More on Loops

In this Week

- The *for* loop
- The *do-while* loop
- The *break* and *continue* statements
- Nested Loops: *loops inside loops*

The *for* loop

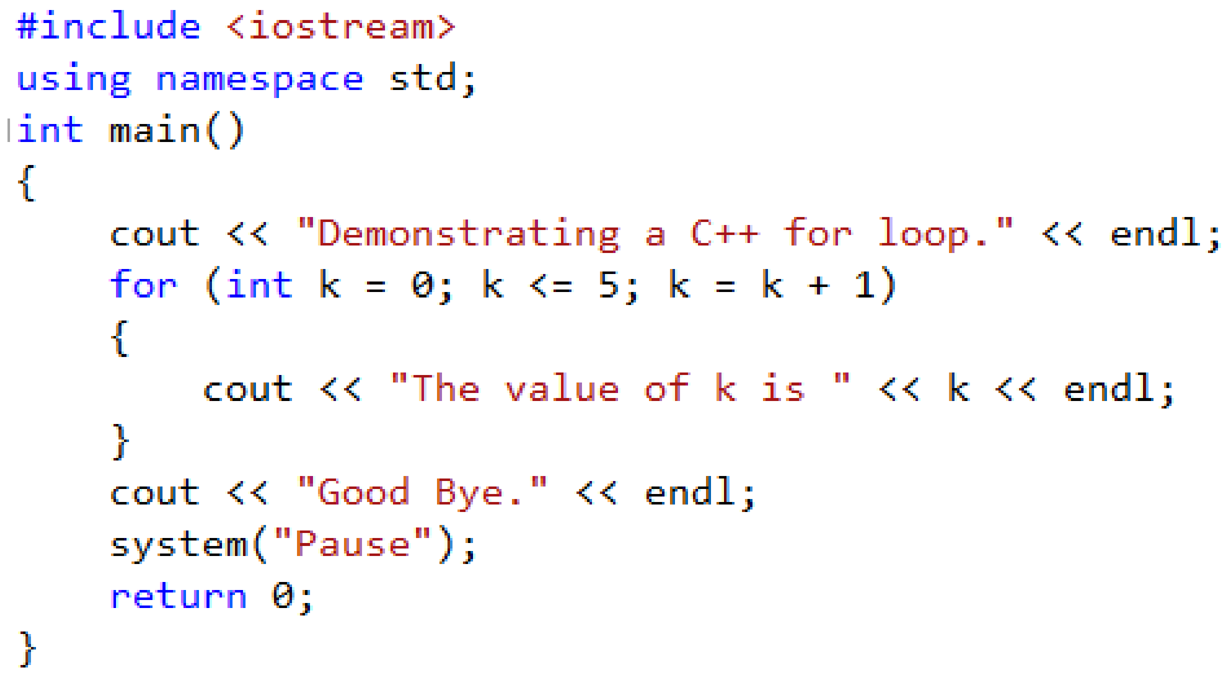
- **for loop: Syntax**

```
for (initialization; boolean_exp; update_action)
{
        Block of the for loop
}
```

- **Initialization** is executed only once before the loop begins
- The block is repeatedly executed as long as the **boolean_exp** is evaluated to true
- The **update_action** is executed at the end of each iteration
- The **boolean_exp** is checked after each **update_action**

The *for* loop: Example

- Consider the following for loop example



```
#include <iostream>
using namespace std;
int main()
{
    cout << "Demonstrating a C++ for loop." << endl;
    for (int k = 0; k <= 5; k = k + 1)
    {
        cout << "The value of k is " << k << endl;
    }
    cout << "Good Bye." << endl;
    system("Pause");
    return 0;
}
```

- In this case, the initialization section is used to declare and initialize the variable **k**. This makes **k** a variable declared inside the for loop block only. Therefore it does **NOT** exist outside the for loop block
- The curly brackets designate the block of the for loop. They can be omitted if the block contains only one statement
- If the curly brackets are omitted then **only the first** statement belongs to the block
- How many iterations does the for loop perform?

The *for* loop: practice questions

1. Write a program to print the even integers 0,2,4,6,…28 using a *for* loop
2. Write a program that declares an integer variable ***n***, assigns ***n*** an integer input from the user, and finally prints the integers ***n, n-1, n-2, …, 1.*** What is the output of the program if the input value for n ***n*** is less than ***1.***
3. Write a program that reads an integer value **n** and then prints **n** randomly generated integers in the range [-10, 10].
4. Write a program that reads an integer value **n** and then prints **n** randomly generated integer numbers in the range [-10, 10] and finally prints the sum, the product and the minimum of the numbers.
5. Write a C++ program that declares two integers ***a*** and ***b***, assigns each of the variables a random integer in the range [-10, 15], and then prints all the integers between a and b (or vice versa) **exclusive** (that is without including **a** and **b**).
6. Write a program that prints the integers 7,10, 13, 16,… **n** exclusive (that is without including **n**) where **n** is the first integer in the list divisible by 41.
7. Write a program that reads an integer number **n** greater than 1 from the user and then prints the number is prime or the number is not prime. Remember an integer number **n** greater than 1 is prime if it has no divisors among 2,3,4,…,n-1. Assume the user input **n** is greater than 1.

