C++ Programming:
Program Design Including Data Structures

Recursion: Part I [Chap 16]

Objectives

In this chapter you will:

- Learn about recursive definitions
- Explore the base case and the general case of a recursive definition
- Discover what is a recursive algorithm
- Learn about recursive functions
- Explore how to use recursive functions to implement recursive algorithms

Recursive Definitions

- Recursion: solving a problem by reducing it to smaller versions of itself
- $0! = 1$ (1)

  $n! = n \times (n-1)!$ if $n > 0$ (2)
- The definition of factorial in equations (1) and (2) is called a recursive definition
- Equation (1) is called the base case
- Equation (2) is called the general case

Recursive Definitions (continued)

- Recursive definition: defining a problem in terms of a smaller version of itself
  - Every recursive definition must have one (or more) base cases
  - The general case must eventually reduce to a base case
  - The base case stops the recursion

Recursive Algorithms

- Recursive algorithm: finds a solution by reducing problem to smaller versions of itself
  - Must have one (or more) base cases
- General solution must eventually reduce to a base case
- Recursive function: a function that calls itself
- Recursive algorithms are implemented using recursive functions

Recursive Functions

- Think of a recursive function as having many copies of itself
- Every call to a recursive function has
  - Its own code
  - Its own set of parameters and local variables
- After completing a particular recursive call
  - Control goes back to the calling environment, which is the previous call

Recursive Functions (continued)

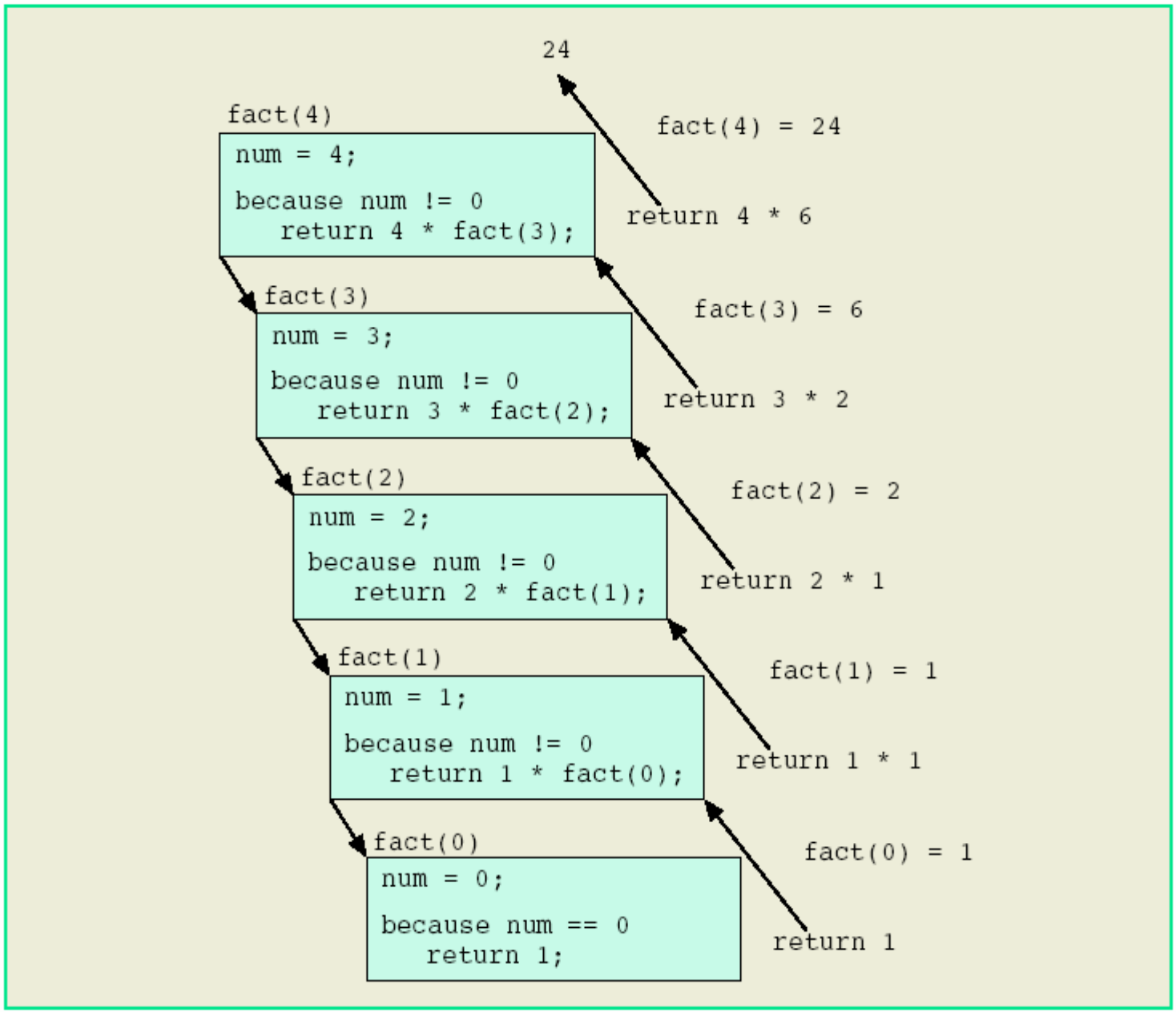
- The current (recursive) call must execute completely before control goes back to the previous call
- Execution in the previous call begins from the point immediately following the recursive call

