CS1010E Lecture 4
Control Structures and Data Files (Part 1)

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
www.comp.nus.edu.sg/~joxan
cs1010e@comp.nus.edu.sg

Semester II, 2016/2017

Lecture Outline

- Algorithm Development.
- Conditional Expressions.
- Selection Statements.

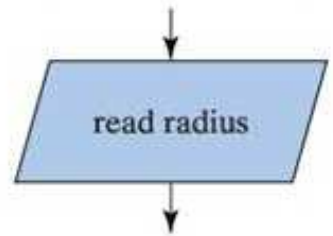
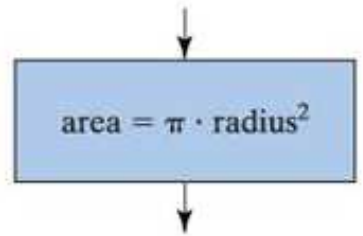
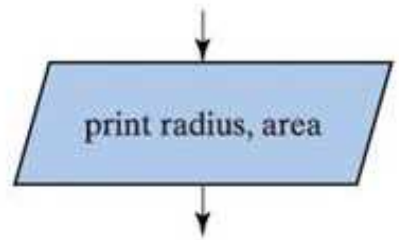
Algorithm Development

- So far, the C programs that we developed are very simple.
- The steps were sequential and involved:
  - Reading information from the keyboard.
  - Computing new information using assignments.
  - Printing the new information.
- In solving engineering problems, most of the solutions require more complicated steps.
- Need to expand the algorithm development part of our problem-solving process.

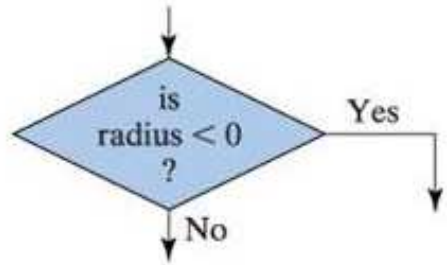
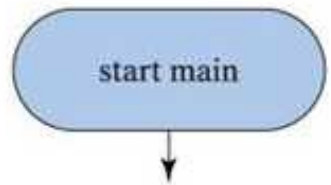
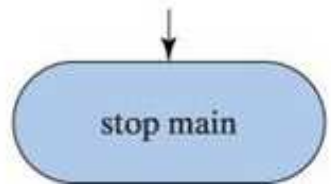
Pseudocode and Flowchart

- **Pseudocode** uses English-like statements to describe the steps in an algorithm.
- A **flowchart** uses a diagram to describe the steps in an algorithm.
- The fundamental steps in most algorithms together with pseudocode and a flowchart are shown in the next two slides.

Pseudocode and Flowchart

Basic Operation	Pseudocode Notation	Flowchart Symbol
Input	read radius	
Computation	set area to $\pi \cdot \text{radius}^2$	
Output	print radius, area	

Pseudocode and Flowchart

Comparisons	if radius < 0 then …	
Beginning of algorithm	main:	
End of algorithm		

Structured Programming

A **structured program** is written using simple control structures to organize the solution to a problem.
These control structures are:

- A sequence structure contains steps that are performed one after another.
- A selection structure contains one set of steps that is performed if a condition is true, and another set of steps that is performed if a condition is false.
- A repetition structure contains a set of steps that is repeated as long as a condition is true.

Note: *Non-structured* programs are not covered in this course.

Sequence

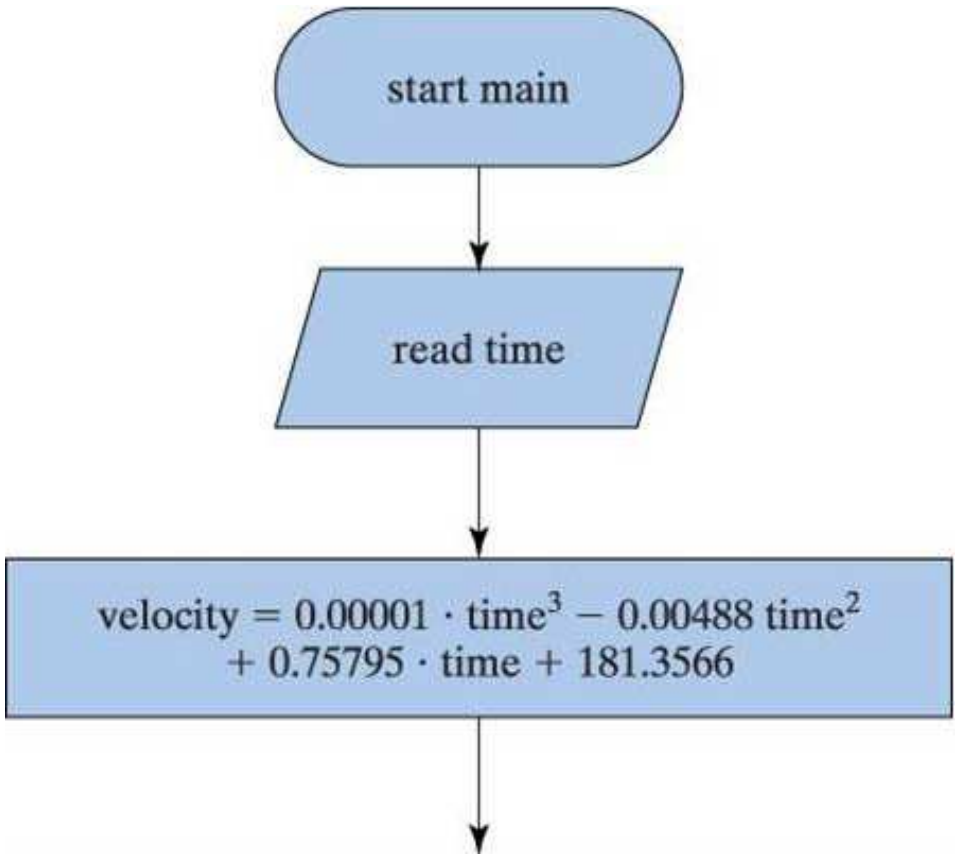
- A sequence contains steps that are performed one after another.
- All the programs we have developed so far have a sequence structure.
- For example, the pseudocode for the program that performed the linear interpolation is shown in the next slide, and the flowchart for the program that computed the velocity and acceleration of the aircraft with the unducted engine is shown in the two slides that follow.

