Ch3: if…else, double, Type Conversion, while, Operators

Control Structures

Control Structures

It is demonstrated that all programs could be written in terms of only 3 control structures:

- **Sequence Structure**: Unless directed otherwise, the C++ statements are executed one after the other in the order in which they're written (sequential execution).
- **Selection Structure**: C++ has 3 types of selection statements.
  - if statement (single-selection statement)
  - if…else statement (double-selection statement)
  - switch statement (multiple-selection statement)
- **Iteration Structure**: C++ provides 3 iteration statements that enable programs to perform statements repeatedly as long as a condition remains true.
  - while statement
  - do…while statement
  - for statement (and range-based for statement)
- In algorithms, these 3 control structures can be combined in only two ways: **stacking** (one after another) and **nesting** (one inside another).

if…else **Double-Selection Statement**

if…else (or Double-Selection Statement)

if…else statement (or double-selection statement) performs an action (or group of actions) if a condition is true and performs a *different* action (or group of actions) if the condition is false. Conditions are usually formed by using the relational and equality operators.

Body of if and else that can be a single statement or a **Block** of several statements in {}.

```
if (condition){
  statement(s);
}
else {
  statement(s);
}
```

- You do not need to use braces, { }, around single-statement bodies. However, it is always recommended to enclose all the statement bodies in braces to avoid logic errors called the **dangling-else problem**.

```
if (condition)
  statement;
else
  statement;
```

- The **indentation** of the statement(s) in the bodies of an if…else statement, which enhances readability, is optional, but recommended. If there are several levels of indentation, each level should be indented the same additional amount of space. Many IDEs do indentation automatically.

Dangling-else Problem & Local Variables in Blocks

- These two snippets are not identical:

```
if (grade >= 60) {
  std::cout << "Passed";
}
else {
  std::cout << "Failed\n";
  std::cout << "You must take this course again.";
}
```

```
if (grade >= 60)
  std::cout << "Passed";
else
  std::cout << "Failed\n";
  std::cout << "You must take this course again.";
```

The last line (statement) is outside the body of the else part of the if…else statement and would execute regardless of the condition.

- In general, a **variable declared in a block** (enclosed in braces {}) is a **local variable** and can be used only from the line of its declaration to the closing right brace of the block (This restricted use is known as the variable's **scope**, which defines where it can be used in a program). The blocks can appear in **all control structures** and **functions**.

Remarks

- It is possible to have an **empty statement**, by placing a semicolon (;) where a statement would normally be or using and empty block.

```
if (grade >= 60) {
   std::cout << "Passed";
}
else {
}
```

```
if (grade >= 60)
   std::cout << "Passed";
else
   ;
```

- Placing a semicolon after the parenthesized condition in an if or if…else statement leads to a **logic error** in if statements and a **syntax error** in if…else statements (when the if-part contains a body statement).

Nested if…else Statements

A program can test multiple cases by placing if…else statements inside other if…else statements to create **Nested if…else statements**.

```
if (grade >= 90) {
  std::cout << "A";
}
else {
  if (grade >= 80) {
    std::cout << "B";
  }
  else {
    if (grade >= 70) {
      std::cout << "C";
    }
    else {
      if (grade >= 60) {
        std::cout << "D";
      }
      else {
        std::cout << "F";
      }
    }
  }
}
```

- Nested if…else statements are usually preferred to be identically written as

```
if (grade >= 90) {
  std::cout << "A";
}
else if (grade >= 80) {
  std::cout << "B";
}
else if (grade >= 70) {
  std::cout << "C";
}
else if (grade >= 60) {
  std::cout << "D";
}
else {
  std::cout << "F";
}
```

- This form avoids deep indentation of the code to the right (although the compiler ignores indentations).

Sample Program: Computing Student's Letter Grade

Class **Student** defined in the header **Student.h**

```
// Student class that stores a student name and average.
#include <string>
class Student {
public:
  // constructor initializes data members
  Student(std::string studentName, int studentAverage) : name{studentName} {
    // sets average data member if studentAverage is valid
    setAverage(studentAverage);
  }

  // sets the Student's name
  void setName(std::string studentName) {
    name = studentName;
  }

  // sets the Student's average
  void setAverage(int studentAverage) {
    // validate that studentAverage is > 0 and <= 100; otherwise,
    // keep data member average's current value
    if (studentAverage > 0) {
      if (studentAverage <= 100) {
        average = studentAverage; // assign to data member
      }
    }
  }
  // retrieves the Student's name
  std::string getName() const {
    return name;
  }
```

```
  // retrieves the Student's average
  int getAverage() const {
    return average;
  }
  // determines and returns the Student's letter grade
  std::string getLetterGrade() const {
    // initialized to empty string by class string's constructor
    std::string letterGrade;
    if (average >= 90) {
      letterGrade = "A";
    }
    else if (average >= 80) {
      letterGrade = "B";
    }
    else if (average >= 70) {
      letterGrade = "C";
    }
    else if (average >= 60) {
      letterGrade = "D";
    }
    else {
      letterGrade = "F";
    }
    return letterGrade;
  }
private:
  std::string name;
  int average{0}; // initialize average to 0
}; // end class Student
```

Sample Program: Computing Student's Letter Grade (cont.)

