Chapter 2 - Control Structures

Outline

Chapter 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

- Computing problems
  - Solved by executing a series of actions in a specific order
- Algorithm a procedure determining
  - Actions to be executed
  - Order to be executed
  - Example: recipe
- Program control
  - Specifies the order in which statements are executed

2.3 Pseudocode

- Pseudocode
  - Artificial, informal language used to develop algorithms
  - Similar to everyday English
- Not executed on computers
  - Used to think out program before coding
    - Easy to convert into C++ program
  - Only executable statements
    - No need to declare variables

2.4 Control Structures

- Sequential execution
  - Statements executed in order
- Transfer of control
  - Next statement executed *not* next one in sequence
- 3 control structures (Bohm and Jacopini)
  - Sequence structure
    - Programs executed sequentially by default
  - Selection structures
    - **`if`**, **`if/else`**, **`switch`**
  - Repetition structures
    - **`while`**, **`do/while`**, **`for`**

2.4 Control Structures

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

C++ Keywords

Keywords common to the C and C++ programming languages

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

C++ only keywords

`asm`	`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`false`	`friend`
`inline`	`mutable`	`namespace`	`new`	`operator`
`private`	`protected`	`public`	`reinterpret_cast`	
`static_cast`	`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`	`virtual`
`wchar_t`				

2.4 Control Structures

- Flowchart
  - Graphical representation of an algorithm
  - Special-purpose symbols connected by arrows (flowlines)
  - Rectangle symbol (action symbol)
    - Any type of action
  - Oval symbol
    - Beginning or end of a program, or a section of code (circles)
- Single-entry/single-exit control structures
  - Connect exit point of one to entry point of the next
  - Control structure stacking

2.5 if Selection Structure

- Selection structure
  - 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, program continues to next statement
  - If the condition is **`false`**
    - Print statement ignored, program continues
  - Indenting makes programs easier to read
    - C++ ignores whitespace characters (tabs, spaces, etc.)

2.5 if Selection Structure

- Translation into C++

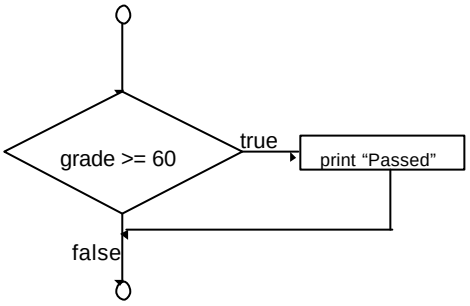
  If student's grade is greater than or equal to 60
  Print "Passed"

```
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 condition, follow path
- **`if`** structure
  - Single-entry/single-exit

2.5 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 `if/else` Selection Structure

- **`if`**
  - Performs action if condition true
- **`if/else`**
  - Different actions if conditions true or false
- Pseudocode

  if student’s grade is greater than or equal to 60
  print “Passed”
  else
  print “Failed”

- C++ code

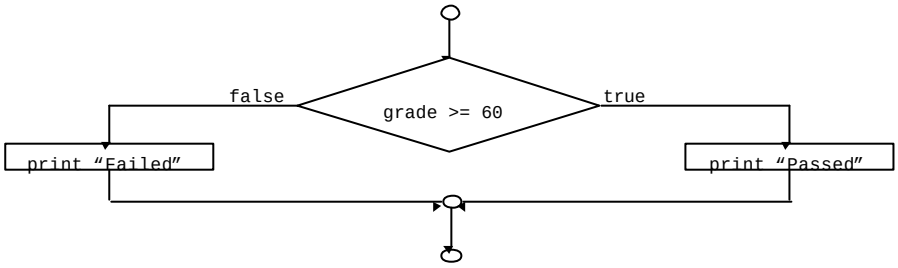
```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```


2.6 `if/else` Selection Structure

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

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

Condition Value if true Value if false



2.6 `if/else` Selection Structure

- Nested **`if/else`** structures
  - One inside another, test for multiple cases
  - Once condition met, other statements skipped

```
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”
```