Sequence

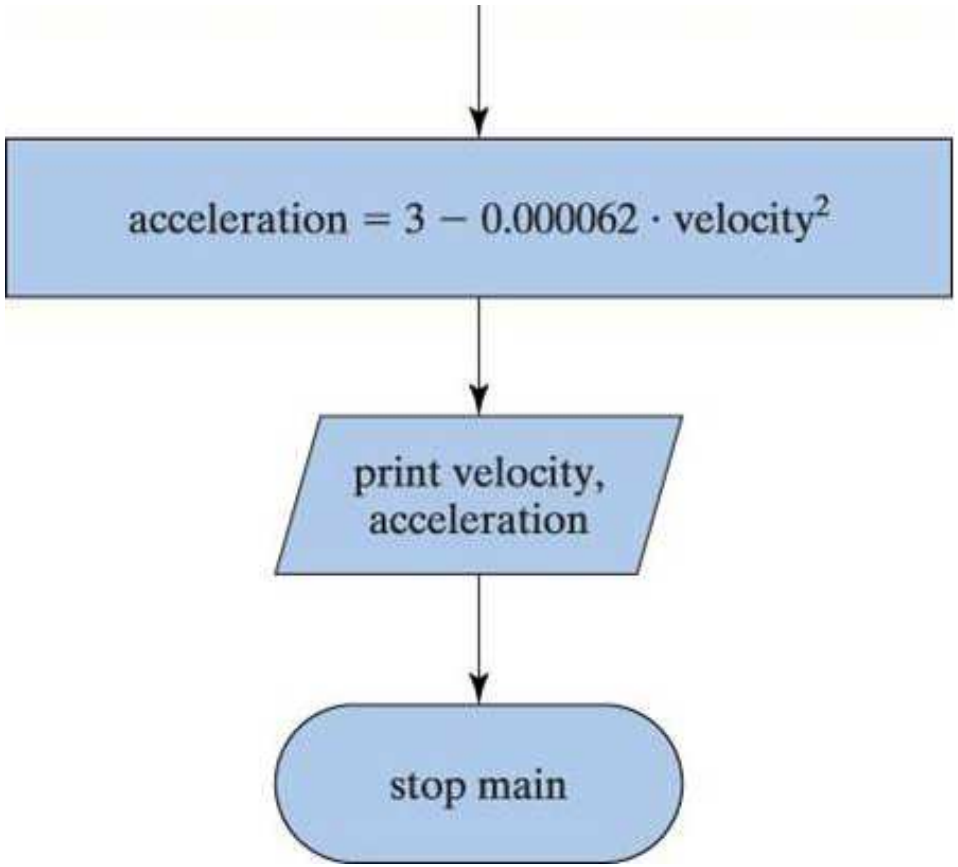
Refinement in Pseudocode

main: *read a, f_a*
read c, f_c
read b
set f_b to $f_a + \frac{b-a}{c-a} \cdot (f_c - f_a)$
print f_b

Sequence


start main
read time
velocity = 0.00001 · time³ − 0.00488 time²
+ 0.75795 · time + 181.3566

Sequence


acceleration = 3 − 0.000062 · velocity²
print velocity, acceleration
stop main

Selection

- A selection structure contains a **condition** that can be evaluated as either true or false.
- If the condition is true, then one set of statements is executed; if the condition is false, then another set of statements is executed.
- For example, suppose that we have computed values for the numerator and denominator of a fraction.
- Before we compute the division, we want to make sure that the denominator is not close to zero.

Selection

- Therefore, the condition that we want to test is “denominator close to zero”.
- If the condition is true, then print a message indicating that we cannot compute the value.
- If the condition is false, which means that the denominator is not close to zero, then we compute and print the value of the fraction.
- In defining this condition, we need to define “close to zero”.
- We will assume that “close to zero” means that the absolute value is less than 0.0001.

