GAME2001 Data Structures and Algorithms

Fall 2020

Week 3

Bitwise Operators

Recursion

Bitwise Operators

- AND Operator
  - logical AND (&&)

```
if ((x == 5) && (y == 7))
    DoSomething();
```

  - bitwise AND (&)

```
  0110 1011 1000 0101
& 0001 1111 1011 1001
-------------------------------
  0000 1011 1000 0001
```

```
0 & 0 = 0
0 & 1 = 0        x & 0 = 0
1 & 0 = 0        x & 1 = x
1 & 1 = 1
```

Bitwise Operators

- OR Operator
  - logical OR (||)

```
if ((x == 5) || (y == 7))
    DoSomething();
```

  - bitwise OR (|)

```
  0110 1011 1000 0101
| 0001 1111 1011 1001
---------------------------------
  0111 1111 1011 1101
```

```
0 | 0 = 0
0 | 1 = 1        x | 0 = x
1 | 0 = 1        x | 1 = 1
1 | 1 = 1
```

Bitwise Operators

- XOR Operator (exclusive OR)
  - no logical equivalent for it in C++
  - Bitwise XOR (^)

```
   0110 1011 1000 0101
^  0001 1111 1011 1001
-------------------------------
   0111 0100 0011 1100
```

```
0 ^ 0 = 0
0 ^ 1 = 1      x ^ 0 = x
1 ^ 0 = 1      x ^ 1 = ~x
1 ^ 1 = 0
```

Bitwise Operators

- Bitwise Shifts
  - shift bits to the left or to the right
  - Shift left (<<)
  - Shift right (>>)

[integer] [operator] [number of places];

```
// Precondition:   x == 0000 0110 1001 0011
// Postcondition: y == 0000 1101 0010 0110
y = x << 1;
```

Bitwise Operators

```
// Precondition:   x == 0110 1111 1001 0001
// Postcondition: y == 0000 0110 1111 1001
y = x >> 4;
```

Bitwise Operators

- Extracting and Clearing Values
- 32-bit color dword

  AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB
- Extract red value:

```
dwColor:   AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB
mask:    & 0000 0000 1111 1111 0000 0000 0000 0000
           ---------------------------------------
result:    0000 0000 RRRR RRRR 0000 0000 0000 0000


Previous:   0000 0000 RRRR RRRR 0000 0000 0000 0000
Shift:     >> 16
---------------------------------------
Result:     0000 0000 0000 0000 0000 0000 RRRR RRRR == RRRR RRRR
```

Bitwise Operators

- Masks for 16-bit color

```
Color word:   RRRR RGGG GGGB BBBB
Red mask:     1111 1000 0000 0000   ==   0xF800
Green mask:   0000 0111 1110 0000   ==   0x07E0
Blue mask:    0000 0000 0001 1111   ==   0x001F
```

Bitwise Operators

- Swapping Variables

```
                    //  Value of x                          Value of y
                    // ----------------------------------------------
int x, y;           //  0                                   0
x = CONST_A;        //  CONST_A                             0
y = CONST_B;        //  CONST_A                             CONST_B
x = x ^ y;          //  CONST_A ^ CONST_B                   CONST_B
y = x ^ y;          //  CONST_A ^ CONST_B                   CONST_A ^ CONST_B ^ CONST_B == CONST_A ^ 0 == CONST_A
x = x ^ y;          //  CONST_A ^ CONST_A ^ CONST_B         CONST_A
                    //  == 0 ^ CONST_B == CONST_B           CONST_A
                    //  CONST_B                             CONST_A
```

Bitwise Operators

- Replacing Arithmetic Operations

  0010 1101 = (**1** * 2^5) + (**1** * 2^3) + (**1** * 2^2) + (**1** * 2^0)
  0010 1101 * 2 = 2(**1** * 2^5) + 2(**1** * 2^3) + 2(**1** * 2^2) + 2(**1** * 2^0)
  0010 1101 * 2 = (**1** * 2^6) + (**1** * 2^4) + (**1** * 2^3) + (**1** * 2^1)
  0010 1101 * 2 = **0101 1010**

- The following pairs of statements are all equivalent to one another:

```
x = y * 8;
x = y << 3;

x = y * 64;
x = y << 6;

x = y * 32768;
x = y << 15;
```