2.6 `if/else` Selection Structure

- Example

```
if ( grade >= 90 )       // 90 and above
   cout << "A";
else if ( grade >= 80 )  // 80-89
   cout << "B";
else if ( grade >= 70 )  // 70-79
   cout << "C";
else if ( grade >= 60 )  // 60-69
   cout << "D";
else                     // less than 60
   cout << "F";
```


2.6 `if/else` Selection Structure

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

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

  - Without braces,

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

  always executed
- Block
  - Set of statements within braces

2.7 while Repetition Structure

- Repetition structure
  - Action repeated while some condition remains true
  - Psuedocode

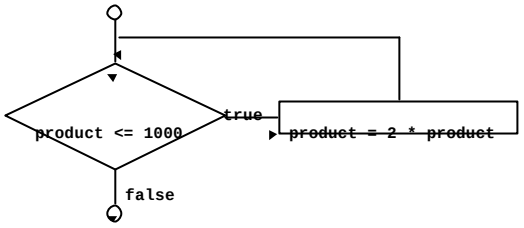
    while there are more items on my shopping list

    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 certain value
- Definite repetition
  - Number of repetitions 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

- Next: C++ code for this example

Outline

fig02_07.cpp (1 of 2)

```
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  // function main begins program execution
10 int main()
11 {
12    int total;        // sum of grades input by user
13    int gradeCounter; // number of grade to be entered next
14    int grade;        // grade value
15    int average;      // average of grades
16
17    // initialization phase
18    total = 0;          // initialize total
19    gradeCounter = 1;   // initialize loop counter
20
```


Outline

fig02_07.cpp (2 of 2)

fig02_07.cpp output (1 of 1)

```
21    // processing phase
22    while ( gradeCounter <= 10 ) {       // loop 10 times
23       cout << "Enter grade: ";          // prompt for input
24       cin >> grade;                     // read grade from user
25       total = total + grade;            // add grade to total
26       gradeCounter = gradeCounter + 1;  // increment counter
27    }
28
29    // termination phase
30    average = total / 10;                // integer division
31
32    // display result
33    cout << "Class average is
34
35    return 0;   // indicate p
36
37 } // end function main
```

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

```
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 (Sentinel-Controlled Repetition)

- Suppose 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 program know when to end?
- Sentinel value
  – Indicates "end of data entry"
  – Loop ends when sentinel input
  – Sentinel chosen so it cannot be confused with regular input
    - -1 in this case

2.9 Formulating Algorithms (Sentinel-Controlled Repetition)

- Top-down, stepwise refinement
  – Begin with pseudocode representation of top

    Determine the class average for the quiz
  – Divide top into smaller tasks, list in order

    Initialize variables

    Input, sum and count the quiz grades

    Calculate and print the class average

2.9 Formulating Algorithms (Sentinel-Controlled Repetition)

- Many programs have three phases
  - Initialization
    - Initializes the program variables
  - Processing
    - Input data, adjusts program variables
  - Termination
    - Calculate and print the final results
  - Helps break up programs for top-down refinement

2.9 Formulating Algorithms (Sentinel-Controlled Repetition)

- Refine the initialization phase

  Initialize variables

  goes to

  Initialize total to zero

  Initialize counter to zero

- Processing

  Input, sum and count the quiz grades

  goes 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)

2.9 Formulating Algorithms (Sentinel-Controlled Repetition)

- Termination

  Calculate and print the class average

  goes 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"

- Next: C++ program

Outline

fig02_09.cpp (1 of 3)

```
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::fixed;
9
10 #include <iomanip>        // parameterized stream manipulators
11
12 using std::setprecision;  // sets numeric output precision
13
14 // function main begins program execution
15 int main()
16 {
17    int total;         // sum of grades
18    int gradeCounter;  // number of grades entered
19    int grade;         // grade value
20
21    double average;    // number with decimal point for average
22
23    // initialization phase
24    total = 0;         // initialize total
25    gradeCounter = 0;  // initialize loop counter
```

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

Outline

fig02_09.cpp (2 of 3)

