

Iteration causing a set of statements (the body) to be executed repeatedly.

C++ provides three control structures to support iteration (or looping).

Before considering specifics we define some general terms that apply to any iteration construct:

- |                       |   |
|-----------------------|---|
| pass (or iteration)   | - one complete execution of the body  |
| loop entry            | - the point where flow of control passes to the body  |
| loop test             | - the point at which the decision is made to (re)enter the body, or not                         |
| loop exit             | - the point at which the iteration ends and control passes to the next statement after the loop |
| termination condition | - the condition that causes the iteration to stop   |

When designing a loop, it is important to clearly identify each of these.

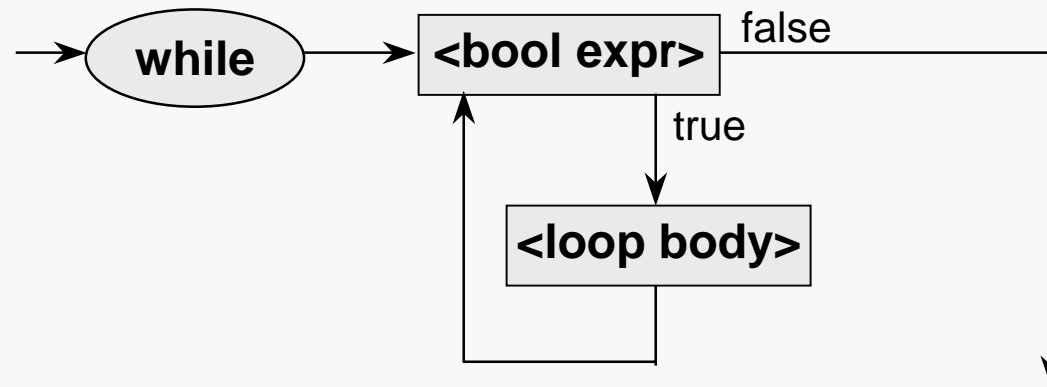
Understanding the termination condition, and what guarantees it will eventually occur, are particularly vital.

The most versatile loop construct in C++ is the `while` statement. Syntactically:



The Boolean expression must be enclosed in parentheses, and `<loop body>` can be either a single statement or a compound statement.

Semantically:



The Boolean expression is examined before each pass through the body of the loop. If the Boolean expression is true, the loop body is executed; if it is false, execution proceeds to the first statement after the loop body..

# while Loop Example

Example:

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

int main() {

    const char ASTERISK = '*';           // 1
    int numAst;                          // 2
    int numPrinted = 0;                  // 3

    cout << "How many asterisks do you want? "; // 4
    cin >> numAst;                        // 5

    while (numPrinted < numAst) {        // 6

        cout << ASTERISK;                 // 7
        numPrinted++;                     // 8
    }
    cout << endl;                          // 9

    return 0;                             // 10
}
```

This is a count-controlled loop. The loop control variable (LCV) is a variable whose value determines whether the loop body is executed.

Some observations:

- It is possible that the body of the loop will never be executed.
- It is possible that the loop test condition will become false in the middle of an iteration. Even so, the remainder of the loop body will be executed before the loop test is performed and iteration stops.
- The loop test must involve at least one variable whose value is (potentially) changed by at least one statement within the loop body. Why?

```
. . .  
while (numPrinted < numAst) { // 6  
    cout << ASTERISK; // 7  
    numPrinted++; // 8  
}  
cout << endl; // 9  
. . .
```

Count Controlled Loop a loop that terminates when a counter reaches some limiting value

The LCV (loop control variable) will be:

- an integer variable used in the Boolean expression
- initialized before the loop,
- incremented or decremented within the loop body

```
. . .
cout << "How many asterisks do you want? "; // 4
cin >> numAst; // 5

// modified implementation
while ( numAst > 0 ) { // 6

    cout << ASTERISK; // 7
    numAst--; // 8
}
cout << endl; // 9
. . .
```

# Event Controlled Loop

Event Controlled Loop: loop that executes until a specified situation arises to signal the end of the iteration.