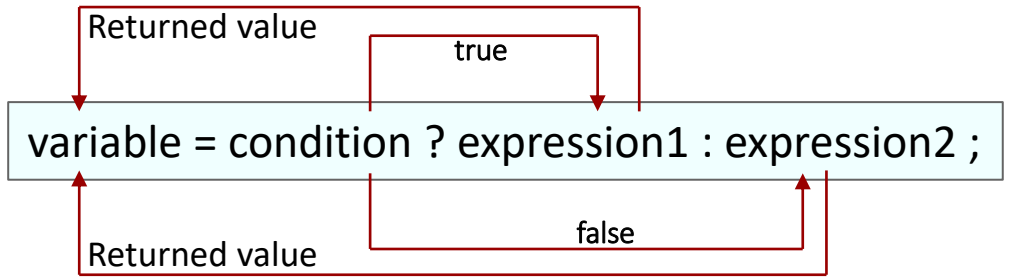
.cpp source-code file

```
// Create and test Student objects.
#include <iostream>
#include "Student.h"

int main() {
  Student account1{"Jane Green", 93};
  Student account2{"John Blue", 72};

  std::cout << account1.getName() << "'s letter grade equivalent of "
    << account1.getAverage() << " is: "
    << account1.getLetterGrade() << "\n";
  std::cout << account2.getName() << "'s letter grade equivalent of "
    << account2.getAverage() << " is: "
    << account2.getLetterGrade() << std::endl;
}
```

Conditional Operator (?:)

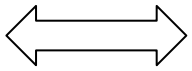
The **Conditional Operator (?:)** is C++'s only **ternary operator** (i.e., an operator that takes three operands) that can be used in place of an if…else statement (with single-statement blocks) to make the code shorter and clearer.

Form 1:

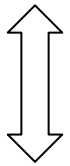



```
std::cout << ( grade >= 60 ? "Passed" : "Failed");
```

```
var = (y < 10) ? 30 : 40;
```



```
if (grade >= 60) {
  std::cout << "Passed";
}
else {
  std::cout << "Failed";
}
```



Form 2:

```
condition ? statement1 : statement2 ;
```

statements without semicolon

```
grade >= 60 ? std::cout << "Passed" : std::cout << "Failed“ ;
```

- Conditional expressions can appear in some program locations where if…else statements cannot.

UML Activity Diagram

UML Activity Diagram

An **UML Activity Diagram** models the workflow (activity) of a portion of a software system or algorithm by several symbols. These symbols are connected by **transition arrows**, which represent the flow of the activity (or the order in which the actions should occur).

Symbols:

rectangle with rounded corners

Containing an **action expression**

solid circles

Representing **initial state** or entry point

solid circle surrounded by a hollow circle

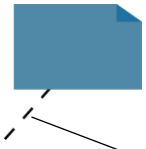
Representing **final state** or exit point

diamond

Indicating a **decision symbol** or a **merge symbol**

transition arrow

Represent the flow of the activity



rectangles with the upper-right corners folded over

Representing **UML notes** (like comments in C++)

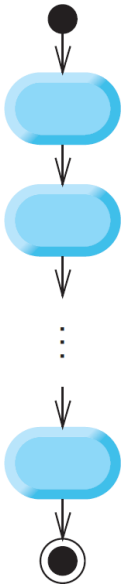
dotted line

Connecting each note with the element it describes

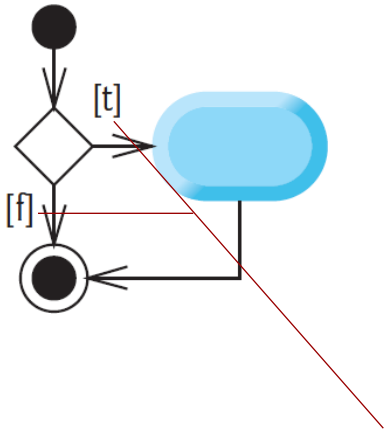
UML Activity Diagram for Sequence Structure and if Single-Selection Statement

Like pseudocode, activity diagrams help you develop and represent algorithms. All control structures can be modeled as activity diagrams.

Sequence Structure



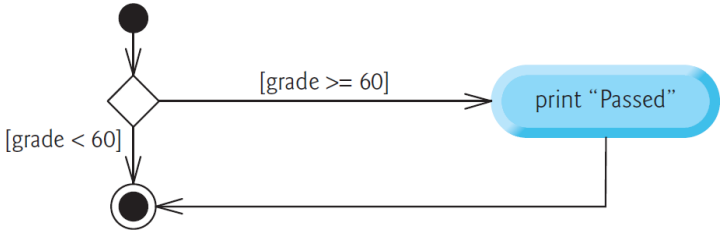
if statement
(single selection)



guard conditions (specified by []) can be true or false.