Infinite Loops

- One of the most common mistakes in looping structures is getting into an infinite loop
- An infinite loop is a looping structure that does infinite iterations
- Most often the cause of an infinite loop mistake is not paying attention to the update_action of a loop
- That is if we do not modify the value of the looping variable properly in the update_action, then we will most likely end up with an infinite loop
- Consider the following program and determine its output

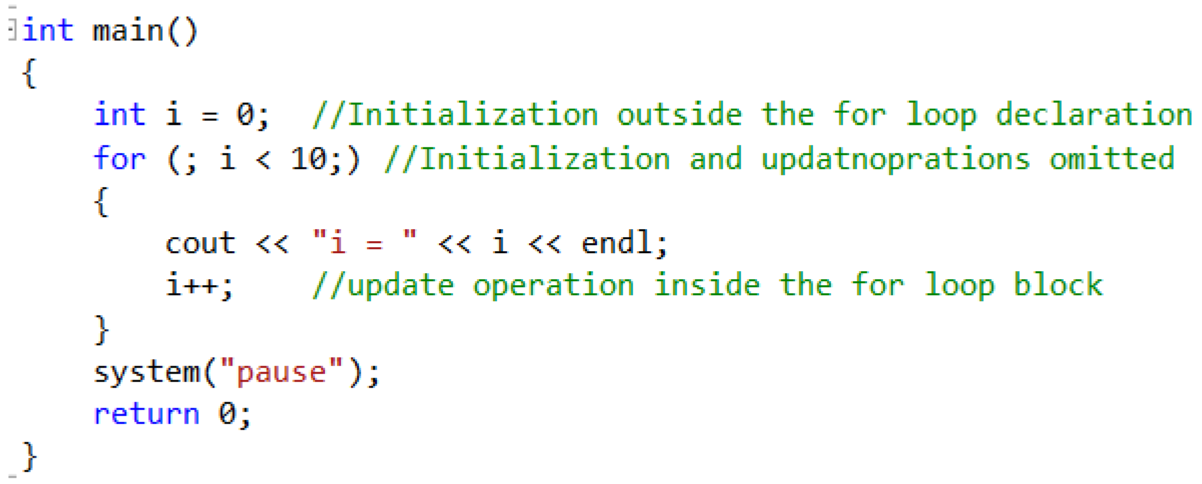
```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Demonstrating a C++ for loop." << endl;
    for (int k = 0; k <= 5; k + 1)
    {
        cout << "The value of k is " << k << endl;
    }
    cout << "Good Bye." << endl;
    system("Pause");
    return 0;
}
```

More on *for* loop

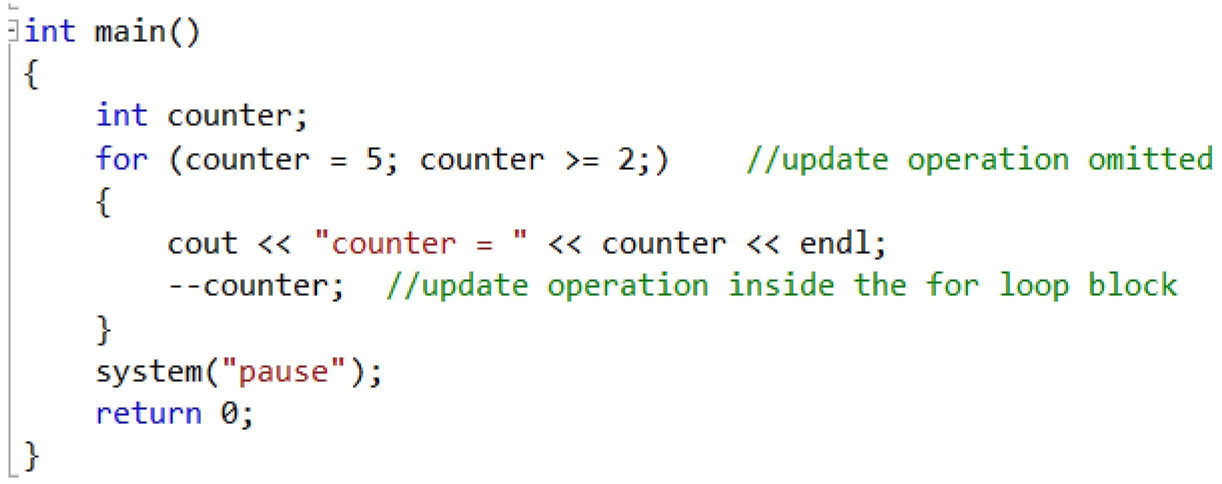
- Each section of the *for* loop (**initialization**, **boolean_exp**, and **update_action**) are optional
- If **initialization** is omitted, then no initialization is performed in the **for** loop; but it must be done before the loop
- If the **boolean_exp** is omitted, then it is always assumed **true** and therefore may create an infinite loop
- If the **update_action** is omitted, then no update action is performed; can be done inside the block

