

# Chapter 3: More Flow of Control

CISC1600, Fordham Univ.

X. Zhang

# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches
  - Embedded if-else statement
- More about C++ Loop Statements
- Designing Loops

# Flow Of Control

- **Flow of control** refers to the order in which program statements are performed
  - We have seen following ways to specify flow of control
    - if-else-statement, if statement
    - while-statement
  - New methods
    - switch-statements
    - for-statements

# Review: if statements

- To implement branching, use if statements

```
if (boolean_expression)
    statement1
else
    statement2
```

```
if (boolean_expression)
    statement
```

- Boolean expression is enclosed by ( )
- **statements** should be **indented by one more level**. It can be
  1. simple statement
  2. block statement, i.e.,  
a sequence statement enclosed with { }
  3. empty statement
  4. loop statement
  5. if statement, i.e., nested if statement

# Using Boolean Expressions

- A **Boolean Expression** is an expression that is either true or false
  - Using relational operations such as
    - `==`, `<`, and `>=` which produce a boolean value
  - boolean operations
    - `&&`, `||`, and `!`
- Type **bool** allows declaration of variables that store value true or false

# Evaluating Boolean Expressions

- Boolean expressions are evaluated step-by-step
  - Refer to truth tables
  - e.g., if y is 8, expression

$!( (y < 3) \mid \mid (y > 7) )$

is evaluated in following sequence

$! ( \text{false} \mid \mid \text{true} )$

$! ( \text{true} )$

false

## Truth Tables

---

### AND

<b><i>Exp_1</i></b>	<b><i>Exp_2</i></b>	<b><i>Exp_1 &amp;&amp; Exp_2</i></b>
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

### OR

<b><i>Exp_1</i></b>	<b><i>Exp_2</i></b>	<b><i>Exp_1     Exp_2</i></b>
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>

### NOT

<b><i>Exp</i></b>	<b><i>!(Exp)</i></b>
<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>

# Order of Precedence: boolean expression

- If parenthesis are omitted from boolean expressions, default precedence of operations is:
  - Perform **!** operations first
  - Perform **relational operations** such as `<` next
  - Perform **&&** operations next
  - Perform **| |** operations last



# Precedence Rules

- Items in expressions are grouped by precedence rules for arithmetic and boolean operators
  - Operators with higher precedence are performed first
  - Binary operators with equal precedence are performed left to right
  - Unary operators of equal precedence are performed right to left

## Precedence Rules

---

The unary operators `+`, `-`, `++`, `--`, and `!`.

The binary arithmetic operations `*`, `/`, `%`

The binary arithmetic operations `+`, `-`

The Boolean operations `<`, `>`, `<=`, `>=`

The Boolean operations `==`, `!=`

The Boolean operations `&&`

The Boolean operations `||`

*Highest precedence  
(done first)*



*Lowest precedence  
(done last)*

# Precedence Rule Example

- The expression

$$(x+1) > 2 \mid \mid (x + 1) < -3$$

is equivalent to

$$((x + 1) > 2) \mid \mid ((x + 1) < -3)$$

- Because  $>$  and  $<$  have higher precedence than  $\mid \mid$ , and is equivalent to

$$x + 1 > 2 \mid \mid x + 1 < -3$$

- Evaluating  $x + 1 > 2 \mid \mid x + 1 < -3$ 
  - First apply the unary  $-$
  - Next apply the  $+$ 's
  - Now apply the  $>$  and  $<$
  - Finally do the  $\mid \mid$

# Short-Circuit Evaluation (lazy evaluation)

- Sometimes, boolean expressions do not need to be completely evaluated
  - E.g., if  $x$  is negative, the value of expression  $(x \geq 0) \ \&\& \ (y > 1)$  can be determined by evaluating only  $(x \geq 0)$
- C++ **short-circuit evaluation**
  - If value of leftmost sub-expression determines final value of expression, rest of expression is not evaluated
  - E.g.,  $(x==1 \ || \ x==2)$  if  $x$  equals to 1, second part is not evaluated

# Using Short-Circuit Evaluation

- to prevent run-time errors
  - Consider

```
if ((kids != 0) && (pieces / kids >= 2) )  
    cout << "Each child may have two pieces!";
```
  - If value of kids is zero, short-circuit evaluation prevents evaluation of `(pieces / 0 >= 2)`
    - Division by zero causes a run-time error

# Type bool and Type int

- C++ can use integers as if they were Boolean values
  - Any non-zero number (typically 1) is true
  - 0 (zero) is false
- E.g., 

```
int n = 10;
while (n)
{
    cout << "*";
    n--;
}
```

# Problems with !

- The expression ( ! time > limit ), with limit = 60, is evaluated as

(!time) > limit

- If time is an int with value 36, what is !time?
  - False or zero since it will be compared to an integer
  - The expression is further evaluated as