```
26
27    // processing phase
28    // get first grade from user
29    cout << "Enter grade, -1 to end: ";  // prompt for input
30    cin >> grade;                        // read grade from user
31
32    // loop until sentin
33    while ( grade != -1
34       total = total + g
35       gradeCounter = gr
36
37       cout << "Enter gr
38       cin >> grade;
39
40    } // end while
41
42    // termination phase
43    // if user entered at least one grade ...
44    if ( gradeCounter != 0 ) {
45
46       // calculate average of all grades entered
47       average = static_cast< double >( total ) / gradeCounter;
48
```

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

Required because dividing two integers truncates the remainder.

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

Outline

fig02_09.cpp (3 of 3)

fig02_09.cpp output (1 of 1)

```
49       // display average with two digits of precision
50       cout << "Class average is " << setprecision( 2 )
51            << fixed << average << endl;
52
53    } // end if part of if/else
54
55    else // if no grades were entered, output appropriate message
56       cout << "No grades were entered" << endl;
57
58    return 0;   // indicate program ended successfully
59
60 } // end function main
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
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
```

`fixed` forces output to print in fixed point format (not scientific notation). Also, forces trailing zeros and decimal point to print.

~~Include~~ **~~`<iostream>`~~**

`ion(2)` prints two digits past (rounded to fit precision).

~~t use this must include~~ **~~`<iomanip>`~~**

2.10 Nested Control Structures

- Problem statement

  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".

- Notice that
  - Program processes 10 results
    - Fixed number, use counter-controlled loop
  - Two counters can be used
    - One counts number that passed
    - Another counts number that fail
  - Each test result is 1 or 2
    - If not 1, assume 2

2.10 Nested Control Structures

- Top level outline

  Analyze exam results and decide if tuition should be raised

- 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

2.10 Nested Control Structures

- 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"

- Program next

Outline

fig02_11.cpp
(1 of 2)

```
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  // function main begins program execution
10 int main()
11 {
12    // initialize variables in declarations
13    int passes = 0;           // number of passes
14    int failures = 0;         // number of failures
15    int studentCounter = 1;   // student counter
16    int result;               // one exam result
17
18    // process 10 students using counter-controlled loop
19    while ( studentCounter <= 10 ) {
20
21       // prompt user for input and obtain value from user
22       cout << "Enter result (1 = pass, 2 = fail): ";
23       cin >> result;
24
```


Outline

fig02_11.cpp
(2 of 2)

```
25       // if result 1, increment passes; if/else nested in while
26       if ( result == 1 )        // if/else nested in while
27          passes = passes + 1;
28
29       else  // if result not 1, increment failures
30          failures = failures + 1;
31
32       // increment studentCounter so loop eventually terminates
33       studentCounter = studentCounter + 1;
34
35    } // end while
36
37    // termination phase; display number of passes and failures
38    cout << "Passed " << passes << endl;
39    cout << "Failed " << failures << endl;
40
41    // if more than eight students passed, print "raise tuition"
42    if ( passes > 8 )
43       cout << "Raise tuition " << endl;
44
45    return 0;   // successful termination
46
47 } // end function main
```

Outline

fig02_11.cpp
output (1 of 1)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
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): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4

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
  - Addition assignment operator
    `c = c + 3;` abbreviated to
    `c += 3;`
- Statements of the form
  `variable = variable operator expression;`
  can be rewritten as
  `variable operator= expression;`
- Other assignment operators

```
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.