```

. . .
const int CENTSPERDOLLAR = 100;           // 1
string Description;                       // 2
int Dollars,                               // 3
    Cents;                                 // 4
int totalCents = 0;                       // 5

getline(In, Description, '\t');           // 6
In >> Dollars;                            // 7
In.ignore(1, '.');                        // 8
In >> Cents;                               // 8
In.ignore(INT_MAX, '\n');                 // 10

while ( In ) {                            // 11

    totalCents = totalCents +
                + CENTSPERDOLLAR*Dollars
                + Cents;                  // 12

    // repeat lines 6 through 10
}
. . .

```

Visual C++	46.50
Excedrin	2.99
Excedrin Migraine	3.99
How to Solve It	33.95
The Art of War	17.50

This loop is terminated by the event of an input failure.

# Sentinel Controlled Loop

Sentinel loop: loop that terminates when a specified marker (dummy data value) is read, signaling the end of the iteration.

Here we modify the previous input file to include a special marker line to signify the end of the price data in the file:

```
const int CENTSPERDOLLAR = 100; // 1
const string SENTINEL = "xxxNoItemxxx"; // 2
. . .
getline(In, Description, '\t'); // 7
In >> Dollars; // 8
In.ignore(1, '.'); // 9
In >> Cents; // 10
In.ignore(INT_MAX, '\n'); // 11

while ( Description != SENTINEL ) { // 12

    totalCents = totalCents +
                + CENTSPERDOLLAR*Dollars
                + Cents; // 13

    // repeat lines 7 through 11
}
. . .
```

```
Visual C++ 46.50
Excedrin 2.99
Excedrin Migraine 3.99
How to Solve It 33.95
The Art of War 17.50
xxxNoItemxxx 0.00

Deliver to:
Joe Bob Hokie
666 Pritchard Hall
. . .
```

This loop is terminated by the event of reading the special sentinel value.

# An Improvement Using Hybrid Control

The pure sentinel control on the previous slide is risky. What would happen if the sentinel line were omitted?

We can eliminate this problem by adding a check for input failure to the loop test:

```
const int CENTSPERDOLLAR = 100; // 1
const string SENTINEL = " xxxNoItemxxx "; // 2
. . .
getline(In, Description, '\t'); // 7
In >> Dollars; // 8
In.ignore(1, '.'); // 9
In >> Cents; // 10
In.ignore(INT_MAX, '\n'); // 11

while ( In && Description != SENTINEL ) { // 12

    totalCents = totalCents +
                + CENTSPERDOLLAR*Dollars
                + Cents; // 13

    // repeat lines 7 through 11
}
. . .
```

Visual C++	46.50
Excedrin	2.99
Excedrin Migraine	3.99
How to Solve It	33.95
The Art of War	17.50
xxxNoItemxxx	0.00
. . .	

This loop is terminated by either the event of an input failure, or by the event of reading the special sentinel value.

It's fairly common to use a Boolean variable to record the results of a more complicated test. Here we assume that there is a sentinel line in which the price is invalid:

```
. . .
getline(In, Description, '\t');           // 6
In >> Dollars;                           // 7
In.ignore(1, '.');                       // 8
In >> Cents;                              // 9
In.ignore(INT_MAX, '\n');                 // 10

bool priceOK = (Dollars > 0) ||
               (Dollars == 0 && Cents > 0); // 11

while ( In && priceOK ) {                  // 12

    totalCents = totalCents +
                + CENTSPERDOLLAR*Dollars
                + Cents;                  // 13

    // repeat lines 6 through 10
    . . .
    priceOK = (Dollars > 0) ||
              (Dollars == 0 && Cents > 0); // 19
}
. . .
```

Visual C++	46.50
Excedrin	2.99
Excedrin Migraine	3.99
How to Solve It	33.95
The Art of War	17.50
xxxNoItemxxx	0.00
. . .	

Note that the Boolean loop control variable `priceOK` could be replaced by the condition used to set it, but the resulting code would perhaps not be as readable.

# Count Controlled Input

In some cases, the number of input records to be read will be specified in the input file itself. This is sometimes called the data header approach.

Here, we have a file of temperature data, with the first line specifying how many temperature values are given.

We might process this input file with the following loop:

