CHAPTER

2

Primitive Data Types and Operations

Objectives

- To write C++ programs to perform simple calculations (§2.2).
- To read input from the keyboard using the cin object (§2.3).
- To simplify programming by omitting the std:: prefix (§2.4).
- To use identifiers to name variables, constants, functions, and classes (§2.5).
- To use variables to store data (§§2.6-2.7).
- To program with assignment statements and assignment expressions (§2.7).
- To use constants to store permanent data (§2.8).
- To declare variables using numeric data types (§2.9).
- To use operators to write numeric expressions (§2.9).
- To convert numbers to a different type using casting (§2.10).
- To represent character using the char type (§2.11).
- To become familiar with C++ documentation, programming style, and naming conventions (§2.13).
- To distinguish syntax errors, runtime errors, and logic errors (§2.14).
- To debug logic errors (§2.15).
2.1 Introduction
In the preceding chapter, you learned how to create, compile, and run a C++ program. In this chapter, you will be introduced to C++ primitive data types and related subjects, such as variables, constants, data types, operators, and expressions. You will learn how to write programs using primitive data types, input and output, and simple calculations.

2.2 Writing Simple Programs
To begin, let’s look at a simple program that computes the area of a circle. The program reads in the radius of the circle and displays its area. The program will use variables to store the radius and the area, and will use an expression to compute the area.

<Side Remark: algorithm>
Writing this program involves designing algorithms and data structures, as well as translating algorithms into programming codes. An algorithm describes how a problem is solved in terms of the actions to be executed, and it specifies the order in which the actions should be executed. Algorithms can help the programmer plan a program before writing it in a programming language. The algorithm for this program can be described as follows:

1. Read in the radius.
2. Compute the area using the following formula:
   \[ \text{area} = \text{radius} \times \text{radius} \times \pi \]
3. Display the area.

Many of the problems you will meet when taking an introductory course in programming using this text can be described with simple, straightforward algorithms. As your education progresses, and you take courses on data structures or on algorithm design and analysis, you will encounter complex problems that require sophisticated solutions. You will need to design accurate, efficient algorithms with appropriate data structures in order to solve such problems.

<Side Remark: primitive data types>
Data structures involve data representation and manipulation. C++ provides data types for representing integers, floating-point numbers (i.e., numbers with a decimal point), characters, and Boolean types. These types are known as primitive data types, or fundamental types. C++ also supports array and some advanced data structures, such as strings and vectors.

To novice programmers, coding is a daunting task. When you code, you translate an algorithm into a programming language understood by the computer. You already know that every C++
program begins its execution from the main function. The outline of the main function would look like this:

```c
int main() {
    // Step 1: Read in radius
    // Step 2: Compute area
    // Step 3: Display the area
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

**<Side Remark: variable>**

Let’s address the second issue first. In order to store the radius, the program needs to declare a symbol called a variable that will represent the radius. Variables are used to store data and computational results in the program.

**<Side Remark: descriptive names>**

Rather than using `x` and `y`, choose descriptive names: in this case, `radius` for radius, and `area` for area. Specify their data types to let the compiler know what `radius` and `area` are, indicating whether they are integer, floating-point number, or something else. Declare `radius` and `area` as double-precision floating-point numbers. The program can be expanded as follows:

```c
int main() {
    double radius;
    double area;

    // Step 1: Read in radius
    // Step 2: Compute area
    // Step 3: Display the area
}
```

The program declares `radius` and `area` as variables. The reserved word `double` indicates that `radius` and `area` are double-precision floating-point values stored in the computer.

The first step is to read in `radius`. You will learn how to read a number from the keyboard later. For the time being, let us assign a fixed number to `radius` in the program.

The second step is to compute `area` by assigning the expression `radius * radius * 3.14159` to `area`.

In the final step, print `area` on the console by directing it to the `std::cout` object.

The complete program is shown in Listing 2.1.
Listing 2.1 ComputeArea.cpp (Computing the Area for a Circle)

```cpp
#include <iostream>

int main() {
    double radius;
    double area;

    // Step 1: Read in radius
    radius = 20;

    // Step 2: Compute area
    area = radius * radius * 3.14159;

    // Step 3: Display the area
    std::cout << "The area is ";
    std::cout << area << std::endl;
}
```

<output>
The area is 1256.62
<end of output>

**Side Remark 1: declaring variable**

**Side Remark 2: assign value**

Variables such as radius and area correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 4 declares that `radius` can store a `double` value. The value is not defined until you assign a value. Line 8 assigns 20 into `radius`. Similarly, line 5 declares variable `area` and line 11 assigns a value into `area`. If you comment line 8, the program would compile and run, but the result is unpredictable, because radius may be assigned any value.

Line 14 sends a string "The area is " to the console. Line 15 sends the value in variable `area` to the console. Note that there are no quotation marks around `area`. If quotation marks were placed around `area`, it would send the string literal "area" to the console.

**TIP**

**side remark: incremental development and testing**

This example consists of three steps. It is a good approach to develop and test these steps incrementally by adding one step at a time. You should apply this approach to all the programs, although the problem solving steps are not explicitly stated in many programs in this book.

**TIP**

**side remark: concatenating output**

You can combine multiple outputs in one statement. For example, the following one statement performs the same function as lines 14-15.

```cpp
std::cout << "The area is " << area << std::endl;
```
When executing this statement, the string "The area is" is sent to the output object first, then the value of variable area, and finally std::endl outputs a new line.

***End TIP

2.3 Reading Input from the Keyboard
In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, it is not convenient. You can use the std::cin object to read input from the keyboard, as shown in Listing 2.2.

Listing 2.2 ComputeArea1.cpp (Using Console Input)

```cpp
#include <iostream>

int main() {
    // Step 1: Read in radius
    double radius;
    std::cout << "Enter a radius: ";
    std::cin >> radius;

    // Step 2: Compute area
    double area = radius * radius * 3.14159;

    // Step 3: Display the area
    std::cout << "The area is " << area << std::endl;
}
```

Enter a radius: 2
The area is 12.5664

Line 6 displays a string "Enter a radius: " to the console. This is known as a prompt, because it directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard.

Line 7 uses the std::cin object to read a value from the keyboard. The >> symbol, referred to as the stream extraction operator, assigns an input to a variable. As shown in the sample output, the program displays the prompting message "Enter a radius: ", the user then enters number 2, which is assigned to variable radius. The cin object causes a program to wait until data is typed at the keyboard and the Enter key is pressed. C++
automatically converts the data read from the keyboard to the data type of the variable.

Note that the >> symbol is the opposite of the << symbol. The >> indicates that the data flows from cin to a variable. The << symbol shows that data flows from a variable or a string to cout.

.Side Remark: multiple input
You can use one statement to read multiple values. For example, the following statement reads three values into variable x1, x2, and x3:

```cpp
std::cin >> x1 >> x2 >> x3;
```

You need to enter three numbers separated by spaces and presses the Enter key to end the input. For example, if you enter

```
10 20 30
```

then press the Enter key, x1, x2, and x3 become 10, 20, and 30, respectively.

2.4 Omitting the std:: Prefix
.Side Remark: identifier
You have noticed that std::cout, std::endl, and std::cin all start with std::. So what is std? std means the standard namespace. C++ divides the world into “namespaces” to resolve potential naming conflicts. std::cout means that cout belongs to the standard namespace. It is tedious to type std:: repeatedly. There are two solutions to eliminate the std:: prefix.

The first solution is to add the statement:

```
using namespace std;
```

This tells the compiler that any object without an explicit qualifier belongs to the standard namespace. Listing 2.3 uses this statement to rewrite Listing 2.2. The std:: prefix for cout, cin, and endl are omitted.

Listing 2.3 ComputeArea2.cpp (Using Namespace)

