Recursion

- For some problems it is useful to have a function call itself.
- A *recursive function* is a function that calls itself, either directly, or indirectly through another function.
- Often the decision to use recursion is suggestion by the nature of the problem itself. To be appropriate for a recursive solution a problem must:

1. Be possible to decompose the original problem into simpler instances of the same problem
2. Once each simpler subproblem has been solved it must be possible to combine these solutions to produce a solution to the original problem.
3. As the large problem is broken down into less complex ones, those subproblems must eventually become so simple they can be solved without further subdivision.

- For example the factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), is the product

  n * (n - 1) * (n - 2) * … * 1

- with 1! equal to 1, and 0! defined to be 1.
- to clarify 5! is the product 5 * 4 * 3 * 2 * 1 which = 120.
- The factorial of an integer, greater than or equal to 0, can be calculated iteratively (non-recursively) by using `for` as follows:

```
//
// iterative solution for factorial
uint factorialIter(uint number)
{
      int factorial = 1;
      for (int i = number; i >= 1; --i)
      {
            factorial *= i;
      }

      return factorial;
}

int main()
{
      uint number;
      cout << "Compute factorial for: ";
      cin >> number;

      cout << number << "! = " << factorialIter(number)
            << " [iterative solution]" << endl;
}
```

- A recursive solution of factorial is arrived at by observing the following relationship.
  n! = n * *(n - 1)!*
- For example, 5! is clearly equal to 5 * 4! as shown by
  5! = 5 * 4 * 3 * 2 * 1
  5! = 5 * (4 * 3 * 2 * 1)
  5! = 5 * (4!)
- Which can be expressed as a recursive function like so,

```
//
// recursive solution for factorial
uint factorialRecur(uint number)
{
    if (number == 0)
    {
        return 1;   // base case
    }
    else
    {               // recursive step
        return number * factorialRecur(number - 1);
    }
}
```

- This recursive function first determines if the terminating condition (`number == 0`) is true, if so, the function returns `1`. If `number` is > `0` the problem is expressed as the product of `number` and a recursive call to itself passing `number - 1`. Note that `factorialRecur(number - 1)` is a slightly simpler problem than the original calculation of `factorialRecur(number)`.
- A function is called *directly recursive* if it calls itself. The above `factorialRecur()` function is directly recursive.
- A function that calls another function and eventually results in the original call is said to be *indirectly recursive*.
- Another simple example of recursion is a function that computes x^*n* using the formula x^n = x * x^(n-1)

```
//
// recursive solution for power
int power(int x, int n)
{
    if (n == 0)
    {
        return 1;   // base case
    }
    else
    {               // recursive step
        return x * power(x, n - 1);
    }
}
```

- Note that the C++ standard document indicates that main should not be called within a program or recursively. Its sole purpose is to be the starting point for program execution.

Fibonacci Example

- In mathematics, the Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that, every number after the first two is the sum of the two preceding ones:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- The sequence is named after Italian mathematician Leonardo of Pisa, known as Fibonacci, who was considered "the most talented Western mathematician of the Middle Ages." He is best known for popularizing Hindu–Arabic numerals in the Western World.
- Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study.
- Fibonacci numbers are often seen in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern and the arrangement of a pine cone's bracts.
- In his book, "Liber Abaci" (1202) Fibonacci considers the growth of a rabbit population, assuming that: a newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?
- An iterative solution to calculate Fibonacci's problem,

```
int fibonacciIter(int n)
{
    int first = 0, second = 1, next = 0;
    for (int i = 0; i <= n; ++i)
    {
        if (i <= 1)
        {
            next = i;
        }
        else
        {
            next = first + second;
            first = second;
            second = next;
        }
    }
    return next;
}
```

- A recursive solution to calculate Fibonacci's problem,

```
//
// recursive solution for Fibonacci
int fibonacciRecur(int n)
{
    if (n == 0 || n == 1)
    {
        return n;   // base case
```

```
        }
        else
        {                // recursive step
            return fibonacciRecur(n - 1) + fibonacciRecur(n - 2);
        }
    }
```

