Loops in C++

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

1 Introduction to Loops

In programming, we often need to repeat a block of code multiple times. Instead of writing the same code over and over, we use **loops**. A loop is a control structure that allows a program to execute a block of code repeatedly as long as a specified condition remains true. This repetition continues until the condition becomes false, at which point the loop terminates and the program continues with the next statement after the loop.

C++ provides three main types of loops: `while`, `do-while`, and `for` loops. Each type has its own characteristics and is suited for different situations. We can also place loops inside other loops, creating **nested loops** for more complex repetitive tasks. In this section, we will explore each type of loop in detail, understand when to use each one, and learn about common mistakes to avoid.

2 The `while` Loop

2.1 Basic Structure and Concept

The `while` loop is the simplest form of loop in C++. It repeatedly executes a block of code as long as a given condition is true. The general syntax is:

Syntax:

```
while (condition) {
    // statements to execute repeatedly
}
```

The `while` loop works as follows:

1. First, the condition is evaluated.
2. If the condition is true, the code inside the loop body (the statements within the braces) is executed.
3. After executing the loop body, the program goes back to step 1 and checks the condition again.
4. This process continues until the condition becomes false.
5. When the condition is false, the loop terminates and the program continues with the statement after the closing brace.

An important characteristic of the `while` loop is that the condition is checked **before** the loop body executes. This means if the condition is false from the beginning, the loop body will never execute (not even once).

2.2 Example: Counting with `while`

Let's start with a simple example that prints numbers from 1 to 5:

Example: Basic while loop - counting

```
#include <iostream>
using namespace std;

int main() {
    int counter = 1;  // Initialize counter variable

    while (counter <= 5) {
        cout << "Count: " << counter << endl;
        counter++;  // Increment counter
    }

    cout << "Loop finished!" << endl;
    return 0;
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Loop finished!
```

Let's trace through this example:

- We start with `counter = 1`.
- First iteration: Is `1 <= 5`? Yes, so print "Count: 1" and increment counter to 2.
- Second iteration: Is `2 <= 5`? Yes, so print "Count: 2" and increment counter to 3.
- This continues until counter becomes 6.
- When `counter = 6`, the condition `6 <= 5` is false, so the loop terminates.
- The program then prints "Loop finished!" and ends.

2.3 Example: User Input Validation

The `while` loop is excellent for input validation, where we keep asking for input until the user provides valid data:

Example: Input validation with while loop

```
#include <iostream>
using namespace std;

int main() {
    int user_age;

    cout << "Please enter your age (must be between 1 and 120): ";
    cin >> user_age;

    while (user_age < 1 || user_age > 120) {
        cout << "Invalid age! Please enter a valid age (1-120): ";
        cin >> user_age;
    }

    cout << "Thank you! Your age is " << user_age << endl;
    return 0;
}
```

Output:

```
Please enter your age (must be between 1 and 120): -5
Invalid age! Please enter a valid age (1-120): 150
Invalid age! Please enter a valid age (1-120): 25
Thank you! Your age is 25
```

This example demonstrates:

- The loop continues as long as the input is invalid (less than 1 or greater than 120).
- Each time through the loop, we prompt for new input.
- Once valid input is received, the condition becomes false and the loop exits.
- This pattern ensures we don't proceed with invalid data.

2.4 Example: Sum of Numbers

Here's an example that calculates the sum of positive numbers entered by the user, stopping when they enter 0:

Example: Accumulating sum with while loop

```
#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    cout << "Enter positive numbers to sum (enter 0 to stop):" <<
       endl;
    cout << "Enter a number: ";
    cin >> number;

    while (number != 0) {
        sum = sum + number;
        cout << "Current sum: " << sum << endl;
        cout << "Enter a number: ";
        cin >> number;
    }

    cout << "Final sum: " << sum << endl;
    return 0;
}
```

Output:

```
Enter positive numbers to sum (enter 0 to stop):
Enter a number: 5
Current sum: 5
Enter a number: 10
Current sum: 15
Enter a number: 3
Current sum: 18
Enter a number: 0
Final sum: 18
```

2.5 Common Pitfalls with `while` Loops

- **Infinite loops:** If the condition never becomes false, the loop will run forever. This often happens when you forget to update the variable that controls the loop condition:

```
int x = 1;
while (x <= 5) {
    cout << x << endl;
    // Forgot to increment x - infinite loop!
}
```

- **Off-by-one errors:** Using the wrong comparison operator can cause the loop to execute one too many or one too few times:

```
int i = 1;
while (i < 5) {  // This prints 1,2,3,4 (not 5)
    cout << i << endl;
    i++;
}
// vs.
while (i <= 5) {  // This prints 1,2,3,4,5
    cout << i << endl;
    i++;
}
```

- **Uninitialized variables:** Using a variable in the condition before initializing it leads to undefined behavior:

```
int count;  // Not initialized!
while (count < 10) {  // Undefined behavior
    // ...
}
```

3 The `do-while` Loop

3.1 Basic Structure and Key Difference

The `do-while` loop is similar to the `while` loop, but with one crucial difference: the condition is checked **after** the loop body executes. This guarantees that the loop body will execute **at least once**, even if the condition is false from the start.

