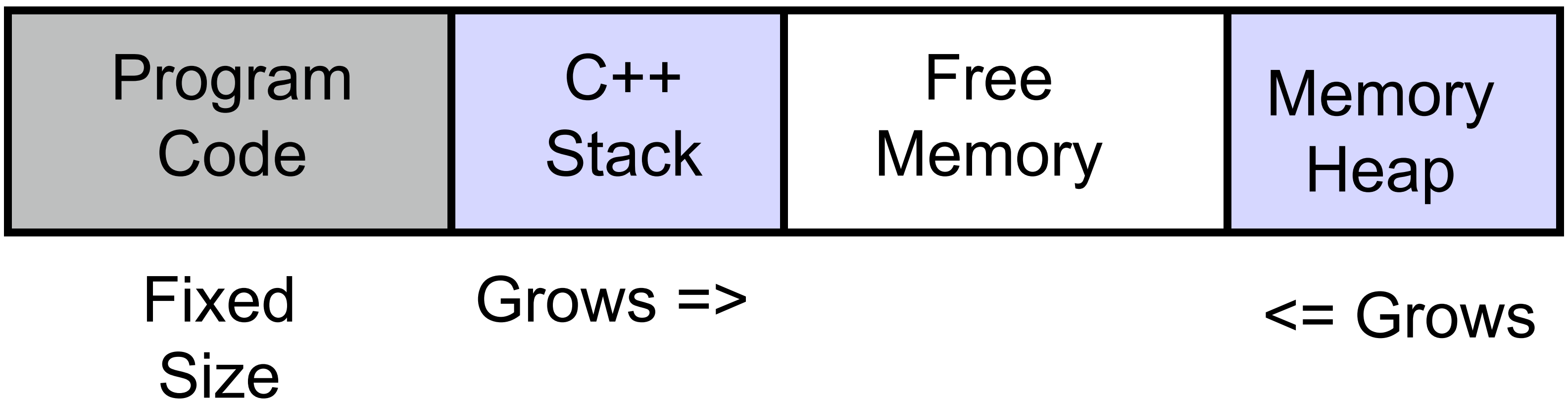
Recursion

CPE 212 -- Lecture 13

UAHuntsville

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*

Run-Time Stack

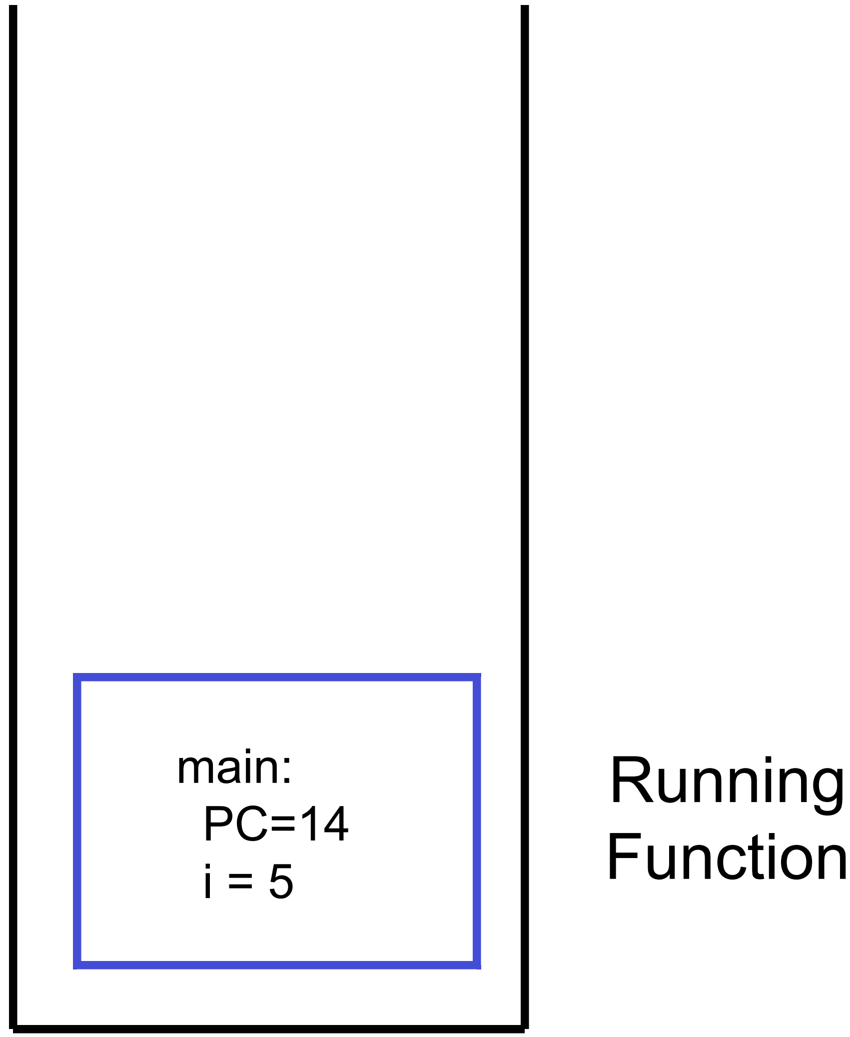
- Data structure used to keep track of activation records during the execution of a program