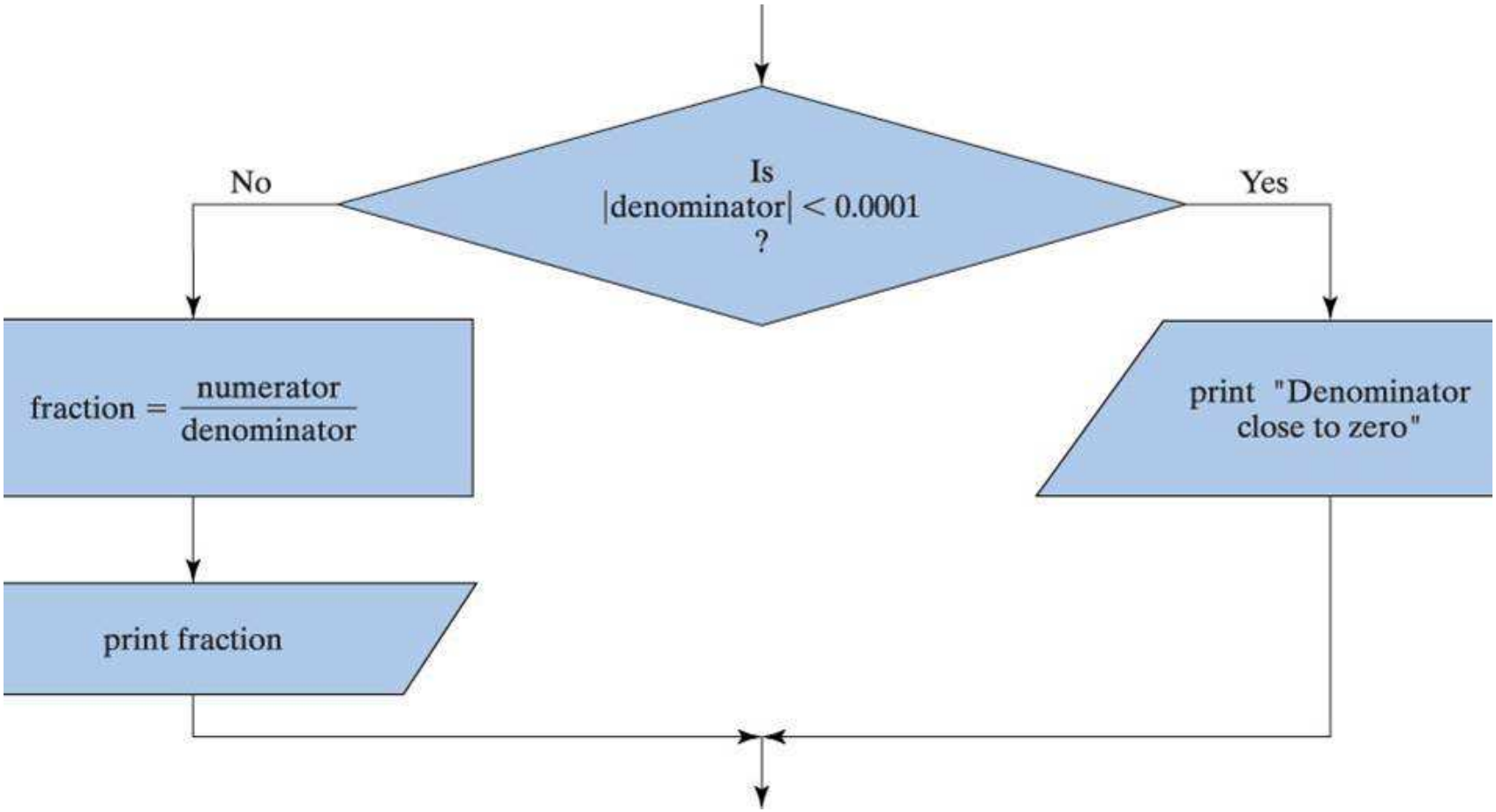
Selection

- A pseudocode description is as follows:
  - *if* $|denominator| < 0.0001$
    print "Denominator close to zero"
    else
    set fraction to numerator/denominator
    print fraction

- A flowchart description of this structure is shown in the next slide. Note that the structure also contains a sequence structure that is executed when the condition is false.

- More variations of the selection structure will be covered later in this lecture

Selection


Is
|denominator| < 0.0001
?
No
Yes
fraction = numerator / denominator
print fraction
print "Denominator close to zero"

Repetition

- Although this structure would require 11 statements in this case, it could require hundreds of statements if we wanted to compute the velocity values over a long period.

- If we use a repetition structure, we can develop a solution in which we initialize the time to 0.

- As long as the time value is less than or equal to 10, we compute and print a velocity value and increment the time value by 1.

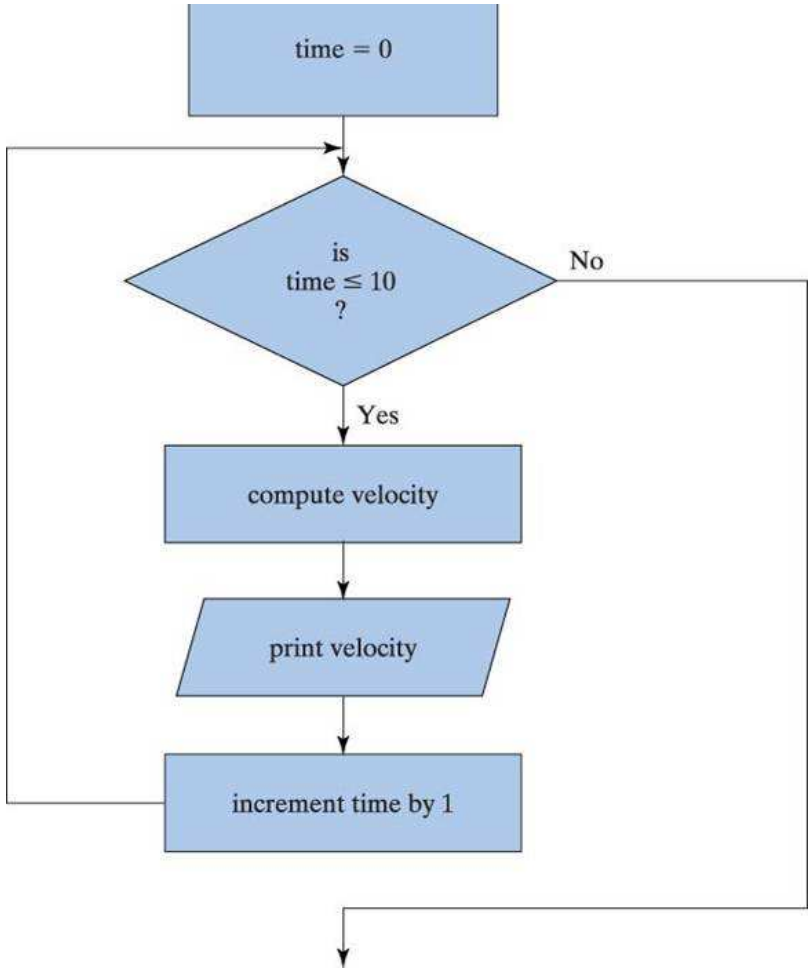
Repetition

- When the time value is greater than 10, we exit the structure.