```cpp
***PD: Please add line numbers (including space lines) in the following code***
.Side Remark line 2: standard namespace
#include <iostream>
using namespace std;

int main() {
    // Step 1: Read in radius
    double radius;
    cout << "Enter a radius: ",
    cin >> radius;
}
```
// Step 2: Compute area
  double area = radius * radius * 3.14159;

// Step 3: Display the area
  cout << "The area is " << area << endl;

The second solution is to add the statements:

using std::cout;
using std::cin;
using std::endl;

So the compiler knows that cout, cin, and endl are in the standard namespace before they are referenced. Listing 2.4 uses these statements to rewrite Listing 2.2.

Listing 2.4 ComputeArea3.cpp (Using Namespace)

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main() {
  // Step 1: Read in radius
  double radius;
  cout << "Enter a radius: ";
  cin >> radius;

  // Step 2: Compute area
  double area = radius * radius * 3.14159;

  // Step 3: Display the area
  cout << "The area is " << area << endl;
}
```

2.5 Identifiers

Just as every entity in the real world has a name, so you need to choose names for the things you will refer to in your programs. Programming languages use special symbols called identifiers to name such programming entities as variables, constants and functions. Here are the rules for naming identifiers:

- An identifier is a sequence of characters that consists of letters, digits, and underscores (_).
- An identifier must start with a letter or an underscore. It cannot start with a digit.
An identifier cannot be a reserved word. (See Appendix A, “C++ Keywords,” for a list of reserved words.)

An identifier can be of any length, but your C++ compiler may impose some restriction. Use identifiers of 31 characters or fewer to ensure portability.

For example, area and radius are legal identifiers, whereas 2A and d+4 are illegal identifiers because they do not follow the rules. The compiler detects illegal identifiers and reports syntax errors.

**NOTE**

*<side remark: case-sensitive>*

Since C++ is case-sensitive, X and x are different identifiers.

**TIP**

*<side remark: descriptive names>*

Identifiers are used for naming variables and functions and other things in a program. Descriptive identifiers make programs easy to read. Besides choosing descriptive names for identifiers, there are naming conventions for different kinds of identifiers. Naming conventions are summarized in §2.15, “Programming Style and Documentation.”

**TIP**

*<side remark: the _ character>*

Do not name identifiers that begin with underscores to avoid confusion, because C++ compilers may use names like that internally.

### 2.6 Variables

Variables are used to store data in a program. In the program in Listing 2.4, radius and area are variables of double-precision, floating-point type. You can assign any numerical value to radius and area, and the values of radius and area can be reassigned. For example, you can write the code shown below to compute the area for different radii:

```cpp
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
std::cout << area;

// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
std::cout << area;
```

#### 2.6.1 Declaring Variables

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler the name of the
variable as well as what type of data it represents. This is called a variable declaration. Declaring a variable tells the compiler to allocate appropriate memory space for the variable based on its data type. Here is the syntax for declaring a variable:

```
<Side Remark: declaring variable>
datatype variableName;
```

Here are some examples of variable declarations:

```
int x;               // Declare x to be an integer variable;
double radius;       // Declare radius to be a double variable;
double interestRate; // Declare interestRate to be a double variable;
char a;              // Declare a to be a character variable;
```

The examples use the data types int, double, and char. Later in this chapter you will be introduced to additional data types, such as byte, short, long, float, and char.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```

**NOTE**

<side remark: naming variables>

By convention, variable names are in lowercase. If a name consists of several words, concatenate all of them and capitalize the first letter of each word except the first. Examples of variables are radius and interestRate.

### 2.7 Assignment Statements and Assignment Expressions

**<Side Remark: assignment statement>**

**<Side Remark: assignment operator>**

After a variable is declared, you can assign a value to it by using an assignment statement. In C++, the equal sign (=) is used as the assignment operator. The syntax for assignment statements is as follows:

```
variable = expression;
```

**<Side Remark: expression>**

An expression represents a computation involving values, variables, and operators that together evaluates to a value. For example, consider the following code:

```
int x = 1;                // Assign 1 to variable x;
double radius = 1.0;       // Assign 1.0 to variable radius;
x = 5 * (3 / 2) + 3 * 2;  // Assign the value of the expression to x;
x = y + 1;                // Assign the addition of y and 1 to x;
area = radius * radius * 3.14159; // Compute area
```
A variable can also be used in an expression. For example,

\[ x = x + 1; \]

In this assignment statement, the result of \( x + 1 \) is assigned to \( x \). If \( x \) is 1 before the statement is executed, then it becomes 2 after the statement is executed.

To assign a value to a variable, the variable name must be on the left of the assignment operator. Thus, \( 1 = x \) would be wrong.

<Side Remark: assignment expression>

In C++, an assignment statement can also be treated as an expression that evaluates to the value being assigned to the variable on the left-hand side of the assignment operator. For this reason, an assignment statement is also known as an assignment expression. For example, the following statement is correct:

```cpp
cout << (x = 1);
```

which is equivalent to

```cpp
x = 1;
cout << x;
```

The following statement is also correct:

```cpp
i = j = k = 1;
```

Which is equivalent to

```cpp
k = 1;
j = k;
i = j;
```

NOTE

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, \( \text{int } x = 1.0 \) would be illegal because the data type of \( x \) is \( \text{int} \). You cannot assign a double value (1.0) to an int variable without using type casting. Type casting is introduced in §2.8, "Numeric Type Conversions."

NOTE

In C++, any expression can be used as statement. So the following statement is correct, but does not make any practical sense.

```cpp
2 / 3; // A syntactic correct statement
```

The GNU compiler (gcc) warns about this kind of statements if you compile with the option `-Wall` (enable all warnings). It reports "statement has no effect."
2.7.1 Declaring and Initializing Variables in One Step

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```c
int x = 1;
```

This is equivalent to the next two statements:

```c
int x;
x = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```c
int i = 1, j = 2;
```

A variable must be declared before it can be assigned a value. A variable declared in a function must be assigned a value. Otherwise, the value is unpredictable. Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

***End of TIP***

2.8 Named Constants

The value of a variable may change during the execution of the program, but a constant represents permanent data that never changes. In our ComputeArea program, \( \pi \) is a constant. If you use it frequently, you don’t want to keep typing 3.14159; instead, you can name a constant for \( \pi \). Here is the syntax for declaring a constant:

```c
const datatype CONSTANTNAME = VALUE;
```

A constant must be declared and initialized in the same statement. The word `const` is a C++ keyword which means that the constant cannot be changed. For example, you may define \( \pi \) as a constant and rewrite the program in Listing 2.4 as in Listing 2.5.

**Listing 2.5 ComputeArea4.cpp (Using Constants)**

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

int main() { 
```

***PD: Please add line numbers (including space lines) in the following code***

```c
const double PI = 3.14159;
```

```c
int main() { 
```
const double PI = 3.14159;

// Step 1: Read in radius
double radius = 20;

// Step 2: Compute area
double area = radius * radius * PI;

// Step 3: Display the area
cout << "The area is ";
cout << area << std::endl;
}

CAUTION

By convention, constants are named in uppercase: PI, not pi or Pi.

NOTE

There are three benefits of using constants: (1) you don’t have to repeatedly type the same value; (2) the value can be changed in a single location, if necessary; (3) the program is easy to read.

2.9 Numeric Data Types and Operations

Every data type has a range of values. The compiler allocates memory space to store each variable or constant according to its data type. C++ provides primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types. Table 2.1 lists the numeric data types with their typical ranges storage sizes.

