Lesson #5

Repetition and Loops

Repetition and Loops

Loop: A control structure that repeats a group of steps (statements) in a program.

Loop body: Contains the statements that are repeated in the loop.

Do I Need a Loop?

1. Are there any steps you must repeat to solve the problem? If yes, you need a loop.

2. Do you know in advance the number of repetitions? If yes, you need a counting loop.

3. Do you know when to stop the repetition? If not, you will not be able to program the loop.

Types of Loops

Counting Loop: A loop with a fixed number of repetitions.

- *Ex: Repeat 100 times...*

Sentinel-Controlled Loop: *A loop that reads values from a file or keyboard and stops reading when a certain value (called a sentinel) is read.*

- *Ex: Read numbers continuously until you encounter a value of -99.*

Types of Loops

End-of-File Controlled Loop: A loop that reads values from a file and stops when there are no more values to read.

- *Ex: Read numbers continuously until you encounter the end of the file.*

Input Validation Loop: *A loop that keeps asking a value from the user until the user gets it right.*

- *Ex: Keep asking the user for a positive number until the user enters one.*

Types of Loops

General Conditional Loop: A loop that checks a certain condition and repeats the statements in the loop body if the condition is true. When the condition is false, the loop is ended and the statements are not repeated. This kind of loop encompasses all the other kinds.

- *Ex: Print numbers on the screen while the numbers are below 100.*

Loops in C

- In C, all loop are implemented with general conditional loops. By programming them properly, we can achieve all the types of loops.

- For example, a counting loop (repeat 100 times) will be done by starting a variable at 1, then have a condition to stop the loop when the variable becomes larger than 100. Inside the loop the program will add 1 to the variable at each repetition.

The *while* Statement

- The while statement permits the repetition of the statements until the condition becomes false.

Syntax:

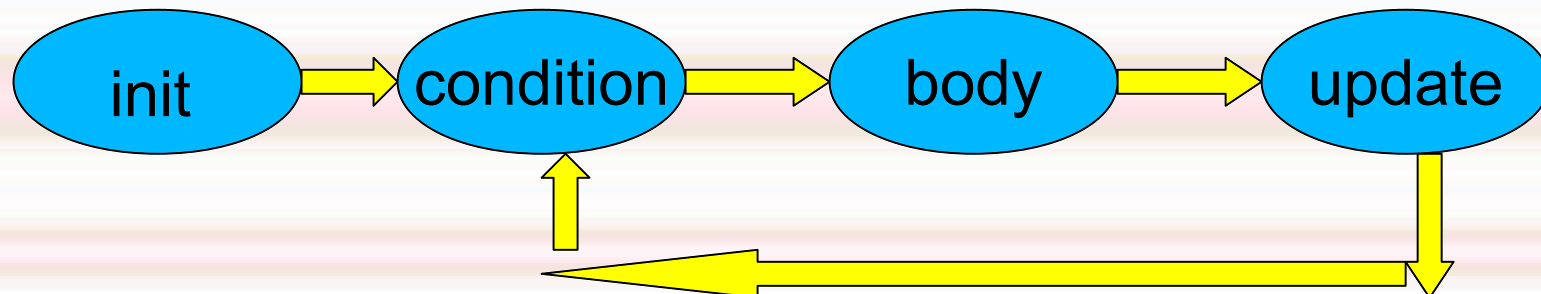
```
while (condition)
{
  statements executed/repeated if
  condition is true
}
```

A Counting Loop

```
int n;
n = 1; /* initialization */
while (n <= 100) /* condition */
{
    printf ("%d ", n); /* body */
    n = n + 1; /* update */
}
```

Execution of the while Statement

- **In a while loop, the initialization is performed once, then the condition is checked. If the condition is true, the body and update statements are executed and the condition is then checked again. If the condition is false, the loop ends and the program continues to the next statement.**