0 > limit

false

- The intent of the previous expression was most likely the expression

( ! ( time > limit ) )

which evaluates as

( ! ( false ) )  
true

# Avoiding !

- Just as not in English can make things not undifficult to read, the ! operator can make C++ expressions difficult to understand
- Before using the ! operator see if you can express the same idea more clearly without the ! operator



# Exercise

- Determine the value of these Boolean expressions, assuming count is 0 and limit is 10?
  - `(count == 0) && (limit < 20)`
  - `!(count == 12)`
  - `(limit < 0) && ((limit / count) > 7)`
  - `(limit && count==0)`

# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches
  - nested if-else statement
  - switch statement
- More about C++ Loop Statements
- Designing Loops

# Multiway Branches

- A **branching mechanism** selects one out of a number of alternative actions
  - An if-else-statement is a two-way branch
  - **Three or four (or more) way branches:** can be designed using **nested if-else-statements**
    - Where an if-else statement is a part of another if-else statement
  - e.g.,

```
if (count < 10)
    if ( x < y)
        cout << x << " is less than " << y;
    else
        cout << y << " is less than " << x;
```

## **An *if-else* Statement within an *if* Statement**

---

```
if (count > 0)
```

```
    if (score > 5)
```

```
        cout << "count > 0 and score > 5\n";
```

```
    else
```

```
        cout << "count > 0 and score <= 5\n";
```

# Nested if-else statements

- Example: To design an if-else statement to warn a driver when fuel is low, but tells the driver to bypass pit stops if the fuel is close to full. Otherwise there should be no output.

Pseudocode: if fuel gauge is below  $\frac{3}{4}$  then:

    if fuel gauge is below  $\frac{1}{4}$  then:

        issue a warning

    otherwise (gauge  $> \frac{3}{4}$ ) then:

        output a statement saying don't stop

# First Try Nested if's

- Translating the previous pseudocode to C++ could yield

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
    else
        cout << "Fuel over 3/4. Don't stop now!\n";
```

- compile and run, but does not produce desired results
  - When fuel\_gauge\_reading is 0.3, it prints “Fuel over  $\frac{3}{4}$ , Don’t stop now!”
- Why?

# Dangling else problem

- There is **ambiguity** in the code below
  - There are two if, only one else
  - Which if is this “else” for?

This is sometimes called Dangling else problem.

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
else
    cout << "Fuel over 3/4. Don't stop now!\n";
```

- C++ standard: pairs the "else" with nearest previous "if"

# Braces and Nested Statements

- Braces in nested statements are like parenthesis in expressions
  - To pair else with first if: use `{}` to enclose the if statement.

```
if (fuel_gauge_reading < 0.75)
{
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low.  Caution!\n";
}
else
    cout << "Fuel over 3/4.  Don't stop now!\n";
```



## The Importance of Braces

*//Illustrates the importance of using braces in if-else statements.*

```
#include <iostream>
using namespace std;
int main()
{
    double fuel_gauge_reading;

    cout << "Enter fuel gauge reading: ";
    cin >> fuel_gauge_reading;

    cout << "First with braces:\n";
    if (fuel_gauge_reading < 0.75)
    {
        if (fuel_gauge_reading < 0.25)
            cout << "Fuel very low. Caution!\n";
    }
    else
    {
        cout << "Fuel over 3/4. Don't stop now!\n";
    }

    cout << "Now without braces:\n";
    if (fuel_gauge_reading < 0.75)
        if (fuel_gauge_reading < 0.25)
            cout << "Fuel very low. Caution!\n";
    else
        cout << "Fuel over 3/4. Don't stop now!\n";

    return 0;
}
```

*This indenting is nice,  
but is not what the  
computer follows.*

### Sample Dialogue 1

Enter fuel gauge reading: 0.1  
First with braces:  
Fuel very low. Caution!  
Now without braces:  
Fuel very low. Caution!

*Braces make no difference in  
this case, but see Dialogue 2.*

### Sample Dialogue 2

Enter fuel gauge reading: 0.5  
First with braces:  
Now without braces:  
Fuel over 3/4. Don't stop now!

*There should be no output here,  
and thanks to braces, there is none.*

*Incorrect output from the  
version without braces.*

# Multi-way if-else-statements: Number Guessing

- number guessing game:

```
if (guess > number)
    cout << "Too high.";
else
    if (guess < number)
        cout << "Too low.";
    else
        if (guess == number)
            cout << "Correct!";
```

If there are more nested if statements, indentation levels keep increasing, less and less space ...

