Standard Version of
Starting Out with C++, 4th Edition

Recursion
Topics

• Introduction to Recursion
• The Recursive Factorial Function
• The Recursive gcd Function
• Solving Recursively Defined Problems
Topics

• Recursive Linked List Operations
• A Recursive Binary Search Function
• The QuickSort Algorithm
• Exhaustive Algorithms
• Recursion Versus Iteration
Introduction to Recursion

- A **recursive function** contains a call to itself:

```cpp
void countDown(int num)
{
    if (num == 0)
        cout << "Blastoff!";
    else
    {
        cout << num << "...\n";
        countDown(num-1); // recursive
    }
} // call
```
What Happens When Called?

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output 2..., then it calls `countDown(1)`
2. `countDown(1)` generates the output 1..., then it calls `countDown(0)`
3. `countDown(0)` generates the output Blastoff!, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the calling function

Chapter 19 slide 5
What Happens When Called?

first call to countDown
num is 2

countDown(1);

second call to countDown
num is 1

countDown(0);

third call to countDown
num is 0

// no // recursive // call

output:

2...

1...

Blastoff!
Recursive Functions - Purpose

• Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
• The simpler-to-solve problem is known as the base case.
• Recursive calls stop when the base case is reached.
Stopping the Recursion

• A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call

• In the sample program, the test is:

  ```java
  if (num == 0)
  ```
Stopping the Recursion

```cpp
void countDown(int num)
{
    if (num == 0) // test
        cout << "Blastoff!";
    else
    {
        cout << num << "...
";
        countDown(num-1); // recursive
    } // call
}
```
Stopping the Recursion

• A different value is passed to the function each time it is called
• Eventually, the parameter reaches the value in the test, and the recursion stops
Stopping the Recursion

```cpp
void countDown(int num)
{
    if (num == 0)
        cout << "Blastoff!";
    else
    {
        cout << num << "...
";
        countDown(num-1); // note that the value
                        // passed to recursive
                        // calls decreases by
                        // one for each call
    }
}
```
What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created.
- As each copy finishes executing, it returns to the copy of the function that called it.
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function.
What Happens When Called?

first call to countDown
num is 2

countDown(1);

return

second call to countDown
num is 1

countDown(0);

return

third call to countDown
num is 0

// no
// recursive
call

// no recursive call

output:

2...

1...

Blastoff!
Types of Recursion

• Direct
  – a function calls itself

• Indirect
  – function A calls function B, and function B calls function A
  – function A calls function B, which calls …, which calls function A
The Recursive Factorial Function

- The factorial function:
  \[ n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \text{ if } n > 0 \]
  \[ n! = 1 \text{ if } n = 0 \]
- Can compute factorial of \( n \) if the factorial of \( (n-1) \) is known:
  \[ n! = n \times (n-1)! \]
- \( n = 0 \) is the base case
The Recursive Factorial Function

```c
int factorial (int num)
{
    if (num > 0)
        return num * factorial(num - 1);
    else
        return 1;
}
```
The Recursive gcd Function

• Greatest common divisor (gcd) is the largest factor that two integers have in common

• Computed using Euclid's algorithm:
  \[ \text{gcd}(x, y) = y \text{ if } y \text{ divides } x \text{ evenly} \]
  \[ \text{gcd}(x, y) = \text{gcd}(y, x \% y) \text{ otherwise} \]

• \( \text{gcd}(x, y) = y \) is the base case
The Recursive gcd Function

```c
int gcd(int x, int y) {
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```
Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:
  \[ 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \]
- After the starting 0, 1, each number is the sum of the two preceding numbers
- Recursive solution:
  \[ \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2); \]
- Base cases: \( n \leq 0 \), \( n == 1 \)
Solving Recursively Defined Problems

```c
int fib(int n)
{
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```
Recursive Linked List
Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of (number of nodes in) a list
  - Traverse the list in reverse order
Size of a Linked List

- Uses a pointer to visit each node
- Algorithm:
  - pointer starts at head of list
  - If pointer is NULL, return 0 (base case)
  - else, return 1 + number of nodes in the list pointed to by current node
Contents of a List in Reverse Order

• Algorithm:
  – pointer starts at head of list
  – If the pointer is NULL, return (base case)
  – If the pointer is not NULL, advance to next node
    – Upon returning from recursive call, display contents of current node
A Recursive Binary Search Function

• Binary search algorithm can easily be written to use recursion
• Base cases: desired value is found, or no more array elements to search
• Algorithm (array in ascending order):
  – If middle element of array segment is desired value, then done
  – Else, if the middle element is too large, repeat binary search in first half of array segment
  – Else, if the middle element is too small, repeat binary search on the second half of array segment
The QuickSort Algorithm

• Recursive algorithm that can sort an array or a linear linked list
• Determines an element/node to use as **pivot** value:

```
sublist 1  pivot  sublist 2
```
The QuickSort Algorithm

• Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are > pivot
• Algorithm then sorts sublist1 and sublist2
• Base case: sublist has size 1
Exhaustive and Enumeration Algorithms

- **Exhaustive algorithm**: search a set of combinations to find an optimal one.
  
  Example: change for a certain amount of money that uses the fewest coins.

- Uses the generation of all possible combinations when determining the optimal one.
Recursion vs. Iteration

• Benefits (+), disadvantages(-) for recursion:
  + Models certain algorithms most accurately
  + Results in shorter, simpler functions
  – May not execute very efficiently
• Benefits (+), disadvantages(-) for iteration:
  + Executes more efficiently than recursion
  – Often is harder to code or understand