Repetition

Content Overview

- Repetition (looping) Control Structures
- The **`while-do`** loop
  - Sentinel-controlled structures
  - Count-controlled structures
  - Flag-controlled structures
- **`break`** and **`continue`** statements
- The **`for`** loop
- The do-while loop
- A use for comma expressions
- Variable Scope

Why Do We Need Repetition?

Sometimes the same set of instructions must be executed several times.

Retyping the same set of instructions in a program is impractical and we would always need to know the exact number of times the set of instructions was to be repeated.

For these reasons we need *repetition control structures.*

Why Do We Need Repetition?

For example, we can add five numbers together by:

- declaring a variable for each number, inputting the numbers and adding the variables together

Why Do We Need Repetition?

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

int main()
{
      int num1, num2, num3, num4, num5, sum;

      cin >> num1 >> num2 >> num3 >> num4 >> num5;
      sum = num1 + num2 + num3 + num4 + num5;
      cout << "sum = " << sum << endl;

      return 0;
}
```

Why Do We Need Repetition?

A much better alternative is:

To construct a repetition structure that reads a number into a variable and adds it to the variable that contains the sum of the numbers and repeat this procedure until all numbers are read

```
1.int num, sum = 0;
2.cin >> num;
3.sum = sum + num;
```

Repeat statements 2 and 3 for each number

Why Do We Need Repetition?

```
BEGIN add 5 numbers together
    sum = 0
    REPEAT 5 times
            INPUT a_number
            sum = sum + a_number
    ENDREPEAT
    PRINT  sum
END add 5 numbers together
```

Why Do We Need Repetition?

An example of pseudocode to determine and print out which of N numbers is the largest:

```
BEGIN print the largest of N numbers
        INPUT a_number
        largest_number = a_number
        REPEAT until there are no more numbers left
                INPUT a_number
                IF a_number is larger than largest_number THEN
                        largest_number = a_number
                ENDIF
        ENDREPEAT
        PRINT  largest_number
END print the largest of N numbers
```

C++ Repetition Statements

C++ has three repetition structures

- while loop
- for loop
- do-while loop

The while-do Loop

The general form of the while-do statement is:

```
while(expression)
    statement
```

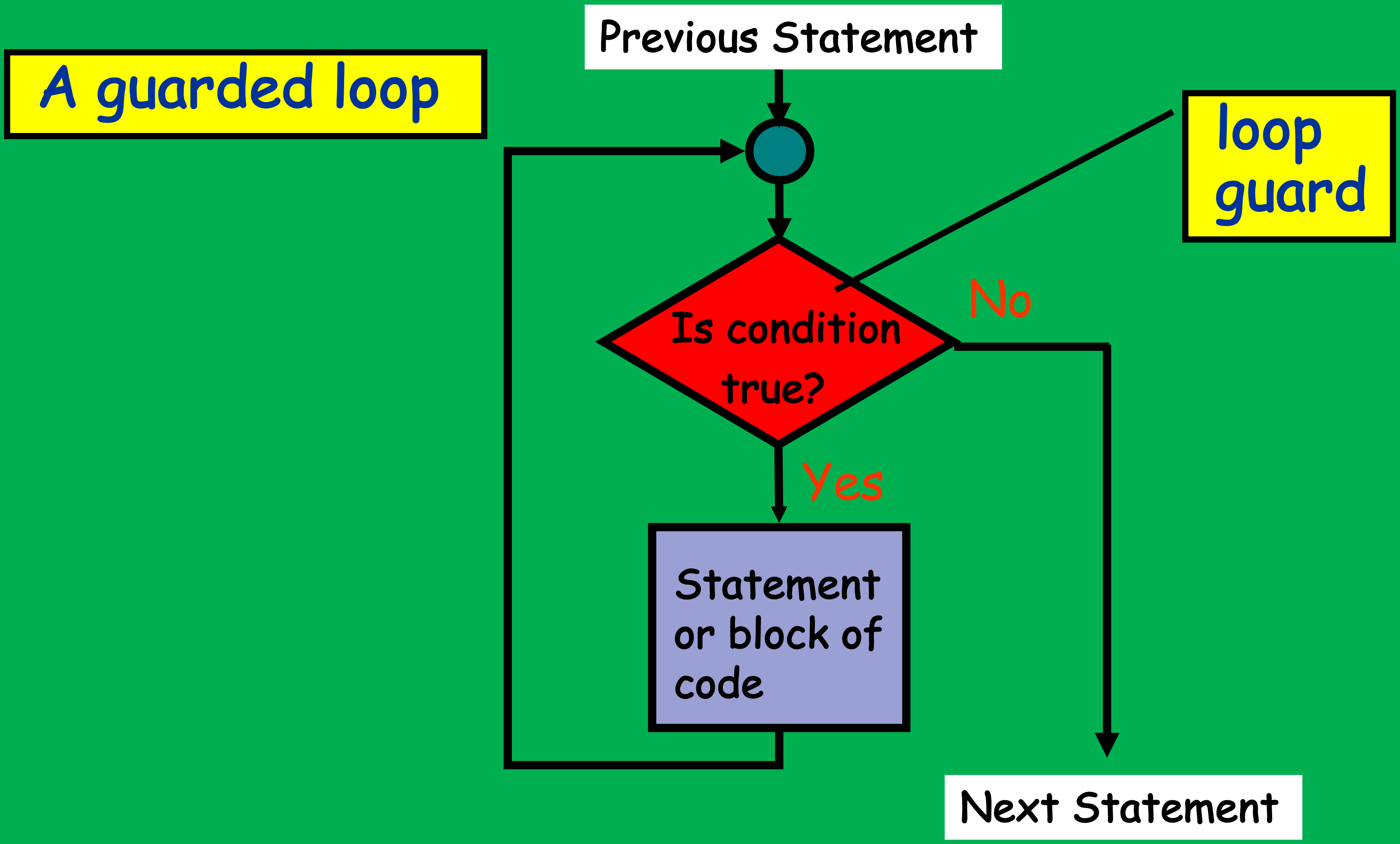
while is a reserved word

The statement can be simple or compound

The expression acts as a decision maker and is a logical expression

The statement is called the body of the loop

The `while-do` Loop Flowchart



A guarded loop

Previous Statement

loop guard

Is condition true?

No

Yes

Statement or block of code

Next Statement

The Guarded while-do Loop

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

int main()
{
      int sum = 0;
      while (sum <= 10)
      {
            sum = sum + 2;
            cout << sum << " ";
      }


      return 0;
}
```

