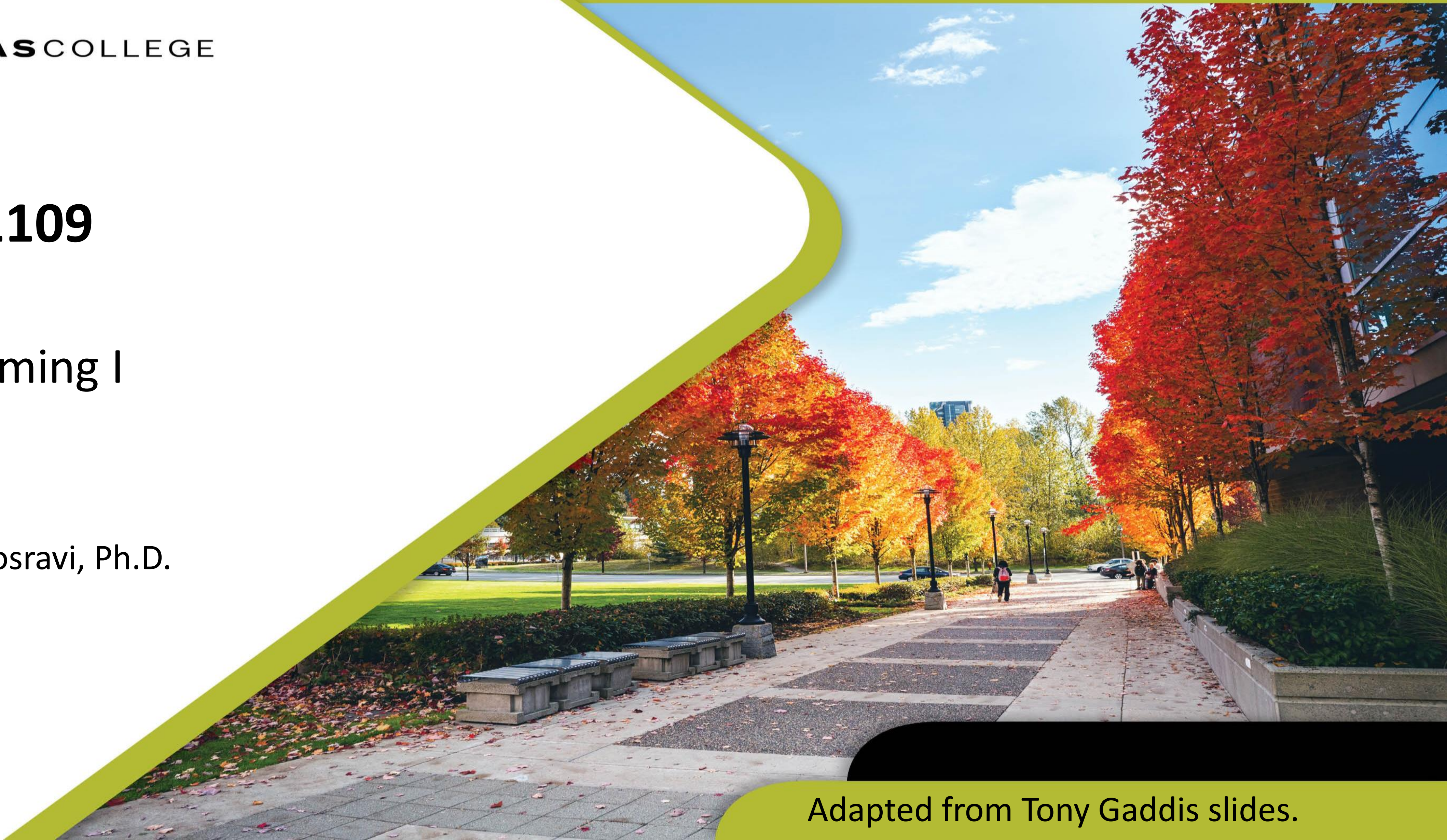
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 11



Adapted from Tony Gaddis slides.