C++ Run-Time Stack

```
PC
      int main()
      {
        int i=5;
        …
14      cool(i);
        …
      }

      void cool(int j)
      {
        int k=7;
        …
216     fool(k);
        …
      }

320   void fool(int m)
      {
        …
      }
```

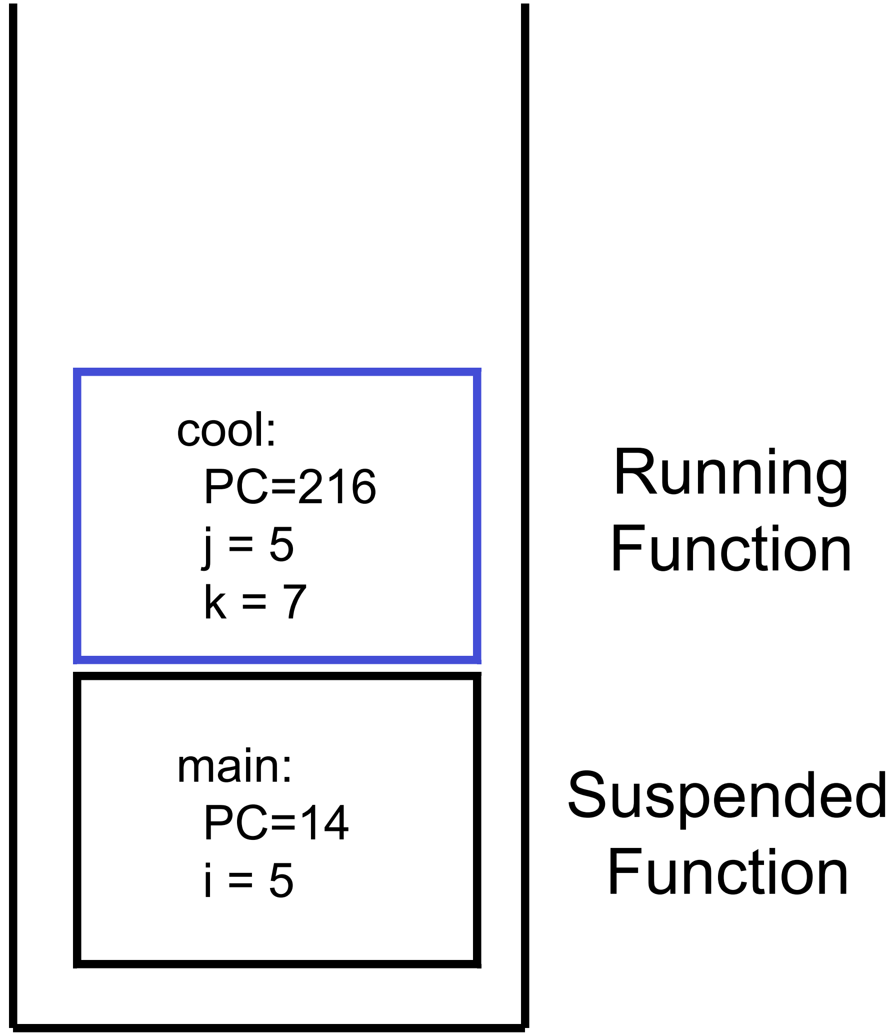


C++ Run-Time Stack

```
PC
       int main()
       {
         int i=5;
         …
14       cool(i);
         …
       }

       void cool(int j)
       {
         int k=7;
         …
216      fool(k);
         …
       }

320    void fool(int m)
       {
         …
       }
```

