Loops

Part 1

Looping, The **while** Statement

- The *if* statement chooses a course of action; the *while* statement is used to **repeat a course of action**
- A *loop* executes the same statement (simple or compound) over and over as long as a condition or set of conditions is satisfied

- **Syntax Template:** 
  ```
  WhileStatement
  while ( Expression )
  Statement
  ```

- The statement to be executed each time through the loop is called the **body** of the loop.
- If the Expression is **true**, execute the body and then test the Expression again. This continues until the expression is **false**
The **while** Statement (continued)

- Typical use of the while loop is with a block body
  ```c++
  int a = 0;
  while (a < 3)
  {
    cout << "square of: " << a << " is " << a*a << endl;
    a++;// note this line is important! Why?
  }
  cout << "Done with the while loop\n";
  ```

- What does the output look like if a typo occurs and we have `while (a > 3)` instead of `while (a < 3)`?

  See Loops1_01.cpp for C++ example of this program

---

**Figure 6-1: while Statement Flow of Control**

```
Program flow of control

While (Expression)

False (Loop Exit)  True (Loop Entry)

Statement 2 - next line after while loop

Statement 1 - body of while loop

Program flow of control
```
Phases of Loop Execution

- **Loop Test** - The point at which the `while` expression is evaluated
- **Loop Entry** - The point at which the flow of control reaches the first statement inside a loop
- **Iteration** - An individual pass through, or repetition of, the body of the loop
- **Termination Condition** - Condition that causes the exit of a loop
- **Loop Exit** - Point at which the repetition of the loop body ends and control passes to the first statement following the loop body

Types of Loops

- **Count Controlled Loops** -
  - Executed a specified number of times

- **Event Controlled Loops** -
  - Terminates when something changes the value of the expression
  - Depends on some event occurring while the loop body is executing

- If a loop is reading in data values, *A PRIMING READ needs to be performed before the while statement (Important point)*
  - Initializes the variables that are used for testing
  - Avoids using arbitrary values that are never used in the program
Count Controlled Loops

- Count Controlled loops use a *loop control variable*
  - Must be initialized before it is tested
  - must be incremented in the body of the loop. If the counting variable is not incremented, an infinite loop is created

**Example**

```c
int count = 1;
while (count <= 5) {
    Statement 1
    ...
    Statement n
    count++; // same as count = count + 1;
}
```

See Loops1_02.cpp for C++ example of this program

Event Controlled Loops

- **Sentinel-controlled** - Loop is executed until a special value is used (typically read) indicating the termination of the loop

- **End-of-file controlled** - Loop is executed until the End-of-File (EOF) is reached

- **Flag-controlled** - Loop is executed until a Boolean variable is changed (from true to false or false to true) such that the loop is exited. Can be used in a way that makes the loop very similar to sentinel-controlled loops
Sentinel-Controlled Loops

- A special data value (called the sentinel) is used to signal the program that there is no more data to be processed.
- The best sentinel is a value that would not be expected as normal input to the program (i.e. a negative number or key word).
- It is best to use a priming read with these loops:

  ```
  cin >> var1 >> var2; // Priming read
  while ( var1 > 0 && var2 < 0)
  {
    ... // statements for processing information
    cin >> var1 >> var2; // input next set of data points
  }
  ```

- What are possible sentinel values for this loop?
  - var1 = or var2 =

  See Loops1_03.cpp for C++ example of this program.

Sentinel-Controlled Loops

- The problem usually dictates the sentinel that is used.
- Entering a line of data one character at a time:

  ```
  char inChar;
  // use the get function to avoid missing white spaces
  cin.get(inChar); // priming read
  while ( inChar != '\n' ) // loop until end of line
  {
    cout << inChar;
    cin.get(inChar); // next character on the line
  }
  ```
Infinite Loop Example

- This example shows how poor code writing produces one of the common sources of error which causes the program to run indefinitely.

```cpp
bool moreData = true;
cout <<“enter 2 integers: ”;
cin >>var1 >>var2; // Priming read
if (var1 > var2) moreData = false;
while (moreData = true) // error here
{
    // processing statements
    // Obtain the next set of numbers
    cout <<“enter 2 integers: “;
cin >>var1 >>var2;
    if (var1 > var2) moreData = false;
}
```