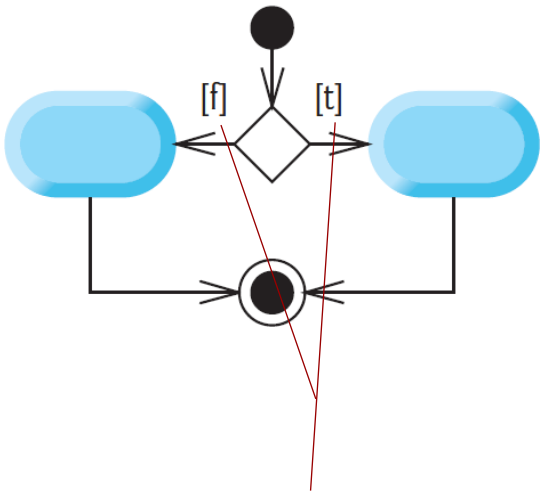
Example:

```
if (grade >= 60) {
   std::cout << "Passed";
}
```



UML Activity Diagram for if...else Double-Selection Statement

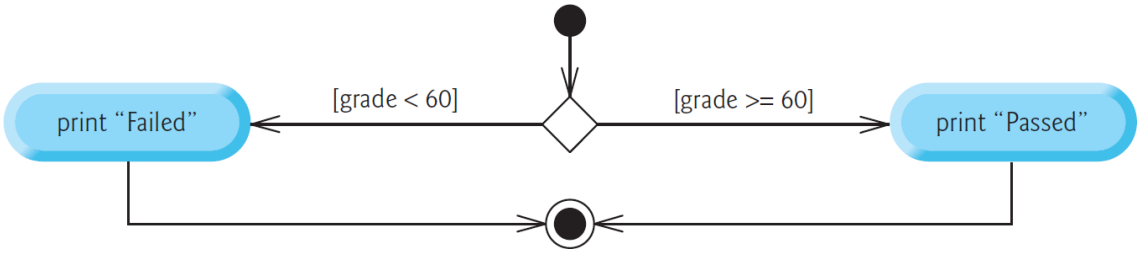
`if...else` statement
(double selection)



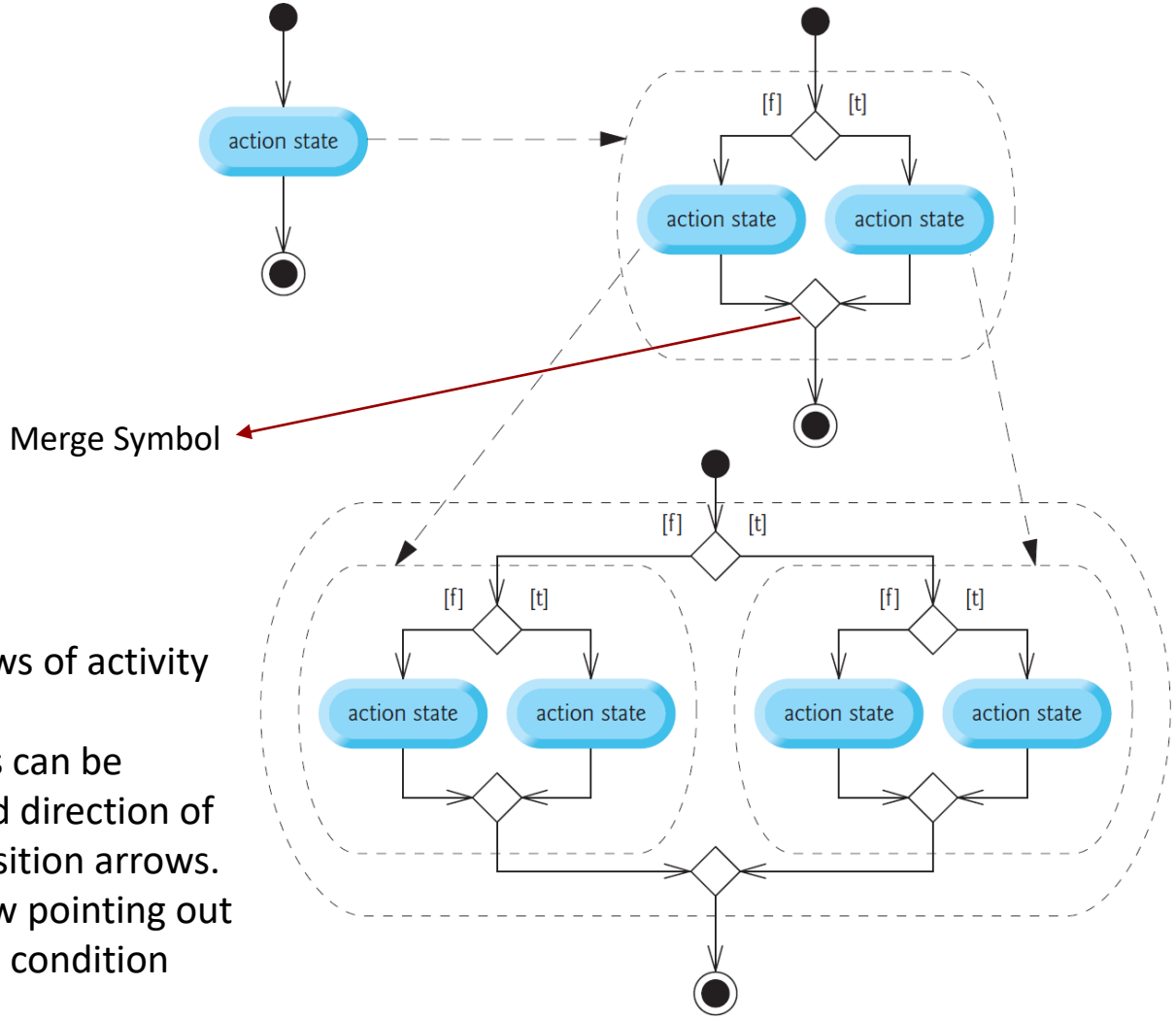
guard conditions (specified by []) can be true or false.

