COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 6

Looping

Chapter 6 Topics

- While Statement Syntax
- Count-Controlled Loops
- Event-Controlled Loops
- Using the End-of-File Condition to Control Input Data

Chapter 6 Topics

- Using a While Statement for Summing and Counting
- Nested While Loops
- Loop Testing and Debugging

Loops

What is a loop?

A loop is a repetition control structure that causes a single statement or block to be executed repeatedly

Two Types of Loops

Count controlled loops

Repeat a statement or block a specified number of times

Event-controlled loops

Repeat a statement or block until a condition within the loop body changes that causes the repetition to stop

While Statement

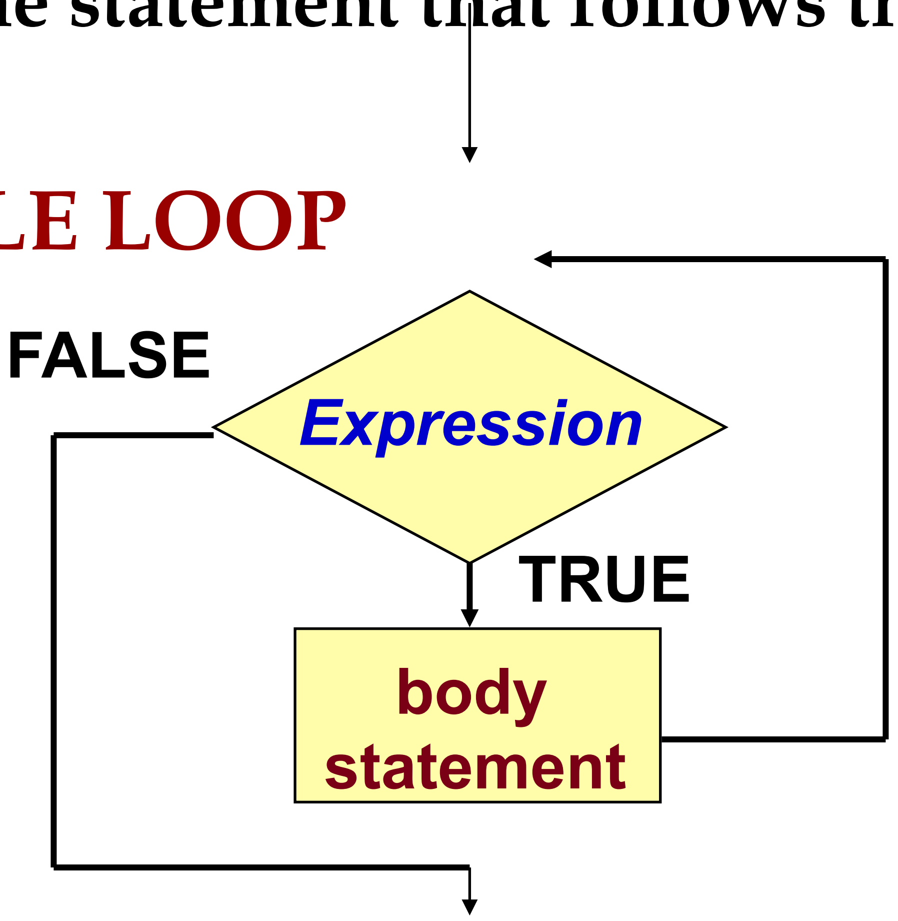
SYNTAX

```
while (Expression)
{       .
        .                  // loop body
        .
}
```

Loop body can be a single statement, a null statement, or a block

- **When the expression is tested and found to be false, the loop is exited and control passes to the statement that follows the loop body**

WHILE LOOP



Count-Controlled Loops

Count-controlled loops contain:

- An initialization of the loop control variable
- An expression to test if the proper number of repetitions has been completed
- An update of the loop control variable to be executed with each iteration of the body

Count-Controlled Loop Example

```
int   count;          // Loop-control variable

count  =  4;          // Initialize loop variable

while(count > 0)         // Test expression
{
    cout  << count  << endl; // Repeated action

    count --;    // Update loop variable
}
cout  << “Done” << endl;
```

Count-controlled Loop

```
int    count;

count  =  4;


while(count > 0)
{
     cout  << count  << endl;

     count --;
}
cout  << “Done” << endl;
```

count

OUTPUT

Count-controlled Loop

```
int    count;

count   =   4;


while(count > 0)
{
    cout   << count   << endl;

    count --;
}
cout   << “Done” << endl;
```

count

4

OUTPUT

Count-controlled Loop

```
int   count;

count  =  4;

while(count > 0)   TRUE
{
    cout  << count  << endl;

    count --;
}
cout  << “Done” << endl;
```

count

4

OUTPUT

Count-controlled Loop

```
int    count;

count   =   4;


while(count > 0)
{
     cout   << count   << endl;

     count --;
}
cout   << "Done" << endl;
```

count

4

OUTPUT

```
4
```

Count-controlled Loop

```
int   count;

count  =  4;


while(count > 0)
{
    cout  << count  << endl;

    count --;
}
cout  << "Done" << endl;
```

