The Awk Utility: Awk as a UNIX Tool

What is awk ??

- **The word awk is derived from the names of its inventors!!!**
- **awk is actually Aho Weinberger and Kernighan .**
- **From the original awk paper published by Bell Labs, awk is**
  - **“ Awk is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.”**
- **Simply put, awk is a programming language designed to search for, match patterns, and perform actions on files.**

What is *Awk*

- *Awk* is a programming language used for manipulating data and generating reports
- The data may come from standard input, one or more files, or as output from a process
- *Awk* scans a file (or input) line by line, from the first to the last line, searching for lines that match a specified pattern and performing selected actions (enclosed in curly braces) on those lines.
- If there is a pattern with no specific action, all lines that match the pattern are displayed;
- If there is an action with no pattern, all input lines specified by the action are executed upon.

awk Versions

- awk – Original Bell Labs awk (Version 7 UNIX, around 1978) awk.
- awk – New awk (released with SVR4 around 1989)
- gawk – GNU implementation of awk standard.
- mawk – Michael's awk.
- …… and the list goes on.

Basic about awk

- **awk reads from a file or from its standard input, and outputs to its standard output.**
- **awk recognizes the concepts of "file", "record" and "field".**
- **A file consists of records, which by default are the lines of the file. One line becomes one record.**
- **awk operates on one record at a time.**
- **A record consists of fields, which by default are separated by any number of spaces or tabs.**
- **Field number 1 is accessed with $1, field 2 with $2, and so forth. $0 refers to the whole record.**

Awk's format

- **An *awk* program consists of:**
  - the *awk* command
  - the program instructions enclosed in quotes (or a file) , and
  - the name of the input file
- **If an input file is not specified, input comes from standard input (*stdin*), the keyboard**
- **Awk instructions consists of**
  - patterns,
  - actions, or
  - a combination of patterns and actions
- **A pattern is a statement consisting of an expression of some type**

Awk's format (continue.)

- Actions consist of one or more statements separated by semicolons or new lines and enclosed in curly braces
- Patterns cannot be enclosed in curly braces, and consist of regular expressions enclosed in forward slashes or expressions consisting of one or more of the many operators provided by *awk*
- *awk* commands can be typed at the command line or in *awk* script files
- The input lines can come from files, pipes, or standard input

Awk's format (continue.)

- Format:

```
awk 'pattern' filename
awk '{action}' filename
awk 'pattern {action}' filename
```

Running an AWK Program

- There are several ways to run an Awk program
  - awk ‘program’ input_file(s)
    - program and input files are provided as command-line arguments
  - awk ‘program’
    - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)
  - awk -f program_file_name input_files
    - program is read from a file

Input from Files

- Example 1:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58

$ awk '/Tom/' employees
Tom Billy         4/12/45 913-972-4536    $102
$
```

Input from Files (continue.)

- Example 2:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58

$ awk '{print $1}' employees
Chen
Tom
Larry
Bill
Steve
```

Awk's format (continue.)

- Example 3:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '/Steve/{print $1, $2}' employees
Steve Ann
```

Some of the Built-In Variables

- NF - Number of fields in current record
- NR - Number of records read so far
- $0 - Entire line
- $*n* - Field *n*
- $NF - Last field of current record

The *print* function

- **The default action is to print the lines that are matched to the screen**
- **The *print* function can also be explicitly used in the action part of *awk* as *{print}***
- **The *print* function accepts arguments as**
  - variables,
  - computed values, or
  - string constants
- **String must be enclosed in double quotes**
- **Commas are used to separate the arguments: if commas are not provided, the arguments are concatenated together**

The *print* function (continue.)

- The comma evaluates to the value of the output field separator (*OFS*), which is by default a space
- The output of the *print* function can be redirected or piped to another program, and another program can be piped to *awk* for printing

The *print* function (continue.)

- Example:

```
$ date
Fri Feb  9 07:49:28 EST 2001
$ date | awk '{ print "Month: " $2
"\nYear: ", $6}'
Month: Feb
Year:  2001
```

Escape sequences

- Escape sequences are represented by a backslash and a letter or number

Escape sequence	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\047	Octal value 47, a single quote
\c	*c* represents any other character, e.g., \"

Escape sequences (continue.)

- Example:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '/Ann/{print "\t\tHave a nice day, " $1,
$2 "\!"}' employees
Have a nice day, Steve Ann!
```

The *printf* Function

- The *printf* function can be used for formatting fancy output
- The *printf* function returns a formatted string to standard output, like the *printf* statement in C.
- Unlike the *print* function, *printf* does not provide a newline. The escape, *n*, must be provided if a newline is desired
- When an argument is printed, the place where the output is printed is called the *field*, and when the *width* of the field is the number of characters contained in that field

The *printf* Function (continue.)