- Notice here how the recursive approach more naturally expresses the problem, is simpler, and is easier to understand than the iterative approach.
- But word of caution about recursive programs like the one used here to generate Fibonacci numbers. Each level of recursion in Fibonacci has a doubling effect on the number of function calls. The number of recursive calls that are required to calculate the *n*th Fibonacci number is on the order of 2^n. This rapidly gets out of hand. Calculating only the 20th number requires the order of 2^20, or about a million calls. The 30th number would require 2^30, or about a billion calls. The iterative example runs much faster, particularly when calculating the 30th number!

Recursion vs. Iteration

- Both iteration and recursion are based on a control structure. Iteration uses a repetition structure, recursion uses a selection structure. Both involve repetition.
- Iteration and recursion both involve a termination test. Iteration terminates when the loop condition fails, recursion terminations when a base case is recognized. Both gradually approach termination.
- Iteration modifies a counter until the counter reaches a value causing the loop to exit. Recursion produces *simpler versions of the original problem until a base case is reached*.
- Both iteration and recursion can occur infinitely. With iteration it occurs if the counter never reaches the value required to exit the loop. With recursion it can occur if the recursion step does not reduce the problem in a manner that converges on the base case.
- Recursion has many negatives. It repeatedly invokes the mechanism and overhead of function calls. The can be expensive in both time and memory space, as each recursive call causes another copy of the function's variables to be created, which can consume considerable memory.
- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment typical of recursion is omitted.
- Any problem that can be solved recursively can also be solved iteratively. Recursion is not absolutely necessary. In fact, some programming languages do not allow it.
- So why choose recursion? A recursive solution is often chosen when the recursive approach more naturally mirrors the problem, and results in a program that is easier to understand and debug. Often an iterative solution is not apparent.
- In practice, recursion often arises naturally as a result of *divide-and-conquer* style algorithms, in which a larger problem is divided into smaller pieces that are then tackled by the same algorithm. A classic example of a *divide-and-conquer* strategy can be found in the popular sorting algorithm known as *Quicksort*.
- Even with Quicksort, although it is a recursive algorithm by nature—and easiest to understand in recursive from—it's actually more efficient if the recursion is removed.
- Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Towers of Hanoi Example

- The Tower of Hanoi is a mathematical puzzle that consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, making a conical shape.
- The objective of the puzzle is to move the entire stack to another rod, obeying the following rules: 1) Only one disk can be moved at a time. 2) Each disk can only be moved if it is the uppermost disk on a stack. 3) No disk may be placed on top of a smaller disk.
- With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is 2^n – 1, where n is the number of disks.
- The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a story about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of the Brahma. According to the legend, when the last move of the puzzle will be completed, the world will end. If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves it would take them 2^64–1 seconds, or roughly 585 billion years to finish, which is about 42 times the current age of the Universe.
- Using recursion often involves a key insight that makes everything simpler.
- The Tower of Hanoi solution consists of moving disks from the source peg A to the target peg C, using B as the spare peg.

```
void hanoi(int disc, char a = 'A', char b = 'B', char c = 'C')
{
    if (disc == 1)
    {   // base case
        cout << "Move disc " << disc << " from " << a
             << " to " << c << endl;
    }
    else
    {
        // recursive step
        hanoi(disc - 1, a, c, b);
        cout << "Move disc " << disc << " from " << a
             << " to " << c << endl;
        hanoi(disc - 1, b, a, c);
    }
}
```

- a more refined, but less clear solution.

```
void hanoi2(int disc, char a = 'A', char c = 'C', char b = 'B')
{
    if (disc > 0)
    {
        hanoi2(disc - 1, a, b, c);
        cout << "Move disk " << disc << " from " << a
             << " to " << c << endl;
        hanoi2(disc - 1, b, c, a);
    }
}
```