2

Introduction to C++ Programming

What's in a name? that which we call a rose
By any other name would smell as sweet.

— William Shakespeare

When faced with a decision, I always ask,
"What would be the most fun?"

— Peggy Walker

“Take some more tea,” the March Hare said to Alice, very earnestly. “I’ve had nothing yet,” Alice replied in an offended tone: “so I can’t take more.” “You mean you can’t take less,” said the Hatter: “it’s very easy to take more than nothing.”

— Lewis Carroll

High thoughts must have high language.

— Aristophanes

OBJECTIVES

- In this chapter you will learn:
- To write simple computer programs in C++.
- To write simple input and output statements.
- To use fundamental types.
- Basic computer memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write simple decision-making statements.

Outline

2.1 Introduction

- **C++ programming**
  - **Facilitates disciplined approach to computer program design**
  - **Programs process information and display results**
- **Five examples demonstrate**
  - **How to display messages**
  - **How to obtain information from the user**
  - **How to perform arithmetic calculations**
  - **How to make decisions**

2.2 First Program in C++: Printing a Line of Text

- **Simple program**
  - **Prints a line of text**
  - **Illustrates several important features of C++**

2.2 First Program in C++: Printing a Line of Text (Cont.)

- **Comments**
  - **Explain programs to other programmers**
    - **Improve program readability**
  - **Ignored by compiler**
  - **Single-line comment**
    - **Begin with `//`**
    - **Example**
      - `// This is a text-printing program.`
  - **Multi-line comment**
    - **Start with `/*`**
    - **End with `*/`**

Outline

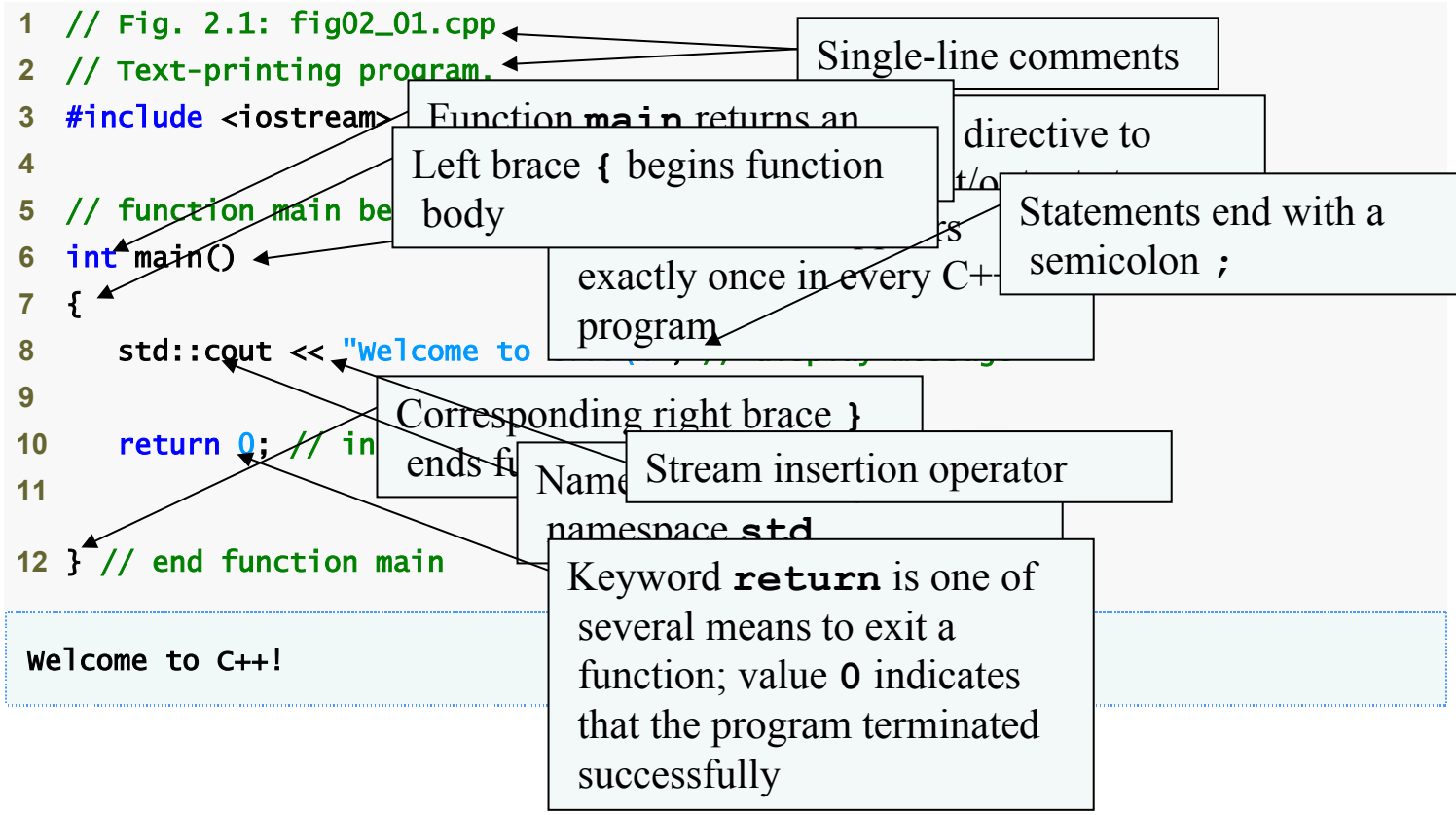

fig02_01.cpp
output (1 of 1)

Good Programming Practice 2.1

Every program should begin with a comment that describes the purpose of the program, author, date and time. (We are not showing the author, date and time in this book’ s programs because this information would be redundant.)