ANSWER: 2 4 6 8 10 12

The Guarded `while-do` Loop

The loop guard (condition) is tested first to see whether one execution of the loop's statement should be allowed.

If the condition is `true` the statements in the body of the loop are executed, control is passed back to the start of the loop and the test carried out again.

The Guarded `while-do` Loop

If the condition is **`false`**, then the statements in the body of the loop are not executed.

This guarded loop is also known as

a sentinel-controlled while loop

Sentinels

In computer programming, data values used to signal either the start or end of a data series are called sentinels.

The sentinel value must be selected so that it will not conflict with legitimate data values.

'Print the Largest Number 'Revisited

```
BEGIN print the largest of N numbers
    INPUT a_number
    largest_number = a_number
    REPEAT until there are no more numbers left
        INPUT a_number
        IF a_number is larger than largest_number THEN
                          largest_number = a_number
        ENDIF
    ENDREPEAT
    PRINT  largest_number
END print the largest of N numbers
```

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

int main()
{
   int num, largest;

   cout <<"Enter numbers "
        <<"ending with -1: ";
   cin >> num;
   largest = num;
   while (num != -1)
   {
      cin >> num;
      if (num > largest)
           largest = num;
   }
   cout << largest;

   return 0;
}
```

Using a
sentinel-controlled
while loop

A better version

```
BEGIN print the largest of N numbers
    INPUT a_number
    largest_number = a_number
    REPEAT until there are no more numbers left
        INPUT a_number
        IF a_number is larger than largest_number THEN
                      largest_number = a_number
        ENDIF
    ENDREPEAT
    PRINT  largest_number
END pri
```

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

int main()
{
   int num, largest=-1;

   cout <<"Enter numbers "
        <<"ending with -1: ";
   cin >> num;
   while (num != -1)
   {
       if(num > largest)
           largest = num;
       cin >> num;
   }
   cout << largest;

   return 0;
}
```

This code is better because the sentinel is never compared with the number

The Average of N Positive Numbers

```
BEGIN average of N positive numbers
    count = 0;
    sum = 0;
    INPUT a_number
    REPEAT until number is -999
        sum = sum + a_number
        count = count + 1
        INPUT a_number
    ENDREPEAT
    PRINT sum/count
END average of N positive numbers
```

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

int main()
{
    int num, count=0, sum=0;

    cout <<"Enter numbers "
         <<"ending with -999: ";
    cin >> num;
    while (num != -999)
    {
        sum += num;
        count++;
        cin >> num;
    }
    cout << sum/count;

    return 0;
}
```

LOOK!
correct use of
the sentinel

The Average of N Positive Numbers

```
BEGIN average of N positive numbers
    count = 0;
    sum = 0;
    INPUT a_number
    REPEAT until number is -999
        sum = sum + a_number
        count = count + 1
        INPUT a_number
    ENDREPEAT
    PRINT sum/count
END average of N positive numbers
```

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

int main()
{
   int num, count=0, sum=0;

   cout <<"Enter numbers "
        <<"ending with -999: ";
   cin >> num;
   while (num != -999)
   {
       sum += num;
       count++;
       cin >> num;
   }
   cout << sum/count;

   return 0;
}
```

BUT
Wrong answer!

Integer division

The Guarded `while-do` Loop

Another type of guarded while loop is the *counter-controlled while loop.*

If we know exactly how many times the loop must be executed, the while loop might look like this:

```
while (counter < 50)
{
    .  .  .
    counter++;
    .  .  .
}
```

A Guarded while-do Loop Example

```
while (counter < limit)
{
      cin >> number;
      sum = sum + number;
      counter++;
}
```

Note that variables `counter`, `limit` and `sum` must be previously declared and initialised.

