C++ Object-Oriented Prog.
Unit 6: Generic Programming

CHAPTER 25: BASIC ALGORITHM 2: RECURSION

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Overview of This Course

Chapter 18 Topics

- Meaning of Recursion
- Base Case and General Case in Recursive Function Definitions
- Writing Recursive Functions with Simple Type Parameters
- Writing Recursive Functions with Array Parameters
- Writing Recursive Functions with Pointer Parameters
- Understanding How Recursion Works

Recursive Function Call

- A recursive call is a function call in which the called function is the same as the one making the call
- In other words, *recursion occurs when a function calls itself!*
- But we need to avoid making an infinite sequence of function calls (infinite recursion)

Finding a Recursive Solution

- A recursive solution to a problem must be written carefully
- The idea is for each successive recursive call to bring you one step closer to a situation in which the problem can easily be solved
- This easily solved situation is called the base case
- Each recursive algorithm must have at least one base case, as well as a general (recursive) case

General format for Many Recursive Functions

```
if   (some easily-solved condition)     // Base case

    solution statement

else                                    // General case

    recursive function call
```

Some examples . . .

Writing a Recursive Function to Find the Sum of the Numbers from 1 to n

DISCUSSION

- The function call Summation(4) should have value 10, because that is 1 + 2 + 3 + 4
- For an easily-solved situation, the sum of the numbers from 1 to 1 is certainly just 1
- So our base case could be along the lines of

```
if  (n == 1) return 1;
```

Writing a Recursive Function to Find the Sum of the Numbers from 1 to n

Now for the general case. . .

- The sum of the numbers from 1 to n, that is,

  1 + 2 + . . . + n   can be written as

- n + the sum of the numbers from 1 to (n - 1),

  that is, n + 1 + 2 + . . . + (n - 1)

  or,   n   +   Summation(n - 1)

- And notice that the recursive call Summation(n - 1) gets us “closer” to the base case of Summation(1)

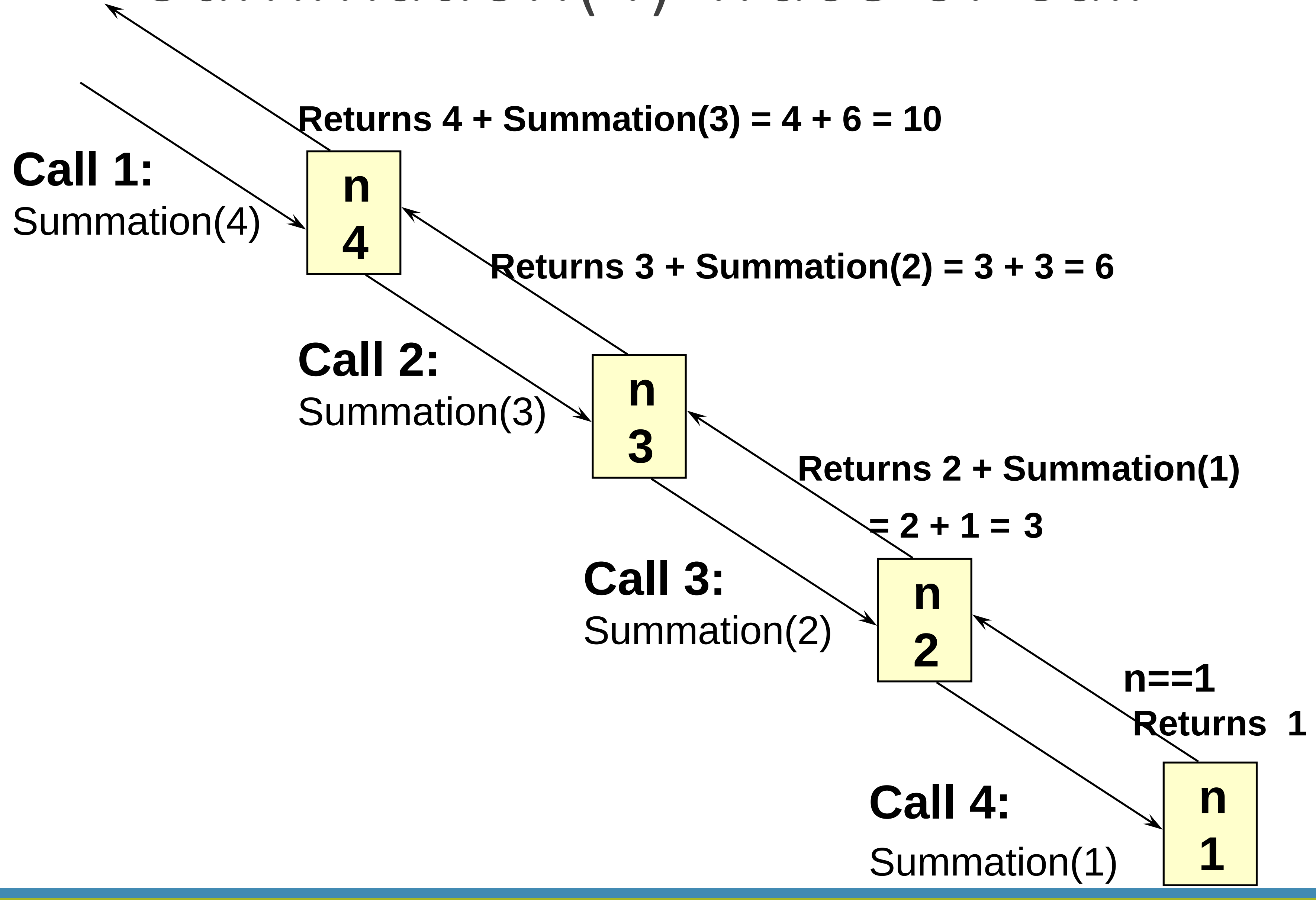
LECTURE 2

Summation and Factorial

Finding the Sum of the Numbers from 1 to n

```
int   Summation (/* in */  int   n)
// Computes the sum of the numbers from 1 to
//  n by adding n to the sum of the numbers
//  from 1 to (n-1)
// Precondition: n is assigned && n > 0
// Postcondition: Return value == sum of
//  numbers from 1 to n
{
    if  (n == 1)  // Base case
          return  1;
   else                     // General case
          return (n + Summation (n - 1));
}
```

Summation(4) Trace of Call



Writing a Recursive Function to Find n Factorial

DISCUSSION

- The function call Factorial(4) should have value 24, because that is 4 * 3 * 2 * 1
- For a situation in which the answer is known, the value of 0! is 1
- So our base case could be along the lines of

```
if  (number == 0) return 1;
```

Writing a Recursive Function to Find Factorial(n)

Now for the general case . . .