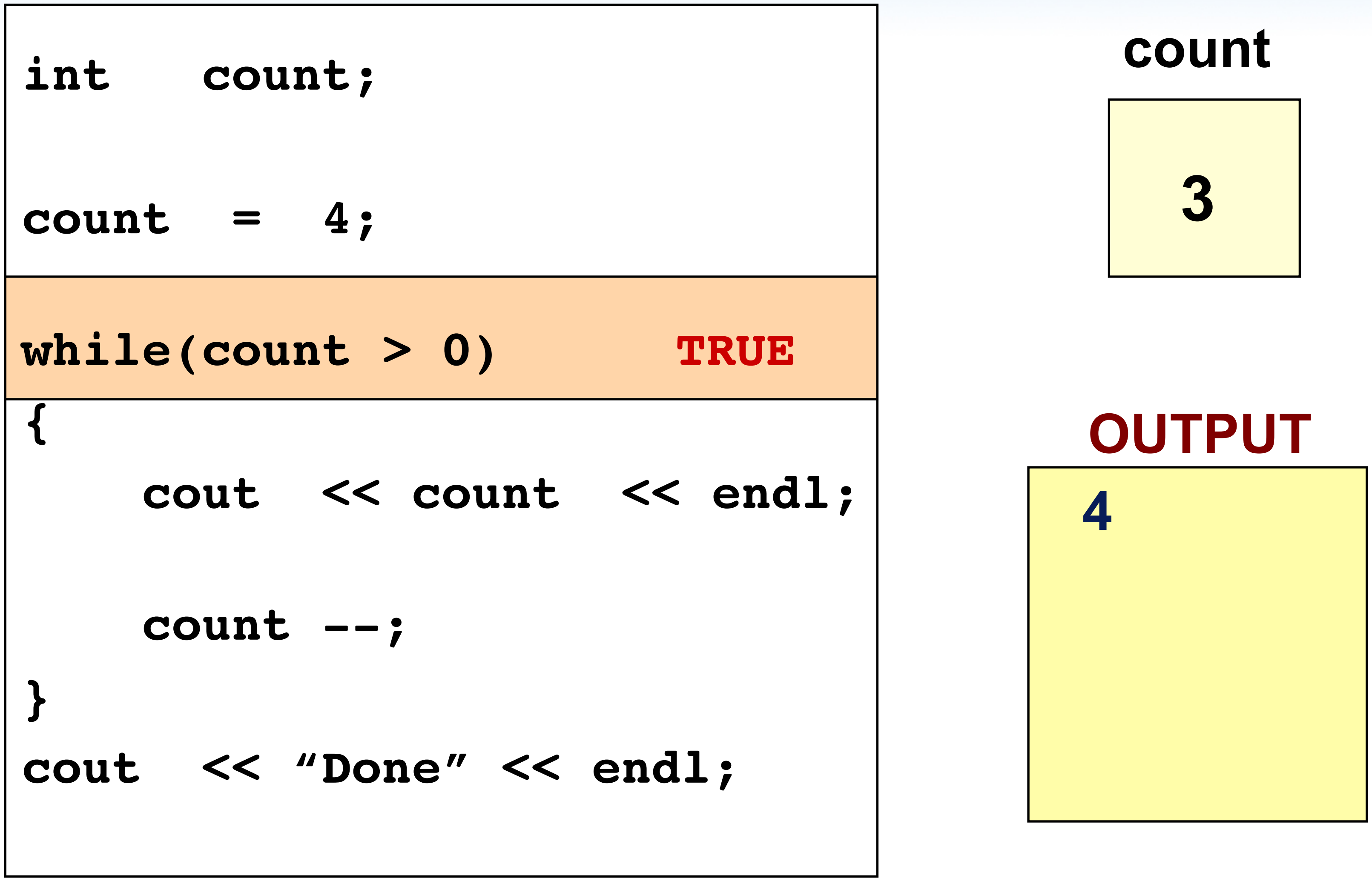
count

3

OUTPUT

```
4
```

Count-controlled Loop



```
int    count;

count   =   4;

while(count > 0)         TRUE
{
     cout   << count   << endl;

     count --;
}
cout   << "Done" << endl;
```

