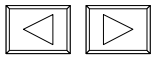
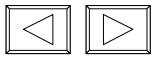
Capitolo 2 - Control Structures

Outline



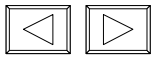
Capitolo 2 - Control Structures

Outline



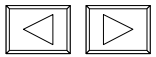
2.1 Introduction

- Before writing a program:
  - Have a thorough understanding of problem
  - Carefully plan your approach for solving it
- While writing a program:
  - Know what “building blocks” are available
  - Use good programming principles



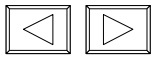
2.2 Algorithms

- All computing problems
  - can be solved by executing a series of actions in a specific order
- Algorithm
  - A procedure determining the
    - Actions to be executed
    - Order in which these actions are to be executed
- Program control
  - Specifies the order in which statements are to executed



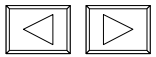
2.3 Pseudocode

- Pseudocode
  - Artificial, informal language used to develop algorithms
  - Similar to everyday English
  - Not actually executed on computers
  - Allows us to “think out” a program before writing the code for it
  - Easy to convert into a corresponding C++ program
  - Consists only of executable statements



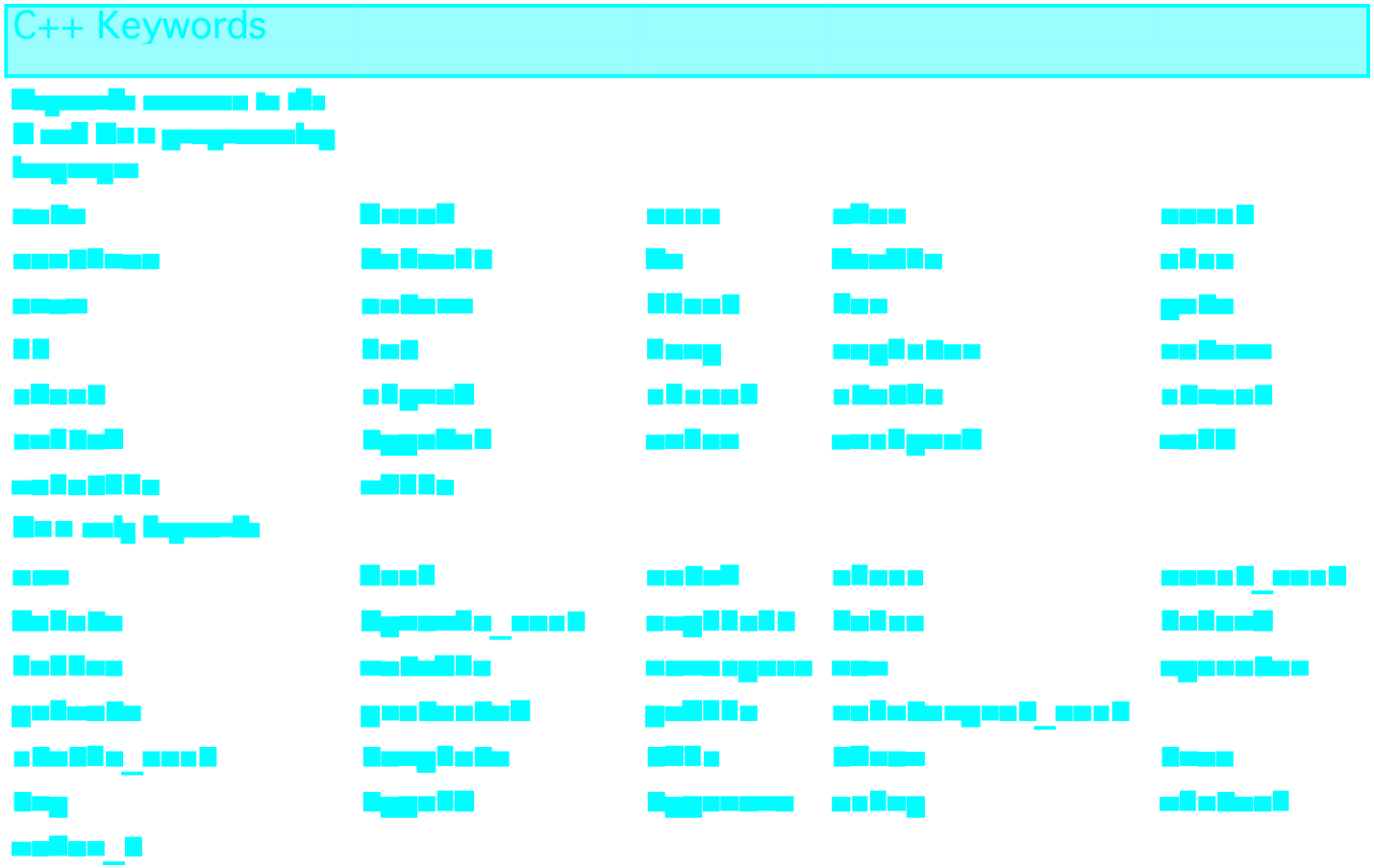
2.4 Control Structures

- Sequential execution
  - Statements executed one after the other in the order written
- Transfer of control
  - When the next statement executed is not the next one in sequence
- Bohm and Jacopini: all programs written in terms of 3 control structures
  - Sequence structure
    - Built into C++. Programs executed sequentially by default.
  - Selection structures
    - C++ has three types - **`if`**, **`if/else`**, and **`switch`**
  - Repetition structures
    - C++ has three types - **`while`**, **`do/while`**, and **`for`**



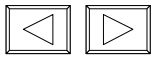
2.4 Control Structures

- C++ keywords
  - Cannot be used as identifiers or variable names.