- The value of Factorial(n) can be written as n * the product of the numbers from (n - 1) to 1,

  that is,

  $$n \; * \; \underbrace{(n - 1) \; * \; . \; . \; . \; * \; 1}$$

  or, $\quad n \; * \; \mathbf{Factorial(n - 1)}$

- And notice that the recursive call Factorial(n - 1) gets us "closer" to the base case of Factorial(0)

Demo Program:

Summation

Go Dev C++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Summation\Summation.exe
55

--------------------------------
Process exited after 0.05625 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
using namespace std;

// Computes the sum of the numbers from 1 to
//  n by adding n to the sum of the numbers
//  from 1 to (n-1)
// Precondition: n is assigned && n > 0
// Postcondition: Return value == sum of
//  numbers from 1 to n
int   Summation (/* in */  int   n)
{
    if  (n == 1)     // Base case
        return  1;
    else             // General case
        return (n + Summation (n - 1));
}

int main(int argc, char** argv) {
    cout << Summation(10) << endl;
    return 0;
}
```

Recursive Solution

```
int    Factorial ( int    number)
// Pre: number is assigned and number >= 0
{
    if   (number == 0)      // Base case
         return  1;
   else                     // General case
         return
           number * Factorial (number - 1);
}
```

Demo Program:

Factorial

Go Dev C++!!!

```
#include <iostream>
using namespace std;

// Pre: number is assigned and number >= 0
int   Factorial ( int   number) {
    if  (number == 0)   // Base case
        return  1;
    else                  // General case
        return
        number * Factorial (number - 1);
}

int main(int argc, char** argv) {
    cout << Factorial(5) << endl;
    return 0;
}
```

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Factorial\Factorial.exe

```
120

--------------------------------
Process exited after 0.05924 seconds with return value 0
Press any key to continue . . .
```

LECTURE 3

Power

Another Example Where Recursion Comes Naturally

From mathematics, we know that

$2^0 = 1$ and $2^5 = 2 * 2^4$

In general,

$x^0 = 1$ and $x^n = x * x^{n-1}$

for integer x, and integer n > 0

Here we are defining x^n recursively, in terms of x^{n-1}

```
// Recursive definition of power function
int  Power ( int   x,   int   n)

// Pre:  n >= 0; x, n are not both zero
// Post: Return value == x raised to the
//  power n.

{
    if  (n == 0)
        return  1; // Base case

    else            // General case
        return ( x * Power (x, n-1));
}
```

Of course, an alternative would have been to use an iterative solution instead of recursion

Demo Program:

Power1

Go Dev C++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Power1\Power1.exe
64

--------------------------------
Process exited after 0.1564 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
using namespace std;

// Recursive definition of power function
// Pre:  n >= 0; x, n are not both zero
// Post: Return value == x raised to the
//  power n.
int  Power ( int   x,   int   n){
    if  (n == 0)
        return  1; // Base case

    else              // General case
        return ( x * Power (x, n-1));
}

int main(int argc, char** argv) {
    cout << Power (2, 6) << endl;
    return 0;
}
```

Extending the Definition

- *What is the value of* 2^{-3} *?*
- Again from mathematics, we know that it is

$$2^{-3} = 1 / 2^3 = 1 / 8$$

- In general,

$x^n = 1/ x^{-n}$ for non-zero x, and integer $n < 0$

- Here we again defining x^n recursively, in terms of x^{-n} when $n < 0$

```
// Recursive definition of power function
float  Power ( /* in */  float   x,
               /* in */  int   n)
// Pre:  x  !=  0 && Assigned(n)
// Post: Return value == x raised to the power n

{
    if  (n == 0)       // Base case

        return  1;

    else  if  (n > 0) // First  general  case

        return ( x * Power (x, n - 1));
    else                   // Second general case

        return ( 1.0 / Power (x, - n));

}
```

Demo Program:

Power2

Go Dev C++!!!

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Power2\Power2.exe

```
8
0.125

--------------------------------
Process exited after 0.1133 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
using namespace std;
// Recursive definition of power function
// Pre:  x  !=  0 && Assigned(n)
// Post: Return value == x raised to the power n
float Power(/*in*/ float x,  /*in*/ int n){
    if  (n == 0)        // Base case
        return  1;
    else  if  (n > 0) // First  general  case
        return ( x * Power (x, n - 1));
    else                    // Second general case
        return ( 1.0 / Power (x, - n));
}

int main(int argc, char** argv) {
    cout << Power(2, 3) << endl;
    cout << Power(2, -3) << endl;
    return 0;
}
```

LECTURE 4

Print Stars

The Base Case Can Be “Do Nothing”

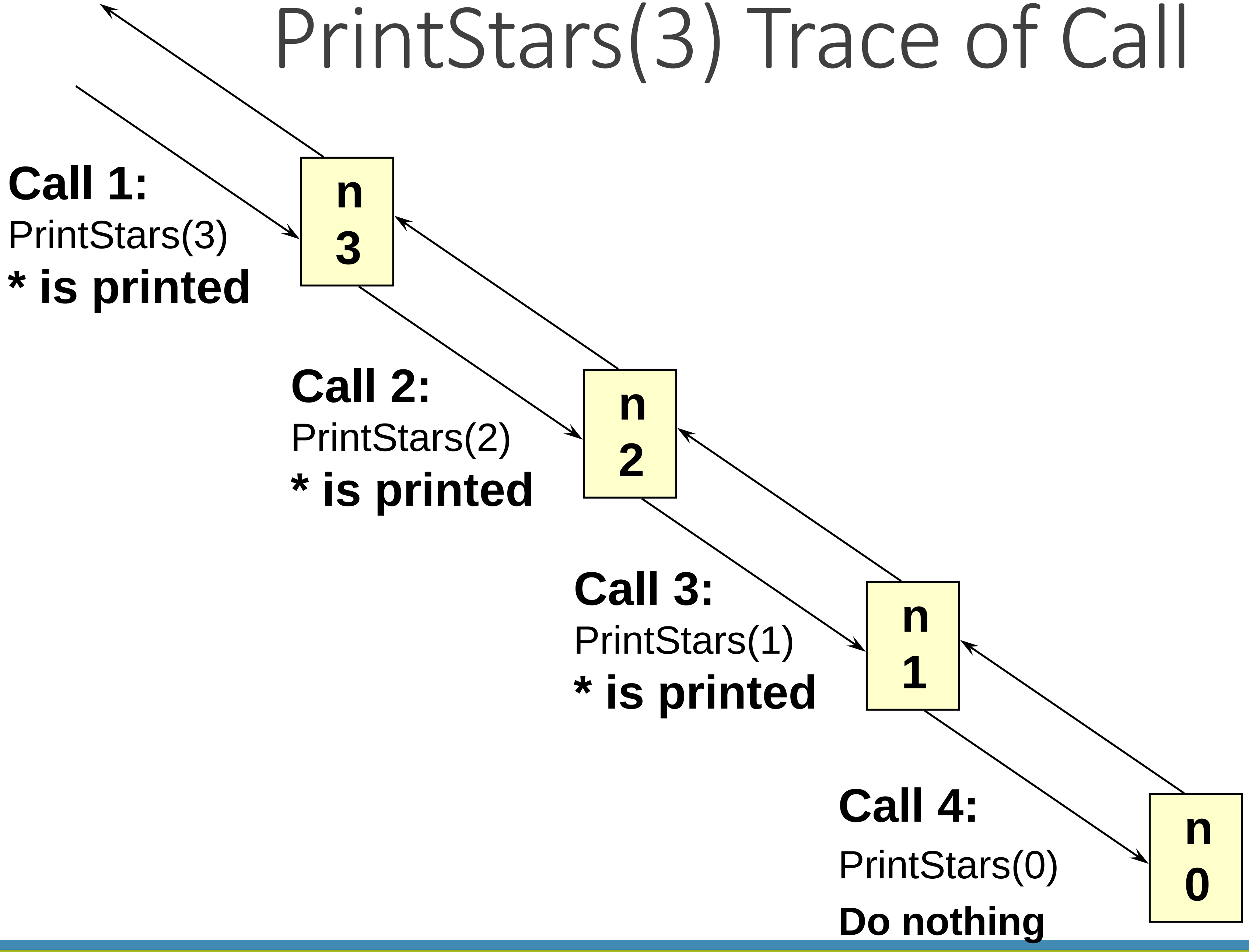
```
void   PrintStars (/* in */  int   n)
// Prints n asterisks, one to a line
// Precondition:     n is assigned
// Postcondition:
//    IF n <= 0, n stars have been written
//    ELSE call PrintStarg
{
    if  (n <= 0)  // Base case: do nothing
    else
    {
        cout   <<  '*'  <<  endl;
        PrintStars (n - 1);
    }
}
```