Bitwise Operators

- These pairs of statements are equivalent to one another as well:

```
x = y / 4;
x = y >> 2;

x = y / 32;
x = y >> 5;
```

Bitwise Operators

- Mod operation

```
x = y % 8;
x = y & 7;

x = y % 32;
x = y & 31;

x = y % 256;
x = y & 255;
```

Recursion

- occurs when a method calls itself within the body of its definition
  - execute their contents under some condition
    - when met, the recursion stops
- the purpose of a recursive function is to find the solution to a small piece of a bigger problem

The Pros And Cons Of Recursion

- Pros
  - It is conceptually easier to code
  - It is easier to maintain and modify in some situations

- Cons
  - There is overhead with the calling of functions
  - It can cause a stack overflow
  - Using recursion can be less effective in performance than using loops

Recursion

- Two different types:

- Tail Recursion
  - recursive definition that calls itself at the end of the function
    - where no statements follow it and no recursive statements come before it

```
int recursion(int param)
{
    if(param < 1)
        return 0;

    // Perform some task


    return recursion(param - 1);
}
```

Recursion

- Nontail recursion
  - a method that defines statements after the recursive call and/or if there is more than one recursive call in the same body

```
int recursion(int param)
{
    if(param < 1)
        return 0;

    recursion(param - 1);

    // Perform some task


    recursion(param - 1);
}
```

Recursion

- For both a condition is specified somewhere in the function that keeps the calls from becoming infinite
- Altering the parameter each time provides uniqueness to exist to monitor the depth of the method

Recursion – Example 2

- Binary search recursion
  - Very similar to our non recursive implementation
  - Returns
    - -1 for not found
    - Index found
    - the recursive call for cases where we must keep searching

```cpp
template <class T>
class OrderedArray
{
  public:
    int search(T searchKey)
    {
      return binarySearch(searchKey, 0, m_numElements - 1);
    }

    int binarySearch(T searchKey, int lowerBound, int upperBound)
    {
      assert(m_array != NULL);
      assert(lowerBound >= 0);
      assert(upperBound < m_numElements);

      int current = (lowerBound + upperBound) >> 1;

      if(m_array[current] == searchKey)
      {
        return current;
      }
      else if(lowerBound > upperBound)
      {
        return -1;
      }
      else
      {
        if(m_array[current] < searchKey)
          return binarySearch(searchKey, current + 1, upperBound);
        else
          return binarySearch(searchKey, lowerBound, current - 1);
      }

      return -1;
    }
};
```

```cpp
int search(T searchKey)
{
    assert(m_array != NULL);

    int lowerBound = 0;
    int upperBound = m_numElements - 1;
    int current = 0;

    while(1)
    {
        current = (lowerBound + upperBound) >> 1;

        if(m_array[current] == searchKey)
        {
            return current;
        }
        else if(lowerBound > upperBound)
        {
            return -1;
        }
        else
        {
            if(m_array[current] < searchKey)
                lowerBound = current + 1;
            else
                upperBound = current - 1;
        }
    }

    return -1;
}
```