2.4 Control Structures

- Flowchart
  - Graphical representation of an algorithm
  - Drawn using certain special-purpose symbols connected by arrows called flowlines.
  - Rectangle symbol (action symbol)
    - Indicates any type of action.
  - Oval symbol
    - indicates beginning or end of a program, or a section of code (circles).
- single-entry/single-exit control structures
  - Connect exit point of one control structure to entry point of the next (control-structure stacking).
  - Makes programs easy to build.

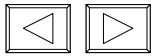


2.5 The if Selection Structure

- Selection structure
  - used to choose among alternative courses of action
  - Pseudocode example:

    If student's grade is greater than or equal to 60

    Print "Passed"
  - If the condition is **`true`**
    - print statement executed and program goes on to next statement
  - If the condition is **`false`**
    - print statement is ignored and the program goes onto the next statement
  - Indenting makes programs easier to read
    - C++ ignores whitespace characters

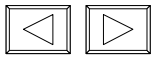


2.5 The if Selection Structure

- Translation of pseudocode statement into C++:

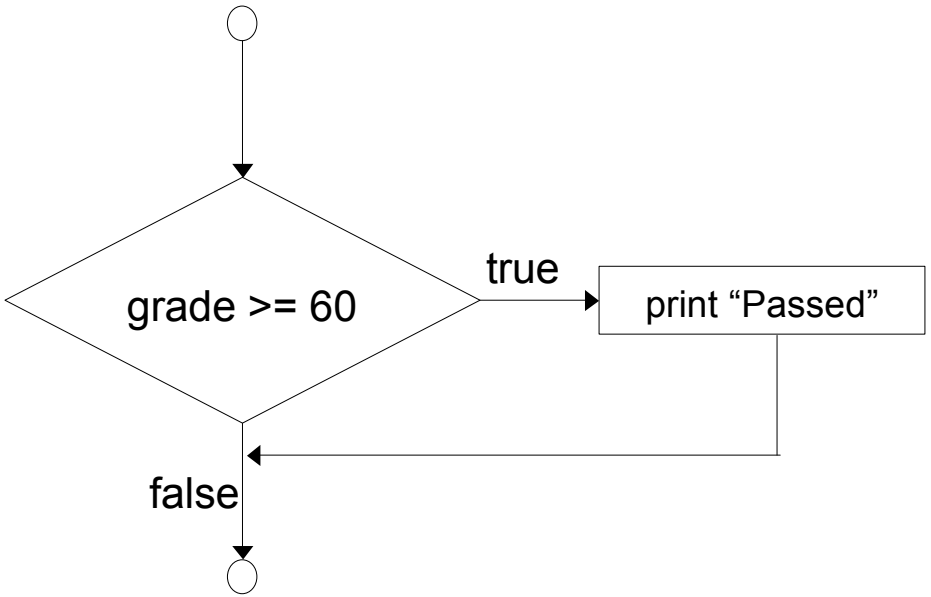
```
if ( grade >= 60 )
   cout << "Passed";
```

- Diamond symbol (decision symbol)
  - indicates decision is to be made
  - Contains an expression that can be true or false.
    - Test the condition, follow appropriate path
- **`if`** structure is a single-entry/single-exit structure



2.5 The if Selection Structure

- Flowchart of pseudocode statement



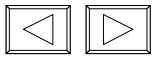
A decision can be made on any expression.

zero - **`false`**

nonzero - **`true`**

Example:

`3 - 4` is **`true`**

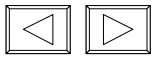


2.6 The `if/else` Selection Structure

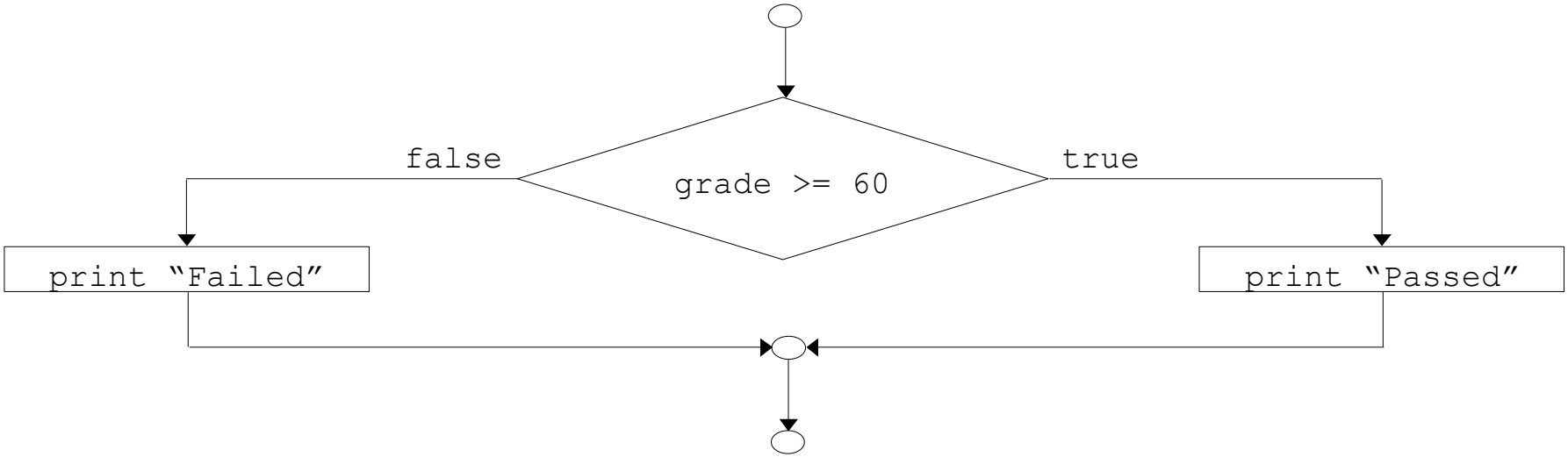
- **`if`**
  - Only performs an action if the condition is true
- **`if/else`**
  - A different action is performed when condition is true and when condition is false
- Psuedocode

  if student's grade is greater than or equal to 60
  print "Passed"
  else
  print "Failed"
- C++ code

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

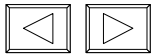



2.6 The `if/else` Selection Structure



- Ternary conditional operator (**`?:`**)
  - Takes three arguments (condition, value if **`true`**, value if **`false`**)
- Our pseudocode could be written:

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

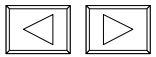



2.6 The `if/else` Selection Structure

- Nested **`if/else`** structures
  - Test for multiple cases by placing **`if/else`** selection structures inside **`if/else`** selection structures.

```
if student's grade is greater than or equal to 90
   Print "A"
else
   if student's grade is greater than or equal to 80
         Print "B"
      else
    if student's grade is greater than or equal to 70
          Print "C"
       else
          if student's grade is greater than or equal to 60
            Print "D"
   else
     Print "F"
```

  - Once a condition is met, the rest of the statements are skipped



2.6 The if/else Selection Structure

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

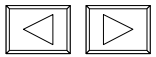
```
if ( grade >= 60 )
   cout << "Passed.\n";
else {
   cout << "Failed.\n";
   cout << "You must take this course
again.\n";
}
```

  - Without the braces,

```
cout << "You must take this course again.\n";
```

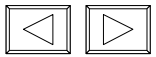
  would be automatically executed

- Block
  - Compound statements with declarations



2.6 The if/else Selection Structure

- Syntax errors
  - Errors caught by compiler
- Logic errors
  - Errors which have their effect at execution time
    - Non-fatal logic errors
      - program runs, but has incorrect output
    - Fatal logic errors
      - program exits prematurely



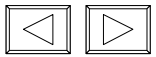
2.7 The while Repetition Structure

- Repetition structure
  - Programmer specifies an action to be repeated while some condition remains true
  - Psuedocode

    while there are more items on my shopping list

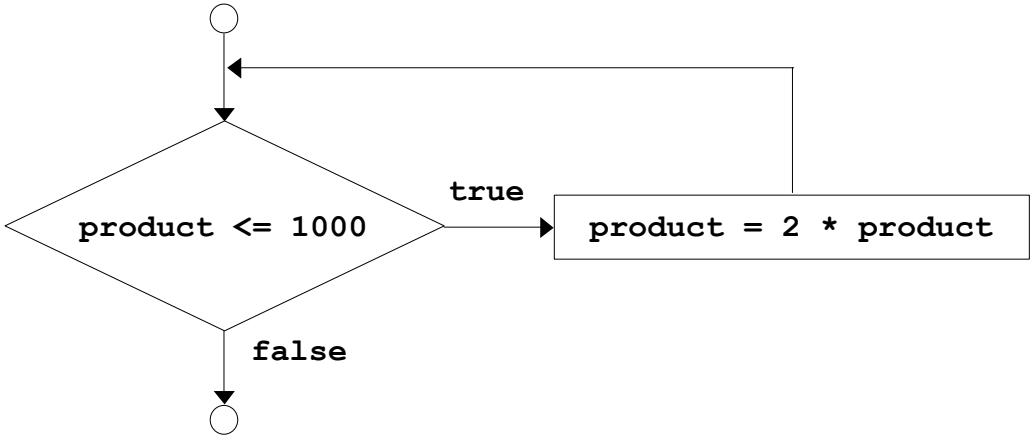
    Purchase next item and cross it off my list
  - **`while`** loop repeated until condition becomes false.
- Example

```
int product = 2;
while ( product <= 1000 )
   product = 2 * product;
```

