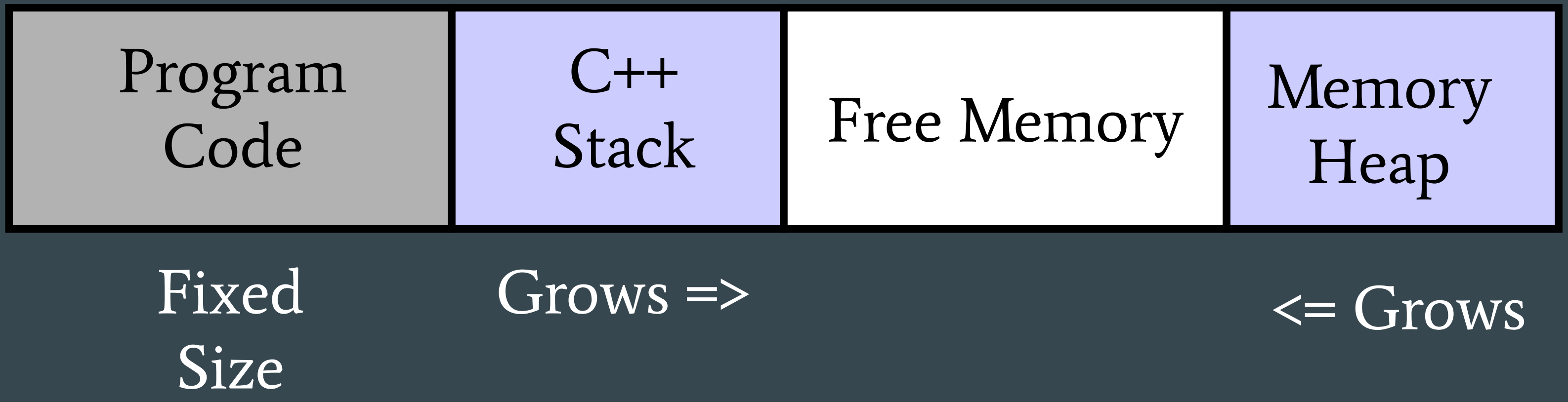
CPE 212 - Fundamentals of Software Engineering

Recursion

Outline

- C++ Memory Allocation
- C++ Runtime Stack Intro
- Recursion Definition
- Example
- Terminology
- Best Practices

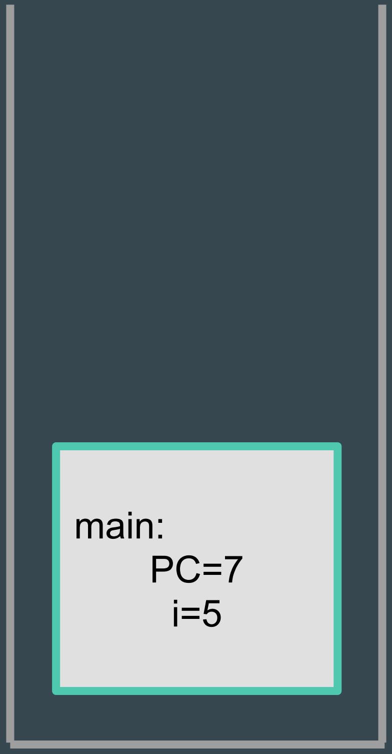
C++ Memory Allocation



Activation Record

- A record used at run time to store information about a function call, including the parameters, local variables, register values, and return address
- Also called a stack frame
- Gets put on the run-time stack
  - Data structure used to keep track of activation records during the execution of a program

