Lab 04 – Repetitions & Loop Statements

SE1012 – Programming Methodology **Semester 2, 2023**

Repetition of steps in a program is called "Loop". There are three main repletion control statements used in C:

•while
•for
•do - while

Repetition, also known as looping or iteration, is a fundamental concept in programming that involves executing a certain block of code multiple times. It allows you to automate tasks that need to be performed repeatedly without having to write the same code over and over again. Repetition is essential for creating efficient and effective programs, as it helps streamline processes and manage complex tasks.

In the context of the C programming language, repetition is achieved using loops. There are three main types of loops in C:

1. **while Loops:** A **while** loop repeatedly executes a block of code as long as a specified condition remains true. It's useful when you want to continue looping until a certain condition is no longer met.
2. **for Loops:** A **for** loop is a compact way of specifying loop initialization, condition, and increment/decrement all in one line. It's often used when you know the number of iterations in advance.
3. **do-while Loops:** A **do-while** loop is similar to a **while** loop, but it guarantees that the loop body is executed at least once, even if the condition is initially false.

Loops provide a powerful mechanism for handling repetition in your programs. They help you avoid redundant code and make your code more concise and maintainable. However, it's important to use loops judiciously and consider factors such as loop termination conditions and potential performance impacts.

Kind	When Used	C Implementation Structures
Counting loop	We can determine before loop execution exactly how many loop repetitions will be needed to solve the problem.	`while` `for`
Sentinel-controlled loop	Input of a list of data of any length ended by a special value	`while, for`
Endfile-controlled loop	Input of a single list of data of any length from a data file	`while, for`
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered	`do-while`
General conditional loop	Repeated processing of data until a desired condition is met	`while, for`

The while statement

A **while** loop is a fundamental control structure in programming that allows you to execute a specific block of code repeatedly as long as a given condition remains true. It provides a way to automate tasks that need to be performed multiple times without duplicating the same code. The loop continues to execute as long as the condition remains true, and it terminates when the condition becomes false.

The general syntax of a **while** loop in most programming languages, including C, looks like this:

```
while (condition) {
    // Loop body: code to be executed repeatedly as long as the condition is true
}
```

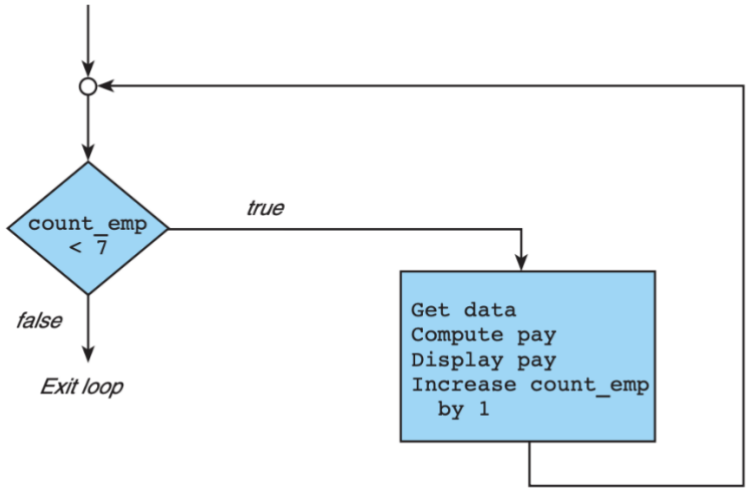
Here's a simple step-by-step explanation of how a **while** loop works:

1. The loop begins with the evaluation of the condition. If the condition is true, the control enters the loop body; if false, the loop is bypassed, and execution proceeds to the statement following the loop.
2. The code within the loop body is executed.
3. After executing the loop body, the program returns to the beginning of the loop, re-evaluates the condition, and decides whether to repeat the loop or exit it.
4. The loop continues to iterate as long as the condition remains true. Once the condition becomes false, the loop terminates, and the program continues with the statement after the loop.

Example code which contains while loop:

```
count_emp = 0;                /* no employees processed yet      */
while (count_emp < 7) {       /* test value of count_emp         */
    printf("Hours> ");
    scanf("%d", &hours);
    printf("Rate> ");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n", pay);
    count_emp = count_emp + 1; /* increment count_emp            */
}
printf("\nAll employees processed\n");
```

This is how the while loop works if we convert it to a flow chart.