Count-controlled Loop

```
int    count;

count   =   4;

while(count > 0)
{
    cout   << count   << endl;

    count --;
}
cout   << “Done” << endl;
```

count

3

OUTPUT

```
4
3
```

Count-controlled Loop

```
int    count;

count   =   4;

while(count > 0)
{
    cout   << count   << endl;

    count --;
}
cout   << “Done” << endl;
```

count

2

OUTPUT

```
4
3
```

Count-controlled Loop

```
int   count;

count  =  4;

while(count > 0)       TRUE
{
    cout  << count  << endl;

    count --;
}
cout  << "Done" << endl;
```

count

2

OUTPUT

```
4
3
```

Count-controlled Loop

```
int   count;


count  =  4;


while(count > 0)
{
    cout  << count  << endl;

    count --;
}
cout  << "Done" << endl;
```

count

2

OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int    count;

count   =   4;

while(count > 0)
{
     cout   << count   << endl;

     count --;
}
cout   << "Done" << endl;
```

count

1

OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int    count;

count   =   4;

while(count > 0)        TRUE
{
     cout   << count   << endl;

     count --;
}
cout   << “Done” << endl;
```

count

1

OUTPUT

```
4
3
2
```

Count-controlled Loop

```
int    count;

count  =  4;

while(count > 0)
{
     cout  << count  << endl;

     count --;
}
cout  << "Done" << endl;
```

count

1

OUTPUT

```
4
3
2
1
```

Count-controlled Loop

```
int   count;

count  =  4;

while(count > 0)
{
    cout  << count  << endl;

    count --;
}
cout  << "Done" << endl;
```

count

0

OUTPUT

```
4
3
2
1
```

Count-controlled Loop

```
int    count;

count   =   4;

while(count > 0)          FALSE
{
      cout   << count   << endl

      count --;
}
cout   << "Done" << endl;
```

count

0

OUTPUT

```
4
3
2
1
```

Count-controlled Loop

```
int    count;

count   =   4;


while(count > 0)
{
     cout   << count   << endl;


     count --;
}

cout   << "Done" << endl;
```

count

0

OUTPUT

```
4
3
2
1
Done
```

Example

myInfile contains 100 blood pressures

Use a while loop to read the 100 blood pressures and find their total

```
ifstream  myInfile;
int       thisBP;
int       total;
int       count;

count = 0;                          // Initialize

while  (count < 100)                // Test expression
{
    myInfile   >>  thisBP;
    total = total + thisBP;
    count++;                        // Update
}

cout  <<  "The total = "  <<  total  <<  endl;
```

Types of Event-Controlled Loops

- **Sentinel controlled**

  Keep processing data until a special value that is not a possible data value is entered to indicate that processing should stop

- **End-of-file controlled**

  Keep processing data as long as there is more data in the file

- **Flag controlled**

  Keep processing data until the value of a flag changes in the loop body

Examples of Kinds of Loops

Count controlled loop	Read exactly 100 blood pressures from a file
End-of-file controlled loop	Read all the blood pressures from a file no matter how many are there

Examples of Kinds of Loops

Sentinel controlled loop	Read blood pressures until a special value selected by you(like -1) is read
Flag controlled loop	Read blood pressures until a dangerously high BP(200 or more) is read

A Sentinel-controlled Loop

- Requires a "priming read"

- A priming read is the reading of one set of data before the loop to initialize the variables in the expression

// Sentinel controlled loop

```
total = 0;

cout  <<  “Enter a blood pressure(-1 to stop) ”;
cin  >> thisBP;
```

// Sentinel controlled loop, cont...

```
while(thisBP != -1)    // While not sentinel
{
    total = total + thisBP;
    cout << “Enter a blood pressure(-1 to stop)”;
    cin >> thisBP;
}
cout << total;
```

End-of-File Controlled Loop

- Uses the fact that a file goes into the fail state when you try to read a data value beyond the end of the file to control the loop

// End-of-file controlled loop

```
total = 0;

myInfile  >>  thisBP; // Priming read
```

// End-of-file controlled loop, cont...

```
while(cin)  // While last read successful
{
    total = total + thisBP;
    cout  << “Enter blood pressure”;
    cin >> thisBP;     // Read another
}
cout << total;
```

// End-of-file at keyboard

```
total = 0;

cout  << “Enter blood pressure “
      << “(Ctrl-Z to stop)”;
cin  >> thisBP;          // Priming read
```

// End-of-file at keyboard, cont...

```
while(cin)  // While last read successful
{
    total = total + thisBP;
    cout  << “Enter blood pressure”;
    cin >> thisBP;     // Read another
}
cout << total;
```