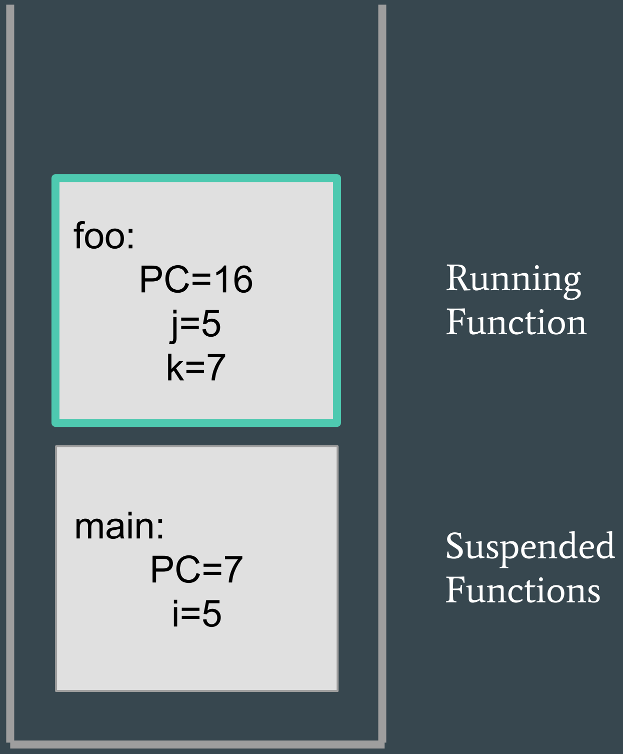
C++ Run-time Stack



Running
Function

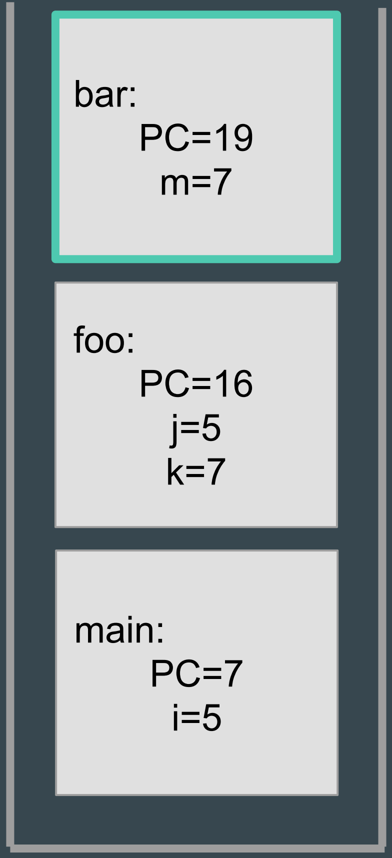
```
1   int main()
2   {
3       int i = 5;
4       /*
5        *  Stuff
6        */
7       foo(i);
8       /*
9        *  More Stuff
10       */
11  }
12
13  void foo(int j)
14  {
15      int k = 7;
16      bar(k);
17  }
18
19  void bar(int m)
20  {
21      // Implementation
22  }
```

C++ Run-time Stack




```cpp
1  int main()
2  {
3      int i = 5;
4      /*
5       *  Stuff
6       */
7      foo(i);
8      /*
9       *  More Stuff
10      */
11 }
12
13 void foo(int j)
14 {
15     int k = 7;
16     bar(k);
17 }
18
19 void bar(int m)
20 {
21     // Implementation
22 }
```

C++ Run-time Stack

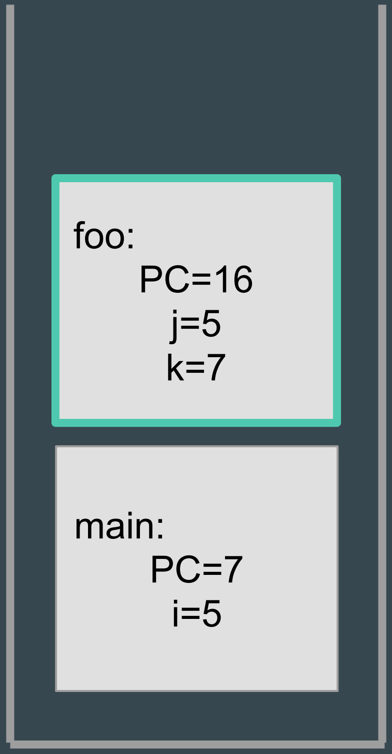


Running Function

Suspended Functions

```
1   int main()
2   {
3       int i = 5;
4       /*
5        *  Stuff
6        */
7       foo(i);
8       /*
9        *  More Stuff
10       */
11  }
12
13  void foo(int j)
14  {
15      int k = 7;
16      bar(k);
17  }
18
19  void bar(int m)
20  {
21      // Implementation
22  }
```

