5 Math Operations

OBJECTIVES

- *Use the arithmetic operators.*
- *Increment and decrement variables.*
- *Understand the order of operations.*
- *Use mixed data types.*
- *Avoid overflow, underflow, and floating-point errors.*

Overview

In this chapter, you will learn about C++ math operations and demonstrate each concept by compiling and running programs.

First you will learn about C++ operators, some of which have already been mentioned. Next, you will learn how to build expressions using the operators. You will also learn shortcuts and how to avoid pitfalls.

CHAPTER 5, SECTION 1

The Fundamental Operators

There are several types of operators. In this chapter you will be concerned only with the assignment operator, arithmetic operators, and some special operators.

ASSIGNMENT OPERATOR

You have already used the assignment operator (=) to initialize variables. So you already know most of what there is to know about the assignment operator. The ***assignment operator*** changes the value of the variable to the left of the operator. Consider the statement below:

```
i = 25;
```

The statement `i = 25;` changes the value of variable `i` to 25, regardless of what it was before the statement.

EXERCISE 5-1 THE ASSIGNMENT OPERATOR

1. Turn on your computer and access the C++ compiler's text editor. Enter the following program onto a blank editor screen and save your source code file as *IASSIGN.CPP*.

```
#include <iostream.h> // necessary for cout command

main()
{
  int i;   // declare i as an integer
  i = 10000;   // assign the value 10000 to i
  cout << i << '\n';
  i = 25;      // assign the value 25 to i
  cout << i << '\n';
  return 0;
}
```

2. Compile and run the program. Notice the difference between the value of **i** when it is displayed after the first **cout << i** and after the second.

3. Leave the source code file open for the next exercise.

Recall from Chapter 4 that you can declare more than one variable in a single statement. For example, instead of:

```
int i;
int j;
int k;
```

you can use:

```
int i,j,k;
```

You can use a similar shortcut when initializing multiple variables. If you have more than one variable that you want to initialize to the same value, you can use a statement like:

```
i = j = k = 25;
```

EXERCISE 5-2 MULTIPLE ASSIGNMENTS

1. Modify the program on your screen to read as follows:

```
#include <iostream.h> // necessary for cout command

main()
{
  int i,j,k;          // declare i,j, and k as integers

  i = j = k = 10;     // initialize all of the variables to 10
  cout << i << '\n';
  cout << j << '\n';
  cout << k << '\n';
  return 0;
}
```

2. Compile and run the program. The program's output is:

```
10
10
10
```

3. Close the source code file without saving.

Variables may also be declared and initialized in a single statement. For example, both of the following are valid C++ statements.

```
int i = 2;
float n = 4.5;
```

On the Net

As you have learned, when a variable is declared, it has a value that may be left over from a previously executed program or some other indeterminate value. To learn more about these indeterminate values and to understand more about why this behavior occurs, see http://www.ProgramCPP.com. See topic 5.1.1.

ARITHMETIC OPERATORS

A specific set of ***arithmetic operators*** is used to perform calculations in C++. These arithmetic operators, shown in Table 5-1, may be somewhat familiar to you. Addition and subtraction are performed with the familiar + and - operators. Multiplication uses an asterisk (*), and division uses a forward slash (/). C++ also uses what is known as a modulus operator (%) to determine the integer remainder of division. A more detailed discussion of the modulus operator is presented later in this section.

SYMBOL	OPERATION	EXAMPLE	READ AS...
+	Addition	3 + 8	three plus eight
-	Subtraction	7 - 2	seven minus two
*	Multiplication	4 * 9	four times nine
/	Division	6 / 2	six divided by two
%	Modulus	7 % 3	seven modulo three

TABLE 5-1

USING ARITHMETIC OPERATORS

The arithmetic operators are used with two operands, as in the examples in Table 5-1. The exception to this is the minus symbol which can be used to change the sign of an operand. Arithmetic operators are most often used on the right side of an assignment operator, like the examples in Table 5-2. The portion of the statement on the right side of the assignment operator is called an *expression*.