- Example 1:

```
$ echo "UNIX" | awk ' {printf "|%-15s|\n", $1}'
|UNIX           |
$ echo "UNIX" | awk '{printf "|%15s|\n", $1}'
|           UNIX|
$
```

The *printf* Function (continue.)

- Example 2:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '{printf "The name is: %-15s ID is
%8d\n", $1, $3}' employees
The name is: Chen                ID is        5
The name is: Tom                 ID is        4
The name is: Larry               ID is       11
The name is: Bill                ID is        1
The name is: Steve               ID is        9
$
```

The *printf* Function (continue.)

Conversion Character	Definition
c	Character
s	String
d	Decimal number
ld	Long decimal number
u	Unsigned decimal number
lu	Long unsigned decimal number

The *printf* Function (continue.)

Conversion Character	Definition
x	Hexadecimal number
lx	Long hexadecimal number
o	Octal number
lo	Long octal number
e	Floating point number in scientific notation (*e*-notation)
f	Floating point number
g	Floating point number using either *e* or *f* conversion, whichever takes the least space

The *printf* Function (continue.)

Printf **Format Specifier**	**What it Does**
Given x='A',y=15,z=2.3, and $1 = Bob Smith:	
%c	Prints a single ASCII character. *printf("The character is %c|n",x)* prints: *The character is A*
%d	Prints a decimal number *printf("The boy is %d years old|n", y)* prints: *The boy is 15 years old*
%e	Prints the *e* notation of a number *printf("z is %f|n", z)* prints: z *is 2.3e+01*
%f	Prints a floating point number *printf("z is %e|n", z)* prints: z *is 4.600000*

The *printf* Function (continue.)

Printf **Format Specifier**	**What it Does**
Given x='A',y=15,z=2.3, and $1 = Bob Smith:	
%o	Prints the octal value of a number *printf("y is %o|n",y)* prints: *y is 16*
%s	Prints a string of characters *printf("The name of the culprit is %s|n", $1)* prints: *The name of the culprit is Bob Smith*
%x	Prints the *hex value* of a number *printf("y is %x|n", y)* prints: *y is F*

awk commands from within a file (continue.)

- If *awk* commands are placed in a file, the *-f* option is used with the name of the *awk* file, followed by the name of the input file to be processed
- A record is read into *awk*'s buffer and each of the commands in the *awk* file are tested and executed for that record
- If an action is not controlled by a pattern, the default behavior is to print the entire record

awk commands from within a file (continue.)

- If a pattern does not have an action associated with it, the default is to print the record where the pattern matches an input line

awk commands from within a file (continue.)

- Example:

```
Chen   Cho          5/19/63       203-344-1234    $76
Tom    Billy        4/12/45       913-972-4536    $102
Larry  White        11/2/54       908-657-2389    $54
Bill   Clinton      1/14/60       654-576-4114    $201
Steve  Ann          9/15/71       202-545-8899    $58
$ cat awkfile
/Steve/{print "Hello Steve!"}
{print $1, $2, $3}
$ awk -f awkfile employees
Chen Cho 5/19/63
Tom Billy 4/12/45
Larry White 11/2/54
Bill Clinton 1/14/60
Hello Steve!
Steve Ann 9/15/71
```

Records

- By default, each line is called a *record* and is terminated with a newline

The Record Separator

- By default, the output and input record separator (line separator) is a carriage return, stored in the built-in *awk* variables *ORS* and *RS*, respectively
- The *ORS* and *RS* values can be changed, but only in a limited fashion

The *$0* Variable

- An entire record is referenced as *$0* by *awk*
- When *$0* is changed by substitution or assignment, the value of *NF*, the number of fields, may be changed
- The newline value is stored in *awk*'s built-in variable *RS*, a carriage return by default

The *$0* Variable (continue.)

- Example:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '{print $0}' employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
```

The *NR* Variable

- The number of each record is stored in *awk*'s built-in variable, *NR*
- After a record has been processed, the value of *NR* is incremented by one

The *NR* Variable (continue.)