The Guarded `while-do` Loop

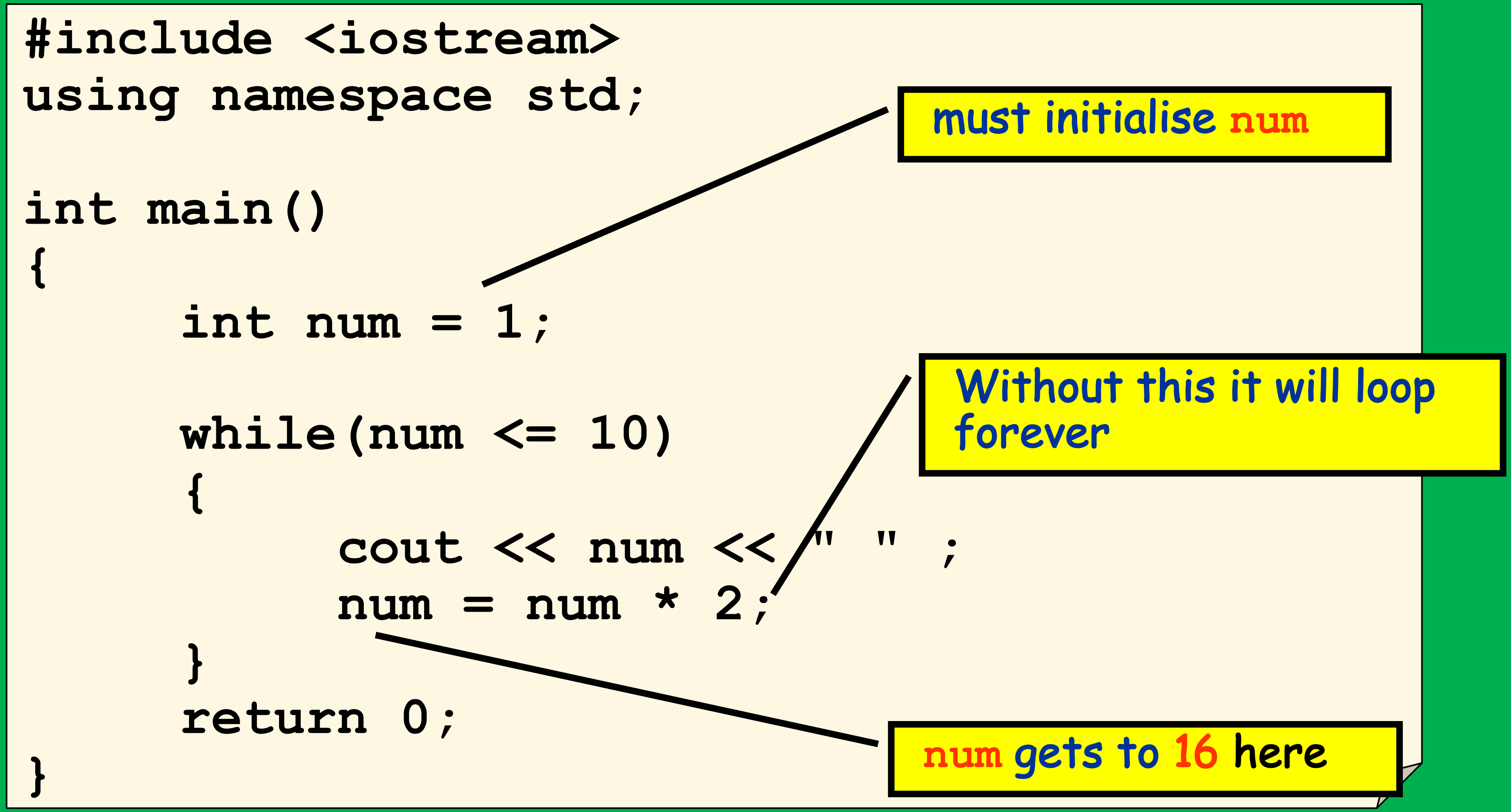
Note also that:

If initial the condition is false, the loop's statement is never executed.

One thing that must be provided in the body of the loop is a possibility for the guard condition to be changed.

If the guard condition is not changed the loop never terminates (infinite loop).

Example



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

int main()
{
      int num = 1;

      while(num <= 10)
      {
            cout << num << " " ;
            num = num * 2;
      }
      return 0;
}
```

must initialise num

Without this it will loop forever

num gets to 16 here

Output:

```
1 2 4 8
```

Example

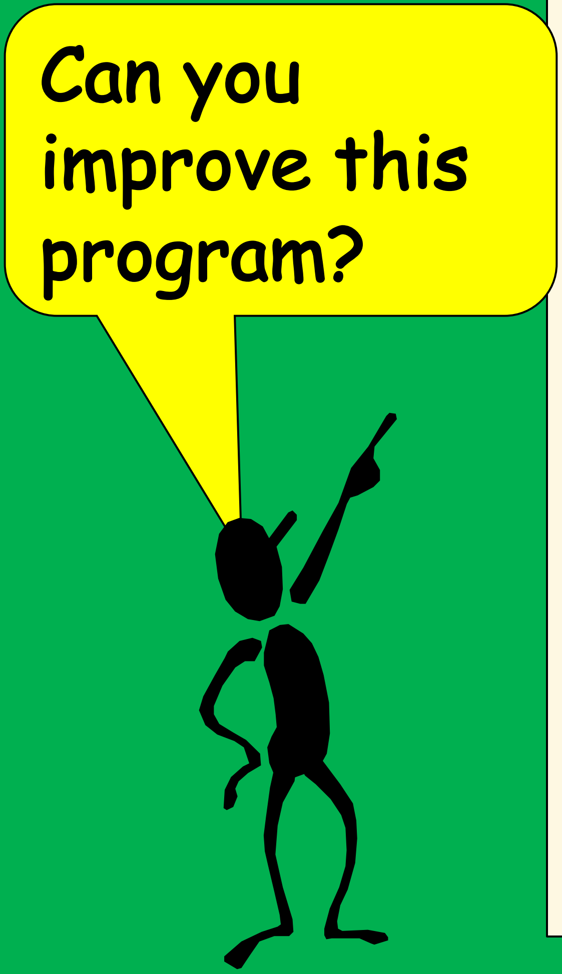
What is the output of this program?

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

int main()
{
        int count = 1;

        while(count <= 7)
        {
                cout << count << " ";
                count++;
        }

        return 0;
}
```

Answer:

```
1 2 3 4 5 6 7
```

Example



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

int main()
{
        int count = 1;

        while(count <= 7)
        {
                cout << count << " ";
                count++;
        }

        return 0;
}
```

Example

What is the output of this program?

Answer:

```
1 2 3 4 5 6 7
```

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

int main()
{
      int count = 1;

      while(count <= 7)
      {
            cout << count++ << " ";
      }

      return 0;
}
```

Example

What is the output of this program if I enter 1 2 3 -1?