Flag-controlled Loops

- Initialize a flag (to true or false)
- Use meaningful name for the flag
- A condition in the loop body changes the value of the flag
- Test for the flag in the loop test expression

Example of Flag-controlled Loop

```
countGoodReadings = 0;
isSafe = true;    // Initialize Boolean flag

while(isSafe)
{
    cin  >>  thisBP;
    if (thisBP >=  200)
        isSafe = false;  // Change flag value
```

Example, continued

```
else
        countGoodReadings++;
}

cout  <<  countGoodReadings  <<   endl;
```

Common Loop Uses

- Count all data values
- Count special data values
- Sum data values
- Keep track of current and previous values

Current and Previous Values

- Write a program that counts the number of != operators in a program file
- Read one character in the file at a time
- Keep track of current and previous characters

Keeping Track of Values

```
(x != 3)
{
   cout << endl;
}
```

FILE CONTENTS

previous	current	count
(	x	0
x	‘ ‘	0
‘ ‘	!	0
!	=	1
=	‘ ‘	1
‘ ‘	3	1
3	)	1

Loop Program Keeping Track of Current and Previous Values

```
int   count;
char  previous;
char  current;

count = 0;
inFile.get(previous);        // Priming reads
inFile.get(current);
```

Keeping Track of Current and Previous Values , continued

```
while(inFile)
{
    if((current == ‘=‘) && (previous == ‘!’))
        count++;
    previous = current;              // Update
    inFile.get(current);    // Read another
}
```

Nested Loops

```
initialize outer loop
while (outer loop condition)
{       . . .

    initialize inner loop
    while(inner loop condition)
    {
        inner loop processing and update
    }
     . . .
}
```

Patient Data

A file contains blood pressure data for different people. Each line has a patient ID, the number of readings for that patient, followed by the actual readings.

ID	howMany	Readings				
4567	5	180	140	150	170	120
2318	2	170	210			
5232	3	150	151	151		

Read the data and display a chart

Patient ID	BP Average
4567	152
2318	190
5232	151
.	.
.	.
.	.

There were 432 patients in file.

Algorithm

- Initialize patientCount to 0
- Read first ID and howMany from file

Algorithim, cont...

- While not end-of-file
  - Increment patientCount
  - Display ID
  - Read and sum this patient's BP's
  - Calculate and display average for patient
  - Read next ID and howMany from file
- Display patientCount

Designing Nested Loops

- **Begin with outer loop**
- **When you get to where the inner loop appears, make it a separate module and come back to its design later**

Designed Nested Loop Example

```
#include <iostream>
#include <fstream>

using namespace std;
```

Designed Nested Loop Example

```
int  main()
  {
      int patientCount; // Declarations
      int thisID;
      int howMany;
      int thisBP;
      int totalForPatient;
      int count;

      float average;

      ifstream  myInfile;
```

Designed Nested Loop Example, cont....

```
myInfile.open(“BP.dat”);

if (!myInfile) // Opening failed
{
  cout <<
   “File opening error. Program
  terminated.”;
  return  1;
}
cout << “ID Number Average BP” << endl;
patientCount = 0;
// Priming read
myInfile >> thisID >> howMany;
```

Designed Nested Loop Example, cont....

```
while(myInfile)// Last read successful
{
    patientCount++;
    cout << thisID;
    // Initialize inner loop
    totalForPatient = 0;
    count = 0;
    while(count < howMany)
    {
        myInfile >> thisBP;
        count ++;
        totalForPatient =
           totalForPatient + thisBP;
    }
```

Designed Nested Loop Example, cont....