Example:

```
if (grade >= 60) {
   std::cout << "Passed";
}
else {
   std::cout << "Failed";
}
```



UML Activity Diagram for Nested if…else Statements



- The **merge symbol** joins two flows of activity into one.
- The decision and merge symbols can be distinguished by the number and direction of "incoming" and "outgoing" transition arrows. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it.

float & double Data Types

float & double Data Types

C++ provides data types float and double to store floating-point numbers (real numbers) in memory.

```
float x{10.12};
```

```
double x{1000.12345};
```

- double variables can typically store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point, also known as the number's precision) than float variables (~ 6 vs 15 significant digits). ▶
- There are two different ways to declare floating-point numbers: Standard Notation & Scientific Notation.

```
double pi { 3.14159 }; // standard notation
double avogadro { 6.02e23 }; // scientific notation
double electronCharge { 1.6e-19 }; // scientific notation
```

- C++ also supports data type long double for floating-point values with larger magnitude and more precision than double.
- C++ treats all floating-point numbers as double values by default. Thus, most programmers represent floating-point numbers with type double (rather that float).

Presentation of Floating-Point Numbers

- setprecision(n) is a (parameterized) stream manipulator which sets the decimal precision (digits to the right of the decimal point) to n after rounding for printing on the screen.
- fixed and scientific are stream manipulator which write floating-point values in fixed-point or scientific notation.

```
#include <iostream>
#include <iomanip>

int main () {
  double a{3.1415926534};
  double b{2006.0624};
  double c{2.23e-10};
  double d{88};

  std::cout << a << "\n" << b << "\n" << c << "\n" << d << "\n\n";

  std::cout << std::setprecision(3) << std::fixed;
  std::cout << "fixed:\n" << a << "\n" << b << "\n" << c << "\n" << d << "\n\n";

  std::cout << std::setprecision(2) << std::scientific;
  std::cout << "scientific:\n" << a << "\n" << b << "\n" << c << "\n" << d << "\n";
}
```

- These format settings are **sticky settings** and remain in effect until they are changed.
- setprecision belongs to namespace std and is defined in <iomanip> header file.
- fixed and scientific belong to namespace std and are defined in <iostream> header file.

Representational Error of Floating-Point Numbers (double)

- 10 divided by 3 is $3.3333333 \ldots = 3.\overline{3}$. However, the computer allocates only a fixed amount of space to hold such a value. Thus, the stored floating-point value can be only an **approximation**.

- When two floating-point numbers with two digits to the right of the decimal point are added, the output could appear incorrectly.

```
#include <iostream>
#include <iomanip>
int main() {
  double a{14.234};
  double b{18.673};
  std::cout << std::setprecision(2) << std::fixed;
  std::cout << "a = " << a << "\n"
            << "b = " << b << "\n"
            << "a + b = " << a + b << "\n"; // a + b = 32.907
}
```

```
a = 14.23
b = 18.67
a + b = 32.91
```

- Floating-point numbers with many digits of precision to the right of the decimal point are represented incorrectly.

```
#include <iostream>
#include <iomanip>
int main() {
  double a{123.02};
  std::cout << std::setprecision(15) << std::fixed;
  std::cout << "a = " << a << "\n";
}
```

```
a = 123.019999999999996
```

Type Conversion

Implicit Type Conversion: Narrowing Conversions

Implicit Type Conversion (Automatic Type Conversion) is done automatically by the compiler on its own. Implicit Type Conversion is either **Narrowing Conversion** or **Promotion**.

- C++ converts the double value 12.7 to an int, by truncating the floating-point part (.7), a **narrowing conversion** that loses data.
- For fundamental-type variables, **list-initialization** syntax prevents Narrowing Conversions that could result in *data loss*. Thus,

```
int x2{12.7};
int x3 = {12.7};
```

yield a compilation error.

