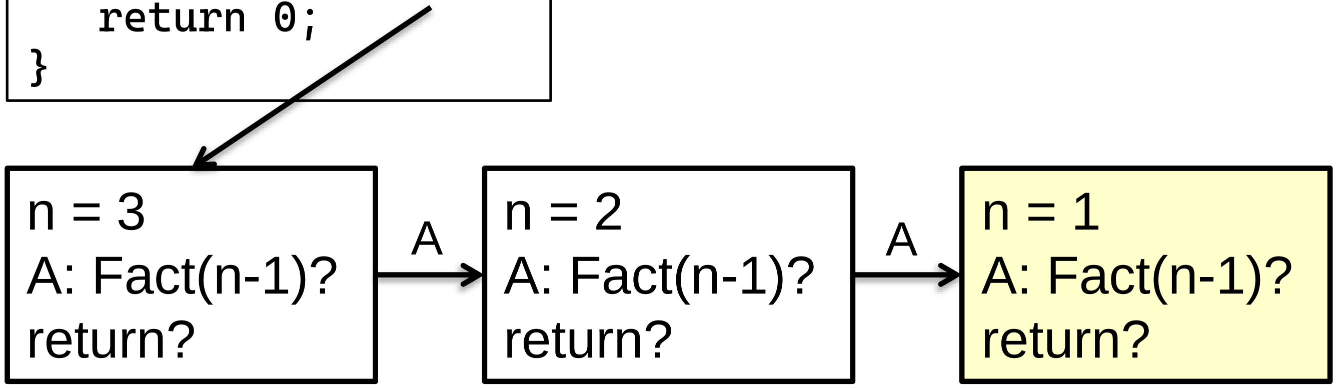


Box Trace – Step 4

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

```
int main()
{
    cout << Fact(3);
    return 0;
}
```

4. Start executing the body of the function

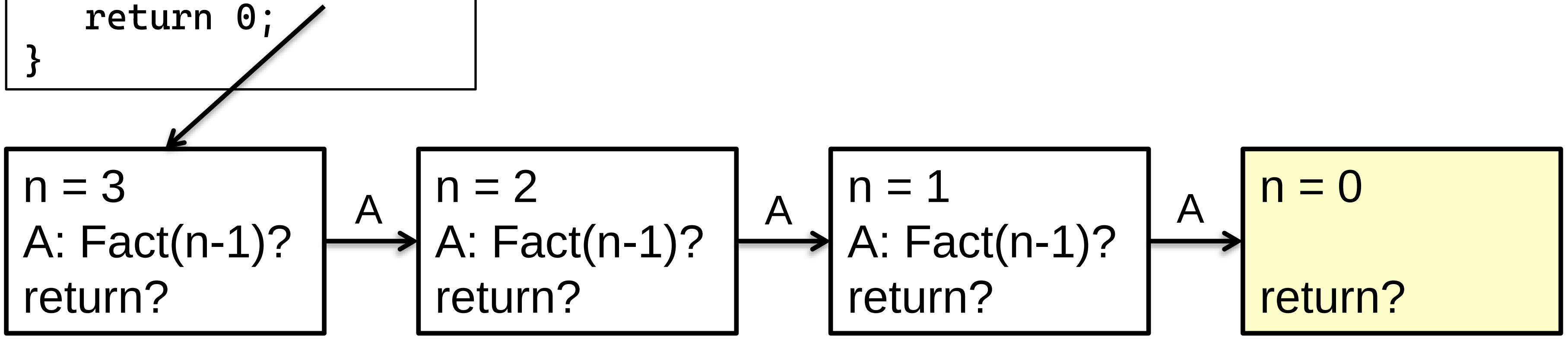


Box Trace – Step 4

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

```
int main()
{
    cout << Fact(3);
    return 0;
}
```

