Loops and Iteration

CS10003 PROGRAMMING AND DATA STRUCTURES
Repeated Execution of Groups of Instructions

**Loop**: A group of instructions that is executed repeatedly while some condition remains true. Each execution of that group of instructions is called an *iteration* of the loop.

The group of instructions being repeated is called the *body* of the loop.

**How many iterations:**

a) **Counter-controlled**: A counter steps through a set of values. The loop runs once for each value.

b) **Achievement of some condition**: A loop is repeated until some desired condition (like a mathematical equality of inequality) is established.

c) **Input-controlled**: A loop is repeated until the user enters some special value indicating end of input.

A loop may have infinite number of iterations (e.g. the desired condition is never satisfied, or the user never enters a terminating value).
Counter Controlled Loop

Read 5 integers and display the value of their sum.

In this example, the loop ‘counts up’ to a maximum value. We can also have loops that ‘count down’.
Given an exam marks as input, display the appropriate message based on the rules below:

- If marks is greater than 49, display “PASS”, otherwise display “FAIL”

- However, for input outside the 0-100 range, display “WRONG INPUT” and prompt the user to input again until a valid input is entered
Condition-Controlled Loop

- `input m`
- `m < 0 || m > 100`
- `m > 49`

**Conditions:**
- If `m < 0` or `m > 100`, output "WRONG INPUT".
- If `m > 49`, output "PASS".
- If none of the above conditions are true, output "FAIL".

**Flow Diagram:**
- Start with input `m`.
- Check if `m < 0` or `m > 100`.
- If true, go to "WRONG INPUT".
- If false, check if `m > 49`.
  - If true, go to "PASS".
  - If false, go to "FAIL".

**Note:**
- "PASS" represents passing the condition-controlled loop.
- "FAIL" represents failing the condition-controlled loop.
- "WRONG INPUT" indicates an input error.
Input-Controlled Loop

Read a sequence of non-negative integers from the user, and display the sum of these integers.

A negative input indicates the end of input process.
When is the Continuation Condition Checked?

Pre-Test Loop

- Continuation Condition
  - True: Loop Body
  - False

Post-Test Loop

- Loop Body
  - True: Continuation Condition
  - False
**while Statement**

The “while” statement is used to carry out looping operations, in which a group of statements is executed repeatedly, as long as some condition remains satisfied.

```c
while (condition) {
    statement_1;
    ...
    statement_N;
}
```

The condition to be tested is any expression enclosed in parentheses. The condition is evaluated, and if its value is true (non-zero), the loop-body is executed. Then the condition is evaluated again and the same sequence of operations repeats. The loop **terminates** when the condition evaluates to 0.

**Note:** The while-loop will not be entered at all, if the loop-control condition evaluates to false (zero) even before the first iteration.

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
Example

```c
int main()
{
    int i=1, n;
    scanf("%d", &n);
    while (i <= n) {
        printf ("Line no : %d\n",i);
        i = i + 1;
    }
}
```

Output

```
4
Line no : 1
Line no : 2
Line no : 3
Line no : 4
```
Sum of first $N$ natural numbers

```
int main () {
    int N, count, sum;
    scanf (“%d”, &N);
    sum = 0; count = 1;
    while (count <= N) {
        sum = sum + count;
        count = count + 1;
    }
    printf ("Sum = %d\n", sum);
    return 0;
}
```

Output

```
9
Sum of first 9 numbers = 45
```
Double your money

Suppose your Rs 10000 is earning interest at 1% per month. How many months until you double your money?

```c
int main() {
    double my_money = 10000.0;
    int n=0;
    while (my_money < 20000.0) {
        my_money = my_money * 1.01;
        n++;
    }
    printf ("My money will double in %d months.", n);
    return 0;
}
```
int main() {
    double max = 0.0, next;
    printf ("Enter positive numbers only, end with 0 or a negative number\n");
    scanf("%lf", &next);
    while (next > 0) {
        if (next > max)  max = next;
        scanf("%lf", &next);
    }
    printf ("The maximum number is %lf\n", max);
    return 0;
}
Find the sum of digits of a number

```c
int main()
{
    int n, sum=0;
    scanf("%d", &n);
    while (n != 0) {
        sum = sum + (n % 10);
        n = n / 10;
    }
    printf("The sum of digits is %d\n", sum);
    return 0;
}
```

Output

573254
The sum of digits is 26

Note how to separate out the digits of an integer number
Compute GCD of two numbers

```c
int main() {
    int A, B, temp;
    scanf ("%d %d", &A, &B);
    if (A > B) {
        temp = A;  A = B;  B = temp;
    }
    while ((B % A) != 0) {
        temp = B % A;
        B = A;
        A = temp;
    }
    printf ("The GCD is %d", A);
    return 0;
}
```

Initial: $A=12, B=45$
Iteration 1: $temp=9, B=12, A=9$
Iteration 2: $temp=3, B=9, A=3$

$B \% A = 0 \quad \therefore \text{GCD is 3}$
Which expressions decide how long a while loop will run?