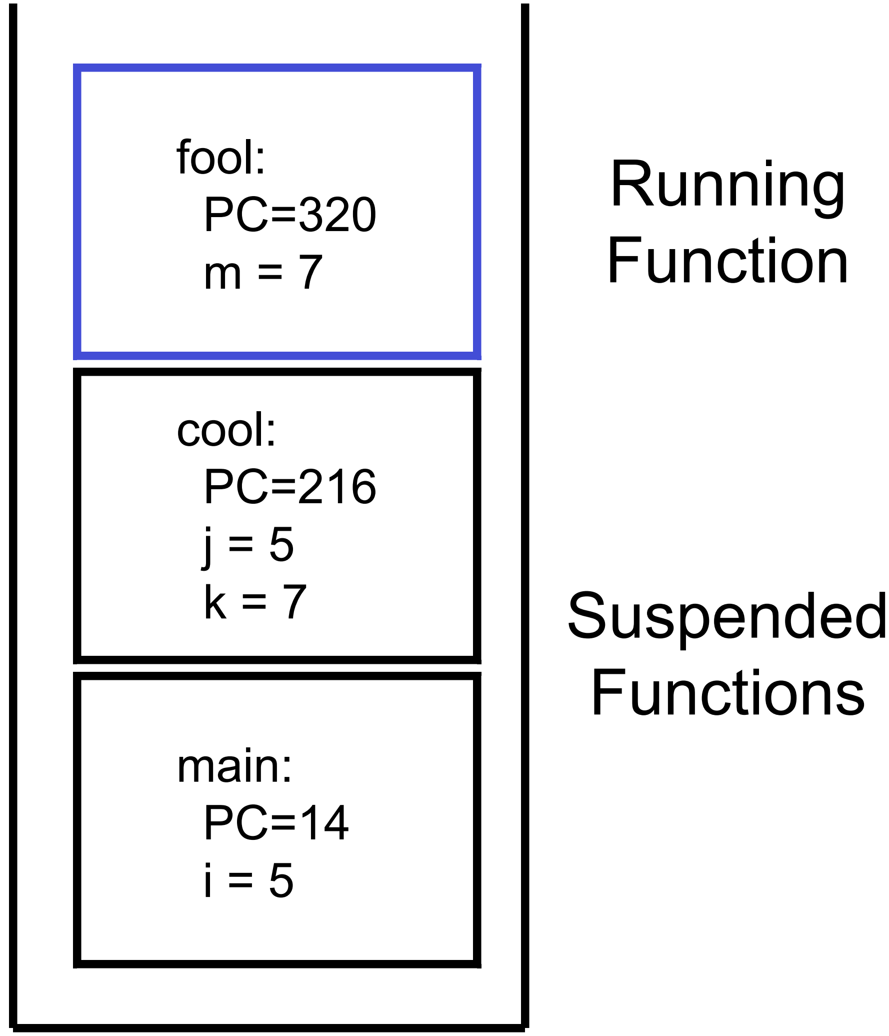


C++ Run-Time Stack

```
PC
        int main()
        {
          int i=5;
          …
14        cool(i);
          …
        }

        void cool(int j)
        {
          int k=7;
          …
216       fool(k);
          …
        }

320     void fool(int m)
        {
          …
        }
```



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

Recursion Example

Factorial Calculation:

n! = 1, if n=0
= n*(n-1)*(n-2)*…*1, if n>0

Or

n! = 1, if n=0
= n*(n-1)!, if n>0 <== **Recursive Definition**

- ***Recursive Definition***
  - A definition in which something is defined in terms of a smaller version of itself

More Recursion Terminology

- ***Base Case***
  - The case for which the solution can be stated nonrecursively
- ***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

Recursive Implementation Example

Factorial(3)

Factorial(3)

3*Factorial(2)