C++ Run-time Stack

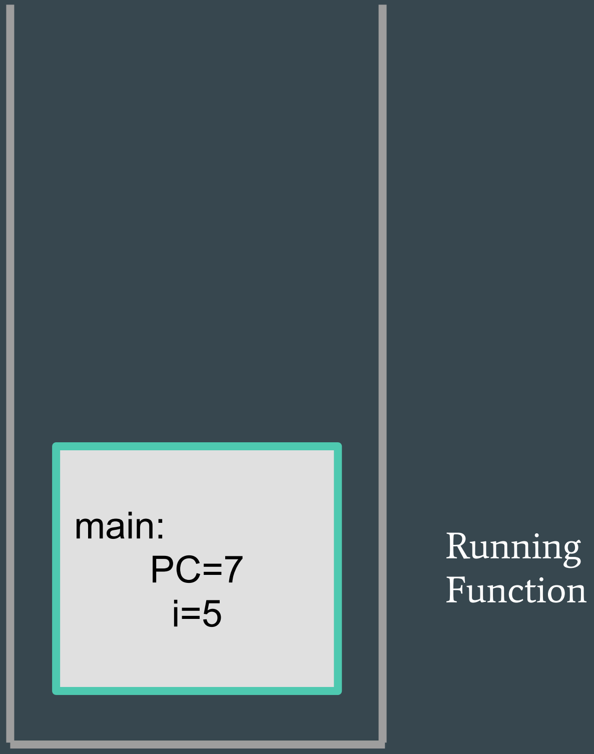


Running Function

Suspended Functions

```cpp
1  int main()
2  {
3      int i = 5;
4      /*
5       *  Stuff
6       */
7      foo(i);
8      /*
9       *  More Stuff
10      */
11 }
12
13 void foo(int j)
14 {
15     int k = 7;
16     bar(k);
17 }
18
19 void bar(int m)
20 {
21     // Implementation
22 }
```

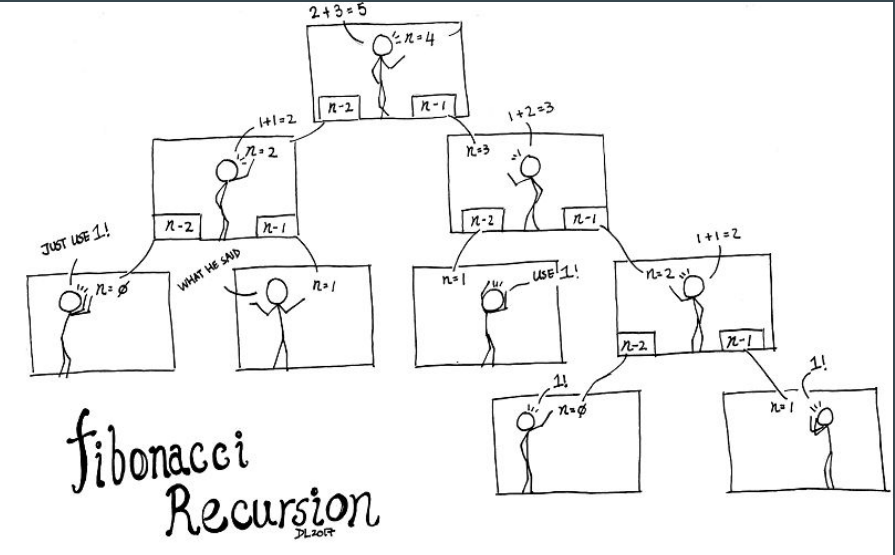
C++ Run-time Stack




```
1   int main()
2   {
3       int i = 5;
4       /*
5        *  Stuff
6        */
7       foo(i);
8       /*
9        *  More Stuff
10       */
11  }
12
13  void foo(int j)
14  {
15      int k = 7;
16      bar(k);
17  }
18
19  void bar(int m)
20  {
21      // Implementation
22  }
```