2.12 Increment and Decrement Operators

- Increment operator (**++**)
  - Increment variable by one
  - **`c++`**
    - Same as **`c += 1`**
- Decrement operator (**--**) similar
  - Decrement variable by one
  - **`c--`**

2.12 Increment and Decrement Operators

- Preincrement
  - Variable changed before used in expression
    - Operator before variable (**`++c`** or **`--c`**)
- Postincrement
  - Incremented changed after expression
    - Operator after variable (**`c++, c--`**)

2.12 Increment and Decrement Operators

- If **`c = 5`**, then
  - **`cout << ++c;`**
    - **`c`** is changed to **`6`**, then printed out
  - **`cout << c++;`**
    - Prints out **`5`** (**`cout`** is executed before the increment.
    - **`c`** then becomes **`6`**

2.12 Increment and Decrement Operators

- When variable not in expression
  - Preincrementing and postincrementing have same effect

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

and

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

are the same

Outline

fig02_14.cpp
(1 of 2)

```
1  // Fig. 2.14: fig02_14.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11    int c;                    // declare variable
12
13    // demonstrate postincrement
14    c = 5;                    // assign 5 to c
15    cout << c << endl;        // print 5
16    cout << c++ << endl;      // print 5 then postincrement
17    cout << c << endl << endl; // print 6
18
19    // demonstrate preincrement
20    c = 5;                    // assign 5 to c
21    cout << c << endl;        // print 5
22    cout << ++c << endl;      // preincrement then print 6
23    cout << c << endl;        // print 6
```

```
24
25    return 0;   // indicate successful termination
26
27 } // end function main
```

```
5
5
6

5
6
6
```

Outline

fig02_14.cpp (2 of 2)

fig02_14.cpp output (1 of 1)

2.13 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - Name of control variable/loop counter
  - Initial value of control variable
  - Condition to test for final value
  - Increment/decrement to modify control variable when looping

Outline

fig02_16.cpp (1 of 1)

```
1  // Fig. 2.16: fig02_16.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11    int counter = 1;             // initialization
12
13    while ( counter <= 10 ) {    // repetition condition
14       cout << counter << endl;  // display counter
15       ++counter;                // increment
16
17    } // end while
18
19    return 0;   // indicate successful termination
20
21 } // end function main
```


Outline

fig02_16.cpp output (1 of 1)

```
1
2
3
4
5
6
7
8
9
10
```

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 for Repetition Structure

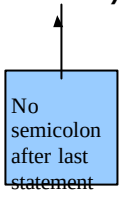
- General format when using **`for`** loops

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

- Example

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

  - Prints integers from one to ten



Outline

fig02_17.cpp (1 of 1)

```
1  // Fig. 2.17: fig02_17.cpp
2  // Counter-controlled repetition with the for structure.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11    // Initialization, repetition condition and incrementing
12    // are all included in the for structure header.
13
14    for ( int counter = 1; counter <= 10; counter++ )
15       cout << counter << endl;
16
17    return 0;   // indicate successful termination
18
19 } // end function main
```


Outline

fig02_17.cpp output (1 of 1)

```
1
2
3
4
5
6
7
8
9
10
```

2.14 for Repetition Structure

- **for** loops can usually be rewritten as **while** loops

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

- Initialization and increment
  - For multiple variables, use comma-separated lists

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


Outline

fig02_20.cpp (1 of 1)

fig02_20.cpp output (1 of 1)

```
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  // function main begins program execution
9  int main()
10 {
11    int sum = 0;                      // initialize sum
12
13    // sum even integers from 2 through 100
14    for ( int number = 2; number <= 100; number += 2 )
15       sum += number;                 // add number to sum
16
17    cout << "Sum is " << sum << endl;  // output sum
18    return 0;                         // successful termination
19
20 } // end function main
```

```
Sum is 2550
```

2.15 Examples Using the for Structure

- Program to calculate compound interest
- *A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:*
  $a = p(1+r)^n$
- *p* is the original amount invested (i.e., the principal),
  r is the annual interest rate,
  n is the number of years and
  a is the amount on deposit at the end of the *n*th year

Outline

fig02_21.cpp
(1 of 2)

```
1  // Fig. 2.21: fig02_21.cpp
2  // Calculating compound interest.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::ios;
8  using std::fixed;
9
10 #include <iomanip>
11
12 using std::setw;
13 using std::setprecision;
14
15 #include <cmath>  // enables program to use function pow
16
17 // function main begins program execution
18 int main()
19 {
20    double amount;              // amount on deposit
21    double principal = 1000.0;  // starting principal
22    double rate = .05;          // interest rate
23
```

<cmath> header needed for the **pow** function (program will not compile without it).