2.2 First Program in C++: Printing a Line of Text (Cont.)

- **Preprocessor directives**
  - **Processed by preprocessor before compiling**
  - **Begin with #**
  - **Example**
    - `#include <iostream>`
      - **Tells preprocessor to include the input/output stream header file `<iostream>`**
- **White space**
  - **Blank lines, space characters and tabs**
  - **Used to make programs easier to read**
  - **Ignored by the compiler**

Common Programming Error 2.1

Forgetting to include the `<iostream>` header file in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message, because the compiler cannot recognize references to the stream components (e.g., `cout`).

Good Programming Practice 2.2

Use blank lines and space characters to enhance program readability.

2.2 First Program in C++: Printing a Line of Text (Cont.)

- **Function `main`**
  - **A part of every C++ program**
    - **Exactly one function in a program must be `main`**
  - **Can "return" a value**
  - **Example**
    - `int main()`
      - **This `main` function returns an integer (whole number)**
  - **Body is delimited by braces (`{}`)**
- **Statements**
  - **Instruct the program to perform an action**
  - **All statements end with a semicolon (`;`)**

2.2 First Program in C++: Printing a Line of Text (Cont.)

- **Namespace**
  - `std::`
    - Specifies using a name that belongs to "namespace" `std`
    - Can be removed through use of `using` statements
- **Standard output stream object**
  - `std::cout`
    - "Connected" to screen
    - Defined in input/output stream header file `<iostream>`

2.2 First Program in C++: Printing a Line of Text (Cont.)

- **Stream insertion operator <<**
  - **Value to right (right operand) inserted into left operand**
  - **Example**
    - `std::cout << "Hello";`
      - **Inserts the string `"Hello"` into the standard output**
        - **Displays to the screen**
- **Escape characters**
  - **A character preceded by `"\"`**
    - **Indicates "special" character output**
  - **Example**
    - `"\n"`
      - **Cursor moves to beginning of next line on the screen**

Common Programming Error 2.2

Omitting the semicolon at the end of a C++ statement is a syntax error. (Again, preprocessor directives do not end in a semicolon.) The syntax of a programming language specifies the rules for creating a proper program in that language. A syntax error occurs when the compiler encounters code that violates C++'s language rules (i.e., its syntax). The compiler normally issues an error message to help the programmer locate and fix the incorrect code. (cont…)

Common Programming Error 2.2

Syntax errors are also called compiler errors, compile-time errors or compilation errors, because the compiler detects them during the compilation phase. You will be unable to execute your program until you correct all the syntax errors in it. As you will see, some compilation errors are not syntax errors.

2.2 First Program in C++: Printing a Line of Text (Cont.)

- `return` statement
  - One of several means to exit a function
  - When used at the end of `main`
    - The value `0` indicates the program terminated successfully
    - Example
      - `return 0;`

Good Programming Practice 2.3

Many programmers make the last character printed by a function a newline (\n). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development.

Escape sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\'`	Single quote. Use to print a single quote character.
`\"`	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

Good Programming Practice 2.4

Indent the entire body of each function one level within the braces that delimit the body of the function. This makes a program's functional structure stand out and helps make the program easier to read.

Good Programming Practice 2.5

Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We recommend using either 1/4-inch tab stops or (preferably) three spaces to form a level of indent.

2.3 Modifying Our First C++ Program

- **Two examples**
  - **Print text on one line using multiple statements (Fig. 2.3)**
    - **Each stream insertion resumes printing where the previous one stopped**
  - **Print text on several lines using a single statement (Fig. 2.4)**
    - **Each newline escape sequence positions the cursor to the beginning of the next line**
    - **Two newline characters back to back outputs a blank line**

Outline

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10
11    return 0; // indicate that program ended successfully
12
13 } // end function main
```

Multiple stream insertion statements produce one line of output

fig02_03.cpp

(1 of 1)

fig02_03.cpp output (1 of 1)

```
Welcome to C++!
```

Outline

```
1  // Fig. 2.4: fig02_04.cpp
2  // Printing multiple lines of text with a single statement.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome\nto\n\nC++!\n";
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

Use newline characters to print on multiple lines

fig02_04.cpp

(1 of 1)

fig02_04.cpp output (1 of 1)

```
Welcome
to

C++!
```

2.4 Another C++ Program: Adding Integers

- **Variables**
  - **Location in memory where value can be stored**
  - **Common data types (fundamental, primitive or built-in)**
    - `int` **– integer numbers**
    - `char` **– characters**
    - `double` **– floating point numbers**
  - **Declare variables with name and data type before use**
    - `int integer1;`
    - `int integer2;`
    - `int sum;`