// Can rewrite as . . .

Recursive Void Function

```
void  PrintStars (/* in */  int   n)
//   Prints n asterisks, one to a line
//   Precondition:  n is assigned
//   Postcondition:
//        IF n > 0, call PrintSars
//        ELSE n stars have been written
{
    if  (n > 0)  // General case
    {
        cout  <<  '*'  <<  endl;
        PrintStars (n - 1);
    }
   // Base case is empty else-clause
}
```

PrintStars(3) Trace of Call



Demo Program:

PrintStars

Go Dev C++!!!

Select C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\PrintStars\PrintStars.exe

```
*
*
*
*
*
*
*
*
*
*

--------------------------------
Process exited after 0.2028 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
using namespace std;

//   Prints n asterisks, one to a line
//   Precondition:  n is assigned
//   Postcondition:
//       IF n > 0, call PrintSars
//       ELSE n stars have been written
void  PrintStars (/* in */  int   n){
    if  (n > 0)  // General case
    {
        cout  << "*" <<  endl;
        PrintStars (n - 1);
    }
     // Base case is empty else-clause
}

int main(int argc, char** argv) {
    PrintStars(10);
    return 0;
}
```

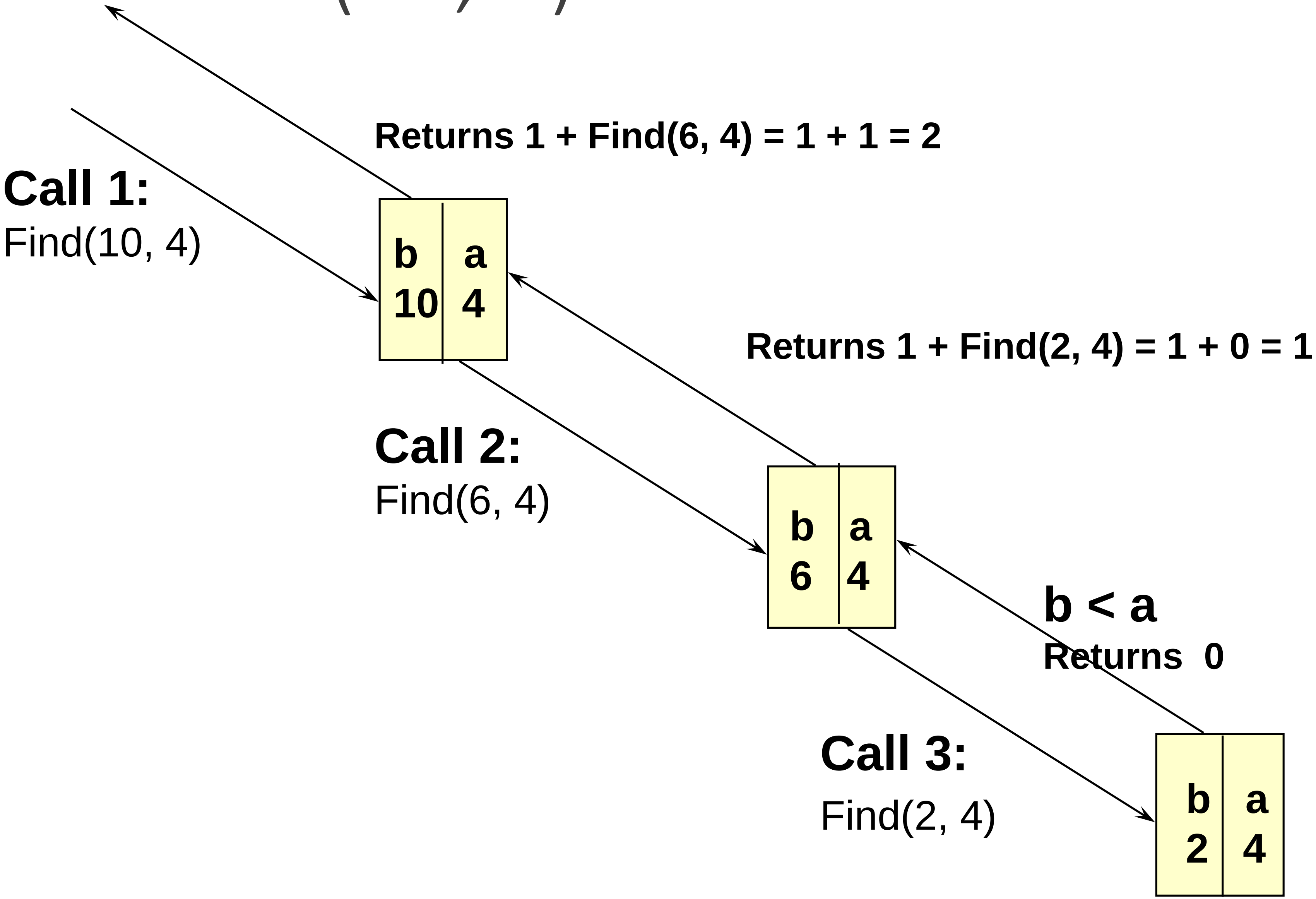
LECTURE 4

Tracing Recursive Programs

Recursive Mystery Function

```
int Find(/* in */ int b, /* in */ int a)
// Simulates a familiar integer operator
// Precondition: a is assigned && a > 0
//    &&  b is assigned  &&    b >= 0
// Postcondition: Return value  ==   ???
{
    if  (b < a)          // Base case
        return  0;
    else                 // General case
        return (1 + Find (b - a, a));
}
```