Outline

```
24    // output table column heads
25    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
26
27    // set floating-point number format
28    cout << fixed << setprecision( 2 );
29
30    // calculate amount on deposit for each of ten years
31    for ( int year = 1; year <= 10; year++ ) {
32
33       // calculate new amount for specified year
34       amount = principal * pow( 1.0 + rate, year );
35
36       // output one table row
37       cout << setw( 4 ) << year
38            << setw( 21 ) << amount << endl;
39
40    } // end for
41
42    return 0;   // indicate successful termination
43
44 } // end function main
```

Sets the field width to at least 21 characters. If output less than 21, it is right-justified.

pow(x,y) = x raised to the yth power.

fig02_21.cpp
(2 of 2)

Outline

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Numbers are right-justified due to setw statements (at positions 4 and 21).

fig02_21.cpp
output (1 of 1)

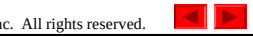
2.16 switch Multiple-Selection Structure

- **switch**
  - Test variable for multiple values
  - Series of **case** labels and optional **default** case

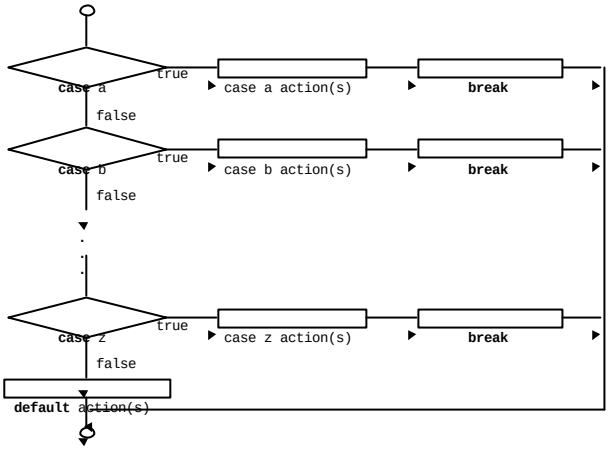
```
switch ( variable ) {
     case value1:         // taken if variable == value1
     statements
     break;              // necessary to exit switch

     case value2:
     case value3:       // taken if variable == value2 or == value3
     statements
     break;

     default:          // taken if variable matches no other cases
     statements
      break;
}
```

2.16 switch Multiple-Selection Structure



2.16 switch Multiple-Selection Structure

- Example upcoming
  - Program to read grades (A-F)
  - Display number of each grade entered
- Details about characters
  - Single characters typically stored in a **char** data type
    - **char** a 1-byte integer, so **char**s can be stored as **int**s
  - Can treat character as **int** or **char**
    - 97 is the numerical representation of lowercase ‘a’ (ASCII)
    - Use single quotes to get numerical representation of character

```
cout << "The character (" << 'a' << ") has the value "
       << static_cast< int > ( 'a' ) << endl;
```

Prints

```
The character (a) has the value 97
```


Outline

fig02_22.cpp (1 of 4)

```
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  // function main begins program execution
10 int main()
11 {
12    int grade;       // one grade
13    int aCount = 0;  // number of As
14    int bCount = 0;  // number of Bs
15    int cCount = 0;  // number of Cs
16    int dCount = 0;  // number of Ds
17    int fCount = 0;  // number of Fs
18
19    cout << "Enter the letter grades." << endl
20         << "Enter the EOF character to end input." << endl;
21
```

Outline

```
22    // loop until user types end-of-file key sequence
23    while ( ( grade = cin.get() ) != EOF ) {
24
25       // determine which grade was input
26       switch ( grade ) {  // switch structu
27
28          case 'A':        // grade was uppercase A
29          case 'a':        // or lowercase a
30             ++aCount;     // increment aCount
31             break;        //
32
33          case 'B':        //
34          case 'b':        //
35             ++bCount;     //
36             //
37
38             //
39             //
40             ++cCount;     //
41             break;        //
42
```

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

cin.get() uses dot notation (explained chapter 6). This function gets 1 character from the keyboard (after *Enter* pressed), and it is assigned to **grade**.

cin.get() returns EOF (end-of-file) after the EOF character is input, to indicate the end of data. EOF may be ctrl-d or ctrl-z, depending on your OS.

