Chapter 19

Recursion

Introduction to Recursion
Introduction to Recursion

A recursive function is one that calls itself:

```c
void message( void )
{ cout << "This is a recursive function.\n";
   message();
} // message
```

- The function above displays the string "This is a recursive function.\n", and then calls itself

**Can you see a problem with the function?**
Recursion

- The function is like an infinite loop because there is no code to stop it from repeating
- Like a loop, a recursive function must have some way to control the number of times it repeats

Shown below is a modification of the message function.
- It passes an integer argument, that holds the number of times the function calls itself

```cpp
void message( int times )
{if ( times > 0 )
 { cout << "This is a recursive function.\n";
    message( times - 1 );
 } // if
} // message
```
Recursion

- The function contains an if statement that controls the repetition
- As long as the times argument is greater than zero, it displays the message and calls itself again
- Each time it calls itself, it passes times - 1 as the argument

For example, let's say a program calls the function with the following statement:
Message(5);
- The argument, 5, causes the function to call itself 5 times
  - The first time the function is called, the if statement displays the message and then calls itself with 4 as the argument
  - The next slide illustrates this
Recursion

Each time the function is called, a new instance of the times parameter is created.

1st call of the function
Value of times: 5

In the first call to the function, times is set to 5

2nd call of the function
Value of times: 4

When the function calls itself, a new instance of times is created with the value 4

This cycle repeats itself until 0 is passed to the function.

Depth of recursion: 6
Recursion

- The role of recursive functions in programming is to break complex problems down to a solvable problem.
- The solvable problem is known as the base case.
  - There must always be a base case in a recursive algorithm.
- A recursive function is designed to terminate when it reaches its base case.

The numChars Function

```c
int numChars ( char search, char str[], int subscript )
{
    if ( str[subscript] == '\0' )
        return 0; // base case
    else
    {
        if ( str[subscript] == search )
            return 1 + numChars( search, str, subscript+1 );
        else
            return numChars( search, str, subscript+1 );
    } // if
} // numChars
```

The function's parameters are:

- `str`: An array containing the C-string to be searched.
- `subscript`: The starting subscript for the search.
- `search`: The character to be searched for and counted.
The numChars function

The first if statement determines if the end of the string has been reached:

\[
\text{if ( str[subscript] == '\0' )}
\]

If so, the function returns 0, indicating there are no more characters to count. Otherwise, the following if/else statement is executed:

\[
\text{if (str[subscript] == search)}
\]

\[
\text{return 1 + numChars(search, str, subscript+1);}
\]

\[
\text{else}
\]

\[
\text{return numChars(search, str, subscript+1);}
\]

If str[subscript] contains the search character, the function performs a recursive call. The return statement returns 1 + the number of times the search character appears in the string, starting at subscript + 1. If str[subscript] does not contain the search character, a recursive call is made to search the remainder of the string.

Types of Recursion

- **Direct**
  - A recursive functions that calls itself

- **Indirect**
  - Function A calls function B, which in turn calls function A
  - Function A calls function B, which call …, which calls function A
The Recursive Factorial Function

- The factorial of a number is defined as:
  \[ n! = 1 \times 2 \times 3 \times \ldots \times n \quad \text{if } n > 0 \]
  \[ n! = 1 \quad \text{if } n = 0 \]
- Another definition:
  \[ n! = n \times (n-1)! \quad \text{if } n > 0 \]
  \[ n! = 1 \quad \text{if } n = 0 \quad (\text{base case}) \]
- *Write this function!*

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

- The next example of recursion is the calculation of the greatest common divisor, or gcd, of two numbers.
- Using Euclid's Algorithm, the gcd of two positive integers, x and y, is:

  \[
  \text{gcd}(x, y) = \begin{cases} 
  y & \text{if } x \text{ divides } y \text{ evenly} - \text{base case} \\
  \text{gcd}(y, \text{remainder of } y/x) & \text{otherwise}
  \end{cases}
  \]

  The definition above states that the gcd of x and y is y if x/y has no remainder.
  Otherwise, the answer is the gcd of y and the remainder of x/y.
  \textit{Write this function!}

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

- Some mathematical problems are designed to be solved recursively.
- One example is the calculation of Fibonacci numbers, which is the following sequence:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci series can be defined as:

\[
F_0 = 0, \\
F_1 = 1, \\
F_N = F_{N-1} + F_{N-2} \text{ for } N \geq 2.
\]

Write this function!
Solving Recursively Defined Problems

- A recursive C++ function to calculate the n\textsuperscript{th} number in the Fibonacci series is shown below:

```cpp
int fib ( int n )
{
  if ( n <= 0 )
    return 0; // base case
  else if ( n == 1 )
    return 1; // base case
  else
    return fib( n – 1 ) + fib( n – 2 );
}
```

Recursive Linked List Operations

- Recursion may be used in some operations on linked lists
- We look at functions that:
  - Count the number of nodes in a list, and
  - Display the value of the list nodes in reverse order.
Counting the Nodes in the List

- We use a pointer to visit each node

Algorithm:
- The pointer starts at the head of the list
- If the pointer is NULL, return 0 (base case)
- Else return 1 + number of nodes in the rest of the list pointed to by the current node
- *Write this function!*

```cpp
int NumberList::countNodes ( Node *nodePtr )
{ if ( nodePtr )
    return 1 + countNodes( nodePtr->next );
else
    return 0; // base case
} // NumberList::countNodes
```
Displaying the Value of the List Nodes in Reverse Order