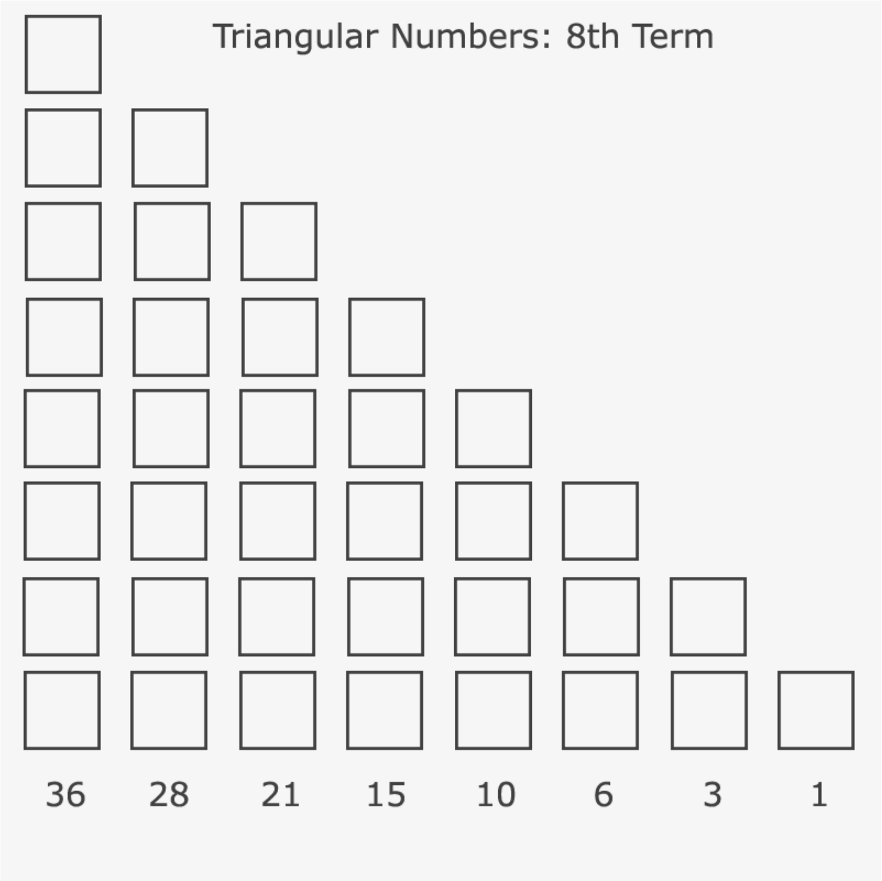
```cpp
#include <iostream>
#include "Arrays.h"

using namespace std;

int main(int args, char **argc)
{
    cout << "Recursive Binary Search Example" << endl;

    OrderedArray<int> array(3);

    array.push(43);
    array.push(8);
    array.push(23);
    array.push(94);
    array.push(17);
    array.push(83);
    array.push(44);
    array.push(28);

    cout << "Ordered array contents:";

    for(int i = 0; i < array.GetSize(); i++)
    {
        cout << " " << array[i];
    }

    cout << "Search for 43 was found at index: ";
    cout << array.search(43) << endl << endl;

    return 1;
}
```

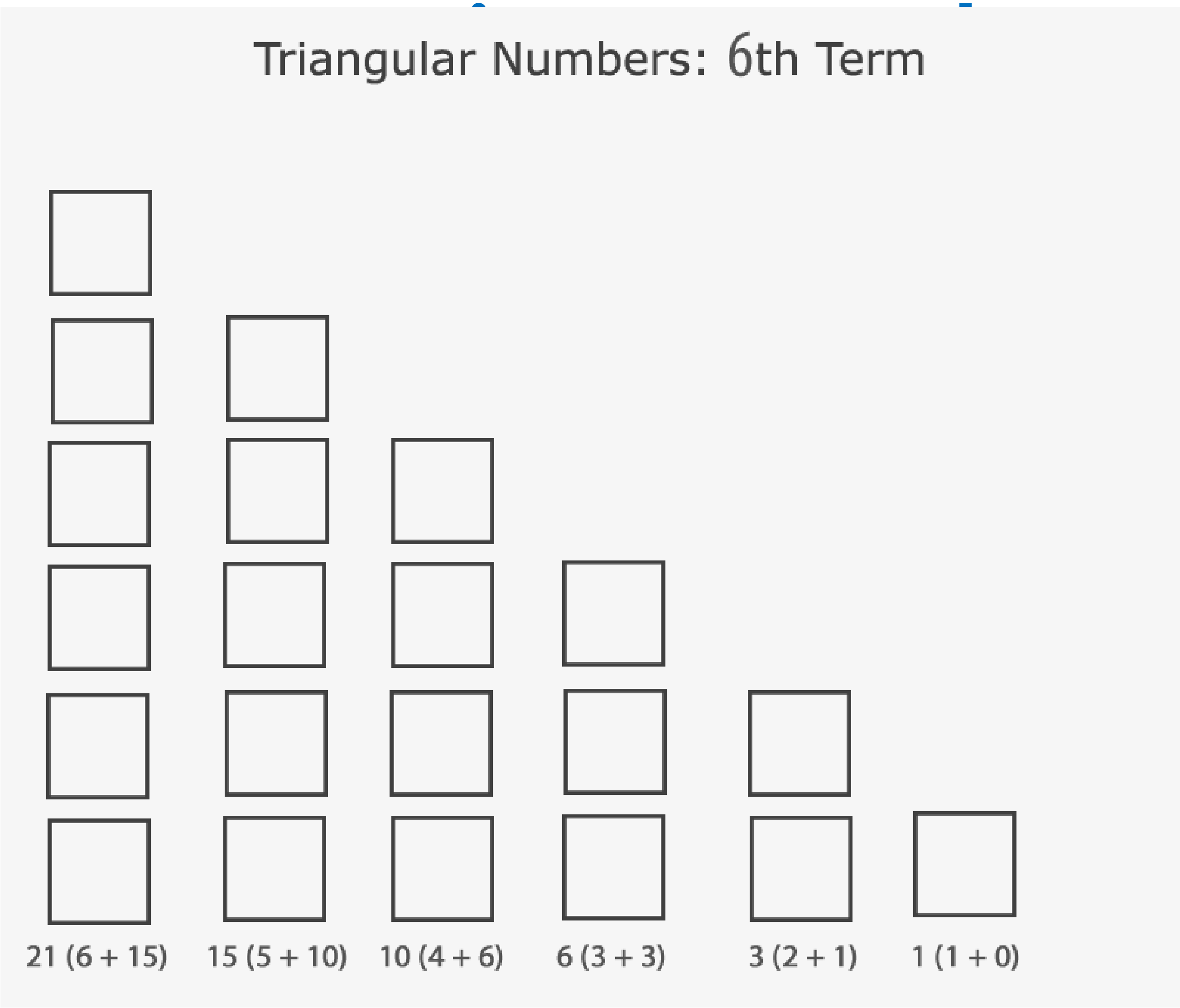
```
Ordered array contents: 8 17 23 28 43 44 83 94.
Search for 43 was found at index: 4.
```