Table 2.1

Numeric Data Types
<table>
<thead>
<tr>
<th>Name</th>
<th>Synonymy</th>
<th>Range</th>
<th>Storage Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>short int</td>
<td>$-2^{15}$ (-32,768) to $2^{15}-1$ (32,767)</td>
<td>16-bit signed</td>
</tr>
<tr>
<td>unsigned short</td>
<td>unsigned short int</td>
<td>0 to $2^{16}-1$ (65535)</td>
<td>16-bit unsigned</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td>$-2^{31}$ (-2147483648) to $2^{31}-1$ (2147483647)</td>
<td>32-bit signed</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsigned</td>
<td>0 to $2^{32}-1$ (4294967295)</td>
<td>32-bit unsigned</td>
</tr>
<tr>
<td>long</td>
<td>long int</td>
<td>$-2^{31}$ (-2147483648) to $2^{31}-1$ (2147483647)</td>
<td>32-bit signed</td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsigned long int</td>
<td>0 to $2^{32}-1$ (4294967295)</td>
<td>32-bit unsigned</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>Negative range: $-3.4028235E+38$ to $-1.4E-45$</td>
<td>32-bit IEEE 754</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positive range: $1.4E-45$ to $3.4028235E+38$</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>Negative range: $-1.7976931348623157E+308$ to $-4.9E-324$</td>
<td>64-bit IEEE 754</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positive range: $4.9E-324$ to $1.7976931348623157E+308$</td>
<td></td>
</tr>
<tr>
<td>long double</td>
<td></td>
<td>Negative range: $-1.18E+4932$ to $3.37E-4932$</td>
<td>80-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Positive range: $3.37E-4932$ to $1.18E+4932$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Significant decimal digits: 19</td>
<td></td>
</tr>
</tbody>
</table>

**Side Remark: signed vs. unsigned**

C++ uses three types for integers: short, int, and long. Each integer type comes in two flavors: signed and unsigned. Half the numbers represented by a signed short is negative and the other half is positive. All the numbers represented by an unsigned short are non-negative. Because you have the same storage size for both, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer. If you know the value stored in a variable is always non-negative, declare it as unsigned.

**Side Remark: size may vary**

The size of the data types may vary depending on the compiler and computer you are using. Typically, int and long have the same size. On some systems, long requires 8 bytes.

**TIP**

**Side Remark: sizeof function**

You can use the `sizeof` function to find the size of a type. For example, the following statement displays the size of int, long, and double on your machine.

```cpp
    cout << sizeof(int) << " " << sizeof(long) << " " << sizeof(double);```

***END TIP***

**NOTE**

**NOTE**

**Side Remark: synonymous**
short int is synonymous to short. unsigned short int is synonymous to unsigned short. unsigned int is synonymous to unsigned. long int is synonymous to long. unsigned long int is synonymous to unsigned long. For example,

```c
short int i = 2;
```

is same as

```c
short i = 2;
```

***END NOTE

**<Side Remark: floating-point>**

C++ uses three types for floating-point numbers: float, double, and long double. The double type is usually twice as big as float. So, the double is known as double precision, while float is single precision. The long double is even bigger than double. For most applications, using the double type is desirable.

2.9.1 Numeric Literals

**<Side Remark: literal>**

A literal is a constant value that appears directly in a program. For example, 34, 1000000, and 5.0 are literals in the following statements:

```c
int i = 34;
long k = 1000000;
double d = 5.0;
```

**<Side Remark: integer literal>**

**<Side Remark: long literal>**

An integer literal is stored in memory as an int type value. On a system that uses 4 bytes to represent an int value, value is between \(-2^{31}\) (-2147483648) and \(2^{31}-1\) (2147483647). The literal 2147483648 is too long as an int literal. To denote an integer literal of the long type, append the letter L or l to it (e.g., 2147483648L). L is preferred because l (lowercase L) can easily be confused with 1 (the digit one). Since 2147483648 exceeds the range for the int values, it must be denoted as 2147483648L.

**NOTE**

**<side remark: octal and hex literals>**

By default, an integer literal is a decimal number. To denote an octal integer literal, use a leading 0 (zero), and to denote a hexadecimal integer literal, use a leading Ox or 0X (zero x). For example, the following code displays the decimal value 65535 for hexadecimal number FFFF and decimal value 8 for octal number 10.

```c
cout << 0xFFFF << " " << 010;
```
Hexadecimal numbers, binary numbers, and octal numbers were introduced in §1.5, “Number systems.”

***End of NOTE

**Side Remark: floating-point literals**
Floating-point literals are written with a decimal point. 2.0 is a floating-point number, but 2 is an integer. By default, a floating-point literal is treated as a double type value. For example, 5.0 is considered a double value, not a float value. You can make a number a float by appending the letter f or F. For example, you can use 100.2f or 100.2F to denote a float literal.

**Side Remark: scientific notation**
Floating-point literals can also be specified in scientific notation; for example, 1.23456e+2, the same as 1.23456e2, is equivalent to 1.23456×10² = 123.456, and 1.23456e-2 is equivalent to 1.23456×10⁻² = 0.0123456. E (or e) represents an exponent and can be either in lowercase or uppercase.

**NOTE**
**Side remark: why called floating-point?**
The float and double types are used to represent numbers with a decimal point. Why are they called floating-point numbers? These numbers are stored into scientific notation. When a number such as 50.534 is converted into scientific notation such as 5.0534, its decimal point is moved (i.e., floated) to a new position.

***End of NOTE

2.9.2 Numeric Operators

**Side Remark: operators +, -, *, /, %**
The operators for numeric data types include the standard arithmetic operators: addition (+), subtraction (−), multiplication (*), division (/), and modulus (%), as shown in Table 2.a.

**Table 2.a**
Numeric Operators
<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>34 + 1</td>
<td>35</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>34.0 - 0.1</td>
<td>33.9</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>300 * 30</td>
<td>9000</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>1.0 / 2.0</td>
<td>0.5</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>20 % 3</td>
<td>33.9</td>
</tr>
</tbody>
</table>

**<Side Remark: integer division>**

The result of integer division is an integer. The fractional part is truncated. For example, 5 / 2 yields 2, not 2.5, and -5 / 2 yields -2, not -2.5.

**<Side Remark: integer modulus>**

The modulus (%) operator yields the remainder after division. The left-hand operand is the dividend and the right-hand operand is the divisor. Therefore, 7 % 3 yields 1, 12 % 4 yields 0, 26 % 8 yields 2, and 20 % 13 yields 7.

The % operator is often used for positive integers but also can be used with negative integers. The remainder is negative only if the dividend is negative. For example, -7 % 3 yields -1, -12 % 4 yields 0, -26 % -8 yields -2, and 20 % -13 yields 7. In C++, the % operator is for integers only.

Modulus is very useful in programming. For example, an even number % 2 is always 0 and an odd number % 2 is always 1. So you can use this property to determine whether a number is even or odd. Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression:

Saturday is the 6th day in a week

A week has 7 days

\[(6 + 10) \mod 7 = 2\]

The 2nd day in a week is Tuesday

After 10 days

Listing 2.6 gives a program that obtains minutes and remaining seconds from an amount of time in seconds. For example, 500 seconds contains 8 minutes and 20 seconds.

Listing 2.6 DisplayTime.cpp (Getting Hours and Minutes)
```cpp
#include <iostream>
using namespace std;

int main()
{
    int seconds = 500;
    int minutes = seconds / 60;
    int remainingSeconds = seconds % 60;
    cout << seconds << " seconds is " << minutes << " minutes and " << remainingSeconds << " seconds " << endl;
    return 0;
}
```

<output>
500 seconds is 8 minutes and 20 seconds
</end of output>

Line 7 obtains the minutes using `seconds / 60`. Line 8 (`seconds % 60`) obtains the remaining seconds after taking away minutes.

**Side Remark 1: unary operator**
**Side Remark 2: binary operator**
The `+` and `-` operators can be both unary and binary. A unary operator has only one operand; a binary operator has two operands. For example, the `-` operator in `-5` can be considered a unary operator to negate number 5, whereas the `-` operator in `4 - 5` is a binary operator for subtracting 5 from 4.

**NOTE**
**Side Remark: floating-point to integer**
C++ allows you to assign an integer value to a floating-point variable and a floating-point value to an integer variable vice versa. When assigning a floating-point value to an integer variable, the fraction part of the floating-point value is truncated (not rounded). For example, see the following code:

```cpp
int i = 34.7; // i becomes 34
float f = i;  // f is now 34
float g = 34.3; // g becomes 34.3
long j = g;    // j become 34
```
When a variable is assigned a value that is too large to be stored, it causes overflow. For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the short type is 32767. 32768 is too large.