STATEMENT	RESULT
`cost = price + tax;`	*cost* is assigned the value of *price* plus *tax*.
`owed = total - discount;`	*owed* is assigned the value of *total* minus *discount*.
`area = l * w;`	*area* is assigned the value of *l* times *w*.
`one_eighth = 1 / 8;`	*one_eighth* is assigned the value of 1 divided by 8.
`r = 5 % 2;`	*r* is assigned the remainder of 5 divided by 2 by using the modulus operator.
`x = -y;`	*x* is assigned the value of *-y*.

TABLE 5-2

The assignment operator (=) functions differently in C++ than the equal sign functions in algebra. Consider the following statement:

```
x = x + 10;
```

The statement above is invalid for use in algebra because the equal sign is the symbol around which both sides of an equation are balanced. The left side equals the right side. But your C++ compiler looks at the statement differently. The expression on the right side of the equal sign is evaluated, and the result is *stored* in the variable to the left of the equal sign. In the statement above, the value of **x** is increased by 10.

EXERCISE 5-3 USING ARITHMETIC OPERATORS

1. Retrieve the file named *ASSIGN.CPP*.
2. Look at the source code and try to predict the program's output.
3. Run the program and see if you were correct in your prediction.
4. Close the source code file.

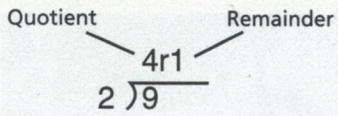

FIGURE 5-1
The division operator and modulus operator return the quotient and the remainder.

MORE ABOUT MODULUS

The *modulus operator*, which may be used only for integer division, returns the remainder rather than the result of the division. As shown in Figure 5-1, integer division is similar to the way you divide manually.

When integer division is performed, any fractional part that may be in the answer is lost when the result is stored into the integer variable. The modulus operator allows you to obtain the fractional part of the result as an integer remainder.

Consider the program in Figure 5-2. The user is prompted for two integers. Notice the program calculates the quotient using the division operator (/) and the remainder using the modulus operator (%).

```
// REMAIN.CPP
#include <iostream.h> // necessary for cin and cout commands

main()
{
  int dividend, divisor, quotient, remainder;

  // Get the dividend and divisor from the user.
  cout << "Enter the dividend ";
  cin >> dividend;
  cout << "Enter the divisor ";
  cin >> divisor;

  // Calculate the quotient and remainder
  quotient = dividend / divisor;
  remainder = dividend % divisor;

  // Output the quotient and remainder
  cout << "The quotient is " << quotient;
  cout << " with a remainder of " << remainder << '\n';
  return 0;
}
```

FIGURE 5-2
This program uses the modulus operator.

EXERCISE 5-4 THE MODULUS OPERATOR

1. Enter, compile, and run the program from Figure 5-2. Save the source file as *REMAIN.CPP*.
2. Run the program several times using values that will produce different remainders.
3. Leave the source file open for the next exercise.

USING OPERATORS IN OUTPUT STATEMENTS

The program in Figure 5-2 required four variables and nine program statements. The program in Figure 5-3 accomplishes the same output with only two variables and seven statements. Notice in Figure 5-3 that the calculations are performed in the output statements. Rather than storing the results of the expressions in variables, the program sends the results to the screen as part of the output.

```
// REMAIN.CPP
#include <iostream.h> // necessary for cin and cout commands

main()
{
  int dividend, divisor;

  // Get the dividend and divisor from the user.
  cout << "Enter the dividend ";
  cin >> dividend;
  cout << "Enter the divisor ";
  cin >> divisor;

  // Output the quotient and remainder
  cout << "The quotient is " << dividend/divisor;
  cout << " with a remainder of " << dividend % divisor << '\n';
  return 0;
}
```

FIGURE 5-3 Operators can be used in an output statement.

Avoid including the calculations in the output statements if you need to store the quotient or remainder and use them again in the program. But in situations like this, it is perfectly fine to use operators in the output statement.