2.7 The while Repetition Structure

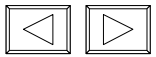
- Flowchart of **`while`** loop



2.8 Formulating Algorithms (Counter-Controlled Repetition)

- Counter-controlled repetition
  - Loop repeated until counter reaches a certain value.
- Definite repetition
  - Number of repetitions is known
- Example

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



2.8 Formulating Algorithms (Counter-Controlled Repetition)

- Pseudocode for example:

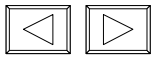
Set total to zero

Set grade counter to one

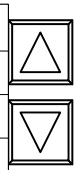
While grade counter is less than or equal to ten
Input the next grade
Add the grade into the total
Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

- Following is the C++ code for this example



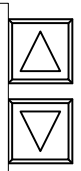
```
1  // Fig. 2.7: fig02_07.cpp
2  // Class average program with counter-controlled repetition
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11    int total,        // sum of grades
12        gradeCounter, // number of grades entered
13        grade,        // one grade
14        average;      // average of grades
15
16    // initialization phase
17    total = 0;                          // clear total
18    gradeCounter = 1;                   // prepare to loop
19
20    // processing phase
21    while ( gradeCounter <= 10 ) {      // loop 10
22       cout << "Enter grade: ";         // prompt for input
23       cin >> grade;                   // input grade
24       total = total + grade;          // add grade to total
25       gradeCounter = gradeCounter + 1;  // increment counter
26    }
27
28    // termination phase
29    average = total / 10;               // integer division
30    cout << "Class average is " << average << endl;
31
32    return 0;   // indicate program ended successfully
33 }
```



Outline

1. **Initialize Variables**
2. **Execute Loop**
3. **Output results**

The counter gets incremented each time the loop executes. Eventually, the counter causes the loop to end.



Outline

Program Output

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

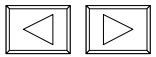
2.9 Formulating Algorithms with Top-Down, Stepwise Refinement (Sentinel-Controlled Repetition)

- Suppose the problem becomes:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

  – Unknown number of students - how will the program know to end?

- Sentinel value
  – Indicates “end of data entry”
  – Loop ends when sentinel inputted
  – Sentinel value chosen so it cannot be confused with a regular input (such as -1 in this case)



2.9 Formulating Algorithms with Top-Down, Stepwise Refinement (Sentinel-Controlled Repetition)

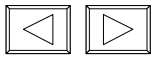
- Top-down, stepwise refinement
  - begin with a pseudocode representation of the top:

    Determine the class average for the quiz
  - Divide top into smaller tasks and list them in order:

    Initialize variables

    Input, sum and count the quiz grades

    Calculate and print the class average



2.9 Formulating Algorithms with Top-Down, Stepwise Refinement

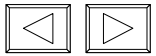
- Many programs can be divided into three phases:
  - Initialization
    - Initializes the program variables
  - Processing
    - Inputs data values and adjusts program variables accordingly
  - Termination
    - Calculates and prints the final results.
    - Helps the breakup of programs for top-down refinement.
- Refine the initialization phase from

  Initialize variables

  to

  Initialize total to zero

  Initialize counter to zero



2.9 Formulating Algorithms with Top-Down, Stepwise Refinement

- Refine

  Input, sum and count the quiz grades

  to

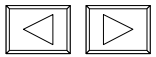
  Input the first grade (possibly the sentinel)
  While the user has not as yet entered the sentinel
  Add this grade into the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

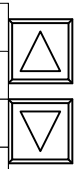
- Refine

  Calculate and print the class average

  to

  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"





Outline

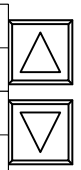
1. Initialize Variables

2. Get user input

2.1 Perform Loop

```
1  // Fig. 2.9: fig02_09.cpp
2  // Class average program with sentinel-controlled repetition.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::ios;
9
10 #include <iomanip>
11
12 using std::setprecision;
13 using std::setiosflags;
14
15 int main()
16 {
17    int total,        // sum of grades
18        gradeCounter, // number of grades entered
19        grade;        // one grade
20    double average;   // number with decimal point for average
21
22    // initialization phase
23    total = 0;
24    gradeCounter = 0;
25
26    // processing phase
27    cout << "Enter grade, -1 to end: ";
28    cin >> grade;
29
30    while ( grade != -1 ) {
```

Data type **double** used to represent decimal numbers.



Outline

3. Calculate Average

3.1 Print Results

```
31       total = total + grade;
32       gradeCounter = gradeCounter + 1;
33       cout << "Enter grade, -1 to end: ";
34       cin >> grade;
35    }
36
37    // termination phase
38    if ( gradeCounter != 0 ) {
39       average = static_cast< double >( total ) / gradeCounter;
40       cout << "Class average is " << setprecision( 2 )
41            << setiosflags( ios::fixed | ios::showpoint )
42            << average << endl;
43    }
44
45
46
47
48
```

`static_cast<double>()` - treats **`total`** as a **`double`** temporarily.

Required because dividing two integers truncates the remainder.