# Indenting Nested if-else

- Alternative indentation for nested if-else-statements:

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else if (guess == number)
    cout << "Correct!";
```

# Final if-else-statement

- When conditions tested in an if-else-statement are mutually exclusive, final **if** can sometimes be omitted.

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.");
else // (guess == number), sure guess must be equal to number
    cout << "Correct!";
```

# Nested if-else Syntax

```
if (Boolean_Expression_1)
    Statement_1
else if ( Boolean_Expression_2)
    Statement_2
...
else if (Boolean_Expression_n)
    Statement _n
else
    Statement_For_All_Other_Possibilities
```

# Review: if statements

- To implement branching, use if statements

```
if (boolean_expression)
    statement1
else
    statement2
```

```
if (boolean_expression)
    statement
```

- Boolean expression is enclosed by ( )
- **statements** should be **indented by one more level**. It can be
  1. simple statement
  2. block statement, i.e.,  
a sequence statement enclosed with { }
  3. empty statement
  4. loop statement
  5. if statement, i.e., nested if statement

# Making sense of statements...

- By default language is English. If the country is USA and the state is PR, then set the language to Spanish. If the country is China, set the language to Chinese.

```
string language="English";  
if (country=="USA")  
    if (state=="PR") language="Spanish";  
else if (country=="China")  
    language="Chinese";
```

First: try to rewrite it using our indentation convention

# Exercise

- Write a code segment that assign appropriate value to variable `days_in_month`, based on value of variable `month`

```
int month,days_in_month;
```

```
cin >> month;
```

```
//your code here ... assuming month takes value between
```

```
// 1 and 12, and assuming it's not leap year
```



# Program Example: State Income Tax

- Write a program for a state that computes tax according to rate schedule:

No tax on first \$15,000 of income

5% tax on each dollar from \$15,001 to \$25,000

10% tax on each dollar over \$25,000

## Multiway if-else Statement (part 2 of 2)

---

```
if (net_income <= 15000)
    return 0;
else if ((net_income > 15000) && (net_income <= 25000))
    //return 5% of amount over $15,000
    return (0.05*(net_income - 15000));
else //net_income > $25,000
{
    //five_percent_tax = 5% of income from $15,000 to $25,000.
    five_percent_tax = 0.05*10000;
    //ten_percent_tax = 10% of income over $25,000.
    ten_percent_tax = 0.10*(net_income - 25000);
    return (five_percent_tax + ten_percent_tax);
}
}
```

else if (net\_income <= 25000)

## Sample Dialogue

Enter net income (rounded to whole dollars) **\$25100**  
Net income = \$25100.00  
Tax bill = \$510.00

# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Coding Style and Errors
- Multiway Branches using switch statement
- C++ Loop Statements: while, do/while loop,
- C++ Loop Statements: for loop
- Designing Loops

# Program Style

- A program written with attention to style
  - is easier to read
  - easier to correct
  - easier to change
- Indentations, Comments, Named Constants

# Program Style - **Indenting**

- Items considered a group should look like a group
  - Skip lines between logical groups of statements
  - Indent statements within statements

```
    if (x == 0)  
        statement;
```

- Braces `{}` create groups
  - Indent within braces to make the group clear
  - Braces placed on separate lines are easier to locate

# Program Style - Comments

- `//` is the symbol for a single line comment
  - Comments are explanatory notes for the programmer
  - All text on the line following `//` is ignored by the compiler
  - Example: `// calculate regular wages`  
`gross_pay = rate * hours;`
- `/* and */` enclose multiple line comments
  - Example: `/* This is a comment that spans`  
`multiple lines without a`  
`comment symbol on the middle line`  
`*/`

# Program Style - Constants

- Number constants have no mnemonic value
- Number constants used throughout a program are difficult to find and change when needed
- Constants
  - Allow us to name number constants so they have meaning
  - Allow us to change all occurrences simply by changing the value of the constant

# Constants

- `const` is the keyword to declare a constant
- Example:  

```
const int WINDOW_COUNT = 10;
```

declares a constant named `WINDOW_COUNT`
  - Its value cannot be changed by program like a variable
  - It is common to name constants with all capitals
- Create a named constant of type `double` with its value being 3.1415?



# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Coding Style and Errors
- Multiway Branches using switch statement
- C++ Loop Statements: while, do/while loop,
- C++ Loop Statements: for loop
- Designing Loops

# switch-statement

- switch-statement: for constructing multi-way branches

```
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
}
```

## A switch Statement (part 1 of 2)

---

```
//Program to illustrate the switch statement.
#include <iostream>
using namespace std;

int main()
{
    char grade;

    cout << "Enter your midterm grade and press Return: ";
    cin >> grade;

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

# Understanding switch statement

