Chapter 7

Programming with Recursion
What Is Recursion?

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

- You must be careful when using recursion.
- Recursive solutions can be less efficient than iterative solutions.
- Still, many problems lend themselves to simple, elegant, recursive solutions.
Some Definitions

- **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 (a) smaller instances of itself and (b) a base case
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!

We must avoid making an infinite sequence of function calls (infinite recursion).
Finding a Recursive Solution

- Each successive recursive call should bring you closer to a situation in which the answer is known.

- A case for which the answer is known (and can be expressed without recursion) is called a base case.

- Each recursive algorithm must have at least one base case, as well as the general (recursive) case
General format for many recursive functions

if (some condition for which answer is known)  // base case

solution statement

else  // general case

recursive function call

SOME EXAMPLES . . .
DISCUSSION

The function call Factorial(4) should have value 24, because that is \(4 \times 3 \times 2 \times 1\).

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

```javascript
if ( number == 0 )
    return 1;
```
Now for the **general case** . . .

The value of Factorial(n) can be written as

\[ n \times \text{the product of the numbers from } (n - 1) \text{ to } 1, \]

that is,

\[ n \times (n - 1) \times \ldots \times 1 \]

or, \[ n \times \text{Factorial}(n - 1) \]

And notice that the recursive call Factorial(n - 1) gets us “closer” to the base case of Factorial(0).
Recursive Solution

```c
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 ) ;
}
```
Three-Question Method of verifying recursive functions

- **Base-Case Question:** Is there a nonrecursive way out of the function?

- **Smaller-Caller Question:** Does each recursive function call involve a smaller case of the original problem leading to the base case?

- **General-Case Question:** Assuming each recursive call works correctly, does the whole function work correctly?
Another example where recursion comes naturally

- From mathematics, we know that
  \[2^0 = 1 \quad \text{and} \quad 2^5 = 2 \times 2^4\]

- In general,
  \[x^0 = 1 \quad \text{and} \quad x^n = x \times x^{n-1}\]
  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: Function 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 looping instead of a recursive call in the function body.
struct ListType
{
    int length;   // number of elements in the list
    int info[MAX_ITEMS];
};

ListType list;
Recursive function to determine if value is in list

**PROTOTYPE**

```c
bool ValueInList( ListType list, int value, int startIndex );
```

<table>
<thead>
<tr>
<th>list[0]</th>
<th>[1]</th>
<th>[startIndex]</th>
<th>[length -1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>36</td>
<td>...</td>
<td>95 75 29 47</td>
</tr>
</tbody>
</table>

- Already searched
- Needs to be searched
- Index of current element to examine
bool ValueInList ( ListType list, int value, int startIndex )

// Searches list for value between positions startIndex
// and list.length-1
// Pre: list.info[startIndex] . . list.info[list.length - 1]
// contain values to be searched
// Post: Function value =
// ( value exists in list.info[startIndex] . .
// list.info[list.length - 1] )
{
    if ( list.info[startIndex] == value )       // one base case
        return true;
    else if (startIndex == list.length -1)  // another base case
        return false;
    else 
        // general case
        return ValueInList( list, value, startIndex + 1 );
}
Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement.

However, for certain problems the recursive solution is the most natural solution. This often occurs when pointer variables are used.
struct ListType
{
    struct NodeType
    {
        int info;
        NodeType* next;
    }
    class SortedType
    {
        public:
            . . . // member function prototypes
        private:
            NodeType* listData;
    }
};
RevPrint(listData);

listData

A → B → C → D → E

FIRST, print out this section of list, backwards

THEN, print this element
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. That is, the number of list elements to be processed decreases by 1 with each recursive call. By printing one element in the general case, and also processing the smaller remaining list, we will eventually reach the situation where 0 list elements are left to be processed.

In the general case, we will print the elements of the smaller remaining list in reverse order, and then print the current pointed to element.
void RevPrint ( NodeType* listPtr )

// Pre: listPtr points to an element of a list.
// Post: all elements of list pointed to by listPtr
// have been printed out in reverse order.
{
    if ( listPtr != NULL ) // general case
    {
        RevPrint ( listPtr->next ); // process the rest
        std::cout << listPtr->info << std::endl ; // print this element
    }
    // Base case: if the list is empty, do nothing
}
Function BinarySearch()

- BinarySearch takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments. It returns false if item is not found in the elements info[fromLoc...toLoc]. Otherwise, it returns true.

- BinarySearch can be written using iteration, or using recursion.
```c
found = BinarySearch(info, 25, 0, 14);
```

**indexes**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
</table>

**info**

| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 |

**NOTE:** denotes element examined
template<class ItemType>
bool BinarySearch ( ItemType info[ ] , ItemType item ,
     int fromLoc , int toLoc )
    // Pre: info [ fromLoc .. toLoc ] sorted in ascending order
    // Post: Function value = ( item in info [ fromLoc .. toLoc] )
{
    int mid ;
    if ( fromLoc > toLoc )  // base case -- not found
        return false ;
    else {
        mid = ( fromLoc + toLoc ) / 2 ;
        if ( info [ mid ] == item )  // base case-- found at mi
            return true ;
        else if ( item < info [ mid ] )  // search lower half
            return BinarySearch ( info, item, fromLoc, mid-1 ) ;
        else  // search upper half
            return BinarySearch( info, item, mid + 1, toLoc ) ;
    }
}
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. On this stack is placed an **activation record (stack frame)** for the function call.
The activation record stores the return address for this function call, and also the parameters, local variables, and 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 ( int a, int b )

// Pre: a and b have been assigned values
// Post: Function value = ??