```c
short value = 32767 + 1;
```

When a variable is assigned a value that is too small to be stored, it causes underflow. For example, executing the following statement causes underflow, because the smallest value that can be stored in a variable of the short type is -32768. -32769 is too small.

```c
short value = -32769;
```

C++ does not report warnings or errors on overflow and underflow. So be careful when working with numbers close to the maximum or minimum range of a given type.

***END CAUTION

2.9.3 Arithmetic Expressions

Writing numeric expressions in C++ involves a straightforward translation of an arithmetic expression using C++ operators. For example, the arithmetic expression

\[
\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + \frac{9\left(\frac{4}{x} + \frac{9+x}{y}\right)}{y}
\]

can be translated into a C++ expression as:

```c
(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x + 9 * (4 / x + (9 + x) / y)
```

The numeric operators in a C++ expression are applied the same way as in an arithmetic expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. Multiplication, division, and modulus operators are applied next. If an expression contains several multiplication, division, and modulus operators, they are applied from left to right. Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Listing 2.7 gives a program that converts a Fahrenheit degree to Celsius using the formula \( celsius = \frac{\frac{5}{9}}{9}(fahrenheit - 32) \).

Listing 2.7 FahrenheitToCelsius.cpp (Converting Fahrenheit to Celsius)
```cpp
#include <iostream>
using namespace std;

int main()
{
    // Enter a degree in Fahrenheit
    double fahrenheit;
    cout << "Enter a degree in Fahrenheit: ";
    cin >> fahrenheit;

    // Obtain a celsius degree
    double celsius = (5.0 / 9) * (fahrenheit - 32);

    // Display result
    cout << "Fahrenheit " << fahrenheit << " is " << celsius << " in Celsius" << endl;
    return 0;
}
```

Enter a degree in Fahrenheit: 100
Fahrenheit 100 is 37.7778 in Celsius

**<side remark: integer vs. decimal division>**
Be careful when applying division. Division of two integers yields an integer in C++. $\frac{5}{9}$ is translated to $\frac{5.0}{9}$ instead of $\frac{5}{9}$ in line 4, because $5 / 9$ yields 0 in C++.

**2.9.4 Shorthand Operators**
Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement adds the current value of $i$ with value 8 and assigns the result back to $i$:

$i = i + 8;$

**<Side Remark: shorthand operator>**
C++ allows you to combine assignment and addition operators using a short cut operator. For example, the preceding statement can be written as:

$i += 8;$
The += is called the addition assignment operator. Other shorthand operators are shown in Table 2.2.

### Table 2.2

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>Addition assignment</td>
<td>i += 8</td>
<td>i = i + 8</td>
</tr>
<tr>
<td>-=</td>
<td>Subtraction assignment</td>
<td>i -= 8</td>
<td>i = i - 8</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplication assignment</td>
<td>i *= 8</td>
<td>i = i * 8</td>
</tr>
<tr>
<td>/=</td>
<td>Division assignment</td>
<td>i /= 8</td>
<td>i = i / 8</td>
</tr>
<tr>
<td>%=</td>
<td>Modulus assignment</td>
<td>i %= 8</td>
<td>i = i % 8</td>
</tr>
</tbody>
</table>

**Side Remark: ++ and -->**

There are two more shorthand operators for incrementing and decrementing a variable by 1. This is handy because that’s often how much the value needs to be changed. These two operators are ++ and --. They can be used in prefix or suffix notation, as shown in Table 2.3.

### Table 2.3

**Increment and Decrement Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++var</td>
<td>preincrement</td>
<td>The expression (++var) increments var by 1 and evaluates to the new value in var after the increment.</td>
</tr>
<tr>
<td>var++</td>
<td>postincrement</td>
<td>The expression (var++) evaluates to the original value in var and increments var by 1.</td>
</tr>
<tr>
<td>--var</td>
<td>predecrement</td>
<td>The expression (--var) decrements var by 1 and evaluates to the new value in var after the decrement.</td>
</tr>
<tr>
<td>var--</td>
<td>postdecrement</td>
<td>The expression (var--) evaluates to the original value in var and decrements var by 1.</td>
</tr>
</tbody>
</table>

**Side Remark 1: preincrement, predecrement**

**Side Remark 2: postincrement, postdecrement**

If the operator is before (prefixed to) the variable, the variable is incremented or decremented by 1, then the new value of the variable is returned. If the operator is after (suffixed to) the variable, the original old value of the variable is returned, then the variable is incremented or decremented by 1. Therefore, the prefixes ++x and --x are referred to, respectively, as the preincrement operator and the predecrement operator; and the suffixes x++ and x-- are referred to, respectively, as the postincrement operator and the postdecrement operator. The prefix form of ++ (or --) and the suffix form of ++ (or --) are the same if they are used in isolation, but they cause different effects when used in an expression. The following code illustrates this:
```plaintext
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```plaintext
int newNum = 10 * i;
i = i + 1;
```

In this case, `i` is incremented by 1, then the old value of `i` is returned and used in the multiplication. So `newNum` becomes 100. If `i++` is replaced by `++i` as follows,

```plaintext
int i = 10;
int newNum = 10 * (++i);
```

`i` is incremented by 1, and the new value of `i` is returned and used in the multiplication. Thus `newNum` becomes 110.

Here is another example:

```plaintext
double x = 1.0;
double y = 5.0;
double z = x-- + (++y);
```

After all three lines are executed, `y` becomes 6.0, `z` becomes 7.0, and `x` becomes 0.0.

The increment operator `++` and the decrement operator `--` can be applied to all integer and floating-point types. These operators are often used in loop statements. A loop statement is a structure that controls how many times an operation or a sequence of operations is performed in succession. This structure, and the subject of loop statements, is introduced in Chapter 4, "Loops."

**TIP**

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i`.

**NOTE**

 `<Side Remark: expression statement>`

Like the assignment operator (=), the operators (+=, -=, *=, /=, %=, ++, and --) can be used to form an assignment statement as well as an expression. For example, in the following code, `x = 2` is a statement in the first line and is an expression in the second line.

```plaintext
x = 2; // statement
cout << (x = 2); // expression
```

If a statement is used as an expression, it is called an expression statement.
There are no spaces in the shorthand operators. For example, += should be +.

2.10 Numeric Type Conversions
Sometimes it is necessary to mix numeric values of different types in a computation. Consider the following statements:

```cpp
byte i = 100;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
```

Are these statements correct? C++ allows binary operations on values of different types. When performing a binary operation involving two operands of different types, C++ automatically converts the operand based on the following rules:

<Side Remark: converting operands>

1. If one of the operands is `long double`, the other is converted into `long double`.
2. Otherwise, if one of the operands is `double`, the other is converted into `double`.
3. Otherwise, if one of the operands is `float`, the other is converted into `float`.
4. Otherwise, if one of the operands is `unsigned long`, the other is converted into `unsigned long`.
5. Otherwise, if one of the operands is `long`, the other is converted into `long`.
6. Otherwise, if one of the operands is `unsigned int`, the other is converted into `unsigned int`.
7. Otherwise, both operands are converted into `int`.

For example, the result of `1 / 2` is 0, because both operands are `int` values. The result of `1.0 / 2` is 0.5, since `1.0` is `double` and 2 is converted to 2.0.

<Side Remark 1: type casting>
C++ also allows you to manually convert a value from one type to another using a casting operator. The syntax is

```
static_cast<type>(value)
```

where `value` is a variable, a literal, or an expression and `type` is the type you wish to convert the `value` to. For example, the following statement

```
cout << static_cast<double>(1) / 2;
```

displays 0.5, because 1 is converted to 1.0 first, then 1.0 is divided by 2. However, the statement
cout << 1 / 2;

displays 0.

NOTE

<Side Remark: (type) casting>
It is worth mentioning that static casting can also be done using the (type) syntax, i.e., giving the target type in parentheses, followed by a variable, a literal, or an expression. For example,