EXERCISE 5-5 OPERATORS IN OUTPUT STATEMENTS

1. Modify the program on your screen to match Figure 5-3. Save the source file as *REMAIN2.CPP*. Compile it and run it.
2. Test the program to make sure you are still getting the same results.
3. Leave the source file open for the next exercise.

DIVIDING BY ZERO

Dividing by zero in math is without a practical purpose. The same is true with computers. In fact, division by zero always generates some type of error message.

EXERCISE 5-6 DIVISION BY ZERO

1. Run the program on your screen again. Enter zero for the divisor and see what error message is generated.
2. Close the source code file.

SPACES AROUND OPERATORS

Your C++ compiler ignores blank spaces. Both of the statements shown below are valid.

```
x=y+z;
x = y + z;
```

The only time you must be careful with spacing is when using the minus sign to change the sign of a variable or number. For example, `x = y - -z;` is perfectly fine. The sign of the value in the variable `z` is changed and then it is subtracted from `y`. If you failed to include the space before the `-z`, you would have created a problem because two minus signs together (--) has another meaning, which is examined later in this chapter.

COMPOUND OPERATORS

Often you will write statements which have the same variable on both sides of the assignment operator. For example, you may write a statement like `x = x + 2;` which adds 2 to the variable `x`. In cases like this, you may wish to use special operators, called ***compound operators,*** available in C++ that provide a shorthand notation for writing such statements.

Table 5-3 lists the compound operators, gives an example of each, and shows the longhand equivalent.

COMPOUND OPERATOR	EXAMPLE	LONGHAND EQUIVALENT
`+=`	`i += 1;`	`i = i + 1;`
`-=`	`j -= 12;`	`j = j - 12;`
`*=`	`z *= 5.25;`	`z = z * 5.25;`
`/=`	`w /= 2;`	`w = w / 2;`
`%=`	`d %= 3;`	`d = d % 3;`

TABLE 5-3

If you want to use a compound operator on a statement such as the one below, be careful.

```
price = price - discount + tax + shipping;
```

Because the compound operators have a lower precedence in the order of operations (see Appendix B), you may get unexpected results. For example, simplifying the statement as shown below results in the wrong value because `discount + tax + shipping` is calculated and the sum of those are subtracted from `price`.

```
price -= discount + tax + shipping;  // this gives the incorrect result!
```

In this case, you can work around the problem by using parentheses and changing the compound operator to +=. To properly calculate the discount, the unary operator (-) is used to subtract the discount by adding it as a negative value.

```
price += (-discount + tax + shipping);
```

The complexity of dealing with the order of operations on statements with more than one value on the right side of the operator sometimes makes the shorthand notation more trouble than it's worth. Until you are comfortable with the problems presented by the order of operators, you may want to use compound operators with simple statements only.

On the Net

To learn more about the use of compound operators, and to see other related operators, see http://www.ProgramCPP.com. See topic 5.1.2.

SECTION 5.1 QUESTIONS

1. What is the assignment operator?
2. What does the modulus operator do?
3. Write a statement that stores the remainder of dividing the variable **i** by **j** in a variable named **k**.
4. Write a statement that calculates the volume of a box (V = abc) given the dimensions of the box. Use appropriate identifiers for the variables. You do not have to declare types.
5. Write a statement that calculates the sales tax (use 7%) for an item given the cost of the item and the tax rate. Store the value in a variable named **tax_due**. Use appropriate identifiers for the other variables. You do not have to declare types.

PROBLEM 5.1.1

Write a program that uses the statement you wrote in question 4 above in a complete program that calculates the volume of a box. Save the source code file as *VOLBOX.CPP*.

PROBLEM 5.1.2

Write a program that uses the statement you wrote in question 5 above in a complete program that calculates the sales tax for an item given the cost of the item and the tax rate. Save the source code file as *SALESTAX.CPP*.

Counting by One and the Order of Operations

In this section, you will learn two operators that allow you to increase or decrease the value stored in an integer variable by one. You will also learn the order that the computer applies operators in an expression.

