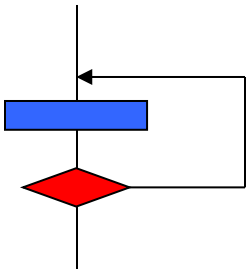
Repetition − many problems require an action or a set of actions to be repeated; this requirement can be accomplished in two different ways: using loops or using recursion.

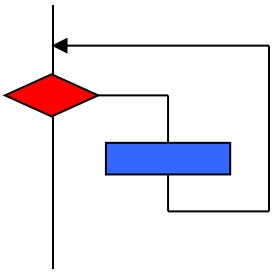
Loop – a statement that controls the execution of an action (or a series of actions) that is continually repeated as long as a condition is reached.

Iteration – a single execution of the action(s) in a loop.

Posttest Loop – test after action(s)



Pretest Loop – test before action(s)



Counter-Controlled Loop – the number of iterations is known in advance (repeat "this" n times)

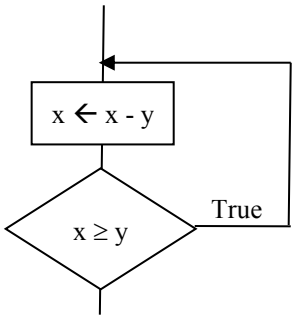
Event-Controlled Loop – the number of iterations is not known in advance (repeat "this" as long as some condition is true)

Loops in C++ – there are three loop statements:

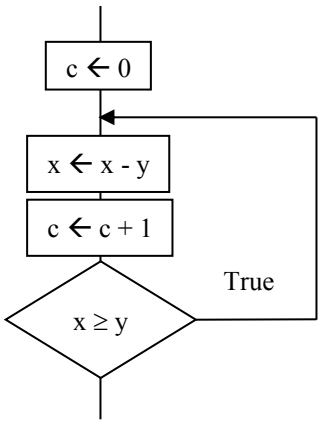
do …while
while
for

All of them can be used for counter-controlled and event-controlled loops, but the **for** loop is most naturally used for counter-controlled loops.

do …while *// posttest loop - action(s) followed by test*

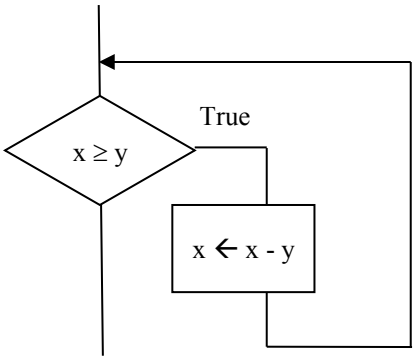



```
do
    x = x - y;
while ( x >= y );
```

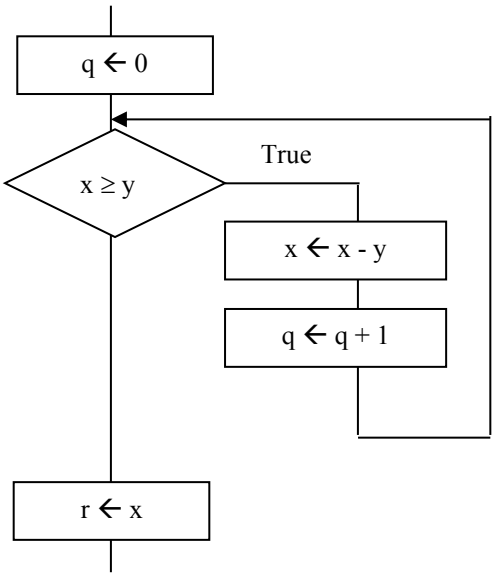



```
c = 0;
do
{
    x = x - y;
    c = c + 1;
}while ( x >= y );
```

while *// pretest loop - test followed by action(s)*



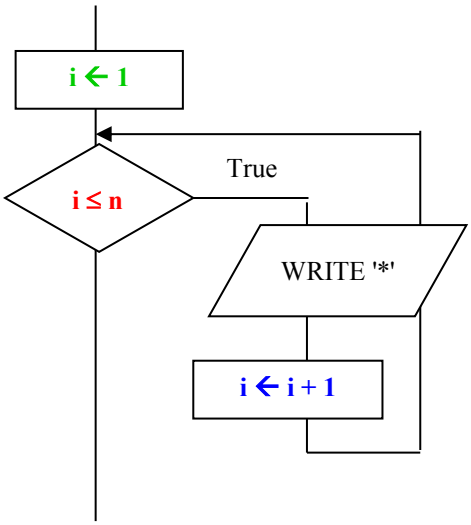

```
while ( x >= y )
   x = x - y;
```




```
q = 0;
while ( x >= 0 )
{
    x = x - y;
    q = q + 1;
}
r = x;
```

for *// pretest loop pretest loop - test followed by action(s)*

```
i = 1;
while ( i <= n )
{
    printf("*\n");
    i++;
}
```




```
for ( i = 1; i <= n ; i++ )
    printf("*");
```

initialize
check - true
action(s)
update
check - true
action(s)
update
.
.
.
check - false

A pretest loop in C++ can be written either as a while loop or as a for loop;
while is most naturally used for event-controlled loops, and
for for counter-controlled loops (*it keeps the loop control statements close together).*

```
for (expr1; expr2; expr3)
     statement
```

The three components of the **for** loop are expressions. Most often `expr1` *(initialize)* and `expr3` *(update)* are assignments, and the `expr3` *(check)* is a relational expression. Any of the three expressions may be omitted, but the semicolons must remain.

```
for (     ; n > 0; n-- )

for (i = 1; i <= n;     )

for (  ;  ;  )   // forever true
```

Comma operator – ',' a binary operator that allows two expressions to be treated as one. It is most often used in for statements.

```
sum = 0;
for ( i = 1; i <= n; i++ )
    sum += i;
```

```
for (sum = 0, i = 1; i <= n; i++ )
    sum += i;
```

Comma expression – an expression created using the comma operator:

```
sum = 0, i = 1
```

The expressions are evaluated from left to right; the comma expression has as value and type the value and the type of its right expression. The commas in a parameter list or in a variable declaration list (etc.) are not comma operators.