More *for* loop examples

Analyze the following two programs and determine their outputs



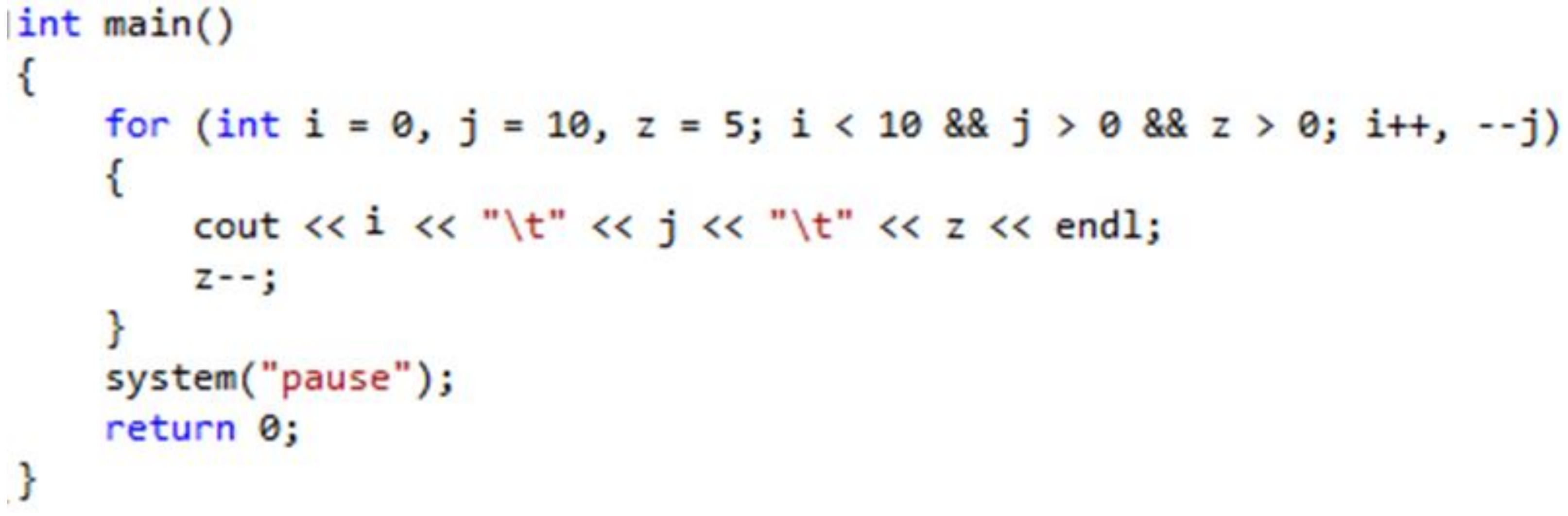
```
int main()
{
    int i = 0;  //Initialization outside the for loop declaration
    for (; i < 10;) //Initialization and updatnoprations omitted
    {
        cout << "i = " << i << endl;
        i++;    //update operation inside the for loop block
    }
    system("pause");
    return 0;
}
```



```
int main()
{
    int counter;
    for (counter = 5; counter >= 2;)    //update operation omitted
    {
        cout << "counter = " << counter << endl;
        --counter;  //update operation inside the for loop block
    }
    system("pause");
    return 0;
}
```

More *for* loop examples

- It is possible to have multiple variable initializations and multiple variables updates in the for loop; all we need is separate them with commas
- Also it is possible to have a Boolean expression with multiple relational operations connected by logical operators
- Consider the following program and determine its output



```
int main()
{
    for (int i = 0, j = 10, z = 5; i < 10 && j > 0 && z > 0; i++, --j)
    {
        cout << i << "\t" << j << "\t" << z << endl;
        z--;
    }
    system("pause");
    return 0;
}
```

More *for* loop examples

- **Example:-** Write a program that asks the user to enter a positive integer and then prints the positive integer entered.
- **Remark:-** Note that if the user enters a negative or zero inputs then your program must ask for the input again and again and again until the user enters a positive integer
- Obviously this requires a loop. Therefore the loop should be designed to keep on asking for the input until a positive number is entered