INCREMENTING AND DECREMENTING

Adding or subtracting 1 from a variable is very common in programs. Adding 1 to a variable is called ***incrementing*** and subtracting 1 from a variable is called ***decrementing.*** You will have the need to increment or decrement a variable when a program must execute a section of code a specified number of times or when you need to count the number of times a process has been repeated.

THE ++ AND -- OPERATORS

C++ provides operators for incrementing and decrementing. In C++, you can increment an integer variable using the ++ ***operator,*** and decrement using the -- ***operator,*** as shown in Table 5-4.

TABLE 5-4

STATEMENT	EQUIVALENT TO...
`counter++;`	`counter = counter + 1;`
`counter--;`	`counter = counter - 1;`

C = C + 1 or C++

The ++ operator is also part of the C programming language which was invented before C++. The ++ operator was made part of the new language's name because it was the next step after the C language.

Note

The ++ and -- operators can be used with any arithmetic data type, which includes all of the integer and floating-point types.

You can increment or decrement a variable without the ++ or -- operators. For example, instead of `j++` you can use `j = j + 1`. With today's compilers, either way will produce efficient machine code. Some of the early C and C++ compilers translated the ++ and -- operators into more efficient machine language than would be generated by the addition or subtraction operator. Now, however, compilers optimize the machine code and will recognize that `j = j + 1` is the equivalent of `j++`.

EXERCISE 5-7

USING ++ AND --

1. Retrieve the file *INC_DEC.CPP*.
2. Compile and run the program.
3. Examine the output and leave the source code file open for the next exercise.

On the Net

Computers can increment and decrement numbers more quickly than they can add to or subtract from numbers. To learn why, go to http://www.ProgramCPP.com. See topic 5.2.1.

VARIATIONS OF INCREMENT AND DECREMENT

At first glance, the ++ and -- operators seem very simple. But there are two ways each of these operators can be used. The operators can be placed either before or after the variable. The way you use the operators changes the way they work.

Used in a statement by themselves, the ++ and -- operators can be placed before or after the variable. For example, the two statements shown below both increment whatever value is in `j`.

```
j++;
++j;
```

The difference in where you place the operator becomes important if you use the ++ or -- operator in a more complex expression, or if you use the operators in an output statement. First let's look at how the placement of the operators affects the following statement. Assume that `j` holds a value of 10.

```
k = j++;
```

In the case of the statement above, `k` is assigned the value of the variable `j` *before* `j` is incremented. Therefore, the value of 10 is assigned to `k` rather than the new value of `j`, which is 11. If the placement of the ++ operator is changed to precede the variable `j` (for example `k = ++j;`), then `k` is assigned the value of `j` *after* `j` is incremented to 11.

EXERCISE 5-8 OPERATOR PLACEMENT

1. Add a statement to the file named *INC_DEC.CPP* that declares `k` as a variable of type int.
2. Add the following lines to the program on your screen before the closing brace.

```
k = j++;
cout << "k = " << k << '\n';
cout << "j = " << j << '\n';
k = ++j;
cout << "k = " << k << '\n';
cout << "j = " << j << '\n';
```

3. Save the new source code file as *INC_DEC2.CPP*.
4. Compile and run the program to see the new output.
5. Close the source code file.

The increment and decrement operators can be used to reduce the number of statements necessary in a program. Consider the two statements below.

```
attempts = attempts + 1;
cout << "This is attempt number " << attempts << '\n';
```

Using the increment operator, one statement can do the job of two.

```
cout << "This is attempt number " << ++attempts << '\n';
```

In both cases, the variable named **attempts** is incremented and its value is printed to the screen. In the second example, the variable is incremented within the cout statement. The ++ operator must come before the variable named **attempts**. Otherwise, the value of the variable will be printed before it is incremented.

Note

Remember, the ++ operator actually changes the variable. If you replaced `++attempts` *with* `attempts + 1` *in the cout statement, the value of the variable would remain the same. The screen, however, would print the result of* `attempts + 1`.