```
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
}
```

- A switch statement's **controlling expression** must be one of these types: **bool, int, long, char**
- Control expression's value is first evaluated, and then compared to constant values after each "case"
  - When a match is found, the code for that case is executed

# Understand switch statement (cont'd)

- **break** statement: “break out” of switch statement, i.e., go to next statement after the switch
- **Usually**: we use a break at the end of the statement sequence for each case
- Omitting break statement will cause the code for the next case to be executed!
  - Cascading effect, or falling-through ...
  - This allows us to **use multiple case labels for a section of code**
  - case 'A':  
case 'a':  
    cout << "Excellent.";   
    break;
  - Runs the same code for either 'A' or 'a'

# default case

- If no case label has a constant that matches controlling expression, statements following default label are executed
  - If there is no default label, nothing happens when the switch statement is executed
- It is a good idea to include a default section

# Nested if/else vs switch

- Nested if-else statements are more versatile than a switch statement
  - Question: can switch statement be used to compute tax according to rate schedule:
    - No tax on first \$15,000 of income
    - 5% tax on each dollar from \$15,001 to \$25,000
    - 10% tax on each dollar over \$25,000
- Switch-statements can make some code more clear
  - A menu is a natural application for a switch-statement

# switch statement exercise

- Can we use switch statement to rewrite the expression.cpp?
  - What is used to control/select which branch to execute?
- Can we use switch statement to set the days\_in\_month based upon month? (i.e., if month is 1,3,5,7,8,10,12, days\_in\_month is set to 31, ...)?
  - Would you choose nested if/else or switch?



# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Multiway Branches using switch statement
- Coding Style and Errors
- C++ Loop Statements: while, do/while loop,
- C++ Loop Statements: for loop
- Designing Loops

# Loop statement to repeat an action

- To repeat an action, we use a **loop**
- For example, to output "Hello" for ten times, we can:

e.g.,

```
int count=10;
while (count> 0)
{
    cout << "Hello ";
    count -= 1;
}
```

Output:      Hello Hello Hello Hello Hello .... Hello

## A *while* Loop

---

```
#include <iostream>
using namespace std;
int main()
{
    int count_down;

    cout << "How many greetings do you want? ";
    cin >> count_down;

    while (count_down > 0)
    {
        cout << "Hello ";
        count_down = count_down - 1;
    }

    cout << endl;
    cout << "That's all!\n";

    return 0;
}
```

### Sample Dialogue 1

How many greetings do you want? 3  
Hello Hello Hello  
That's all!


### Sample Dialogue 2

How many greetings do you want? 1  
Hello  
That's all!

### Sample Dialogue 3

How many greetings do you want? 0  
That's all!

*The loop body  
is executed  
zero times.*



# While Loop Operation

Boolean expression

```
count_down = 3;  
while (count_down > 0)  
{  
    cout << "Hello ";  
    count_down -= 1;  
}
```

Statement follows while loop

The body of loop

- First, boolean expression is evaluated
  - If false, program skips to line following while loop
  - If true, body of loop is executed
    - During execution, some item from boolean expression is changed
- After executing loop body, boolean expression is checked again: repeating process until expression becomes false
- A while loop might not execute at all if boolean expression is false on the first check

## Syntax of the *while* Statement

---

### A Loop Body with Several Statements:

```
while (Boolean_Expression )  
{  
    Statement_1  
    Statement_2  
    ...  
    Statement_Last  
}
```

Do NOT put a semicolon here.

body

### A Loop Body with a Single Statement:

```
while (Boolean_Expression )  
    Statement
```

body

Statement

# Exercise



- Can you write a program that prints numbers 1,2,3...,100 to the terminal?
- Write a program that
  - Ask the user to enter an integer
  - Calculate the sum of all numbers from 1 to this number
  - Output the result
- Write some code segment to keep reading from cin the number of 12 inch pizza until the user enter a non-negative number.

# do-while loop

- A **do-while loop** is always executed at least once
  - body of the loop is first executed
  - boolean expression is checked after the body has been executed
- Syntax:

```
do
{
    statements to repeat
} while (boolean_expression);
```

## Syntax of the *do-while* Statement

---

### A Loop Body with Several Statements:

```
do  
{
```

*Statement\_1*

*Statement\_2*

...

*Statement\_Last*

*body*

```
} while (Boolean_Expression);
```

*Do not forget the  
final semicolon.*

### A Loop Body with a Single Statement:

```
do
```

*body*

*Statement*

```
while (Boolean_Expression);
```



## A *do-while* Loop

---

```
#include <iostream>
using namespace std;
int main()
{
    char ans;

    do
    {
        cout << "Hello\n";
        cout << "Do you want another greeting?\n"
              << "Press y for yes, n for no,\n"
              << "and then press return: ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');

    cout << "Good-Bye\n";

    return 0;
}
```

## Sample Dialogue

```
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press return: Y
Hello
Do you want another greeting?
Press y for yes, n for no,
```

# Sample Program

- Bank charge card balance of \$50
- 2% per month interest
- How many months without payments before your balance exceeds \$100
- After 1 month:  $\$50 + 2\% \text{ of } \$50 = \$51$
- After 2 months:  $\$51 + 2\% \text{ of } \$51 = \$52.02$
- After 3 months:  $\$52.02 + 2\% \text{ of } \$52.02 \dots$

## Charge Card Program

---