Recursion

- Recursive Call
  - A function call in which the function being called is the same as the one making the call
- Direct Recursion
  - When a function directly calls itself
- Indirect Recursion
  - When a chain of two or more function calls returns to the function that originated the chain



https://stackoverflow.com/questions/41903017/how-to-visualize-fibonacci-recursion

Recursion Example

Factorial of n:

```
n! = 1 x 2 x 3 x … x (n-2) x (n-1) x n
```

Factorial of 3:

```
3! = 1 x 2 x 3
   = 6
```

Factorial using recursion:

```
F(n) = 1        when n = 0 or 1
     = F(n-1)   when n > 1
```

```
Fact(n)
Begin
  if n == 0 or 1 then
    Return 1;
  else
    Return n*Call Fact(n-1);
  endif
End
```

Recursion Terminology

- Base Case
  - The case for which the solution can be stated non recursively
- General (recursive) Case
  - The case for which the solution is expressed in terms of a smaller version of itself
- Recursive Algorithm
  - A solution that is expressed in terms of (1) smaller instances of itself and (2) a base case

Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)          // Line 1
    return 1;         // Line 2
 else
    return n*Factorial(n - 1);  // Line 3
}
```

Factorial(3)

Factorial(3)

3 * Factorial(2)

Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)          // Line 1
    return 1;         // Line 2
 else
    return n*Factorial(n - 1);   // Line 3
}
```

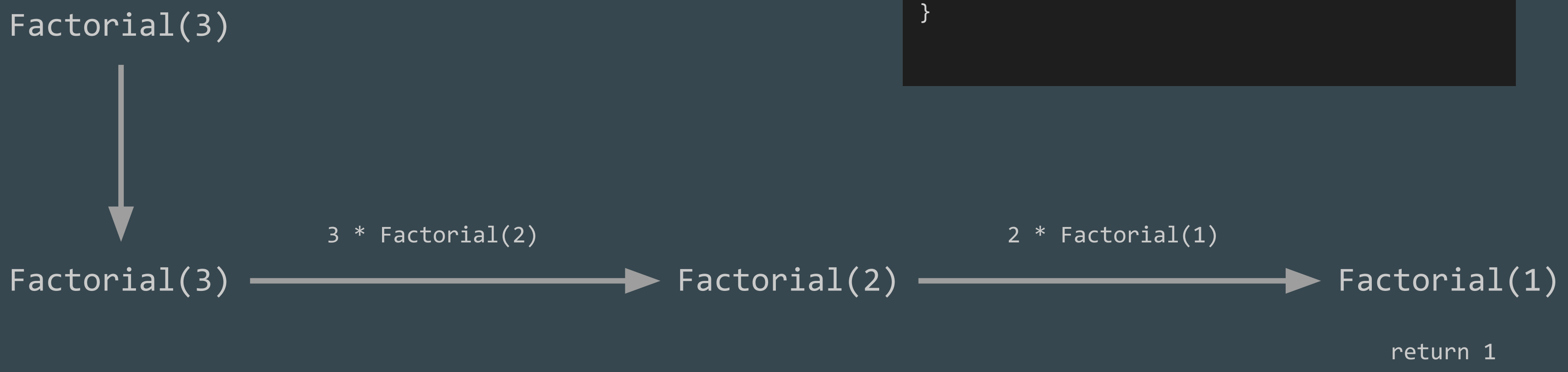
Factorial(3)

Factorial(3) $\xrightarrow{3 \; * \; \text{Factorial(2)}}$ Factorial(2)

2 * Factorial(1)