ORDER OF OPERATIONS

You may recall the rules related to the order in which operations are performed from your math classes. These rules are called the ***order of operations.*** The C++ compiler uses a similar set of rules for its calculations.

1. Minus sign used to change sign (-)
2. Multiplication and division (* / %)
3. Addition and subtraction (+ -)

C++ lets you use parentheses to override the order of operations. For example, consider the two statements in Figure 5-4.

There are additional operators you will learn about later. To see how they fit into the order of operations, see Appendix B for a complete table of the order of operators.

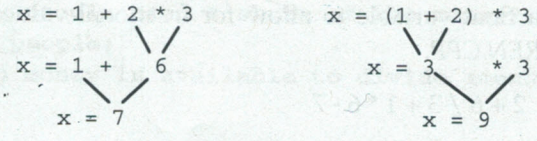

FIGURE 5-4 Parentheses can be used to change the order of operations.

EXERCISE 5-9 ORDER OF OPERATIONS

1. Load the program ORDER.CPP.
2. Look at the source code and try to predict the program's output.
3. Run the program and see if your prediction is correct.
4. Close the source code file.

SECTION 5.2 QUESTIONS

1. Write a statement that increments a variable `m` using the increment operator.
2. If the value of `i` is 10 before the statement below is executed, what is the value of `j` after the statement?

```
j = i++;
```

3. If the value of `i` is 4 before the statement below is executed, what is the value of `j` after the statement?

```
j = --i;
```

4. What will the value of `j` be after the following statement is executed?

```
j = 3 + 4 / 2 + 5 * 2 - 3;
```

5. What will the value of `j` be after the following statement is executed?

```
j = (3 + 4) / (2 + 5) * 2 - 3;
```

PROBLEM 5.2.1

1. Write a program that declares an integer named **up_down** and initializes it to 3.
2. Have the program print the value of **up_down** to the screen.
3. Have the program increment the variable and print the value to the screen.
4. Add statements to the program to decrement the variable and print the value to the screen again.
5. Save the source code as *UPDOWN.CPP*, compile it, and run it.

PROBLEM 5.2.2

Write a program that evaluates the following expressions and prints the different values that result from the varied placement of the parentheses. Store the result in a float variable to allow for fractional values. Save the source code file as *PAREN.CPP*.

2 + 6 / 3 + 1 * 6 - 7

(2 + 6) / (3 + 1) * 6 - 7

(2 + 6) / (3 + 1) * (6 - 7)

How Data Types Affect Calculations

C++ allows you to mix data types in calculations (such as dividing a float value of 125.25 by an integer like 5). Although you should avoid it whenever possible, this section will show you the way to do it should you need to.

You learned in Chapter 4 that each data type is able to hold a specific range of values. When performing calculations, the capacity of your variables must be kept in mind. It is possible for the result of an expression to be too large or too small for a given data type.

MIXING DATA TYPES

Sometimes you may need to perform operations that mix data types but doing so is less than desirable. In fact, many programming languages do not allow it. But if you have to mix data types, it is important that you understand how to do it properly.

C++ can automatically handle the mixing of data types (called promotion), or you can direct the compiler on how to handle the data (called typecasting).

PROMOTION

Consider the program in Figure 5-5. The variable **number_of_people** is an integer. The other variables involved in the calculation are floating-point numbers. Before you mix data types, you should understand the way the compiler is going to process the variables.

```
// SHARE.CPP
#include <iostream.h>

main()
{
  int number_of_people;  // declare number_of_people as an integer
  float money;           // declare money as a float
  float share;           // declare share as a float

  cout << "How many people need a share of the money? ";
  cin >> number_of_people;
  cout << "How much money is available to divide among the people? ";
  cin >> money;

  share = money / number_of_people;

  cout << "Give each person $" << share << '\n';
  return 0;
}
```

FIGURE 5-5
This program uses mixed data types in its calculation.