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

Recursive Implementation Example

```
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)

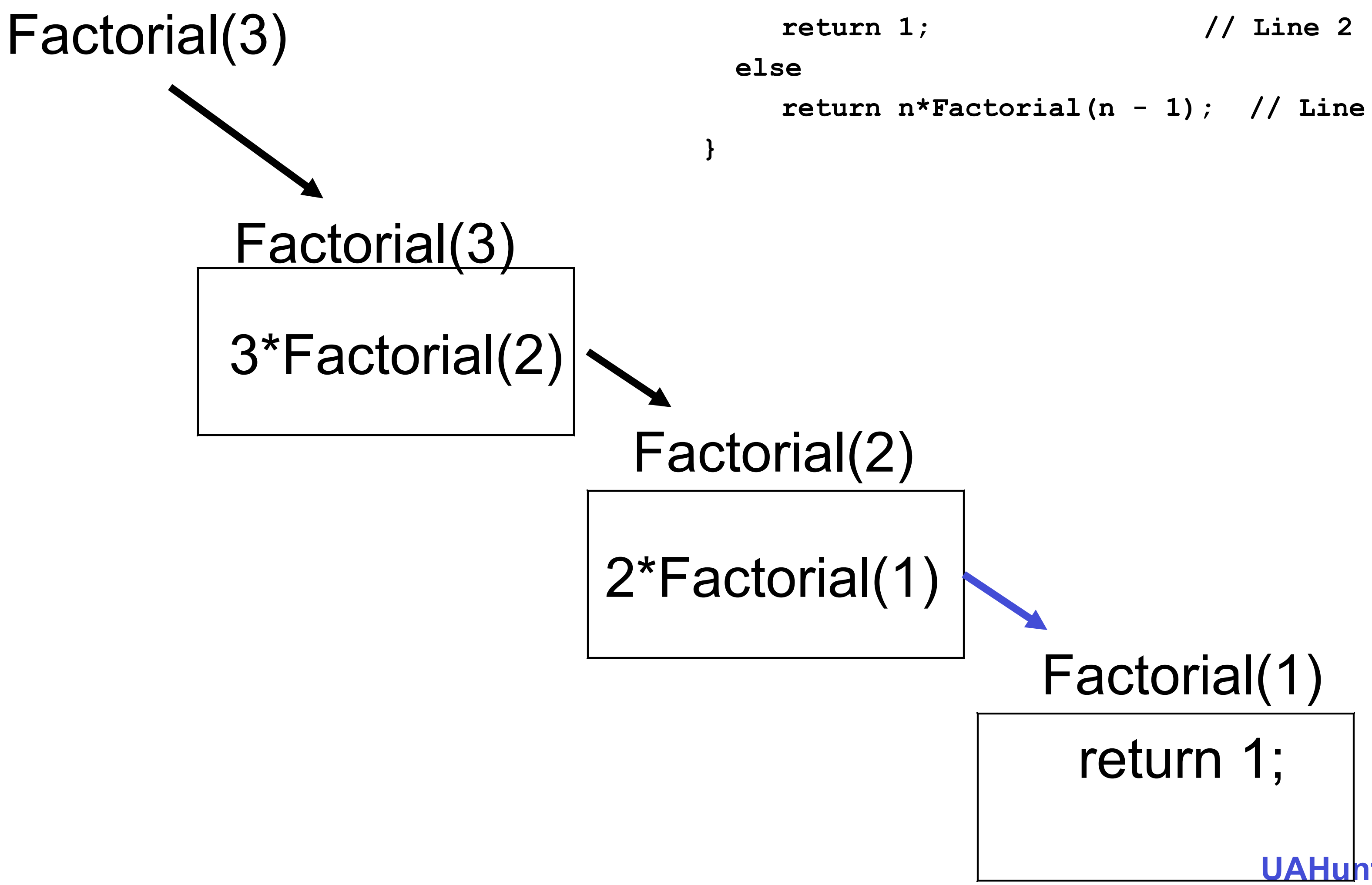
3*Factorial(2)

Factorial(2)

2*Factorial(1)

Recursive Implementation Example

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



UAHuntsville

Recursive Implementation Example

```
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)

Factorial(2)

2*Factorial(1)