- The pseudocode is as follows:
  - *set time to* 0
    while time $\leq$ 10
        compute velocity
        print velocity
        increment time by 1

- The flowchart is shown in the next slide.

Repetition


time = 0
is
time ≤ 10
?
No
Yes
compute velocity
print velocity
increment time by 1

Find maximum and average of a list of numbers (1/3)

- Version 1

Declare variables *sum*, *count* and *max*.

First, you initialise *sum*, *count* and *max* to zero.

Then, you enter the input numbers, one by one.

For each number that you have entered, assign it to *num* and add it to the *sum*.

Increase *count* by 1.

At the same time, you compare *num* with *max*. If *num* is larger than *max*, let *max* be *num* instead.

After all the numbers have been entered, you divide *sum* by the numbers of items entered, and let *ave* be this result.

Print *max* and *ave*.

End of algorithm.

Find maximum and average of a list of numbers (2/3)

- Version 2

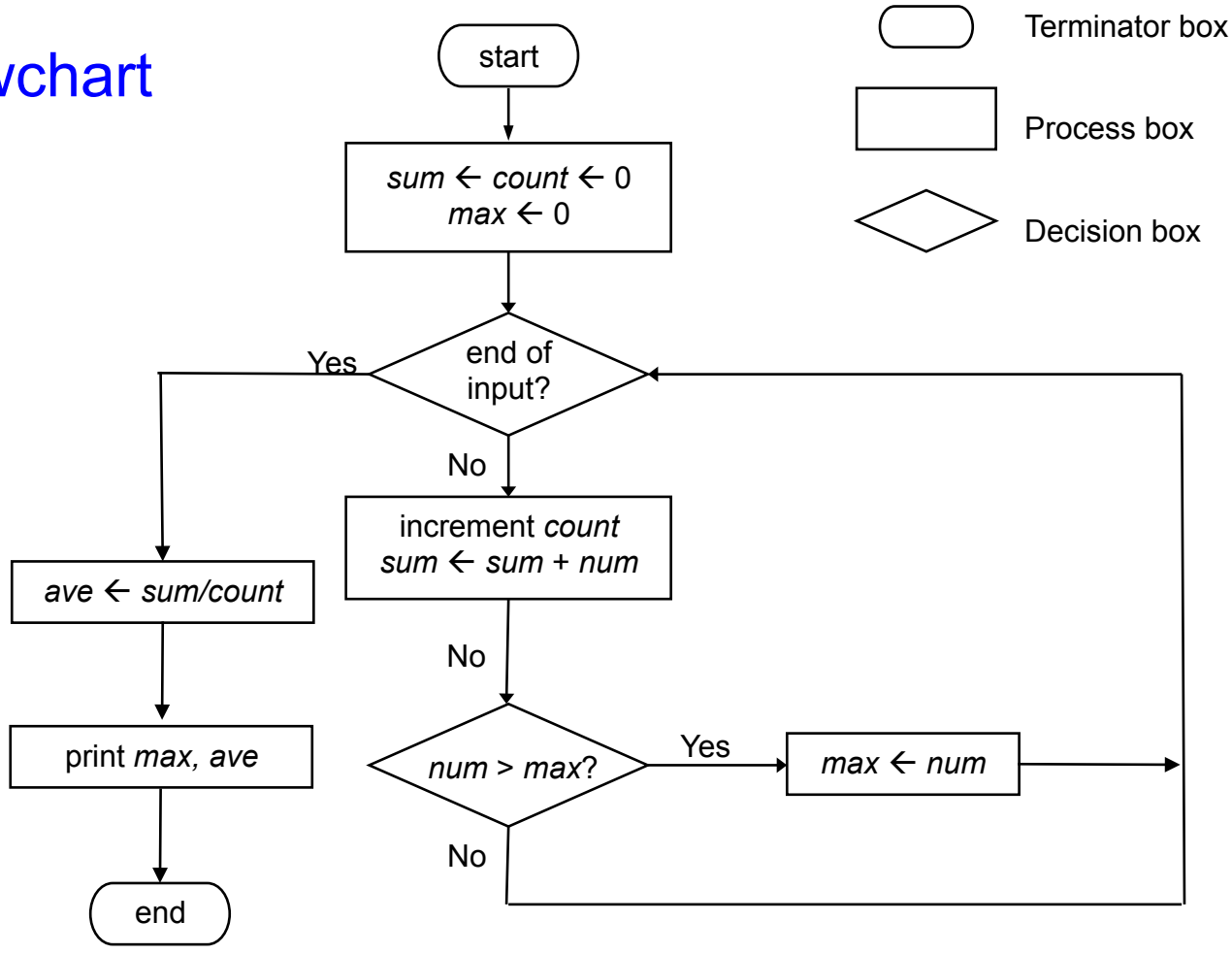
```
sum ← count ← 0      // sum = sum of numbers
                     // count = how many numbers are entered?
max ← 0              // max to hold the largest value eventually
for each num entered,
        count ← count + 1
        sum ← sum + num
        if num > max
                then max ← num

ave ← sum / count

print max, ave
```

Find maximum and average of a list of numbers (3/3)

- Flowchart



Conditional Expressions

- Because both selection and repetition structures use conditions, we must discuss conditions first.
- A condition is an expression that can be evaluated to be true or false.
- It is composed of expressions combined with relational operators.
- It can also include logical operators.
- We need to take care of the evaluation order when relational and logical operators are combined into a single condition.

Relational Operators

The **relational operators** that can be used to compare two expressions in C are shown below:

Relational Operator	Interpretation
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Relational Operators

- Blanks can be used on either side of a relational operator, but blanks cannot be used to separate a two-character operator, such as ==.