`gradeCounter` is an **`int`**, but it gets *promoted* to **`double`**.

`| ios::showpoint)` - stream

ers with a fixed number of decimal

decimal point and trailing zeros, even if

`cision(2)` - prints only two digits

mal point.

`|` – separates multip

Programs that use this must include **`<iomanip>`**

```
Ent
Ent
Ent
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

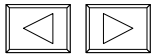
2.10 Nested control structures

- Problem:

  A college has a list of test results (1 = pass, 2 = fail) for 10 students. Write a program that analyzes the results. If more than 8 students pass, print "Raise Tuition".

- We can see that
  - The program must process 10 test results. A counter-controlled loop will be used.
  - Two counters can be used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
  - Each test result is a number—either a 1 or a 2. If the number is not a 1, we assume that it is a 2.
- Top level outline:

  Analyze exam results and decide if tuition should be raised



2.10 Nested control structures

- First Refinement:

  Initialize variables

  Input the ten quiz grades and count passes and failures

  Print a summary of the exam results and decide if tuition should be raised

- Refine

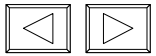
  Initialize variables

  to

  Initialize passes to zero

  Initialize failures to zero

  Initialize student counter to one



2.10 Nested control structures

- Refine

Input the ten quiz grades and count passes and failures

to

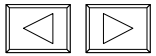
While student counter is less than or equal to ten
Input the next exam result
If the student passed
Add one to passes
Else
Add one to failures
Add one to student counter

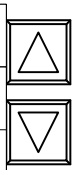
- Refine

Print a summary of the exam results and decide if tuition should be raised

to

Print the number of passes
Print the number of failures
If more than eight students passed
Print "Raise tuition"





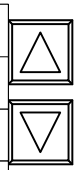
Outline

1. Initialize variables

2. Input data and count passes/failures

```
1  // Fig. 2.11: fig02_11.cpp
2  // Analysis of examination results
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11    // initialize variables in declarations
12    int passes = 0,           // number of passes
13        failures = 0,         // number of failures
14        studentCounter = 1,   // student counter
15        result;               // one exam result
16
17    // process 10 students; counter-controlled loop
18    while ( studentCounter <= 10 ) {
19       cout << "Enter result (1=pass,2=fail): ";
20       cin >> result;
21
22       if ( result == 1 )        // if/else nested in while
23          passes = passes + 1;
```

```
24       else
25          failures = failures + 1;
26
27       studentCounter = studentCounter + 1;
28    }
29
30    // termination phase
31    cout << "Passed " << passes << endl;
32    cout << "Failed " << failures << endl;
33
34    if ( passes > 8 )
35       cout << "Raise tuition " << endl;
36
37    return 0;   // successful termination
38 }
```



Outline

3. Print results

Program Output

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

2.11 Assignment Operators

- Assignment expression abbreviations

  `c = c + 3;` can be abbreviated as `c += 3;` using the addition assignment operator

- Statements of the form

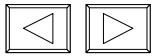
  ```
  variable = variable operator expression;
  ```

  can be rewritten as

  ```
  variable operator= expression;
  ```

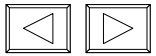
- Examples of other assignment operators include:

  ```
  d -= 4      (d = d - 4)
  e *= 5      (e = e * 5)
  f /= 3      (f = f / 3)
  g %= 9      (g = g % 9)
  ```

2.12 Increment and Decrement Operators

- Increment operator (`++`) - can be used instead of `c += 1`
- Decrement operator (`--`) - can be used instead of `c -= 1`
  - Preincrement
    - When the operator is used before the variable (`++c` or `--c`)
    - Variable is changed, then the expression it is in is evaluated.
  - Posincrement
    - When the operator is used after the variable (`c++` or `c--`)
    - Expression the variable is in executes, then the variable is changed.
- If `c = 5`, then
  - `cout << ++c;` prints out `6` (`c` is changed before `cout` is executed)
  - `cout << c++;` prints out `5` (`cout` is executed before the increment. `c` now has the value of `6`)



2.12 Increment and Decrement Operators

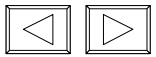
- When Variable is not in an expression
  - Preincrementing and postincrementing have the same effect.

```
++c;
cout << c;
```

and

```
c++;
cout << c;
```

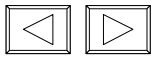
have the same effect.



2.13 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - The name of a control variable (or loop counter).
  - The initial value of the control variable.
  - The condition that tests for the final value of the control variable (i.e., whether looping should continue).
  - The increment (or decrement) by which the control variable is modified each time through the loop.
- Example:

```
int counter =1;           //initialization
while (counter <= 10){    //repetition
condition
   cout << counter << endl;
   ++counter;             //increment
 }
```

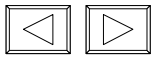



2.13 Essentials of Counter-Controlled Repetition

- The declaration