Next, we will see another way of writing the same loop, where these 3 parts will be written together.
Looping: `for` Statement

Most commonly used looping structure in C

```c
for ( expr1; expr2; expr3)
    statement;
```

- **expr1** *(init)*: initialize parameters
- **expr2** *(test)*: test condition, loop continues if expression is non-zero
- **expr3** *(update)*: used to alter the value of the parameters after each iteration
- **statement** *(body)*: body of loop
Example: Computing Factorial

```c
int main() {
    int N, count, prod;
    scanf("%d", &N);
    prod = 1;
    for (count = 1; count <= N; ++count)
        prod = prod * count;
    printf("Factorial = %d\n", prod);
    return 0;
}
```

Output

7
Factorial = 5040
Equivalence of **for** and **while**

```
for ( expr1; expr2; expr3)
    statement;
```

Same as

```
expr1;
while (expr2) {
    statement
    expr3;
}
```
Sum of first \( N \) natural numbers

```c
int main () {
    int N, count, sum;
    scanf ("%d", &N);

    sum = 0;
    count = 1;
    while (count <= N) {
        sum = sum + count;
        count++;
    }

    printf ("Sum = %d\n", sum);
    return 0;
}
```
Example: Computing $e^x$ series up to $N$ terms \((1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots)\)
Some observations on for

Initialization, loop-continuation test, and update can contain arithmetic expressions

```
for ( k = x;   k <= 4 * x * y;   k += y / x )
```

Update may be negative (decrement)

```
for (digit = 9; digit >= 0; --digit)
```

If loop continuation test is initially 0 (false)

- Body of for structure not performed
  - No statement executed
- Program proceeds with statement after for structure
- This is exactly the same as in the case of while loop
Looping: do-while statement

```c
main () {
    int digit=0;
    do
        printf("%d\n",digit++);
    while (digit <= 9) ;
}
```
Example

Problem: Prompt user to input “month” value, keep prompting until a correct value of month is given as input

do {
    printf ("Please input month {1-12}"");
    scanf ("%d", &month);
} while ((month < 1) || (month > 12));
int main () {
    char echo;
    do {
        scanf ("%c", &echo);
        printf ("%c", echo);
    } while (echo != '\n') ;
    return 0;
}

Output
This is a test line
This is a test line
Specifying “Infinite Loop”

while (1) {
    statements
}

for (; ;) {
    statements
}

do {
    statements
} while (1);
The break Statement

Break out of the loop {}

- can use with
  - while
  - do while
  - for
  - switch
- does not work with
  - if
  - if-else

Causes immediate exit from a while, do/while, for or switch structure.
Program execution continues with the first statement after the structure.
Example: *Find smallest n such that n! exceeds 100*

```c
#include <stdio.h>
int main() {
    int fact, i;
    fact = 1;  i = 1;
    while (1) {  /* run loop -break when fact >100*/
        fact = fact * i;
        if ( fact > 100 ) {
            printf("Factorial of %d above 100", i);
            break;  /* break out of the while loop */
        }
        i ++ ;
    }
    return 0;
}
```
Test if a number is prime or not

```c
int main() {
    int n, i=2;
    double limit;
    scanf ("%d", &n);
    limit = sqrt(n);
    for (i = 2, i <= limit; i++)  {
        if (n % i == 0)  {
            printf ("%d is not a prime \n", n);
            break;
        }
    }
    if (i > limit) printf ("%d is a prime \n", n);
    return 0;
}
```
int main() {
    int n, i = 2, found = 0;
    double limit;
    scanf ("%d", &n);
    limit = sqrt(n);
    while (i <= limit) {
        if (n % i == 0) {
            found = 1; break;
        }
        i = i + 1;
    }
    if (found == 0) printf ("%d is a prime \n", n);
    else printf ("%d is not a prime \n", n);
    return 0;
}
The continue Statement

Skips the remaining statements in the body of a \textit{while}, \textit{for} or \textit{do/while} loop in the ongoing iteration.

- Proceeds with the next iteration of the loop.

\textbf{while and do/while loop}

- Loop-continuation test is evaluated immediately after the continue statement is executed.

