Chapter 3
Assignment, Formatting, and Interactive Input

Objectives

In this chapter, you will learn about:

- Assignment operations
- Formatting numbers for program output
- Using mathematical library functions
- Program input using the **`cin`** object
- Symbolic constants
- A case study involving acid rain
- Common programming errors

Assignment Operations

- **Assignment Statement:** Assigns the value of the expression on the right side of the = to the variable on the left side of the =
- Another assignment statement using the same variable will overwrite the previous value with the new value

  Examples:

```
slope = 3.7;
slope = 6.28;
```

Assignment Operations (continued)

- Right side of an assignment statement may contain any expression that can be evaluated to a value

  Examples:

```
newtotal = 18.3 + total;
taxes = .06*amount;
average = sum / items;
```

- Only one variable can be on the left side of an assignment statement

Assignment Operations (continued)

Program 3.1

```
// this program calculates the volume of a cylinder,
// given its radius and height
#include <iostream>
using namespace std;

int main()
{
  double radius, height, volume;

  radius = 2.5;
  height = 16.0;
  volume = 3.1416 * radius * radius * height;
  cout << "The volume of the cylinder is " << volume << endl;

  return 0;
}
```

Assignment Operations (continued)

- **Assignment operator:** The = sign
- **C++ statement:** Any expression terminated by a semicolon
- Multiple assignments in the same expression are possible

  Example:

```
a = b = c = 25;
```

Assignment Operations (continued)

- **Coercion:** Forcing a data value to another data type
- Value of the expression on the right side of an assignment statement will be coerced (converted) to the data type of the variable on the left side during evaluation
- Variable on the left side may also be used on the right side of an assignment statement

Assignment Operations (continued)

Program 3.2

```
#include <iostream>
using namespace std;

int main()
{
  int sum;

  sum = 25;
  cout << "The number stored in sum is " << sum << endl;
  sum = sum + 10;
  cout << "The number now stored in sum is " << sum << endl;

  return 0;
}
```

Assignment Operations (continued)

- **Accumulation statement:** Has the effect of accumulating, or totaling

  Syntax:

```
variable = variable + newValue;
```

Assignment Operations (continued)

- Additional assignment operators provide short cuts:
  +=, **-=**, ***=**, **/=**, **%=**

  Example:

```
sum = sum + 10;
```

  is equivalent to: **sum += 10;**

```
price *= rate +1;
```

  is equivalent to:

```
price = price * (rate + 1);
```

Assignment Operations (continued)

Program 3.3

```
#include <iostream>
using namespace std;

int main()
{
  int sum;

  sum = 0;
  cout << "The value of sum is initially set to " << sum << endl;
  sum = sum + 96;
  cout << "  sum is now " << sum << endl;
  sum = sum + 70;
  cout << "  sum is now " << sum << endl;
  sum = sum + 85;
  cout << "  sum is now " << sum  << endl;
  sum = sum + 60;
  cout << "  The final sum is " << sum << endl;

  return 0;
}
```

Assignment Operations (continued)

- **Counting statement:** Adds a fixed value to the variable's current value

  Syntax:

```
variable = variable + fixedNumber;
```

  Example:

```
i = i + 1;
count = count + 1;
```

Assignment Operations (continued)

- **Increment operator ++:** Unary operator for the special case when a variable is increased by 1
- **Prefix increment operator** appears before the variable
  - —Example: `++i`
- **Postfix increment operator** appears after the variable
  - —Example: `i++`

Assignment Operations (continued)

- Example: `k = ++n; //prefix increment`

  is equivalent to:

```
n = n + 1;  //increment n first
k = n;      //assign n's value to k
```

- Example: `k = n++; //postfix increment`

  is equivalent to

```
k = n;      //assign n's value to k
n = n + 1;  //and then increment n
```

Assignment Operations (continued)

- **Decrement operator `--`:** Unary operator for the special case when a variable is decreased by 1
- **Prefix decrement operator** appears before the variable
  - —Example: `--i;`
- **Postfix decrement operator** appears after the variable
  - —Example: `i--;`

Formatting Numbers for Program Output

- Proper output formatting contributes to ease of use and user satisfaction
- `cout` with stream manipulators can control output formatting

