COMP 2710
Software Construction

Chapter1: Basics and Flow control

Dr. Xuechao Li

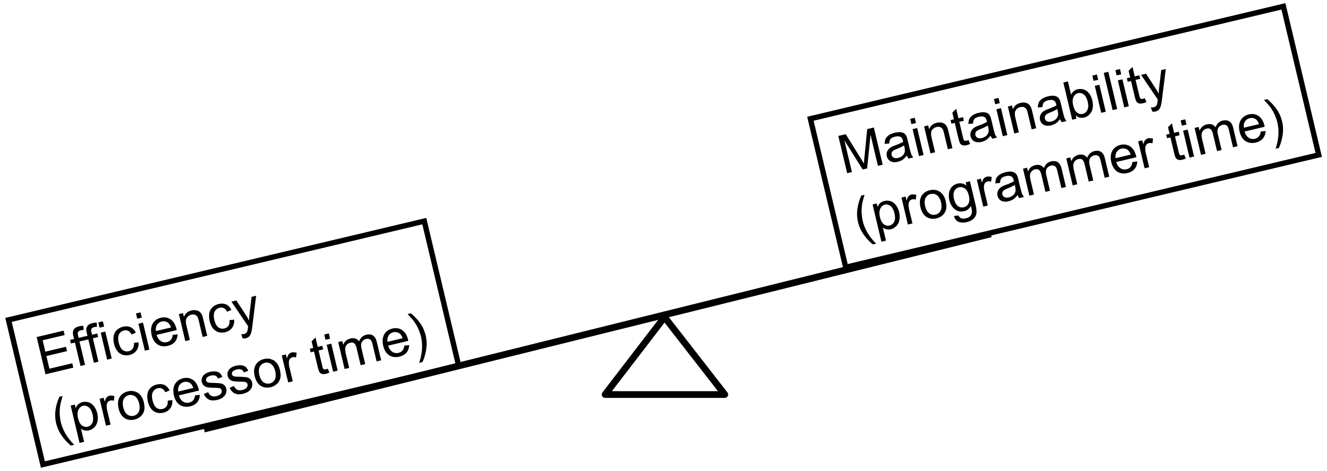
AUBURN
UNIVERSITY

Samuel Ginn
College of Engineering

Why C++?

- Popular and relevant (used in nearly every application domain):
  - end-user applications (Word, Excel, PowerPoint, Photoshop, Acrobat, Quicken, games)
  - operating systems (Windows 9x, NT, XP; IBM's K42; some Apple OS X)
  - large-scale web servers/apps (Amazon, Google)
  - central database control (Israel's census bureau; Amadeus; Morgan-Stanley financial modeling)
  - communications (Alcatel; Nokia; 800 telephone numbers; major transmission nodes in Germany and France)
  - numerical computation / graphics (Maya)
  - device drivers under real-time constraints
- Stable, compatible, scalable

Efficiency and Maintainability



90/10 rule: 10% of your program will take 90% of the processor time to run

→ optimize what needs to be optimized, but no more

→ focus on design

Programming paradigms

- *procedural* – implement algorithms via functions (variables, functions, etc.)
- *modular* – partition program into modules (separate compilation)
- *object-oriented* – divide problem into classes (data hiding, inheritance)
- *abstract* – separate interface from implementation (abstract classes)
- *generic* – manipulate arbitrary data types (STL: containers, algorithms)

What is object-oriented?

- Encapsulation
  “black box” – internal data hidden

© SDC

- Inheritance
  related classes share implementation
  and/or interface

- Polymorphism
  ability to use a class without knowing its type

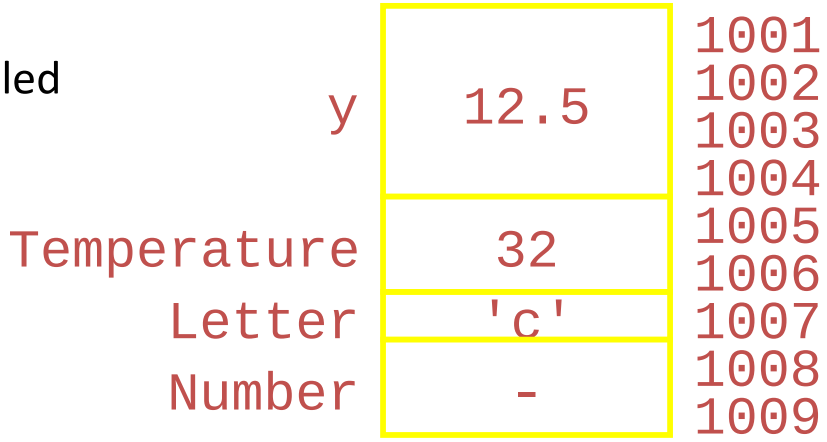
AUBURN UNIVERSITY
SAMUEL GINN COLLEGE OF ENGINEERING

Java Simplifications of C++

- no pointers — **just references**
- no functions — can declare **`static`** methods
- no global variables — use **`public static`** variables
- no destructors — **garbage collection** and **`finalize`**
- no linking — dynamic class loading
- no header files — can define **`interface`**
- no operator overloading — only method overloading
- no member initialization lists — call **`super`** constructor
- no preprocessor — **`static final`** **constants** and automatic inlining
- no multiple inheritance — **implement multiple interfaces**
- no structs, unions, enums — **typically not needed**