- Example of conditions are the following:
  - `a < b`
  - `x+y >= 10.5`
  - `fabs(denominator) < 0.0001`

- Note that we use spaces around the relational operator in a logical expression, but not around the arithmetic operators in the conditions.

Relational Operators

- Given the values of the identifiers in these conditions, we can evaluate each one to be true or false.
- For example, if `a` is equal to 5 and `b` is equal to 8.4, then `a < b` is a true condition.
- If `x` is equal to 2.3 and `y` is equal to 4.1, then `x+y >= 10.5` is a false condition.
- If `denominator` is equal to -0.0025, then `fabs(denominator) < 0.0001` is a false condition.

Relational Operators

- A true condition is assigned a value of 1 and a false condition is assigned a value of 0.
- The following statement is valid:
  - `d = b>c;`
- If `b>c`, then the value of `d` is 1; otherwise, the value of `d` is 0.

Relational Operators

- A single value can be used in place of a condition:
  - ```
    if (a)
        count++;
    ```
- If the condition value is zero, then the condition is assumed to be false.
- If the condition value is nonzero, then the condition is assumed to be true.
- The value of `count` will be incremented if `a` is nonzero.
    ```

Logical Operators

- Logical operators can also be used within conditions.
- Logical operators compare conditions, not expressions.
- C supports three **logical operators**: and, or, and not.

Logical Operators

These logical operators are represented by the following symbols:

Logical Operator	Symbol
and	`&&`
or	`||`
not	!

Logical Operators

- Consider the following condition:
  - `a<b && b<c`
- The relational operators have higher precedence than logical operators.
- This is read as "`a` is less than `b`, and `b` is less than `c`."
- We insert spaces around the logical operators, but not around the relational operators for readability.
- Given values for `a`, `b`, and `c`, we can evaluate this condition as true or false.

Logical Operators

- If `a` is equal to 1, `b` is equal to 5, and `c` is equal to 8, then the condition is true.

- If `a` is equal to -2, `b` is equal to 9, and `c` is equal to 2, then the condition is false.

Logical Operators

- If `A` and `B` are conditions, then the logical operators can be used to generate new conditions `A && B`, `A || B`, `!A`, and `!B`.
- The condition `A && B` is true only if both `A` and `B` are true.
- The condition `A || B` is true if either or both of `A` and `B` are true.
- The condition `!A` is true only if `A` is false.
- The condition `!B` is true only if `B` is false.
- These definitions are summarized in the next slide.

Logical Operators

A	B	A && B	A || B
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Logical Operators

- When expressions with logical operators are executed, C will only evaluate as much of the expression as necessary to evaluate it.

- For example, if `A` is false, then the expression `A && B` is also false, and there is no need to evaluate `B`.

- Similarly, if `A` is true, then the expression `A || B` is also true, and there is no need to evaluate `B`.

Precedence and Associativity

- A condition can contain several logical operators:
  - `!(b==c || b==5.5)`
- The hierarchy, from highest to lowest, is `!`, `&&`, and `||`, but parentheses can be used to change the hierarchy.
- The expressions `b==c` and `b==5.5` are evaluated first.
- Suppose `b` is equal to 3 and `c` is equal to 5.

Precedence and Associativity

- Neither expression is true, so the expression `b==c || b==5.5` is false.
- We then apply the `!` operator to the false condition, which gives a true condition.
- Blanks cannot be used to separate the characters in either the `&&` or `||`.
- A common error is to use = instead of == in a logical expression.
- A condition can contain both arithmetic operators and relational operators, as well as logical operators.

Operator Precedence Table

Precedence	Operation
1	( )
2	++ -- + - ! (type)
3	* / %
4	+ -
5	< <= > >=
6	== !=
7	&&
8	||
9	= += -= *= /= %=

Selection Statements

- The `if` statement allows us to test conditions and then perform statements base on whether the conditions are true or false.
- C contains two forms of `if` statements—the simple `if` statement and the `if/else` statement.
- C also contains a `switch` statement that allows us to test multiple conditions and then execute groups of statements based on whether the conditions are true or false.

Simple if Statement

- The simplest form of an `if` statement has the following general form:
  - `if` (condition)
    statement 1;

- If the condition is true, we execute statement 1; if the condition is false, we skip statement 1.

- The statement within the `if` statement is indented so that it is easier to visualize the structure of the program from the statements.

Simple if Statement

- If we wish to execute several statements (or a sequence structure) when the condition is true, we use a **compound statement**, or block, which is composed of a set of statements enclosed in braces.

- The location of the braces is a matter of style; two common styles are shown in the next slide.

Simple if Statement

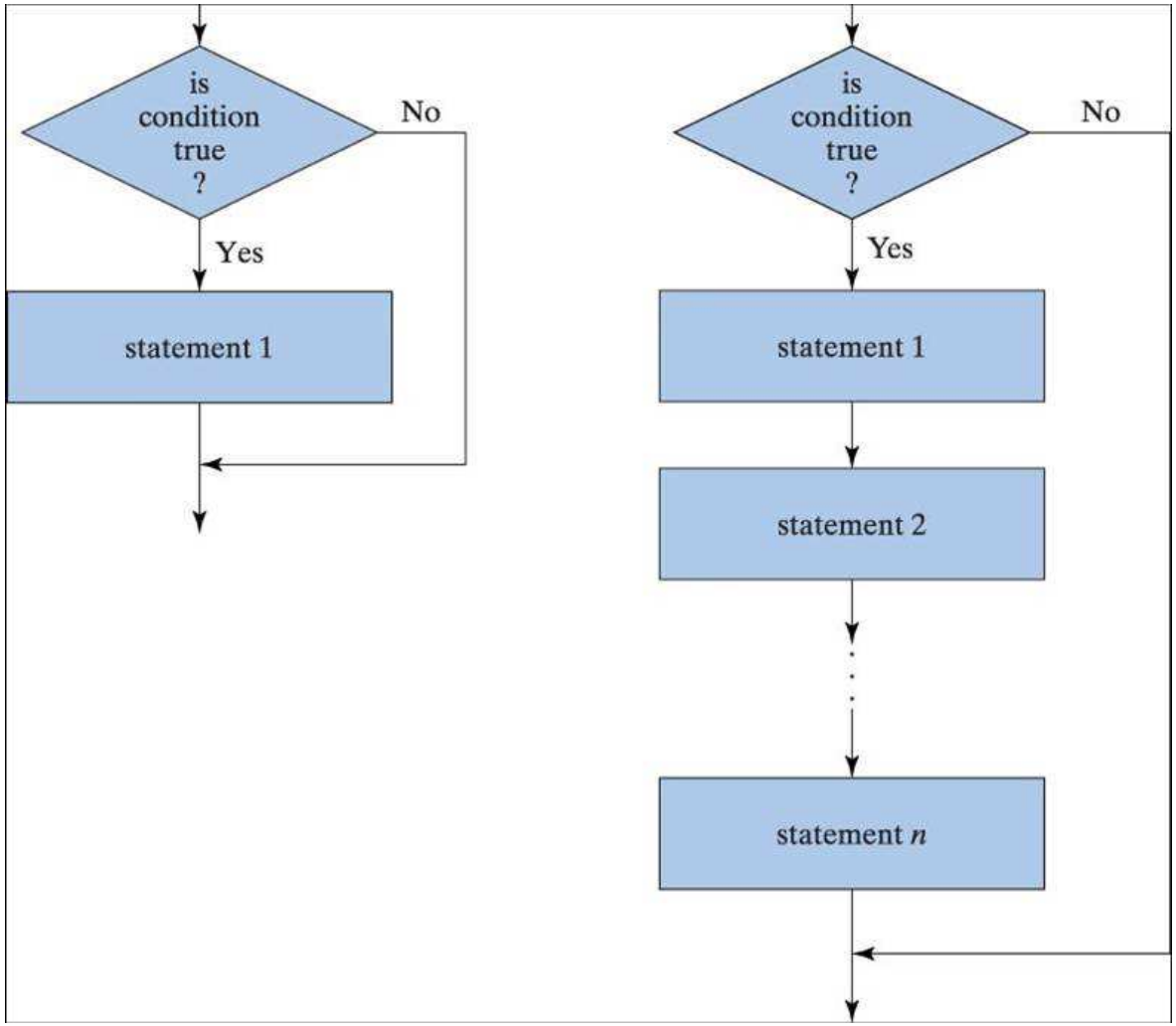
Style 1