4. Start executing the body of the function

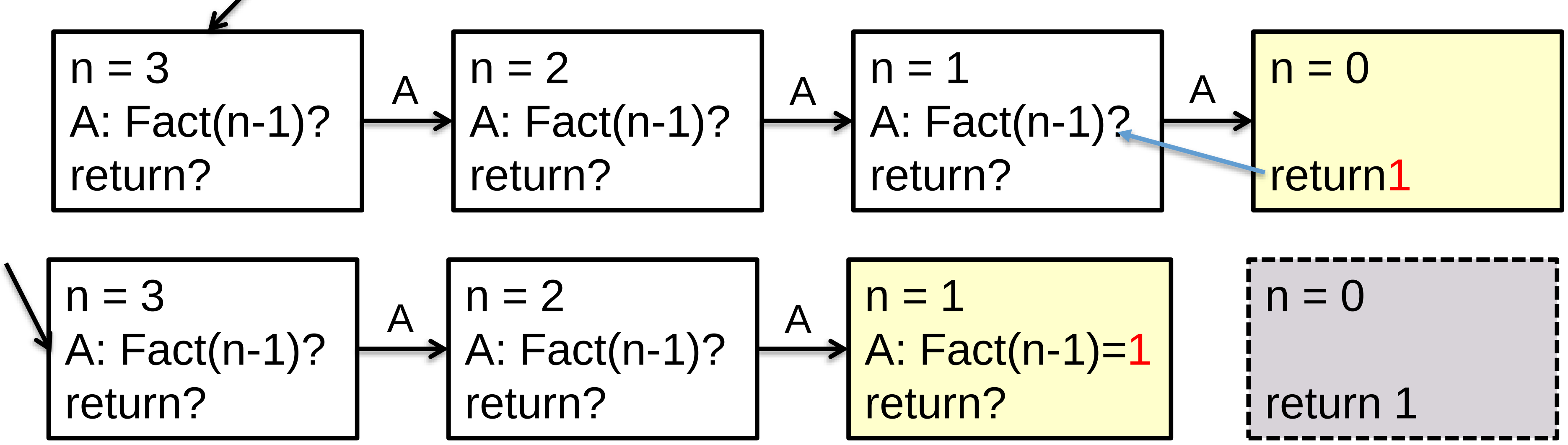

$$\begin{tikzcd}
\boxed{\begin{array}{l} n = 3 \\ \text{A: Fact(n-1)?} \\ \text{return?} \end{array}} \arrow[r, "A"] & \boxed{\begin{array}{l} n = 2 \\ \text{A: Fact(n-1)?} \\ \text{return?} \end{array}} \arrow[r, "A"] & \boxed{\begin{array}{l} n = 1 \\ \text{A: Fact(n-1)?} \\ \text{return?} \end{array}} \arrow[r, "A"] & \boxed{\begin{array}{l} n = 0 \\ \\ \text{return?} \end{array}}
\end{tikzcd}$$

Box Trace – Step 5

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

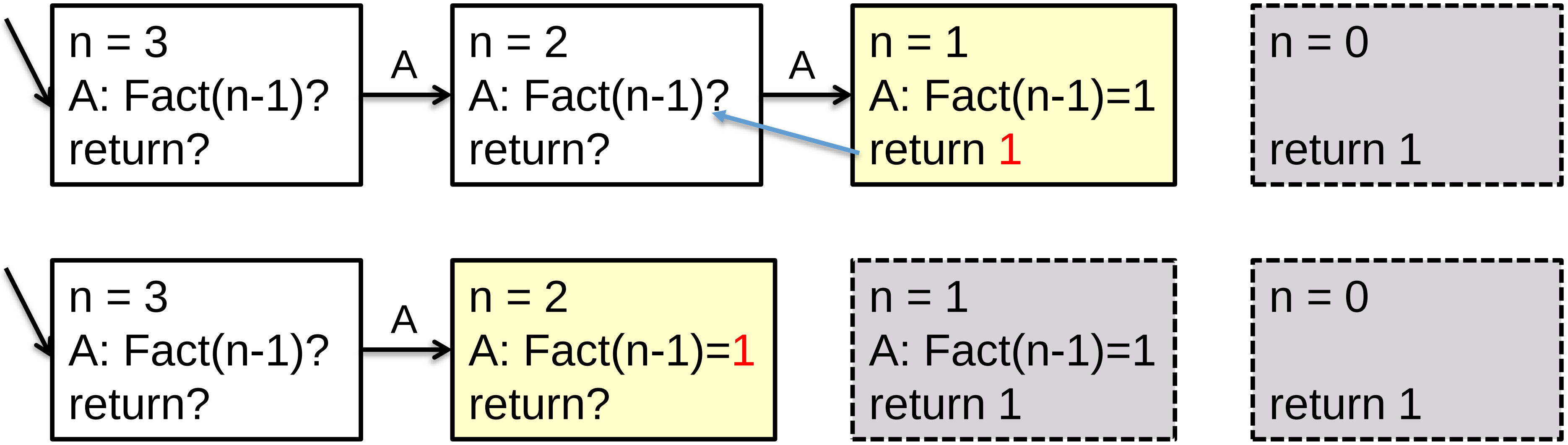
```
int main()
{
    cout << Fact(3);
    return 0;
}
```