Variables

- *variable* is a named memory location
- variable *value* is data stored in variable
  - variable always has a value
- compiler removes variable name and assigns memory location
  - however, it is convenient to think that memory locations are labeled with variable names



y 12.5

Temperature 32

Letter 'c'

Number -

1001
1002
1003
1004
1005
1006
1007
1008
1009

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`

Assignment

- *assignment statement* is an order to the computer to set the value of the variable on the left hand side of equal sign to what is written on the right hand side

```
variable = value;
```

- it looks like a math equation, but it is not
- example:

```
numberOfBars = 37;
totalWeight = oneWeight;
totalWeight = oneWeight * numberOfBars;
numberOfBars = numberOfBars + 3;
```

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:

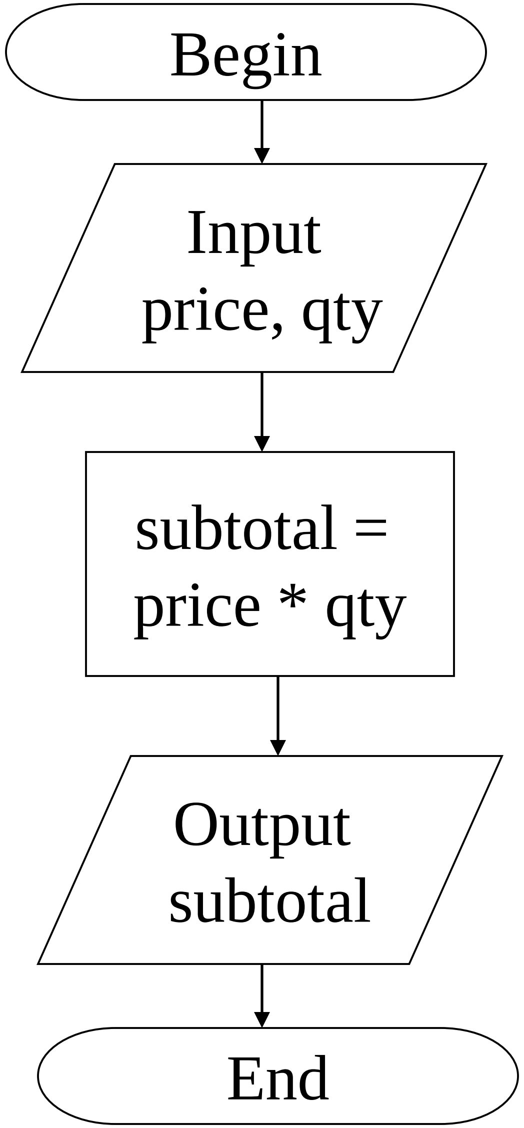
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout   << "The price is "
       << price << endl;
```

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



Begin

Input price, qty

subtotal = price * qty

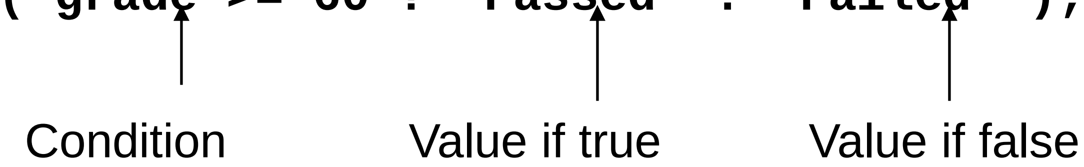
Output subtotal

End

`if/else` Selection Structure

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

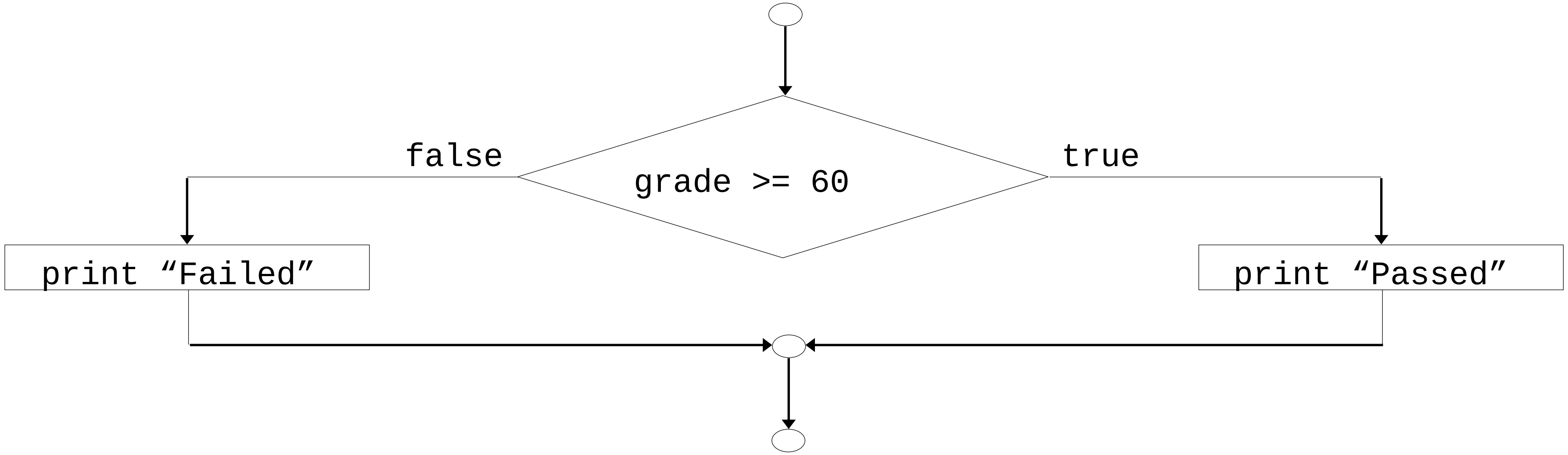
```
cout << ( grade >= 60 ? “Passed” : “Failed” );
```



Condition

Value if true

Value if false



The if...else Statement