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

int main()
{
    int totalMarks = 0, marks = 0;
    while(marks >= 0)
    {
        cout << "Enter a mark (-1 to end): ";
        cin >> marks;
        totalMarks += (marks >= 0 ? marks : 0);
    }
    cout << "Total Marks: " << totalMarks
        << endl;
    return 0;
}
```

Answer:

```
Total Marks: 6
```

Example

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

int main()
{
    int totalMarks = 0, marks = 0;
    while(marks >= 0)
    {
        cout << "Enter a mark (-1 to end): ";
        cin >> marks;
        totalMarks += (marks >= 0 ? marks : 0);
    }
    cout << "Total Marks: " << totalMarks
        << endl;
    return 0;
}
```

Example

What is the output of this program if I enter 4?

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

int main()
{
      int num, result = 1;

      cout << "Enter a number: ";
      cin >> num;

      while(num > 0)
      {
            result = result * num--;
      }
      cout << "result is: " << result
            << endl;
      return 0;
}
```

Answer:

```
result is 24
```

Example

Wh
out
pro
ente

No!!!
BIG
MISTAKE
result is
multiplied
by 0

Answer:

```
result is 0
```

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

int main()
{
        int num, result = 1;

        cout << "Enter a number: ";
        cin >> num;

        while(num > 0)
        {
                result = result * --num;
        }
        cout << "result is: " << result
                << endl;
        return 0;
}
```

The Guarded while-do Loop

Another type of guarded while loop is the *flag-controlled while loop.*

It uses a Boolean variable to control the while loop. The loop might look like this:

```
while (!found)
{
        . . .
        if(expression)
             found = true;
        . . .
}
```

This variable is called a flag variable

Example (revisited)

What happens if by mistake I put a ; here?

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

int main()
{
        int num = 1;

        while(num <= 10)
        {
                cout << "num = " << num
                         << " " ;
                num = num * 2;
        }

        return 0;
}
```

Example (revisited)

No output produced, the program hangs…

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

int main()

        int num = 1;

        while(num <= 10);
        {
                cout << "num = " << num
                         << " " ;
                num = num * 2;
        }

        return 0;
}
```

while-do Statement Summary

The *while-do* statement is a *pre-test loop;* the loop guard is tested first.

It uses an expression to control the loop.

No semicolon is required at the end of the *while-do* statement.

If we want to include multiple statements in the body, we must put them in a compound statement.

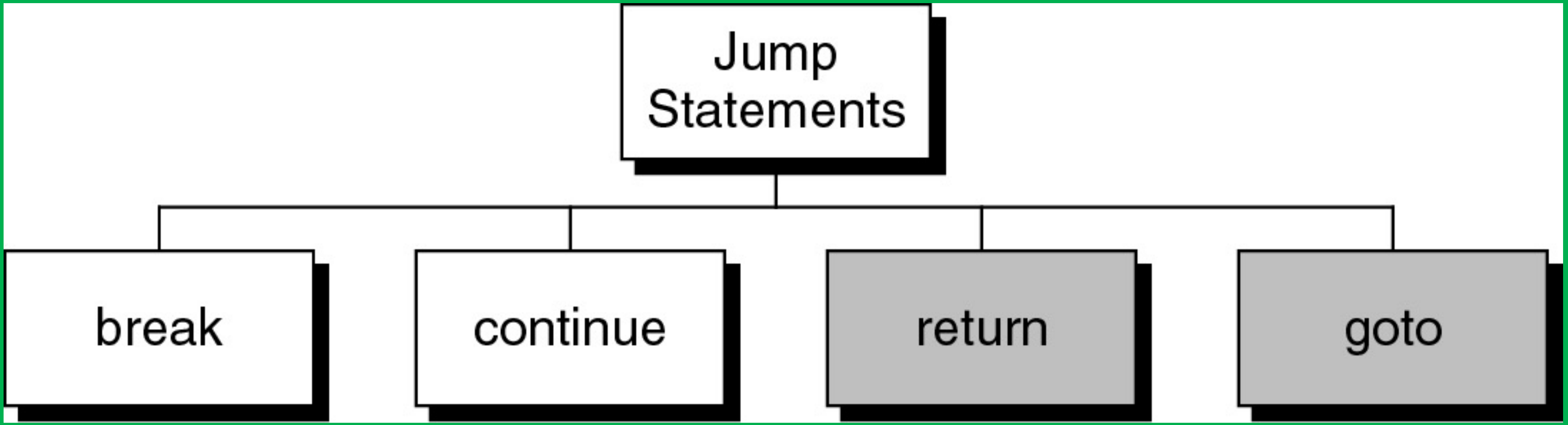
Example (revisited again)

What is the output of this program?

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

int main()
{
        int num = 1;
        while(num <= 10)
                cout << num << " " ;
                num = num * 2;