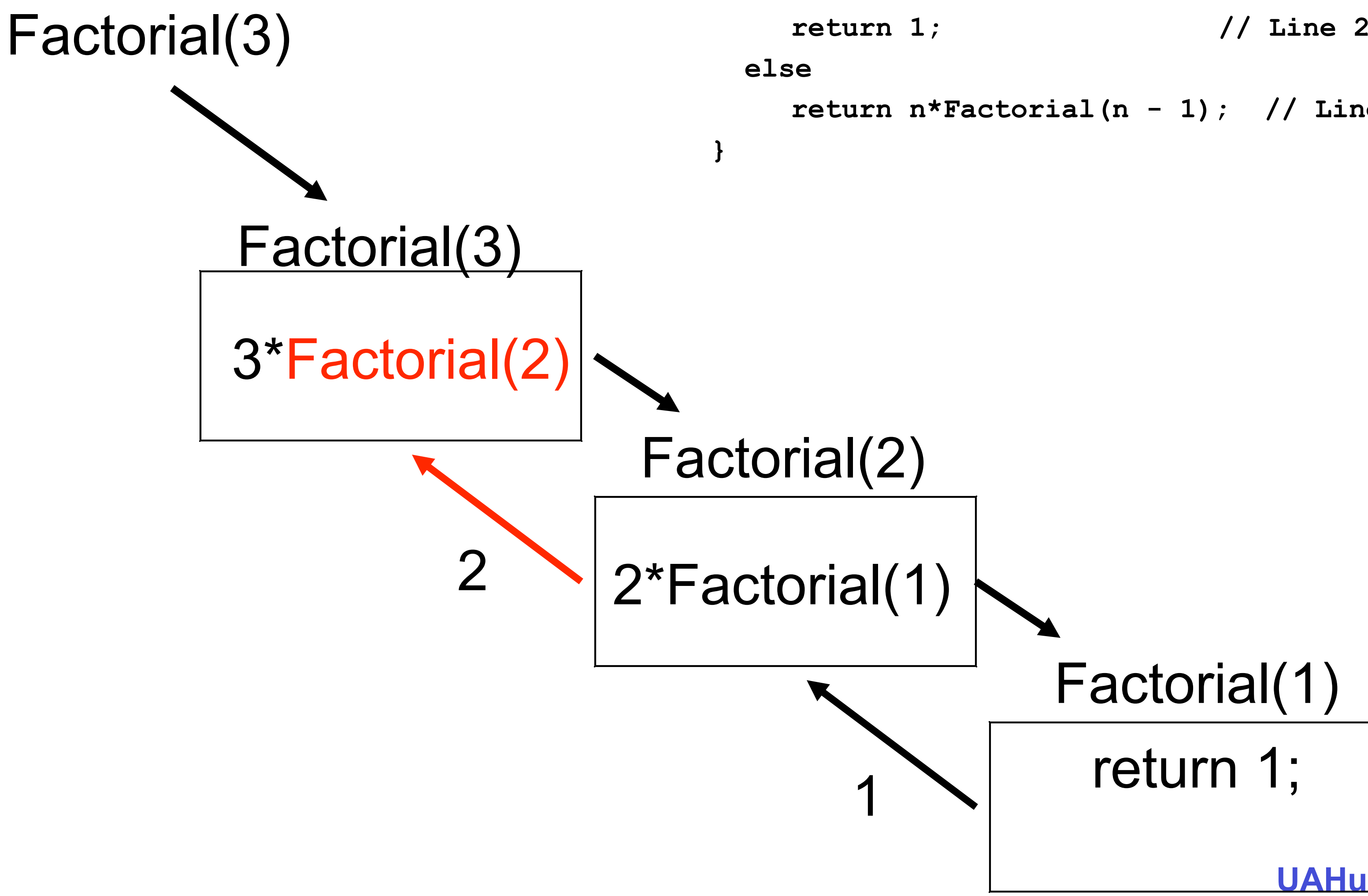
Factorial(1)

return 1;