- Example:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '{print NR, $0}' employees
1 Chen Cho        5/19/63 203-344-1234    $76
2 Tom Billy       4/12/45 913-972-4536    $102
3 Larry White     11/2/54 908-657-2389    $54
4 Bill Clinton    1/14/60 654-576-4114    $201
5 Steve Ann       9/15/71 202-545-8899    $58
```

Fields

- Each record consists of words called *fields* which, by default, are separated by white space, that is, blank spaces or tabs. Each of these words is called a *field*, an *awk* keeps track of the number of fields in its built-in variable, *NF*
- The value of *NF* can vary from line to line, and the limit is implementation-dependent, typically 100 fields per line

Fields (continue.)

- Example 1:

```
$1     $2          $3              $4                  $5
Chen   Cho          5/19/63         203-344-1234        $76
Tom    Billy        4/12/45         913-972-4536        $102
Larry  White        11/2/54         908-657-2389        $54
Bill   Clinton      1/14/60         654-576-4114        $201
Steve  Ann          9/15/71         202-545-8899        $58
$ awk '{print NR, $1, $2, $5}' employees
1 Chen Cho $76
2 Tom Billy $102
3 Larry White $54
4 Bill Clinton $201
5 Steve Ann $58
```

Fields (continue.)

- Example 2:

```
awk '{print $0, NF}' employees
Chen Cho          5/19/63 203-344-1234    $76  5
Tom Billy         4/12/45 913-972-4536    $102 5
Larry White       11/2/54 908-657-2389    $54  5
Bill Clinton      1/14/60 654-576-4114    $201 5
Steve Ann         9/15/71 202-545-8899    $58  5
```

The Input Field Separator

- *awk*'s built-in variable, *FS*, holds the value of the input field separator.
- When the default value of FS is used, *awk* separates fields by spaces and/or tabs, stripping leading blanks and tabs
- The *FS* can be changed by assigning new value to it, either:
  - in a *BEGIN* statement, or
  - at the command line

The Input Field Separator (continue.)

- To change the value of *FS* at the command line, the *–F* option is used, followed by the character representing the new separator

The Input Field Separator (continue.)

- Example:

```
$ cat employees
Chen Cho:5/19/63:203-344-1234:$76
Tom Billy:4/12/45:913-972-4536:$102
Larry White:11/2/54:908-657-2389:$54
Bill Clinton:1/14/60:654-576-4114:$201
Steve Ann:9/15/71:202-545-8899:$58
$ awk -F: '/Tom Billy/{print $1, $2}'
employees
Tom Billy 4/12/45
-F (FS Field Separator)-f (prog file)
```

The Output Field Separator

- The default output field separator is a single space and is stored in *awk*'s internal variable, *OFS*
- The *OFS* will not be evaluated unless the comma separates the fields
- Example:

```
$ cat employees
Chen Cho:5/19/63:203-344-1234:$76
Tom Billy:4/12/45:913-972-4536:$102
Larry White:11/2/54:908-657-2389:$54
Bill Clinton:1/14/60:654-576-4114:$201
Steve Ann:9/15/71:202-545-8899:$58
$ awk -F: '/Tom Billy/{print $1 $2 $3 $4}' employees
Tom Billy4/12/45913-972-4536$102
```

Patterns

- A pattern consists of
  - a regular expression,
  - an expression resulting in a true or false condition, or
  - a combination of these
- When reading a pattern expression, there is an implied *if* statement

Actions

- Actions are statements enclosed within curly braces and separated by semicolons
- Actions can be simple statements or complex groups of statements
- Statements are separated
  - by semicolons, or
  - by a newline if placed on their own line

Regular Expressions

- A *regular expression* to *awk* is a pattern that consists of characters enclosed in forward slashes
- Example 1:

```
$ awk '/Steve/' employees
Steve Ann        9/15/71 202-545-8899    $58
```

- Example 2:

```
$ awk '/Steve/{print $1, $2}' employees
Steve Ann
```

Regular Expression Meta characters

^	Matches at the beginning of string
$	Matches at the end of string
.	Matches for a single character
*	Matches zero or more of preceding character
+	Matches for one or more of preceding character
?	Matches for zero or one of preceding character
[ABC]	Matches for any one character in the set of characters, i.e., *A*,*B*, or *C*

Regular Expression Meta characters (cont

[^ABC]	Matches characters not in the set of characters, i.e.,*A,B* or *C*
[A-Z]	Matches for any character in the range from *A* to *Z*
A/B	Matches either *A* or *B*
(AB)+	Matches one or more sets of *AB*
*|**	Matches for a literal asterisk
&	Used in the replacement to represent what was found in the search string

Regular Expressions (continue.)

- Example 3:

```
$ awk '/^Steve/' employees
Steve Ann        9/15/71 202-545-8899    $58
```

- Example 4:

```
$ awk '/^[A-Z][a-z]+ /' employees
Chen Cho         5/19/63 203-344-1234    $76
Tom Billy        4/12/45 913-972-4536    $102
Larry White     11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114     $201
Steve Ann        9/15/71 202-545-8899    $58
```

The Match Operator

- The match operator, the tilde (~), is used to match an expression within a record or a field
- Example 1:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
$ awk '$1 ~ /[Bb]ill/' employees
Bill Clinton      1/14/60 654-576-4114    $201
```

The Match Operator (continue.)

- Example 2:

```
$ awk '$1 !~ /lee$/' employees
Chen Cho          5/19/63 203-344-1234    $76
Tom Billy         4/12/45 913-972-4536    $102
Larry White       11/2/54 908-657-2389    $54
Bill Clinton      1/14/60 654-576-4114    $201
Steve Ann         9/15/71 202-545-8899    $58
```

awk Commands in a Script File

- When you have multiple *awk* pattern/action statements, it is often easier to put the statements in a script
- The script file is a file containing *awk* comments and statements
- If statements and actions are on the same line, they are separated by semicolons
- Comments are preceded by a pound (#) sign

awk Commands in a Script File (continue.)

- Example:

```
$ cat employees
Chen Cho:5/19/63:203-344-1234:$76
Tom Billy:4/12/45:913-972-4536:$102
Larry White:11/2/54:908-657-2389:$54
Bill Clinton:1/14/60:654-576-4114:$201
Steve Ann:9/15/71:202-545-8899:$58
$ cat info
# My first awk script by Abdelshakour Abuzneid
# Script name: info; Date: February 09, 2001
/Tom/{print "Tom's birthday is "$3}
/Bill/{print NR, $0}
/^Steve/{print "Hi Steve. " $1 " has a salary of " $4
"."}
#End of info script
```

awk Commands in a Script File (continue.)

- Example (continue.):

```
$ awk -F: -f info employees
Tom's birthday is 913-972-4536
2 Tom Billy:4/12/45:913-972-4536:$102
4 Bill Clinton:1/14/60:654-576-4114:$201
Hi Steve. Steve Ann has a salary of $58.
```

The Awk Utility:

Awk Programming Constructs

Comparison Expressions

- Comparison expressions match lines where if the condition is true, the action is performed
- The value of the expression evaluates true, and 0 if false

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
<=	Less than or equal to	x <= y
==	Equal to	x == y
!=	Not equal to	x != y
>=	Greater than or equal to	x >= y
>	Greater than	x > y
~	Matched by regular expression	x ~ / y/
!~	Not matched by regular expression	x !~ /y/

Relational Operators (continue.)