```
if (condition)
{
      statement 1;
      statement 2;
      ...
      statement n;
}
```

Style 2

```
if (condition) {
      statement 1;
      statement 2;
      ...
      statement n;
}
```

Flowcharts


is condition true ?
No
Yes
statement 1
is condition true ?
No
Yes
statement 1
statement 2
statement n

Simple if Statement

- A specific example of an `if` statement is:

```
if (a < 50) {
    ++count;
    sum += a;
}
```

- If `a` is less than 50, then `count` is incremented by 1 and `a` is added to `sum`.

- Otherwise, these two statements are skipped.

Simple if Statement

- The `if` statements can also be nested:

```
if (a < 50) {
    ++count;
    sum += a;
    if (b > a)
        b = 0;
}
```

- If `a` is less than 50, then `count` is incremented by 1 and `a` is added to `sum`.

- In addition, if `b` is greater than `a`, then we also set `b` to 0.

if/else Statement

- An `if/else` statement allows us to execute one set of statements if a condition is true and a different set if the condition is false.

- The simplest form of an `if/else` statement is the following:
  - `if` (condition)
    statement 1;
    `else`
    statement 2;

- Statements 1 and 2 can also be replaced by compound statements.

if/else Statement

- Statements 1 and 2 can also be an **empty statement**, which is just a semicolon.

- The following two statements are equivalent:

  - ```
    if (a < b)
        ;
    else
        count++;
    ```
  - ```
    if (a >= b)
        count++;
    ```

if/else Statement

- Consider this `if/else` statement:

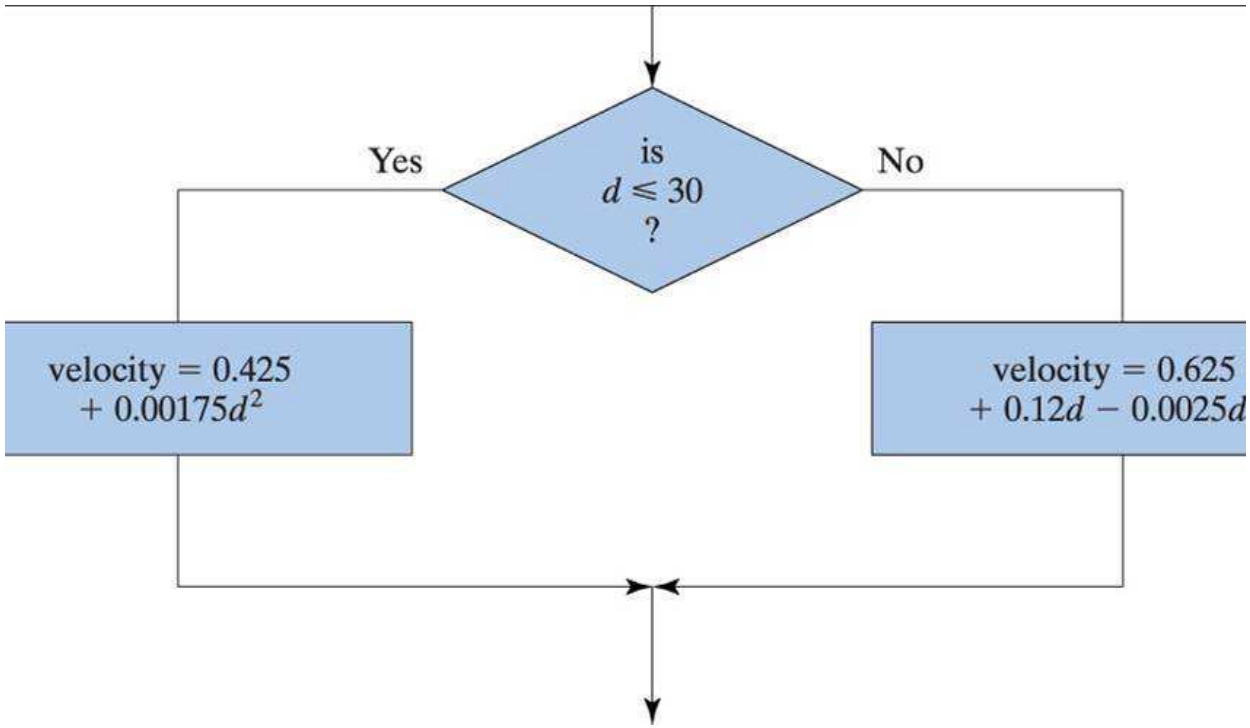
```
if (d <= 30)
   velocity = 0.425 + 0.00175*d*d;
else
   velocity = 0.625 + 0.12*d - 0.0025*d*d;
```

- `velocity` is computed with the first assignment statement if the distance `d` is less than or equal to 30.

- Otherwise, `velocity` is computed with the second assignment statement.

- A flowchart for this `if/else` statement is shown in the next slide.

Flowchart

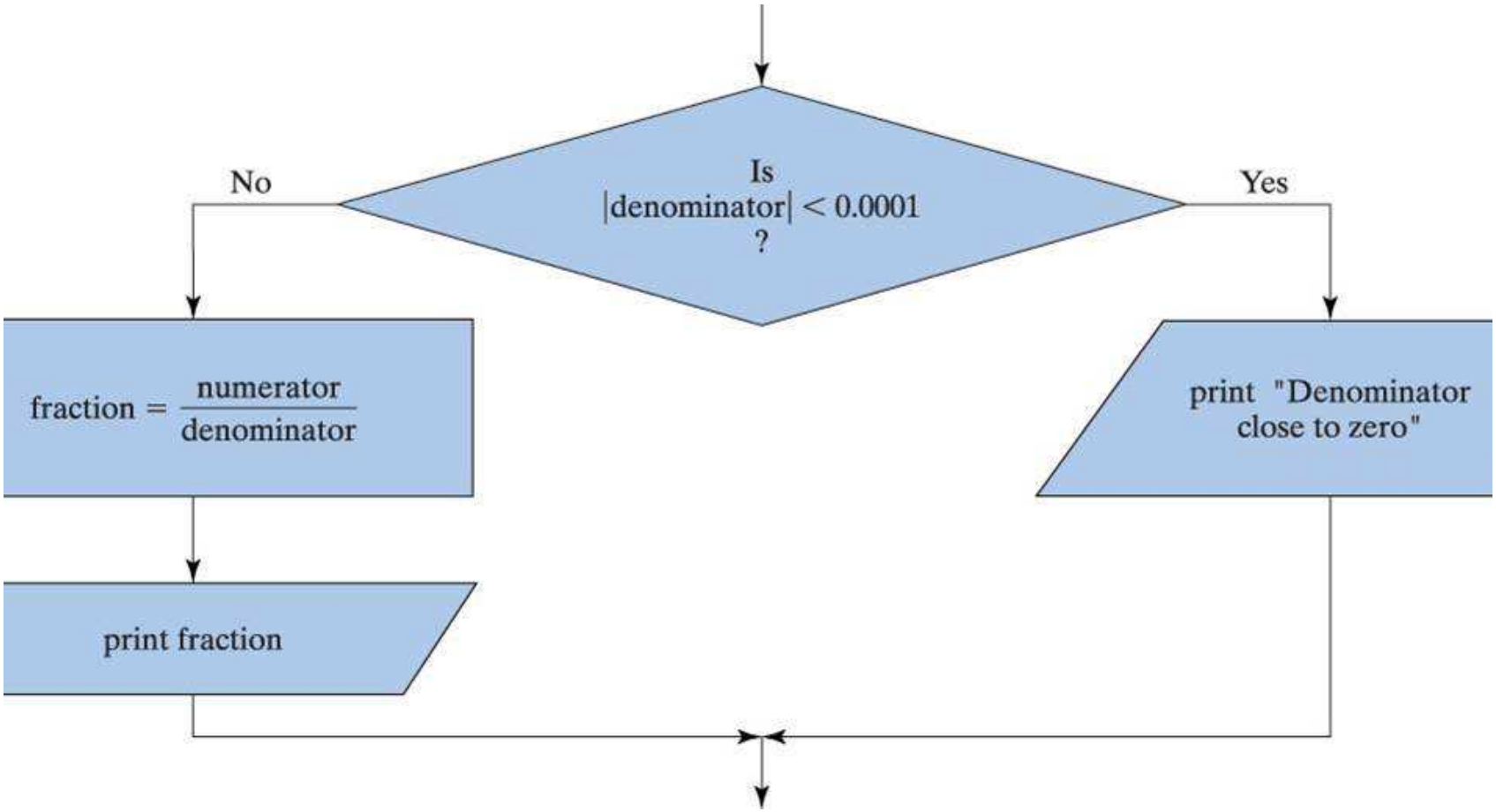

Yes
is
$d \leq 30$
?
No
velocity = 0.425
+ 0.00175d^2
velocity = 0.625
+ 0.12d − 0.0025d

if/else Statement

- Another example of the `if/else` statement:
  - ```
    if (fabs(denominator) < 0.0001)
       printf("Denominator close to zero \n");
    else
    {
       x = numerator/denominator;
       printf("x = %f \n",x);
    }
    ```

- We examine the absolute value of the variable `denominator`.
  If this is close to zero, we print an error message;
  otherwise, we compute and print the value of `x`.
    ```

Flowchart


Is
|denominator| < 0.0001
?
No
Yes
fraction = numerator / denominator
print fraction
print "Denominator close to zero"

Nested if/else Statement

- Consider the following set of nested `if/else` statements:

```
if (x > y)
    if (y < z)
       k++;
    else
       m++;
else
    j++;
```

- The value of `k` is incremented when `x > y` and `y < z`.