See Loops1_03.cpp for C++ example of this program

Infinite Loop Example - fixed

- The following code is written better and more concisely. It removes the possibility of the mistake on the previous slide.

```cpp
bool moreData = true; // init not necessary
cout <<“enter 2 integers: “;
cin >>var1 >>var2; // Priming read
moreData = var1 <= var2;
while (moreData) // or better while(var1 <= var2)
{
    // processing statements
    // Obtain the next set of numbers
    cout <<“enter 2 integers: “;
cin >>var1 >>var2;
    moreData = var1 <= var2;
}
```

See Loops1_03.cpp for C++ example of this program
End-of-File Controlled Loops

- After the last piece of data has been read from a file, the input stream associated with that file is at the end of file (EOF).
- If any more attempts to read from the stream are made, the input stream goes into the fail state.
- Can use the fact that a stream has entered the fail state as a sentinel when reading from a file of unknown length.

```cpp
int value; ifstream inData;
inData.open("data_file.dat");
inData >> value; // Priming read
while (inData) // while still data in the file
{
    // perform any necessary statements
    // obtain the next value
    inData >> value;
}
```
See Loops1_04.cpp for C++ example of this program.

More on End-of-File Controlled Loops

- Remember that input streams can go into the fail state for any input error. So if invalid characters are in the input data file, the while loop will terminate prematurely.

- EOF-controlled and Sentinel-controlled loops do not know how many times the loop is to be executed.

- An EOF-controlled loop can be used with the standard input device using cin. To terminate the loop (indicate EOF), use Ctrl-d.
**File Stream Status Functions**

- **good()** – returns true if the file stream has no error flags set – file stream can be used
- **bad()** – returns true if previous I/O operation results in a loss of integrity of the stream (This is not the opposite of good)
- **fail()** – checks if either the fail status bit or bad status bit has been set. This function returns true when some error occurred with a stream operation other than reaching the end of the file
- **eof()** – returns true if the end of the file is reached with the previous I/O operation.
  - This function returns false until there is no more data in the file (typically the last character in a file will be the newline character).
  - This function will return true if the last newline character is read, or a read beyond the end of the file is attempted.
- **Examples:** In.good(), In.bad(), In.fail(), In.eof()

**Flag-Controlled Loops**

- A *flag* is a Boolean variable that is used to control the logical flow of a program

- Use the flag to record whether or not the event that controls the loop has occurred

- Flags can be initialized to either true or false - value is usually decided based on the variable name so that reading the statement indicates the desired action
  - **while (!Negative)** // Negative = false is the initial value
  - **while (!Finished)** // Finished = false is the initial value
  - **while (DataOk)** // DataOk = true is the initial value

  See Loops1_05.cpp for C++ example of this program
Looping Subtasks

- Look at three tasks that are often implemented within loops
  - **Counting** - loop keeps track of the number of times it has been executed. This does not have to be a count-controlled loop (and in most cases is not)
  - **Summing** - Loops can be used to sum a set of data values
  - **Keeping track of a previous value**

- **Iteration counter** - a counter variable that is incremented with each iteration of a loop.
- **Event counter** - a counter variable that is incremented only when a certain event occurs.

```cpp
ifstream inFile; inFile.open("input.dat");
int num_lines = 0;
giveFile(line); // priming read
while (inFile)
{
    num_lines++; //iteration counter
    getline(inFile,line);
}
// num_lines contains the number of lines
// in the input file
See Loops1_06.cpp for C++ example of this program
```
Looping Subtasks (continued)

- **Summing** - Loops can be used to sum a set of data values
  - **Iteration counter** - Counter is incremented for each iteration of the loop
  - **Event counter** - counter is only incremented when a certain event occurs.

```cpp
bool lessThanTen; int num_sum = 0; int sum = 0;
int num_loop = 0; int number;
lessThanTen = true; // flag to terminate loop
cin >> number;
while (lessThanTen && cin)
{
    num_loop++; // increment iteration counter
    if (number > 0)
    {
        num_sum++; // increment event counter
        sum = sum + number;
        lessThanTen = (num_sum <10); // new flag value
    }
    if (lessThanTen) cin >> number; // obtain the next value
}
// See Loops1_07.cpp for C++ example of this program
```

Looping Subtasks (continued)

- **Keeping track of a previous value**
  - Use a loop to read in input file entries one at a time, but keep track of the current entry and the previous entry
  - Once the EOF has been reached, the current entry variable contains the EOF and the previous entry variable contains the last entry from the file
  - See the NotEqualCount program in the book - page 258

```cpp
...  
inFile.get(currChar);
while (inFile)
{
    ...  
    prevChar = currChar;
    inFile.get(currChar);
}
```
For Statement - Chapter 7 - pages 314-320

• The for statement is designed to simplify the writing of count-controlled loops. for loops are a more compact version of a while loop.

