# COMP 2710
# Software Construction

## Chapter2-1: Flow control and Pointers

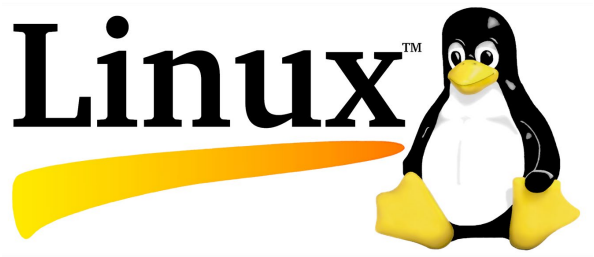
Dr. Xuechao Li

AUBURN UNIVERSITY

SAMUEL GINN COLLEGE OF ENGINEERING

# Prepare Your Development Environment:

## Three Candidate Environments




**Linux Environment:**
No IDE: vi, g++, gdb

**Windows Environment:**
Eclips IDE, MinGW or Cygwin


**Mac OS Environment:**
xCode IDE, Clang C++ compiler

# Identifier Style

- careful selection of identifiers makes program more understandable
- identifiers should be
  - short enough to be reasonable to type (single word is norm)
    - standard abbreviations are acceptable
  - long enough to be understandable
- two styles of identifiers
  - C-style - terse, use abbreviations and underscores to separate the words, never use capital letters for variables
  - Camel Case - if multiple words: capitalize, do not use underscores
    - variant: first letter lowercased
- pick identifier style and use it consistently
- ex:

| Camel Case 1 | C-style | Camel Case 2 |
|---|---|---|
| `Min` | `min` | `min` |
| `Temperature` | `temperature` | `temperature` |
| `CameraAngle` | `camera_angle` | `cameraAngle` |
| `CurrentNumberPoints` | `cur_point_nmbr` | `currentNumberPoints` |

# Output

- to do input/output, at the beginning of your program insert
```
#include <iostream>
using std::cout; using std::endl;
```
- C++ uses streams for input an output
- *stream* – a sequence of data to be processed
  - *input stream* – data to be input into program
  - *output stream* - data generated by the program to be output

- variable values as well as strings of text can be output to the screen using `cout` (console output) stream:
```
cout << numberOfBars;
cout << "candy bars";
cout << endl;
```
- `<<` is *insertion operator,* it inserts data into the output stream
  - anything within double quotes will be output *literally* (without changes)
    `"candy bars taste good"`
  - note the space before letter " `c`" - the computer does not insert space on its own
- keyword `endl` tells the computer to start the output from the next line

# Input

- `cin` (Console INput) – stream used to give variables user-input values
- need to add the following to the beginning of your program

```
using std::cin;
```

- when the program reaches the input statement it pauses until the user types something and presses <Enter> key
- therefore, it is beneficial to precede the input statement with some explanatory output called *prompt*:

```
cout << "Enter the number of candy bars";
cout << "and weight in ounces.\n";
cout << "then press return\n";
cin >> numberOfBars >> oneWeight;
```

- `>>` is *extraction operator*
- *dialog* – collection of program prompts and user responses
- input operator (similar to output operator) can be stacked
- *input token* – sequence of characters separated by white space (spaces, tabs, newlines)
- the values typed are inserted into variables when <Enter> is pressed
  - if more values needed - program waits
  - if extra typed - are used in next input statements if needed

# Formatting Real Numbers

- Real numbers (type double) produce a variety of outputs

```
double price = 78.5;
cout << "The price is $" << price << endl;
```

  - The output could be any of these:

```
The price is $78.5
The price is $78.500000
The price is $7.850000e01
```

  - The most unlikely output is:

```
The price is $78.50
```

# Showing Decimal Places

- cout includes tools to specify the output of type double

- To specify fixed point notation
  - setf(ios::fixed)
- To specify that the decimal point will always be shown
  - setf(ios::showpoint)
- To specify that two decimal places will always be shown
  - precision(2)

- Example:

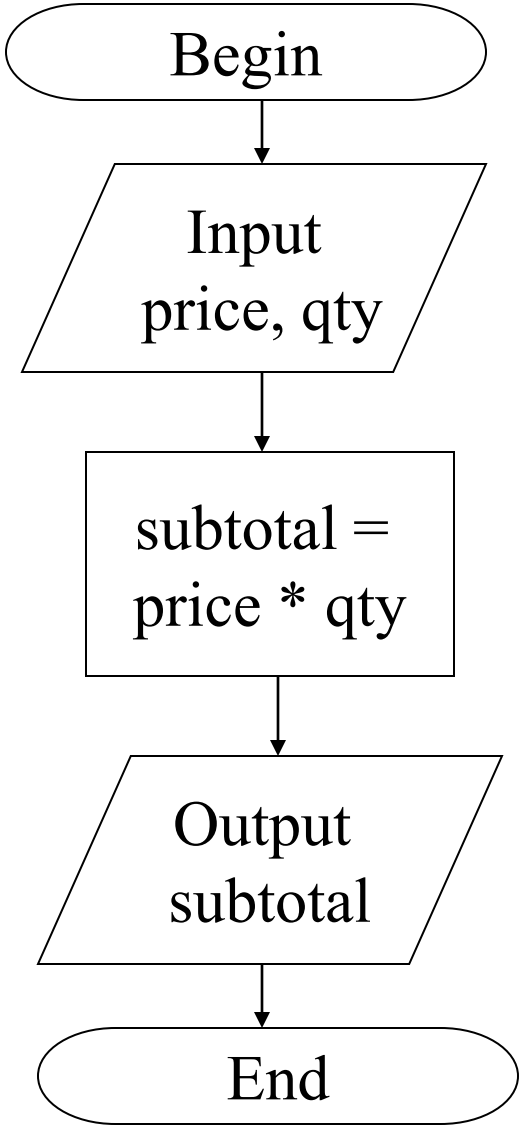
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout   << "The price is "
       << price << endl;
```

# Flow Control Structures

- The order in which statements are executed.
- There are four structures.

1. Sequence Control Structure
2. Selection Control Structure
   - Also referred to as branching (if and if-else)
3. Case Control Structure (switch)
4. Repetition Control Structure (loops)

# Flowchart – Sequence Control

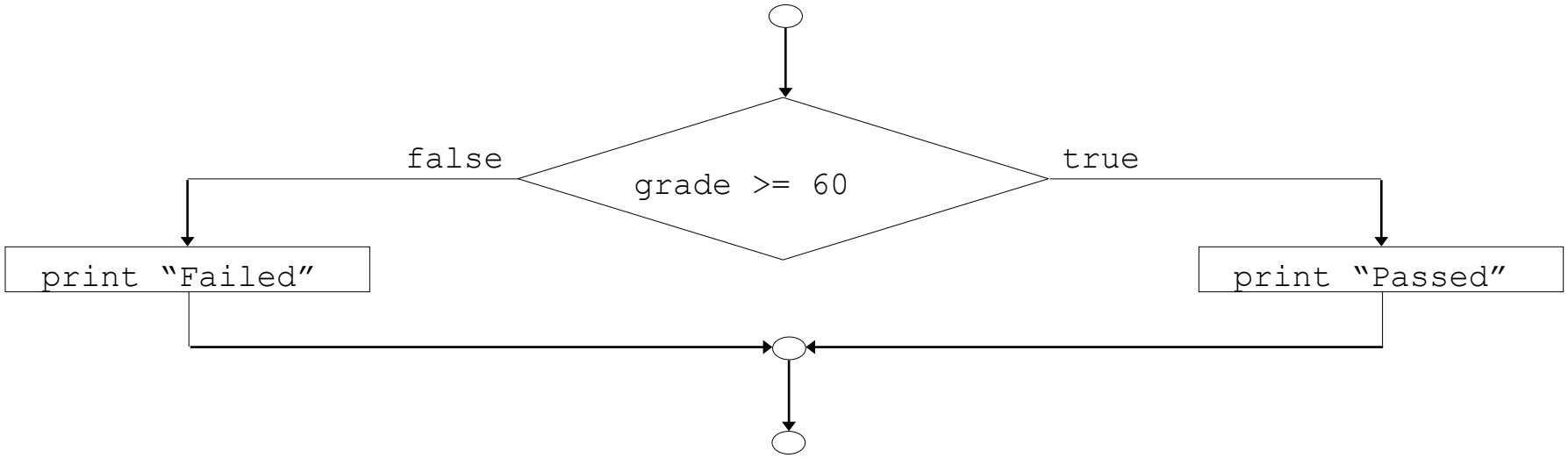


# `if/else` Selection Structure

- Ternary conditional operator (`?:`)
  - Three arguments (condition, value if **`true`**, value if **`false`**)
- Code could be written:

```
cout << ( grade >= 60 ? "Failed" : "Passed" );
```

Condition  Value if true  Value if false



# The switch Multiple-Selection Structure

- **`switch`**
  - Useful when variable or expression is tested for multiple values
  - Consists of a series of **`case`** labels and an optional **`default`** case
  - **`break`** is (almost always) necessary

# Switch