The *do-while* loop

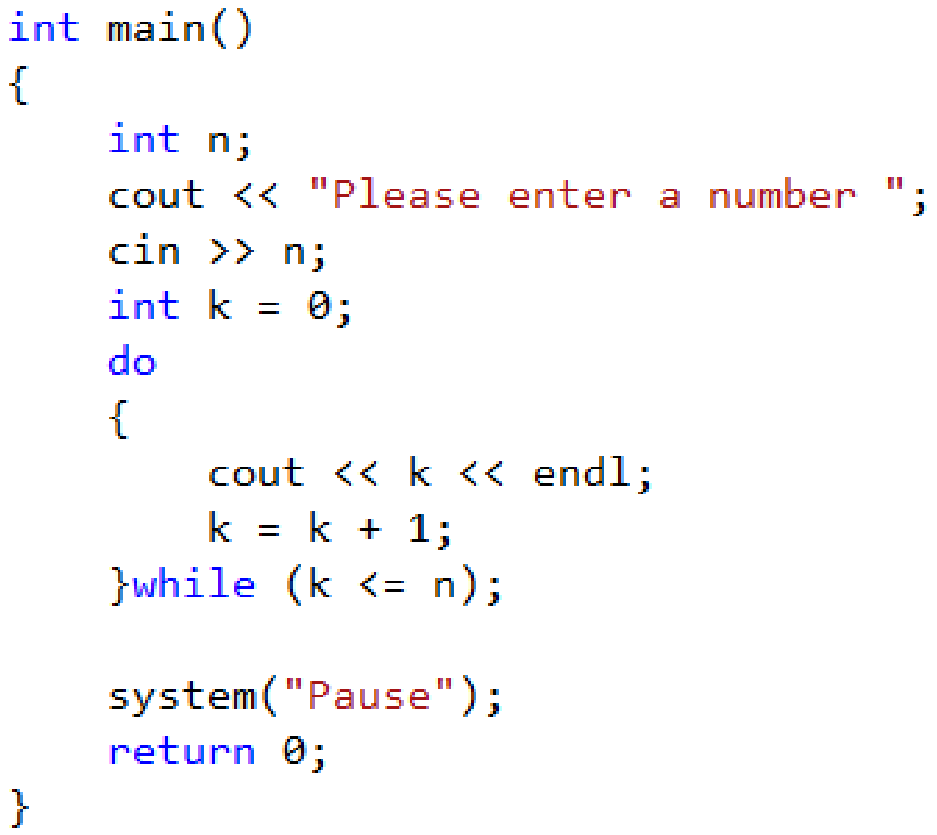
- Syntax

```
do
{
      Block of the do-while loop
} while (Boolean_expression); ← semicolon MUST!!!
```

- First the block is executed, then the Boolean expression is evaluated
- The block is executed as long as the Boolean expression is evaluated to true
- Therefore the block is always executed at least once

The *do-while* loop

- The following program reads an integer value **n** from the user and then uses a do-while loop to print all the integers 0, 1, 2, ..., **n**



```
int main()
{
    int n;
    cout << "Please enter a number ";
    cin >> n;
    int k = 0;
    do
    {
        cout << k << endl;
        k = k + 1;
    }while (k <= n);

    system("Pause");
    return 0;
}
```

The *do-while* loop

- Observe that the previous program is not correct!
- Logically speaking, we are not supposed to print any output if the user input value for **n** is less than zero
- But the program will always print at least one number output irrespective of the value of **n**
- This is due to the fact that the do-while loop always performs at least one iteration
- In order to remedy this shortcoming, we need to have an if statement that will skip the loop when we are not supposed print any output
- **Class Exercise:-** Modify the previous program so that it won't print any output if the value of **n** is less than zero

do-while loop: practice questions

1. Write a program to print the even integers 0,2,4,6,…28 using a *while* loop
2. Write a program that declares an integer variable ***n***, assigns ***n*** an integer input from from the user, and finally prints the integers ***n, n-1, n-2, …, 1.*** What is the output of the program if the input value for n ***n*** is less than ***1.***
3. Write a program that reads an integer value **n** and then prints **n** randomly generated integers in the range [-10, 10].
4. Write a program that reads an integer value **n** and then prints **n** randomly generated integer numbers in the range [-10, 10] and finally prints the sum, the product and the minimum of the numbers.
5. Write a C++ program that declares two integers ***a*** and ***b***, assigns each of the variables a random integer in the range [-10, 15], and then prints all the integers between a and b (or vice versa) **exclusive** (that is without including **a** and **b**).
6. Write a program that prints the integers 7,10, 13, 16,… **n** exclusive (that is without including **n**) where **n** is the first integer divisible by 41.
7. Write a program that reads an integer number **n** greater than 1 from user and then prints the number is prime or the number is not prime. Remember an integer number **n** greater than 1 is prime if it has no divisors from 2,3,4,…,n-1. Assume the user input **n** is greater than 1.

More *do-while* loop examples

- **Example:-** Write a program that asks the user to enter a positive integer and then prints the positive integer entered.
- **Remark:-** Note that if the user enters a negative or zero inputs then your program must ask for the input again and again and again until the user enters a positive integer
- Obviously this requires a loop. Therefore the loop should be designed to keep on asking for the input until a positive number is entered