Example: the factorial function

```
int fact(int num)  // num is nonegative integer
{
    if(num==0)
        return 1;
   else {
        int x = fact(num-1);
        return x*num;
   }
}
```

Alternative implementation

```
int fact(int num)  // num is nonegative integer
{
    if(num==0)
        return 1;
   else
        return num*fact(num-1);
}
```



Execution of the expression `fact (4)`

Direct and Indirect Recursion

- Directly recursive: a function that it calls itself
- Indirectly recursive: a function that calls another function and eventually results in the original function call

Infinite Recursion

- Infinite recursion: every recursive call results in another recursive call
  - In theory, infinite recursion executes forever
- Because computer memory is finite:
  - Function executes until the system runs out of memory
  - Results in an abnormal program termination

Tracking the recursion process example

- Consider the function

```
void f(int n) { cout<<n<<" ";
                if(n>=1)  f(n-1);}
```

- Consider calling it from main via:

```
f(10);
```

- What do we get? Track recursion
- Answer: 10 9 8 7 6 5 4 3 2 1 0

Tracking the recursion process example (Continued)

- Consider swapping the if and the cout statements, i.e., consider the function g:

```
void g(int n) {if(n>=1)  g(n-1);
                         cout<<n<<" "; }
```

- Consider calling it from main via:

```
g(10);
```

- What do we get? Track recursion
- Answer: 0 1 2 3 4 5 6 7 8 9 10

Number of recursive branches

- Recursive factorial (and the above examples f and g): one recursive call
- Next: an example of of a function with 2 recursive calls

Fibonacci numbers

- Defined recursively:

  $a_1 = 1$

  $a_2 = 1$

  $a_n = a_{n-1} + a_{n-2}$, if $n \geq 3$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Problem: given **n**, compute the **n'th** Fibonacci number **a_n**

Iterative function

```
int FibNum(int n)
{
    if(n==1 || n==2) return 1;
    int previous1=1,  previous2=1, current;
    for(int i=3;i<=n;i++) {
        current = previous1+previous2;
        previous2 = previous1;
        previous1 = current;
     }
    return current;
}
```

Arrays version (new/delete review)

```
int FibNumArraysVersion(int n)
{
    if(n==1 || n==2) return 1;
    int* A = new int[n];
    A[0] = 1; A[1] = 1;
    for(int i=2;i<n;i++)
          A[i] =A[i-1]+A[i-2];
    int answer = A[n-1];
    delete [] A;
    return answer;
}
```

Recursive function

```
int rFibNum(int n)
{
    if(n==1 || n==2)    return 1;  // base case
    else {
      int prev1 = rFibNum(n-1); // 1st recursive call
      int prev2 = rFibNum(n-2); // 2nd recursive call
      return prev1 + prev2;
    }
}
```

Compact form of the recursive function

```
int rFibNum(int n)
{
    if(n==1 || n==2)
        return 1;
    else
        return rFibNum(n-1)+rFibNum(n-2);
}
```

Flow of execution

- Track the exectution of

  cout << rFibNum(5);

- Draw recursion tree

rFibNum versus FibNum

- The recursive function rFibNum is not efficient at all compared the iterative function FibNum!!!