        return 0;
}
```

Jump Statements

Jump Statements allow the flow of the program to “jump” from its normal flow.



Jump Statements

We will discuss the `return` statement later when we discuss functions.

The `goto` statement is UNSUITABLE for structured programs. Therefore, we won't even discuss it here.

`break` Statement

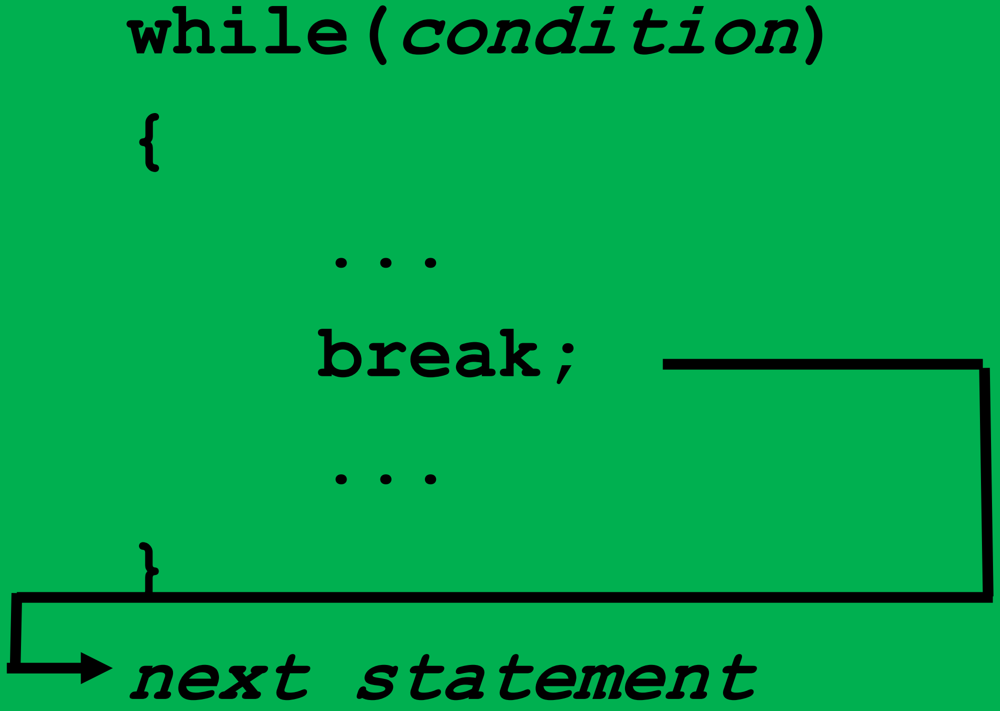
We used the **`break`** statement inside the **`switch`** statement.

The **`break`** statement can also be used to terminate a loop – but should not be.

In a series of nested loops, **`break`** only terminates the inner loop - the one the program is currently in.

The **`break`** statement needs a semicolon.

break Statement



```
while(condition)
{
      ...
      break;
      ...
}
next statement
```

Example using break

```
#include <iostream>
using namespace std;
int main()
{
   int totalMarks = 0, marks = 0;
   while (true)
   {
        cout << "Enter a mark (-1 to end): ";
        cin >> marks;
        if (marks < 0)
               break;
        totalMarks += marks;
   }
   cout << "Total Marks: " << totalMarks << endl;
   return 0;
}
```

The loop terminates when `marks < 0`. That's the purpose of *break*

Example using **`break`**

Using *break* is NOT a good structured programming style.

You can always rewrite your code WITHOUT the break statement.

NEVER USE **`break`** to exit a loop in this subject!!!

Using `break` - Rules of Thumb

Limit the use of `break` to `switch` statements.

Although `break` statements are valid in all loop structures, it is not regarded as good structured programming style.

Instead of using `break` statements inside loops, try to redesign your program.

`continue` Statement

The `continue` statement does not terminate the loop, but it transfers execution to the testing expression.

In a *pre-test loop*, it is similar to a jump to the beginning of the loop.

The `continue` statement is also considered to be unstructured programming.

Try to avoid using it at all.

Example using continue

```
#include <iostream>
using namespace std;
int main()
{
   int totalEven = 0, number = 0;
   while (number >= 0)
   {
       cout << "Number (-1 to exit) = ";
       cin >> number;
       if (number % 2 != 0)
             continue;
       totalEven += number;
   }
   cout << "Total Even Numbers = " << totalEven << endl;
   return 0;
}
```

Example

```
//While loop with break *** IN THIS SUBJECT NEVER USE BREAK INSIDE A LOOP ***
#include <iostream>
using namespace std;

int main()
{
        int num, sum;
        sum = 0;

        cout << "Enter numbers: ";
        cin >> num;

        while (cin)
        {
                if (num < 0)    //if number is negative, terminate the loop
                {
                     cout << "Negative number found in the data" << endl;
                     break;
                }
                sum = sum + num;
                cin >> num;
        }
        cout << endl;
        cout << "The sum is: " << sum << endl;

        return 0;
}
```

What's this ?
This is not logical.

Example

```
//While loop with break *** IN THIS SUBJECT NEVER USE BREAK INSIDE A LOOP ***
#include <iostream>
using namespace std;

int main()
{
        int num, sum;
        sum = 0;

        cout << "Enter numbers: ";
        cin >> num;

        while (!cin.eof())
        {
                if (num < 0)    //if number is negative, terminate the loop
                {
                     cout << "Negative number found in the data" << endl;
                     break;
                }
                sum = sum + num;
                cin >> num;
        }
        cout << endl;
        cout << "The sum is: " << sum << endl;

        return 0;
}
```

Example

```
//While loop with break *** IN THIS SUBJECT NEVER USE BREAK INSIDE A LOOP ***
#include <iostream>
using namespace std;

int main()
{
        int num, sum;
        sum = 0;

        cout << "Enter numbers: ";
        cin >> num;

        while (!cin.eof() && num >=0)
        {
                sum = sum + num;
                cin >> num;
        }
        if (num < 0)    //if number is negative, loop terminated
                cout << "Negative number found in the data" << endl;

        cout << endl;
        cout << "The sum is: " << sum << endl;

        return 0;
}
```

The `for` Loop

The `for` loop is a *pre-test loop* that uses three expressions: the *initialisation expression*, the *conditional expression* and the *updating expression*

The format of the `for` statement is

```
for(init_expr; test_expr; update_expr)
    statement;
```

The for Loop

This loop structure

```
    for(init_expr; test_expr; update_expr)
      statement;
```

is equivalent to