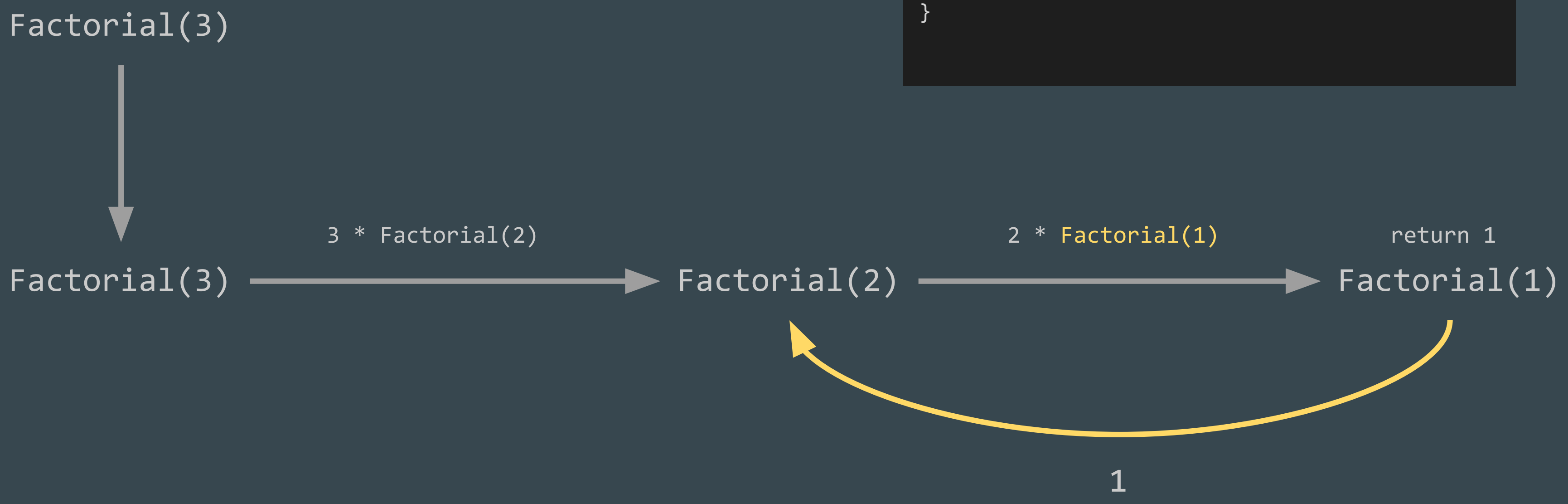
Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)        // Line 1
    return 1;       // Line 2
 else
    return n*Factorial(n - 1);  // Line 3
}
```



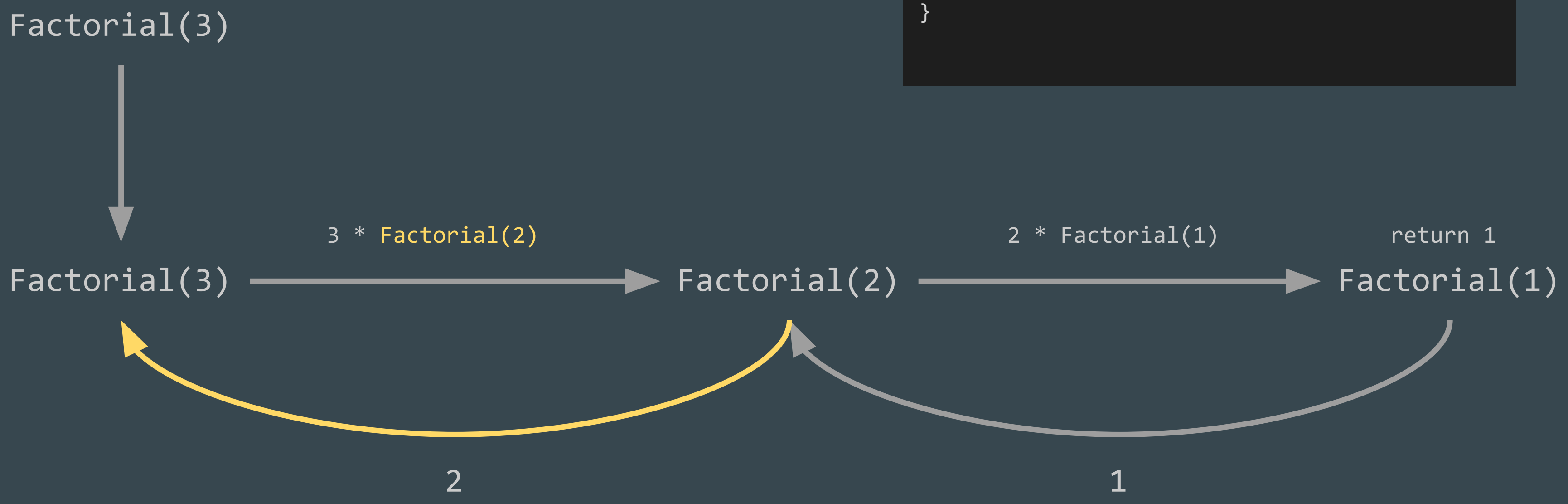
Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)         // Line 1
    return 1;        // Line 2
 else
    return n*Factorial(n - 1);  // Line 3
}
```