Assignment statements have a value, which is the same as the variable on the left of the **=**. The value of this statement is the same as the value returned by **cin.get()**.

This can also be used to initialize multiple variables: **a = b = c = 0;**

Compares **grade** (an **int**) to the numerical representations of **A** and **a**.

Outline

fig02_22.cpp
(3 of 4)

```
43          case 'D':        // grade was uppercase D
44          case 'd':        // or lowercase d
45             ++dCount;     // increment dCount
46             break;        // exit swit
47
48          case 'F':        // grade was
49          case 'f':        // or lowerc
50             ++fCount;     // increment
51             break;        // exit swit
52
53          case '\n':       // ignore ne
54          case '\t':       // tabs,
55          case ' ':        // and spac
56             break;        // exit swi
57
58          default:         // catch all other characters
59             cout << "Incorrect letter grade entered."
60                  << " Enter a new grade." << endl;
61             break;        // optional; will exit switch anyway
62
63       } // end switch
64
65    } // end while
66
```

This test is necessary because *Enter* is pressed after each letter grade is input. This adds a newline character that must be removed. Likewise, we want to ignore any whitespace.

Notice the **default** statement, which catches all other cases.

Outline

fig02_22.cpp (4 of 4)

```
67    // output summary of results
68    cout << "\n\nTotals for each letter grade are:"
69         << "\nA: " << aCount    // display number of A grades
70         << "\nB: " << bCount    // display number of B grades
71         << "\nC: " << cCount    // display number of C grades
72         << "\nD: " << dCount    // display number of D grades
73         << "\nF: " << fCount    // display number of F grades
74         << endl;
75
76    return 0;  // indicate successful termination
77
78 } // end function main
```


Outline

fig02_22.cpp output (1 of 1)

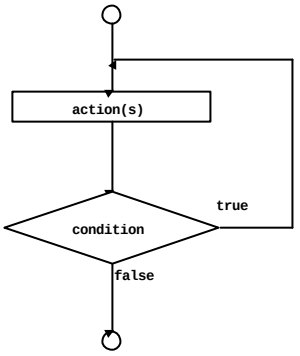
```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z


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

2.17 do/while Repetition Structure

- Similar to **`while`** structure
  - Makes loop continuation test at end, not beginning
  - Loop body executes at least once
- Format

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



Outline

fig02_24.cpp (1 of 1)

fig02_24.cpp output (1 of 1)

```
1  // Fig. 2.24: fig02_24.cpp
2  // Using the do/while repetition structure.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11    int counter = 1;
12
13    do {
14       cout << counter << "  ";   // display counter
15    } while ( ++counter <= 10 );  // end do/while
16
17    cout << endl;
18
19    return 0;   // indicate successful termination
20
21 } // end function main
```

Notice the preincrement in loop-continuation test.

```
1  2  3  4  5  6  7  8  9  10
```

2.18 break and continue Statements

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

Outline

fig02_26.cpp (1 of 2)

```
1  // Fig. 2.26: fig02_26.cpp
2  // Using the break statement in a for structure.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11
12    int x;  // x declared here so it can be used after the loop
13
14    // loop 10 times
15    for ( x = 1; x <= 10; x++ ) {
16
17       // if x is 5, terminate loop
18       if ( x == 5 )
19          break;           // break loop only if x is 5
20
21       cout << x << " ";   // display value of x
22
23    } // end for
24
25    cout << "\nBroke out of loop when x became " << x << endl;
```

Exits **`for`** structure when **`break`** executed.