```
#include <iostream>
using namespace std;
int main()
{
    double balance = 50.00;
    int count = 0;

    cout << "This program tells you how long it takes\n"
         << "to accumulate a debt of $100, starting with\n"
         << "an initial balance of $50 owed.\n"
         << "The interest rate is 2% per month.\n";

    while (balance < 100.00)
    {
        balance = balance + 0.02 * balance;
        count++;
    }

    cout << "After " << count << " months,\n";
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "your balance due will be $" << balance << endl;

    return 0;
}
```

# Infinite Loops

- **infinite loops**: loops that never stop are
  - loop body should contain a line that will eventually cause boolean expression to become false
- e.g.,: print odd numbers less than 12

```
x = 1;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```
- Better to use this comparison: `while ( x < 12)`

# Exercises



- Can you show the output of this code if `x` is of type `int`?

```
x = 10;  
while ( x > 0)  
{  
    cout << x << endl;  
    x = x - 3;  
}
```

- Show the output of the previous code using the comparison `x < 0` instead of `x > 0`?


# C++ Loop Statements

- A **loop** is a program construction that repeats a statement or sequence of statements a number of times
  - The **body of the loop** is the statement(s) repeated
  - Each repetition of the loop is an **iteration**
- Loop design questions:
  - What should loop body be?
  - How many times should the body be iterated?

## Syntax of the *while* Statement and *do-while* Statement

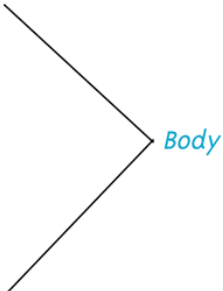
### A *while* Statement with a Single Statement Body

```
while (Boolean_Expression)
    Statement
```




### A *while* Statement with a Multistatement Body

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    .
    .
    Statement_Last
}
```



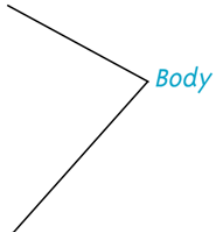
### A *do-while* Statement with a Single Statement Body

```
do
    Statement
while (Boolean_Expression);
```



### A *do-while* Statement with a Multistatement Body

```
do
{
    Statement_1
    Statement_2
    .
    .
    Statement_Last
}while (Boolean_Expression);
```



# Review: **while** and **do-while**

- A while loop checks the Boolean expression at the beginning of the loop
  - A while loop **might never be executed!**
- A do-while loop checks the Boolean expression at the end of the loop
  - A do-while loop is **always executed at least once**

# What's the problem with the code?

- Prompt the user to enter the month, check if the input is valid (whether the value is between 1 and 12). Repeat it until the input is valid.