In cases of mixed data types, the compiler makes adjustments so as to produce the most accurate answer. In the program in Figure 5-5, for example, the integer value is temporarily converted to a float so that the fractional part of the variable **money** can be used in the calculation. This is called ***promotion.*** The variable called **number_of_people** is not actually changed. Internally, the computer treats the data as if it were stored in a float. But after the calculation, the variable is still an integer.

EXERCISE 5-10 MIXING DATA TYPES

1. Retrieve the file named *SHARE.CPP*. The program from Figure 5-5 appears in your editor.
2. Compile and run the program and observe how the mixed data types function.
3. Leave the source code file open for the next exercise.

Promotion of the data type can occur only while an expression is being evaluated. Consider the program in Figure 5-6.

The variable **i** is promoted to a float when the expression is calculated, which gives the result 1.5. But then the result is stored in the integer variable **answer**. You are unable to store a floating-point number in space reserved for an integer variable. The floating-point number is ***truncated***, which means the digits after the decimal point are dropped. The number in **answer** is 1, which is not correct.

Note

Truncation is the equivalent of chopping off everything to the right of the decimal point. When a number is truncated, 1.00001 becomes 1 and 1.999999 also becomes 1.

TYPECASTING

Even though C++ handles the mixing of data types fairly well, unexpected results can occur. C++ allows you to explicitly change one data type to another using operators called ***typecast operators.*** Using a typecast operator is usually referred to as ***typecasting.***

Consider the program on your screen (*SHARE.CPP*). The calculated value in the variable **share** is of type float. If you are interested only in even dollar amounts, you can force the compiler to interpret the variable **money** as an integer data type by typecasting.

```
#include <iostream.h>

main()
{
  int answer, i;
  float x;

  i = 3;
  x = 0.5;
  answer = x * i;

  cout << answer << '\n';
  return 0;
}
```

FIGURE 5-6
Mixing data types can cause incorrect results.

To typecast a variable, simply supply the name of the data type you want to use to interpret the variable, followed by the variable placed in parentheses. The statement below, for example, typecasts the variable **diameter** to a float.

```
C = PI * float(diameter);
```

In cases where the data type to which you want to typecast is more than one word (for example long double), place both the data type and the variable in parentheses as shown in the example below.

```
C = PI * (long double)(diameter);
```

On the Net

There is more than one way to typecast variables in C++: an older style, which was used in C; the style you are using in this chapter; and a new style that is not yet supported by all C++ compilers. To learn more about typecasting and to see examples of all three styles, see http: //www.ProgramCPP.com. See topic 5.3.1.

EXERCISE 5-11 TYPECASTING

1. Change the type of **share** to int.
2. Change the calculation statement to read as follows.

```
share = int (money) / number_of_people;
```

3. Compile and run the program again.
4. Save the source code file and close it.

There are a number of ways to accomplish what was done in Exercise 5-11. The purpose of the exercise is to show you how to use the typecast operator in case you ever need it.

OVERFLOW

Overflow is the condition where an integer becomes too large for its data type. The program in Figure 5-7 shows a simple example of overflow. The expression **j = i + 2000;** results in a value of 34000, which is too large for the short data type.

An overflow condition may be difficult to identify. Figure 5-8 shows a program very similar to the one in Figure 5-7. In Figure 5-8, 34000 is divided by 2 before it is stored in the short data type. The result of 34000/2 is 17000, which is within the range of a short type. Although everything appears to be fine, a problem exists.

Because the short type is used in the calculation, even an intermediate result that exceeds the range of the short type (-32768 to 32767) will cause an overflow that results in an incorrect answer.

```
#include <iostream.h> // necessary for cout command

main()
{
  short i,j;

  i = 32000;
  j = i + 2000;  // The result (34000) overflows the short int type
  cout << j << '\n';
  return 0;
}
```

FIGURE 5-7
The result of the expression will not fit in the variable `j`.

```
#include <iostream.h> // necessary for cout command

main()
{
  short i,j;

  i = 32000;
  j = (i + 2000) / 2;
  cout << j << '\n';
  return 0;
}
```

FIGURE 5-8
What is wrong with this program?