- This does not contain a narrowing conversion, however, due to integer division, the fractional part is truncated.
- Note:

```
int x5{x2 / 10};
int x5{x4 / 10.0};
```

yield a compilation error, due to an attempted narrowing conversion.

```
#include<iostream>
int main() {
  int x1 = 12.7; // prior to C++11, Narrowing Conversion
  double x2{12.7}; // direct list initialization (preferred)
  double x3 = {12.7}; // copy list initialization
  int x4{12};
  int x5{x4 / 10};
  std::cout << "x1: " << x1 << "\n";
  std::cout << "x2: " << x2 << "\n";
  std::cout << "x3: " << x3 << "\n";
  std::cout << "x5: " << x5 << std::endl;
}
```

Implicit Type Conversion: Promotion

Promotion happens based on the following order

bool < char & signed char < unsigned char < short int < unsigned short int < int < unsigned int < long int < unsigned long int < long long int < unsigned long long int < float < double < long double

in the following cases, to <u>avoid lose of data</u>:

1. Assignments:

2. Arithmetic (*, /, %, +, -) and Relational (<, >, <=, >=, ==, !=) Operations:

int values 3 and (x1 + 30) are converted to type double.

```
#include<iostream>
int main() {
  int x1{20};
  double x2{3}; // promotion of integer 3
  x2 = x1 + 30;
  int x3{12};
  double x4{12.7};
  x1 = x3 + x4; // narrowing conversions
  std::cout << "x1: " << x1 << "\n";
  x2 = x3 + x4; // promotion of x3
  std::cout << "x2: " << x2 << "\n";
  double x5{x4 / x3}; // promotion of x3
  std::cout << "x5: " << x5 << std::endl;
}
```

For these operations, the compiler knows how to evaluate <u>only</u> expressions in which the operand data types are <u>identical</u>. In expressions containing values of two or more data types (**mixed-type expressions**), the compiler **promotes** the type of each value to the "highest" type in the expression (actually a temporary version of each value is created and used for the expression, the original values remain unchanged).

Explicit Type Conversion or Type Casting

Explicit Type Conversion or **Type Casting** happens when the user manually convert a value of one data type to a value of another data type.

- C++ supports 5 different types of casts: C-style Cast, static_cast, const_cast, dynamic_cast, and reinterpret_cast. Cast operators are unary operators and available for use with every fundamental type and with class types as well.

- C-style Cast: (data type) expression or data type (expression)

- Static Cast: static_cast<data type>(expression)

(preferred)

Type casting operators converts a **temporary copy** of its operand to the intended data type to be used in the calculations (without changing the data type of its operand).

Best practice: Avoid using C-style cast and use static_cast when you need to convert the data type of a value.

```
#include <iostream>
int main() {
  int x{10};
  int y{4};
  double d1{(double)x / y};
  std::cout << d1 << "\n"; // prints 2.5
  double d2{double(x) / y};
  std::cout << d2 << "\n"; // prints 2.5
  double d3{static_cast<double>(x) / y};
  std::cout << d3 << "\n"; // prints 2.5
  int d4{static_cast<int>(d3)};
  std::cout << d4 << "\n"; // prints 2
}
```

Explicit Promotion

Explicit Narrowing Conversion

while **Iteration Statement**

while **Iteration Statements**

while statements repeat an action (or group of actions) in their bodies while a condition remains true. When the condition is false, the iteration terminates, and the first statement after the body of while will execute. If the condition is initially false, the action (or group of actions) will not execute. Conditions are usually formed by using the relational and equality operators.

Body of while that can be a single statement or a **Block** of several statements in {}.

```
while (condition){
  statement;
  ...
  statement;
}
```

```
int x{3};
while (x <= 100) {
  x = 3 * x;
}
```

- Not providing in the body of a while statement an action that eventually causes the condition to become false results in a logic error called an **infinite loop** (the loop never terminates).

Counter-Controlled and Sentinel-Controlled Iterations

The iterations can be classified into two general categories:

- **Counter-Controlled Iteration** (**Definite Iteration**): The iteration/loop where we know the number of loop executions in advance.
  - A control variable called a **counter** is used to control the number of iterations.