The For statement

A for loop is a fundamental control structure in programming that allows you to execute a specific block of code repeatedly for a known number of iterations. It's particularly useful when you need to perform a task a fixed number of times or when you have a clear starting point, ending point, and step size. The for loop provides a compact and organized way to manage iteration.

The general syntax of a for loop in most programming languages, including C, looks like this:

```
for (initialization; condition; update) {
    // Loop body: code to be executed repeatedly until the
condition is false
}
```

Example code which contains for loop:

```
/* Process payroll for all employees */
total_pay = 0.0;
for (count_emp = 0;                    /* initialization              */
     count_emp < number_emp;           /* loop repetition condition   */
     count_emp += 1) {                 /* update                      */
    printf("Hours> ");
    scanf("%lf", &hours);
    printf("Rate > $");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n\n", pay);
    total_pay = total_pay + pay;
}
printf("All employees processed\n");
printf("Total payroll is $%8.2f\n", total_pay);
```

The **for** loop consists of three main components:

1. **Initialization:** You initialize a loop control variable before entering the loop. This variable is typically used to track the progress of the loop.
2. **Condition:** The loop condition is an expression that is evaluated before each iteration. If the condition is true, the loop body is executed; if the condition is false, the loop terminates, and control is transferred to the next statement after the loop.
3. **Update:** After each iteration of the loop body, the loop control variable is updated based on the defined increment or decrement. This update ensures that the loop progresses towards the termination condition.

Here's a simple step-by-step explanation of how a **for** loop works:

1. The **for** loop starts with the initialization of the loop control variable. This variable is typically used to track the current state of the loop.
2. The loop enters a loop cycle, which includes three main phases: evaluation of the condition, execution of the loop body, and update of the loop control variable.
3. The condition is evaluated. If the condition is true, the loop body is executed; if false, the loop terminates, and execution continues with the statement following the loop.
4. After executing the loop body, the loop control variable is updated based on the specified increment or decrement. This step ensures that the loop makes progress toward the termination condition.
5. The loop repeats steps 3 and 4 until the condition becomes false. Once the condition is no longer satisfied, the loop terminates, and the program proceeds to the statement after the loop.

The do - while statement

A do-while loop is a type of loop in programming that is used to repeatedly execute a block of code while a given condition remains true. What sets the do-while loop apart from other loops like the while and for loops is that the condition is evaluated after the loop body has been executed. This ensures that the loop body is executed at least once, regardless of whether the condition is initially true or false.

The general syntax of a do-while loop in most programming languages, including C, looks like this:

```
do {
    // Loop body: code to be executed at least once, and then
repeatedly as long as the condition is true
} while (condition);
```

The do-while loop consists of two main components:

1. Loop Body: The block of code within the loop body contains the instructions you want to repeat. This code is executed during each iteration of the loop.
2. Condition: The loop condition is an expression that is evaluated after the loop body has been executed. If the condition is true, the loop body is executed again; if the condition is false, the loop terminates, and control is transferred to the next statement after the loop.

Here's a simple step-by-step explanation of how a do-while loop works:

1. The loop starts with the execution of the loop body. This ensures that the loop body is executed at least once, regardless of the condition's initial value.
2. After executing the loop body, the loop enters the evaluation phase. The condition is evaluated.
3. If the condition is true, the loop body is executed again, and the cycle repeats. If the condition is false, the loop terminates, and execution continues with the statement after the loop.
4. The loop continues to repeat steps 2 and 3 until the condition becomes false. Since the condition is evaluated after each iteration, there is always at least one execution of the loop body.

Example code for do-while loop

```
#include <stdio.h>

int main() {
   int number;

   do {
      printf("Please enter a number greater than 10: ");
      scanf("%d", &number);

      if (number <= 10) {
         printf("The entered number is not greater than 10. Please try again.\n");
      }
   } while (number <= 10);


   printf("You entered a valid number: %d\n", number);

   return 0;
}
```

Note: For more details on repetition go through the Chapter 5 - Repetition and Loop Statements of the reference book "PROBLEM SOLVING AND PROGRAM DESIGN in C" 7th edition

Task 01

Write a program to process weekly employee time cards for all employees of an organization. Each employee will have three data items: an identification number, the hourly wage rate, and the number of hours worked during a given week. Each employee is to be paid time and a half for all hours worked over 40. A tax amount of 3.625% of gross salary will be deducted. The program output should show the employee's number and net pay. Display the total payroll and the average amount paid at the end of the run.