Formatting Numbers for Program Output

Manipulator	Action
`setw(n)`	Set the field width to `n`.
`setprecision(n)`	Set the floating-point precision to `n` places. If the `fixed` manipulator is designated, `n` specifies the total number of displayed digits after the decimal point; otherwise, `n` specifies the total number of significant digits displayed (integer plus fractional digits).
`setfill('x')`	Set the default leading fill character to `x`. (The default leading fill character is a space, which is used to fill the beginning of an output field when the field width is larger than the value being displayed.)
`setiosflags (flags)`	Set the format flags. (See Table 3.3 for flag settings.)
`scientific`	Set the output to display real numbers in scientific notation.
`showbase`	Display the base used for numbers. A leading `0` is displayed for octal numbers and a leading `0x` for hexadecimal numbers.
`showpoint`	Always display six digits total (combination of integer and fractional parts). Fill with trailing zeros, if necessary. For larger integer values, revert to scientific notation.
`showpos`	Display all positive numbers with a leading `+` sign.
`boolalpha`	Display Boolean values as `true` and `false` rather than `1` and `0`.
`dec`	Set the output for decimal display, which is the default.
`endl`	Output a newline character and display all characters in the buffer.
`fixed`	Always show a decimal point and use a default of six digits after the decimal point. Fill with trailing zeros, if necessary.

Table 3.1 Commonly Used Stream Manipulators

Formatting Numbers for Program Output

Manipulator	Action
`flush`	Display all characters in the buffer.
`left`	Left-justify all numbers.
`hex`	Set the output for hexadecimal display.
`oct`	Set the output for octal display.
`uppercase`	Display hexadecimal digits and the exponent in scientific notation in uppercase.
`right`	Right-justify all numbers (the default).
`noboolalpha`	Display Boolean values as `1` and `0` rather than `true` and `false`.
`noshowbase`	Don't display octal numbers with a leading `0` and hexadecimal numbers with a leading `0x`.

Table 3.1 Commonly Used Stream Manipulators (continued)

Formatting Numbers for Program Output

Program 3.6

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

Formatting Numbers for Program Output

- The field width manipulator must be included for each value in the data stream sent to **`cout`**
- Other manipulators remain in effect until they are changed
- **`iomanip`** header file must be included to use manipulators requiring arguments

Formatting Numbers for Program Output

- Formatting floating-point numbers requires three field-width manipulators to:
  - —Set the total width of the display
  - —Force a decimal place
  - —Set the number of significant digits after the decimal point
- Example:

```
cout << "|" << setw(10) << fixed
     << setprecision(3) << 25.67 << "|";
```

produces this output: `|  25.670|`

Formatting Numbers for Program Output

- **`setprecision`:** Sets number of digits after decimal point if a decimal point has been explicitly forced; otherwise, it sets the total number of displayed digits
- If the field width is too small, **`cout`** ignores the **`setw`** manipulator setting and allocates enough space for printing
- If **`setprecision`** setting is too small, the fractional part of the value is rounded to the specified number of decimal places

Formatting Numbers for Program Output(continued)

- If **`setprecision`** value is too large, the fractional value is displayed with its current size

Formatting Numbers for Program Output(continued)

Manipulators	Number	Display	Comments
`setw(2)`	3	| 3|	Number fits in the field.
`setw(2)`	43	|43|	Number fits in the field.
`setw(2)`	143	|143|	Field width is ignored.
`setw(2)`	2.3	|2.3|	Field width is ignored.
`setw(5) fixed setprecision(2)`	2.366	| 2.37|	Field width of five with two decimal digits.
`setw(5) fixed setprecision(2)`	42.3	|42.30|	Number fits in the field with the specified precision. Note that the decimal point takes up one location in the field width.
`setw(5) setprecision(2)`	142.364	|1.4e+002|	Field width is ignored, and scientific notation is used with the `setprecision` manipulator.

Table 3.2 Effect of Format Manipulators

Formatting Numbers for Program Output(continued)

Manipulators	Number	Display	Comments
`setw(5) fixed setprecision(2)`	142.364	|142.36|	Field width is ignored, but precision specification is used. The `setprecision` manipulator specifies the number of fractional digits.
`setw(5) fixed setprecision(2)`	142.366	|142.37|	Field width is ignored, but precision specification used. The `setprecision` manipulator specifies the number of fractional digits. (Note the rounding of the last decimal digit.)
`setw(5) fixed setprecision(2)`	142	| 142|	Field width is used; `fixed` and `setprecision` manipulators are irrelevant because the number is an integer that specifies the total number of significant digits (integer plus fractional digits).