Find(10, 4) Trace of Call



```cpp
#include <iostream>
using namespace std;
// Simulates a familiar integer operator
// Precondition: a is assigned && a > 0
//  &&  b is assigned  &&    b >= 0
// Postcondition: Return value  ==   ???
int Find(/* in */ int b, /* in */ int a){
    cout << "Find("<< b << ", " << a << ")"<< endl;
    if  (b < a)        // Base case
        return  0;
    else               // General case
    {
      return (1 + Find (b - a, a));
    }
}
int main(int argc, char** argv) {
    cout << "Find(10, 4)=" << Find(10, 4) << endl;
    return 0;
}
```

```
Find(10, 4)
Find(6, 4)
Find(2, 4)
Find(10, 4)=2
```

LECTURE 4

Recursive Function Over Data Structures

Recursive algorithms with Structured Variables

- Recursive case: a smaller structure (rather than a smaller value)
- Base case: there are no values left to process in the structure

Printing the contents of a one-dimensional array

Recursive case:

```
IF more elements
 Print the values of the first element
 Print array of n-1 elements
```

Base case:

```
Do nothing
```

Printing the contents of a one-dimensional array

```
void Print( /* in */ const int data[], // Array to be printed
            /* in */       int first,  // Index of first element
            /* in */       int last )  // Index of last element
{
   if (first <= last){     // Recursive case
       cout << data[first] << endl;
       Print(data, first+1, last);
   }
   // Empty else-clause is the base-case
}
```

Print (data, 0, 2) Trace

Call 1:
Print (data, 0, 2)
data[0] printed

first 0
last 2

Call 2:
Print (data, 1, 2)
data[1] printed

first 1
last 2

Call 3:
Print (data, 2, 2)
data[2] printed

first 2
last 2

Call 4:
Print (data, 3, 2)

first 3
last 2

NOTE: data address 6000 is also passed **Do nothing**

Demo Program:

Print.cpp

Go Dev C++!!!

```cpp
#include <iostream>
using namespace std;
void Print(const int data[], int first, int last )
{
        if (first <= last){    // Recursive case
          cout << data[first] << endl;
          Print(data, first+1, last);
        }
        // Empty else-clause is the base-case
}

int main(int argc, char** argv) {
    int data[3] = {0, 1, 2};
    Print(data, 0, 2);
    return 0;
}
```

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Print\Print.exe

```
0
1
2

--------------------------------
Process exited after 0.2101 seconds with return value 0
Press any key to continue . . .
```

LECTURE 4

Tail Recursion

Tail Recursion

- Once the deepest call was reached, each the calls before it returned without doing anything.
- Tail recursion: No statements are executed after the return from the recursive call to the function
- Tail recursion often indicates that the problem could be solved more easily using iteration. (It is easy to translate tail recursion into iteration.)

Writing a Recursive Function to Print Array Elements in Reverse Order

DISCUSSION

For this task, we will use the prototype:

```
void PrintRev(const  int  data[ ], int  first,  int  last);
```

6000

data[0]	data[1]	data[2]	data[3]
74	**36**	**87**	**95**

The call

```
PrintRev (data, 0, 3);
```

should produce this output: 95 87 36 74

Base Case and General Case

- A base case may be a solution in terms of a "smaller" array Certainly for an array with 0 elements, there is no more processing to do
- The general case needs to bring us closer to the base case situation if the length of the array to be processed decreases by 1 with each recursive call, we eventually reach the situation where 0 array elements are left to be processed
- In the general case, we could print either the first element, that is, data[first or we could print the last element, that is, data[last]
- Let's print data[last]: After we print data[last], we still need to print the remaining elements in reverse order

Using Recursion with Arrays

```
int PrintRev (
   /* in */ const int data [ ],// Array to be printed
   /* in */       int  first,  // Index of first element
  /* in */        int  last ) // Index of last element
// Prints items in data [first..last] in reverse order
// Precondition: first assigned  &&  last assigned
//     && if first <= last, data [first..last] assigned
{
     if  (first  <=  last)      // General case
     {
          cout << data[last]  << "    "; // Print last
          PrintRev(data, first, last - 1); //Print rest
     }           // Base case is empty else-clause
}
```

PrintRev(data, 0, 2) Trace

Call 1:
PrintRev(data, 0, 2)
data[2] printed

first 0
last 2

Call 2:
PrintRev(data, 0, 1)
data[1] printed

first 0
last 1

Call 3:
PrintRev(data, 0, 0)
data[0] printed

first 0
last 0

Call 4:
PrintRev(data, 0, -1)

first 0
last -1

NOTE: data address 6000 is also passed **Do nothing**

Another solution

```
void PrintRev( /* in */ const int data[],
                 // Array to be printed
           /* in */    int first,
                 // Index of first element
           /* in */        int last )
                 // Index of last element
{
   if (first <= last)    // Recursive case
   {
    Print(data, first+1, last);
    cout << data[first] << endl;
   }
   // Empty else-clause is the base-case
}
```

Demo Program:

PrintRev.cpp

Go Dev C++!!!

```cpp
#include <iostream>
using namespace std;
void PrintRev(const int data[], int first, int last ){
    if (first <= last)  // Recursive case
    {
        PrintRev(data, first+1, last);
        cout << data[first] << endl;
    }
    // Empty else-clause is the base-case
}

int main(int argc, char** argv) {
    int data[3] = {0, 1, 2};
    PrintRev(data, 0, 2);
    return 0;
}
```

Select C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\PrintRev\PrintRev.exe

```
2
1
0

--------------------------------
Process exited after 0.1133 seconds with return value 0
Press any key to continue . . .
```

LECTURE 4

Why Recursion?

Why use recursion?

- These examples could all have been written more easily using iteration
- However, for certain problems the recursive solution is the most natural solution
- This often occurs when structured variables are used

Remember The iterative solution uses a loop, and the recursive solution uses a selection statement

Recursion with Linked Lists

- For certain problems the recursive solution is the most natural solution
- This often occurs when pointer variables are used