```
26
27     return 0;   // indicate successful termination
28
29  } // end function main
```

```
1 2 3 4
Broke out of loop when x became 5
```

Outline

fig02_26.cpp (2 of 2)

fig02_26.cpp output (1 of 1)

2.18 break and continue Statements

- **`continue`** statement
  - Used in **`while`**, **`for`**, **`do/while`**
  - Skips remainder of loop body
  - Proceeds with next iteration of loop
- **`while`** and **`do/while`** structure
  - Loop-continuation test evaluated immediately after the **`continue`** statement
- **`for`** structure
  - Increment expression executed
  - Next, loop-continuation test evaluated

Outline

fig02_27.cpp (1 of 2)

```
1  // Fig. 2.27: fig02_27.cpp
2  // Using the continue statement in a for structure.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  // function main begins program execution
9  int main()
10 {
11    // loop 10 times
12    for ( int x = 1; x <= 10; x++ ) {
13
14       // if x is 5, continue with n
15       if ( x == 5 )
16          continue;          // skip remaining code in loop body
17
18       cout << x << " ";    // display value of x
19
20    } // end for structure
21
22    cout << "\nUsed continue to skip printing the value 5"
23         << endl;
24
25    return 0;               // indicate successful termination
```

Skips to next iteration of the loop.

Outline

fig02_27.cpp (2 of 2)

```
26
27 } // end function main
```

fig02_27.cpp output (1 of 1)

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

2.19 Logical Operators

- Used as conditions in loops, if statements
- **&&** (logical **AND**)
  - **true** if both conditions are **true**

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

- **||** (logical **OR**)
  - **true** if either of condition is **true**

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   cout << "Student grade is A" << endl;
```


2.19 Logical Operators

- **!** (logical **NOT**, logical negation)
  - Returns **true** when its condition is **false**, & vice versa

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

Alternative:

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```


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

- Common error
  - Does not typically cause syntax errors
- Aspects of problem
  - Expressions that have a value can be used for decision
    - Zero = false, nonzero = true
  - Assignment statements produce a value (the value to be assigned)

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

- Example

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

  - If paycode is 4, bonus given

- If **==** was replaced with **=**

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

  - Paycode set to 4 (no matter what it was before)
  - Statement is true (since 4 is non-zero)
  - Bonus given in every case

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

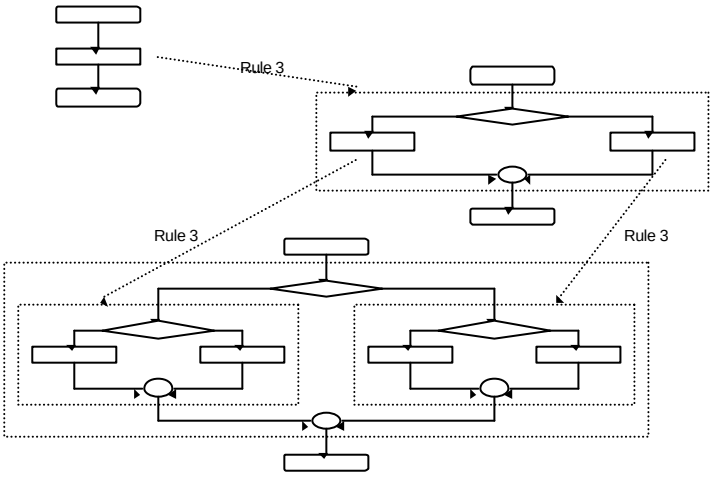
- Lvalues
  - Expressions that can appear on left side of equation
  - Can be changed (I.e., variables)
    - `x = 4;`
- Rvalues
  - Only appear on right side of equation
  - Constants, such as numbers (i.e. cannot write `4 = x;`)
- Lvalues can be used as rvalues, but not vice versa

2.21 Structured-Programming Summary

- Structured programming
  - Programs easier to understand, test, debug and modify
- Rules for structured programming
  - Only use single-entry/single-exit control structures
  - 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 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