Outline

```
1  // Fig. 2.5: fig02_05.cpp
2  // Addition program that displays the sum of two numbers.
3  #include <iostream> // allows program to perform input and output
4
5  // function main begins program execution
6  int main()
7  {
8     // variable declarations
9     int number1; // first integer to add
10    int number2; // second integ
11    int sum; // sum of number1 a
12
13    std::cout << "Enter first in
14    std::cin >> number1; // read first integer from user into number1
15
16    std::cout << "Enter second integer: "; // prompt user for data
17    std::cin >> number2; // read second integer from user into number2
18
19    sum = number1 + number2; // add the numbers; store result in sum
20
21    std::cout << "Sum is " << sum << std::endl; // display sum; end line
22
23    return 0; // indicate that program ended successfully
24
25 } // end function main
```

Declare integer variables

Use stream extraction operator with standard input stream to obtain user input

Stream manipulator **`std::endl`** outputs a newline, then "flushes output buffer"

Concatenating, chaining or cascading stream insertion operations

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

`fig02_05.cpp` (1 of 1)

`fig02_05.cpp` **`output`** (1 of 1)

2.4 Another C++ Program: Adding Integers (Cont.)

- **Variables (Cont.)**
  - **Can declare several variables of same type in one declaration**
    - **Comma-separated list**
    - `int integer1, integer2, sum;`
  - **Variable names**
    - **Valid identifier**
      - **Series of characters (letters, digits, underscores)**
      - **Cannot begin with digit**
      - **Case sensitive**

Good Programming Practice 2.6

Place a space after each comma (,) to make programs more readable.

Good Programming Practice 2.7

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

Portability Tip 2.1

C++ allows identifiers of any length, but your C++ implementation may impose some restrictions on the length of identifiers. Use identifiers of 31 characters or fewer to ensure portability.

Good Programming Practice 2.8

Choosing meaningful identifiers helps make a program self-documenting—a person can understand the program simply by reading it rather than having to refer to manuals or comments.

Good Programming Practice 2.9

Avoid using abbreviations in identifiers. This promotes program readability.

Good Programming Practice 2.10

Avoid identifiers that begin with underscores and double underscores, because C++ compilers may use names like that for their own purposes internally. This will prevent names you choose from being confused with names the compilers choose.

Error-Prevention Tip 2.1

Languages like C++ are "moving targets." As they evolve, more keywords could be added to the language. Avoid using "loaded" words like "object" as identifiers. Even though "object" is not currently a keyword in C++, it could become one; therefore, future compiling with new compilers could break existing code.

Good Programming Practice 2.11

Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program and contributes to program clarity.

Good Programming Practice 2.12

If you prefer to place declarations at the beginning of a function, separate them from the executable statements in that function with one blank line to highlight where the declarations end and the executable statements begin.

2.4 Another C++ Program: Adding Integers (Cont.)

- **Input stream object**
  - `std::cin` **from** `<iostream>`
    - **Usually connected to keyboard**
    - **Stream extraction operator >>**
      - **Waits for user to input value, press *Enter* (Return) key**
      - **Stores value in variable to right of operator**
        - **Converts value to variable data type**
    - **Example**
      - `std::cin >> number1;`
        - **Reads an integer typed at the keyboard**
        - **Stores the integer in variable** `number1`

Error-Prevention Tip 2.2

Programs should validate the correctness of all input values to prevent erroneous information from affecting a program’s calculations.

2.4 Another C++ Program: Adding Integers (Cont.)

- **Assignment operator =**
  - **Assigns value on left to variable on right**
  - **Binary operator (two operands)**
  - **Example:**
    - `sum = variable1 + variable2;`
      - **Add the values of `variable1` and `variable2`**
      - **Store result in `sum`**
- **Stream manipulator `std::endl`**
  - **Outputs a newline**
  - **Flushes the output buffer**

Good Programming Practice 2.13

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

2.4 Another C++ Program: Adding Integers (Cont.)

- **Concatenating stream insertion operations**
  - **Use multiple stream insertion operators in a single statement**
    - **Stream insertion operation knows how to output each type of data**
  - **Also called chaining or cascading**
  - **Example**
    - `std::cout << "Sum is " << number1 + number2 << std::endl;`
      - **Outputs** `"Sum is "`
      - **Then, outputs sum of** `number1` **and** `number2`
      - **Then, outputs newline and flushes output buffer**

2.5 Memory Concept

- **Variable names**
  - **Correspond to actual locations in computer's memory**
    - **Every variable has name, type, size and value**
  - **When new value placed into variable, overwrites old value**
    - **Writing to memory is destructive**
  - **Reading variables from memory nondestructive**
  - **Example**
    - `sum = number1 + number2;`
      - **Value of `sum` is overwritten**
      - **Values of `number1` and `number2` remain intact**

Fig. 2.6 | Memory location showing the name and value of variable `number1`.