1

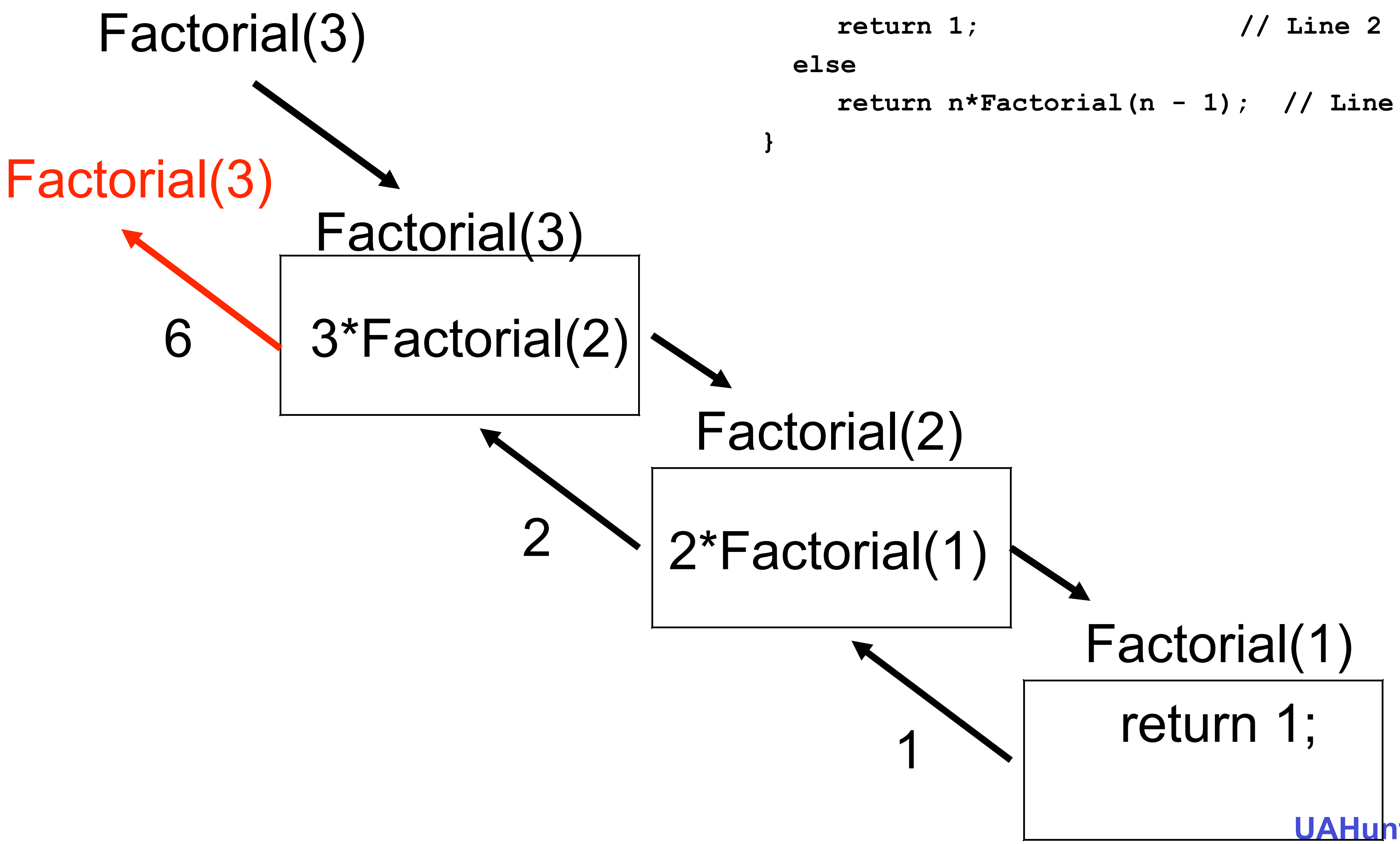
Recursive Implementation Example

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



Recursive Implementation Example

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



Recursive Implementation Example

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

Recursive Implementation Example

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

Proof-By-Induction Procedure

1. Prove that f(n) is true for some value n_0
2. Assume that f(n) is true for some value $n > n_0$
3. Show that f(n+1) is true

Conclude that f(n) is true for all $n >= n_0$

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) equals (N+1)!

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.

Proof-by-Induction Example

```
int SumOfInts(int N)    // Sums integers from 1 through N for N >= 1
{
   if (N == 1)
       return(1);
   else
       return( (N-1) + SumOfInts(N) );    // <= Deliberate error!!
}
```

<u>Correctness Proof</u>:

Prove for N = 1.
Does SumOfInts(1) = 1?   Yes!!