EXERCISE 5-12 OVERFLOW

1. Enter, compile, and run the program in Figure 5-7. Notice that the calculation resulted in an overflow and an incorrect result.
2. Change the calculation statement to match Figure 5-8. Compile and run to see the result of the intermediate overflow.
3. Change the data type from short to long. Compile and run again. This time the result should not overflow.
4. Close the source file. There is no need to save.

UNDERFLOW

Underflow is similar to overflow. ***Underflow*** occurs with floating-point numbers when a number is too small for the data type. For example, the number 1.5×10^{-44} is too small to fit in a standard float variable. A variable of type double, however, can hold the value.

FLOATING-POINT ROUNDING ERRORS

Using floating-point numbers can produce incorrect results if you fail to take the precision of floating-point data types into account.

When working with very large or very small floating-point numbers, you can use a form of exponential notation, called ***"E" notation,*** in your programs. For example, the number 3.5×10^{20} can be represented as 3.5e20 in your program.

You must keep the precision of your data type in mind when working with numbers in "E" notation. Look at the program in Figure 5-9.

At first glance, the two calculation statements appear simple enough. The first statement adds 3.9×10^{10} and 500. The second one subtracts the 3.9×10^{10} which should leave the 500. The result assigned to **y**, however, is not 500. Actual values vary depending on the compiler, but the result is incorrect whatever the case.

The reason is that the float type is not precise enough for the addition of the number 500 to be included in its digits of precision. So when the larger value is subtracted, the result is not 500 because the 500 was never properly added.

```
// FLOATERR.CPP
#include <iostream.h> // necessary for cout command

main()
{
  float x,y;

  x = 3.9e10 + 500.0;
  y = x - 3.9e10;

  cout << y << '\n';
  return 0;
}
```

FIGURE 5-9
The precision of floating-point data types must be considered to avoid rounding errors.

EXERCISE 5-13 FLOATING-POINT PRECISION

1. Enter, compile, and run the program in Figure 5-9. See that the result in the variable **y** is not 500.
2. Change the data type of **x** and **y** to double and run again. The increased precision of the double data type should result in the correct value in **y**.
3. Save the source code file as *FLOATERR.CPP* and close the source code file.

SECTION 5.3 QUESTIONS

1. What is the term that means the numbers to the right of the decimal point are removed?

2. What is the typecast operator that changes a variable to an int?
3. Define overflow.
4. What is "E" notation?
5. How would you write 6.9×10^8 in "E" notation?

KEY TERMS

++ operator	modulus operator
-- operator	order of operations
arithmetic operators	overflow
assignment operator	promotion
compound operators	truncate
decrementing	typecast operators
"E" notation	typecasting
expression	underflow
incrementing	

SUMMARY

- The assignment operator (=) changes the value of the variable to the left of the operator to the result of the expression to the right of the operator.
- You can initialize multiple variables to the same value in a single statement.
- The arithmetic operators are used to create expressions.
- The modulus operator (%) returns the remainder of integer division.
- Expressions can be placed in output statements.
- Dividing by zero generates an error in C++.
- Spaces can be placed around all operators, but are not required in most cases.
- The ++ and -- operators increment and decrement arithmetic variables respectively.
- The placement of the ++ and -- operators becomes important when the operators are used as part of a larger expression or in an output statement.
- C++ calculations follow an order of operations.
- C++ allows data types to be mixed in calculations, but it should be avoided. When C++ is allowed to handle mixed data types automatically, variables are promoted to other types. You can explicitly change data types using typecasting.
- Overflow, underflow, and floating-point rounding errors can occur if you are not aware of the data types used in calculations.

PROJECTS

PROJECT 5-1 • TRANSPORTATION

Suppose you have a group of people who need to be transported on buses and vans. You can charter a bus only if you can fill it. Each bus holds 50 people. You must provide vans for the 49 or fewer people who will be left over after you charter buses. Write a program that accepts a number of people and determines how many buses must be chartered and reports the number of people left over that must be placed on vans. *Hint:* Use the modulus operator to determine the number of people left over.