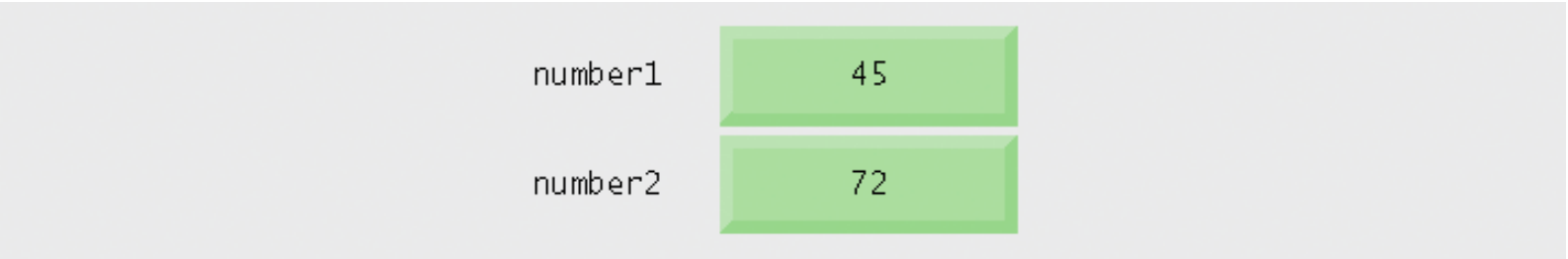

Fig. 2.7 | Memory locations after storing values for `number1` and `number2`.

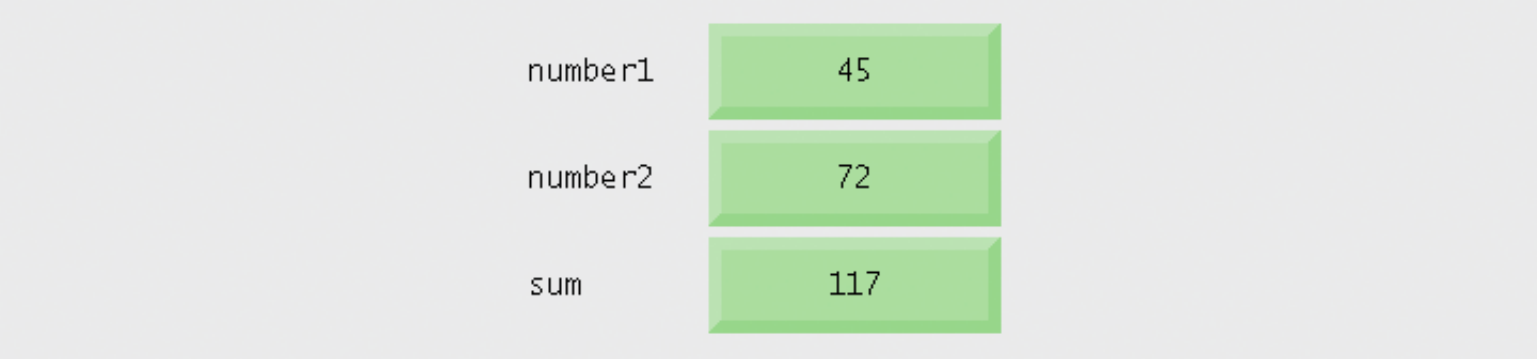

Fig. 2.8 | Memory locations after calculating and storing the `sum` of `number1` and `number2`.

2.6 Arithmetic

- **Arithmetic operators**
  - `*`
    - **Multiplication**
  - `/`
    - **Division**
    - **Integer division truncates remainder**
      - `7 / 5` **evaluates to 1**
  - `%`
    - **Modulus operator returns remainder**
      - `7 % 5` **evaluates to 2**

Common Programming Error 2.3

Attempting to use the modulus operator (%) with noninteger operands is a compilation error.

2.6 Arithmetic (Cont.)

- **Straight-line form**
  - **Required for arithmetic expressions in C++**
  - **All constants, variables and operators appear in a straight line**
- **Grouping subexpressions**
  - **Parentheses are used in C++ expressions to group subexpressions**
    - **Same manner as in algebraic expressions**
  - **Example**
    - `a * ( b + c )`
      - **Multiple `a` times the quantity `b + c`**

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm or $b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 2.9 | **Arithmetic operators.**

2.6 Arithmetic (Cont.)

- **Rules of operator precedence**
  - **Operators in parentheses evaluated first**
    - **Nested/embedded parentheses**
      - **Operators in innermost pair first**
  - **Multiplication, division, modulus applied next**
    - **Operators applied from left to right**
  - **Addition, subtraction applied last**
    - **Operators applied from left to right**

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*	Multiplication	Evaluated second. If there are several, they are evaluated left to right.
/	Division	
%	Modulus	
+	Addition	Evaluated last. If there are several, they are evaluated left to right.
-	Subtraction	

Fig. 2.10 | Precedence of arithmetic operators.

Common Programming Error 2.4

Some programming languages use operators ** or ^ to represent exponentiation. C++ does not support these exponentiation operators; using them for exponentiation results in errors.

Good Programming Practice 2.14

Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.