Table 3.2 Effect of Format Manipulators (continued)

Formatting Numbers for Program Output(continued)

- **`setiosflags`** manipulator: Allows additional formatting:
  - Right or left justification
  - Fixed display with 6 decimal places
  - Scientific notation with exponential display
  - Display of a leading + sign
- **Parameterized manipulator:** One which requires arguments, or parameters

Formatting Numbers for Program Output(continued))

Flag	Meaning
`ios::fixed`	Always show the decimal point with six digits after the decimal point. Fill with trailing zeros after the decimal point, if necessary. This flag takes precedence if it's set with the `ios::showpoint` flag.
`ios::scientific`	Use exponential display in the output.
`ios::showpoint`	Always display a decimal point and six significant digits total (combination of integer and fractional parts). Fill with trailing zeros after the decimal point, if necessary. For larger integer values, revert to scientific notation unless the `ios::fixed` flag is set.
`ios::showpos`	Display a leading + sign when the number is positive.
`ios::left`	Left-justify the output.
`ios::right`	Right-justify the output.

Table 3.3 Format Flags for Use with **`setiosflags()`**

Formatting Numbers for Program Output(continued)

Program 3.7

```
// a program that illustrates output conversions
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "The decimal (base 10) value of 15 is " << 15 << endl;
  cout << "The octal (base 8) value of 15 is "
       << showbase << oct << 15 <<endl;
  cout << "The hexadecimal (base 16) value of 15 is "
         << showbase << hex << 15 << endl;

  return 0;
}
```

Formatting Numbers for Program Output(continued)

- To designate an octal integer constant, use a leading zero
- To designate a hexadecimal integer constant, use a leading 0x
- Manipulators affect only output; the value stored internally does not change

Formatting Numbers for Program Output(continued)

Program 3.8

```
#include <iostream>
using namespace std;

int main()
{
  cout << "The decimal value of 025 is " << 025 << endl
       << "The decimal value of 0x37 is "<< 0x37 << endl;

  return 0;
}
```

Formatting Numbers for Program Output(continued))

- Manipulators can also be set using the **`ostream`** class methods
- Separate the **`cout`** object name from the method name with a period

  Example:

```
cout.precision(2)
```

Formatting Numbers for Program Output(continued)

Method	Comment	Example
`precision(n)`	Equivalent to `setprecision()`	`cout.precision(2)`
`fill('x')`	Equivalent to `setfill()`	`cout.fill('*')`
`setf(ios::fixed)`	Equivalent to `cout.setf(ios::fixed)`	`setiosflags(ios::fixed)`
`setf(ios::showpoint)`	Equivalent to `cout.setf(ios::showpoint)`	`setiosflags(ios::showpoint)`
`setf(iof::left)`	Equivalent to `left`	`cout.setf(ios::left)`
`setf(ios::right)`	Equivalent to `right`	`cout.setf(ios::right)`
`setf(ios::flush)`	Equivalent to `endl`	`cout.setf(ios::flush)`

Table 3.4 `ostream` Class Functions

Using Mathematical Library Functions

- C++ has preprogrammed mathematical functions that can be included in a program
- You must include the **`cmath`** header file:

```
#include <cmath>
```

- Math functions require one or more arguments as input, but will return only one value
- All functions are overloaded, and can be used with integer and real arguments

Using Mathematical Library Functions

Function Name	Description	Returned Value
`abs(a)`	absolute value	Same data type as argument
`pow(a1,a2)`	`a1` raised to the `a2` power	Same data type as argument `a1`
`sqrt(a)`	square root of a real number	Double-precision
`sin(a)`	sine of `a` (`a` in radians)	Double
`cos(a)`	cosine of `a` (`a` in radians)	Double
`tan(a)`	tangent of `a` (`a` in radians)	Double
`log(a)`	natural logarithm of `a`	Double
`log10(a)`	common log (base 10) of `a`	Double
`exp(a)`	`e` raised to the `a` power	Double

Table 3.5 Common C++ Functions

Using Mathematical Library Functions

- To use a math function, give its name and pass the input arguments within parentheses
- Expressions that can be evaluated to a value can be passed as arguments

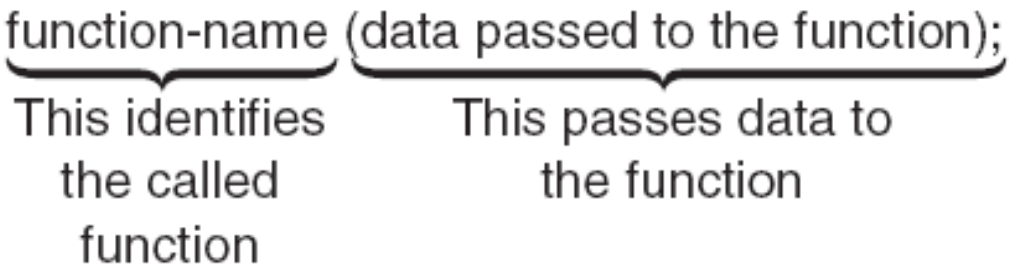

Figure 3.10 Using and passing data to a function

Using Mathematical Library Functions

Program 3.9

```
#include <iostream>  // this line can be placed second instead of first
#include <cmath>     // this line can be placed first instead of second
using namespace std;

int main()
{
  int height;
  double time;

  height = 800;
  time = sqrt(2 * height / 32.2);
  cout << "It will take " << time << " seconds to fall "
       << height << " feet.\n";

  return 0;
}
```

