More Flow of Control

Chapter 3
Objectives

In this chapter, you will learn about:

- Using Boolean Expressions
- Multiway Branches
- More about C++ Loop Statements
- Designing Loops
Using Boolean Expressions

- **Boolean Expressions**
  - Take the Value **true** or **false**
- **Boolean Value is produced using**
  - Relational Operations:
    - =, <, >, <=, >=
  - Boolean Operations
    - &&, ||, !
- **Boolean Variable (okay) Declaration**
  - bool okay // okay takes value true or false
Evaluating Boolean Expressions

- Let’s review evaluating Arithmetic Expressions
  - Rule: *PEMDAS*
  - \((x + 1) \times (x + 3)\)

- Boolean Expression Evaluation
  - Truth Tables and Precedence Rules
  - Let’s review Truth Tables for &&, || and !
  - Example: !\((y < 3) \ | \ | (y > 7)\)
    - !(false || true)
    - false

Assume $y = 8$
When parenthesis are omitted from boolean expressions, computer groups items using rules known as Precedence Rules:

<table>
<thead>
<tr>
<th>Boolean &amp; Arithmetic Operators</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary: +, -, ++, --, and !</td>
<td>Highest Precedence</td>
</tr>
<tr>
<td></td>
<td>Perform ! Operation first</td>
</tr>
<tr>
<td>Binary: *, /, %</td>
<td></td>
</tr>
<tr>
<td>Binary: +, -</td>
<td></td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>Perform relational operations next</td>
</tr>
<tr>
<td>==, !=</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Perform &amp;&amp; operations next</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Boolean Expressions
Order of Precedence

1. First Evaluate Operators with higher precedence

2. When Operators have equal Precedence:
   - If Binary Operators
     - Evaluate expression from left to right
   - If Unary Operators
     - Evaluate item from right to left
(x+1) > 2 || (x + 1) < -3

is equivalent to:

(x + 1) > 2) || (x + 1) < -3)

<table>
<thead>
<tr>
<th>Boolean &amp; Arithmetic Operators</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary: +, -, ++, --, and !</td>
<td>Highest Precedence</td>
</tr>
<tr>
<td></td>
<td>Perform ! Operation first</td>
</tr>
<tr>
<td>Binary: *, /, %</td>
<td></td>
</tr>
<tr>
<td>Binary: +, -</td>
<td></td>
</tr>
<tr>
<td>&lt;=, &gt;=</td>
<td>Perform relational operations next</td>
</tr>
<tr>
<td>==, !=</td>
<td>Perform relational operations next</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Perform &amp;&amp; operations next</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Precedence Rule**

**Example 2**

- \((x+1) > 2 \, || \, (x + 1) < -3\)
  is equivalent to:
- \(x + 1 > 2 \, || \, x + 1 < -3\)

**Why?**

**What is the order of operation?**

<table>
<thead>
<tr>
<th>Boolean &amp; Arithmetic Operators</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary: ++, --, +, -</strong></td>
<td>Highest Precedence</td>
</tr>
<tr>
<td></td>
<td>Perform + Operation first</td>
</tr>
<tr>
<td>Binary: * , /, %</td>
<td></td>
</tr>
<tr>
<td>Binary: +, -</td>
<td></td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>Perform relational operations next</td>
</tr>
<tr>
<td>==, !=</td>
<td>Perform relational operations next</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Perform &amp;&amp; operations next</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using Boolean Expressions

Short Circuit Evaluation

- If value of the leftmost sub-expression determines the final value of entire expression, the trailing expressions are not evaluated

Example:

$$\text{if } x \geq 0 \text{ and } y > 1$$

A && ( y > 1)
false && ( y > 1)
false

Assume

Check Truth Table for &&

Short Circuit Evaluation and Complete Evaluation give the same result
Short Circuit Evaluation
Avoids Runtime Errors

Consider C++ statement:
if ((kids != 0) && ((pieces/kids) >= 2))
cout << “Each child may have tow pieces!\n”;

Assume kids = 0

Short Circuit Evaluation:
if (false && ((pieces/kids) >= 2))

Second sub-expression is not evaluated

Complete Evaluation:
if (false && (pieces/0) >= 2)

Run-time error (divide by 0)
Variables of type bool take values \texttt{true} or \texttt{false}.