Assume for N. Prove for N+1.
Assume that 1 + 2 + … + N = SumOfInts(N)

Mathematically,

$$\begin{aligned} 1 + 2 + \ldots + N + (N+1) &= (1 + 2 + \ldots + N) + (N+1) \\ &= \text{SumOfInts}(N) + (N+1) \end{aligned}$$

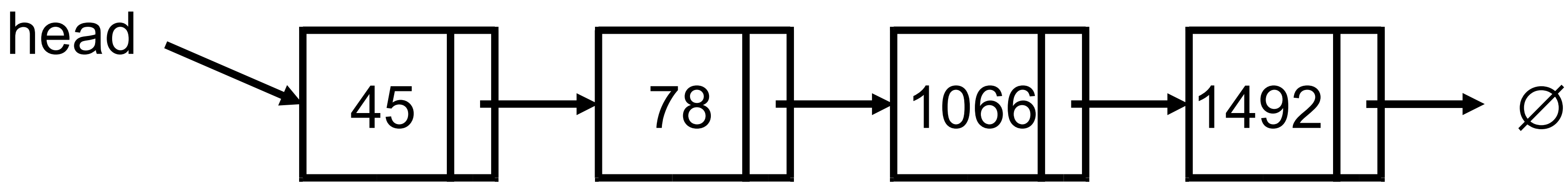
Using the function definition

$$\begin{aligned} \text{SumOfInts}(N+1) &= ((N+1) - 1) + \text{SumOfInts}(N+1) \\ &= N + \text{SumOfInts}(N+1) \end{aligned}$$

==> **N = 0 Inconsistent with our assumption N >= 1, error detected**

UAHuntsville

Recursion with Data Structures




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

Client Code:
```
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

```cpp
//*******  Stack.h Standard Header Information Here **********

#ifndef GSTACK_H
#define GSTACK_H

template <typename ItemType>
struct NodeType;                    // Forward declaration

template <typename ItemType>
class GStack                        // Node-based Stack class
{
 private:
  NodeType<ItemType>* topPtr;       // Top of stack pointer
  void PrintEachItem(NodeType<ItemType>* item); // Prints a each stack item to stdout
  NodeType<ItemType>* PopEachItem(NodeType<ItemType>* item);   // Pops and deallocates each stack item

 public:
  GStack();                         // Default constructor
                                    // Postcondition: Empty stack created

  ~GStack();                        // Destructor


  bool IsEmpty() const;             // Checks to see if stack is empty
                                    // Postcondition: Returns TRUE if empty, FALSE otherwise

  bool IsFull() const;              // Checks to see if stack is full
                                    // Postcondition: Returns TRUE if full, FALSE otherwise

  void Push(ItemType item);         // Adds item to top of stack

  void Pop();                       // Removes top item from stack

  ItemType Top() const;             // Returns a copy of top item on stack
                                    // Postcondition: item still on stack, copy returned

  void MakeEmpty();                 // Removes all items from stack

  void PrintStack();                // Prints all items in stack
};

#endif
```

UAHuntsville

```cpp
//*******  Stack.cpp Standard Header Information Here **********
#include "gstack.h"
#include <cstddef>
#include <new>                                  // for bad_alloc

using namespace std;

template <typename ItemType>
struct NodeType
{
  ItemType info;
  NodeType* next;
};

//****************************************************************

template <typename ItemType>
GStack<ItemType>::GStack()                      // Default constructor
{                                               // Postcondition: Empty stack created
  topPtr = NULL;
}

//****************************************************************

template <typename ItemType>
GStack<ItemType>::~GStack()                     // Destructor
{
  NodeType<ItemType>* tempPtr;

  while ( topPtr != NULL )                      // Deallocate any nodes on the stack
  {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

//****************************************************************

template <typename ItemType>
bool GStack<ItemType>::IsEmpty() const          // Checks to see if stack is empty
{                                               // Postcondition: Returns TRUE if empty, FALSE otherwise
  return (topPtr == NULL);
}
```

UAHuntsville