```
int counter = 1;
```

  - Names **`counter`**
  - Declares **`counter`** to be an integer
  - Reserves space for **`counter`** in memory
  - Sets **`counter`** to an initial value of **`1`**



2.14 The for Repetition Structure

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

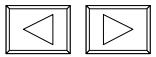
```
for ( initialization; LoopContinuationTest;
      increment )
   statement
```

- Example:

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

  – Prints the integers from one to ten

No semicolon after last statement



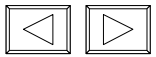
2.14 The for Repetition Structure

- **`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;
```

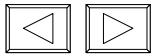



2.15 Examples Using the for Structure

- Program to sum the even numbers from 2 to 100

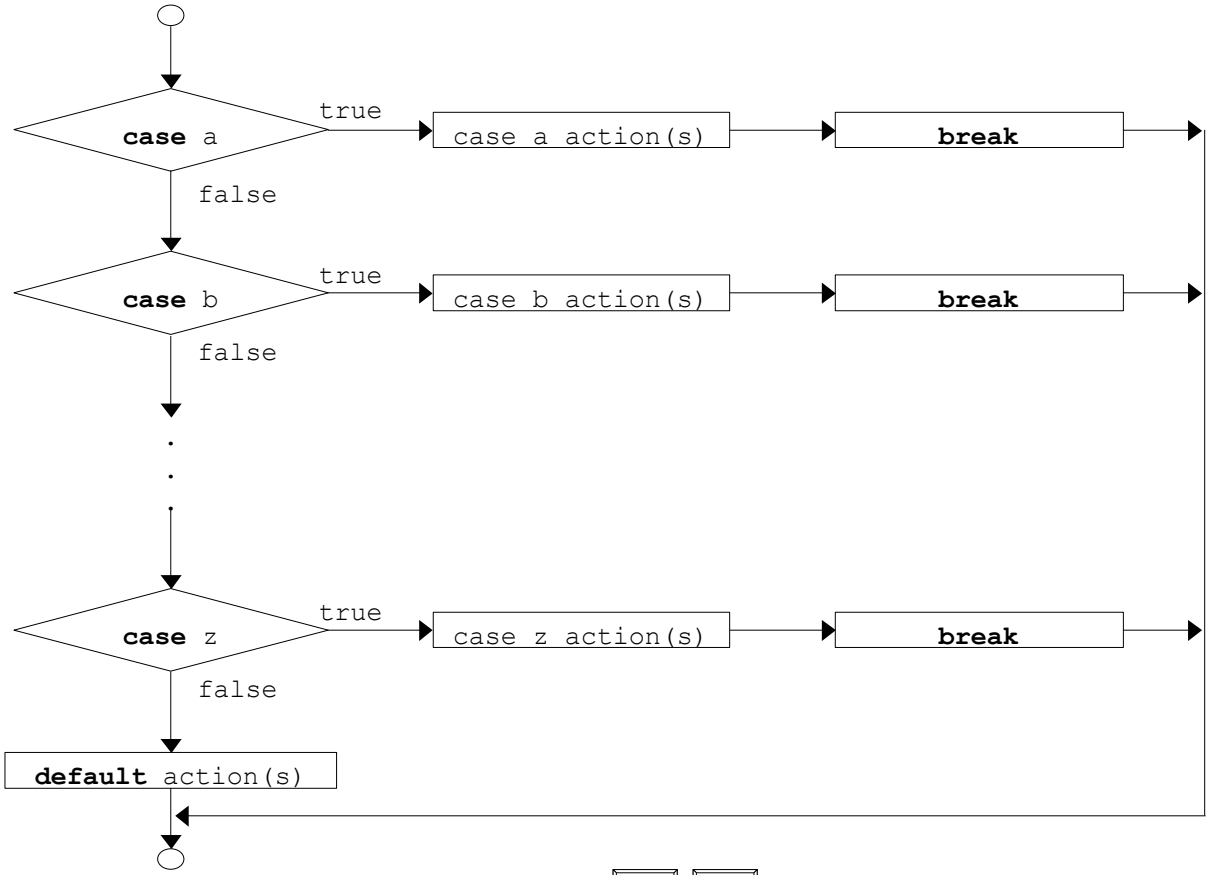
```
1  // Fig. 2.20: fig02_20.cpp
2  // Summation with for
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int sum = 0;
11
12    for ( int number = 2; number <= 100; number += 2 )
13       sum += number;
14
15    cout << "Sum is " << sum << endl;
16
17    return 0;
18 }
```

```
Sum is 2550
```

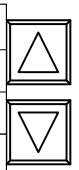
2.16 The switch Multiple-Selection Structure

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



```
1  // Fig. 2.22: fig02_22.cpp
2  // Counting letter grades
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11    int grade,        // one grade
12        aCount = 0,  // number of A's
13        bCount = 0,  // number of B's
14        cCount = 0,  // number of C's
15        dCount = 0,  // number of D's
16        fCount = 0;  // number of F's
17
18    cout << "Enter the letter grades." << endl
19         << "Enter the EOF character to end input." << endl;
20
21    while ( ( grade = cin.get() ) != EOF ) {
22
23       switch ( grade ) {
24
25          case 'A':  // grade was uppercase A
26          case 'a':  // or lowercase a
27             ++aCount;
28             break;  // necessary to exit switch
29
30          case 'B':  // grade was uppercase B
31          case 'b':  // or lowercase b
32             ++bCount;
33             break;
34
```