C++ also converts integers to Boolean values:
- A non-zero number (e.g., -10, -1, 1, 4, 10..) is \texttt{true}
- Zero (0) is \texttt{false}

Let’s examine pitfalls associated with Boolean expressions
Boolean Expressions

bool values Convert to int values

1.

```cpp
if ( !(time > limit) )
    cout << “ CS23021 Lab Session\n”;
else
    cout << “End of CS23021 Lab Session\n”; 
```

Assume:

<table>
<thead>
<tr>
<th>time</th>
<th>limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>60</td>
</tr>
</tbody>
</table>

Output: CS23021 Lab Session

2.

```cpp
if ( !time > limit )
    cout << “ CS 23021 Lab Session\n”; 
else
    cout << “End of CS 23021 Lab Session\n”; 
```

What ‘s `!time`?

!(true) → false → 0

Hence,

if (0 > 60) → false

Output: **End of CS23021 Lab Session**

Avoid using not (!) operator
**Enumeration Types**

**enum**

- **Type** `enum` refers to a list of constants of type `int`
- **Example Declaration:**
  ```
  enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
                   MAR_LENGTH = 31, APRIL_LENGTH = 30,
                   MAY_LENGTH = 31, JUNE_LENGTH = 30
  };
  ```

- **Default enum values:**
  - Identifiers are assigned consecutive values starting with zero
    - Value assigned to a trailing identifier is one more than previous value
  - **Examples** ....
    - `enum Direction { NORTH, SOUTH, EAST, WEST };`
    - `enum myEnum { ONE =17, TWO, THREE, FOUR = -3, FIVE };`
What is the output of the code segment?

```cpp
enum Direction {N, S, E, W};
cout << W << " " << E << " " << S << " " << N << endl;
```

Output: 3 2 1 0

What is the output of the code segment?

```cpp
enum Direction {N, S, E, W};
cout << " " << Direction::E << " " << Direction::S
<< " " << Direction::N << endl;
```

Output: 2 1 0
What is the output of the code segment?

```
enum Direction {N = 5, S = 7, E = 1, W};
cout << W << " " << E << " " << S << " " << N << endl;
```

Output: 2 1 7 5

What is the output of the code segment?

```
enum Direction {N = 5, S = 7, E = 1, W};
cout << Direction::W << " " << Direction::E << " " << Direction::S << " " << N << endl;
```

Output: 2 1 7 5
What is the output of the code segment?

```cpp
int x = 2;
cout << "Start\n";
if (x <= 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
else
    cout << "Hello from the else.\n";
cout << "End\n";
cout << "Start again\n";
if (x > 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
else
    cout << "Hello from the else.\n";
cout << "End again\n";
```

**output**

```
Hello from the second if.
End
Start again
End again
```
if – else statement

Braces

- Braces tell the compiler how to group statements

- **Always Use braces to take control of if-else constructs**

```cpp
int x = 2;
cout << "Start\n";
if (x <= 3)
{
    if (x != 0)
        cout << "Hello from the second if.\n";
}
else
{
    cout << "Hello from the else.\n";
}
cout << "End\n";
cout << "Start again\n";
if (x > 3)
{
    if (x != 0)
        cout << "Hello from the second if.\n";
}
else
{
    cout << "Hello from the else.\n";
}
cout << "End again\n";
```

Output

Hello from the second if
End
Start again
Hello from the else
End again
Multiway Branches
Nested if-else Statement

- An if-else-statement includes one or more blocks of if-else-statements

```
if (guess > number)
    cout << "Higher. " ;
else
    if (guess < number)
        cout << "Lower.";
    else
        if (guess == number)
            cout << "Correct Number";

// Use the if-else-if construct for nested if constructs

if (guess > number)
    cout << "Higher. ";
else if (guess < number)
    cout << "Lower. ";
else
    cout << "Correct Number";
```
Multiway if-else Statement
Syntax

\[ \text{if } (\text{Boolean Expression}_1) \]
\[ \text{Statement}_1 \]
\[ \text{else if } (\text{Boolean Expression}_2) \]
\[ \text{Statement}_2 \]
\[ \vphantom{\text{else if}} \]
\[ \downarrow \]
\[ \text{else if } (\text{Boolean Expression}_n) \]
\[ \text{Statement}_n \]
\[ \text{else} \]
\[ \text{Default statements} \]
What is the output of the code segment?

```cpp
int x = 200;
cout << "Start\n";
if ( x < 100 )
    cout << "First Output. \n";
else if (x > 10)
    cout << "Second Output.\n";
else
    cout << " Third Output.\n";
cout << "Start\n";
```