```cpp
//*******  Stack.cpp continued above **********
template <typename ItemType>
void GStack<ItemType>::Push(ItemType item)      // Adds item to top of stack
{                                               // Precondition: stack is not full
  NodeType<ItemType>* tempPtr = new NodeType<ItemType>;
  tempPtr->info = item;
  tempPtr->next = topPtr;
  topPtr = tempPtr;
}


template <typename ItemType>
void GStack<ItemType>::Pop()                    // Removes top item from stack
{                                               // Precondition: stack is not empty
  NodeType<ItemType>* tempPtr;
  tempPtr = topPtr;
  topPtr = topPtr->next;
  delete tempPtr;
}


template <typename ItemType>
ItemType GStack<ItemType>::Top() const          // Returns a copy of top item on stack
{                                               // Precondition: stack is not empty
  return topPtr->info;                          // Postcondition: item still on stack, copy returned
}


template <typename ItemType>
bool GStack<ItemType>::IsFull() const           // Returns true if there is no room for another ItemType
{                                               //  on the free store; false otherwise.
  NodeType<ItemType>* location;
  try
  {
    location = new NodeType<ItemType>;          // new raises an exception if no memory is available
    delete location;
    return false;
  }
  catch(std::bad_alloc)                         // This catch block processes the bad_alloc exception
  {                                             // should it occur
    return true;
  }
}
```

```
//*******  Stack.cpp continued above **********

template <typename ItemType>
void GStack<ItemType>::PrintStack()               // Prints stack to stdout
{
  PrintEachItem(topPtr);
}


template <typename ItemType>
void GStack<ItemType>::PrintEachItem(NodeType<ItemType>* itemPtr)       // Prints next item on stack to stdout
{
  if (itemPtr == NULL)
  {
    return;
  }
  else
  {
    cout << itemPtr->info << endl;
    PrintEachItem(itemPtr->next);
  }
}
```

```cpp
//*******  Stack.cpp continued above **********


template <typename ItemType>
void GStack<ItemType>::MakeEmpty()              // Removes all items from stack
{
  topPtr = PopEachItem(topPtr);
}



template <typename ItemType>
NodeType<ItemType>* GStack<ItemType>::PopEachItem(NodeType<ItemType>* itemPtr)  // Removes all items from stack
{
  NodeType<ItemType>* tempPtr;

  if ( itemPtr != NULL )
  {
    tempPtr = itemPtr;
    itemPtr = itemPtr->next;
    delete tempPtr;
    itemPtr = PopEachItem(itemPtr);
  }
  return itemPtr;
}
```

Tail Recursion

- A recursive algorithm in which no statements are executed after the return from the recursive call
- Often indicates that an iterative solution would be more direct
- Some compilers are smart enough to identify tail recursion and substitute an iterative solution automatically

```cpp
//***  StackClient.cpp Standard Header Information Here ****
#include "gstack.h"
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
  // ***** Now create and use a stack of integers
  GStack<int> temps;
  ifstream datafile;
  int someTemp;

  datafile.open("June05Temps");

  cout << "Raw Data" << endl;
  datafile >> someTemp;
  while (datafile)
  {
    cout << someTemp << endl;
    if ( !temps.IsFull() )
    {
      temps.Push(someTemp);
    }
    datafile >> someTemp;
  }

  cout << "Test PrintStack" << endl;
  temps.PrintStack();

  cout << "Stack Values" << endl;
  while ( !temps.IsEmpty() )
  {
    cout << temps.Top() << endl;
    temps.Pop();
  }
  datafile.close();

// ***** Now create and use a stack of characters
  GStack<char> text;
  char someChar;
  datafile.open("mytext.txt");

  cout << "Raw Data" << endl;
  datafile >> someChar;
  while (datafile)
  {
    cout << someChar << endl;
    if ( !text.IsFull() )
    {
      text.Push(someChar);
    }
    datafile >> someChar;
  }

  cout << "Test PrintStack" << endl;
  text.PrintStack();

  cout << "Test MakeEmpty" << endl;
  text.MakeEmpty();

  cout << "Stack Values" << endl;
  while ( !text.IsEmpty() )
  {
    cout << text.Top() << endl;
    text.Pop();
  }
  datafile.close();

  return 0;
}
```

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 I 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