```
if (booleanExpression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```

Multiple Alternative if Statements

```
if (score >= 90)
  grade = ‘F’;
else
  if (score >= 80)
    grade = ‘D’;
  else
    if (score >= 70)
      grade = ‘C’;
    else
      if (score >= 60)
        grade = ‘B’;
      else
        grade = ‘A’;
```

```
if (score >= 90)
  grade = ‘F’;
else if (score >= 80)
  grade = ‘D’;
else if (score >= 70)
  grade = ‘C’;
else if (score >= 60)
  grade = ‘B’;
else
  grade = ‘A’;
```

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

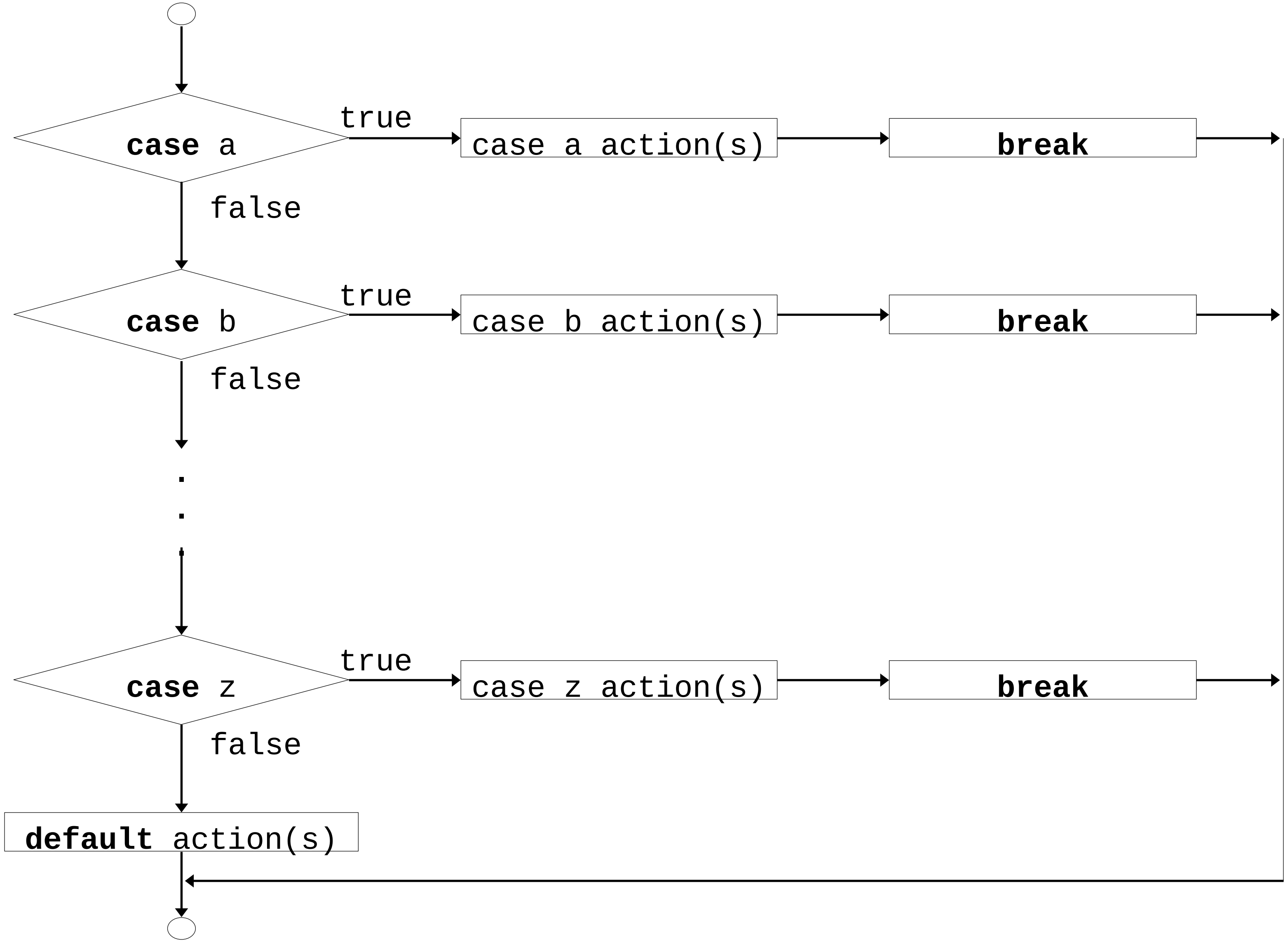
```
switch (letter) {
   case ‘N’: cout < “New York\n”;
             break;
   case ‘L’: cout < “London\n”;
             break;
   case ‘A’: cout < “Amsterdam\n”;
             break;
   default:  cout < “Somewhere else\n”;
             break;
}
```

```
switch (expression) {
      case val1:
                    statement
                    break;
      case val2:
                    statement
                    break;
          ….

      case valn:
                    statement
                    break;
      default:
                    statement
                    break;
}
```