```
cout << "Enter the month:";  
int month;  
cin >> month;  
while (month >12 || month <1);  
    cout << "Invalid input. Try again:";  
    cin >> month;
```





# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Multiway Branches using switch statement
- Coding Style and Errors
- C++ Loop Statements: while, do/while loop,
  - for loop
- C++ Loop Statements: for loop
- Designing Loops

# For loop

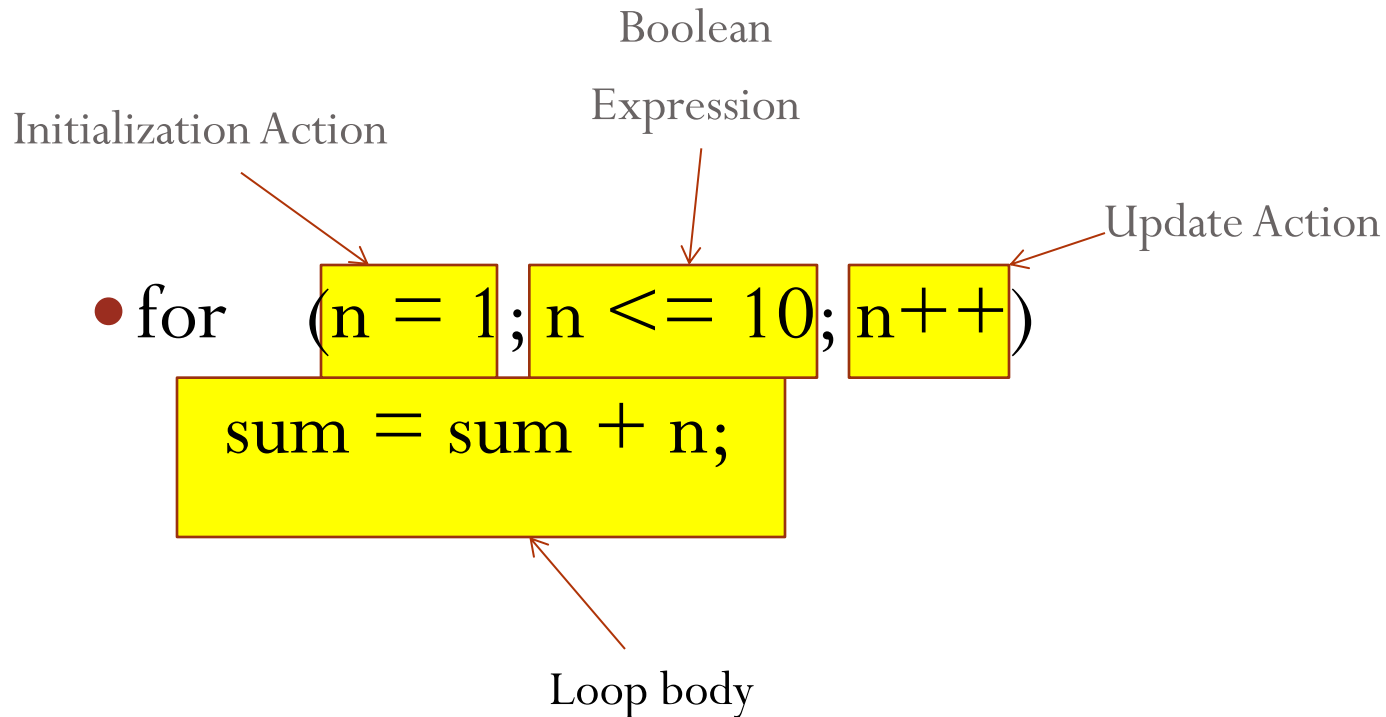
- **For loop:** more compact form
  - Normally used to write a loop with known iteration numbers
- for loop to add the numbers 1 - 10

```
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
```

- while loop for same task

```
sum = 0;
n = 1;
while (n <= 10)
{
    sum = sum + n;
    n++;
}
```

# For Loop Dissection



1. initialize n with value 1
2. Continue to iterate the body as long as `n <= 10`
3. Increment n by one after each iteration

# For loop => while loop

```
sum = 0;  
for (n = 1; n <= 10; n++)  
    sum = sum + n;
```

```
sum = 0;  
n = 1;  
while (n <= 10)  
{  
    sum = sum + n;  
    n++;  
}
```

# for Loop Alternative

- A for loop can also include a variable declaration in initialization action
  - `for (int n = 1; n <= 10; n++)`

This line means

- Create a variable, `n`, of type `int` and initialize it with 1
- Continue to iterate the body as long as `n <= 10`
- Increment `n` by one after each iteration

# The *for* Statement

---

## *for* Statement

### Syntax

```
for (Initialization_Action; Boolean_Expression; Update_Action)  
    Body_Statement
```

### Example

```
for (number = 100; number >= 0; number--)  
    cout << number  
        << " bottles of beer on the shelf.\n";
```

## Equivalent *while* loop

### Equivalent Syntax

```
Initialization_Action;  
while (Boolean_Expression)  
{  
    Body_Statement  
    Update_Action;  
}
```

### Equivalent Example

```
number = 100;  
while (number >= 0)
```

# for-loop Details

- Initialization and update actions of for-loops often contain more complex expressions

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

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

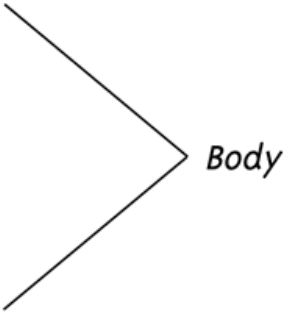
```
for(double x = pow(y,3.0); x > 2.0; x = sqrt(x) )
```

## for Loop with a Multistatement Body

---

### Syntax

```
for (Initialization_Action; Boolean_Expression; Update_Action)  
{  
    Statement_1  
    Statement_2  
    .  
    .  
    .  
    Statement_Last  
}
```



### Example

```
for (int number = 100; number >= 0; number--)  
{  
    cout << number  
        << " bottles of beer on the shelf.\n";  
    if (number > 0)  
        cout << "Take one down and pass it around.\n";  
}
```



# Exercise: how many Hello?