Output:

Second Output. Start
Multiway if-else Statement

Example 2

What is the output of the code segment?

```cpp
int x = SOME_CONSTANT;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 100)
    cout << "second Output.\n";
else
    cout << x << endl;
cout << "End\n";
```

Assume that code does not output “First Output” or “Second Output”

Output

```
100
End
```
Switch Statement
Example 1

What is the output of the code segment?

```cpp
switch (grade)
{
    case 'A':
        cout << " Excellent. You need to take the final.\n";
        break;
    case 'B':
        cout << " Very good. \n";
        cout << " You midterm grade is now " << grade << endl;
        break;
    case 'C':
        cout << "Passing.\n";
        break;
    case 'D':
    case 'E':
        cout << "Not good. Go study.\n";
        break;
    default:
        cout << "That is not a possible grade.\n"
}
```

Assume:

1. `char grade;` 'A'
   - Excellent. You need to take the final

2. `char grade;` 'D'
   - Not good. Go study
Switch Statement
Example 2

What is the output of the code segment?

```cpp
int first_choice = 1;
switch (first_choice + 1) {
    case 1:
        cout << " Roast beef \n";
        break;
    case 2:
        cout << "Roast worms\n";
        break;
    case 3:
        cout << "Chocolate ice cream\n";
        break;
    case 4:
        cout << "Onion ice cream\n";
        break;
    default:
        cout << "Bon appetit! \n";
}
```

What if

1. ```cpp
    int first_choice;
    3
```  
   Onion ice cream

2. ```cpp
    int first_choice;
    6
```  
   Bon appetit!
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
}
```
The switch controlling expression must return one of the value Types:

- A Character
- An Integer
- An enum constant
- A bool

The returned value is subsequently compared with each case value. If there is a match the corresponding case statement is executed.
The break statement ends execution of the switch-statement

- If the break statement is omitted, the code for the trailing case(s) will be executed!
- Use this technique if you wish to execute multiple case labels for one section of code
Switch Statement

Default

- The default statement is executed if no case label has a constant that matches the controlling expression

- If the default statement is missing, nothing happens when the switch statement is executed

Always include a default section in a switch statement
Switch and if-else-statements allow the use of multiple statements in a branch.

- Multiple statements in a branch can make the switch or if-else-statement difficult to read
  - Use function calls instead of multiple statements
  - Function calls are easier to read
Blocks

Sub-task

- Each branch of a switch or if-else statement is a separate sub-task
  - If the action of a branch is too simple to warrant a function call, use multiple statements between braces
  - A block is a section of code enclosed by braces
  - Variables declared within a block, are local to the block or have the block as their scope.
    - Variable names declared in the block can be reused outside the block (not a good practice though)
Blocks
Scope of Variables

- When you declare a single identifier as a variable in two distinct blocks, where one block is within the other, you end up with two different variables with the same name (identifier)
  - One of the variables exists only within the inner block and cannot be accessed outside the inner block
  - The other variable exists only in the outer block and cannot be accessed in the inner block
What is the output of the code segment

```cpp
int main( )
{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 4;
        cout << x << endl;
    }
    cout << x << endl;
    return 0;
}
```
What is the output of the code segment

```cpp
int main( )
{
    int x = 1;     // Declares Local integer variable in block1
    cout << x << endl;
    {
        cout << x << endl;
        int x = 4;    // Declares Local integer variable in second block
        cout << x << endl;
    }
    cout << x << endl;
    return 0;
}
```

Output

1
1
1
4
1

Increment Operator

Statement

- **Statement**
  - `number++;

```
int number = 41;
number++;
cout << number;
```

*Increment value stored in variable* `number` *by 1*

Outputs 42
Increment Operator
Expression

- Expression
  - 2 * (number++);

First return the value stored in variable `number` to be multiplied by 2; then increment value stored in variable `number` by 1

```c++
int number = 4;
int value_produced = 2 * (number++);
cout << value_produced;
cout << number << endl;
```

Outputs
8
5
Increment Operator

`++number` vs `number++`

- `++number;`  
  First increment by 1 the value stored in variable `number` then use the incremented value in the expression

- `number++;`  
  First return the current value stored in variable `number` to be used in the expression; then increment the current value in variable `number` by 1
**++number vs number++**

**Examples**