Recursion – Example 3

- Triangular Numbers
  - a series of numbers are created using the nth term
  - 1, 3, 6, 10, 15, 21, 28, 36, 45, 55
    - the 11th term would be 66 because adding 11 to 55 gives us 66
  - the nth term is added to the value that came before it

Recursion – Example 3




Triangular Numbers: 6th Term
21 (6 + 15)
15 (5 + 10)
10 (4 + 6)
6 (3 + 3)
3 (2 + 1)
1 (1 + 0)

Recursion – Example 3

- Non recursive

```
int TriangularNumber(int term)
{
    int value = 0;
    for(; term > 0; term--)
    {
        value += term;
    }
    return value;
}
```

Recursion – Example 3

- Recursive

```
int TriNumRecursion(int term)
{
   assert(term >= 1);

   if(term == 1)
      return 1;

   return(TriNumRecursion(term - 1) + term);
}
```

Recursion – Example 3

```
int main(int args, char **argc)
{
    cout << "Triangular Numbers Example" << endl;

    cout << "The value of the 18th term using a loop: ";
    cout << TriNumLoop(18) << endl;

    cout << "The value of the 25th term using recursion: ";
    cout << TriNumRecursion(25) << endl;

    return 1;
}
```

```
Triangular Numbers Example
The value of the 18th term using a loop: 171
The value of the 25th term using recursion: 325
```

Recursion – Example 4

- Factorials
  - To find the value of the nth term we take the multiplication of the term and the term –1 recursively
  - Double factorial
    - same as calculating a factorial except we subtract 2 from the term with each call instead of 1.

- Finding the term of 0 when looking at factorials, by definition, returns a value of 1. This is also true for 1 itself.

Recursion – Example 4

```
int factorial(int x)
{
   assert(x >= 0);

   if(x == 0)
      return 1;

   return(factorial(x - 1) * x);
}
```

```
int doubleFactorial(int x)
{
   assert(x >= 0);

   if(x == 0)
      return 1;

   return(factorial(x - 2) * x);
}
```

- in general looks exactly like the triangular numbers
  - with the exception that we are looking for 0,
    - which returns 1 by definition of factorials,
  - and we are using multiplication during each step instead of addition

Recursion – Example 4

```
int main(int args, char **argc)
{
    cout << "Factorials" << endl;

    cout << "The factorial of 3: ";
    cout << factorial(3) << endl;

    cout << "The double factorial of 4: ";
    cout << doubleFactorial(4) << endl;

    cout << endl;

    return 1;
}
```

```
Factorials
The factorial of 3: 6
The double factorial of 4: 8
```