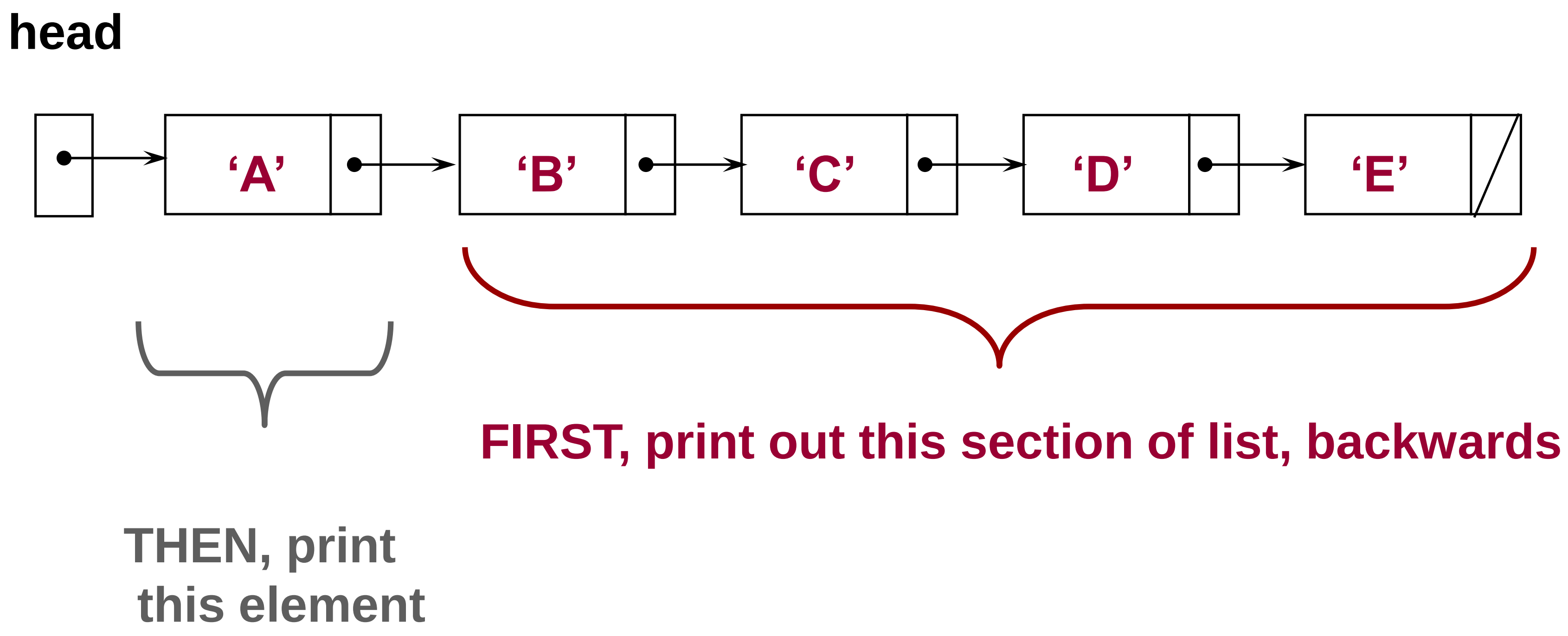
struct NodeType

```
typedef  char  ComponentType;

struct  NodeType
{
   ComponentType  component;
   NodeType*      link;
}

NodeType*  head;
```

`RevPrint(head);`



Base Case and General Case

- A base case may be a solution in terms of a “smaller” list Certainly for a list with 0 elements, there is no more processing to do
- Our general case needs to bring us closer to the base case situation
- If the number of list elements to be processed decreases by 1 with each recursive call, the smaller remaining list will eventually reach the situation where 0 list elements are left to be processed
- In the general case, we print the elements of the (smaller) remaining list in reverse order and then print the current element

Using Recursion with a Linked List

```
void  RevPrint (NodeType*   head){
// Pre:  head points to an element of a list
// Post: All elements of list pointed to by head have
//    been printed in reverse order.
    if (head != NULL){                // General case
        RevPrint (head-> link); // Process the rest
        // Print currrent
        cout << head->component << endl;
    }
   // Base case : if the list is empty, do nothing
}
```

Demo Program:
RecursiveList.cpp

Go Dev C++!!!

```cpp
#include <iostream>
using namespace std;
typedef  char  ComponentType;
struct  NodeType{
   ComponentType  component;
   NodeType*       link;
   NodeType(ComponentType c, NodeType* ptr):component(c), link(ptr){}
};

// Pre:  head points to an element of a list
// Post: All elements of list pointed to by head have
//    been printed in reverse order.
void  RevPrint (NodeType*   head){
      if (head != NULL){             // General case
          RevPrint (head-> link); // Process the rest
          // Print currrent
          cout << head->component << endl;
      }
        // Base case : if the list is empty, do nothing
}
```

```cpp

int main(int argc, char** argv) {
    NodeType*  head;
    NodeType *a = new NodeType('A', NULL);
    NodeType *b = new NodeType('B', NULL);
    NodeType *c = new NodeType('C', NULL);
    NodeType *d = new NodeType('D', NULL);
    NodeType *e = new NodeType('E', NULL);
    a->link = b;
    b->link = c;
    c->link = d;
    d->link = e;

    head = a;
    RevPrint(head);
    return 0;
}
```

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\

```
E
D
C
B
A

--------------------------------
Process exited after 0.06876 seconds with return value 0
Press any key to continue . . .
```

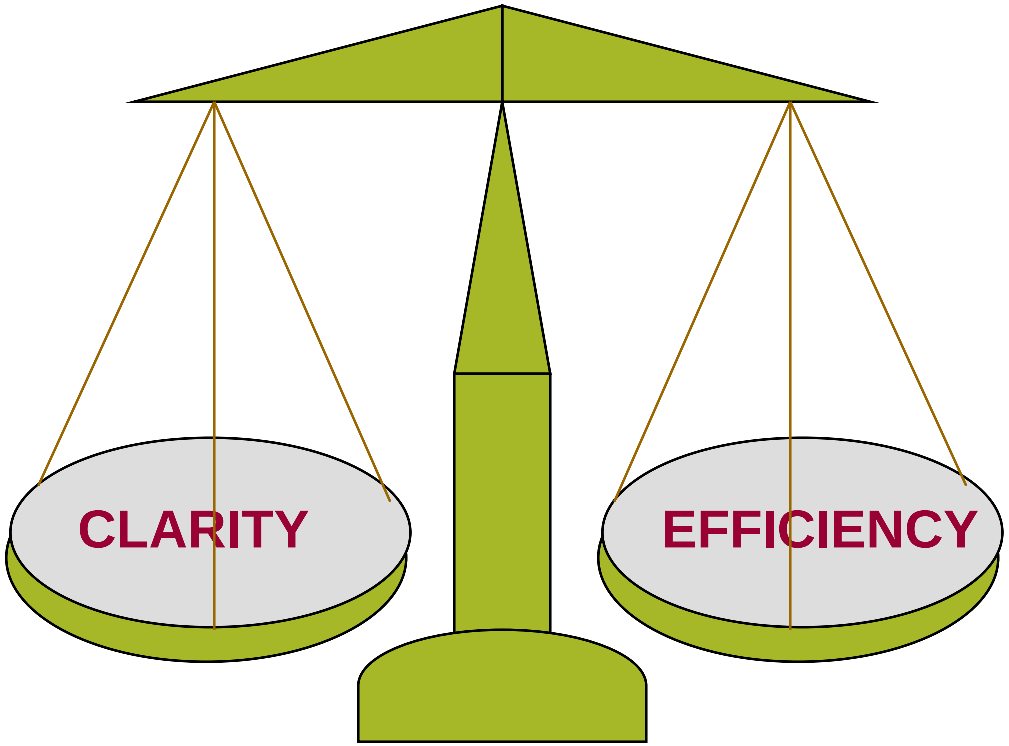
LECTURE 4

Recursion Versus Iteration

Recall that . . .

