Chapter 2
Flow of Control
# Comparison Operators

<table>
<thead>
<tr>
<th>MATH SYMBOL</th>
<th>ENGLISH</th>
<th>C++ NOTATION</th>
<th>C++ SAMPLE</th>
<th>MATH EQUIVALENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
<td>==</td>
<td>x + 7 == 2*y</td>
<td>x + 7 = 2y</td>
</tr>
<tr>
<td>≠</td>
<td>Not equal to</td>
<td>!=</td>
<td>ans != 'n'</td>
<td>ans ≠ 'n'</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
<td>count &lt; m + 3</td>
<td>count &lt; m + 3</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>time &lt;= limit</td>
<td>time ≤ limit</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
<td>time &gt; limit</td>
<td>time &gt; limit</td>
</tr>
<tr>
<td>≥</td>
<td>Greater than or equal to</td>
<td>&gt;=</td>
<td>age &gt;= 21</td>
<td>age ≥ 21</td>
</tr>
</tbody>
</table>
## Logical Operators

### Display 2.2  Truth Tables

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Exp_1 &amp;&amp; Exp_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

### NOT  

<table>
<thead>
<tr>
<th>Exp</th>
<th>!(Exp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

### OR  

| Exp_1 | Exp_2 | Exp_1 ||| Exp_2 |
|-------|-------|--------|
| true  | true  | true   |
| true  | false | true   |
| false | true  | true   |
| false | false | false  |
Boolean Expressions

• Evaluates to true or false
  – Predefined C++ constants
• Precedence rules apply for all operators
  – Arithmetic evaluates before logic
  – && evaluates before ||
  – Always use parentheses for clarity

\[(x > 0) \&\& (x < 10) \quad (y == 0) || (y == 1)\]
\[(z < 0) || ((z >= 5) \&\& (z <= 10))\]
Boolean Expressions

- Short circuit evaluation
  - Stops evaluating when final result can be determined
  - Applies to `&&` and `||`
- Integers interpreted as boolean values
  - Zero is false
  - All non-zero values are true

```
hasMoney = dollarAmtInWallet || dollarAmtInBankAccount
```
if-else

• Chooses between 2 alternative statements based on a boolean expr
• Only one alternative is executed!
• Simple case
  – Each alternative is only one statement

```java
if (hrs <= 40)
    grossPay = rate * hrs;
else
    grossPay = (rate * 40) + (overtimeRate * (hrs - 40));
```
Compound Statements

- Use braces to include more than one statement
  - Anything between {} is considered one statement

```cpp
if (myScore > yourScore) {
    cout << "I win!";
    playAgain = true;
}
else {
    cout << "I wish these were golf scores.";
    playAgain = false;
}
```
Omitting the else

- Sometimes you want to take action only if an expression is true
- Else clause can be omitted
- Works similarly for compound statements

```cpp
if (sales > minimum)  
    salary += bonus;

cout << "Salary = $" << salary;
```
Nested Statements

- If-else statements can contain other if-else statements
- Logic like AND

```cpp
if (speed > 55) {
    cout << "You’re speeding.";

    if (speed > 65) {
        cout << "You’re probably going to get a ticket";
        if (speed > 90)
            cout << "I hope you've written your will";
    }
}
```
Multiway if-else

• Nested if-else with special indentation
• Logic like OR

```
EXAMPLE
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home."
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work."
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm."
else //temperature > 0
    cout << "Work hard and play hard."
```

The Boolean expressions are checked in order until the first true Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is true, then the `Statement_For_All_Other_Possibilities` is executed.
Switch Syntax

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
        ...
        ...
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence

You need not place a break statement in each case. If you omit a break, that case continues until a break (or the end of the switch statement) is reached.
```
Switch

• Specifies what to do for each case.

**Example**

```cpp
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass) {
    case 1:
        cout << "Passenger car."
        toll = 0.50;
        break;
    case 2:
        cout << "Bus."
        toll = 1.50;
        break;
    case 3:
        cout << "Truck."
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!"
}
```

If you forget this `break`, then passenger cars will pay $1.50.
Omitting the break

- Not a syntax error!
- Allows execution to “fall through”
- Multiple cases can be executed

```c++
switch (grade) {
    case "A":
    case "a":
        cout << "Excellent!";
        break;
    case "B":
    case "b":
        cout << "Good!";
        break:
    ...
}
```
Switch with Menus

• Switch is most often used with menus
• Clearly shows menu structure

```cpp
switch (response) {
    case "1":
        // Execute menu option 1
        break;
    case "2":
        // Execute menu option 2
        break;
    case "3":
        // Execute menu option 3
        break;
    default:
        cout << "Please enter a valid response.";
}
```
Loops

● 3 types
  - for
    • Runs for a fixed number of iterations
  - while
    • Runs until some condition is false
  - do while
    • Always executes at least one time

● Logically equivalent
  - All iterations can be implemented using any loop
For Loop

- Natural counting loop
- Contains 3 parts
  - Initialization
  - Stopping Condition
  - Update Action

**Syntax:**

```cpp
for (initialization; condition; update)
    statement;
```

**Example:**

```cpp
for (int i = 0; i < 100; ++i)
    cout << i;
```
While Loop

- Runs until some condition is false
- Implicitly includes same 3 parts as the for loop

Syntax:

```cpp
while (condition) 
    statement;
```

Example:

```cpp
int i = 0; // Initialization
while (i < 100) { // Stopping Condition
    cout << i; // Update Action
    ++i;
}
```
Do While Loop

- Same as while loop except it always executes at least one time
- Don't forget the final semicolon!

**Syntax:**

```cpp
do
    statement;
while (condition);
```

**Example:**

```cpp
int i = 0; // Initialization
do {
    cout << i;
    ++i; // Update Action
} while (i < 100); // Stopping Condition
```
While vs. Do While

- Very similar but one important difference
  - “When” the stopping condition is checked
  - While checks before the body is ever executed
  - Do while checks after the body is executed

- All iterations after the first one behave identically
Common Loop Mistakes

• Forgetting the final semicolon in `do while`
• Adding an extra semicolon
  – Results in an infinite loop
  – `while (x < 10);`
• Stopping condition that is never reached
  – Results in an infinite loop

```cpp
int x = 1;
while (x != 12) {
    cout << x << endl;   // Output all positive even numbers before 12
    x += 2;
}
```
Natural Flow

- Should **always** exit loop “gracefully” by reaching the stopping condition.
  - In rare cases, you can alter the natural flow
- **break**;
  - Forces loop to exit immediately
- **continue**;
  - Skips the rest of the loop body
- **These statements violate natural flow!**
  - Use only if the world is coming to an end
Nested Loops

• Perform one iteration inside another

Remember Multiplication Tables?

```c++
for (int n = 2; n <= 12; ++n) {
    cout << n << “ Times Table” << endl;

    for (int m = 1; m <= 12; ++m)
        cout << n << “ x “ << m << “ = “ << (n * m) << endl;

    cout << endl;
}
```