The syntax is:

Syntax:

```
do {
    // statements to execute
} while (condition);
```

Note the semicolon after the closing parenthesis - this is required and often forgotten by beginners.

The `do-while` loop works as follows:

1. Execute the statements in the loop body.
2. Check the condition.
3. If the condition is true, go back to step 1.
4. If the condition is false, exit the loop.

3.2 Example: Menu-Driven Program

The `do-while` loop is perfect for menu systems where you want to show the menu at least once:

Example: Menu system with do-while loop

```
#include <iostream>
using namespace std;

int main() {
    int choice;

    do {
        cout << "\n=== Calculator Menu ===" << endl;
        cout << "1. Add" << endl;
        cout << "2. Subtract" << endl;
        cout << "3. Multiply" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice (1-4): ";
        cin >> choice;

        switch(choice) {
            case 1:
                cout << "You selected Addition" << endl;
                break;
            case 2:
                cout << "You selected Subtraction" << endl;
                break;
            case 3:
                cout << "You selected Multiplication" << endl;
                break;
            case 4:
                cout << "Exiting program..." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 4);

    cout << "Thank you for using the calculator!" << endl;
    return 0;
}
```

Output:

```
=== Calculator Menu ===
1. Add
2. Subtract
3. Multiply
4. Exit
Enter your choice (1-4): 1
You selected Addition

=== Calculator Menu ===
1. Add
2. Subtract
3. Multiply
4. Exit
Enter your choice (1-4): 4
Exiting program...
Thank you for using the calculator!
```

This example shows why `do-while` is ideal for menus:

- The menu is displayed at least once, regardless of any condition.
- After each action, the menu reappears unless the user chooses to exit.
- The condition `choice != 4` is checked after processing the user's input.

3.3 Example: Password Verification

Here's another practical use of `do-while` for password verification:

Example: Password verification with do-while

```
#include <iostream>
using namespace std;

int main() {
    int password;
    int correct_password = 1234;
    int attempts = 0;

    do {
        cout << "Enter password: ";
        cin >> password;
        attempts++;

        if (password != correct_password) {
            cout << "Incorrect password! Try again." << endl;
        }
    } while (password != correct_password && attempts < 3);

    if (password == correct_password) {
        cout << "Access granted!" << endl;
    } else {
        cout << "Too many attempts. Access denied!" << endl;
    }

    return 0;
}
```

Output:

```
Enter password: 1111
Incorrect password! Try again.
Enter password: 1234
Access granted!
```

3.4 Comparing `while` and `do-while`

Here's a direct comparison to illustrate the key difference:

Example: while vs do-while comparison

```
#include <iostream>
using namespace std;

int main() {
    int value = 10;

    cout << "Using while loop with value = 10:" << endl;
    while (value < 5) {
        cout << "This won't print because 10 < 5 is false" << endl;
    }
    cout << "while loop finished (never executed)" << endl;

    cout << "\nUsing do-while loop with value = 10:" << endl;
    do {
        cout << "This prints once even though 10 < 5 is false" <<
           endl;
    } while (value < 5);
    cout << "do-while loop finished (executed once)" << endl;

    return 0;
}
```

Output:

```
Using while loop with value = 10:
while loop finished (never executed)

Using do-while loop with value = 10:
This prints once even though 10 < 5 is false
do-while loop finished (executed once)
```

3.5 Common Pitfalls with `do-while` Loops

- **Forgetting the semicolon:** The `do-while` loop requires a semicolon after the condition:

```
do {
    // code
} while (condition)  // ERROR: missing semicolon!
```

- **Using when zero iterations are needed:** If there's a possibility that the loop shouldn't execute at all, use `while` instead of `do-while`.
- **Complex conditions:** Since the condition is at the bottom, it's easier to forget what variables are being tested. Keep conditions simple and well-documented.

4 The `for` Loop

4.1 Structure and Components

The `for` loop is the most compact loop structure in C++. It's particularly useful when you know in advance how many times you want to repeat something. The `for` loop combines initialization, condition checking, and update in a single line.