{
    int result;
    if ( b == 0 ) // base case
        result = 0;
    else if ( b > 0 ) // first general case
        result = a + Func ( a, b - 1 ); // instruction 50
    else // second general case
        result = Func ( -a, -b ); // instruction 70

    return result;
}

Run-Time Stack Activation Records

\[ x = \text{Func}(5, 2); \]  // original call is instruction 100

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>Return Address</td>
<td>100</td>
</tr>
</tbody>
</table>

original call at instruction 100 pushes on this record for Func(5,2)
Run-Time Stack Activation Records

\[ x = \text{Func}(5, 2); \]  // original call at instruction 100

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>Return Address</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>5+\text{Func}(5,1) = ?</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>Return Address</td>
<td>100</td>
</tr>
</tbody>
</table>

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

record for \text{Func}(5,2)
x = Func(5, 2); // original call at instruction 100

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>Return Address</td>
<td>50</td>
</tr>
</tbody>
</table>

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

```
FCTVAL       ?
result       ?
b            1
a            5
Return Address 50
```

`record for Func(5,1)`

```
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(5,2)`
Run-Time Stack Activation Records

\[ x = \text{Func}(5, 2); \quad \text{// original call at instruction 100} \]

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>result</th>
<th>b</th>
<th>a</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>?</td>
<td>5+\text{Func}(5,0) = ?</td>
<td>1</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>?</td>
<td>5+\text{Func}(5,1) = ?</td>
<td>2</td>
<td>5</td>
<td>100</td>
</tr>
</tbody>
</table>

- record for \text{Func}(5,0) is popped first with its FCTVAL
- record for \text{Func}(5,1)
- record for \text{Func}(5,2)
The expression `x = Func(5, 2);` with comments indicating the original call at instruction 100.

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>Result</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>5+Func(5,0) = 5 + 0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Record for `Func(5,1)` is popped next with its FCTVAL.

<table>
<thead>
<tr>
<th>FCTVAL</th>
<th>Result</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>?</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>5+Func(5,1) = ?</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Record for `Func(5,2)`
x = Func(5, 2);       // original call at line 100

record for Func(5,2) is popped last with its FCTVAL
x = Func(-5, -3);

x = Func(5, -3);

What operation does Func(a, b) simulate?
Tail Recursion

- The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.

- Tail recursion can be replaced by iteration to remove recursion from the solution as in the next example.
// USES TAIL RECURSION

bool ValueInList ( ListType list , int value , int startIndex )

// Searches list for value between positions startIndex
// and list.length-1
// Pre: list.info[ startIndex ] . . list.info[ list.length - 1 ]
// contain values to be searched
// Post: Function value =
// ( value exists in list.info[ startIndex ] . .
// list.info[ list.length - 1 ] )
{
    if ( list.info[startIndex] == value )       // one base case
        return true ;
    else if (startIndex == list.length -1 )   // another base case
        return false ;
    else                                       // general case
        return ValueInList( list, value, startIndex + 1 ) ;
}
// ITERATIVE SOLUTION

bool ValueInList ( ListType list, int value, int startIndex )

// Searches list for value between positions startIndex
// and list.length-1
// Pre: list.info[startIndex] .. list.info[list.length - 1]
// contain values to be searched
// Post: Function value =
// ( value exists in list.info[startIndex] .. list.info[list.length - 1] )
{
    bool found = false;
    while ( !found && startIndex < list.length )
    {
        if ( value == list.info[startIndex] )
            found = true;
        else    startIndex++;
    }
    return found;
}
Use a recursive solution when:

- The depth of recursive calls is relatively "shallow" compared to the size of the problem.
- The recursive version does about the same amount of work as the nonrecursive version.
- The recursive version is shorter and simpler than the nonrecursive solution.
Using quick sort algorithm

A . . Z

A . . L

A . . F

G . . L

M . . R

S . . Z
Before call to function Split

splitVal = 9

GOAL: place splitVal in its proper position with all values less than or equal to splitVal on its left and all larger values on its right

| 9 | 20 | 6 | 10 | 14 | 3 | 60 | 11 |

values[first] [last]
After call to function Split

splitVal = 9

smaller values

larger values

6  3  9  10  14  20  60  11

values[first]  [splitPoint]  [last]
// Recursive quick sort algorithm

template <class ItemType>
void QuickSort ( ItemType values[], int first, int last )
{
    // Pre:   first <= last
    // Post: Sorts array values[ first . . last ] into
    //       ascending order
    
    if ( first < last )  // general case
    {
        int splitPoint;
        Split ( values, first, last, splitPoint );

        // values [ first ] . . values[splitPoint - 1] <= splitVal
        // values [ splitPoint ] = splitVal
        // values [ splitPoint + 1 ] . . values[ last ] > splitVal

        QuickSort ( values, first, splitPoint - 1 );
        QuickSort ( values, splitPoint + 1, last );
    }
}

}