↔

```
if  (expression == val1)
          statement
else if (expression==val2)
          statement
          ….

else if (expression== valn)
          statement
else
          statement
```

Flowchart--Switch



AUBURN UNIVERSITY
SAMUEL GINN COLLEGE OF ENGINEERING

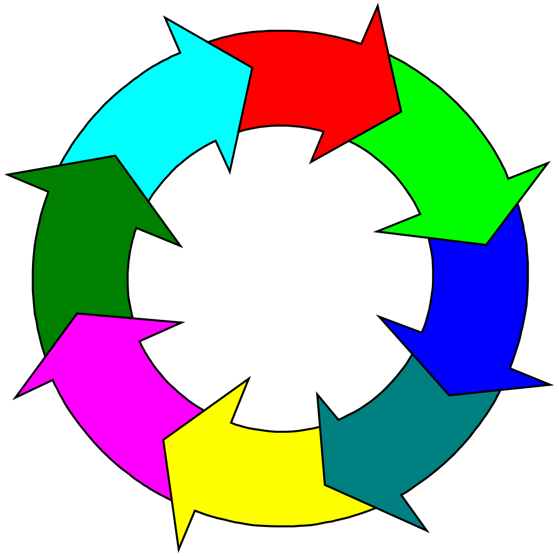
Iteration statements

- while-statement syntax

```
while (expression)
    statement
```

- semantics

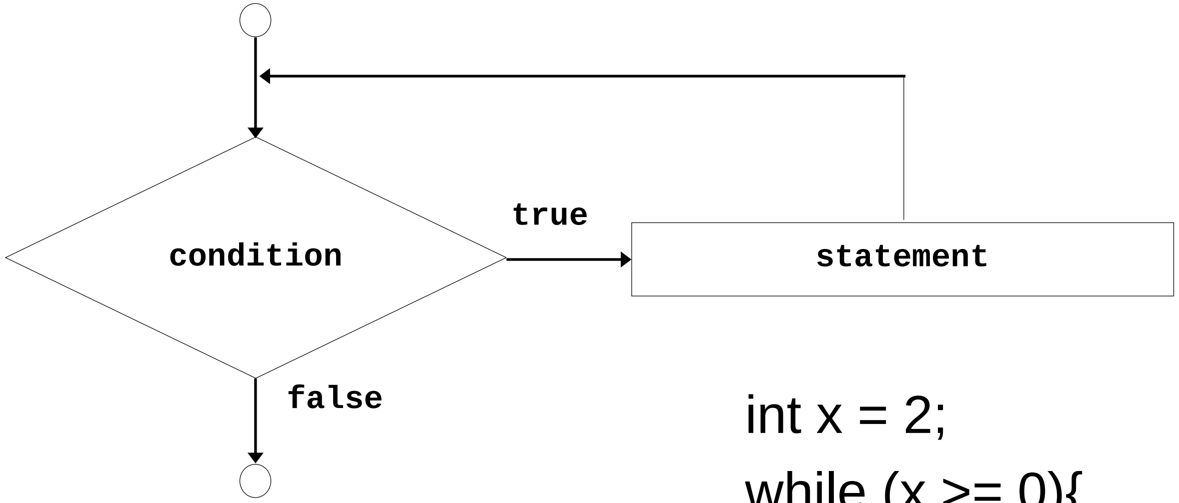
It's a pre-test loop.



