MTH 4300/4299: Programming and Computer Science II

Lecture 03: Control flow (if-else statements, loops)

Jaime Abbariao

Updates from last class

> What's the difference between `size()` and `length()`?

For `std::string`, `size()` and `length()` are identical — they're synonyms that both return the number of characters. `length()` exists because it's intuitive for strings, while `size()` exists for consistency with other STL containers (like `std::vector`, `std::map`, etc.), which only have `size()`.

Use whichever you prefer for strings. `size()` is more common in generic/template code since it works across all containers.

Conditional statements

The if statement allows us to execute code only when a condition is true.

```
int x = 10;

if (x > 5) {
    std::cout << "x is greater than 5" << std::endl;
}
```

Conditional statements (ii)

The if-else statement provides an alternative path when the condition is false.

```
int temperature = 30;

if (temperature > 25) {
    std::cout << "It's hot outside" << std::endl;
} else {
    std::cout << "It's not too hot" << std::endl;
}
```

▨ Conditional statements (iii)

The if-else if-else chain handles multiple conditions.

```cpp
int score = 85;

if (score >= 90) {
    std::cout << "Grade: A" << std::endl;
} else if (score >= 80) {
    std::cout << "Grade: B" << std::endl;
} else if (score >= 70) {
    std::cout << "Grade: C" << std::endl;
} else if (score >= 60) {
    std::cout << "Grade: D" << std::endl;
} else {
    std::cout << "Grade: F" << std::endl;
}
```

Only the first matching condition executes.

Exercise

Write a program that reads three integers from the user and prints the largest of the three.

```
Sample output:
  Enter three numbers: 7 12 5
  The largest number is 12
```

Nested conditional statements

Conditional statements can be placed inside other conditional statements.

```
int age = 25;
bool has_license = true;

if (age >= 18) {
    if (has_license) {
        std::cout << "You can drive" << std::endl;
    } else {
        std::cout << "You need to get a license" << std::endl;
    }
} else {
    std::cout << "You are too young to drive" << std::endl;
}
```

Nested conditions can often be simplified using logical operators (`&&`, `||`).

```
if (age >= 18 && has_license) {
    std::cout << "You can drive" << std::endl;
}
```

▨ Exercise

Write a program that reads a year and determines if it is a leap year. A year is a leap year if:

- It is divisible by 4, and
- It is not divisible by 100, unless it is also divisible by 400

```
Sample output:
  Enter a year: 2000
  2000 is a leap year
```

▨ Order of conditions matter

When using if-else if chains, the order of conditions affects the result.

```cpp
int value = 15;

// Version 1: More specific conditions first
if (value > 20) {
    std::cout << "Greater than 20" << std::endl;
} else if (value > 10) {
    std::cout << "Greater than 10" << std::endl;  // This executes
} else if (value > 0) {
    std::cout << "Greater than 0" << std::endl;
}

// Version 2: Less specific conditions first (wrong!)
if (value > 0) {
    std::cout << "Greater than 0" << std::endl;  // This always executes first
} else if (value > 10) {
    std::cout << "Greater than 10" << std::endl;  // Never reached for value=15
} else if (value > 20) {
    std::cout << "Greater than 20" << std::endl;
}
```

Rule: Place more specific conditions before more general ones.

Exercise

What does the following program print? Trace through it carefully.

```cpp
int x = 50;

if (x > 10) {
    std::cout << "A ";
} else if (x > 30) {
    std::cout << "B ";
} else if (x > 40) {
    std::cout << "C ";
}
```

▨ Switch statements

The switch statement is useful for comparing a variable against multiple constant values.

```
int day = 3;

switch (day) {
    case 1:
        std::cout << "Monday" << std::endl;
        break;
    case 2:
        std::cout << "Tuesday" << std::endl;
        break;
    case 3:
        std::cout << "Wednesday" << std::endl;
        break;
    case 4:
        std::cout << "Thursday" << std::endl;
        break;
    case 5:
        std::cout << "Friday" << std::endl;
        break;
    default:
        std::cout << "Weekend" << std::endl;
        break;
}
```

The break statement prevents fall-through to the next case.

▨ Switch statements (fall-through)

Without break, execution continues to the next case.

```cpp
int month = 2;
int days;

switch (month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        days = 31;
        break;
    case 4: case 6: case 9: case 11:
        days = 30;
        break;
    case 2:
        days = 28;  // Simplified, ignoring leap years
        break;
    default:
        days = 0;
        break;
}

std::cout << "Days in month: " << days << std::endl;
```

Fall-through can be intentional for grouping cases with the same behavior.