```c++
int i = (int)5.4;
```

This is the same as

```c++
int i = static_cast<int>(5.4);
```

***End of NOTE

<Side Remark: widening a type>
<Side Remark: narrowing a type>
<side remark:loss of precision>
Casting a variable of a type with a small range to a variable of a type with a larger range is known as widening a type. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as narrowing a type. Narrowing a type, such as assigning a `double` value to an `int` variable, may cause loss of precision. Lost information might lead to inaccurate results.

NOTE
Casting does not change the variable being cast. For example, `d` is not changed after casting in the following code:

```c++
double d = 4.5;
int i = static_cast<int>(d);  // d is not changed
```

***End of NOTE

NOTE

<side remark: compiler warning>
The GNU C++ compiler will give a warning when you narrow a type unless you use static-cast to make the conversion explicit.

***End of NOTE

Listing 2.8 gives a program that displays the sales tax with two digits after the decimal point.

Listing 2.8 SalesTax.cpp (Displaying Sales Tax)

```c++
#include <iostream>
using namespace std;
```
```cpp
int main()
{
    // Enter purchase amount
    double purchaseAmount;
    cout << "Enter purchase amount: ";
    cin >> purchaseAmount;

    double tax = purchaseAmount * 0.06;
    cout << "Sales tax is " << static_cast<int>(tax * 100) / 100.0;
}
```

The output is:
Enter purchase amount: 197.55
Sales tax is 11.85

---

### Side Remark: Formatting Numbers

Variable `purchaseAmount` stores the purchase amount entered by the user (lines 7-9). Suppose the user entered 197.55. The sales tax is 6% of the purchase, so the tax is evaluated as 11.853 (line 11). The statement in line 12 displays the tax 11.85 with two digits after the decimal point. Note that `static_cast<int>(tax * 100)` is 1185, so `static_cast<int>(tax * 100) / 100.0` is 11.85.

---

### 2.11 Character Data Type and Operations

#### Side Remark: `char` Type

The character data type, `char`, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```cpp
char letter = 'A';
char numChar = '4';
```

The first statement assigns character `A` to the `char` variable `letter`. The second statement assigns the digit character `4` to the `char` variable `numChar`.

#### CAUTION

#### Side Remark: `char` Literal

A string literal must be enclosed in quotation marks. A character literal is a single character enclosed in single quotation marks. So "A" is a string, and 'A' is a character.

#### Side Remark: Character Encoding

Computers use binary numbers internally. A character is stored as a sequence of 0s and 1s in a computer. To convert a character to its binary representation is called encoding. There are different ways to encode a character. How characters are encoded is defined by an encoding scheme.

#### Side Remark: ASCII

---

66
Most computers use ASCII (American Standard Code for Information Interchange), a 7-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. See Appendix B, "The ASCII Character Set," for a list of ASCII characters and their decimal and hexadecimal codes. On most systems, the size of the char type is 1 byte.

NOTE
<side remark: char increment and decrement>
The increment and decrement operators can also be used on char variables to get the next or preceding character. For example, the following statements display character b.
```
char ch = 'a';
cout << ++ch;
```
***End of NOTE

NOTE
<side remark: read character>
To read a character from the keyboard, use
```
cout << "Enter a character: ";
char ch;
cin >> ch;
```
***End of NOTE

2.11.1 Escape Sequences for Special Characters

C++ allows you to use escape sequences to represent special characters, as shown in Table 2.4. An escape sequence begins with the backslash character (\) followed by a character that has a special meaning to the compiler.

<table>
<thead>
<tr>
<th>Character Escape Sequence</th>
<th>Name</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>8</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
<td>9</td>
</tr>
<tr>
<td>\n</td>
<td>Linefeed</td>
<td>10</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed</td>
<td>12</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
<td>13</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
<td>92</td>
</tr>
<tr>
<td>'</td>
<td>Single Quote</td>
<td>39</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double Quote</td>
<td>34</td>
</tr>
</tbody>
</table>

Suppose you want to print the quoted message shown below:
He said "C++ is powerful"

Here is how to write the statement:

```cpp
cout << "He said \"C++ is powerful\"";
```

**NOTE**

<side remarks: whitespace>
The characters ' ', '	', '', '', and '
' are known as the whitespace characters.

**NOTE**

<Side Remark: \n vs. endl>
The following two statements both display a string and move the cursor to the next line.

```cpp
cout << "Welcome to C++\n";
cout << "Welcome to C++" << endl;
```

However, using `endl` ensures that the output is displayed immediately on all platforms.

### 2.11.2 Casting between `char` and Numeric Types

A `char` can be cast into any numeric type, and vice versa. When an integer is cast into a `char`, only its lower eight bits of data are used (assume that your system stores a char in 8 bits); the other part is ignored. For example, see the following code:

```cpp
char c = 0xFF41; // the lower 8 bits hex code 41 is assigned to c
cout << c;       // c is character A
```

When a floating-point value is cast into a `char`, the floating-point value is first cast into an `int`, which is then cast into a `char`.

```cpp
char c = 65.25;  // decimal 65 is assigned to t
cout << c;      // t is character A
```

When a `char` is cast into a numeric type, the character’s ASCII is cast into the specified numeric type. For example, see the following code:

```cpp
int i = 'A';     // the ASCII code of character A is assigned to i
cout << i;       // i is 65
```

**NOTE**

<side remark: numeric operators on characters>
The `char` type is treated as if it is an integer of the byte size. All numeric operators can be applied to `char` operands. A `char` operand is automatically cast into a number if the other operand is a number.
or a character. For example, the following statements

```c++
int i = '2' + '3'; // (int)'2' is 50 and (int)'3' is 51
cout << "i is " << i << endl; // i is decimal 101

int j = 2 + 'a'; // (int)'a' is 97
cout << "j is " << j << endl;
cout << j << " is the ASCII code for character " <<
static_cast<char>(j) << endl;
```

display

```
i is 101
j is 99
99 is the ASCII code for character c
```

***End of NOTE

**NOTE**

It is worthwhile to note that the ASCII for lowercase letters are consecutive integers starting from the code for 'a', then for 'b', 'c', ..., and 'z'. The same is true for the uppercase letters. Furthermore, the ASCII code for 'a' is greater than the code for 'A'. So 'a' - 'A' is the same as 'b' - 'B'. For a lowercase letter `ch`, its corresponding uppercase letter is `static_cast<char>('A' + (ch - 'a'))`.

### 2.12 Case Studies

In the preceding sections, you learned about variables, constants, primitive data types, operators, and expressions. You are now ready to use them to write interesting programs. This section presents three examples: computing loan payments, breaking a sum of money down into smaller units, and displaying the current time.

#### 2.12.1 Example: Computing Loan Payments

This example shows you how to write a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. The program lets the user enter the interest rate, number of years, and loan amount, and then computes the monthly payment and the total payment. It concludes by displaying the monthly and total payments.

The formula to compute the monthly payment is as follows:

\[
\text{monthlyPayment} = \frac{\text{loanAmount} \times \text{monthlyInterestRate}}{1 - \frac{1}{(1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}}}
\]
You don’t have to know how this formula is derived. Nonetheless, given the monthly interest rate, number of years, and loan amount, you can use it to compute the monthly payment.

Here are the steps in developing the program:

Prompt the user to enter the annual interest rate, number of years, and loan amount.
Obtain the monthly interest rate from the annual interest rate.
Compute the monthly payment using the preceding formula.
Compute the total payment, which is the monthly payment multiplied by 12 and multiplied by the number of years.
1. Display the monthly payment and total payment in a message dialog.

<Side Remark: pow(a, b) function>
In the formula, you have to compute \((1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}\). C++ contains the pow(a, b) function in the cmath library, which can be used to compute \(a^b\). For example,

```cpp
cout << pow(2.0, 3)
```
displays 8.