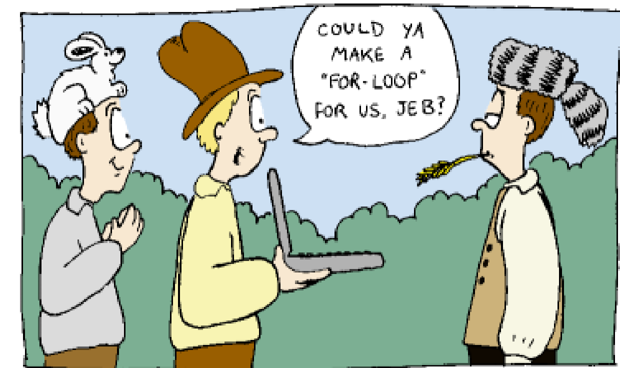


The *for* Statement

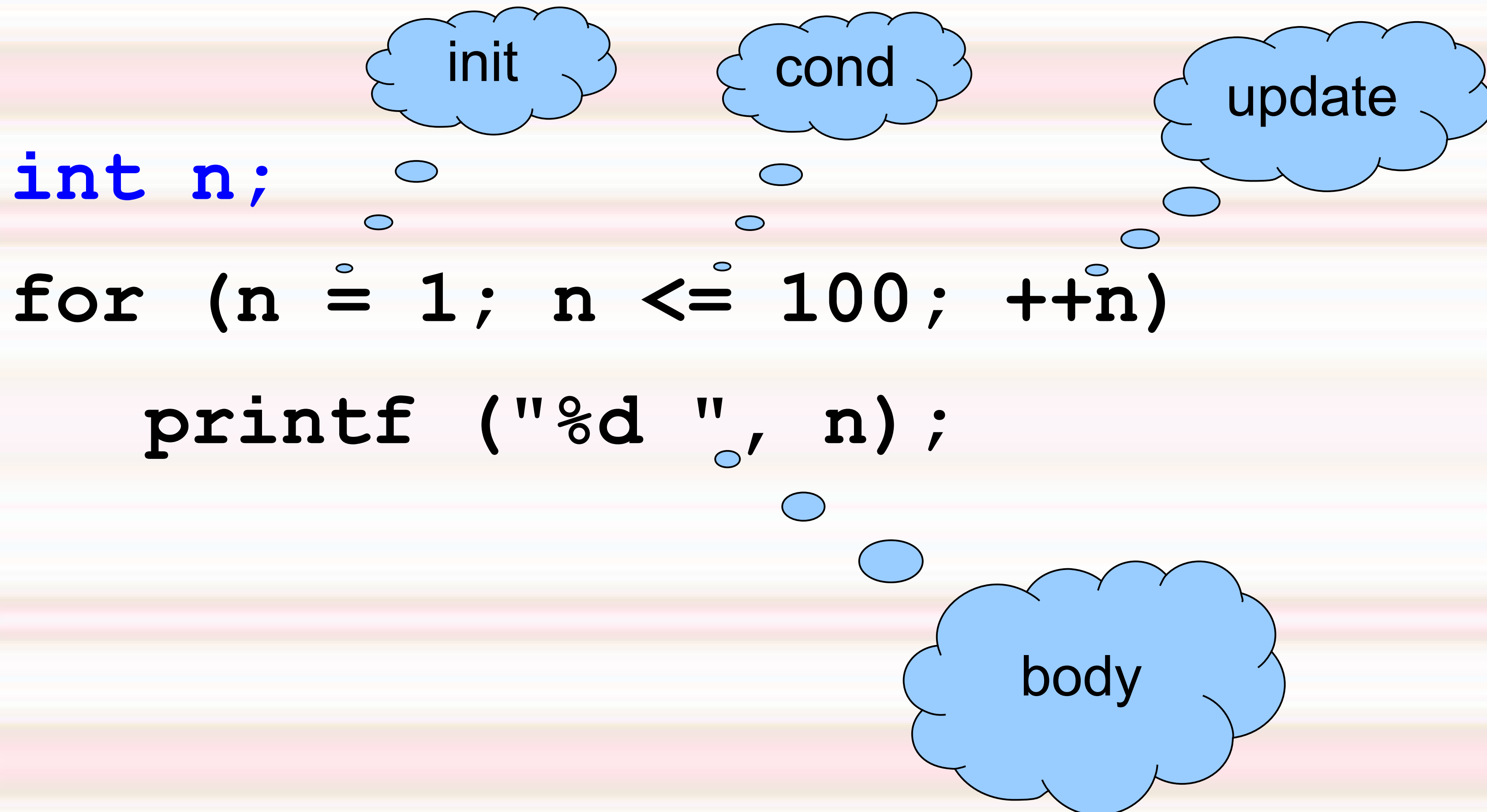
- The for statement permits the repetition of the statements until the condition becomes false. It works exactly like a while statement except that the syntax is slightly different.