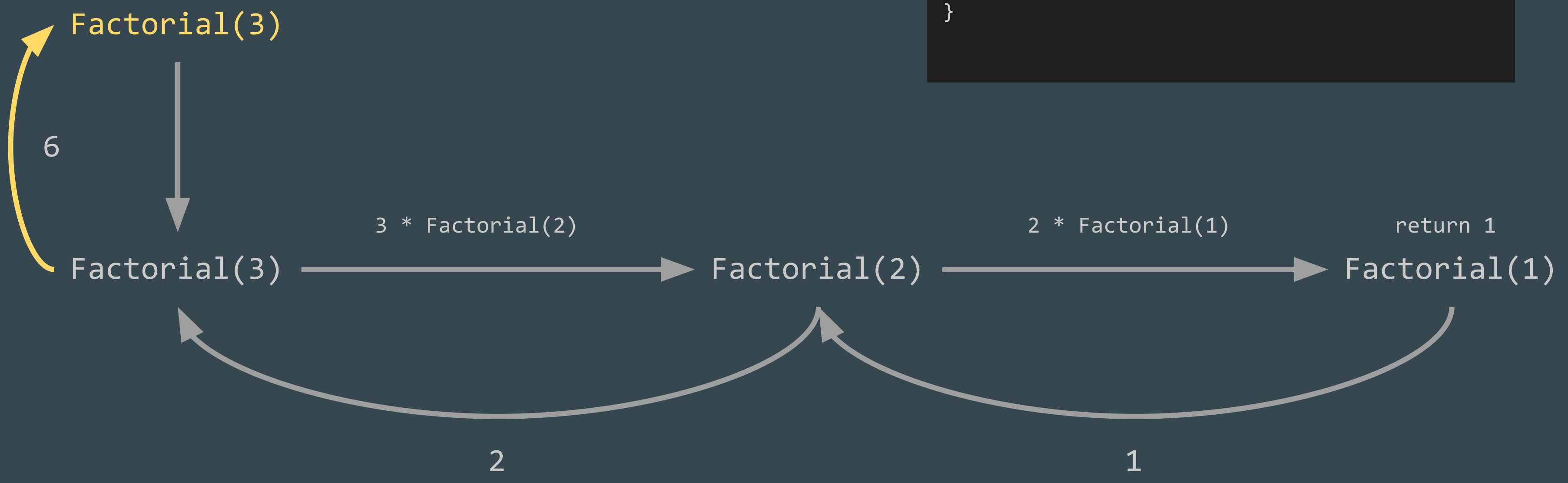
Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)          // Line 1
    return 1;         // Line 2
 else
    return n*Factorial(n - 1);  // Line 3
}
```



Recursion Visualization

```
int Factorial(int n)
{
 if (n == 1)        // Line 1
    return 1;       // Line 2
 else
    return n*Factorial(n - 1);  // Line 3
}
```



Recursion Implementation

```
int Factorial(int n)     // Recursive
{
 if (n == 1)        // Line 1
       return 1;       // Line 2
 else
       return n*Factorial(n - 1);   // Line 3
}
```