```
. . .
inData >> tempsPromised; // get # temps expected
inData >> Temperature;   // read first one

while ( inData && tempsRead < tempsPromised ) {

    tempsRead++;          // count temps read
    // processing code goes here
    . . .

    inData >> Temperature; // read next one
}
. . .
```

7
79
65
78
62
73
81
89

Note that we do NOT place absolute trust in the data header value.

# Complete Example

Let's extend the last code fragment to a complete program. Suppose that we are required to determine the highest temperature and the overall average temperature.

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

int main() {

    ifstream inData("Temperature.data");

    int    tempsPromised;           // # temp values expected
    int    tempsRead  = 0;         // # temps read so far
    int    hiTemp     = INT_MIN;   // max temp so far
    double sumOfTemps = 0.0;      // sum for averaging
    int    Temperature;           // temperature just read

    inData >> tempsPromised;      // get # temps expected
    inData >> Temperature;        // read first one

    . . .
```

7
70
65
73
62
73
81
89

Note that we do NOT place absolute trust in the data header value.

# Complete Example

## 6. Iteration 12

```
    . . .
    while ( inData  && tempsRead < tempsPromised ) {

        tempsRead++;                // count temps read
                                   // and keep sum
        sumOfTemps = sumOfTemps + Temperature;

        if ( Temperature > hiTemp ) // check for new max
            hiTemp = Temperature;   // update if so

        inData >> Temperature;     // read next one
    }

    // Report results:
    if ( tempsRead > 0 ) {
        cout << "Highest temperature was: "
              << hiTemp << '.' << endl;

        double avgTemp = sumOfTemps / tempsRead;
        cout << fixed << showpoint;
        cout << "Average temperature was: "
              << setprecision(1) << avgTemp << '.' << endl;
    }

    return 0;
}
```

7
70
65
73
62
73
81
89

# End-of-line Controlled Loop

6. Iteration 13

```
...
In.get(Next);      // Read 1st char

while ( In ) {
    lineLength = 0;           // Restart line length count.
    while (Next != '\n') {   // Process next line:
        lineLength++;       // count char
        cout << Next;       // echo char
        In.get(Next);       // Read next char.
    }
    numChars = numChars + lineLength;
    numLines++;
    cout << setw(50 - lineLength) << lineLength << endl;
    In.get(Next);
}

cout << "Number of characters: " << numChars << endl;
cout << "Number of lines:      " << numLines << endl;
...
```

Things should be made ¶  
as simple as possible, ¶  
but no simpler. ¶§

¶ represents the newline char  
§ represents the end of file char

Questions that one should consider carefully when coding a loop:

- What is the condition that terminates the loop?
- How should the condition be initialized?
- How should the condition be updated?
- What guarantees this condition will eventually occur?
- What is the process to be repeated?
- How should the process be initialized?
- How should the process be updated?
- What is the state of the program on exiting the loop?
- Are "boundary conditions" handled correctly?

For loops are used whenever the number of times a loops needs to execute is known or can be calculated beforehand.

```
for (initial expression; test expression; update expression)
    <statement>
```

The initial expression is performed just once, prior to loop execution. The initial expression may declare variables as well as specify initializations for them.

The test expression is evaluated and if false then the next statement after the for loop is executed.

If the test expression evaluates to true, then the <statement> of the for loop is executed, the update expression is performed, and test expression is evaluated again.

For loops are generally count-controlled, but hybrid control is also fairly common, especially when input failure control is involved.

# for Loop Example

Example:

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

int main() {

    const char ASTERISK = '*'; // 1
    int numAst; // 2
    int numPrinted; // 3

    cout << "How many asterisks do you want? "; // 4
    cin >> numAst; // 5

    for (numPrinted =0; numPrinted < numAst; numPrinted++) { // 6

        cout << ASTERISK; // 7
    }
    cout << endl; // 8

    return 0; // 9
}
```

Compare this with the implementation on slide 3 using a `while` loop.

The `for` loop is simply an alternative control structure (to the `while`); any `for` loop can be expressed as a `while` loop, and vice versa.

## More for Loop Examples

6. Iteration 17