- Example:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    76
Tom Billy         4/12/45 913-972-4536    102
Larry White       11/2/54 908-657-2389    54
Bill Clinton      1/14/60 654-576-4114    201
Steve Ann         9/15/71 202-545-8899    58
$ awk '$5 == 201' employees
Bill Clinton      1/14/60 654-576-4114    201
$ awk '$5 > 100' employees
Tom Billy         4/12/45 913-972-4536    102
$ awk '$2 ~ /Ann/ ' employees
Steve Ann         9/15/71 202-545-8899    58
```

Relational Operators (continue.)

- Example (continue):

```
$ awk '$2 !~ /Ann/ ' employees
Chen Cho          5/19/63 203-344-1234    76
Tom Billy         4/12/45 913-972-4536    102
Larry White       11/2/54 908-657-2389    54
Bill Clinton      1/14/60 654-576-4114    201
```

Conditional Expressions

- A conditional expression uses two symbols, the question mark and the colon, to evaluate expression
- Format:

```
conditional expression1 ? expression2 : expression3
```

Conditional Expressions (continue.)

- Example:

```
$ awk '{max=($1 > $2) ? $1 : $2; print max}' employees
Cho
Tom
White
Clinton
Steve
```

Computation

- *awk* performs all arithmetic in floating point

Operator	Meaning	Example
+	Add	x + y
-	Substract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponentiation	x ^ y

Computation (continue.)

- Example:

```
$ awk '$5 * $5 > 3500' data.txt
```

Compound Patterns

- Compounds patterns are expressions that combine patterns with logical operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	!a

Compound Patterns (continue.)

- Example :

```
$ awk '$2 > 5 && $2 <= 15' employees
$
$ awk '$5 == 1000 || $3 > 50' employees
Steve Ann        9/15/71 202-545-8899    58
$
```

Range Patterns

- Range patterns match from the first occurrence of one pattern to the first occurrence of the second pattern, then match for the next occurrence of the second pattern, etc
- If the first pattern is matched and the second pattern is not found, *awk* will display all lines to the end of the file
- Example :

```
$ awk '/Tom/,/Steve/' employees
Tom Billy       4/12/45 913-972-4536    102
Larry White     11/2/54 908-657-2389    54
Bill Clinton    1/14/60 654-576-4114    201
Steve Ann       9/15/71 202-545-8899    58
$
```

The Awk Utility: Awk Programming

Numeric and String Constants

- Numeric constants can be represented as
  - Integer like 243
  - Floating point numbers like 3.14, or
  - Numbers using scientific notation like .723E-1 or 3.4
- Strings, such as *Hello* are enclosed in double quotes

User-Defined Variables

- User-defined variables consist of letters, digits, and underscores, and cannot begin with a digit
- Variables in *awk* are not declared
- If the variable is not initialized, *awk* initializes string variables to null and numeric variables to zero
- Variables are assigned values with *awk*'s assignment operators
- Example :

```
$ awk '$1 ~ /Tom/ {wage = $5 * 40; print wage}'
employees
4080
```

Increment and Decrement Operators

- The expression *x++* is equivalent to *x=x+1*
- The expression x– is equivalent to *x=x-1*
- You can use the increment and decrement operators either preceding operator, as in *++x*, or after the operator, as *x++*

```
{x = 1;y = x++; print x, y}
name="Nancy"          name is string
x++                   x is a number; x is
                      initialized to zero and
                      incremented by 1
number=35             number is a number
```

Built-in Variables

- Built-in variables have uppercase names. They can be used in expressions and can be reset

Variable Name	Variable Contents
ARGC	Number of command line argument
ARGV	Array of command line arguments
FILENAME	Name of current input file
FNR	Record number in current file
FS	The input field separator, by default a space

Built-in Variables (continue.)

Variable Name	Variable Contents
NF	Number of fields in current record
NR	Number of record so far
OFMT	Output format for numbers
OFS	Output field separator
ORS	Output record separator
RLENGTH	Length of string matched by *match* function
RS	Input record separator
RSTART	Offset of string matched by *match* function
SUBSEP	Subscript separator

Built-in Variables (continue.)

- Example:

```
$ awk -F: '$1 == "Steve Ann"{print NR, $1, $2, $NF}' employees2
5 Steve Ann 9/15/71 $58
```

BEGIN **Patterns**

- The *BEGIN* pattern is followed by an action block that is executed before *awk* processes any lines from the input file
- The *BEGIN* action is often used to change the value of the built-in variables, *OFS*, *RS*, *FS*, and so forth, to assign initial values to user-defined variables, and to print headers or titles as part of the output

BEGIN **Patterns** (continue.)

- Example 1 :