- Why?

rFibNum versus FibNum (Continued)

- Many repeated sub-problems
- The recursive function rFibNum solves the same sub-problem more than once (typically a huge number of times)
- That was just an illustrative example of a recursive function that calls itself more than once (twice)
- More interesting examples next lecture: Merge Sort & Tower of Hanoi

Designing a Recursive Function

- To design a recursive function:
  - Understand problem requirements
  - Determine limiting conditions
  - Identify base cases and provide a direct solution to each base case
  - Identify general cases and provide a solution to each general case in terms of smaller versions of itself

Problem Solving Using Recursion

- Consider the Array Largest problem:

  Given an array **list[0…n-1]**,find the largest element in list.
- Recall the iterative algorithm
- Now a recursive algorithm

Array Largest recursive algorithm: idea

- To find the largest element in **list[0 … n-1]**
  - Find largest element in **list[1 … n-1]** recursively and call it **max**
  - Compare the elements **list[0]** and max

    if (**list[0]** >= **max**)

    the largest element in **list[0 … n-1]** is **list[0]**

    otherwise

    the largest element in **list[0 … n-1]** is **max**
- Base case: We stop the recursion when list **size = 1**

Array Largest recursive algorithm (Continued)

- The second recursive call is on **list[1…n-1]**
- The third on **list[2 .. n-1]**
- Thus it is convenient (and essential) to write a recursive function to solve the following version of te Array Largest problem:

  Given an array **list** (passed by reference) and two indicies **a** (the lower index) and **b** (the upper index), find the largest elemet in **list[a…b]**.
- When calling the function, we set **a=0** and **b=n-1**
- This techinque is useful for almost all recursive functions with arrays parameters

Array Largest recursive algorithm (Continued)

- To find the largest element in list[a … b]
  - Find largest element in list[a + 1 … b] and call it max
  - Compare the elements list[a] and max

    if (list[a] >= max)

        the largest element in list[a … b] is list[a]