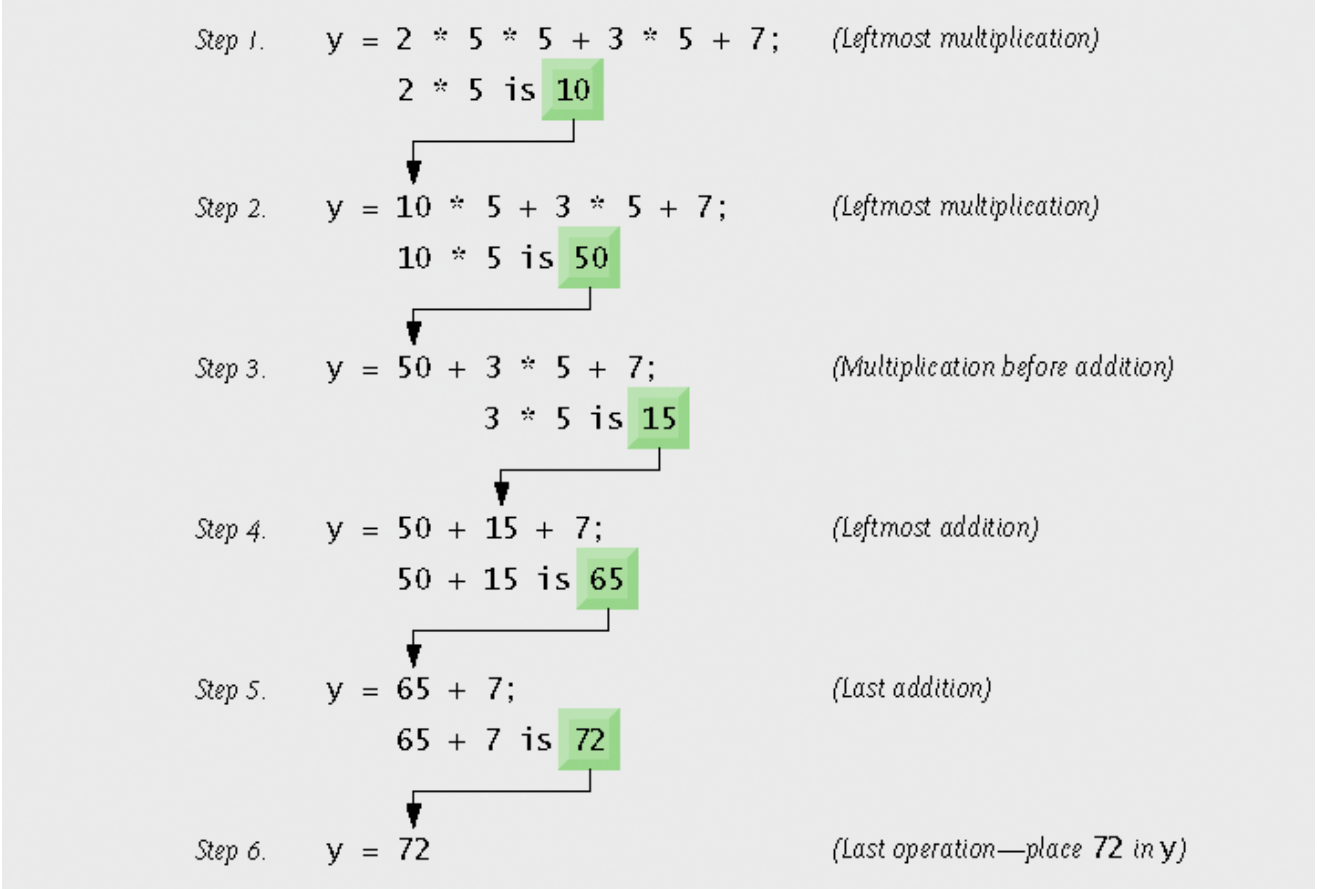

Fig. 2.11 | Order in which a second-degree polynomial is evaluated.

2.7 Decision Making: Equality and Relational Operators

- **Condition**
  - **Expression can be either `true` or `false`**
  - **Can be formed using equality or relational operators**
- **`if` statement**
  - **If condition is `true`, body of the `if` statement executes**
  - **If condition is `false`, body of the `if` statement does not execute**

Standard algebraic equality or relational operator	C++ equality or relational operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. 2.12 | Equality and relational operators.

Common Programming Error 2.5

A syntax error will occur if any of the operators ==, !=, >= and <= appears with spaces between its pair of symbols.

Common Programming Error 2.6

Reversing the order of the pair of symbols in any of the operators !=, >= and <= (by writing them as =!, => and =<, respectively) is normally a syntax error. In some cases, writing != as =! will not be a syntax error, but almost certainly will be a logic error that has an effect at execution time. (cont…)

Common Programming Error 2.6

You will understand why when you learn about logical operators in Chapter 5. A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but usually produces incorrect results.

Common Programming Error 2.7

Confusing the equality operator == with the assignment operator = results in logic errors. The equality operator should be read "is equal to," and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." As we discuss in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.

Outline

fig02_13.cpp

(1 of 2)

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // allows program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin;  // program uses cin
8  using std::endl; // program uses endl
9
10 // function main begins program e
11 int main()
12 {
13    int number1; // first
14    int number2;// second
15
16    cout << "Enter two integers to compare: ";
17    cin >> number1 >> number2; // read two integ
18
19    if ( number1 == number2 )
20       cout << number1 << " == " << number2 << e
21
22    if ( number1 != number2 )
23       cout << number1 << " != " << number2 << endl;
24
25    if ( number1 < number2 )
26       cout << number1 << " < " << number2 << en
27
28    if ( number1 > number2 )
29       cout << number1 << " > " << number2 << endl;
30
```

using declarations eliminate need for **std::** prefix

Declare variables

Can write **cout** and **cin** without **std::** prefix

if statement compar of **number1** and **nu**

If condition is **true** (i.e., values are equal), execute

if statement compares values of **number1** and **nu** test for inequality

If condition is **true** (i.e., values are not equal), execute this statement

Compares two numbers using relational operator **<** and **>**