The syntax is:

Syntax:

```
for (initialization; condition; update) {
    // statements to execute
}
```

The three parts in parentheses are:

1. **Initialization:** Executed once before the loop starts. Usually initializes a counter variable.
2. **Condition:** Checked before each iteration. If true, the loop body executes; if false, the loop terminates.
3. **Update:** Executed after each iteration of the loop body. Usually increments or decrements the counter.

The `for` loop executes in this order:

1. Initialization (once only)
2. Check condition
3. If condition is true, execute loop body
4. Execute update
5. Go back to step 2

4.2 Example: Basic Counting

Here's the classic example of counting from 1 to 10:

Example: Basic for loop counting

```
#include <iostream>
using namespace std;

int main() {
    cout << "Counting from 1 to 10:" << endl;

    for (int i = 1; i <= 10; i++) {
        cout << i << " ";
    }

    cout << endl << "Loop finished!" << endl;
    return 0;
}
```

Output:

```
Counting from 1 to 10:
1 2 3 4 5 6 7 8 9 10
Loop finished!
```

Let's trace through this:

- `int i = 1`: Initialize `i` to 1 (happens once)
- `i <= 10`: Check if 1 ¡= 10 (true)
- Execute body: Print "1 "
- `i++`: Increment `i` to 2
- `i <= 10`: Check if 2 ¡= 10 (true)
- Execute body: Print "2 "
- This continues until `i` becomes 11
- `i <= 10`: Check if 11 ¡= 10 (false)
- Loop terminates

4.3 Example: Counting Backwards

The `for` loop can count in any direction:

Example: Counting backwards with for loop

```
#include <iostream>
using namespace std;

int main() {
    cout << "Countdown:" << endl;

    for (int count = 10; count >= 0; count--) {
        cout << count << "... ";
    }

    cout << "Blast off!" << endl;
    return 0;
}
```

Output:

```
Countdown:
10... 9... 8... 7... 6... 5... 4... 3... 2... 1... 0... Blast off!
```

4.4 Example: Sum and Average

Here's a practical example that calculates the sum and average of numbers:

Example: Calculate sum and average with for loop

```
#include <iostream>
using namespace std;

int main() {
    int num_values;
    int sum = 0;
    int value;

    cout << "How many numbers do you want to enter? ";
    cin >> num_values;

    for (int i = 1; i <= num_values; i++) {
        cout << "Enter number " << i << ": ";
        cin >> value;
        sum = sum + value;
    }

    double average = static_cast<double>(sum) / num_values;

    cout << "Sum: " << sum << endl;
    cout << "Average: " << average << endl;

    return 0;
}
```

Output:

```
How many numbers do you want to enter? 3
Enter number 1: 10
Enter number 2: 20
Enter number 3: 30
Sum: 60
Average: 20
```

4.5 Example: Multiplication Table

The `for` loop is perfect for generating tables:

Example: Multiplication table with for loop

```
#include <iostream>
using namespace std;

int main() {
    int table_number;

    cout << "Enter a number for multiplication table: ";
    cin >> table_number;

    cout << "\nMultiplication table for " << table_number << ":" <<
       endl;

    for (int multiplier = 1; multiplier <= 10; multiplier++) {
        int result = table_number * multiplier;
        cout << table_number << " x " << multiplier << " = " <<
           result << endl;
    }

    return 0;
}
```

Output:

```
Enter a number for multiplication table: 7

Multiplication table for 7:
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

4.6 Variations of `for` Loop

The `for` loop is very flexible. Any of its three parts can be omitted or modified:

Example: for loop variations

```
#include <iostream>
using namespace std;

int main() {
    // Multiple variables in initialization and update
    cout << "Two variables:" << endl;
    for (int i = 0, j = 10; i < 5; i++, j--) {
        cout << "i=" << i << ", j=" << j << endl;
    }

    // Omitting initialization (variable declared outside)
    cout << "\nExternal initialization:" << endl;
    int k = 0;
    for (; k < 3; k++) {
        cout << "k=" << k << endl;
    }

    // Using different increment
    cout << "\nIncrement by 2:" << endl;
    for (int m = 0; m <= 10; m += 2) {
        cout << m << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Two variables:
i=0, j=10
i=1, j=9
i=2, j=8
i=3, j=7
i=4, j=6

External initialization:
k=0
k=1
k=2

Increment by 2:
0 2 4 6 8 10
```

4.7 Common Pitfalls with `for` Loops

- **Incorrect loop bounds:** Off-by-one errors are common:

```
// To print 1 to 10:
for (int i = 1; i < 10; i++)   // Wrong: prints 1-9
for (int i = 1; i <= 10; i++)  // Correct: prints 1-10
```

- **Modifying the loop variable inside the loop:** This can lead to unexpected behavior:

```
for (int i = 0; i < 10; i++) {
    cout << i << endl;
    i = i + 2;  // Dangerous: modifying i inside loop
}
```

- **Infinite loops from wrong update:** Forgetting to update or updating in the wrong direction:

```
for (int i = 10; i > 0; i++) {  // Should be i--
    cout << i << endl;  // Infinite loop!
}
```

- **Scope of loop variable:** Variables declared in the initialization are only available inside the loop:

```
for (int i = 0; i < 5; i++) {
    // i is available here
}
// i is NOT available here (out of scope)
```


5 Nested Loops

5.1 Concept and Structure

A nested loop is a loop inside another loop. The inner loop completes all its iterations for each single iteration of the outer loop. This creates a multiplication effect: if the outer loop runs `m` times and the inner loop runs `n` times, the inner loop body executes `m × n` times total.

Nested loops are commonly used for:

- Working with two-dimensional data (like matrices or tables)
- Generating patterns
- Comparing all pairs of elements
- Processing grids or coordinate systems

5.2 Example: Rectangle Pattern

Let's start with a simple example that prints a rectangle of stars:

Example: Rectangle pattern with nested loops

```
#include <iostream>
using namespace std;

int main() {
    int rows, columns;

    cout << "Enter number of rows: ";
    cin >> rows;
    cout << "Enter number of columns: ";
    cin >> columns;

    for (int i = 1; i <= rows; i++) {           // Outer loop for
        rows
        for (int j = 1; j <= columns; j++) {    // Inner loop for
            columns
            cout << "* ";
        }
        cout << endl;  // New line after each row
    }

    return 0;
}
```

Output:

```
Enter number of rows: 3
Enter number of columns: 5
* * * * *
* * * * *
* * * * *
```

How this works:

- The outer loop controls the rows (runs 3 times)
- For each row, the inner loop prints 5 stars
- After the inner loop completes, we print a newline to start the next row
- Total stars printed: $3 \times 5 = 15$

5.3 Example: Number Triangle

Here's a pattern where each row has an increasing number of elements:

Example: Number triangle with nested loops

```
#include <iostream>
using namespace std;

int main() {
    int height;

    cout << "Enter triangle height: ";
    cin >> height;

    for (int row = 1; row <= height; row++) {
        for (int col = 1; col <= row; col++) {
            cout << col << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Enter triangle height: 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Notice how the inner loop's condition depends on the outer loop's variable (`col <= row`). This creates the triangular shape.

5.4 Example: Multiplication Table Grid

A practical use of nested loops is creating a multiplication table:

Example: Full multiplication table with nested loops

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int table_size;

    cout << "Enter size of multiplication table: ";
    cin >> table_size;

    // Print header row
    cout << "   ";
    for (int i = 1; i <= table_size; i++) {
        cout << setw(4) << i;
    }
    cout << endl;

    // Print separator
    cout << "   ";
    for (int i = 1; i <= table_size; i++) {
        cout << "----";
    }
    cout << endl;

    // Print table with nested loops
    for (int row = 1; row <= table_size; row++) {
        cout << setw(2) << row << "|";
        for (int col = 1; col <= table_size; col++) {
            cout << setw(4) << (row * col);
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Enter size of multiplication table: 5
     1 2 3 4 5
  --------------------
 1| 1 2 3 4 5
 2| 2 4 6 8 10
 3| 3 6 9 12 15
 4| 4 8 12 16 20
 5| 5 10 15 20 25
```

5.5 Example: Finding Prime Numbers

Here's a more complex example using nested loops to find prime numbers:

Example: Finding prime numbers with nested loops

```
#include <iostream>
using namespace std;

int main() {
    int limit;

    cout << "Find prime numbers up to: ";
    cin >> limit;

    cout << "Prime numbers from 2 to " << limit << ":" << endl;

    for (int num = 2; num <= limit; num++) {
        bool is_prime = true;

        // Check if num is divisible by any number from 2 to num-1
        for (int divisor = 2; divisor < num; divisor++) {
            if (num % divisor == 0) {
                is_prime = false;
                break;  // No need to check further
            }
        }

        if (is_prime) {
            cout << num << " ";
        }
    }

    cout << endl;
    return 0;
}
```

Output:

```
Find prime numbers up to: 20
Prime numbers from 2 to 20:
2 3 5 7 11 13 17 19
```

5.6 Different Types of Loops Can Be Nested

You can nest any type of loop within any other type:

Example: Mixing different loop types