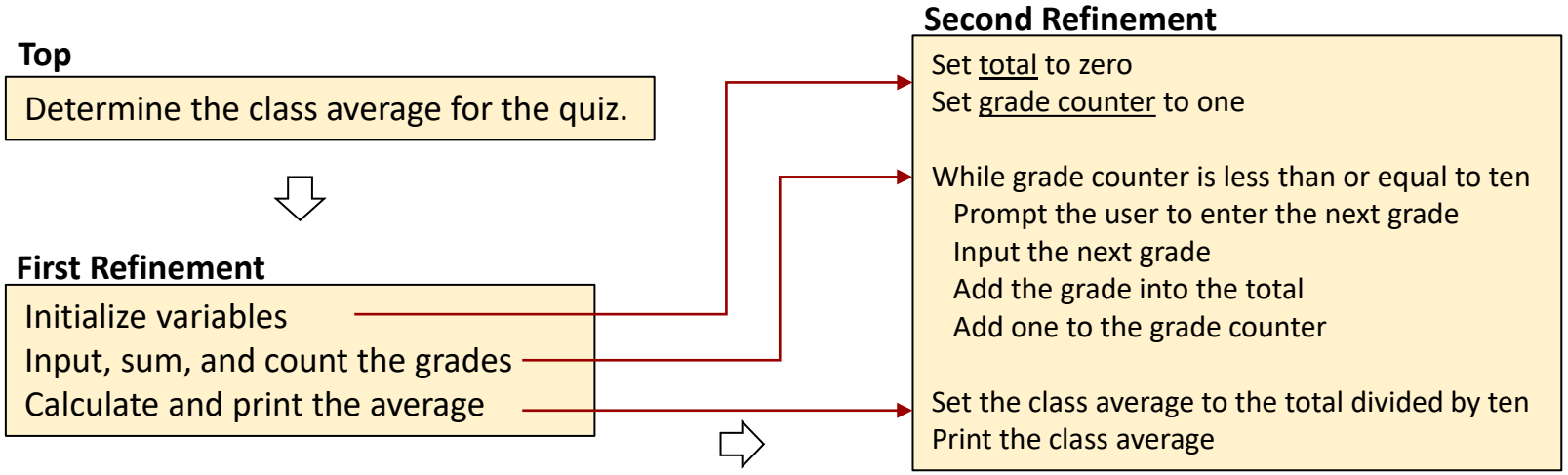
- **Sentinel-Controlled Iteration** (**Indefinite Iteration**): The iteration/loop where we do not know the number of loop executions in advance, and it depends on decisions made inside the loop.
  - A control variable called a **sentinel variable** (signal value, dummy value, or flag value) is used to indicate the end of iterations. The value of this variable changes inside the loop and the loop breaks when this value satisfies the loop condition.

Typically, for statements are used for counter-controlled iteration and while/do…while statements for sentinel-controlled iteration. However, for and while/do…while statements can each be used for either iteration type.

Sample Program: Averaging Student Grades Using Counter-Controlled Iteration

Problem: A class of 10 students took a quiz. The grades (integers in the range 0-100) for this quiz are available to you. Determine the class average on the quiz.

- Before starting to write your code for any problem, it is recommended to first develop an **algorithm** using a tool like **pseudocode** for the solution. This is essential to the development of well-structured programs. Once a correct algorithm has been specified, producing a working code from it is usually straightforward.
- A problem-solving approach using pseudocode is called **Top-Down, Stepwise Refinement**; Top is a single statement that conveys the overall function of the program. It is then divided into a series of smaller tasks. Each refinement is a complete specification of the algorithm, only the level of detail varies.



Sample Program: Averaging Student Grades Using Counter-Controlled Iteration

```
// Solving the class-average problem using counter-controlled iteration.
#include <iostream>

int main() {
  // initialization phase
  int total{0}; // initialize sum of grades entered by the user
  unsigned int gradeCounter{1}; // initialize grade # to be entered next

  // processing phase uses counter-controlled iteration
  while (gradeCounter <= 10) { // loop 10 times
    std::cout << "Enter grade: "; // prompt
    int grade;
    std::cin >> grade; // input next grade
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter by 1
  }

  // termination phase
  int average{total / 10}; // int division yields int result

  // display total and average of grades
  std::cout << "\nTotal of all 10 grades is " << total;
  std::cout << "\nClass average is " << average << std::endl;
}
```

Remarks

```
int total{0};
unsigned int gradeCounter{1};
```

- Variables used to store **totals** are normally initialized to 0.
- In general, **counters** that should store only nonnegative integer values should be declared with unsigned types, and are normally initialized to 0 or 1, depending on how they're used.
- Floating-point values are **approximate**. Thus, controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.

```
int grade;
```

- Variable grade, declared in the body of the while loop, is a **local variable** of the block.
- It is a good practice to define a variable within a loop, since identifiers should be confined to the smallest possible scope. However, it is a bad practice to initialize a variable by a complex function (with long runtime) within a loop if you could just as well do it once before the loop runs. In loops, avoid calculations for which the result never changes; those should be placed before the loop.

Thus, if compute() always returns the same value:

(Good Practice)

```
int value = compute();
while (something) {
   doSomething(value);
}
```

(Bad Practice)

```
while (something) {
   int value = compute();
   doSomething(value);
}
```

(This is more important than scope minimization!)

Sample Program: Averaging Student Grades Using Sentinel-Controlled Iteration and Stacked Control Structures

Problem: Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.

Using Top-Down, Stepwise Refinement Method:

Top

Determine the class average for the quiz.