```
init_expr;
   while(test_expr)
{
      statement;
      update_expr;
}
```

The `for` Loop

Thus the ***`init_expr`*** is executed when the **`for`** loop *starts* and then the ***`test_expr`*** is evaluated.

If the ***`test_expr`*** is non-zero the statement is executed followed by the ***`update_expr`*** (executed at the *end* of each loop).

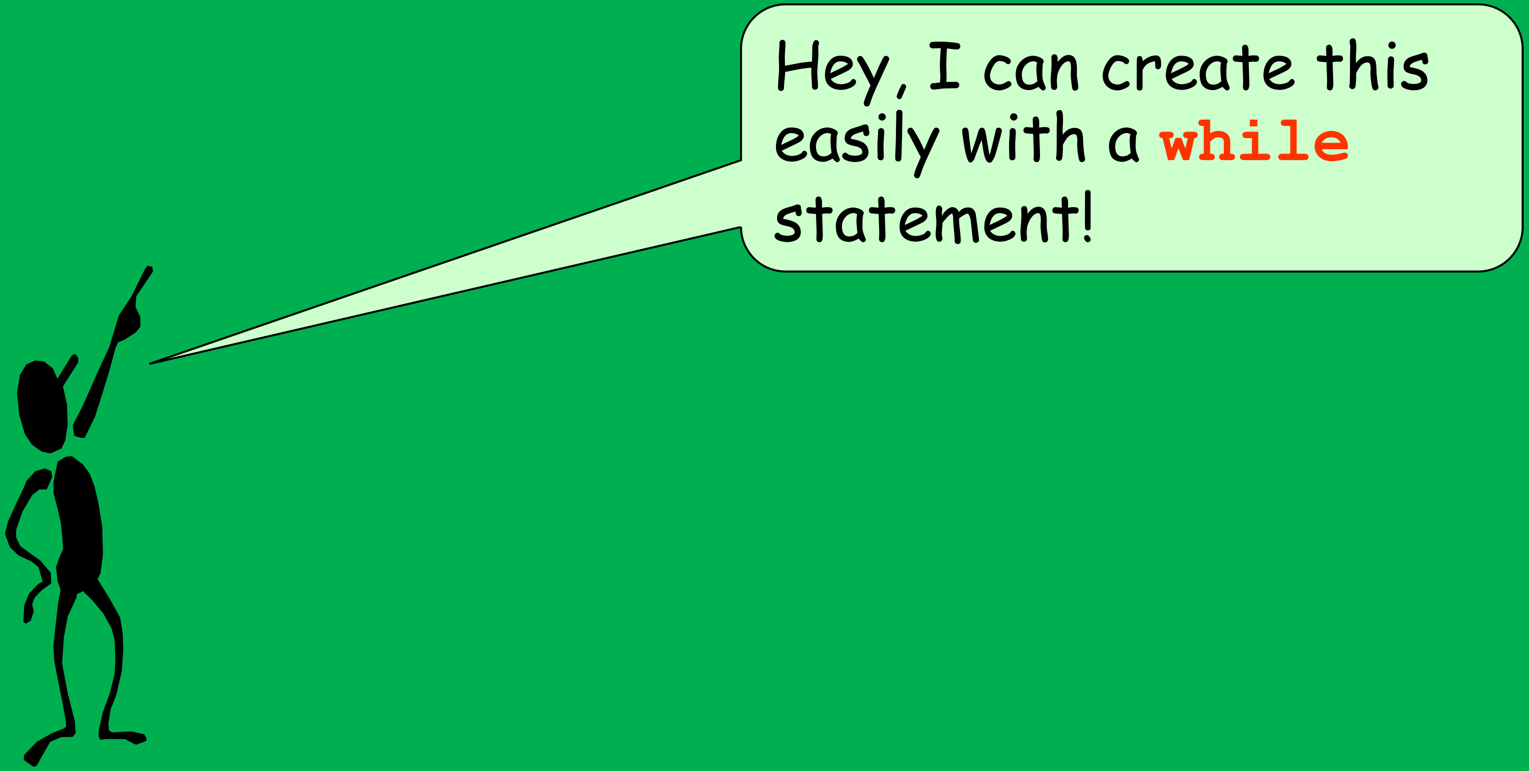
The ***`test_expr`*** is evaluated *before* every loop starts.

Example

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

int main()
{
   int count;

   for (count = 0; count < 10; count++)
      cout << count << " ";

   return 0;
}
```