```
31    if ( number1 <= number2 )
32       cout << number1 << " <= " << number2 << endl;
33
34    if ( number1 >= number2 )
35       cout << number1 << " >= " << number2 << endl;
36
37    return 0; // indicate that program ended successfully
38
39 } // end function main
```

Compares two numbers using relational operators **<=** and **>=**

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Outline

fig02_13.cpp

(2 of 2)

fig02_13.cpp output (1 of 3)

(2 of 3)

(3 of 3)

Good Programming Practice 2.15

Place `using` declarations immediately after the `#include` to which they refer.

Good Programming Practice 2.16

Indent the statement(s) in the body of an `if` statement to enhance readability.

Good Programming Practice 2.17

For readability, there should be no more than one statement per line in a program.

Common Programming Error 2.8

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now would become a statement in sequence with the `if` statement and would always execute, often causing the program to produce incorrect results.

Common Programming Error 2.9

It is a syntax error to split an identifier by inserting white-space characters (e.g., writing `main` as `ma in`).

Good Programming Practice 2.18

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group.

Operators				Associativity	Type
()				left to right	parentheses
*	/	%		left to right	multiplicative
+	-			left to right	additive
<<	>>			left to right	stream insertion/extraction
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Good Programming Practice 2.19

Refer to the operator precedence and associativity chart when writing expressions containing many opera-tors. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you would do in an algebraic expression. Be sure to ob-serve that some operators such as assignment (=) associate right to left rather than left to right.

2.8 (Optional) Software Engineering Case Study: Examining the ATM Requirements Document

- **Object-oriented design (OOD) process using UML**
  - **Performed in chapters 3 to 7, 9 and 13**
  - **Requirements document**
    - **Specifies overall purpose and what the system must do**
- **Object-oriented programming (OOP) implementation**
  - **Complete implementation in appendix G**

2.8 (Optional) Software Engineering Case Study (Cont.)

- **Requirements document**
  - **New automated teller machine (ATM)**
  - **Allows basic financial transaction**
    - **View balance, withdraw cash, deposit funds**
  - **User interface**
    - **Display screen, keypad, cash dispenser, deposit slot**
  - **ATM session**
    - **Authenticate user, execute financial transaction**

Fig. 2.15 | Automated teller machine user interface.

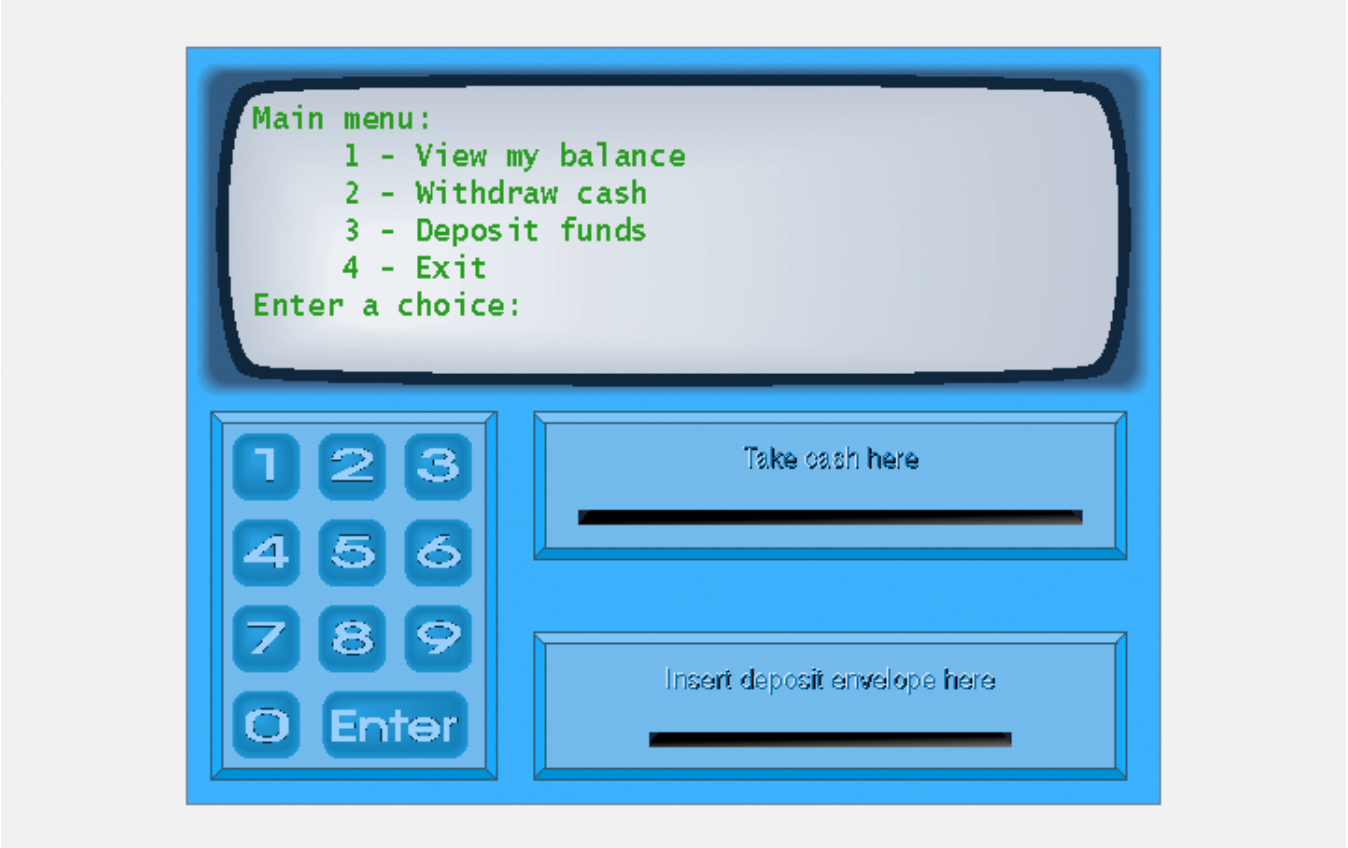

Fig. 2.16 | ATM main menu.

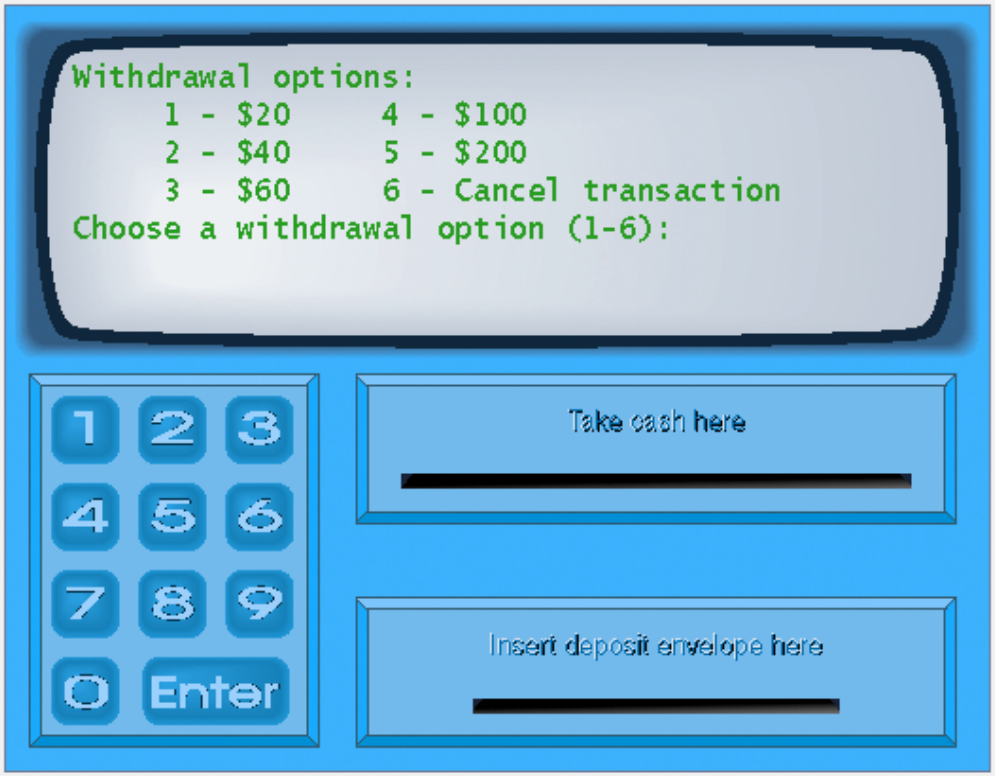

Fig. 2.17 | ATM withdrawal menu.