Nested if/else Statement

- The value of `m` is incremented when `x > y` and `y >= z`.
- The value of `j` is incremented when `x <= y`.
- With careful indenting, this statement is easy to follow.

Nested if/else Statement

- Suppose we eliminate the `else` portion of the inner `if` statement:

```
if (x > y)
    if (y < z)
        k++;
else
    j++;
```

- It might appear that `j` is incremented when `x <= y`, but that is not correct.

- The C compiler will associate an `else` statement with the closest `if` statement within a block.

Nested if/else Statement

- Therefore, no matter what indenting is used, the previous statement is executed as if it were the following:

```
if (x > y)
    if (y < z)
        k++;
    else
        j++;
```

- Thus, `j` is incremented when `x > y` and `y >= z`.

Nested if/else Statement

- If we intended for `j` to be incremented when `x <= y`, then we would need to use braces to define the inner statement as a block:

```
if (x > y)
{
   if (y < z)
      k++;
}
else
   j++;
```

- When confusion is possible, *use braces*.

Conditional Operator

- To illustrate, the following two statements are equivalent:

```
if (a<b)
   c = b - a;
else
   c = b + a;
```

```
c = a<b ? b - a : b + a;
```

- The conditional operator (specified as `?:`) is evaluated before assignment operators.

- If there is more than one conditional operator in an expression, they are associated from right to left.

Comparing Floating-Point Values

- A caution is necessary when comparing floating-point values.
- Floating-point values can sometimes be slightly different than we expect them to be because of the conversions between binary and decimal values.
- For example, if we wanted to know if `y` was close to the value 10.5, we should use a condition such as `fabs(y-10.5) <= 0.0001` instead of `y == 10.5`.
- In general, do not use the equality operator with floating-point values.

switch Statement

- The `switch` statement is used for multiple-selection decision making.
- It is often used to replace nested `if/else` statements.
- We present an example that uses nested `if/else` statements and then an equivalent solution that uses the `switch` statement.

switch Statement

- Suppose that we have a temperature reading from a sensor inside a large piece of machinery. We want to print a message on the control screen to inform the operator of the temperature status.
- If the status code is 10, the temperature is too hot and the equipment should be turned off.
- If the status code is 11, the operator should check the temperature every 5 minutes.
- If the status code is 13, the operator should turn on the circulating fan.
- For all other status codes, the equipment is operating in a normal mode.
- The correct message could be printed with the set of nested `if/else` statements shown on the next slide, followed by an equivalent `switch` statement on the following slide.

Nested If Statements

```
if (code == 10)
   printf("Too hot - turn equipment off \n");
else
{
   if (code == 11)
      printf("Caution - recheck in 5 minutes \n");
   else
   {
      if (code == 13)
         printf("Turn on circulating fan \n");
      else
         printf("Normal mode of operation \n");
   }
}
```

switch Statement

```
switch (code)
{
   case 10:
      printf("Too hot - turn equipment off \n");
      break;
   case 11:
      printf("Caution - recheck in 5 minutes \n");
      break;
   case 13:
      printf("Turn on circulating fan \n");
      break;
   default:
      printf("Normal mode of operation \n");
      break;
}
Statement 1;
```

switch Statement

- The `break` statement causes the execution of the program to continue with the statement following the `switch` statement (Statement 1), thus skipping the rest of the statements in the braces.
- Nested `if/else` statements do not always easily translate to a `switch` statement.
- However, when the conversion works, the `switch` statement is usually easier to read.
- It is also easier to determine the punctuation needed for the `switch` statement.

switch Statement

- The `switch` statement selects the statements to perform based on a **controlling expression**, which must be an expression of type integer or character.

- In the general form that follows, **case labels** (`label_1`, `label_2`, `...`) determine which statements are executed. This is structure is sometimes called a **case structure**.

- The statements executed are the ones that correspond to the case for which the label is equal to the controlling expression.

switch Statement

- The case labels must be unique constants.
- An error occurs if two or more of the case labels have the same value.
- The **default label** provides a statement to execute if no other statement is executed; the default label is optional.
- The next slide shows the general form of the `switch` statement.

switch Statement

```
switch (controlling expression)
    {
        case label_1:
            statements;
        case label_2:
            statements;
        ...
        default:
            statements;
    }
```

switch Statement

- The statements in the `switch` structure usually contain the `break` statement.
- When the `break` statement is executed, the execution of the program breaks out of the `switch` structure, and continues executing with the statement following the `switch` structure.
- Without the `break` statement, the program will execute all statements that follow the ones selected with the case label.

switch Statement

- It is valid to use several case labels with the same statement:
  - ```
    switch (op_code)
    {
       case 'N': case 'R':
          printf("Normal operating range \n");
          break;
       case 'M':
          printf("Maintenance needed \n");
          break;
       default:
          printf("Error in code value \n");
          break;
    }
    ```
    ```

switch Statement

- When more than one case label is used for the same statement, the evaluation is performed as if the logical operator `||` joined the cases.

- For example, the first statement is executed if `op_code` is equal to `'N'` or if `op_code` is equal to `'R'`.

Next Lecture

Control Structures and Data Files

(Part 2)