```
$  awk 'BEGIN{FS=":"; OFS="\t"; ORS="\n\n"}{print
$1,$2,$3}' employees2
Chen Cho          5/19/63 203-344-1234

Tom Billy         4/12/45 913-972-4536

Larry White       11/2/54 908-657-2389

Bill Clinton      1/14/60 654-576-4114

Steve Ann         9/15/71 202-545-8899

$
```

BEGIN **Patterns** (continue.)

- Example 2 :

```
$ awk 'BEGIN{print "Make  Year"}'
Make  Year
```

END Patterns

- *END* patterns do not match any input lines, but executes any actions that are associated with the *END* pattern. *END* patterns are handled *after* all lines of input have been processed
- Examples:

```
$ awk 'END{print "The number of records is " NR }'
employees
The number of records is 5
$ awk '/Steve/{count++}END{print "Steve was found
" count " times."}' employees
Steve was found 1 times.
$
```

Output Redirection

- When redirecting output from within *awk* to a *UNIX* file, the shell redirection operators are used
- The filename must be enclosed in double quotes
- Once the file is opened, it remains opened until explicitly closed or the *awk* program terminates
- Example:

```
$ awk '$5 >= 70 {print $1, $2 > "passing_file" }' employees
$ cat passing_file
Chen Cho
Tom Billy
Bill Clinton
```

The *getline* Function

- Reads input from
  - The standard input,
  - a pipe, or
  - a file other than from the current file being processed
- It gets the next line of input and sets the *NF*, *NR* and the *FNR* built-in variables

The *getline* Function (continue.)

- Examples :

```
$ awk 'BEGIN{ "date" | getline d; print d}' employees2
Fri Feb  9 09:39:53 EST 2001
$ awk 'BEGIN{ "date" | getline d; split( d, mon); print
mon[2]}' employees
Feb
$ awk 'BEGIN{while("ls" | getline) print}'
$ awk 'BEGIN{for(i=0;i<=NR ;i++) {getline ; print }}'
 data.txt
UNIX
varfile
varfile2
varfile3
varfile4
varfile5
varfile6
```

Pipes

- If you open a pipe in an *awk* program, you must close it before opening another one
- The command on the right-hand side of the pipe symbol is enclosed in double quotes

If **Statement**

- Format:

  If (expression) {

  statement; statement; …

  }

If/else **Statement**

- Format:

```
{If (expression) {
   statement; statement; ...
   }
else {
   statement; statement; ...
   }
}
```

If/else Statement

- Example:

```
$ awk '{if($6 > 50) print $1 "Too high"; \
> else print "Range is OK"}' names
Range is OK
Range is OK
Range is OK
Range is OK
Range is OK
$
```

If/else else if **Statement**

- Format:

```
{ If (expression) {
    statement; statement; ...
    }
  else if (expression) {
    statement; statement; ...
    }
  else if (expression) {
    statement; statement; ...
    }
  else {
    statement; statement; ...
    }
  }
```

Loops

- Loops are used to iterate through the field within a record and to loop through the elements of an array in the *END* block

While **Loop**

- The first step in using a *while* loop is to set a variable to an initial value
- The *do/while* loop is similar to the while loop, except that the expression is not tested until the body of the loop is executed at least once

While **Loop** (continue.)

- Example:

```
$ awk '{ i = 1; while (i <= NF ) { print NF,
$i; i++}}' names
2 jhon
2 smith
2 alice
2 cheba
2 tony
2 tram
2 dan
2 savage
2 eliza
2 goldborg
$
```

for **Loop**

- *for* loop requires three expressions within the parentheses: the initialization expression, the test expression and the expression to update the variables within the test expression
- The first statement within the parentheses of the *for* loop can perform only one initialization

for **Loop** (continue.)

- Example:

```
$ awk '{ i = 1; while (i <= NF ) { print NF,
$i; i++}}' names
2 jhon
2 smith
2 alice
2 cheba
2 tony
2 tram
2 dan
2 savage
2 eliza
2 goldborg
$
```

break and *continue* Statement

- The *break* statement lets you break out of a loop if a certain condition is true
- The *continue* statement causes the loop to skip any statement that follow if a certain condition is true, and returns control to the top of the loop, starting at the next iteration
- Example:

```
(In Script)
{ if ($1  Peter} {next}}
      else {print}
}
```

next Statement

- The *next* statement gets the next line of input from the input file, restarting execution at the top of the *awk* script
- Example:

(In Script)

```
1 {for ( x = 3; x <= NF; x++)
      if ( $x < 0) {print “Bottomed out!”; break}
      # breaks out of the loop
  }
2 {for ( x = 3; x <= NF; x++ )
      if ( $x == 0 )  { print “Get next item”; continue}
      # starts next iteration of the for loop
  }
```

exit Statement

- The *exit* statement is used to terminate the *awk* program. It stops processing records, but does not skip over an *END* statement
- If the exit statement is given a value between 0 and 255 as an argument (exit 1), this value can be printed at the command line to indicate success or failure by typing:

Arrays

- Arrays in *awk* are called *associative* arrays because the subscripts can be either
  - number, or
  - string
- The keys and values are stored internally is a table where a hashing algorithm is applied to the value of the key in question
- An array is created by using it, and *awk* can infer whether or not is used to store numbers or strings

Arrays (continue.)

- Array elements are initialized with
  - numeric value, and

    You do not have to declare the size of an array
- *awk* arrays are used to collect information from records and may be used for accumulating totals, counting words, tracking the number of times a pattern occurred

Arrays (continue.)

- Example:

```
$ cat employees
Chen Cho          5/19/63 203-344-1234    76
Tom Billy         4/12/45 913-972-4536    102
Larry White       11/2/54 908-657-2389    54
Bill Clinton      1/14/60 654-576-4114    201
Steve Ann         9/15/71 202-545-8899    58
$ awk '{name[NR]=$2};END{for(i=1; i<=5;i++)
 print i, name[i]}' data.txt
1 Cho
2 Billy
3 White
4 Clinton
5 Ann
```

Arrays (continue.)

- Example:

```
$ awk '{id[NR]=$3};END{for(x = 1; x<= NR;
x++)print id[x]}' data.txt
5/19/63
4/12/45
11/2/54
1/14/60
9/15/71
$
```

Using Field Values as Array Subscripts (continue.)

- Example 2:

```
$ cat db
Tom Jones
Mary Adams
Sally Chang
Billy Black
Tom Savage
$ awk '{count[$2]++}END{for(name in count)print name,
count[name] }' db
Chang 1
Black 1
Jones 1
Savage 1
Adams 1
$
```

Arrays and the *split* Function

- *awk*'s built-in *split* function allows you to split string into words and store them in an array
- You can define the field separator or use the value currently stored in *FS*
- Format:

```
split(string, array)
```

The *delete* Function

- The *delete* function removes an array elements

ARGV

- Command line arguments are available to *awk* with the built-in array called *ARGV*
- These arguments include the command *awk*, but not any of the options passed to *awk*
- The index of the ARGV array starts at zero

ARGC

- *ARGC* is a built-in variable that contains the number of command line arguments
- Example:

```
$ cat myscript
#This script is called myscript
BEGIN{
        for ( i = 0; i< ARGC; i++){
                printf("argv[%d] is %s\n", i,
ARGV[i])
                }
        printf("The number of arguments, ARGC=%d\n",
ARGC)
}
```

ARGC (continue.)

- Example :

```
$ awk -f ARGVS datafile "Peter Pan" 12
argv[0] is awk
argv[1] is datafile
argv[2] is Peter Pan
argv[3] is 12
The number of arguments, ARGC=4
$
```

The *length* Function

- The *length* function returns the number of characters in a string
- Without an argument, the *length* function returns the number of characters in a record
- Format:

```
length (string)
length
```

The *length* Function

- Example

```
$ awk 'END{ print length("hello") }' data.txt
5
```

The *match* Function

- The *match* function returns the index where the regular expression is found in the string, or zero if not found
- The match function sets the built-in variable *RSTART* to the starting position of the substring within the string, and *RLENGTH* to the number of characters to the end of the substring
- Format:

```
match (string, regular expression)
```