The *while* versus *do-while* loop

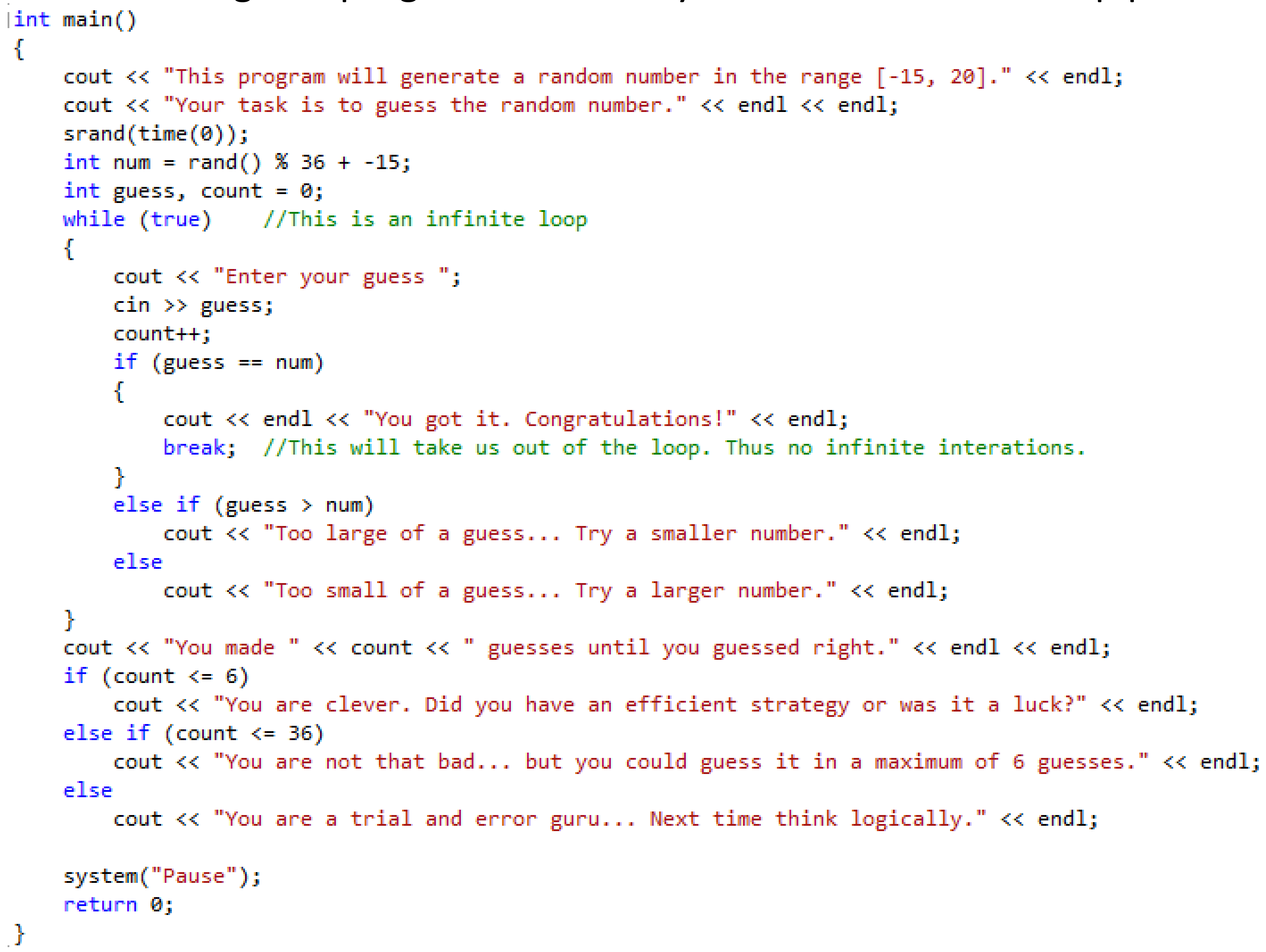
- Both these constructs of loops are very similar
- Their main difference is when the Boolean expression is evaluated
- In the *while* loop, the Boolean expression is evaluated before the loop starts; where as in the *do-while* loop the Boolean expression is evaluated after the loop does the first iteration
- This means the *do-while* loop block is always executed at least once
- For this reason, the *while* loop is preferred for its more flexibility

The *break* and *continue* statements

- C++ provides break and continue statements
- When these statements are inside loop constructs, then
  - ➢ The break statement forces the loop to terminate
  - ➢ The continue statement forces the statements inside the block of the loop which come after the continue statement to be skipped and go to the next iteration. In the case of **for** loop, it goes to the **update section** while in the case of the **while** and **do-while** loops, it goes to the **boolean expression**
- These statements alter the natural flow of execution of loop structures

The *break* Statement: Example

Analyze the following C++ program. How many iterations does the loop perform?