Output of Example 5

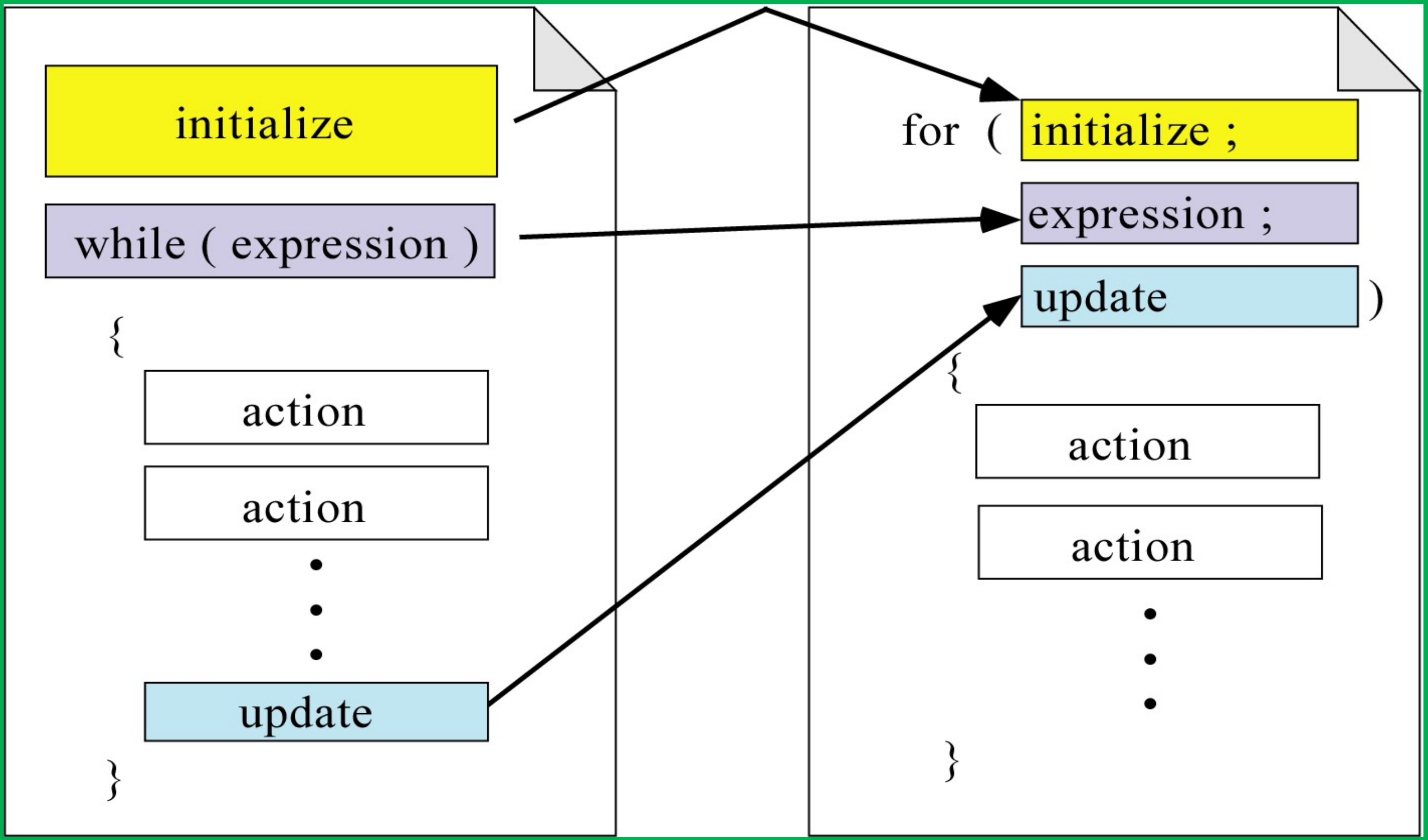
```
0  1  2  3  4  5  6  7  8  9
```



Creating the code with while

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


int main()
{
        int count=0;
        while (count < 10)
        {
          cout << count << " ";
          count++;
        }
        return 0;
}
```

Representing for with while



Example: Printing Even Numbers

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

int main()
{
    int count;

    for (count = 0; count < 10; count += 2)
        cout << count << " ";

    return 0;
}
```

for loops should always contain an index variable

```
0 2 4 6 8
```

Example: Summing 20 Numbers

```
int i = 1;
int num;
int sum = 0;
while (i<=20)
{
    cin >> num;
    sum += num;
    i++;
}
```

```
 int i, num;
 int sum = 0;
 for(i=1; i<=20; i++)
 {
     cin >> num;
     sum += num;
}
```

Example: Summing 20 Numbers

```
int num;
int sum = 0;
for(int i=1; i<=20; i++)
{
      cin >> num;
      sum += num;
}
```

LOOK! index variable i can also be declared here

Example: A more likely C++ solution

```
int num;
int sum=0;

for (int i=0; i<20; i++)
{
     cin >> num;
     sum += num;
}
```

Counting from 0 is much more common in C++

The do-while Loop

```
int num;
int i = 0, sum=0;

do
{
     cin >> num;
     sum += num;
     i++;
} while (i<20);
```

Notes on the `do-while` Loop

Note that the `do-while` ends with a semicolon. This is different to the other loops.

We use this loop when we want the body of the loop to be executed at least once.

A common application is when it is used for data validation.

What's wrong with this program?

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

int main()
{
        int num, sum;
        num = sum = 0;

        cout << "Number (-1 to end):";
        cin >> num;

        do
        {
                sum += num;
                cout << "Number (-1 to end):";
                cin >> num;
        } while (num >= 0);
        cout << "Sum= " << sum << endl;

        return 0;
}
```

Data consisting of just the sentinel is processed

Example: Data Validation

```
do
{
   cout << "Enter an identification number: ";
   cin >> idNum;
} while (idNum < 1000 || idNum > 1999);
```

The `do-while` loop is useful when it does not make sense to check a condition until after the action occurs.

The `for` Loop (again)

As we discussed earlier:

The ***`init_expr`*** is evaluated when the `for` loop starts and the ***`test_expr`*** is evaluated.

If the ***`test_expr`*** is non-zero the statement is executed followed by the ***`update_expr`*** (executed at the end of each loop)

The ***`test_expr`*** is evaluated before every loop.

Example

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

int main()
{
    int count;

    for (count = 0; count < 10; count++)
        cout << count << " ";

    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Notes on `the expressions`

C++ allows the initialisation expression (***`init_expr`***) to be empty.

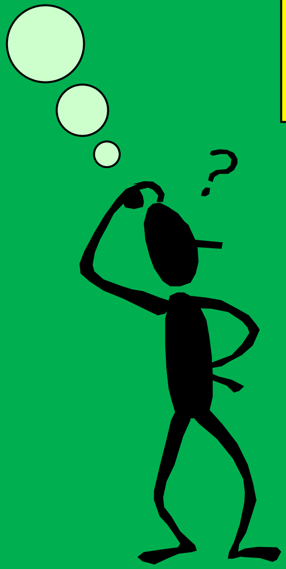
C++ also allows the loop control expression (***`test_expr`***) to be controlled inside the body of the **`for`** statement itself.

The updating expression (***`update_expr`***) can be null and the updating can also be done inside the body of the `for` statement.

None of this is recommended!

What does it mean?

init_expr, *test_expr* and *update_expr* can be empty. What does it mean?



Let's see an example for each case

Example: *init_expr* is omitted

```
#include <iostream>
using namespace std;
int main()
{
    int count=0;

    for (; count < 10; count += 2)
        cout << count << " ";

    return 0;
}
```

The `init_expr` can be omitted because `count` has been set to 0 in the initialisation statement

Example: *update_expr* is omitted