Syntax:

```
for (initialization; condition; update)
{
   statements
   executed/repeated
   if condition is true
}
```



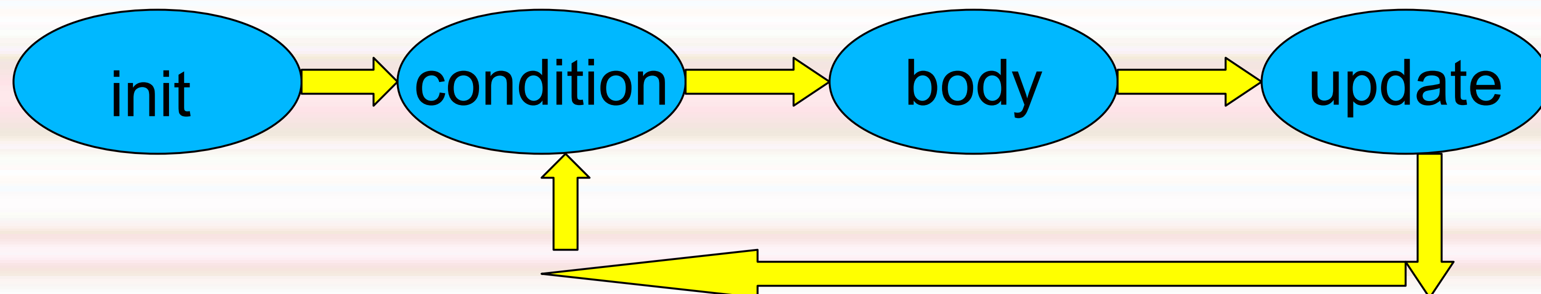
Counting Loop (*for* Statement)




```
int n;
for (n = 1; n <= 100; ++n)
    printf ("%d ", n);
```

Execution of the for Statement

- In a for loop, the initialization is performed once, then the condition is checked. If the condition is true, the body and update statements are executed and the condition is then checked again. If the condition is false, the loop ends and the program continues to the next statement.



for **vs.** *while*

- The **for** loop is used when we know in advance the number of iterations (counting loops).
- The **while** loop is used when we don't.
- This is a style issue only. We know that the for statement works exactly like the while statement.

Sentinel-Controlled Loop

```
/* program to compute the sum of all
  numbers entered */
int number,sum;
sum = 0;
printf ("Enter a number (Enter 0 to
         finish): ");
scanf ("%d", &number);
while (number != 0)
{
   sum = sum + number;
   printf ("Enter a number (Enter 0 to
            finish): ");
   scanf ("%d", &number);
}
printf ("The sum is %d.\n", sum);
```

0 is the sentinel

. not part of %d, just the end of the sentence

EOF-Controlled Loop

- An end-of-file (EOF) controlled loop is used to read values until the end of file is encountered.
- This sort of loop uses the fact that a file goes into a **fail status** when you try to read a data value beyond the end of a file.

Note: We will see five versions of eof-controlled loops using input redirection (scanf) and file I/O (fscanf). Knowledge of one method is sufficient, no need to know all. If you prefer redirection, look at versions I and II, if you prefer file I/O look at versions III, IV, and V. Note that redirection is often easier for beginners.

Status

- In a reading statement (like scanf), we can check the status of the readings by assigning the statement to an integer variable.
- If everything works as intended, the status will be the number of variables filled by the reading statement. (Ex: in status=scanf ("%d%d", &a,&b); , status will have a value of 2).
- When reading from a file, sometimes there are no more numbers to read. In that case the status becomes -1, also known as EOF.

EOF-Controlled Loop I

```
/* program to compute the sum of all the
  numbers present in a file */

int number, sum, status;
sum = 0;
status = scanf ("%d", &number);
while (status != EOF)
{
    sum = sum + number;
    status = scanf ("%d", &number)
}
printf ("The sum is %d.\n", sum);
```

Try to read a first number. Status will be either 1 (success) or -1 (fail).

Check if at end of file (not fail is success). If not, proceed inside loop.

Alternatives to !=EOF in this case are:
!= -1
== 1.

EOF-Controlled Loop II

```
/* program to compute the sum of all the
  numbers present in a file - equivalent
  to previous example */
int number, sum;
sum = 0;
while (scanf ("%d", &number) != EOF)
{
    sum = sum + number;
}
printf ("The sum is %d.\n", sum);
```

Try to read a first number.

Check if at end of file. If not, proceed inside loop.

Can you spot the differences between I and II?

This is the preferred version.

EOF-Controlled Loop III

```
/* similar to version I  - with fscanf */

int number, sum, status;
FILE* in;
in  = fopen ("data.txt", "r");
sum = 0;
status = fscanf (in, "%d", &number);
while (status != EOF)
{
    sum = sum + number;
    status = fscanf (in, "%d", &number);
}
printf ("The sum is %d.\n", sum);
fclose (in);
```

Try to read a first number. Status will be either 1 (success) or -1 (fail).

Check if at end of file (not fail is success). If not, proceed inside loop.

Alternatives to !=EOF in this case are:
!= -1
== 1.