Listing 2.9 gives the complete program.

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

int main()
{
    // Enter yearly interest rate
    cout << "Enter yearly interest rate, for example 8.25: ";
    double annualInterestRate;
    cin >> annualInterestRate;
    // Obtain monthly interest rate
    double monthlyInterestRate = annualInterestRate / 1200;
    // Enter number of years
    cout << "Enter number of years as an integer, for example 5: ";
    int numberOfYears;
    cin >> numberOfYears;
    // Enter loan amount
    cout << "Enter loan amount, for example 120000.95: ";
    double loanAmount;
    cin >> loanAmount;
    // Compute the monthly payment
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 - 1 / (1 + monthlyInterestRate)^(numberOfYears * 12));
    // Compute the total payment
    double totalPayment = monthlyPayment * numberOfYears * 12;
    // Display the monthly payment and total payment in a message dialog
    cout << "Monthly payment: 
```
// Calculate payment
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 - 1 / pow(1 + monthlyInterestRate, numberOfYears * 12));
    double totalPayment = monthlyPayment * numberOfYears * 12;

    // Format to keep two digits after the decimal point
    monthlyPayment = static_cast<int>(monthlyPayment * 100) / 100.0;
    totalPayment = static_cast<int>(totalPayment * 100) / 100.0;

    // Display results
    cout << "The monthly payment is " << monthlyPayment << " \n"
    << "The total payment is " << totalPayment << endl;

<Output>
    Enter yearly interest rate, for example 8.25: 6.25
    Enter number of years as an integer, for example 5: 15
    Enter loan amount, for example 120000.95: 60000
    The monthly payment is 514.45
    The total payment is 92601.7

<End Output>

To use this function, you have to include cmath in the program (line 2) in the same way you include the iostream library (line 1).

The program prompts the user to enter annualInterestRate, numberOfYears, and loanAmount in lines 7-23. If you entered an input other than a numeric value, a runtime error would occur. In Chapter 13, “Exception Handling,” you will learn how to handle the exception so that the program can continue to run.

Each new variable in a function must be declared once and only once. Choose the most appropriate data type for the variable. For example, numberOfYears is better declared as int (line 22), although it could be declared as long, float, or double. Note that unsigned short might be the most appropriate for numberOfYears. For simplicity, however, the examples in this book will use int for integer and double for floating-point values.

The formula for computing the monthly payment is translated into C++ code in lines 26-28.

<Side Remark: formatting numbers>
The statements in lines 31-32 are for formatting the number to keep two digits after the decimal point. For example, if monthlyPayment is 2076.0252175, static cast<int>(monthlyPayment * 100) is 207602. Therefore, static cast<int>(monthlyPayment * 100) / 100.0 yields 2076.02.

2.12.2 Example: Counting Monetary Units
This section presents a program that classifies a given amount of money into smaller monetary units. The program lets the user enter an amount as a double value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in dollars, quarters, dimes, nickels, and pennies, as shown in sample output.

Your program should report the maximum number of dollars, then the maximum number of quarters, and so on, in this order.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number such as 11.56.
2. Convert the amount (e.g., 11.56) into cents (1156).
3. Divide the cents by 100 to find the number of dollars. Obtain the remaining cents using the cents remainder 100.
4. Divide the remaining cents by 25 to find the number of quarters. Obtain the remaining cents using the remaining cents remainder 25.
5. Divide the remaining cents by 10 to find the number of dimes. Obtain the remaining cents using the remaining cents remainder 10.
6. Divide the remaining cents by 5 to find the number of nickels. Obtain the remaining cents using the remaining cents remainder 5.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is given in Listing 2.10.

Listing 2.10 ComputeChange.cpp (Monetary Units)

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

int main()
{
    // Receive the amount
    cout << "Enter an amount in double, for example 11.56: ";
    double amount;
    cin >> amount;
    int remainingAmount = static_cast<int>(amount * 100);

    // Find the number of one dollars
    int numberOfOneDollars = remainingAmount / 100;
    remainingAmount = remainingAmount % 100;

    // Find the number of quarters
    int numberOfQuarters = remainingAmount / 25;
    remainingAmount = remainingAmount % 25;

    // Find the number of dimes
    int numberOfDimes = remainingAmount / 10;
    remainingAmount = remainingAmount % 10;

    // Find the number of nickels
    int numberOfNickels = remainingAmount / 5;
    remainingAmount = remainingAmount % 5;

    // The remaining cents are the pennies
    int numberOfPennies = remainingAmount;

    cout << "The result is: 
    " << numberOfOneDollars << " dollars, 
    " << numberOfQuarters << " quarters, 
    " << numberOfDimes << " dimes, 
    " << numberOfNickels << " nickels, 
    " << numberOfPennies << " pennies."
;
    return 0;
}
```

72
// Find the number of quarters in the remaining amount
    int numberOfQuarters = remainingAmount / 25;
    remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
    int numberOfDimes = remainingAmount / 10;
    remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining amount
    int numberOfNickels = remainingAmount / 5;
    remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining amount
    int numberOfPennies = remainingAmount;

// Display results
    cout << "Your amount " << amount << " consists of 
" <<
        "\t" << numberOfOneDollars << " dollars\n" <<
        "\t" << numberOfQuarters << " quarters\n" <<
        "\t" << numberOfDimes << " dimes\n" <<
        "\t" << numberOfNickels << " nickels\n" <<
        "\t" << numberOfPennies << " pennies";

<Output>
Enter an amount in double, for example 11.56: 11.56
Your amount 11.56 consists of
11 dollars
2 quarters
0 dimes
1 nickels
1 pennies
<END Output>

The variable amount stores the amount entered from the keyboard
(lines 6-8). This variable should not be changed because the
amount has to be used at the end of the program to display the
results. The program introduces the variable remainingAmount
(line 10) to store the changing remainingAmount.

The variable amount is a double decimal representing dollars and
cents. It is converted to an int variable remainingAmount, which
represents all the cents. For instance, if amount is 11.56, then
the initial remainingAmount is 1156. The division operator
yields the integer part of the division. So 1156 / 100 is 11.
The remainder operator obtains the remainder of the division. So
1156 % 100 is 56.

The program extracts the maximum number of singles from the
total amount and obtains the remaining amount in the variable
remainingAmount (lines 13-14). It then extracts the maximum
number of quarters from remainingAmount and obtains a new
remainingAmount (lines 17-18). Continuing the same process, the
program finds the maximum number of dimes, nickels, and pennies
in the remaining amount.

<Side Remark: loss of precision>
One serious problem with this example is the possible loss of
precision when casting a double amount to an int.
remainingAmount. This could lead to an inaccurate result. If you try to enter the amount 10.03, 10.03 * 100 becomes 1002.9999999999999. You will find that the program displays 10 dollars and 2 pennies. To fix the problem, enter the amount as an as integer value representing cents (see Exercise 2.10).

As shown in the sample output, 0 dimes, 1 nickels, and 1 pennies are displayed in the result. It would be better not to display 0 dimes, and to display 1 nickel and 1 penny using the singular forms of the words. You will learn how to use selection statements to modify this program in the next chapter (see Exercise 3.7).

2.12.3 Example: Displaying the Current Time

This section presents a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.

<Side Remark: currentTimeMills>
<Side Remark: Unix epoch>
The time(0) function, in the ctime header file, returns the current time in seconds elapsed since the time 00:00:00 on January 1, 1970 GMT, as shown in Figure 2.1. This time is known as the Unix epoch because 1970 was the year when the Unix operating system was formally introduced.

Figure 2.1
The time(0) returns the number of seconds since the Unix epoch.

You can use this function to obtain the current time, and then compute the current second, minute and hour as follows.