```
#include <iostream>
using namespace std;
int main()
{
    int count=0;

    for (; count < 10;)
    {
        cout << count << " "
        count += 2;
    }
    return 0;
}
```

update_expr is omitted and it is moved to the body of the loop

Example: *test_expr* is omitted

```
#include <iostream>
using namespace std;
int main()
{
    int count=0;

    for (;;)
    {
        if (count == 10) break;
        cout << count << " ";
        count += 2;
    }
    return 0;
}
```

If `test_expr` is omitted, then the condition is **always true**. It's the same with `while(true)`, the loop will never terminate unless, we use `break`. This is NOT a good practice!

So what's the point of a `for` loop ?

Use only when you want to count.

Do not leave out any of the three components

- use a while if you want to do that

Never change the index variable within the loop.

The reader should be able to look at the `for` loop's header and determine how many times the loop will execute.

No break.

Example: The *comma* (,) operator

Consider the following program:

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

int main()
{
     int count;

     for (count = 0; count < 10; count += 2)
          cout << count << " ";

     return 0;
}
```

How many times does the test get performed?

Example: The *comma* (,) operator

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

int main()
{
    int count, ttimes = 0;

    for (count = 0; ttimes++, count < 10;
        count += 2);
        cout << count << " ";
    cout << "Tested " << ttimes << " times\n";

    return 0;
}
```

Note that this is only ONE expression!

The *comma* operator

In general, using operator *comma* (,) is also not good practice.

Although, it might not be good style, sometimes you will see code that uses this style, and you should understand it.

But do not use it yourself.

The *comma* expression

A *comma expression* is a complex expression made up of two or more expressions separated by commas.

It is generally used in `for` statements and in declaration statements.

The *comma* expression

The expressions are evaluated **from left to right** and the comma has the lowest precedence of all operators.

The value and type of the expression is that of the right hand side expression.

Example

```
for (sum = 0, i = 1; i <= 20; i++)
{
    cin >> a;
}
```

is equivalent to:

```
sum = 0;
for (i = 1; i <= 20; i++)
{
    cin >> a;
}
```

Example: while-do and do-while

```
while loop: 12345678910
Loop Count: 11
Number of tests: 11
```

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

int main
{
    int loopCount=1, testCount=0;

    cout << "while loop: ";

    while (testCount++, loopCount <= 10)
        cout << loopCount++;

    cout << "\nLoop Count: " << loopCount << endl;
    cout << "Number of tests: " << testCount
           << endl;
    return 0;
}
```

Example: while-do and do-while

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

int main
{
    int loopCount=1, testCount=0;

    cout << "do..while: ";

    do
        cout << loopCount++;
    while (testCount++, loopCount <= 10)
    cout << "\nLoop Count: " << loopCount << endl;
    cout << "Number of tests: " << testCount
           << endl;
    return 0;
}
```

```
do..while: 12345678910
Loop Count: 11
Number of tests: 10
```

Notes on Example

Both the **`while`** and the **`do-while`** loops contain a comma expression.

Because the value of the whole comma expression is the value of the last expression, the limit test expression (in this example **`loopCount <= 10`**) must be coded last.

Notes on Example

Both the loops count from one to ten, but the loop expression was tested 11 times in the *while* loop and only 10 times in the *do…while* loop.

In a pre-test loop, the test is done n + 1 times.

In a post-test loop, the test is done n times.

Nested Loops

In many situations, it is convenient to use a loop contained within another loop.

Such loops are called nested loops.

Example

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

int main()
{
    int i, j;

    for (i=0; i < 3; i++)
        for (j=3; j >= 0; j--)
            cout << i << ", " << j << endl;

    return 0;
}
```

```
0, 3
0, 2
0, 1
0, 0
```

```
1, 3
1, 2
1, 1
1, 0
```

```
2, 3
2, 2
2, 1
2, 0
```

Variable Scope

A variable that is defined in a loop is only valid in the loop's scope. This variable is not known outside that scope.

```
for (int i=0; i < 5; i++)
      cout << i << endl;

cout << i;
```

This example has two possible outcomes.

If i is not defined before the loop, this generates a compiler error, because the variable i is unknown the loop.

Variable Scope

A variable that is defined in a loop is only valid in the loop's scope. This variable is not known outside that scope.

```
for (int i=0; i < 5; i++)
     cout << i << endl;

cout << i;
```

This example has two possible outcomes.

If i has been defined before the loop, the value printed is the value of i as it was before the loop.

Write a C++ Program

An integer is divisible by 9 if the sum of its digits is divisible by 9.

Write a program that prompts the user to input an integer.

The program should then output the number and a message stating whether the number is divisible by 9.

(*** cannot use % ***)

Develop a C++ Program

Design a program that prints five spreadsheet-style column titles with the values A, B, C, D, and E, and five row titles with the values 1, 2, 3, 4, 5. Use a for loop to output the column headings. Use nested for loops to output the row number and the values within the rows and columns. The output should look like the following.

```
     A    B    C    D    E
1    1    2    3    4    5
2    1    2    3    4    5
3    1    2    3    4    5
4    1    2    3    4    5
5    1    2    3    4    5
```

Program Design

Write a for loop to output the letters A through E with a tab before each letter.

```
Begin a for loop to output 5 rows
    Output the number of the row
    Begin a for loop to output 5 columns
        Output the numbers 1 through 5 with a tab before each number
    End the inner loop
    Output a newline
End the outer loop
```

Program Code

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

int main ()
{
      for (char title = 'A'; title < 'F'; title++)
            cout << '\t' << title;
      cout << endl;

      for (int outer = 1; outer < 6; outer++)
      {
            cout << outer;
            for (int inner = 1; inner < 6; inner++)
                  cout << '\t' << inner;
            cout << endl;
      }
      return 0;
}
```