- Recursion occurs when a function calls itself (directly or indirectly)
- Recursion can be used in place of iteration (looping)
- Some functions can be written more easily using recursion

Recursion or Iteration?



What is the value of `Rose(25)`?

```
int  Rose (int  n){
    if  (n == 1)  // Base case
        return  0;
    else                  // General case
        return (1 + Rose(n / 2));
}
```

Finding the Value of `Rose(25)`

Rose(25)	*the original call*
= 1 + Rose(12)	*first recursive call*
= 1 + (1 + Rose(6))	*second recursive call*
= 1 + (1 + (1 + Rose(3)))	*third recursive call*
= 1 + (1 + (1 + (1 + Rose(1))))	*fourth recursive call*
= 1 + 1 + 1 + 1 + 0	
= 4	

Writing Recursive Functions

- There must be at least one base case and at least one general (recursive) case--the general case should bring you "closer" to the base case
- The arguments(s) in the recursive call cannot all be the same as the formal parameters in the heading, otherwise, infinite recursion would occur
- In function `Rose()`, the base case occurred when `(n == 1)` was true--the general case brought us a step closer to the base case, because in the general case the call was to `Rose(n/2)`, and the argument `n/2` was closer to 1 (than `n` was)

Demo Program:

Rose.cpp

Go Dev C++!!!

```
#include <iostream>
using namespace std;
int  Rose (int  n){
     if  (n == 1)  // Base case
         return  0;
     else                // General case
         return (1 + Rose(n / 2));
}

int main(int argc, char** argv) {
     cout << Rose(25) << endl;
     return 0;
}
```

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\Rose\Rose.exe

```
4

--------------------------------
Process exited after 0.07542 seconds with return value 0
Press any key to continue . . .
```

LECTURE 4

Stack Diagram Activation Records

When a function is called...

- A transfer of control occurs from the calling block to the code of the function
- It is necessary that there be a return to the correct place in the calling block after the function code is executed
- This correct place is called the return address

When any function is called, the run-time stack is used--activation record for the function call is placed on the stack

Stack Activation Record

- The activation record (stack frame) contains the return address for this function call, the parameters, local variables, and space for the function's return value (if non-void)
- The activation record for a particular function call is popped off the run-time stack when the final closing brace in the function code is reached, or when a return statement is reached in the function code

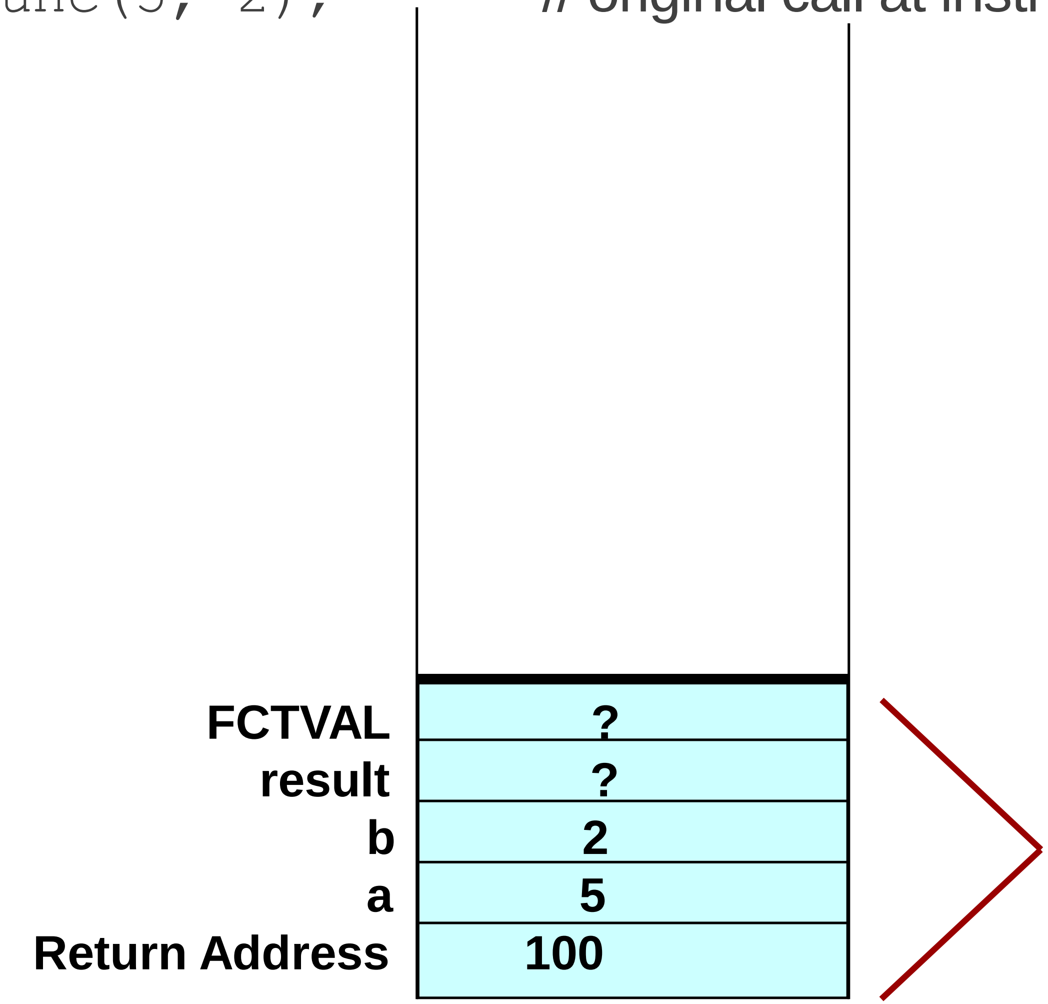
At this time the function's return value, if non-void, is brought back to the calling block return address for use there

```
// Another recursive function

int Func (/* in */  int  a, /* in */  int  b)
//  Pre:  Assigned(a)  &&  Assigned(b)
//  Post: Return value == ??
{
    int  result;
    if  (b == 0)         // Base case
        result = 0;
    else if  (b > 0) // First general case
        result = a + Func (a, b - 1);
          // Say location 50
    else            // Second general case
        result = Func (- a, - b);
          // Say location 70
    return  result;
}
```

Run-Time Stack Activation Records

`x = Func(5, 2);` // original call at instruction 100



original call
at instruction 100
pushes on this record
for Func(5,2)