First Refinement

Initialize variables
Input, sum, and count the grades
Calculate and print the average

Second Refinement

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade or sentinel
Input the first grade (possibly the sentinel)
While the user has not yet entered the sentinel
   Add this grade into the running total
   Add one to the grade counter
   Prompt the user to enter the next grade
   Input the next grade (possibly the sentinel)

If the counter is not equal to zero
   Set the average to the total divided by the counter
   Print the average
else
   Print "No grades were entered"
```

(if…else for preventing of division by zero)

The program tests for a **sentinel value** (here, -1 to prevent confusion with an acceptable input value) after each grade is input and terminates the loop when the user enters the sentinel value.

Sample Program: Averaging Student Grades Using Sentinel-Controlled Iteration and Stacked Control Structures

```
// Solving the class-average problem using sentinel-controlled iteration.
#include <iostream>
#include <iomanip> // parameterized stream manipulators

int main() {
  // initialization phase
  int total{0}; // initialize sum of grades
  unsigned int gradeCounter{0}; // initialize # of grades entered so far

  // processing phase
  // prompt for input and read grade from user
  std::cout << "Enter grade or -1 to quit: ";
  int grade;
  std::cin >> grade;

  // loop until sentinel value read from user
  while (grade != -1)  {
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter

    // prompt for input and read next grade from user
    std::cout << "Enter grade or -1 to quit: ";
    std::cin >> grade;
  }
```

```
  // termination phase
  // if user entered at least one grade...
  if (gradeCounter != 0) {
    // use number with decimal point to calculate average of grades
    double average{static_cast<double>(total) / gradeCounter};

    // display total and average (with two digits of precision)
    std::cout << "\nTotal of the " << gradeCounter
      << " grades entered is " << total;
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "\nClass average is " << average << std::endl;
  }
  else { // no grades were entered, so output appropriate message
    std::cout << "No grades were entered" << std::endl;
  }
}
```

- If the sentinel value is input, the loop terminates, and the program does not add –1 to the total.

- It is a good practice to use prompts to remind the user of the sentinel in a sentinel-controlled loop.

Remarks

This code keep reading values and store into x (using sentinel-controlled iteration) until it encounters a data type other than the data type assigned to x or the end-of-file (EOF) indicator, where the condition evaluates to false.

```
#include <iostream>
int main() {
  int x;
  while (std::cin >> x) {
    std::cout << x << std::endl;
  }
}
```

The end-of-file indicator is system-dependent:

- On UNIX/Linux/Mac OS X systems: <Ctrl> d
- On Windows systems: <Ctrl> z

Whenever x is used inside the loop, you know it has been read successfully.

Sample Program: Analysis of Examination Results Using Nested Control Structures

Problem: By having a list of these 10 students, write a program to analyze how well the students did on an exam:

- Input each test result (i.e., a 1 if the student passed or a 2 if the student failed).
- Count the number of test results of each type.
- Display the number of students who passed and the number who failed.
- If more than eight students passed the exam, print “Congratulations!”.

Top

Analyze exam results and decide whether it is satisfactory

First Refinement

Initialize variables
Input the 10 exam results, and count passes and failures
Print a summary of the exam results and decide whether it is satisfactory

Second Refinement

Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
  Prompt the user to enter the next exam result
  Input the next exam result

  If the student passed
    Add one to passes
  Else
    Add one to failures

  Add one to student counter

Print the number of passes
Print the number of failures

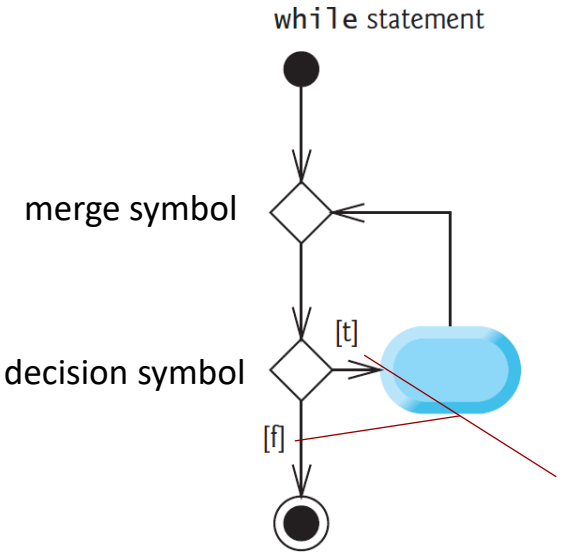
If more than eight students passed
  Print “Congratulations!”

Sample Program: Analysis of Examination Results Using Nested Control Structures

```
// Analysis of examination results using nested control statements.
#include <iostream>

int main() {
  // initializing variables in declarations
  unsigned int passes{0};
  unsigned int failures{0};
  unsigned int studentCounter{1};

  // process 10 students using counter-controlled loop
  while (studentCounter <= 10) {
    // prompt user for input and obtain value from user
    std::cout << "Enter result (1 = pass, 2 = fail): ";
    int result;
    std::cin >> result;