2.8 (Optional) Software Engineering Case Study (Cont.)

- **Analyzing the ATM system**
  - **Requirements gathering**
  - **Software life cycle**
    - **Waterfall model**
    - **Iteractive model**
  - **Use case modeling**
- **Use case diagram**
  - **Model the interactions between clients and its use cases**
  - **Actor**
    - **External entity**

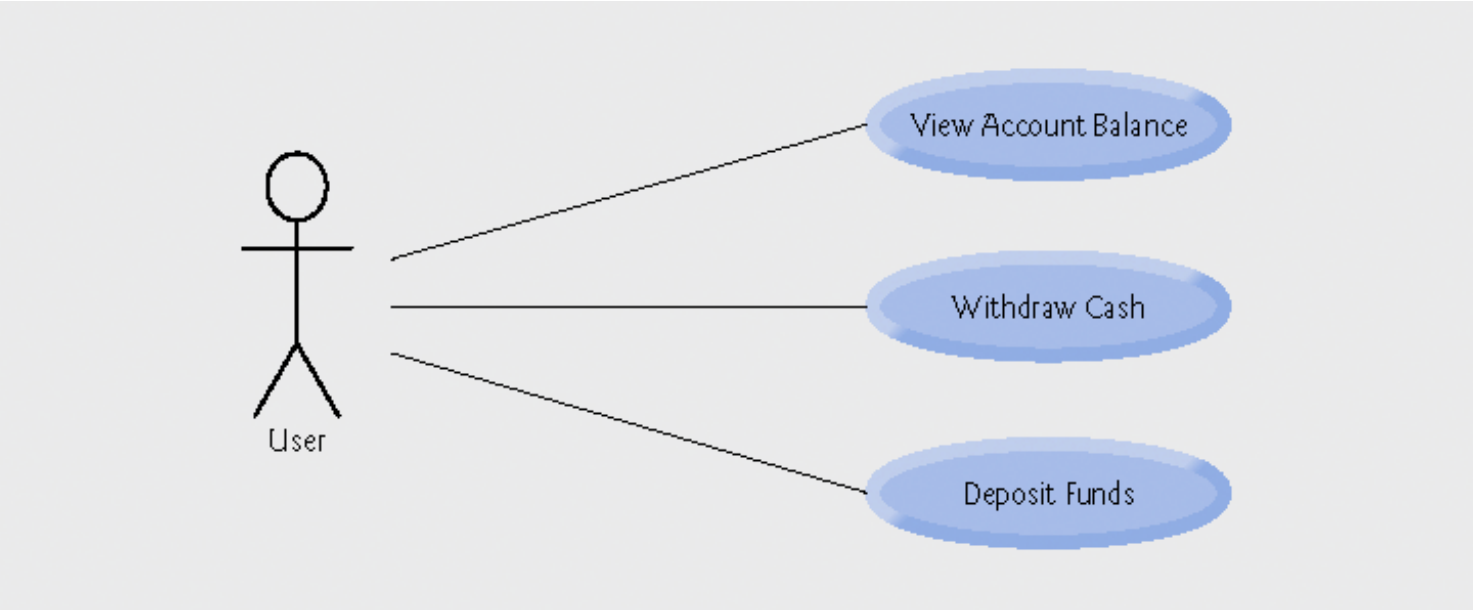

Fig. 2.18 | Use case diagram for the ATM system from the user's perspective.

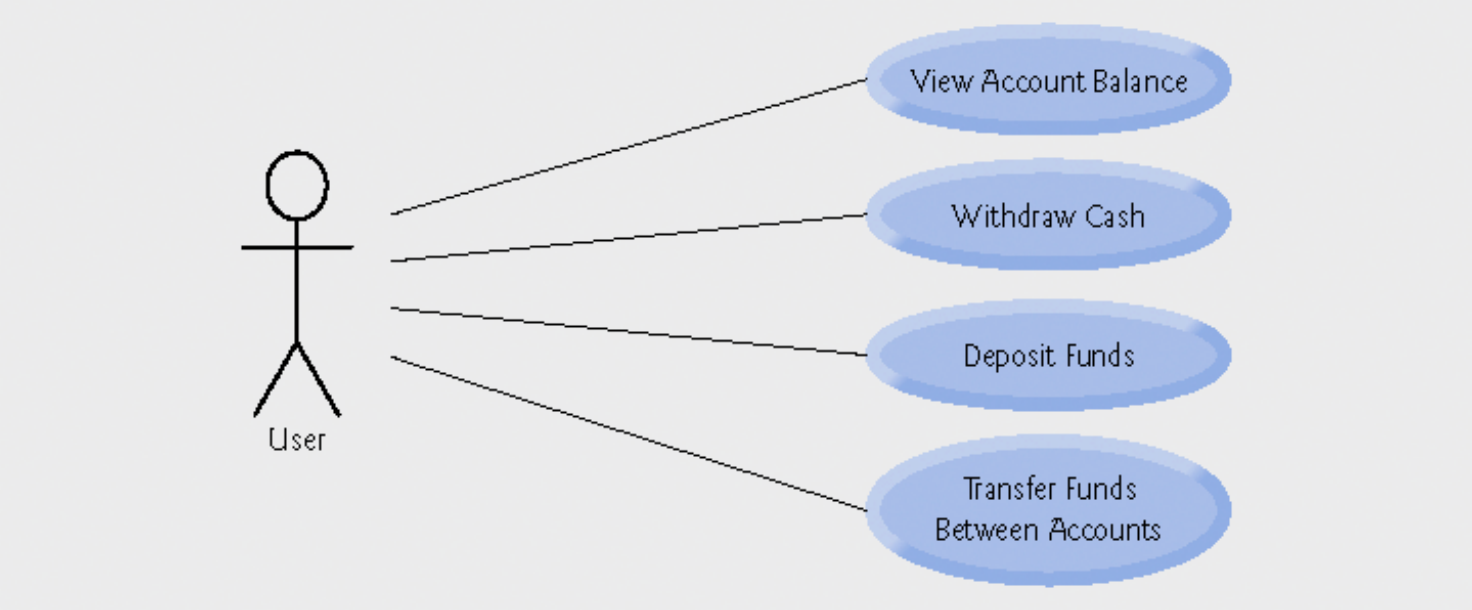

Fig. 2.19 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

2.8 (Optional) Software Engineering Case Study (Cont.)

- **UML diagram types**
  - **Model system structure**
    - **Class diagram**
      - **Models classes, or “building blocks” of a system**
      - **Screen, keypad, cash dispenser, deposit slot.**

2.8 (Optional) Software Engineering Case Study (Cont.)

- **Model system behavior**
  - **Use case diagrams**
    - **Model interactions between user and the system**
  - **State machine diagrams**
    - **Model the ways in which an object changes state**
  - **Activity diagrams**
    - **Model an object's activity during program execution**
  - **Communication diagrams (collaboration diagrams)**
    - **Model the interactions among objects**
    - **Emphasize what interactions occur**
  - **Sequence diagrams**
    - **Model the interactions among objects**
    - **Emphasize when interactions occur**