5. On exiting the function, cross it off and follow its arrow back to the box that called the function



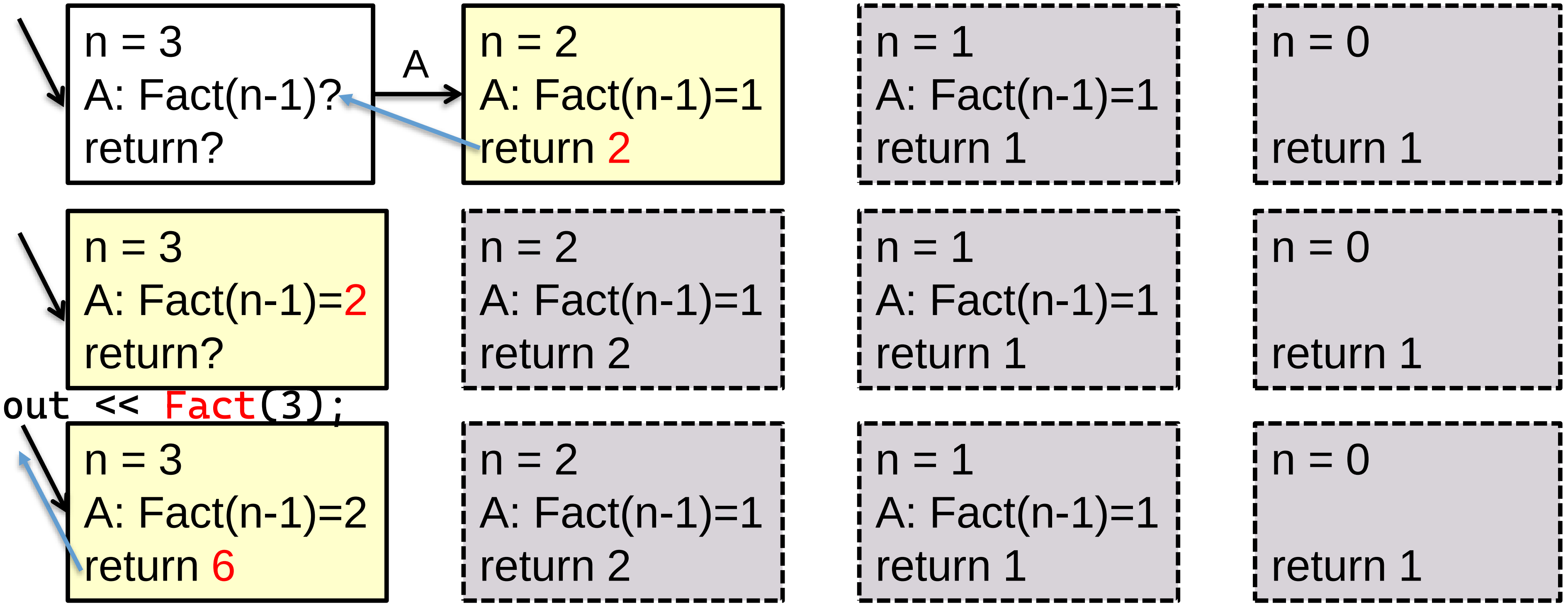
Box Trace – Step 5

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



Box Trace – Step 5

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



The value 6 is returned to the initial call

WORK THROUGH EXAMPLES OF RECURSION

- Recursion That Returns a Value
- Recursion That Perform an Action
- Recursion With Arrays

Recursion that returns a value

- Return the factorial value of a number:

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

- Compare and contrast with the iterative version. Which is better? Why?

Recursion that performs an action

- Print out from numbers from n to 1:

```
void PrintNum(int n)
{
    if (n == 0)
        return;
    cout << n << endl;
    PrintNum(n - 1);
}
```

- Compare and contrast with the iterative version. Which is better? Why?

Recursion with arrays

- Binary Search (return the index of `x` in array `arr`)

```
int BinarySearch(const int arr[], int first, int last, int x)
{
    if (first > last)
        return -1; //x doesn't appear in arr
    int mid = (first + last)/2;
    if (x == arr[mid]) //x is in the middle
        return mid;
    else if (x < arr[mid]) //x maybe in the left
        return BinarySearch(arr, first, mid - 1, x);
    else //x maybe in the right
        return BinarySearch(arr, mid + 1, last, x);
}
```

Recursion – The most common error

- Stack Overflow

  → *This simply means that you've tried to make too many function calls recursively*.

- If you get this error, one clue would be to look to see if you have infinite recursion.

  → This situation will cause you to exceed the size of your stack -- no matter how large your stack is!

RECURSION & ITERATION

Recursion and Iteration

- While recursion is very powerful
  - It should not be used if iteration can be used to solve the problem in a maintainable way (i.e., if it isn't too difficult to solve using iteration)
  - So, think about the problem. Can loops do the trick instead of recursion?

Recursion and Iteration

- Why select iteration versus recursion?
  - Every time we call a function a stack frame is pushed onto the program stack and a jump is made to the corresponding function
  - This is done in addition to evaluating a control structure (such as the conditional expression for an if statement) to determine when to stop the recursive calls.
  - With iteration all we need is to check the control structure (such as the conditional expression for the while, do-while, or for)

Efficiency!

Recursion and Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

Recursion and Iteration

Recursion should **NOT** be used if it makes your algorithm **harder to understand** or if it results in **excessive demands on storage** or **execution time**.

Practice Exercise

1. Implement the Towers of Hanoi problem using recursion. Discuss the benefits and drawbacks of recursion for this algorithm.
2. Rewrite the insert and remove functions with linked lists using recursion (just for practice...)
   - try to add to the end recursively
   - try to remove in the middle recursively

Next week topic

- Recursion (cont)
  - Problem solving with recursion
  - Work through examples to get used to the recursive process
- Quiz:
  - Review Quiz: Recursion

Q&A