EOF-Controlled Loop IV

```
/* similar to version II -  with fscanf */
int number, sum;
FILE* in;
in  = fopen ("data.txt", "r");
sum = 0;
while (fscanf (in, "%d", &number) != EOF)
{
   sum = sum + number;
}
printf ("The sum is %d.\n", sum);
fclose (in);
```

Try to read a first number.

Check if at end of file. If not, proceed inside loop.

EOF-Controlled Loop V

```
/* a version using the feof statement */
int number, sum;
FILE* in;
in  = fopen ("data.txt", "r");
sum = 0;
fscanf (in, "%d", &number);
while (!feof (in))
{
    sum = sum + number;
    fscanf (in, "%d", &number);
}
printf ("The sum is %d.\n", sum);
fclose (in);
```

Try to read first number

Check if at end of file. If **not**, proceed inside loop.

The do-while Statement

- The do-while statement is yet another version of the repetition in C but works slightly different.
- Contrary to the while and for statements, the body of the loop is executed once even if the condition is false because the condition is checked at the end of the loop, not the beginning.

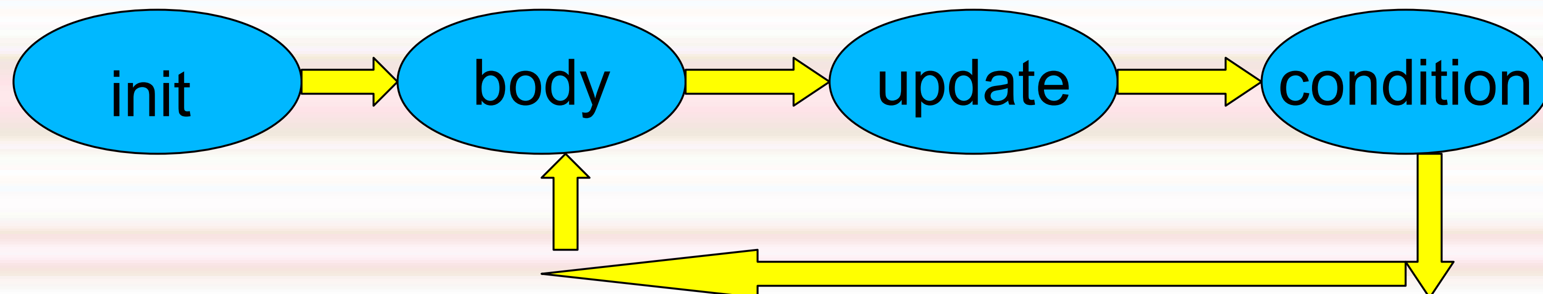
```
do
{
     statements that will be repeated;
} while (condition);
```

The do-while Statement

```
/* A do-while loop  */

int number, sum;
sum = 0; number = 1;
do
{
   sum = sum + number;
   ++number;
}while (sum <= 5); /* see the ; here */

printf ("The sum is %d.\n", sum);
printf ("The number is %d.\n", number);

/*what will be the printed values of sum and
  number? */
```

Execution of the do-while *Statement*

- In a do-while loop, the initialization is performed once, then the body and update statements are executed, and finally the condition is checked. If the condition is true, the body and update are executed once more. If the condition is false, the loop ends and the program continues to the next statement.



Input Validation Loop

```
/* input validation loop  */
int n;

do
{
   printf ("Enter a number between 1 and
             5");
   scanf ("%d", &n);
}while (n < 1 || n > 5);

/* rest of program  */
...
```

Nested Loops

- A nested loop is a loop within a loop, an inner loop within the body of an outer one.
- The inner and outer loops need not be generated by the same control structure (one can be a while loop, the other a for loop).

Nested Loops

- The inner loop must be completely embedded in the outer loop (no overlaps).
- Each loop must be controlled by a different index (loop control variable). i and j are often used and sometimes the outer loop (big loop) is called the i-loop and the inner loop (little loop) the j-loop.

A Nested Loop Example

- Suppose we want to display a 12x12 multiplication table. A nested loop here will make perfect sense. We can use the outer loop for the multiplicand and the inner loop for the multiplicator.
- The idea is to loop the outer loop 12 times and for each value, loop the inner loop 12 times as well.

A Nested Loop Example

```
#include <stdio.h>
int
main (void)
{
   int i, j;
   for (i = 1; i <= 12; ++i)
   {
      for ( j= 1; j <=12 ; ++j)
         printf ("%4d", i * j);
      printf ("\n");
   }
   return (0);
}
```

A Nested Loop Example

Quincy 2005

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144

Press Enter to return to Quincy...
```

See programs 6, 7, 8, and 9 at **c.ihypress.ca/04.php** for more advanced examples of nested loops.

End of Lesson