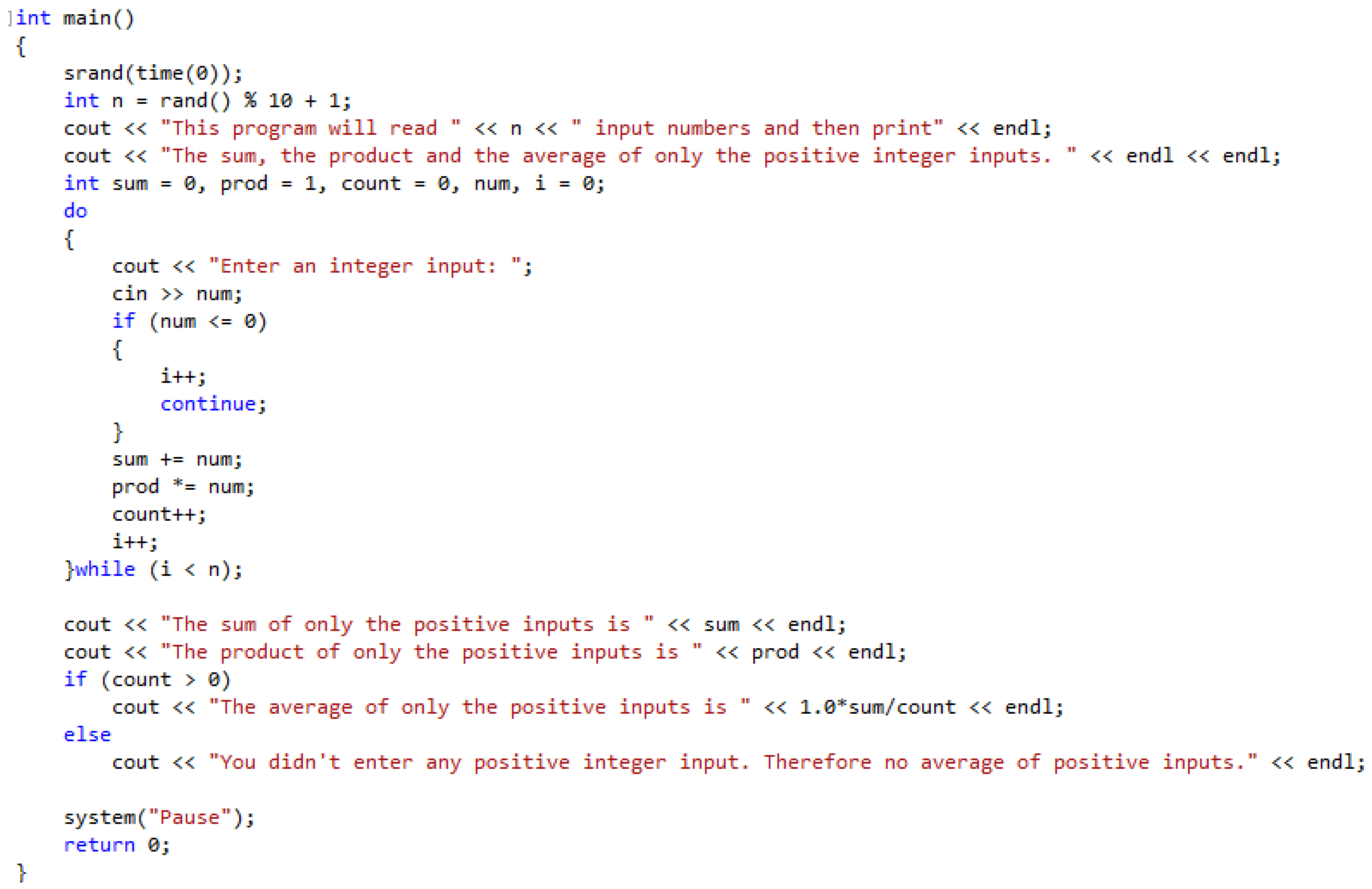


```
int main()
{
    cout << "This program will generate a random number in the range [-15, 20]." << endl;
    cout << "Your task is to guess the random number." << endl << endl;
    srand(time(0));
    int num = rand() % 36 + -15;
    int guess, count = 0;
    while (true)    //This is an infinite loop
    {
        cout << "Enter your guess ";
        cin >> guess;
        count++;
        if (guess == num)
        {
            cout << endl << "You got it. Congratulations!" << endl;
            break;  //This will take us out of the loop. Thus no infinite interations.
        }
        else if (guess > num)
            cout << "Too large of a guess... Try a smaller number." << endl;
        else
            cout << "Too small of a guess... Try a larger number." << endl;
    }
    cout << "You made " << count << " guesses until you guessed right." << endl << endl;
    if (count <= 6)
        cout << "You are clever. Did you have an efficient strategy or was it a luck?" << endl;
    else if (count <= 36)
        cout << "You are not that bad... but you could guess it in a maximum of 6 guesses." << endl;
    else
        cout << "You are a trial and error guru... Next time think logically." << endl;

    system("Pause");
    return 0;
}
```

The *continue* statement: Example

Analyze the following C++ program. How many iterations does the loop perform?



```cpp
int main()
{
    srand(time(0));
    int n = rand() % 10 + 1;
    cout << "This program will read " << n << " input numbers and then print" << endl;
    cout << "The sum, the product and the average of only the positive integer inputs. " << endl << endl;
    int sum = 0, prod = 1, count = 0, num, i = 0;
    do
    {
        cout << "Enter an integer input: ";
        cin >> num;
        if (num <= 0)
        {
            i++;
            continue;
        }
        sum += num;
        prod *= num;
        count++;
        i++;
    }while (i < n);

    cout << "The sum of only the positive inputs is " << sum << endl;
    cout << "The product of only the positive inputs is " << prod << endl;
    if (count > 0)
        cout << "The average of only the positive inputs is " << 1.0*sum/count << endl;
    else
        cout << "You didn't enter any positive integer input. Therefore no average of positive inputs." << endl;

    system("Pause");
    return 0;
}
```

Nested Loops: Motivation

- **Motivation 1:** Write a C++ program to draw the following pattern with 5 rows and 10 columns. Your cout statement must print only one * character at a time. This means you must use loop(s) to achieve the task.

```
**********
**********
**********
**********
**********
```

- **Motivation 2:** Write a C++ program to draw the following pattern.

```
*
**
***
****
*****
```