Nested loops – a loop within a loop. Each iteration of the outer loop triggers a complete execution of the inner loop.

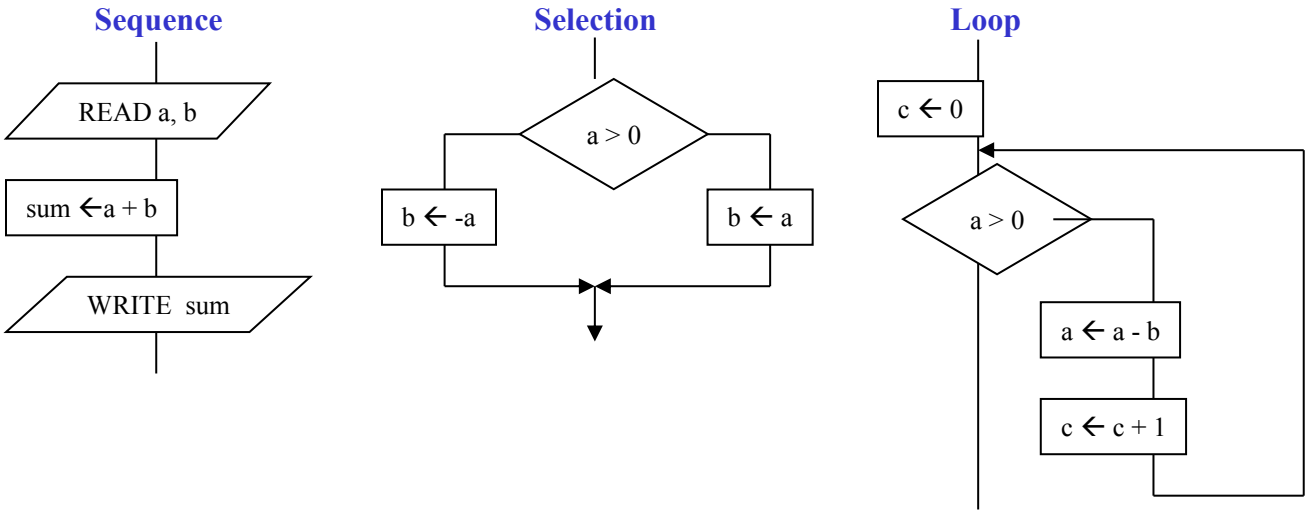
```
// display 3 rows of 5 asterisks each

for ( int r = 1; r <= 3; r++ )
{
     for ( int c = 1; c <= 5; c++ )
          cout << "*";
     cout << endl;
}
```

break – when used inside a loop, interrupts its execution and passes control to the first statement immediately after the loop.

continue – the statements after continue are abandoned for the current iteration and the execution continues with the next iteration

goto and **labels** – should be used rarely, if at all. The code that uses goto statements is harder to understand and to maintain than the code without gotos.

structured programming – one of the structured programming requirements is to write any algorithm using only three constructs: sequences, selections, and loops (without using break, continue, and goto). This theorem (known as the Structured Program Theorem) was demonstrated by two mathematicians: Corrado Bohm and Giuseppe Jacopini, in 1966. In 1968 Edsger Dijkstra wrote the article "Go To Statement Considered Harmful" that emphasized on the importance of structured programming.



common loop application

- **summation**
  (initialize sum to 0)

```
// 5 + 10 + 15 + 20 + 25 + 30
sum = 0;
for (i = 5; i <= 30; i += 5)
    sum += i;
```

- **product**
  (initialize product to 1)

```
// 5 * 10 * 15 * 20 * 25 * 30
prod = 1;
for (i = 5; i <= n; i += 5)
    prod *= i;
```

- **smallest / largest** *(initialize smallest to a very large value; initialize largest to a very small value)*

```
// n numbers: 8  10  9  12  5 7 20 3  25  5
smallest = INT_MAX;
for (int i = 1; i <= n; i++)
{
    cin >> num;
    if (num < smallest)
        smallest = num;
}
```

```
// n numbers: 70 4  9  12 80 7 20 90  5  7
largest = INT_MIN;
for (int i = 1; i <= n; i++)
{
    cin >> num;
    if (num > largest)
        largest = num;
}
```

- **inquiries: any / all** *(to answer an `any` inquiry initialize the result to `false`; to answer an `all` any inquiry initialize the result to `true`)*

```
// any odd?
result = 0; // false
for (i = 1; i <= n && !result; i++)
{
    cin >> num;
    if ( num % 2 )
        result = 1; // true
}
```

```
// all odd?
result = 1; // true
for (i = 1; i <= n && result; i++)
{
    cin >> num;
    if ( !(num % 2) )
        result = 0; // false
}
```

// **n** numbers: 8 10 6 12 **5** 7 20 3 25 5
// the loop stops at 5 – the first odd number: true

// **n** numbers: 7 9 5 11 **6** 7 20 3 25 5
// the loop stops at 6 – the first non-odd number: false

// **n** numbers: 8 10 6 12 2 4 20 4 24 6
// all numbers are tested: result is false (no odds)

// **n** numbers: 7 9 5 11 3 7 21 3 25 5
// all numbers are tested, all are odd: result is true.