AUBURN UNIVERSITY
SAMUEL GINN COLLEGE OF ENGINEERING

The while Repetition Structure

- Flowchart of **`while`** loop



condition

true

statement

false

```
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;
```

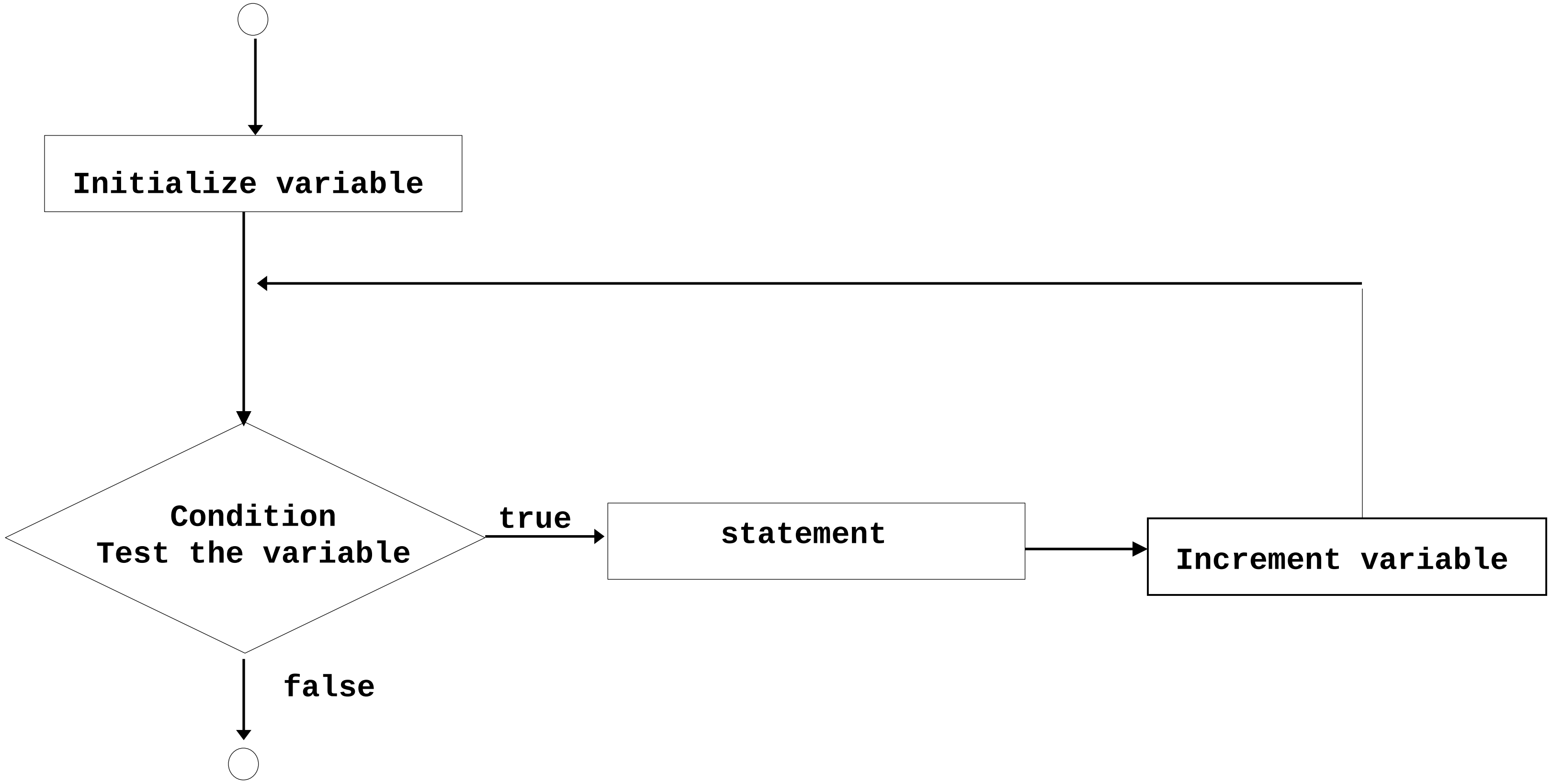
  – Prints the integers from one to ten

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];
        }
    }
}
```

Flowchart--**for**



while < == > for

- **For** loops can usually be rewritten as **while** loops:

```
    initialization;
    while ( loopContinuationTest){
       statement
       increment;
    }
```

- Initialization and increment as comma-separated lists

```
    for (int i = 0, j = 0;  j + i <= 10;
     j++, i++)
       cout << j + i << endl;
```

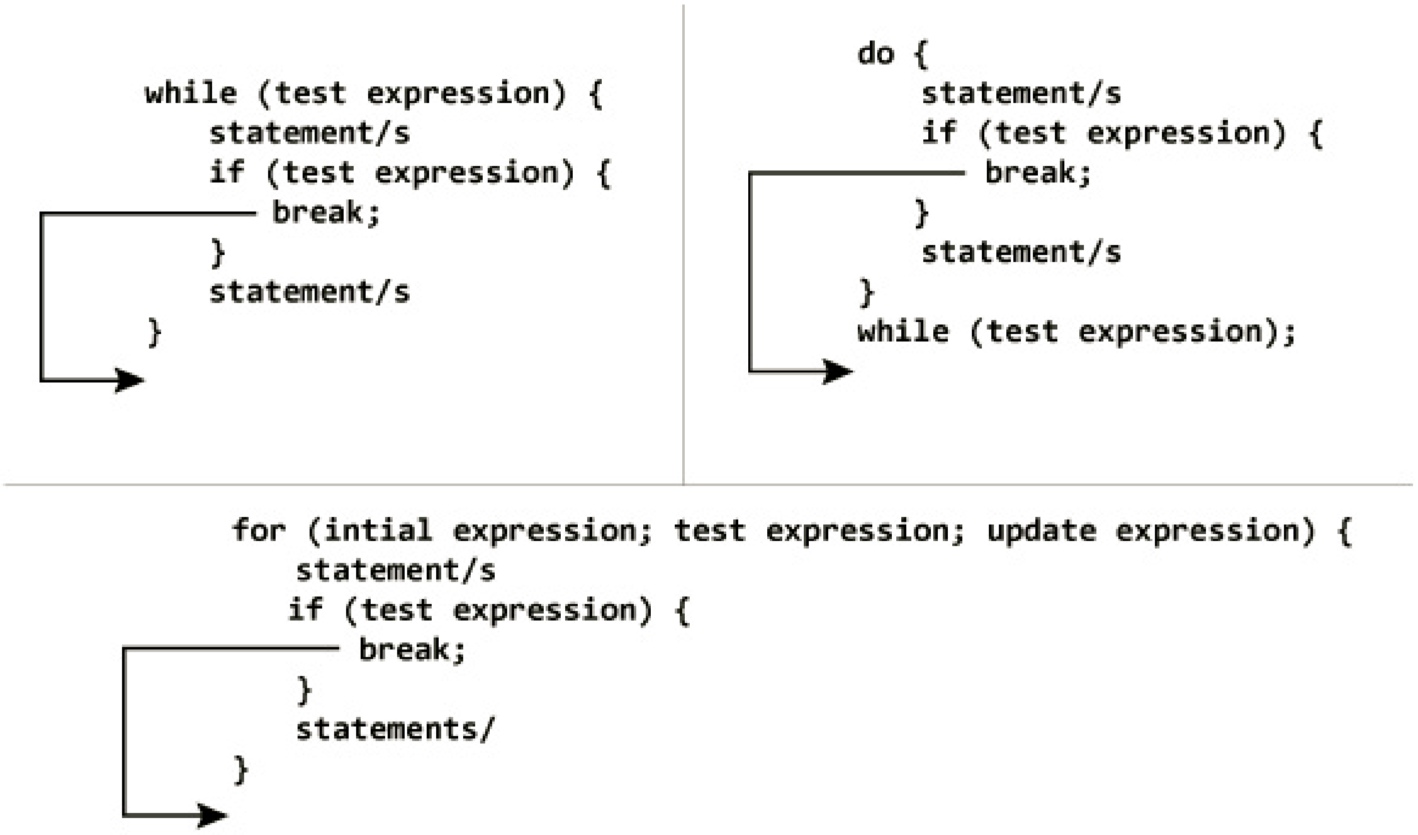
The **break** and **continue** Statements--1

- **`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

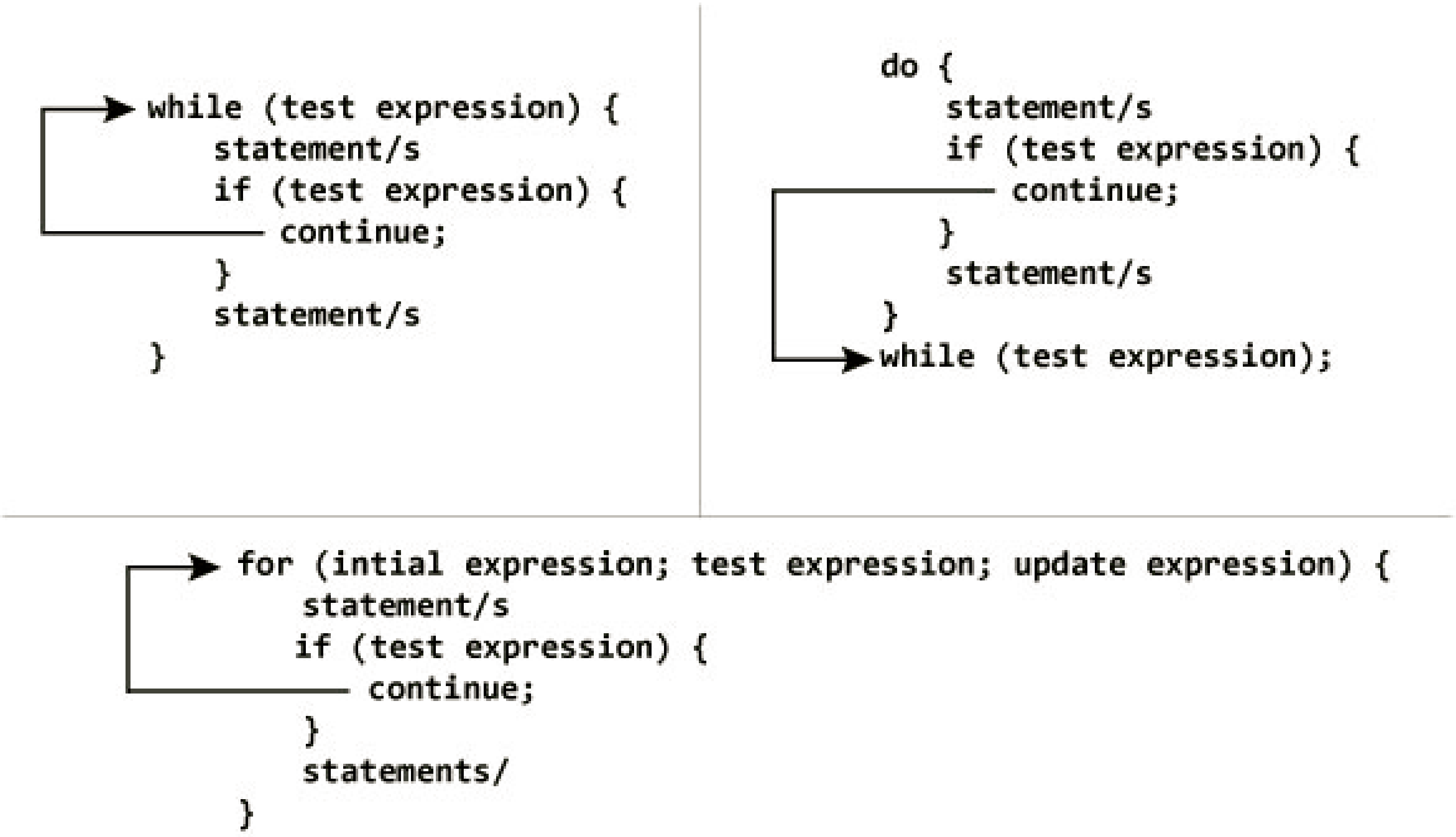