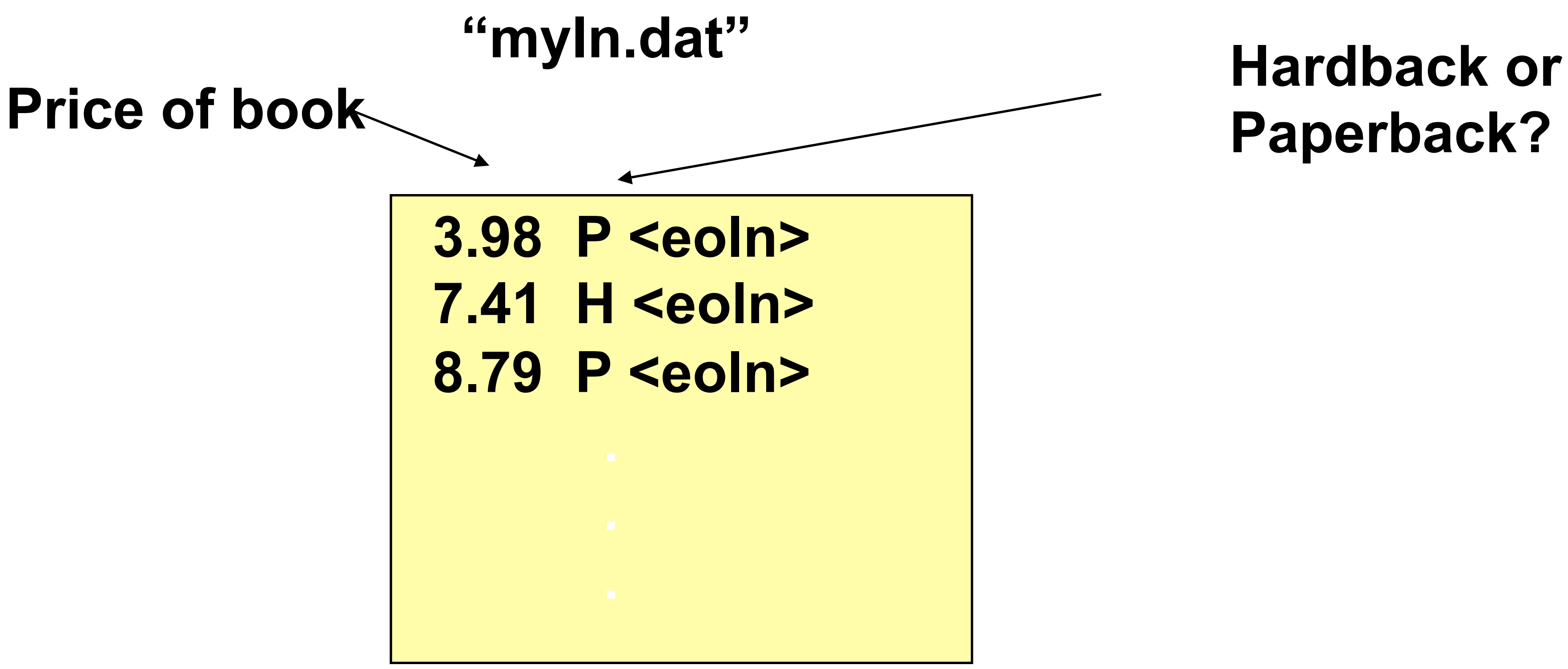
```
        average = totalForPatient / float(howMany);
        cout  <<  int(average + .5) << endl;
        // Another read
        myInfile  >> thisID  >>  howMany;
    }

    cout << “There were “ << patientCount
         << “patients on file.” << endl;

    cout << “Program terminated.” << endl;

    return 0;
}
```

Information About 20 Books in Diskfile



```
3.98  P <eoln>
7.41  H <eoln>
8.79  P <eoln>
        .
        .
        .
```

Write a program to find total value of all books

C++ Program

```
#include <iostream>          // Access cout
#include <fstream>           // Access file I/O

using namespace std;

int  main(void)
{
    float    price;              // Declarations
    char     kind;
    ifstream myInfile;
    float    total  =  0.0;
    int      count  = 1;
```

C++ Program, cont...

```
   myInfile.open(“myIn.dat”);

    // count-controlled processing loop
    while( count <= 20)
    {
        myInfile   >>  price  >>  kind;
       total = total + price;
       count ++;
    }
    cout << “Total is: “ << total << endl;
    myInfile.close();
    return 0;
}
```

Trace of Program Variables

count	price	kind	total
			0.0
1	3.98	‘P’	3.98
2	7.41	‘H’	11.39
3	8.79	‘P’	20.18
4	etc.		
20			
21	so loop terminates		

Complexity

- Complexity is a measure of the amount of work involved in executing an algorithm relative to the size of the problem

Polynomial Times

N	N^0 constant	N^1 linear	N^2 quadratic	N^3 cubic
1	1	1	1	1
10	1	10	100	1,000
100	1	100	10,000	1,000,000
1,000	1	1,000	1,000,000	1,000,000,000
10,000	1	10,000	100,000,000	1,000,000,000,000

Loop Testing and Debugging

- Test data should test all sections of program
- Beware of infinite loops -- program doesn’t stop
- Check loop termination condition, and watch for “off-by-1” bugs(OBOBs)
- Use `get` function for loops controlled by detection of ‘\n’ character

Loop Testing and Debugging

- Use algorithm walk-through to verify pre- and post conditions

- Trace execution of loop by hand with code walk-through

- Use a debugger to run program in “slow motion” or use debug output statements