▨ Exercise

Write a program that reads a single character and uses a switch statement to print whether it is a vowel or a consonant (assume lowercase a-z input only).

```
Sample output:
  Enter a letter: e
  e is a vowel
```

▨ Loops

Loops allow us to execute a block of code repeatedly.

C++ provides several types of loops:

- `for` loop: When you know the number of iterations
- `while` loop: When you don't know the number of iterations in advance
- `do-while` loop: When you need to execute at least once
- Range-based `for` loop: For iterating over collections

Each type has its use cases, but they can often be used interchangeably.

Loop invariants

A loop invariant is a condition that is true before and after each iteration of a loop.

```
// Goal: Calculate the sum of 1 to n
int n = 5;
int sum = 0;

// Loop invariant: sum = 1 + 2 + ... + i (after iteration i)
for (int i = 1; i <= n; ++i) {
    sum += i;
    // Invariant holds: sum now contains 1 + 2 + ... + i
}

// After loop: sum = 1 + 2 + ... + n = 15
std::cout << "Sum: " << sum << std::endl;
```

Loop invariants help us:

- Reason about correctness
- Debug loops
- Prove that loops terminate with the correct result

▨ for-loop

The for loop has three parts: initialization, condition, and update.

```
for (initialization; condition; update) {
    // body
}
```

Example: Print numbers 1 to 5

```
for (int i = 1; i <= 5; ++i) {
    std::cout << i << " ";
}
std::cout << std::endl;
// Output: 1 2 3 4 5
```

Example: Count down from 10 to 1

```
for (int i = 10; i >= 1; --i) {
    std::cout << i << " ";
}
std::cout << std::endl;
// Output: 10 9 8 7 6 5 4 3 2 1
```

▨ for-loop (ii)

Example: Print even numbers from 0 to 10

```cpp
for (int i = 0; i <= 10; i += 2) {
    std::cout << i << " ";
}
std::cout << std::endl;
// Output: 0 2 4 6 8 10
```

Example: Nested for-loops (multiplication table)

```cpp
for (int i = 1; i <= 3; ++i) {
    for (int j = 1; j <= 3; ++j) {
        std::cout << i * j << "\t";
    }
    std::cout << std::endl;
}
// Output:
// 1    2    3
// 2    4    6
// 3    6    9
```

Exercise

Write a program that uses nested for-loops to print the following pattern:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

▨ Range for-loop

C++11 introduced the range-based for loop for iterating over collections.

```cpp
#include <vector>
#include <string>

std::vector<int> numbers = {1, 2, 3, 4, 5};

for (int num : numbers) {
    std::cout << num << " ";
}
std::cout << std::endl;
// Output: 1 2 3 4 5
```

Iterating over a string:

```cpp
std::string word = "Hello";

for (char c : word) {
    std::cout << c << "-";
}
std::cout << std::endl;
// Output: H-e-l-l-o-
```

▨ Range for-loop (with references)

Use references to modify elements or avoid copying large objects.

```cpp
std::vector<int> values = {1, 2, 3, 4, 5};

// Double each value using a reference
for (int& val : values) {
    val *= 2;
}

// Print the modified values
for (int val : values) {
    std::cout << val << " ";
}
std::cout << std::endl;
// Output: 2 4 6 8 10
```

▒▒ Range for-loop (with references) (ii)

Use const references for read-only access to large objects:

```cpp
std::vector<std::string> names = {"Alice", "Bob", "Charlie"};

for (const std::string& name : names) {
    std::cout << name << std::endl;
}
```

▒▒ Exercise

Given the following code, what is the output?

```cpp
std::vector<int> nums = {10, 20, 30, 40, 50};

for (int& n : nums) {
    n += 5;
}

for (int n : nums) {
    std::cout << n << " ";
}
std::cout << std::endl;
```

Now: what would the output be if the first loop used `int n` instead of `int& n`?

▨ while loop

The while loop executes as long as a condition is true.

```
while (condition) {
    // body
}
```

Example: Count down from 5

```
int count = 5;

while (count > 0) {
    std::cout << count << " ";
    --count;
}
std::cout << "Liftoff!" << std::endl;
// Output: 5 4 3 2 1 Liftoff!
```

The condition is checked before each iteration.