```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
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
}
```

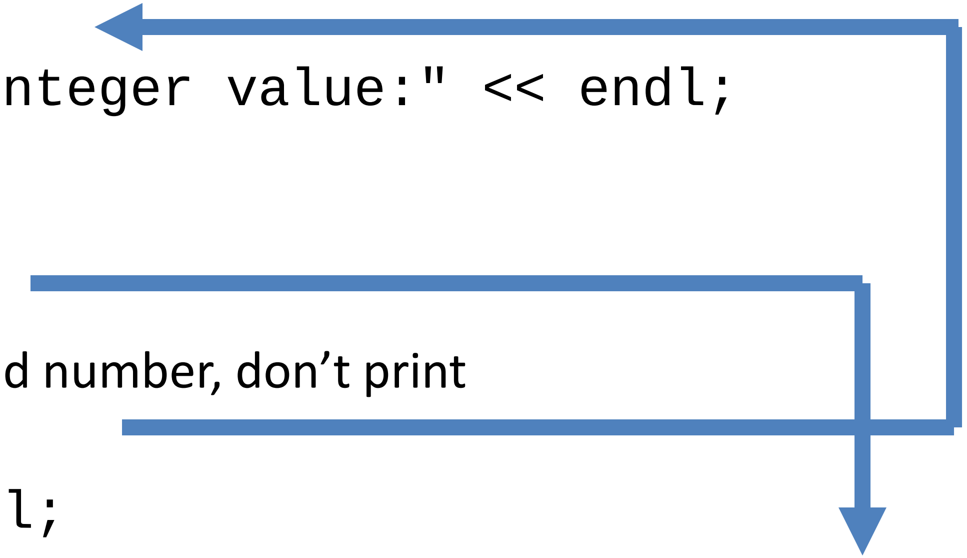
```
do {
     statement/s
     if (test expression) {
          continue;
     }
     statement/s
}
while (test expression);
```

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

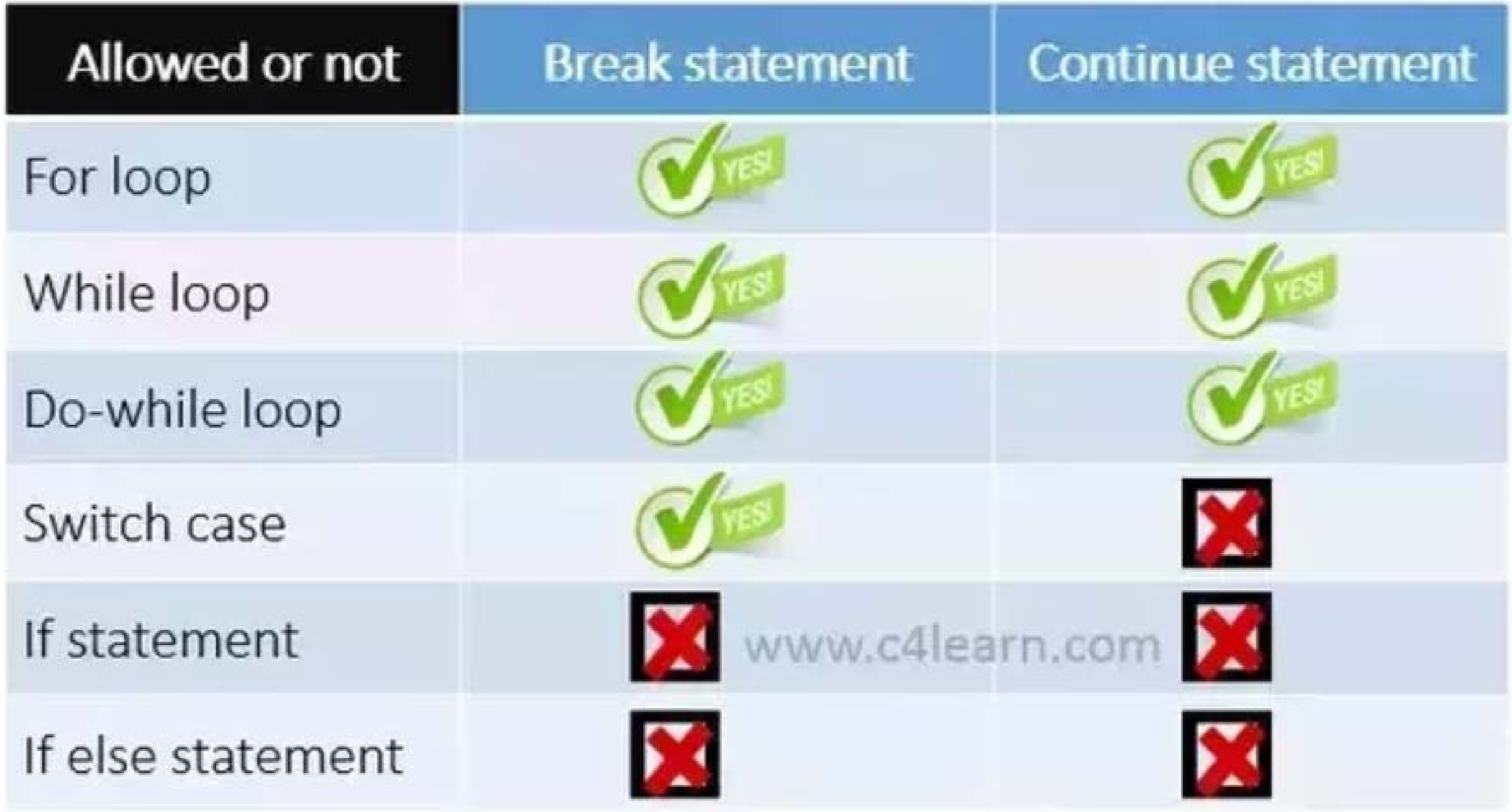
The `continue` Statement

- Causes an immediate jump to the loop test

```
int next = 0;
while (true){
   cout << "Enter an integer value:" << endl;
   cin >> next;
   if (next < 0)
      break;
   if (next % 2)    //odd number, don’t print
      continue;
   cout << next << endl;
}
cout << “negative num so here we are!” << endl;
```



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	☒	☒