```
switch (letter) {
    case 'N': cout << "New York\n";
              break;
    case 'L': cout << "London\n";
              break;
    case 'A': cout << "Amsterdam\n";
              break;
    default:  cout << "Somewhere else\n";
              break;
}
```

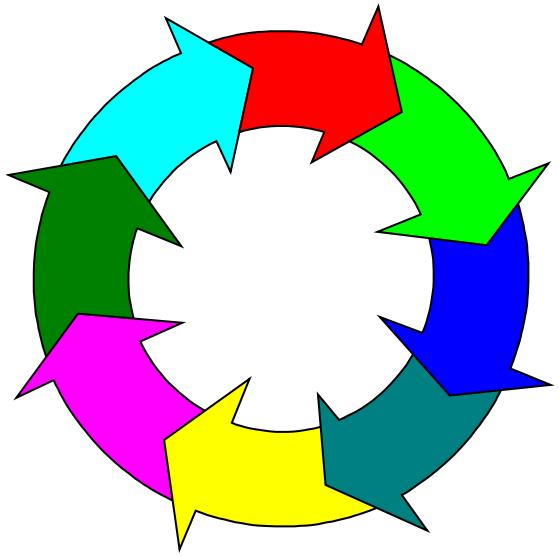
# Iteration statements

- while-statement syntax

```
while (expression)
    statement
```

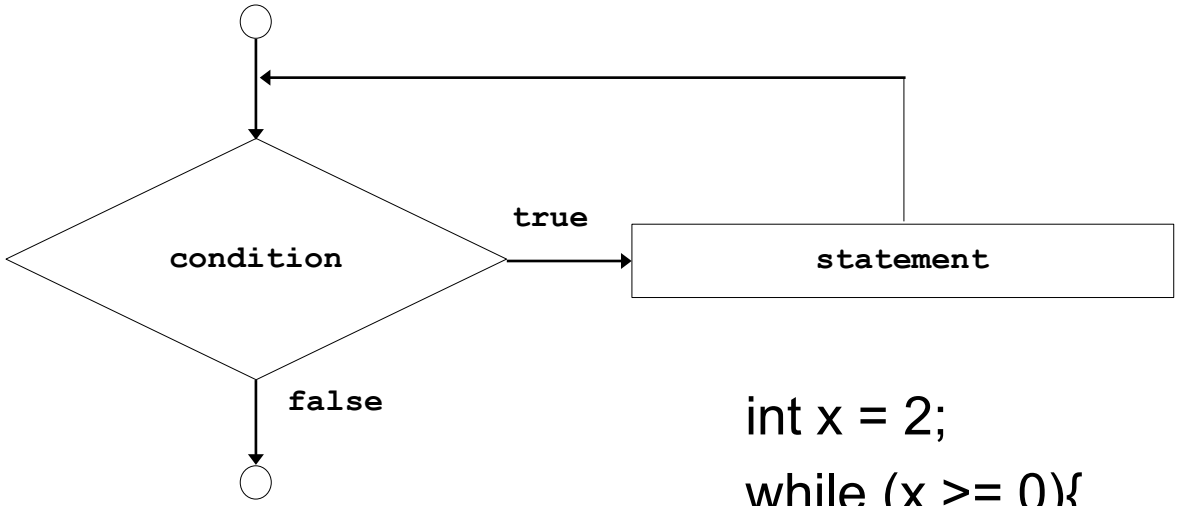
- semantics

It's a pre-test loop.



# The while Repetition Structure

- Flowchart of **`while`** loop




```
int x = 2;
while (x >= 0){
    cout << "Value of x is : " << x << endl;
    x --;
}
```

# The for Repetition Structure

- The general format when using **`for`** loops is

```
for ( initialization; LoopContinuationTest;
            increment )
   statement
```

- Example:

```
for( int counter = 1; counter <= 10;
  counter++ )
   cout << counter << endl;
```

  – Prints the integers from one to ten

# An example: Matrix Multiplication

$$\begin{bmatrix} * & * & * & * \\ 0 & 1 & 2 & 3 \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \times \begin{bmatrix} * & 0 & * & * \\ * & 1 & * & * \\ * & 2 & * & * \\ * & 3 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * \\ * & 14 & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}$$

```
for(i = 0; i < n; i++){
    for(j = 0; j < n; j++){
        for(k = 0; k < n; k++){
            c[i*n + j] += a[i*n + k] * b[k*n + j];
        }
    }
}
```

# The **break** and **continue** Statements--1

- **`Break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement:
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure

# The **break** and **continue** Statements--2

- **`Continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure and proceeds with the next iteration of the loop
  - In **`while`** and **`do/while`**, the loop-continuation test is evaluated immediately after the **`continue`** statement is executed
  - In the **`for`** structure, the increment expression is executed, then the loop-continuation test is evaluated

# How “break” works

```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

# How “continue” works

```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}
```

# Break/Continue

| Allowed or not | Break statement | Continue statement |
|---|---|---|
| For loop | YES! | YES! |
| While loop | YES! | YES! |
| Do-while loop | YES! | YES! |
| Switch case | YES! | ✗ |
| If statement | ✗ | ✗ |
| If else statement | ✗ | ✗ |