Using Mathematical Library Functions

- Function calls can be nested
  - Example: **`sqrt(sin(abs(theta)))`**
- **Cast operator:** A unary operator that forces the data to the desired data type
- Compile-time cast
  - Syntax: **`dataType (expression)`**
  - Example: **`int(a+b)`**

Using Mathematical Library Functions

- **Run-time cast:** The requested conversion is checked at run time and applied if valid
  - —Syntax:

```
staticCast<data-type> (expression)
```

  - —Example:

```
staticCast<int>(a*b)
```

Program Input Using `cin`

- **`cin` Object:** Allows data entry to a running program
- Use of the **`cin`** object causes the program to wait for input from the keyboard
- When keyboard entry is complete, the program resumes execution, using the entered data
- An output statement preceding the **`cin`** object statement provides a **prompt** to the user

Program Input Using `cin` (continued)

Program 3.12

```
#include <iostream>
using namespace std;

int main()
{
  double num1, num2, product;

  cout << "Please type in a number: ";
  cin  >> num1;
  cout << "Please type in another number: ";
  cin  >> num2;
  product = num1 * num2;
  cout << num1 << " times " << num2 << " is " << product << endl;

  return 0;
}
```

Program Input Using `cin` (continued)

- **`cin`** can accept multiple input values to be stored in different variables
- Multiple numeric input values must be separated by spaces

  Example:

  ```
  cin >> num1 >> num2
  ```

  with keyboard entry: **`0.052 245.79`**

Program Input Using `cin` (continued)

Program 3.13

```
#include <iostream>
using namespace std;

int main()
{
  int num1, num2, num3;
  double average;

  cout << "Enter three integer numbers: ";
  cin >> num1 >> num2 >> num3;
  average =  (num1 + num2 + num3) / 3.0;
  cout << "The average of the numbers is " << average << endl;

  return 0;
}
```

Program Input Using `cin` (continued)

- **User-input validation:** The process of ensuring that data entered by the user matches the expected data type
- **Robust program:** One that detects and handles incorrect user entry

Symbolic Constants

- **Symbolic constant:** Constant value that is declared with an identifier using the **`const`** keyword
- A constant's value may not be changed

  Example:

```
const int MAXNUM = 100;
```

- Good programming places statements in appropriate order

Symbolic Constants (continued)

- Proper placement of statements:

```
preprocessor directives
int main()
{
      //symbolic constants
      //main function declarations

      //other executable statements
      return value
}
```

A Case Study: Acid Rain

- **Acid Rain:** Develop a program to calculate the pH level of a substance based on user input of the concentration of hydronium ions
  - Step 1: Analyze the Problem
  - Step 2: Develop a Solution
  - Step 3: Code the Solution
  - Step 4: est and Correct the Program

A Closer Look: Programming Errors

- Program errors may be detected in four ways:
  - Before a program is compiled (desk checking)
  - While it is being compiled (compile-time errors)
  - While it is being run (run-time errors)
  - While examining the output after completion
- Errors may be:
  - Typos in the source code
  - Logic errors

A Closer Look: Programming Errors (continued)

- Logic errors: Often difficult to detect and difficult to find the source
- **Program tracing:** Stepping through the program by hand or with a trace tool
- **Debugger:** Program that allows the interruption of a running program to determine values of its variables at any point

Common Programming Errors

- Failure to declare or initialize variables before use
- Failure to include the preprocessor statement when using a C++ preprogrammed library
- Passing the incorrect number or type of arguments to a function
- Applying increment or decrement operator to an expression instead of an individual variable

Common Programming Errors (continued)

- Failure to separate all variables passed to **`cin`** with the extraction symbol >>
- Failure to test thoroughly
- Compiler-dependent evaluation when increment or decrement operators are used with variables that appear more than once in the same expression

Summary

- Expression: A sequence of one or more operands separated by operators
- Expressions are evaluated based on precedence and associativity
- Assignment operator: =
- Increment operator: ++
- Decrement operator: --

Summary (continued)

- Use **`#include <cmath>`** for math functions
- Arguments to a function must be passed in the proper number, type, and order
- Functions may be included within larger expressions
- **`cin`** object provides data input from a keyboard; program is suspended until the input arrives
- Use a prompt to alert the user to provide input
- Constants are named values that do not change