• Syntax Template: ForStatement
  for (InitStatement Expression1; Expression2)
  Statement

  • The InitStatement ends in a semicolon (not shown)
    • Can be a null statement - just a semicolon
    • Can be a declaration or an expression
  • Expression 1 is the while condition - the condition used to determine when the loop terminates
  • Expression 2 increments, decrements or changes the loop counter
  • The Statement can be one statement or a block
  • all parts of the for statement are optional: for ( ; ; ) is valid

More on for Statement

• The InitStatement can be a declaration with initialization - variables declared in the for statement are available to the for statement only – Therefore, the scope of the variable is associated with the for statement only

• Example:
  int j = 30;
  for (int j = 1; j <= 20; j++)
    cout << “Iteration #” << j << endl;
  cout << “j is: “ << j;
  The last 2 lines output by the program are:
  Iteration #20
  j is: 30

  Here there are two distinct definitions/declarations of the variable j, so there will not be a “MULTIPLY DEFINED IDENTIFIER” compile error. This concept may not work with all compilers.
  See Loops1_08.cpp for C++ example of this concept
**for Loop Versus while Loop**

For ( count = 1 ; count <= n ; count ++ )

cout << count << endl;

count = 1;
while (count <= n)
{
    cout << count << endl;
    count++;
}

- Example

cin >> value; cin >> value;
while (value != -1) for ( ; value != -1; )
{
    cout << value << endl; cout << value << endl;
    cin >> value; cin >> value;
}

• Example

cin >> value; cin >> value;
while (value != -1) for ( ; value != -1; )
{
    cout << value << endl; cout << value << endl;
    cin >> value; cin >> value;
}

More on **for** Statement

- Another example
  - while loop
    const int END_VALUE = 123456; int value;
    cin >> value;
    while (value != END_VALUE)
        cin >> value;
  - Equivalent for loop
    const int END_VALUE = 123456; int value;
    for (cin>>value;value != END_VALUE;cin>>value)
        // NOTE: the for statement requires one statement –
        // here it is the null statement

See Loops1_08.cpp for more C++ examples of for loops
(Note: Loops1_08.cpp works on ray, but not visual C++)
How to Design Loops

- There are seven points associated with the design of loops
- Loops can be broken up into 2 tasks
  - designing the flow of control
    - What condition ends the loop
    - How should the condition be initialized
    - How should the condition be updated
  - designing the processing that takes place in the loop
    - What is the process being repeated
    - How should the process be initialized
    - How should the process be updated
- The loop should exit in the appropriate manner
  - What is the state of the program on exiting the loop

- Read pages 263 - 266

Nested Logic (Read pages 267-273)

- Just like if statements, while statements can be nested
- By nesting while statements, each while loop will have its own initialization, test and update
- General pattern:
  
  init outer loop
  
  while (outer loop condition)
  
  {
    outer loop processing
    
    init inner loop
    
    while (inner loop condition)
    
    {
      inner loop processing and further sub-loops
    }  
  }
  
- Nested loops are a good way to initialize multidimensional arrays (coming later)
More on Nested Loops

- Read Theoretical foundations - analysis of Algorithms. This goes into the complexity issues associated with Algorithms. Pages 274-277

- Loop-testing strategy - read pages 290-292
  - Loops should be tested over all parts of their construction (the seven points mentioned earlier) and for all possible loop conditions
  - Count-Controlled Loops: Is the count entered correctly? Updated correctly? Handle an invalid initialization?
  - Event-Controlled Loops: What happens if the event occurs before the loop is even executed? Can it handle different lengths of input? If sentinels are used, are they unique?

Repeated Opening of a File Stream

- Procedure for continually prompting for the name of an input file until one is successfully entered.

```cpp
// following is in main(or a function) of a program
ifstream inFile; string filename;
cout << "Enter the name of the input file (ctrl-c to exit): ";
cin >> filename;
cin.ignore(200, '\n'); // remove any extraneous characters
inFile.open(filename.c_str());
while (!inFile) // file stream in the fail state cause loop entry
{
    // break the attempted association of the file to the file stream
    inFile.clear(); // clear the file stream variable for reuse
    cout << "input file " << filename << " would not open\n";
    cout << "try again\n";
    cout << "Enter the name of the input file (ctrl-c to exit): ";
cin >> filename;
cin.ignore(200, '\n');
inFile.open(filename.c_str());
}
```