    otherwise

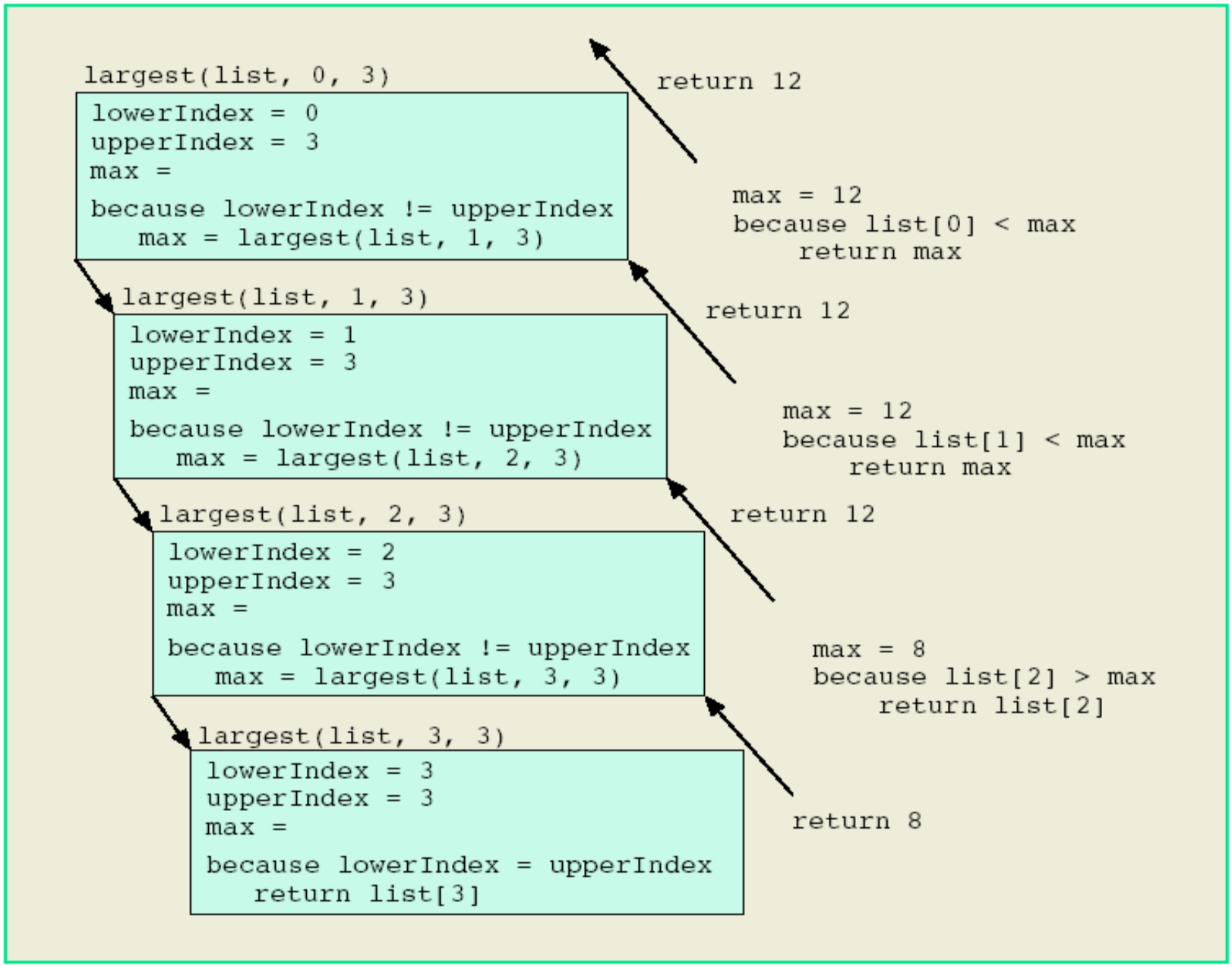
        the largest element in list[a … b] is max
- Base case: List size = 1

Array Largest function (Continued)

```
int largest(const int list[], int a, int b)
// to find the largest element in list[ a … b]
{
  if( a == b) return  list[a];
  else {
     int max = largest(list, a+1,b);
     if(list[a]>max) return list[a];
     else return max;
   }
}
```

We can call the function from main, for instance, as follows:

```
list[] = {5,10,12,8};
cout<< largest(list,0,3);
```



Execution of the expression `largest(list, 0, 3)`

Important Remark

- Since the array it is passed by **reference, it is NOT copied** to recursive subcalls. All the copies of the function work on the same copy of the array.

  (Only *a,b,and the pointer A* are copied).

Why do it recursively?

- You are probably thinking: we can find the largest element in array using a simple for loop, why this recursive algorithms !!!
- You are right … this just an illustrative example
- More interesting examples next lecture

Summary

- The process of solving a problem by reducing it to smaller versions of itself is called recursion
- A recursive definition defines a problem in terms of smaller versions of itself
- Every recursive definition has one or more base cases
- A recursive algorithm solves a problem by reducing it to smaller versions of itself
- Every recursive algorithm has one or more base cases

Summary

- The solution to the problem in a base case is obtained directly
- A function is called recursive if it calls itself
- Recursive algorithms are implemented using recursive functions
- Every recursive function must have one or more base cases
- The general solution breaks the problem into smaller versions of itself

Summary

- The general case must eventually be reduced to a base case
- The base case stops the recursion
- Directly recursive: a function calls itself
- Indirectly recursive: A function calls another function and eventually calls the original