```
const char Separator = '-';
const int Length = 80;
int toWrite;

for (toWrite = Length; toWrite > 0; toWrite--) { // count down
    cout << Separator;
}
```

```
const int LIMIT = 100;
int sumOfSquares,
    Curr;

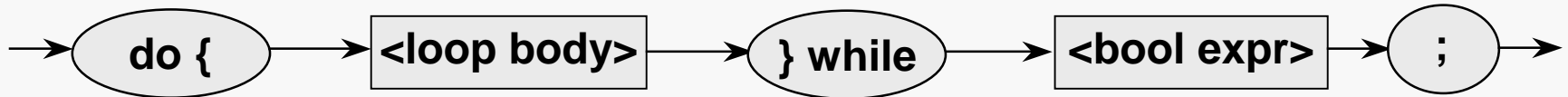
for (Curr = 1, sumOfSquares = 0; Curr <= LIMIT; Curr++)
    sumOfSquares = sumOfSquares + Curr * Curr;

cout << "Sum of squares = " << sumOfSquares;
```

```
. . .
for (Curr = 1, sumOfSquares = 0;
     Curr <= LIMIT;
     sumOfSquares = sumOfSquares + Curr * Curr, Curr++)
; // empty body, all the "action" is in the update section
. . .
```

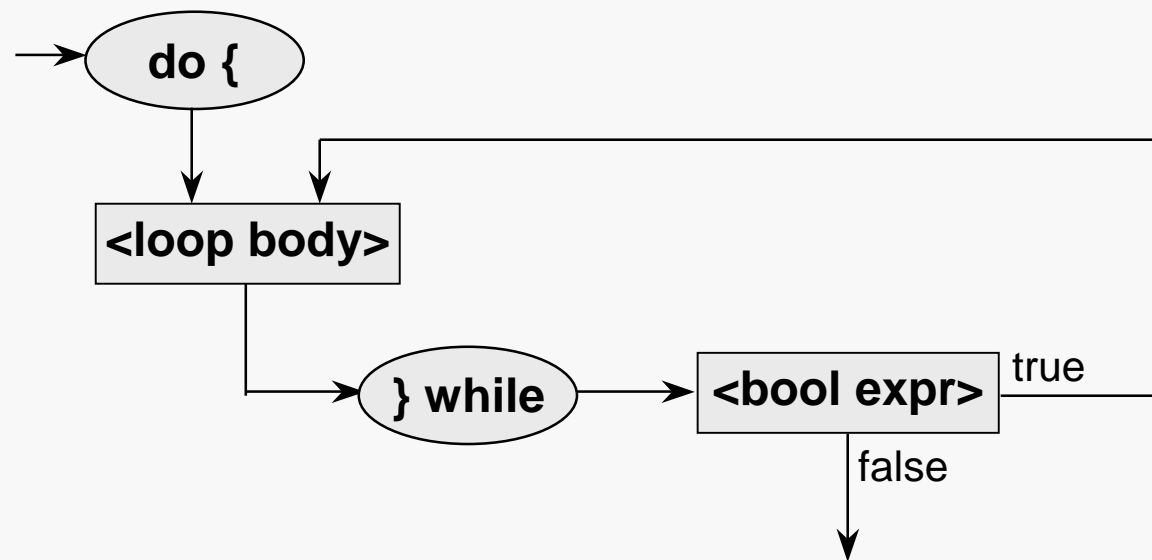
# do-while Loop

The third loop construct in C++ is the do-while statement. Syntactically:



The Boolean expression must be enclosed in parentheses, and **<loop body>** can be either a single statement or a list of statements. A peculiarity is that the loop body must be enclosed in braces, even if it is a single statement.

Semantically:



## do-while Example

Example: find first occurrence, if any, of a specific character in an input stream:

```
const char TOFIND = 'X';
char Next;
int Skipped = -1;

do {
    In.get(Next);
    Skipped++;
} while (In && Next != TOFIND);

if ( In ) {
    cout << TOFIND " found after " << Skipped
         << " characters were skipped." << endl;
}
else {
    cout << TOFIND " was not found." << endl;
}
```