Plan for Today

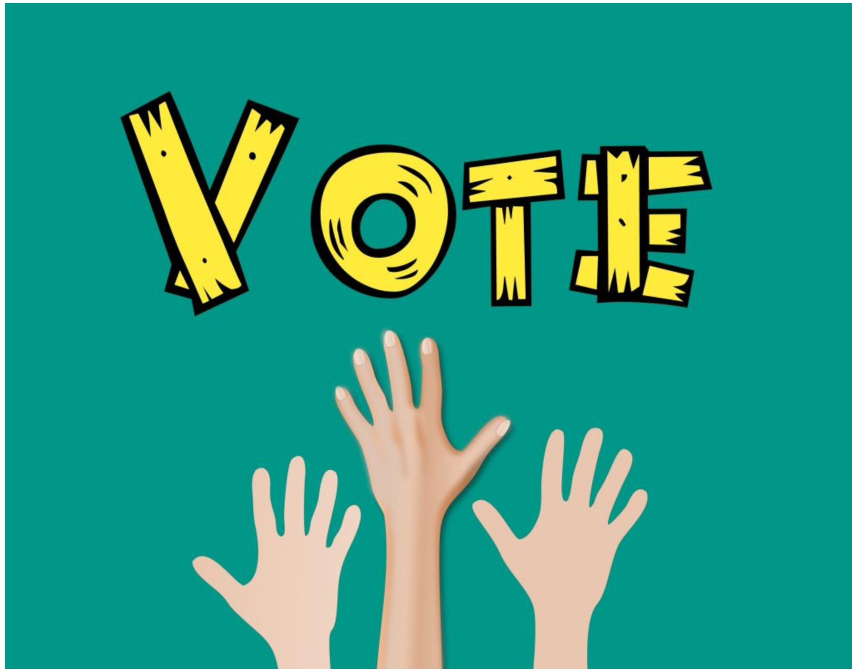
- Introduction to Exceptions
- Introduction to Recursion
- Problem Solving with Recursion

Poll 1 (Extra Credit)

The C++ compiler is able to catch all errors in a program, including run-time errors.

a) True
b) False

Please use the "Poll" window to participate for extra credit! One answer only please!



Introduction to Exceptions

Exceptions

- Error testing is usually a straightforward process involving **`if`** statements or other control mechanisms.
- For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
        cout << "ERROR: Cannot divide by zero.\n";
    else
        quotient = numerator / denominator;
```

- What if similar code is part of a function that returns the quotient:

```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        cout << "ERROR: Cannot divide by zero.\n";
        return 0;
    }
    else
        return static_cast<double>(numerator) / denominator;
}
```

Exceptions

- Error testing is usually a straightforward process involving **`if`** statements or other control mechanisms.
- For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
        cout << "ERROR: Cannot divide by zero.\n";
    else
        quotient = numerator / denominator;
```

- What if similar code is part of a function that returns the quotient:

```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        cout << "ERROR: Cannot divide by zero.\n";
        return 0;
    }
    else
        return static_cast<double>(numerator) / denominator;
}
```

Here, returning **`0`** is unreliable because **`0`** could be a valid result of a division operation. So, this if statement is not an ideal way to catch this error before it occurs.

Throwing an Exception

- One way of handling complex error conditions is with **exceptions**.
- An exception is a value (or an object) that signals an error.
- When the error occurs, an exception is "thrown".

```
double divide(int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR: Cannot divide by zero.\n";
    else
        return static_cast<double>(numerator) / denominator;
}
```

- The **`throw`** keyword is followed by an argument, which can be any value. The line containing a throw statement is known as the **throw point**.
- When a throw statement is executed, control is passed to another part of the program known as an **exception handler**. When an exception is thrown by a function, the function aborts.

Handling an Exception

- To handle an exception, a program must have a **`try/catch`** construct. The general format of the try/catch construct is:

```
try
{
    // code here calls functions or object member
    // functions that might throw an exception.
}
catch (ExceptionParameter)
{
    // code here handles the exception
}
// Repeat as many catch blocks as needed.
```

- The first part of the construct is the **`try`** block, which is followed by a block of code executing any statements that might cause an exception.
- The **`try`** block is immediately followed by one or more **`catch`** blocks, which are the **exception handlers**.
- A **`catch`** block is followed by a set of parentheses containing the definition of an exception parameter.