```
for(int count = 1; count <= 10; count++);  
    cout << "Hello\n";
```

- prints one "Hello", but not as part of the loop!
- The empty statement is the body of the loop
- `cout << "Hello\n";` is not part of the loop body!

# Break statement

- There are times to exit a loop before it ends
  - If loop checks for invalid input that would ruin a calculation, it is often best to end the loop
- Break statement can be used to exit a loop before normal termination
- In nested loops, break only exits loop in which break-statement occurs

## A *break* Statement in a Loop

```
//Sums a list of ten negative numbers.
#include <iostream>
using namespace std;

int main()
{
    int number, sum = 0, count = 0;
    cout << "Enter 10 negative numbers:\n";

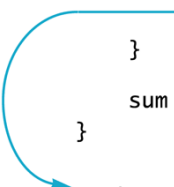
    while (++count <= 10)
    {
        cin >> number;

        if (number >= 0)
        {
            cout << "ERROR: positive number"
                 << " or zero was entered as the\n"
                 << count << "th number! Input ends "
                 << "with the " << count << "th number.\n"
                 << count << "th number was not added in.\n";
            break;
        }

        sum = sum + number;
    }

    cout << sum << " is the sum of the first "
         << (count - 1) << " numbers.\n";

    return 0;
}
```



### Sample Dialogue

```
Enter 10 negative numbers:
-1 -2 -3 4 -5 -6 -7 -8 -9 -10
ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.
4th number was not added in.
-6 is the sum of the first 3 numbers.
```

## Explicitly Nested Loops

```
//Determines the total number of green-necked vulture eggs
//counted by all conservationists in the conservation district.
#include <iostream>
using namespace std;

void instructions();

int main()
{
    instructions();

    int number_of_reports;
    cout << "How many conservationist reports are there? ";
    cin >> number_of_reports;

    int grand_total = 0, subtotal, count;
    for (count = 1; count <= number_of_reports; count++)
    {
        cout << endl << "Enter the report of "
            << "conservationist number " << count << endl;
        cout << "Enter the number of eggs in each nest.\n"
            << "Place a negative integer"
            << " at the end of your list.\n";
        subtotal = 0;
        int next;
        cin >> next;
        while (next >= 0)
        {
            subtotal = subtotal + next;
            cin >> next;
        }
        cout << "Total egg count for conservationist "
            << " number " << count << " is "
            << subtotal << endl;
        grand_total = grand_total + subtotal;
    }

    cout << endl << "Total egg count for all reports = "
        << grand_total << endl;
    return 0;
}
```

# Nested Loops

- The body of a loop may contain any kind of statement, including another loop, if statement
- When loops are nested, all iterations of inner loop are executed for each iteration of the outer loop

# Exercise

- Determine the output of the following?  

```
for(int count = 1; count < 5; count++)  
    cout << (2 * count) << " " ;
```
- Write a for loop to calculate the sum  $1/2 + 1/3 + 1/4 + \dots + 1/10$

# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Multiway Branches using switch statement
- Coding Style and Errors
- C++ Loop Statements: while, do/while loop,
  - for loop
- C++ Loop Statements: for loop
- Designing Loops

# Overview

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence
- Multiway Branches using Embedded if-else statement
- Multiway Branches using switch statement
- Coding Style and Errors
- C++ Loop Statements: while, do/while loop,
  - for loop
- C++ Loop Statements: for loop
- Designing Loops

# Designing Loops

- Designing a loop involves designing
  - The body of the loop
  - The initializing statements
  - The conditions for ending the loop



# Which Loop To Use?

- Choose the type of loop late in the design process
  - First design the loop using **pseudocode**
- Translate pseudocode into C++
  - **for-loops**: when doing numeric calculations, especially when using a **variable changed by equal amounts each time the loop iterates**
  - **While-loops**:
    - When a for-loop is not appropriate
    - When there are circumstances for which loop body should not be executed at all
  - **Do-while** loops:
    - When a for-loop is not appropriate
    - When **the loop body must be executed at least once**

# Sums and Products

- A common task is reading a list of numbers and computing the sum
- Pseudocode for this task might be:

*sum = 0;*

*repeat the following this\_many times*

*cin >> next;*

*sum = sum + next;*

*end of loop*

Pseudocode containing the line

**repeat the following "this many times"**

is often implemented with **a for-loop**

# for-loop for a sum

```
int sum = 0;
for(int count=1; count <= this_many; count++)
{
    cin >> next;
    sum = sum + next;
}
```

- A for-loop is generally the choice when there is a predetermined number of iterations
- sum must be initialized prior to the loop body!

# for-loop For a Product

- Forming a product is very similar to the sum example seen earlier

```
int product = 1;
for(int count=1; count <= this_many; count++)
{
    cin >> next;
    product = product * next;
}
```

- product must be initialized prior to the loop body
- Notice that product is initialized to 1, not 0!

# Ending a read input Loop

- Four common methods to terminate a loop that reads inputs from keyboard
  - List headed by size
    - When we can determine the size of the list beforehand
  - Ask before iterating
    - Ask if the user wants to continue before each iteration
  - List ended with a **sentinel value**
    - Using a particular value to signal the end of the list
  - Running out of input
    - Using the **eof** function to indicate the end of a file

# List Headed By Size

- The for-loops we have seen provide a natural implementation of the list headed by size method of ending a loop