▨ while loop (ii)

Example: Read input until a sentinel value

```
int number;
int sum = 0;

std::cout << "Enter numbers (0 to stop): ";

std::cin >> number;
while (number != 0) {
    sum += number;
    std::cin >> number;
}

std::cout << "Sum: " << sum << std::endl;
```

Example: Find the number of digits in a number

```
int n = 12345;
int digits = 0;

while (n > 0) {
    n /= 10;
    ++digits;
}

std::cout << "Number of digits: " << digits << std::endl;
// Output: Number of digits: 5
```

Exercise

Write a program that uses a while loop to reverse the digits of an integer.

```
Sample output:
  Enter a number: 12345
  Reversed: 54321
```

Hint: Use % 10 to get the last digit and / 10 to remove it.

▒▒▒ do-while loop

The do-while loop executes the body at least once, then checks the condition.

```
do {
    // body
} while (condition);
```

Example: Menu selection

```
int choice;

do {
    std::cout << "Menu:" << std::endl;
    std::cout << "1. Option A" << std::endl;
    std::cout << "2. Option B" << std::endl;
    std::cout << "3. Exit" << std::endl;
    std::cout << "Enter choice: ";
    std::cin >> choice;

    if (choice == 1) {
        std::cout << "You selected Option A" << std::endl;
    } else if (choice == 2) {
        std::cout << "You selected Option B" << std::endl;
    }
} while (choice != 3);
```

▨ do-while vs while

The key difference: do-while always executes at least once.

```cpp
int x = 0;

// while loop: body never executes
while (x > 0) {
    std::cout << "while: " << x << std::endl;
    --x;
}

// do-while loop: body executes once
do {
    std::cout << "do-while: " << x << std::endl;
    --x;
} while (x > 0);

// Output: do-while: 0
```

Use do-while when:

- You need to execute the body at least once
- The condition depends on something computed in the body

▨ Exercise

Write a program using a do-while loop that asks the user to guess a secret number (e.g. 42). The program should keep asking until they guess correctly, and print how many attempts it took.

```
Sample output:
  Guess the number: 10
  Wrong! Try again.
  Guess the number: 42
  Correct! It took you 2 attempts.
```

▨ Infinite loops with break

Sometimes it's cleaner to use an infinite loop with break to exit.

```
while (true) {
    std::cout << "Enter a positive number: ";
    int n;
    std::cin >> n;

    if (n > 0) {
        std::cout << "You entered: " << n << std::endl;
        break;  // Exit the loop
    }

    std::cout << "Invalid input. Try again." << std::endl;
}
```

The continue statement skips to the next iteration:

```
for (int i = 1; i <= 10; ++i) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    std::cout << i << " ";
}
std::cout << std::endl;
// Output: 1 3 5 7 9
```

▨ Exercise

Write a program that repeatedly reads integers from the user. If the input is negative, print "Skipping negative number" and use continue to skip it. If the input is 0, use break to exit. Otherwise, add the number to a running sum. Print the sum at the end.

```
Sample output:
  Enter a number: 5
  Enter a number: -3
  Skipping negative number
  Enter a number: 10
  Enter a number: 0
  Sum: 15
```

▒ Loop control: break and continue

break exits the innermost loop:

```
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
        if (j == 1) {
            break;  // Only exits inner loop
        }
        std::cout << "(" << i << "," << j << ") ";
    }
}
std::cout << std::endl;
// Output: (0,0) (1,0) (2,0)
```

continue skips to the next iteration of the innermost loop:

```
for (int i = 0; i < 5; ++i) {
    if (i == 2) {
        continue;
    }
    std::cout << i << " ";
}
std::cout << std::endl;
// Output: 0 1 3 4
```

▨ Common loop patterns

Accumulator pattern: Build up a result

```
int product = 1;
for (int i = 1; i <= 5; ++i) {
    product *= i;
}
// product = 120 (5!)
```

▨ Common loop patterns (ii)

Search pattern: Find an element

```
std::vector<int> data = {3, 7, 2, 9, 5};
int target = 9;
bool found = false;

for (int val : data) {
  if (val == target) {
    found = true;
    break;
  }
}
```

Common loop patterns (iii)

Counter pattern: Count occurrences

```cpp
std::string text = "hello world";
int vowel_count = 0;

for (char c : text) {
  if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
    ++vowel_count;
  }
}
// vowel_count = 3
```

▨ Lab

Exercise 1: Write a program that reads an integer and prints whether it is positive, negative, or zero.

Exercise 2: Write a program that reads a grade (0-100) and prints the letter grade (A, B, C, D, F) using:

- a) `if-else if-else` statements
- b) A `switch` statement (hint: use `grade / 10`)

Exercise 3: Write a program that prints all prime numbers between 2 and 100.

Exercise 4: Write a program that reads integers until the user enters -1, then prints:

- The count of numbers entered
- The sum of the numbers
- The average of the numbers

Exercise 5: Write a program that prints the following pattern using nested loops:

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