- ```cpp
   int number = 2;
   int value_produced = 2 * (number++);
   cout << value_produced << " " << number;
```

  Output: 4 3

- ```cpp
   int number = 2;
   int value_produced = 2 * (++number);
   cout << value_produced << " " << number;
```

  Output: 6 3
Decrement Operator

Statement

- **number--;**

Decrements value stored in variable number by 1

```cpp
int number = 41;
number--;
cout << number;
```

Outputs 40
Increment Operator

Expression

- **Expression**
  - $2 \times (\text{number}--)$;

  First return the value stored in variable `number` to be multiplied by 2; then decrement value stored in variable `number` by 1

```cpp
int number = 4;
int value_produced = 2 * (number--);
cout << value_produced;
cout << number << endl;
```

Outputs

```
8
3
```
Decrement Operator Expressions
--number vs number--

- **--number;**
  First decrement by 1 the value stored in variable `number` then use the decremented value in the expression

- **number--;**
  First return the current value stored in variable `number` to be used in the expression; then decrement the current value in variable `number` by 1
---number vs number---

**Examples**

- int number = 2;
  int value_produced = 2 * (number--);
  cout << value_produced << " " << number;

  **Output:**
  4 1

- int number = 2;
  int value_produced = 2 * (--number);
  cout << value_produced << " " << number;

  **Output:**
  2 1
Can you determine the output of the following code segments?

```cpp
int count = 3;
while ( count-- > 0)
    cout << count << " ";
```

```cpp
int n = 1;
do
    cout << n << " ";
while ( n++ <= 3);
```
for Loop

- **for** statement:
  - Similar to while loop
  - Ideal for adding numerals

for loop with **single** statement:

```c
sum = 0;
for ( n = 1; n <= 4; n++)
    sum = sum + n;
cout << sum << endl;
```

Output: 10

Diagram:

```
sum:

0 → 1 → 3 → 6 → 10
```

Introduction Computer Science 1 CS 23021
for Loop
Compound statement

```cpp
sum = 0;
for (n = 1; n <= 4; n++) {
    sum = sum + n;
    cout << sum << " ";
}
```

Output:
1 3 6 10

Use braces when for loop body contains multiple statements
for Loop

Components

Initialization
Action

Update Action:
Done after
each iteration

for (n = 1; n <= 4; n++)

Boolean
Expression
Repeat loop if true
for (Initialization_Action; Boolean Expression; Update_Action)
{
    Statement_1
    Statement_2
    \vdots
    Statement_n
}

The body of a for loop can be a single statement or multiple statements enclosed in braces.
We can declare a local variable within a for loop statement.

```
sum = 0;
for (int n = 1; n <= 4; n++)
{
    sum = sum + n;
    cout << sum << "  

    sum = sum + n;
    cout << sum << "  

create variable n and initialize to 1
variable n cannot be accessed outside the loop body*

*ANSI C++ standard
```
for Loop
More Update Actions

Increment by 2

for (n = 1; n <= 10; n + 2)

Decrement by 7

for (n = 0; n > -100; n = n -7)

Increment by fraction

for (double size = 0.75; size <= 5; size = size + 0.05)

Update by function call

for (double x = pow(y,3.0); x > 2.0; x = sqrt(x))

y is computed within the loop
Empty body statement

- **C++ Empty Statement**
  - A Standalone semicolon
  - It is executed by the compiler but does **nothing**

```c++
sum = 0;
;
```

Empty Statement
How does the compiler execute a trailing semicolon in a for statement?

```cpp
sum = 0;
for (n = 1; n <= 4; n++)
    sum = sum + n;
cout << sum << endl;
```

- Body of for loop is an empty statement
- Variable `n` is outside of the for loop
- `n` is undefined (Error in assignment statement!!)

Can you determine the output of the code segment?

```cpp
for (n = 1; n <= 4; n++)
    cout << "Game Over\n"
cout << endl;
```
for Loop
break statement

- break;
  - Use the break statement to abruptly exit the loop containing the break statement
    - Why? → Code detects abnormal conditions
    - → Avoid possible runtime errors
Can you determine the output of the code segment?

```
int n = 5;
while (--n > 0)
{
    if ( n == 2)
        break;
    cout << n << " ";
}
cout << "End of Loop.\n";
```