Handling an Exception

- For example, here is a try/catch construct that can be used with the divide function:

```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
    cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

If this statement throws an exception...

... then this statement is skipped.

If the exception is a string, the program jumps to this catch clause.

After the catch block is finished, the program resumes here.

- Since the **`divide()`** function throws an exception whose value is a **`string`**, there must be an exception handler that catches a **`string`**.
- The catch block shown catches the error message in the **`exceptionString`** parameter and then displays it with **`cout`** .

Uncaught Exceptions

- There are two possible ways for a thrown exception to go “uncaught”.
- The first possibility is for the **`try/catch`** construct to contain no **`catch`** blocks with an exception parameter of the correct data type.
- The second possibility is for the exception to be thrown from outside a **`try`** block.
- In both cases, the exception will cause the entire program to abort execution.

Example

```
// This program demonstrates an exception being thrown and caught.
#include <iostream>
#include <string>
using namespace std;

// Function prototype
double divide(int, int);

int main()
{
    int num1, num2;  // To hold two numbers
    double quotient; // To hold the quotient of the numbers

    // Get two numbers.
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    // Divide num1 by num2 and catch any
    // potential exceptions.
    try
    {
        quotient = divide(num1, num2);
        cout << "The quotient is " << quotient << endl;
    }
    catch (string exceptionString)
    {
        cout << exceptionString;
    }

    cout << "End of the program.\n";
    return 0;
}

double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        string exceptionString = "ERROR: Cannot divide by zero.\n";
        throw exceptionString;
    }

    return static_cast<double>(numerator) / denominator;
}
```

Introduction to Recursion

Introduction to Recursion

- We have seen instances of functions calling other functions.
- In a program, the **`main()`** function might call function **A**, which then might call function **B**.
- It is also possible for a function to call itself!
- A function that calls itself is known as a **recursive function**.

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

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
.
.
.
```

Introduction to Recursion

- Like a loop, a recursive function must have some way to control the number of times it repeats.
- The code below shows a modified version of the **`message()`** function.
- In this program, the **`message()`** function receives an argument that specifies the number of times the function should display the message.

```
#include <iostream>
#include <string>
using namespace std;

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

int main()
{
    message(5);
    return 0;
}
```

```
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
This is a recursive function.
```

DOUGLAS COLLEGE

Problem Solving with Recursion

Introduction to Recursion

- In many instances, the use of a recursive function enables an intuitive and simple solution to otherwise complex problems.
- Recursive functions typically have the following form:

```
if this is a simple case
        solve it
else
        redefine the problem using recursion
```

- In order to apply this approach, first, we identify at least one case in which the problem can be solved without recursion. This is known as the **base case**.
- Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the **recursive case**.

Example – Integer Multiplication

- Integer multiplication is a simple example of recursion:

- **4 × 3 = 4 + (4 × 2)**
  = 4 + (4 + (4 × 1))

- Each **(4 × n)** may be replaced with a function call:

- **mult(4, 3) = 4 + mult(4, 2)**
  = 4 + 4 + mult(4, 1)

```
#include <iostream>
using namespace std;

int mult(int num1, int num2)
{
    int ans;

    if (num2 == 1)
        ans = num1;
    else
        ans = num1 + mult(num1, num2 - 1);

    return ans;
}

void main()
{
    int a = 2;
    int b = 3;

    cout << a << " * " <<  b
        << " = " << mult(a, b) << endl;
}
```

Example – Integer Multiplication

```
int mult(4, 3)
{
    int ans;

    if (num2 == 1)
        ans = 4;
    else
        ans = 4 + mult(4, 2);
```

```
int mult(4, 2)
{
    int ans;

    if (num2 == 1)
        ans = 4;
    else
        ans = 4 + mult(4, 1);
```

```
int mult(4, 1)
{
    int ans;

    if (num2 == 1)
        ans = 4;
    else
        ans = 4 + mult(4, 0);

    return ans;
}
```

```
int mult(4, 2)
{
    int ans;

    if (num2 == 1)
        ans = 4;
    else
        ans = 4 + 4;
    return ans;
}
```

```
int mult(4, 3)
{
    int ans;

    if (num2 == 1)
        ans = 4;
    else
        ans = 4 + 8;
    return ans;
}
```