Task 02

The factorial of a number is the product of all positive integers less than or equal to that number. For instance, the factorial of 5 (denoted as 5!) is calculated as $5 \times 4 \times 3 \times 2 \times 1$, which equals 120. Your program should follow these steps:

1. Prompt the user to enter a positive integer for which they want to calculate the factorial.
2. Read the input integer from the user.
3. Implement a function, either using a loop, to calculate the factorial of the input number.
4. Display the calculated factorial on the screen.

Write the C program that accomplishes the above steps.

Task 03

The pressure of a gas changes as the volume and temperature of the gas vary. Write a program that uses the Van der Waals equation of state for a gas,

$$\left(P + \frac{an^2}{V^2}\right)(V - bn) = nRT$$

to create a file that displays in tabular form the relationship between the pressure and the volume of n moles of carbon dioxide at a constant absolute temperature (T). P is the pressure in atmospheres, and V is the volume in liters. The Van der Waals constants for carbon dioxide are a = 3.592 $L^2 \cdot atm/mol^2$ and b = 0.0427 L/mol. Use 0.08206 L · atm/mol · K for the gas constant R. Inputs to the program include n, the Kelvin temperature, the initial and final volumes in milliliters, and the volume increment between lines of the table. Your program will output a table that varies the volume of the gas from the initial to the final volume in steps prescribed by the volume increment. Here is a sample run:

```
Please enter at the prompts the number of moles of carbon
dioxide, the absolute temperature, the initial volume in
milliliters, the final volume, and the increment volume
between lines of the table.

Quantity of carbon dioxide (moles)> 0.02
Temperature (kelvin)> 300
Initial volume (milliliters)> 400
Final volume (milliliters)> 600
Volume increment (milliliters)> 50
```

Output File

```
0.0200 moles of carbon dioxide at 300 kelvin

Volume (ml)    Pressure (atm)

    400            1.2246
    450            1.0891
    500            0.9807
    550            0.8918
    600            0.8178
```

Task 04

The rate of decay of a radioactive isotope is given in terms of its half-life H, the time lapse required for the isotope to decay to one-half of its original mass. The isotope cobalt-60 (60 Co) has a half-life of 5.272 years. Compute and print in table form the amount of this isotope that remains after each year for 5 years, given the initial presence of an amount in grams. The value of amount should be provided interactively. The amount of 60 Co remaining can be computed by using the following formula:

$$r = amount \times C^{(y/H)}$$

Where amount is the initial amount in grams, C is expressed as $e^{-0.693}$ (e = 2.71828), y is the number of years elapsed, and H is the half-life of the isotope in years.

Task 05

A concrete channel to bring water to Crystal Lake is being designed. It will have vertical walls and be 15 feet wide. It will be 10 feet deep, have a slope of .0015 feet/foot, and a roughness coefficient of .014. How deep will the water be when 1,000 cubic feet per second is flowing through the channel? To solve this problem, we can use Manning's equation

$$Q = \frac{1.486}{N} A R^{2/3} S^{1/2}$$

where Q is the flow of water (cubic feet per second), N is the roughness coefficient (unitless), A is the area (square feet), S is the slope (feet/foot), and R is the hydraulic radius (feet). The hydraulic radius is the cross-sectional area divided by the wetted perimeter. For square channels like the one in this example,

Hydraulic radius = depth * width / (2.0 * depth + width)

To solve this problem, design a program that allows the user to guess a depth and then calculates the corresponding flow. If the flow is too little, the user Programming Projects 309 should guess a depth a little higher; if the flow is too high, the user should guess a depth a little lower. The guessing is repeated until the computed flow is within 0.1% of the flow desired.

To help the user make an initial guess, the program should display the flow for half the channel depth. Note the example run:

```
At a depth of 5.0000 feet, the flow is 641.3255 cubic
feet per second.

Enter your initial guess for the channel depth
when the flow is 1000.0000 cubic feet per second
Enter guess> 6.0

Depth: 6.0000 Flow: 825.5906 cfs Target: 1000.0000 cfs
Difference: 174.4094 Error: 17.4409 percent
Enter guess> 7.0

Depth: 7.0000 Flow: 1017.7784 cfs Target: 1000.0000 cfs
Difference: -17.7784 Error: -1.7778 percent

Enter guess> 6.8
```

References

[1] J. R. Hanly and E. B. Koffman, *Problem Solving and Program Design in C (Seventh Edition)*. PEARSON