    // if...else is nested in the while statement
    if (result == 1) {
      passes = passes + 1;
    }
    else {
      failures = failures + 1;
    }

    // increment studentCounter so loop eventually terminates
    studentCounter = studentCounter + 1;
  }
```

```
  // termination phase; prepare and display results
  std::cout << "Passed: " << passes << "\nFailed: " << failures << std::endl;

  // determine whether more than 8 students passed
  if (passes > 8) {
    std::cout << "Congratulations!" << std::endl;
  }
}
```

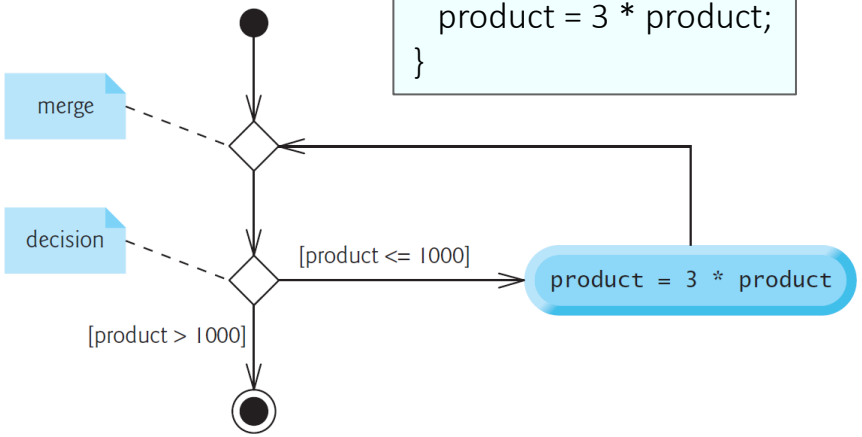
UML Activity Diagram for a while Statement



guard conditions (specified by []) can be true or false.

Example:

```
int product{3};
while (product <= 100) {
  product = 3 * product;
}
```



- The **merge symbol** joins two flows of activity into one.
- The decision and merge symbols can be distinguished by the number and direction of "incoming" and "outgoing" transition arrows. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it.

Compound Assignment, Increment, and Decrement Operators

Compound Assignment Operators

The **Compound Assignment Operators** can be used to simplify assignment expressions.

variable = **variable operator** expression; ⟺ **variable operator**= expression;

(+, -, *, /, %)

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

```
#include<iostream>
int main() {
  int a{2};
  double b{3};
  double c{4};
  c *= (a + b); // or c *= a + b
  std::cout << "c = " << c << std::endl;
}
```

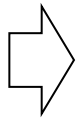
Increment and Decrement Operators

C++ provides two unary operators ++, -- for adding 1 to or subtracting 1 from the value of a **numeric variable** to simplify program statements.

Pre-incrementing Using Prefix Increment Operator	$++v$	Increment v by 1, then use the **new value** of v in the expression in which v resides.
Pre-decrementing Using Prefix Decrement Operator	$--v$	Decrement v by 1, then use the **new value** of v in the expression in which v resides.
Post-incrementing Using Postfix Increment Operator	$v++$	Use the **current value** of v in the expression in which v resides, then increment v by 1.
Post-decrementing Using Postfix Decrement Operator	$v--$	Use the **current value** of v in the expression in which v resides, then decrement v by 1.

See Examples 1, 2 →

- When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect.



same effect

```
v = v + 1;
v += 1;
++v;
v++;
```

- Writing ++(x + 1) is a syntax error, because (x + 1) is not a variable.

Prefix Increment and Postfix Increment Operators

Example 1:

```
#include <iostream>
int main() {
  unsigned int c{5}; // initializes c with the value 5
  std::cout << "c before postincrement: " << c << "\n"; // prints 5
  std::cout << "postincrementing c: " << c++ << "\n"; // prints 5
  std::cout << "c after postincrement: " << c << "\n"; // prints 6
  std::cout << "\n"; // skip a line
  c = 5; // assigns 5 to c
  std::cout << "c before preincrement: " << c << "\n"; // prints 5
  std::cout << "preincrementing c: " << ++c << "\n"; // prints 6
  std::cout << "c after preincrement: " << c << std::endl; // prints 6
}
```

Example 2:

```
#include <iostream>
int main() {
  int x1{10};
  int y1;
  y1 = 2 * (++x1) + 5;
  std::cout << "x1 = " << x1 << ", y1 = " << y1 << "\n"; // Prints x1 = 11, y1 = 27
  int x2{10};
  int y2;
  y2 = 2 * (x2++) + 5;
  std::cout << "x2 = " << x2 << ", y2 = " << y2 << "\n"; // Prints x2 = 11, y2 = 25
}
```

Operator Precedence and Associativity

decreasing order of precedence

Operators	Associativity	Type
`::` `()`	left to right *[For nested parentheses: from innermost pair]*	primary
`++` `--` `static_cast<type>()`	left to right	postfix
`++` `--` `+` `-`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment

- If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you would do in an algebraic expression.