```
#include <iostream>
using namespace std;

int main() {
    int outer_count = 1;

    // while loop with for loop inside
    while (outer_count <= 3) {
        cout << "Group " << outer_count << ": ";
        for (int inner = 1; inner <= 4; inner++) {
            cout << inner << " ";
        }
        cout << endl;
        outer_count++;
    }

    cout << endl;

    // for loop with do-while inside
    for (int i = 1; i <= 2; i++) {
        cout << "Set " << i << ": ";
        int j = 1;
        do {
            cout << (char)('A' + j - 1) << " ";
            j++;
        } while (j <= 3);
        cout << endl;
    }

    return 0;
}
```

Output:

```
Group 1: 1 2 3 4
Group 2: 1 2 3 4
Group 3: 1 2 3 4

Set 1: A B C
Set 2: A B C
```

5.7 Common Pitfalls with Nested Loops

- **Using the same variable name:** Each loop needs its own control variable:

```
// WRONG:
for (int i = 0; i < 5; i++) {
    for (int i = 0; i < 3; i++) {  // ERROR: i already declared
        // ...
    }
```

```
}

// CORRECT:
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 3; j++) {  // Different variable name
        // ...
    }
}
```

- **Performance issues:** Nested loops can be slow. Three nested loops each running 100 times means 1,000,000 iterations!
- **Forgetting which loop you're in:** With deep nesting, it's easy to get confused. Use meaningful variable names and comments:

```
for (int row = 0; row < rows; row++) {        // Process each row
    for (int col = 0; col < cols; col++) {    // Process each column in row
        // Process element at (row, col)
    }
}
```

- **Break and continue confusion:** In nested loops, `break` and `continue` only affect the innermost loop they're in:

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if (j == 2) break;  // Only exits inner loop
        cout << j << " ";
    }
    cout << endl;  // Still executes for each i
}
```


6 Choosing the Right Loop

6.1 When to Use Each Type

Use `while` when:

- You don't know how many iterations you need
- The loop might not execute at all
- You're waiting for a specific condition to become false
- Examples: Reading input until end-of-file, processing user input until they choose to quit, waiting for a calculation to converge

Use `do-while` when:

- The loop body must execute at least once
- You want to perform an action before checking the condition
- Examples: Menu systems, password prompts, input validation where you need to get input first

Use `for` when:

- You know the number of iterations in advance
- You're counting or iterating through a range
- You need a loop counter
- Examples: Processing arrays, counting, generating sequences, iterating a specific number of times

6.2 Loop Control Statements

C++ provides two statements to control loop execution:

The `break` statement:

- Immediately exits the loop (only the innermost loop if nested)
- Execution continues with the statement after the loop

Example: Using break to exit early

```
#include <iostream>
using namespace std;

int main() {
    cout << "Searching for number 7:" << endl;

    for (int i = 1; i <= 10; i++) {
        cout << "Checking " << i << "..." << endl;
        if (i == 7) {
            cout << "Found it!" << endl;
            break;  // Exit the loop
        }
    }

    cout << "Search complete." << endl;
    return 0;
}
```

Output:

```
Searching for number 7:
Checking 1...
Checking 2...
Checking 3...
Checking 4...
Checking 5...
Checking 6...
Checking 7...
Found it!
Search complete.
```

The `continue` statement:

- Skips the rest of the current iteration
- Jumps to the next iteration (checking condition first for `while`/`do-while`, or executing update for `for`)

Example: Using continue to skip iterations

```
#include <iostream>
using namespace std;

int main() {
    cout << "Printing odd numbers from 1 to 10:" << endl;

    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue;  // Skip even numbers
        }
        cout << i << " ";
    }

    cout << endl;
    return 0;
}
```

Output:

```
Printing odd numbers from 1 to 10:
1 3 5 7 9
```

7 Conclusion

Loops are fundamental control structures that allow programs to repeat code efficiently. Understanding when and how to use each type of loop is crucial for writing effective C++ programs:

- `while` loops are best for indefinite iteration based on a condition
- `do-while` loops guarantee at least one execution

- `for` loops are ideal for counted iteration with known bounds
- Nested loops enable processing of multi-dimensional data and complex patterns

Key points to remember:

- Always ensure your loop will eventually terminate (avoid infinite loops)
- Initialize variables before using them in loop conditions
- Be careful with loop boundaries to avoid off-by-one errors
- Use meaningful variable names, especially in nested loops
- Consider performance implications of nested loops
- Use `break` and `continue` judiciously to control loop flow

With practice, you'll develop an intuition for which loop structure best fits each situation, leading to cleaner, more maintainable code.