Nested Loops: Motivation

- Consider the multiplication table given below

1	**2**	**3**	**….**	**n**
2	**4**	**6**	**….**	**2n**
3	**6**	**9**	**….**	**3n**
⋮	**⋮**	**⋮**		**⋮**
⋮	**⋮**	**⋮**		**⋮**
n	**2n**	**3n**	**….**	**n^2**

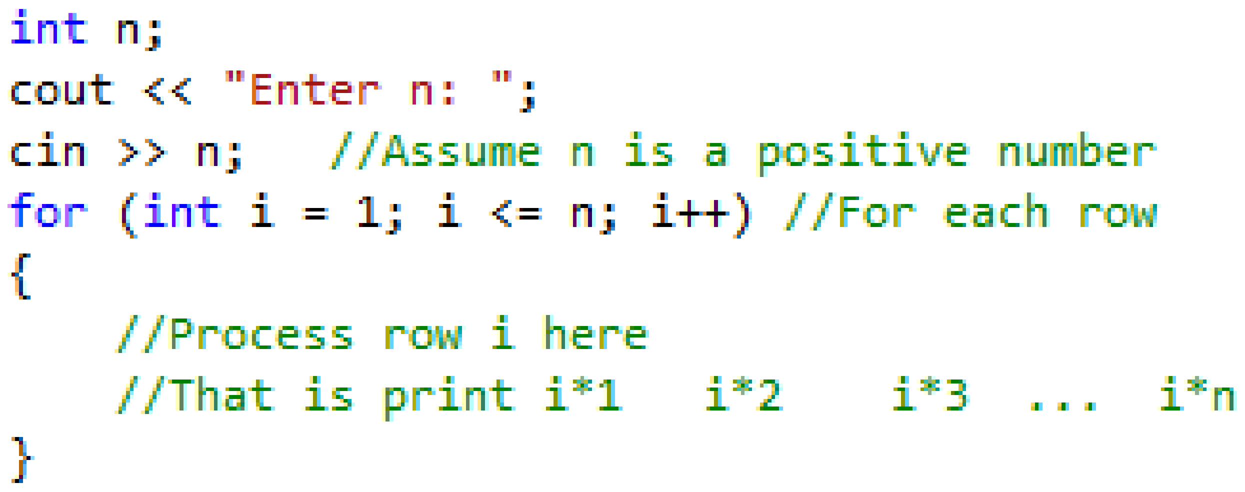
- How can we implement a C++ program to display such a multiplication table for any given value of n?

Nested Loops: Motivation

- We see that this is an example of a two-dimensional problem and can not be solved with the single loop structures we have seen so far. Why?
- This particular problem requires a loop to process the rows; but also each row requires a loop to process its columns
- We say we need a loop inside a loop
- Since any valid C++ code can be put inside a loop block, we can perfectly put a loop inside a loop
- Loops inside loops are known as ***Nested Loops***

Nested Loops

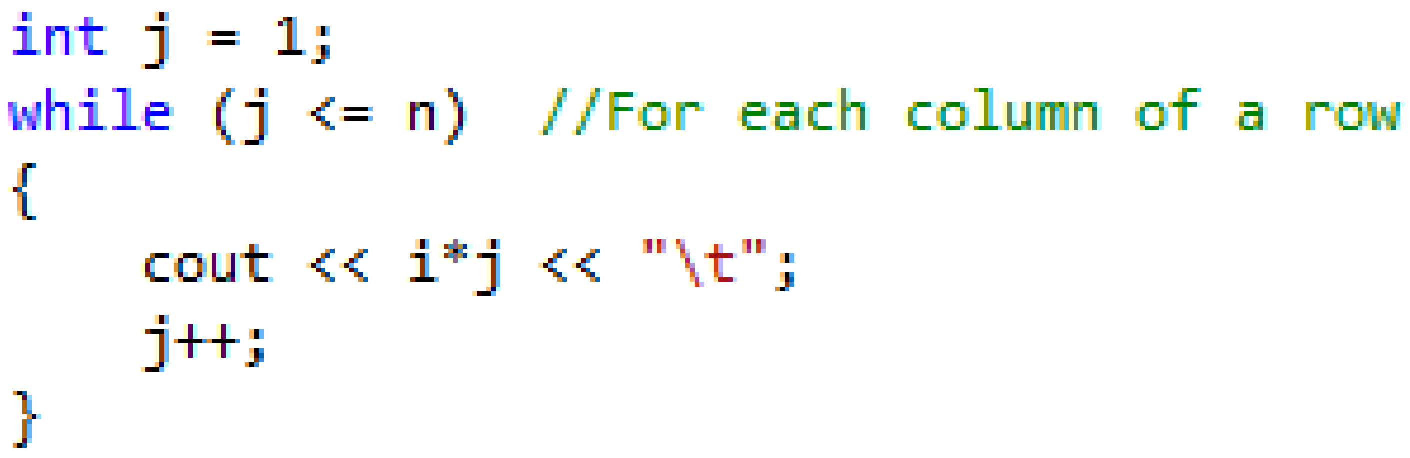
- In order to implement a C++ program for the multiplication table, we start by taking input value for n and then have a loop for the rows as follows:



```
int n;
cout << "Enter n: ";
cin >> n;   //Assume n is a positive number
for (int i = 1; i <= n; i++) //For each row
{
    //Process row i here
    //That is print i*1   i*2    i*3   ...   i*n
}
```