12 is returned

Example – Factorial

- Factorial is another example of recursion:

- **`4! = 4 × 3!`**
  `= 4 × 3 × 2!`
  `= 4 × 3 × 2 × 1!`

- Each **`n!`** may be replaced with a function call:

- **`fact(4) = 4 × fact(3)`**
  `= 4 × 3 × fact(2)`
  `= 4 × 3 × 2 × fact(1)`

```cpp
#include <iostream>
using namespace std;

long int fact(int num)
{
    long int ans;

    if (num == 1)
        ans = num;
    else
        ans = num * fact(num - 1);

    return ans;
}

int main()
{
    int a = 6;

    cout << a << "! = " << fact(a) << endl;
}
```

Example – Factorial

```
long int fact(3)
{
    long int ans;
    if (num == 1)
        ans = num;
    else
        ans = 3 * fact(2);
```

```
long int fact(2)
{
    long int ans;
    if (num == 1)
        ans = num;
    else
        ans = 2 * fact(1);
```

```
long int fact(1)
{
    long int ans;

    if (num == 1)
        ans = 1;
    else
        ans = 1 * fact(0);

    return ans;
}
```

```
long int fact(2)
{
    long int ans;

    if (num == 1)
        ans = num;
    else
        ans = 2 * 1;

    return ans;
}
```

```
long int fact(3)
{
    long int ans;

    if (num == 1)
        ans = num;
    else
        ans = 3 * 2;

    return ans;
}
```

6 is returned

Example – Summing a Range of List Elements

- Function receives a list containing range of elements to be summed, index of starting item in the range, and index of ending item in the range.
- Base case:
  ```
  if start_index > end_index:
        return 0
  ```
- Recursive case:
  ```
  return current_number + sum(list, start+1, end)
  ```
- In essence, this statement says “return the value of the first item in the range plus the sum of the rest of the items in the range.”

```
#include <iostream>
using namespace std;

int range_sum(int* num_arr, int start, int end);

int main()
{
    int numbers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    int my_sum = range_sum(numbers, 2, 5);

    cout << "The sum of items 2 through 5 is " << my_sum << endl;

    return 0;
}

int range_sum(int* num_arr, int start, int end)
{
    if (start > end)
        return 0;
    else
        return num_arr[start] + range_sum(num_arr, start + 1, end);
}
```

Example – The Fibonacci Series

- The Fibonacci Series (Leonardo Fibonacci, circa 1170):
  0, 1, 1,2,3,5,8, 13,21,34,55,89,144,233,…
- Notice after the second number, each number in the series is the sum of the two previous numbers.
- Fibonacci series: has two base cases:
  `if n = 0 then Fib(n) = 0`
  `if n = 1 then Fib(n) = 1`
- The recursive case is:
  `if n > 1 then Fib(n) = Fib(n-1) + Fib(n-2)`