Run-Time Stack Activation Records

```
x = Func(5, 2);
```

// original call at instruction 100

FCTVAL	?
result	?
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

call in Func(5,2) code at instruction 50 pushes on this record for Func(5,1)

record for Func(5,2)

Run-Time Stack Activation Records

```
x = Func(5, 2);
```

// original call at instruction 100

FCTVAL	**?**
result	**?**
b	**0**
a	**5**
Return Address	**50**
FCTVAL	**?**
result	**5+Func(5,0) = ?**
b	**1**
a	**5**
Return Address	**50**
FCTVAL	**?**
result	**5+Func(5,1) = ?**
b	**2**
a	**5**
Return Address	**100**

call in Func(5,1) code at instruction 50 pushes on this record for Func(5,0)

record for Func(5,1)

record for Func(5,2)

Run-Time Stack Activation Records

`x = Func(5, 2);` // original call at instruction 100

FCTVAL	0
result	0
b	0
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,0) = ?
b	1
a	5
Return Address	50
FCTVAL	?
result	5+Func(5,1) = ?
b	2
a	5
Return Address	100

record for Func(5,0) is popped first with its FCTVAL

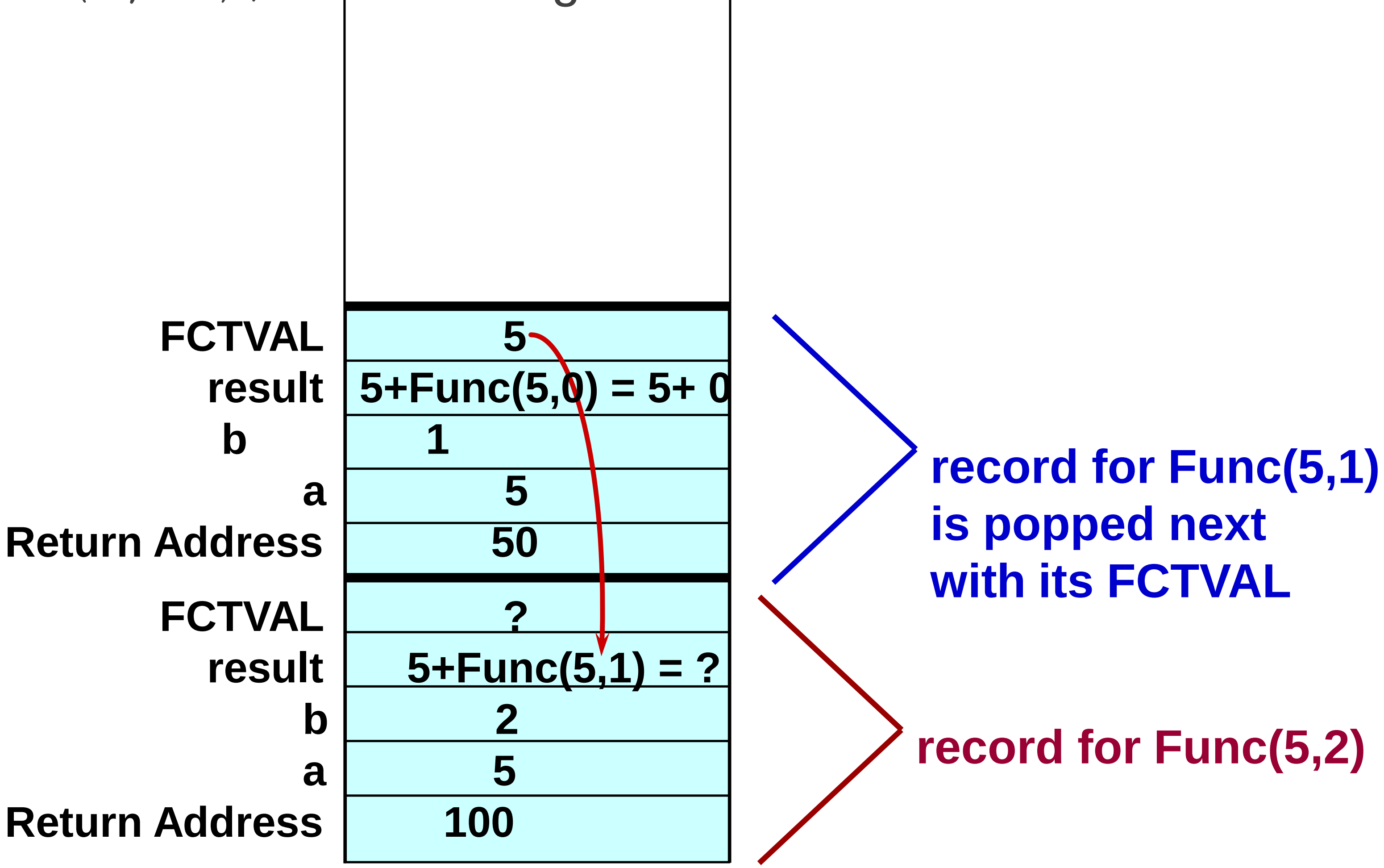
record for Func(5,1)

record for Func(5,2)

Run-Time Stack Activation Records

```
x = Func(5, 2);
```

// original call at instruction 100



Run-Time Stack Activation Records

```
x = Func(5, 2);
```

// original call at instruction 100

FCTVAL	10
result	5+Func(5,1) = 5+5
b	2
a	5
Return Address	100

record for Func(5,2) is popped last with its FCTVAL

Show Activation Records for these calls

x = Func(- 5, - 3);

x = Func(5, - 3);

What operation does Func(a, b) simulate?

Demo Program:
Func.cpp

Go Dev C++!!!

```cpp
#include <iostream>
using namespace std;

//  Pre:  Assigned(a)  &&  Assigned(b)
//  Post: Return value == ??
int Func (/* in */  int  a, /* in */  int  b){
    int  result;
    if  (b == 0)        // Base case
        result = 0;
    else if  (b > 0) // First general case
        result = a + Func (a, b - 1);
          // Say location 50
    else                 // Second general case
        result = Func (- a, - b);
          // Say location 70
    return  result;
}

int main(int argc, char** argv) {
    cout << Func(-5, -3) << endl ;
    cout << Func(5, -3) << endl ;
    return 0;
}
```

C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\

```
15
-15
```

LECTURE 4

Recursive Function on Arrays

Write a function . . .

Write a function that takes an array `a` and two subscripts, `low` and `high` as arguments, and returns the sum of the elements

```
a[low]+..+ a[high]
```

Write the function two ways - - one using iteration and one using recursion

For your recursive definition's base case, for what kind of array do you know the value of `Sum(a, low, high)` right away?

```
// Recursive definition

int  Sum ( /* in */  const  int  a[ ],
            /* in */   int   low,
            /* in */   int   high)

//  Pre:  Assigned(a[low..high]) && low <= high
//  Post: Return value == sum of items a[low..high]
{
    if  (low == high) // Base case

        return  a [low];

    else                 // General case

        return  a [low] + Sum(a, low + 1, high);

}
```

```
// Iterative definition

int  Sum ( /* in */  const  int  a[ ],
           /* in */   int   high)

//  Pre:  Assigned(a[0..high])
//  Post: Return value == sum of items a[0..high]
{
    int sum = 0;
    for (int index= 0; index <= high; index++)

        sum = sum + a[index];
    return sum;

}
```

Write a function . . .