Nested Loops

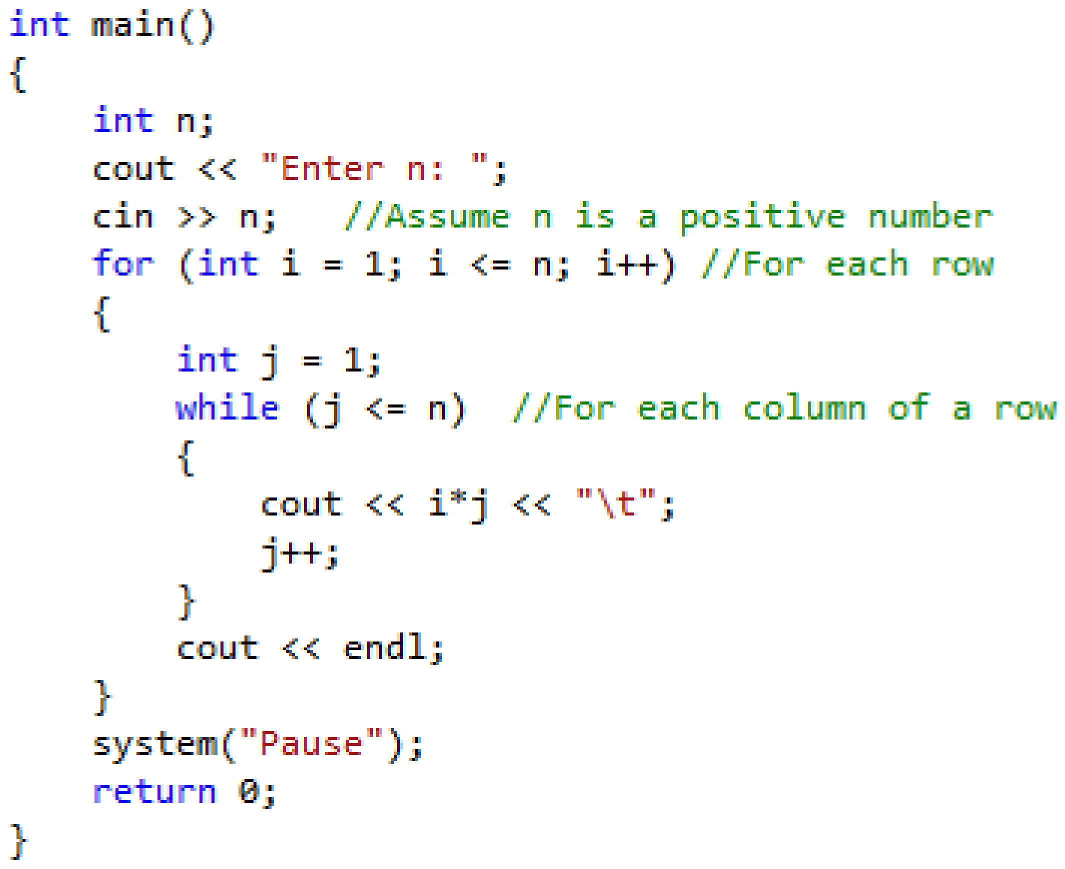
- Now for each row (that is inside the for loop block), we would like to print the row at index **`i`** which is given by **`i*1, i*2, i*3, … i*n`**
- That is we would like to cout **n** times: which is a loop on its own
- Therefore this loop will look like:



```
int j = 1;
while (j <= n)  //For each column of a row
{
    cout << i*j << "\t";
    j++;
}
```

Nested Loops

- Putting the results together, the multiplication table can be computed as follows:



```
int main()
{
    int n;
    cout << "Enter n: ";
    cin >> n;   //Assume n is a positive number
    for (int i = 1; i <= n; i++) //For each row
    {
        int j = 1;
        while (j <= n)  //For each column of a row
        {
            cout << i*j << "\t";
            j++;
        }
        cout << endl;
    }
    system("Pause");
    return 0;
}
```

Nested Loops: Syntax

- Any loop structure can be put inside another loop structure. The general syntax is therefore:

```
        for , while or do-while loop  ← outer loop
        {
                    Some C++ statements
inner loop → for , while or do-while loop
                    {
                              Some C++ statements
                    }
                    Some C++ statements
        }
```

Practice Question

- Write a C++ program that prints an **n*m** multiplication table for any positive values of m and n; as shown below:

1	**2**	**3**	**….**	**m**
2	**4**	**6**	**….**	**2m**
3	**6**	**9**	**….**	**3m**
⋮	**⋮**	**⋮**		**⋮**
⋮	**⋮**	**⋮**		**⋮**
n	**2n**	**3n**	**….**	**nm**

Practice Question

- Write a C++ program that prints half multiplication table for any positive value of n; as shown below:

```
1
2      4
3      6      9
⋮      ⋮      ⋮
⋮      ⋮      ⋮
n      2n     3n     ….     n²
```