```
int items;
cout << "How many items in the list?";
cin >> items;
for(int count = 1; count <= items; count++)
{
    int number;
    cout << "Enter number " << count;
    cin >> number;
    cout << endl;
    // statements to process the number
}
```

# Ask Before Iterating

- A while loop is used here to implement the **ask before iterating** method to end a loop

```
sum = 0;
cout << "Are there numbers in the list (Y/N)?";
char ans;
cin >> ans;

while ((ans == 'Y') || (ans == 'y'))
{
    //statements to read and process the number
    cout << "Are there more numbers(Y/N)? ";
    cin >> ans;
}
```

# List Ended With a Sentinel Value

- A while loop is typically used to end a loop using the **list ended with a sentinel value method**

```
cout << "Enter a list of nonnegative integers.\n"
      << "Place a negative integer after the list.\n";
sum = 0;
cin >> number;
while (number > 0)
{
    // statements to process the number
    cin >> number;
}
```

- Notice that the sentinel value is read, but not processed



# Running Out of Input

- while loop is typically used to implement **running out of input** method of ending a loop

```
ifstream infile;  
infile.open("data.dat");  
while (! infile.eof( ) )  
{  
    // read and process items from the file  
    // File I/O covered in Chapter 6  
}  
infile.close( );
```

# General Methods To Control Loops

- Counter controlled loops: the number of iterations predetermined before loop begins
  - E.g., list headed by size
- Ask before iterating
- Exit on flag condition: loops ended when a particular flag condition exists
  - A variable that changes value to indicate that some event has taken place is a flag
  - E.g., List ended with a sentinel value, running out of input

# Exit on Flag Caution

- Consider this loop to identify a student with a grade of 90 or better

```
int n = 1;
grade = compute_grade(n);
while (grade < 90)
{
    n++;
    grade = compute_grade(n);
}
cout << "Student number " << n
      << " has a score of " << grade << endl;
```

What if no student has a grade of 90 or higher?

# Exit On Flag Solution

- This code solves the problem of having no student grade at 90 or higher

```
int n=1;
grade = compute_grade(n);
while (( grade < 90) && ( n < number_of_students))
{
    // same as before
}
if (grade > 90)
    // same output as before
else
    cout << "No student has a high score.";
```

a second flag: ensure there are still students to consider

# Debugging Loops

- **Off-by-one errors:** loop executes one too many or one too few times
  - Check your comparison:  
should it be  $<$  or  $<=$ ?
  - Check that the initialization uses the correct value
  - Does the loop handle the zero iterations case?
- **Infinite loops:** usually result from a mistake in Boolean expression that controls the loop
  - Check direction of inequalities:  
 $<$  or  $>$  ?
  - Test for  $<$  or  $>$  rather than equality ( $==$ )
    - Remember that doubles are really only approximations

# Loop Debugging Tips

- Be sure that the mistake is really in the loop
- Trace variable to observe how the variable changes
  - Tracing a variable is watching its value change during execution
    - Many systems include utilities to help with this
  - `cout` statements can be used to trace a value

# Debugging Example

- following code is supposed to conclude with variable *product* containing the product of the numbers 2 through 5, i.e.,  
 $2*3*4*5$

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
```

# Tracing Variables

- Add temporary cout statements to trace variables

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
    cout << "next = " << next
         << "product = " << product
         << endl;
}
```

next=3 product=3

next=4 product=12

next=5 product=60

Problem: 2 is not multiplied into product...



# First Fix

- Solve the problem by moving the statement `next++`

```
int next = 2, product = 1;
while (next < 5)
{
    product = product * next;
    next++;

    cout << "next = " << next
    << "product = " << product
    << endl;
}
```

- There is still a problem!

# Second Fix

- New problem: loop never multiplies by 5
  - The fix is to use `<=` instead of `<` in our comparison

```
int next = 2, product = 1;
while (next <= 5)
{
    product = product * next;
    next++;
}
```

# Loop Testing Guidelines

- Every time a program is changed, it must be retested
  - Changing one part may require a change to another
- Every loop should at least be tested using input to cause:
  - Zero iterations of the loop body
  - One iteration of the loop body
  - One less than the maximum number of iterations
  - The maximum number of iterations

# Starting Over

- Sometimes it is more efficient to throw out a buggy program and start over
  - The new program will be easier to read
  - The new program is less likely to be as buggy
  - You may develop a working program faster than if you repair the bad code
    - The lessons learned in the buggy code will help you design a better program faster

# Summary

- Using Boolean Expressions
  - Truth table, evaluate boolean expression
  - Precedence Rules
- Multiway Branches
  - Nested if-else statement
  - Switch statement
- More about C++ Loop Statements
- Designing Loops