1. Obtain the total seconds since midnight, Jan 1, 1970 in totalSeconds by invoking time(0) (e.g., 1103203148 seconds).
2. Compute the current second from totalSeconds % 60 (e.g., 1103203148 seconds % 60 = 8, which is the current second).
3. Obtain the total minutes totalMinutes by dividing totalSeconds by 60 (e.g., 1103203148 seconds / 60 = 18386719 minutes).
4. Compute the current minute from totalMinutes % 60 (e.g., 18386719 minutes % 60 = 19, which is the current minute).
5. Obtain the total hours totalHours by dividing totalMinutes by 60 (e.g., 18386719 minutes / 60 = 306445 hours).
6. Compute the current hour from totalHours % 24 (e.g., 306445 hours % 24 = 19, which is the current hour).

Listing 2.11 shows the complete program follows followed by a sample run.

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

int main()
{
    // Obtain the total seconds since the midnight, Jan 1, 1970
    int totalSeconds = time(0);

    // Compute the current second in the minute in the hour
    int currentSecond = totalSeconds % 60;

    // Obtain the total minutes
    int totalMinutes = totalSeconds / 60;

    // Compute the current minute in the hour
    int currentMinute = totalMinutes % 60;

    // Obtain the total hours
    long totalHours = totalMinutes / 60;

    // Compute the current hour
    int currentHour = (int)(totalHours % 24);

    // Display results
    cout << "Current time is " << currentHour << ":" << currentMinute << ":" << currentSecond << " GMT" << endl;
}
```

<Output>
Current time is 13:19:8 GMT
<End Output>

When `time(0)` (line 8) is invoked, it returns the difference, measured in seconds, between the current GMT and midnight, January 1, 1970 GMT.

2.13 Programming Style and Documentation

**Side Remark: programming style**

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line,
but writing it all on one line would be bad programming style because it would be hard to read. Documentation is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. So far you have learned some good programming styles. This section summarizes them and gives several guidelines. More detailed guidelines on programming style and documentation can be found in Supplement D, “C++ Coding Style Guidelines,” on the Companion Web site.

2.13.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

2.13.2 Naming Variables and Constants

Make sure that you choose descriptive names with straightforward meanings for the variables, constants, and functions in your program. Names are case-sensitive. Follow the conventions adopted in this book for naming variables and constants.

Avoid using abbreviation for identifiers. Using complete words is more descriptive. For example, numberOfStudents is better than numStuds, numOfStuds, or numOfStudents.

2.13.3 Proper Indentation and Spacing

A consistent indentation style makes programs clear and easy to read. Indentation is used to illustrate the structural relationships between a program’s components or statements. C++ compiler can read the program even if all of the statements are in a straight line, but it is easier to read and maintain code that is aligned properly. Indent each subcomponent or statement two spaces more than the structure within which it is nested.

A single space should be added on both sides of a binary operator, as shown in the following statement:

```
int i = 3 + 4 * 5; // Bad style
int i = 3 + 4 * 5; // Good style
```

A single space line should be used to separate segments of the code to make the program easier to read.
2.14 Programming Errors
Programming errors are unavoidable, even for experienced programmers. Errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

2.14.1 Syntax Errors

Errors that occur during compilation are called syntax errors or compilation errors. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them. For example, compiling the following program results in a syntax error, as shown in Figure 2.2.

```
// ShowSyntaxErrors.cpp: The program contains syntax errors
#include <iostream>
using namespace std;

int main()
{
    i = 30;
    cout << i + 4;
}
```

Three lines of errors are reported. All are the result of not declaring variable i. Since a single error will often display many lines of compilation errors, it is a good practice to start debugging from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

2.14.2 Runtime Errors

Runtime errors are errors that cause a program to terminate abnormally. Runtime errors occur while an application is running if the environment detects an operation that is impossible to carry out. Input errors are typical runtime errors.
An input error occurs when the user enters an unexpected input value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program. To prevent input errors, the program should prompt the user to enter the correct type of values. It may display a message like “Please enter an integer” before reading an integer from the keyboard.

2.14.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the following program to add number1 to number2.

```cpp
// ShowLogicErrors.cpp: The program contains a logic error
#include <iostream>
using namespace std;

int main()
{
    int number1 = 3;
    int number2 = 3;
    number2 += number1 + number2;
    cout << "number2 is " << number2 << endl;
}
```

The program does not have syntax errors or runtime errors, but it does not print the correct result for number2. See if you can find the error.

2.15 Debugging

In general, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are there. Runtime errors are not difficult to find either, since the operating system displays them on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

Side Remark 1: bugs
Side Remark 2: debugging
Side Remark 3: hand-traces

Logic errors are called bugs. The process of finding and correcting errors is called debugging. A common approach to debugging is to use a combination of functions to narrow down to the part of the program where the bug is located. You can hand-trace the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach for debugging is to use a debugger utility.
All the C++ IDE tools, such as C++Builder and Visual C++.NET, include integrated debuggers. The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features:

**Executing a single statement at a time:** The debugger allows you to execute one statement at a time so that you can see the effect of each statement.

- **Tracing into or stepping over a function:** If a function is being executed, you can ask the debugger to enter the function and execute one statement at a time in the function, or you can ask it to step over the entire function. You should step over the entire function if you know that the function works. For example, always step over system-supplied functions, such as `pow(a, b)`.

- **Setting breakpoints:** You can also set a breakpoint at a specific statement. Your program pauses when it reaches a breakpoint and displays the line with the breakpoint. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your programming error starts. You can set a breakpoint at that line and have the program execute until it reaches the breakpoint.

- **Displaying variables:** The debugger lets you select several variables and display their values. As you trace through a program, the content of a variable is continuously updated.

- **Displaying call stacks:** The debugger lets you trace all of the function calls and lists all pending functions. This feature is helpful when you need to see a large picture of the program-execution flow.

- **Modifying variables:** Some debuggers enable you to modify the value of a variable when debugging. This is convenient when you want to test a program with different samples but do not want to leave the debugger.

**TIP**

*side remark: debugging in IDE>*

If you use an IDE such as C++Builder and Microsoft Visual C++.NET, please refer to *Learning C++ Effectively with C++Builder/Microsoft Visual C++.NET* in the supplement on the Companion Website. The supplement shows you how to use a debugger to trace programs and how debugging can help learning C++ effectively.

**Key Terms**

*PD: Please place terms in two columns same as in intro5e.*

- **algorithm** 33
- **assignment operator** (=) 37
Chapter Summary

- C++ provides integer types (short, int, long, unsigned short, unsigned int, and unsigned long) that represent signed and unsigned integers of various sizes, and floating-point types (float, double, and long double) that represent floating-point numbers of two different
precisions. Character type (char) represents a single character. These are called primitive data types. When they are declared, the variables of these types are created and assigned memory space. The actual sizes of these types are dependent on the compiler.

- C++ provides operators that perform numeric operations: + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). Integer division (/) yields an integer result. The modulus operator (%) yields the remainder of the division.

- The increment operator (++) and the decrement operator (––) increment or decrement a variable by 1. If the operator is prefixed to the variable, the variable is first incremented or decremented by 1, then used in the expression. If the operator is a suffix to the variable, the variable is incremented or decremented by 1, but then the original old value is used in the expression.

- All the numeric operators can be applied to characters. When an operand is a character, the character’s code value is used in the operation.

- You can use casting to convert a value of one type into another type. Casting a variable of a type with a small range to a variable of a type with a larger range is known as widening a type. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as narrowing a type. Narrowing a type may lose precision.

- Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors. Errors that occur during compilation are called syntax errors or compilation errors. Runtime errors are errors that cause a program to terminate abnormally. Logic errors occur when a program does not perform the way it was intended to.

Review Questions
Sections 2.2 – 2.6

2.1 Which of the following identifiers are valid?
   - x, X, a++, --a, 4#R, $4, #44, apps

2.2 Which of the following are C++ keywords?
   - main, include, int, x, y, radius

2.3 Translate the following pseudocode into C++ code:
   - Step 1: Declare a double variable named miles with initial value 100;
   - Step 2: Declare a double constant named MILE TO KILOMETER with value 1.609;
Step 3: Declare a double variable named \texttt{kilometer}, multiply miles and \texttt{MILE\_TO\_KILOMETER} and assign the result to \texttt{kilometer}.

Step 4: Display \texttt{kilometer} to the console.

2.4
What are the benefits of using constants? Declare an int constant \texttt{SIZE} with value 20.

Section 2.9 Numeric Data Types and Operations

2.5
Assume that \texttt{int a = 1} and \texttt{double d = 1.0}, and that each expression is independent. What are the results of the following expressions?

\[
\begin{align*}
\texttt{a} &= \texttt{46 / 9}; \\
\texttt{a} &= \texttt{46 \% 9 + 4 * 4 - 2}; \\
\texttt{a} &= \texttt{45 + 45 \% (23 \% 3 \% 2)}; \\
\texttt{d} &= \texttt{3 / a + 4}; \\
\texttt{d} &= \texttt{1.5 * 3 + (++a)}; \\
\texttt{d} &= \texttt{1.5 * 3 + a++;}
\end{align*}
\]

2.6
Show the result of the following expressions.

\[
\begin{align*}
56 \% 6 &= 4 \\
78 \% -4 &= 2 \\
-34 \% 5 &= 5 \\
-34 \% -5 &= 5 \\
5 \% 1 &= 5
\end{align*}
\]

2.7
Find the size of \texttt{short}, \texttt{int}, \texttt{long}, \texttt{float}, and \texttt{double} on your machine.

2.8
What is the result of \texttt{25 / 4}? How would you rewrite the expression if you wished the result to be a floating-point number?

2.9
Are the following statements correct? If so, show the output.

\[
\begin{align*}
\text{cout} &\ll \text{"the output for } 25 / 4 \text{ is "} \ll 25 / 4 \ll \text{endl;} \\
\text{cout} &\ll \text{"the output for } 25 / 4.0 \text{ is "} \ll 25 / 4.0 \ll \text{endl;}
\end{align*}
\]

2.10
How would you write the following arithmetic expression in C++?

\[
\frac{4}{3(r+34)} = \frac{9(a + bc)}{a + bd} + \frac{3 + d(2 + a)}{a + bd}
\]

2.11
Which of these statements are true?

- a. Any expression can be used as a statement in C++.
- b. The expression \texttt{x++} can be used as a statement.
- c. The statement \texttt{x = x + 5} is also an expression.
- d. The statement \texttt{x = y = x = 0} is illegal.

2.12
Which of the following are correct literals for floating-point numbers?

- 12.3, 12.3e+2, 23.4e-2, -334.4, 20, 39F, 40D
2.13
Identify and fix the errors in the following code:

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

int Main()
{
    int i = k + 1;
    cout << i++ << endl;

    int i = 1;
    cout << i++ << endl;

    return 0;
}
```