Notice how the **`case`** statement is used



Outline

1. **Initialize variables**

2. **Input data**

2.1 Use switch loop to update `count`

```
35          case 'C':  // grade was uppercase C
36          case 'c':  // or lowercase c
37             ++cCount;
38             break;
39
40          case 'D':  // grade was upper
41          case 'd':  // or lowercase d
42             ++dCount;
43             break;
44
45          case 'F':  // grade was uppercase
46          case 'f':  // or lowercase f
47             ++fCount;
48             break;
49
50          case '\n': // ignore newlines,
51          case '\t': // tabs,
52          case ' ':  // and spaces in
53             break;
54
55          default:   // catch all other characters
56             cout << "Incorrect letter grade entered."
57                  << " Enter a new grade." << endl;
58             break;  // optional
59       }
60    }
61
62    cout << "\n\nTotals for each letter grade are:"
63         << "\nA: " << aCount
64         << "\nB: " << bCount
65         << "\nC: " << cCount
66         << "\nD: " << dCount
67         << "\nF: " << fCount << endl;
68
69    return 0;
70 }
```

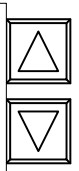
break causes **switch** to end and the program continues with the first statement after the **switch** structure.

Notice the **default** statement.

Outline

2.1 Use switch loop to
te count

int results



Outline

Program Output

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

2.17 The do/while Repetition Structure

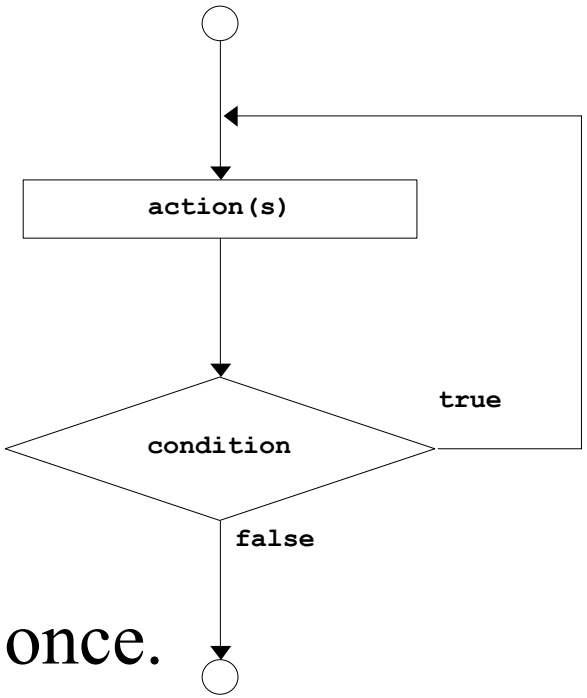
- The **do/while** repetition structure is similar to the **while** structure,
  - Condition for repetition tested after the body of the loop is executed
- Format:

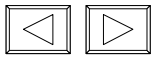
```
do {
   statement
} while ( condition );
```

- Example (letting counter = 1):

```
do {
  cout << counter << " ";
} while (++counter <= 10);
```

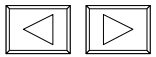
  - This prints the integers from **1** to **10**
- All actions are performed at least once.





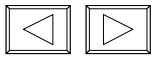
2.18 The break and continue Statements

- **`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



2.18 The break and continue Statements

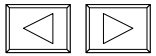
- **`Continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure and proceeds with the next iteration of the loop
  - In **`while`** and **`do/while`**, the loop-continuation test is evaluated immediately after the **`continue`** statement is executed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



2.19 Logical Operators

- **`&&`** (logical **`AND`**)
  - Returns **`true`** if both conditions are **`true`**
- **`||`** (logical **`OR`**)
  - Returns **`true`** if either of its conditions are **`true`**
- **`!`** (logical **`NOT`**, logical negation)
  - Reverses the truth/falsity of its condition
  - Returns **`true`** when its condition is **`false`**
  - Is a unary operator, only takes one condition
- Logical operators used as conditions in loops

Expression	Result
`true && false`	`false`
`true || false`	`true`
`!false`	`true`



2.20 Confusing Equality (==) and Assignment (=) Operators

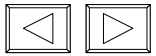
- These errors are damaging because they do not ordinarily cause syntax errors.
  - Recall that any expression that produces a value can be used in control structures. Nonzero values are **`true`**, and zero values are **`false`**
- Example:

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

  - Checks the paycode, and if it is **`4`** then a bonus is awarded
- If **`==`** was replaced with **`=`**