- Write a LinearSearch that takes an array a and two subscripts, low and high, and a key as arguments R
- It returns true if key is found in the elements a[low...high]; otherwise, it returns false
- Write the function two ways - - using iteration and using recursion

For your recursive definition's base case(s), for what kinds of arrays do you know the value of `LinearSearch(a, low, high, key)` right away?

```
// Recursive definition

bool LinearSearch
    (/* in */ const int a[ ],
      /* in */    int    low,
      /* in */    int    high,
      /* in */    int    key)
// Pre: Assigned(a[low..high])
// Post: IF (key in a[low..high])
//               Return value is true,
//          else return value is false
{
    if   (a [ low ] == key) // Base case

        return true;

    else if (low == high) // Second base case

        return  false;

    else                      // General case

        return
          LinearSearch(a, low + 1, high, key);

}
```

```
// Iterative definition

int   LinearSearch  (  /* in */  const  int  a[ ] ,  /* in */   int   low ,
                        /* in */  int   high,          /* in */   int   key )

    //  Pre:        Assigned( a [ low . . high ] )  &&  low <= high
    //              &&  Assigned (key)
    //  Post:     (key in a [ low . . high] ) --> a[FCTVAL] == key
    //              &&   (key not in a [ low . . high] ) -->FCTVAL == -1

{
        int i = low;
        while (i <= high && key != a[i]) i++;
        return (i <= high);

}
```

Function `BinarySearch()`

- BinarySearch that takes sorted array a, and two subscripts, low and high, and a key as arguments
- It returns true if key is found in the elements a[low...high], otherwise, it returns false
- BinarySearch can also be written using iteration or recursion, but it is an inherently recursive algorithm

```
x = BinarySearch(a, 0, 14, 25);
```

low high key

subscripts

subscripts	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
array	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28

16 18 20 22 24 26 28

24 26 28

24

NOTE: denotes element examined

```
// Recursive definition

bool BinarySearch (/* in */ const int  a[ ],
                   /* in */ int  low,
                   /* in */ int  high,
                   /* in */ int   key)
// Pre:  a[low .. high] in ascending order && Assigned (key)
// Post: IF (key in a[low . . high]), return value is true
//   otherwise return value is false
{
    int  mid;
    if  (low > high)
        return false;
    else
    {
        mid = (low + high) / 2;
        if (a [ mid ] == key)
            return  true;
        else if (key < a[mid]) // Look in lower half
            return BinarySearch(a, low, mid-1, key);
        else  // Look in upper half
            return BinarySearch(a, mid+1, high, key);
    }
}
```

```
// Iterative definition

int  BinarySearch   (  /* in */  const  int  a[ ] ,  /* in */   int   low ,
                       /* in */   int   high,         /* in */   int   key )
 //    Pre:   a [ low . . high ] in ascending order  &&  Assigned (key)
 //    Post:     (key in a [ low . . high] ) --> a[FCTVAL] == key
 //          &&   (key not in a [ low . . high] ) -->FCTVAL == -1
{
          int mid;
       while ( low <= high )            {             // more to examine
              mid = (low + high) / 2 ;

              if ( a [ mid ] == key )          // found at mid

                    return  mid  ;

              else  if ( key < a [ mid ] )    //  search in lower half
                 high = mid - 1 ;
           else                          //  search in upper half
                    low = mid + 1 ;
       }
       return -1 ;                          // key was not found

}
```

Write a function . . .

- `Minimum` that takes an array `a` and the `size` of the array as arguments, and returns the smallest element of the array, that is, it returns the smallest value of `a[0] . . . a[size-1]`

- write the function two ways - - using iteration and using recursion

- for your recursive definition's base case, for what kind of array do you know the value of `Minimum(a, size)` right away?

```
// Iterative definition

int  Minimum  (  /* in */  const  int  a[ ] ,   /* in */   int   size )

    //  Pre:     Assigned( a [ 0 . . (size - 1) ] )  &&  size >= 1
    //  Post:   Function value == smallest of a [ 0 . . (size - 1) ]

{
        int min = a[0];
        for (int i = 1; i < size; i++)
                if (min > a[i]) min = a[i];
        return min;
}
```

```
// Recursive definition
int  Minimum  (  /* in */  const  int  a[ ] ,   /* in */   int   size )

 //  Pre:     Assigned( a [ 0 . . (size - 1) ] )  &&  size >= 1
 //  Post:   Function value == smallest of a [ 0 . . (size - 1) ]

{
          if  ( size == 1 )                         //  base case

                    return  a [ 0 ] ;

          else  {                                 //  general case

               int  y = Minimum ( a, size - 1 );

                 if  ( y < a [size - 1] )
                    return  y ;
               else
                    return a [ size -1 ] ;
      }

}
```

Demo Program:

RecursiveArray.cpp

Go Dev C++!!!

RecursiveArray.cpp

```
#include <iostream>
using namespace std;

int  Sum(const int a[], int low, int high){
     if (low == high) // Base case
          return  a [low];
     else                    // General case
          return  a [low] + Sum(a, low + 1, high);
}

int LinearSearch(const int a[], int low, int high, int key){
     if  (a [low] == key) // Base case
          return low;
     else if (low == high) // Second base case
          return -1;
     else                         // General case
          return LinearSearch(a, low + 1, high, key);
}

```

```
int BinarySearch (const int a[], int low, int high, int key) {
    int  mid;
    if  (low > high)
        return -1;
    else{
        mid = (low + high) / 2;
        if (a [mid] == key)
            return mid;
        else if (key < a[mid]) // Look in lower half
            return BinarySearch(a, low, mid-1, key);
        else  // Look in upper half
            return BinarySearch(a, mid+1, high, key);
    }
}

int Minimum(const int a[], int size){
    if ( size == 1 )                              //  base case
        return  a [ 0 ] ;
    else{                                        //  general case
        int  y = Minimum ( a, size - 1 );
        if (y < a [size - 1])
            return  y ;
        else
            return a[size -1];
    }
}
```

```cpp
int main(int argc, char** argv){
    int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int b[10] = {3, 6, 7, 8, 4, 5, 9, 0, 1, 2};
    cout << Sum(a, 0, 9)<< endl;
    cout << LinearSearch(b, 0, 9, 8)<< endl;
    cout << BinarySearch(a, 0, 9, 30)<< endl;
    cout << Minimum(b, 10);
    return 0;
}
```

Select C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 25\RecursiveArray\RecursiveArray.exe

```
550
3
2
0
--------------------------------
Process exited after 0.07426 seconds with return value 0
Press any key to continue . . .
```