Section 2.10 Numeric Type Conversions

2.14
Can different types of numeric values be used together in a computation?

2.15
What does an explicit conversion from a `double` to an `int` do with the fractional part of the `double` value? Does casting change the variable being cast?

2.16
Show the following output.

```cpp
double f = 12.5F;
int i = f;
cout << "f is " << f << endl;
cout << "i is " << i << endl;
```

Section 2.11 Character Data Type and Operations

2.17
Use print statements to find out the ASCII code for '1', 'A', 'B', 'a', 'b'. Use print statements to find out the character for the decimal code 40, 59, 79, 85, 90. Use print statements to find out the character for the hexadecimal code 40, 5A, 71, 72, 7A.

2.18
Which of the following are correct literals for characters?

'1', '	', '&', '', '
'

2.19
How do you display characters \ and "?

2.20
Evaluate the following:

```cpp
int i = '1';
int j = '1' + '2';
int k = 'a';
char c = 'a';
```
2.21

Can the following conversions involving casting be allowed? If so, find the converted result.

```cpp
char c = 'A';
int i = c;

float f = 1000.34f;
int i = f;

double d = 1000.34;
int i = d;

int i = 97;
char c = i;
```

2.22

How do you obtain the current minute?

Sections 2.13-2.15

2.23

How do you denote a comment line and a comment paragraph?

2.24

What are the naming conventions for constants and variables? Which of the following items can be a constant or a variable according to the naming conventions?

MAX VALUE, Test, read, readInt

2.25

Reformat the following program according to the programming style and documentation guidelines.

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

int main()
{
    cout << "2 % 3 = " << 2 % 3;
    return 0;
}
```

2.26

Describe syntax errors, runtime errors, and logic errors.

Programming Exercises

NOTE

<side remark: level of difficulty>

NOTE: Solutions to even-numbered exercises are on the Companion Web site. Solutions to all exercises are on the Instructor Resource Web site. The level of difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***)

Debugging TIP
The compiler usually gives a reason for a syntax error. If you don’t know how to correct it, compare your program closely with similar examples in the text character by character.

Sections 2.2 – 2.10

2.1  
(Converting Fahrenheit to Celsius) Write a program that reads a Fahrenheit degree in double from an input dialog box, then converts it to Celsius and displays the result in a message dialog box. The formula for the conversion is as follows:

\[
celsius = \left(\frac{5}{9}\right) \times (\text{fahrenheit} - 32)
\]

HINT
In C++, \(5 / 9\) is 0, so you need to write \(5.0 / 9\) in the program to obtain the correct result.

2.2  
(Computing the volume of a cylinder) Write a program that reads in the radius and length of a cylinder and computes volume using the following formulas:

\[
\text{area} = \text{radius} \times \text{radius} \times \pi \\
\text{volume} = \text{area} \times \text{length}
\]

2.3  
(Converting feet into meters) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is 0.305 meters.

2.4  
(Converting pounds into kilograms) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is 0.454 kilograms.

2.5*  
(Calculating tips) Write a program that reads the subtotal and the gratuity rate, and computes the gratuity and total. For example, if the user enters 10 for subtotal and 15 percent for gratuity rate, the program displays $1.5 as gratuity and $11.5 as total.

2.6**  
(Summing the digits in an integer) Write a program that reads an integer between 0 and 1000 and adds all the digits in the integer. For example, if an integer is 932, the sum of all its digits is 14.

HINT
Use the \% operator to extract digits, and use the \(/\) operator to remove the extracted digit. For instance, 932 \% 10 = 2 and 932 / 10 = 93.
Section 2.11 Character Data Type and Operations

2.7* (Converting an uppercase letter to lowercase) Write a program that converts an uppercase letter to a lowercase letter.

**HINT**
In the ASCII table (see Appendix B), uppercase letters appear before lowercase letters. The offset between any uppercase letter and its corresponding lowercase letter is the same. So you can find a lowercase letter from its corresponding uppercase letter, as follows:

```c
int offset = 'a' - 'A';
char lowercase = uppercase + offset;
```

2.8* (Finding the character of an ASCII code) Write a program that receives an ASCII code (an integer between 0 and 128) and displays its character. For example, if the user enters 97, the program displays character a.

Section 2.12

2.9* (Calculating the future investment value) Write a program that reads in investment amount, annual interest rate, and number of years, and displays the future investment value using the following formula:

```
futureInvestmentValue = investmentAmount * (1 + monthlyInterestRate)^numberOfYears*12
```

For example, if you entered amount 1000, annual interest rate 3.25%, and number of years 1, the future investment value is 1032.98.

**HINT**
Use the `pow(a, b)` function to compute `a` raised to the power of `b`.

2.10* (Monetary units) Rewrite Listing 2.10, ComputeChange.cpp, to fix the possible loss of accuracy when converting a double value to an int value. Enter the input as an integer whose last two digits represent the cents. For example, the input 1156 represents 11 dollars and 56 cents.

2.11* (Calculating interests) If you know the balance and annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

```
interest = balance * (annualInterestRate / 1200)
```
Write a program that reads the balance and annual percentage interest rate and displays the interest for the next month.