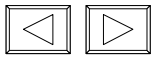
```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

  - Sets **`paycode`** to **`4`**
  - **`4`** is nonzero, so the expression is **`true`** and a bonus is awarded, regardless of **`paycode`**.



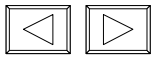
2.20 Confusing Equality (==) and Assignment (=) Operators

- Lvalues
  - Expressions that can appear on the left side of an equation
  - Their values can be changed
  - Variable names are a common example (as in **`x = 4;`**)
- Rvalues
  - Expressions that can only appear on the right side of an equation
  - Constants, such as numbers (i.e. you cannot write **`4 = x;`**)
- Lvalues can be used as rvalues, but not vice versa



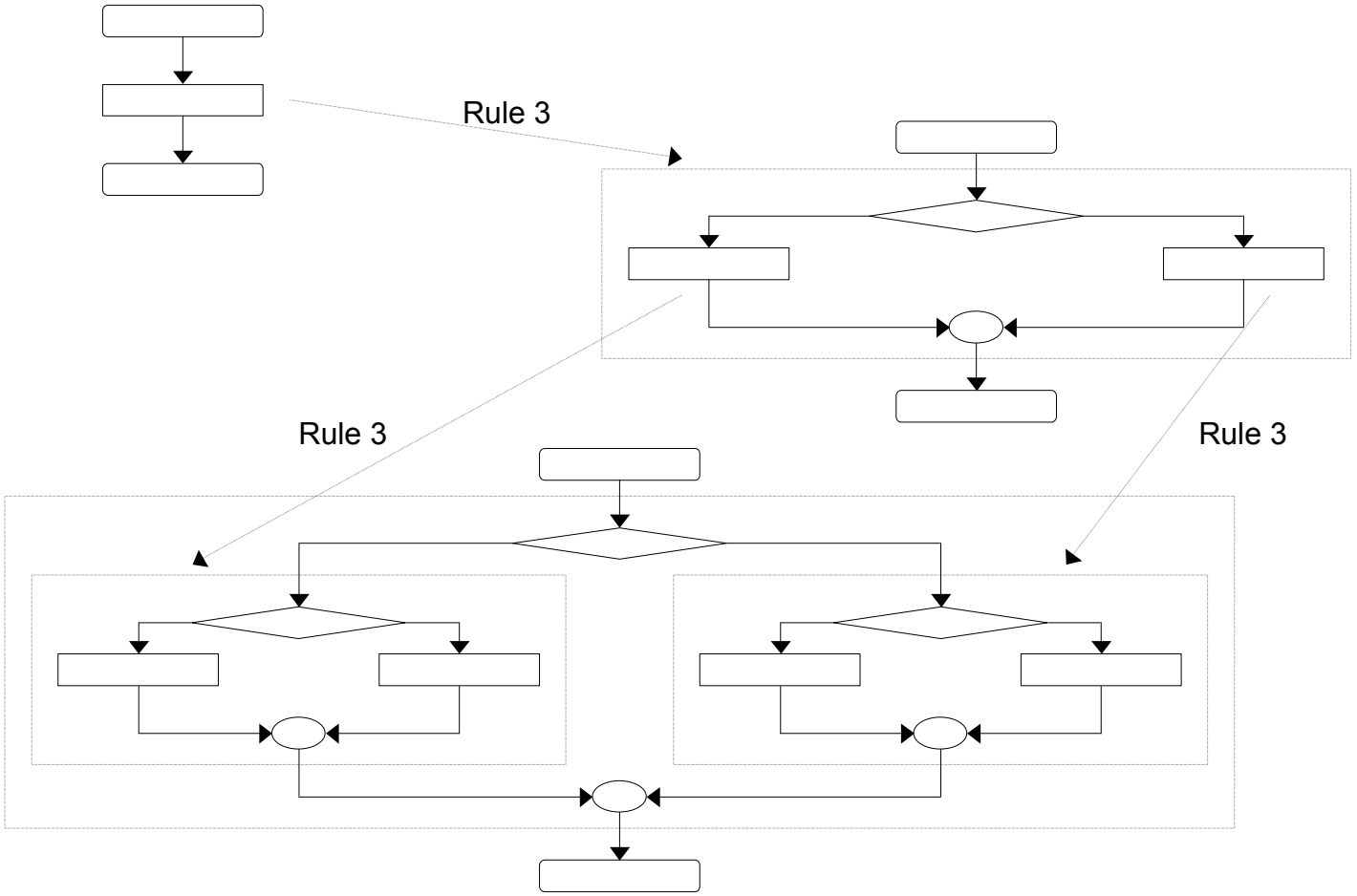
2.21 Structured-Programming Summary

- Structured programming
  - Programs are easier to understand, test, debug and, modify.
- Rules for structured programming
  - Only single-entry/single-exit control structures are used
  - Rules:
    1) Begin with the “simplest flowchart”.
    2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
    3) Any rectangle (action) can be replaced by any control structure (sequence, if, if/else, switch, while, do/while or for).
    4) Rules 2 and 3 can be applied in any order and multiple times.



2.21 Structured-Programming Summary

Representation of Rule 3 (replacing any rectangle with a control structure)



2.21 Structured-Programming Summary

- All programs can be broken down into
  - Sequence
  - Selection
    - **`if`**, **`if/else`**, or **`switch`**
    - Any selection can be rewritten as an **`if`** statement
  - Repetition
    - **`while`**, **`do/while`** or **`for`**
    - Any repetition structure can be rewritten as a **`while`** statement