```
int Factorial(int n)     // Non-recursive
{
 int fact = 1;
 for( int k = 2; k <= n; k++)
 {
   fact = fact*k;
 }
 return fact;
}
```

Infinite Recursion

- The situation in which a function calls itself over and over endlessly
- Consequences of Infinite Recursion
  - Run-time stack grows
  - Memory space consumed
  - Run-Time “Stack Overflow” error occurs

Verifying Recursive Functions

Three-Question Method

1. Base-Case Question
   a. Is there a non-recursive way to exit the function?
   b. Is it correct?
2. Smaller-Case Question
   a. Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
3. General-Case Question
   a. Assuming that the recursive calls work correctly, does the entire function work correctly?

Proof-By-Induction Procedure

1. Prove that f(n) is true for some value k
2. Assume that f(n) is true for some value n > k
3. Show that f(k+1) is true

Conclude that f(n) is true for all n >= k

Proof-By-Induction Example

Correctness Proof:

Assume N = 1.

Does Factorial(1) equal 1! ? Yes! Factorial(1) = 1 = 1!

Assume Factorial(N) is correct, i.e. Factorial(N) = N * (N-1) * ... * 2 * 1 = N!

Prove Factorial(N+1) (N+1)!

Proof-By-Induction Example

Mathematically: (N+1)! = (N+1) * N * (N-1) * ... * 2 * 1 = (N+1) * N!

According to the source code:

Factorial(N+1) = (N+1) * Factorial(N)

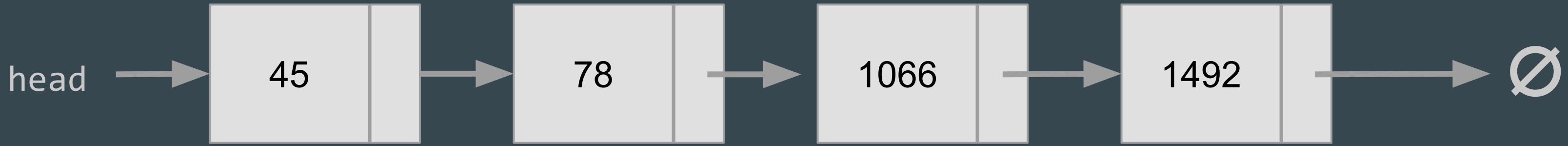
= (N+1) * N! [Assuming Factorial(N) = N!]

= (N+1)!

Since we assumed that Factorial(N) = N!, we can rewrite Factorial(N+1) = (N+1)! = (N+1) * N!

Therefore, the function Factorial(N) will return N! for an arbitrary value of N >= 1.

Recursion with Data Structures




```
void  ReversePrint(NodePtr  head)
{
   if  (head != NULL)
   {
       ReversePrint(head->link);
       cout << head->component << endl;
   }
}


// Usage:
ReversePrint(head_of_list_ptr);
```

```
Call 1:  head != NULL, suspended
Call 2:  head != NULL, suspended
Call 3:  head != NULL, suspended
Call 4:  head != NULL, suspended
Call 5:  head == NULL,
      control returns to Call 4
Call 4:  resumes, prints 1492
      control returns to Call 3
Call 3:  resumes, prints 1066
      control returns to Call 2
Call 2:  resumes, prints 78
      control returns to Call 1
Call 1:  resumes, prints 45
```

Tail Recursion

- A recursive function is tail recursive when recursive call is the last thing executed by the function.
- Tail recursion can be optimized by the compiler
- Is the factorial example tail recursion?

```
// An example of tail recursive function
void print(int n)
{
   if (n < 0)  return;
   cout << " " << n;
    // The last executed statement is recursive call
   print(n-1);
}
```

Writing Recursive Functions

- Understand the problem first!!
- Determine the size of the problem
- Identify and solve the base case
- Identify and solve the general case using smaller instance of the general case

When should you use Recursion?

- Depth of recursion is relatively “shallow”
- Number of recursive calls grows slowly as problem size grows
- Recursive version does roughly the same amount of work as the non-recursive version
- Recursive version is shorter and simpler than the non-recursive version