\textbf{for loop}

- $expression3$ is evaluated, then $expression2$ is evaluated.
Example with **break** and **continue**:

Add positive numbers until a 0 is typed, but ignore any negative numbers typed

```c
int main() {
    int sum = 0, next;
    while (1) {
        scanf("%d", &next);
        if (next < 0) continue;
        if (next == 0) break;
        sum = sum + next;
    }
    printf ("Sum = %d\n", sum) ;
    return 0;
}
```

Output

```
10
-20
30
40
-5
10
0
Sum = 90
```
Some Loop Pitfalls

while (sum <= NUM) ;
  sum = sum+2;

for (i=0; i<=NUM; ++i);
  sum = sum+i;

for (i=1; i!=10; i=i+2)
  sum = sum+i;
Nested Loops: Printing a 2-D Figure

How would you print the following diagram?

```
* * * * *
* * * * *
* * * * *
```

repeat 3 times

print a row of 5 *'s

repeat 5 times

print *
#define ROWS 3
#define COLS 5
...
row=1;
while (row <= ROWS) {
    /* print a row of 5 *’s */
    ...
    printf(“\n”);
    row++;
}
col=1;
while (col <= COLS) {
    printf (“* ”);
    col++;
}
Nested Loops

const int ROWS = 3;
const int COLS = 5;
...
row = 1;
while (row <= ROWS) {
    /* print a row of 5 *'s */
    col = 1;
    while (col <= COLS) {
        printf ("* ");
        col++;
    }
    printf("\n");
    ++row;
}

For every iteration of the outer loop, the inner loop runs its entire length
2-D Figure: with for loop

Print

* * * * *
* * * * *
* * * * *

```c
const int ROWS = 3;
const int COLS = 5;
....
for (row=1; row<=ROWS; ++row) {
    for (col=1; col<=COLS; ++col) {
        printf("* ");
    }
    printf("\n");
}
```
const int ROWS = 5;

....

int row, col;

for (row=1; row<=ROWS; ++row) {
    for (col=1; col<=row; ++col) {
        printf("* ");
    }
    printf("\n");
}
const int ROWS = 5;
....

int row, col;

for (row=0; row<ROWS; ++row) {
    for (col=1; col<=row; ++col)
        printf(" ");
    for (col=1; col<=ROWS-row; ++col)
        printf("* ");
    printf ("\n");
}
break and continue with nested loops

For nested loops, break and continue are matched with the nearest loops (for, while, do-while)

Example:

```javascript
while (i < n) {
    for (k=1; k < m; ++k) {
        if (k % i == 0) break;
    }
    i = i + 1;
}
```

Breaks here
int main()
{
    int low, high, desired, i, flag = 0;
    scanf("%d%d%d", &low, &high, &desired);
    i = low;
    while (i < high) {
        for (j = i+1; j <= high; ++j) {
            if (j % i == desired) {
                flag = 1;
                break;
            }
        }
        if (flag == 1) break;
        i = i + 1;
    }
    return 0;
}
The comma operator

Separates expressions.

Syntax:

- expr-1, expr-2, ..., expr-n

where expr-1, expr-2, ..., expr-n are all expressions.

Is itself an expression, which evaluates to the value of the last expression in sequence.

- Useful in for loops.

Example: \(a=1, \ x=x+2, \ d=5\) \(\rightarrow\) evaluates to 5
We can give several expressions separated by commas in place of `expr1` and `expr3` in a for loop to do multiple assignments.

Example:

```c
for (fact=1,i=1; i<=10; ++i)
    fact = fact * i;

for (sum=0,i=1; i<=N; ++i)
    sum = sum + i * i;
```
Practice Problems
Practice Problems (do each with both for and while loops separately)

1. Read in an integer N. Then print the sum of the squares of the first N natural numbers.
2. Read in an integer N. Then read in N numbers and print their maximum and second maximum (do not use arrays even if you know it).
3. Read in an integer N. Then read in N numbers and print the number of integers between 0 and 10 (including both), between 11 and 20, and > 20. (do not use arrays even if you know it).
4. Repeat 3, but this time print the average of the numbers in each range.
5. Read in a positive integer N. If the user enters a negative integer or 0, print a message asking the user to enter the integer again. When the user enters a positive integer N finally, find the sum of the logarithmic series \(\log_e(1+x)\) up to the first N terms.
6. Read in an integer N. Then read in integers, and find the sum of the first N positive integers read. Ignore any negative integers or 0 read (so you may actually read in more than N integers, just find the sum with only the positive integers and stop when N such positive integers are read).
7. Read in characters until the ‘\n’ character is typed. Count and print the number of lowercase letters, the number of uppercase letters, and the number of digits entered.