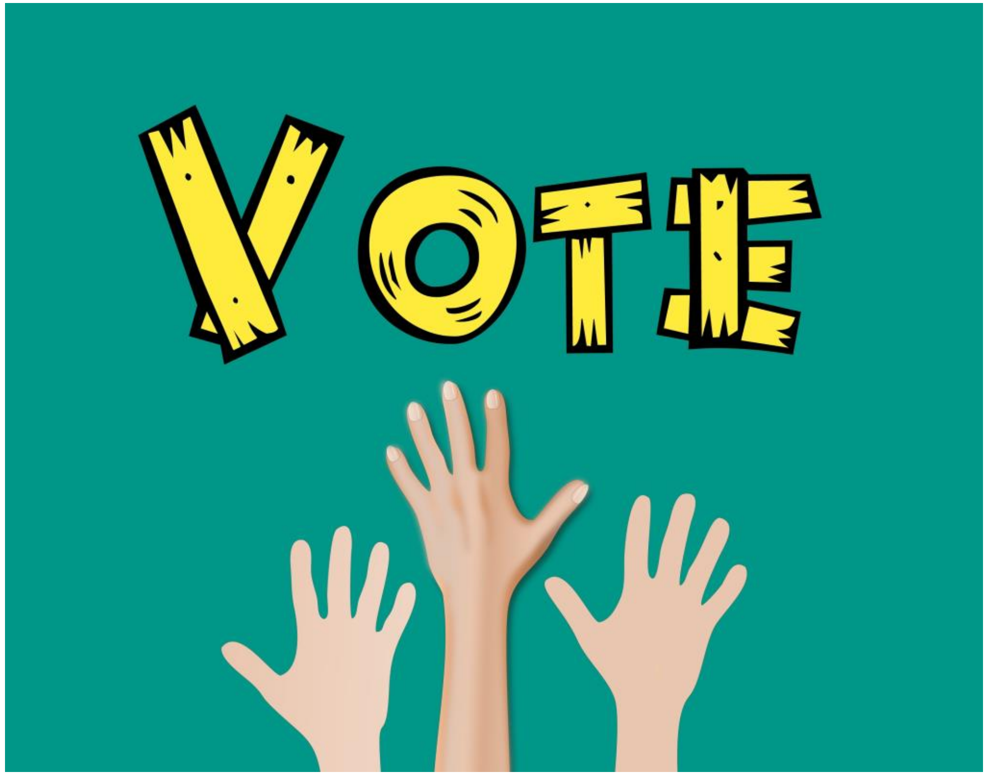
```
// This program demonstrates a recursive function
// that calculates Fibonacci numbers.
#include <iostream>
using namespace std;

// Function prototype
int fib(int);

int main()
{
    cout << "The first 10 Fibonacci numbers are:\n";
    for (int x = 0; x < 10; x++)
        cout << fib(x) << " ";
    cout << endl;
    return 0;
}

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 case
}
```

Poll 2 (Extra Credit)

In general, recursive algorithms are more efficient in terms of compute time than their iterative counterparts.

a) TRUE
b) FALSE

Please use the Poll window to participate for extra credit!



In-Class Exercise #1

Implement a recursive function in C++ for calculating the greatest common divisor (GCD) of two numbers. The GCD of two positive integers x and y is determined as follows:

If x can be evenly divided by y, then gcd(x, y) =y

Otherwise, gcd(x, y) = gcd(y, remainder of x/y)