- Write this function!

```cpp
void NumberList::showReverse(Node *nodePtr)
{
    if (nodePtr)
    {
        showReverse(nodePtr->next);
        cout << nodePtr->value << " ";
    } // if
} // NumberList::showReverse
```

Where is the base case?
A Recursive Binary Search Function

- The binary search algorithm, which you learned previously, can be implemented recursively
- The procedure can be expressed as:
  - If array[middle] equals the search value, then the value is found
  - Else, if array[middle] is less than the search value, perform a binary search on the upper half of the array
  - Else, if array[middle] is greater than the search value, perform a binary search on the lower half of the array
  - Write this function!

```c
int binarySearch ( int array[], int first, int last, int value )
{
  int middle;
  if ( first > last )
    return -1;
  middle = ( first + last ) / 2;
  if ( array[middle] == value )
    return middle;
  else if ( array[middle] < value )
    return binarySearch( array, middle+1,last,value );
  else
    return binarySearch( array, first,middle-1,value );
} // binarySearch
```
The QuickSort Algorithm

- Can be used to sort lists stored in arrays or linear linked lists
- It sorts a list by dividing it into two sublists.
  - Between the sublists is a selected value known as the \textit{pivot}
  - This is illustrated below:

\begin{center}
\begin{tikzpicture}
  \draw[thick] (0,0) -- (4,0) -- (4,1) -- (0,1) -- (0,0);
  \draw[thick,->] (1,0.5) -- (1,0);
  \draw[thick,->] (3,0.5) -- (3,0);
  \node at (2,0.25) {Pivot};
  \node at (0.5,0.5) {Sublist 1};
  \node at (3.5,0.5) {Sublist 2};
\end{tikzpicture}
\end{center}

- Once a pivot value has been selected, the algorithm exchanges the other values in the list until all the elements in sublist 1 are less than the pivot, and all the elements in sublist 2 are greater than the pivot
- Once this is done, the algorithm repeats the procedure on sublist 1, and then on sublist 2.
- The recursion stops when there is only one element in a sublist. At that point the original list is completely sorted
- \textbf{Write this function!}
void quickSort ( int set[], int start, int end )
{
    int pivotPoint;
    if ( start < end )
    {
        pivotPoint = partition( set, start, end );
        quickSort( set, start, pivotPoint – 1 );
        quickSort( set, pivotPoint + 1, end );
    } // if
} // quickSort

int partition ( int set[], int start, int end )
{
    int pivotValue, pivotIndex, mid;
    mid = ( start + end ) / 2;
    swap( set[start], set[mid] );
    pivotIndex = start;
    pivotValue = set[start];
    for ( int scan = start + 1; scan <= end; scan++ )
    {
        if ( set[scan] < pivotValue )
        {
            pivotIndex++;
            swap( set[pivotIndex], set[scan] );
        } // if
    } // for
    exchange( set[start], set[pivotIndex] );
    return pivotIndex;
} // partition
Exhaustive Algorithms

- An exhaustive algorithm is one that finds a best combination of items by looking at all the possible combinations.

For example, consider all the different ways you can make change for $1.00 using our system of coins:
- 1 dollar piece, or
- 2 fifty-cent pieces, or
- 4 quarters, or
- 1 fifty-cent piece and 2 quarters, or
- 3 quarters, 2 dimes, and 1 nickel, or
- … there are many more possibilities
Exhaustive Algorithms

- Although there are many ways to make change for $1.00, some ways are better than others.
  - For example, you would probably rather give a single dollar piece instead of 100 pennies.
- An algorithm that looks at all the possible combinations of items in order to find the best combination of items is called an exhaustive algorithm.

Making Change

```c
void makeChange ( int coinsLeft, int amount, int coinsUsed[], int numCoinsUsed )
{
    int coinPos, // To calculate array position of coin being used
    count; // Loop counter
    if ( coinsLeft == 0 ) // If no more coins are left
        return;
    else if ( amount < 0 ) // If amount to make change for is negative
        return;
    else if ( amount == 0 ) // If solution is found
    { if ( numCoinsUsed < numBestCoins )
        { for ( count = 0; count < numCoinsUsed; count++ )
            bestCoins[count] = coinsUsed[count];
        numBestCoins = numCoinsUsed;
        } // if
        numSolutions++;
        return;
    } // else if
```
Making Change

```c
coinPos = numCoins - coinsLeft;
coinsUsed[numCoinsUsed] = coinValues[coinPos];
makeChange( coinsLeft, amount - coinValues[coinPos], coinsUsed, 
    numCoinsUsed + 1 );
numCoinsUsed--;
makeChange( coinsLeft - 1, amount, coinsUsed, numCoinsUsed );
} // makeChange
```

Recursion vs. Iteration

- **Benefits and disadvantages for recursion:**
  - Models certain algorithms most accurately
  - Results in shorter, simpler functions
  - May not execute very efficiently

- **Benefits and disadvantages for iteration:**
  - Executes more efficiently than recursion
  - Often harder to code and/or understand