In-Class Exercise #1

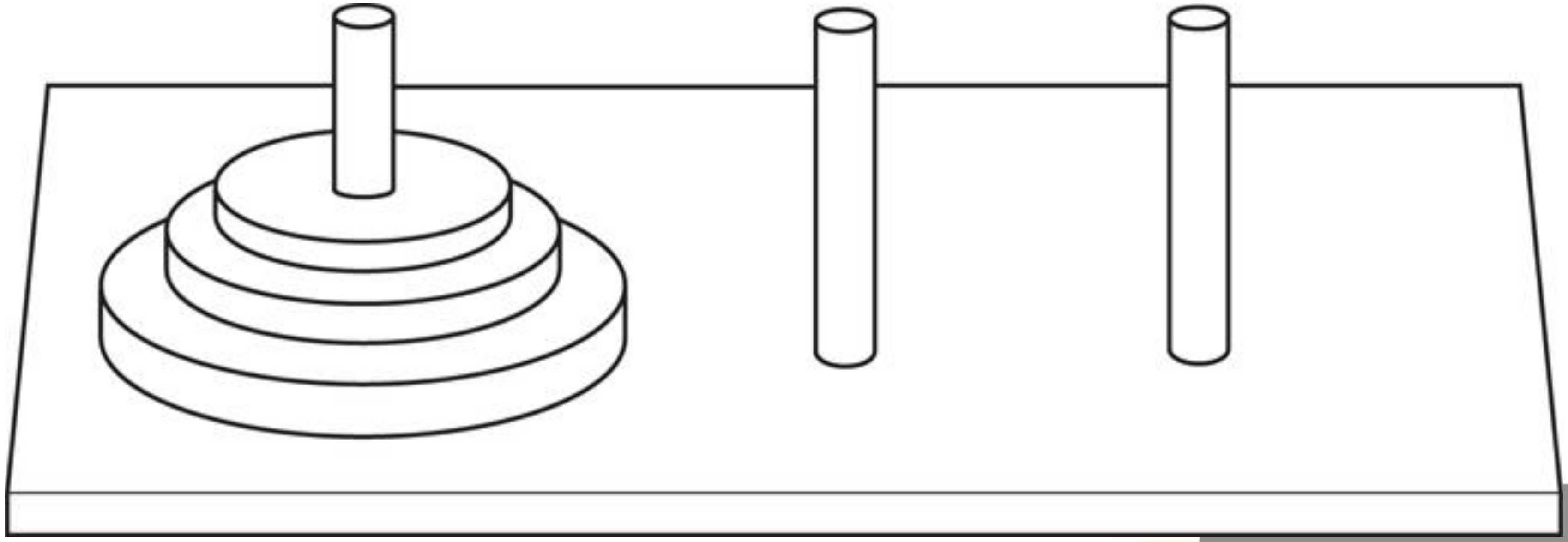
```cpp
// This program demonstrates a recursive function to calculate
// the greatest common divisor (gcd) of two numbers.
#include <iostream>
using namespace std;

// Function prototype
int gcd(int, int);

int main()
{
    int num1, num2;

    // Get two numbers.
    cout << "Enter two integers: ";
    cin >> num1 >> num2;

    // Display the GCD of the numbers.
    cout << "The greatest common divisor of " << num1;
    cout << " and " << num2 << " is ";
    cout << gcd(num1, num2) << endl;
    return 0;
}

int gcd(int x, int y)
{
    if (x % y == 0)
        return y;                 // Base case
    else
        return gcd(y, x % y);     // Recursive case
}
```

Recursion vs. Looping

- Reasons not to use recursion:
  - Less efficient: entails function calling overhead that is not necessary with a loop.
  - Usually a solution using a loop is more intuitive than a recursive solution.
- Some problems are more easily solved with recursion than with a loop.
  - Example: Fibonacci, where the mathematical definition lends itself to recursion.
- If a recursive solution is evident for a particular problem, and the recursive algorithm does not slow system performance an intolerable amount, then recursion would be a good design choice.

The Towers of Hanoi

- Mathematical game commonly used to illustrate the power of recursion.
  - Uses three pegs and a set of discs in decreasing sizes.
  - **Goal of the game**: move the discs from leftmost peg to rightmost peg.
    - Only one disc can be moved at a time.
    - A disc cannot be placed on top of a smaller disc.
    - All discs must be on a peg except while being moved.



Video Illustration

The Towers of Hanoi

- **Problem statement**: move n discs from peg A to peg C using peg B as a temporary peg.
- **Recursive solution**:
  - If n == 1:
    - Move disc from peg A to peg C
  - Otherwise:
    - Move n-1 discs from peg A to peg B, using peg C
    - Move remaining disc from peg A to peg C
    - Move n-1 discs from peg B to peg C, using peg A

The Towers of Hanoi

```cpp
// This program displays a solution to the Towers of
// Hanoi game.
#include <iostream>
using namespace std;

// Function prototype
void moveDiscs(int, int, int, int);

int main()
{
    const int NUM_DISCS = 3;    // Number of discs to move
    const int FROM_PEG = 1;     // Initial "from" peg
    const int TO_PEG = 3;       // Initial "to" peg
    const int TEMP_PEG = 2;     // Initial "temp" peg

    // Play the game.
    moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
    cout << "All the pegs are moved!\n";
    return 0;
}

void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
    if (num > 0)
    {
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
        cout << "Move a disc from peg " << fromPeg
            << " to peg " << toPeg << endl;
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
    }
}
```

Thank you.

DOUGLASCOLLEGE