PROJECT 5-2 • MATHEMATICS

Write a program that calculates the area of an ellipse. Locate the formula for the calculation, prompt the user for the required inputs, and output the area.

PROJECT 5-3 • MATHEMATICS

Write a program that converts degrees to radians using the formula below.

radians = degrees / 57.3

PROJECT 5-4 • SPEED

Write a program that converts miles per hour to feet per second.

PROJECT 5-5 • FINANCE

Write a program that calculates simple interest (I = PRT) given the principal, the interest rate, and a period in years.

PROJECT 5-6 • ELECTRICITY

Write a program that calculates the current through a resistor given the voltage drop across the resistor and the resistance in ohms. Use the formula I = V/R, where V is voltage, R is resistance in ohms, and I is current in amps.

PROJECT 5-7 • PHYSICS

Write a program that calculates force (F=ma), where F = force in Newtons, m = mass in kilograms, and a = acceleration in meters/second.

PROJECT 5-8 • PHYSICS

Write a program that uses Einstein's $e=mc^2$ equation to find the amount of energy for a given mass.

PROJECT 5-9 • LAW ENFORCEMENT

Write a program that calculates the fine for a speeding ticket. Choose a base fine for violating the speed limit and an additional fine for each mile per hour over the speed limit.

PROJECT 5-10 • SURVEYING

Steel measuring tapes vary in length slightly depending on the temperature. When they are manufactured, they are standardized for 20 degrees Celsius (68 degrees Fahrenheit). As the temperature varies above or below 20 degrees C, the

tape becomes slightly inaccurate, which must be taken into consideration. The formula below will produce a length correction given the length measured by the tape and the temperature in Celsius. (T = temperature, L = measured length)

$$C = 0.0000116 * (T - 20) * L$$

The adjusted length can be calculated using the following formula.

$$\text{new length} = L + C$$

Write a program that asks the user for a measured length and temperature in Celsius, and outputs an adjusted length using the formulas above.

Source of formulas: Elementary Surveying, Seventh Edition, Brinker & Wolf, Harper & Row, 1984.

Computer Graphics

By Steve Ferrera

In recent years, computers have changed the lives of many people through their capabilities of generating graphics. Although most people have seen and interacted with computer graphics, most have little or no understanding of the technology behind the pictures. It is helpful, however, to have some knowledge of computer graphics and to be familiar with the methods and terms used when dealing with them.

Computer graphics are pictorial representations of information. Their application can range from a graph of a company's earnings to a maze for a video game. Computers can store many graphic images which can later be used and manipulated by the user. In order to produce these graphics, a user needs a computer, a graphics software program, and the appropriate input and output devices. A common personal computer system, with a keyboard and a color monitor, can run fairly sophisticated graphics programs. Other input and output devices, including electronic sketchpads, video digitizers, video cameras, printers, and color plotters, can increase a computer's graphics capabilities.

Computer graphics utilize particular hardware and software technologies. On the hardware side, a great deal of time has been devoted to improving video displays. Most computer graphics are presented on a monitor, or Video Display Terminal, in the form of a cathode-ray tube, or CRT for short. The display method most commonly used is called raster-scan.

In a raster-scan CRT, an electron beam passes across the screen horizontally many times per second, generating an image that consists of a two-dimensional grid of dots. All of these dots together are known as a bit map and are closely spaced and arranged in rows and columns to produce an image. Each of these dots, called pixels, may be manipulated in color and intensity. As the number of pixels produced increases, the image resolution becomes finer and more distinct. However, the higher this resolution becomes, the greater is the amount of memory required to store the image.

On the software side, graphics programs apply special display algorithms, or internal data processing procedures, to form realistic images. For example, fractal algorithms create computer-generated geometric images. They are suitable for producing and analyzing irregular patterns and shapes in nature, such as clouds, mountains, and coastlines. Algorithms based on statistical probabilities, called fuzzy sets, are helpful in generating images of natural phenomena. An iterative algorithm can duplicate the same image many times, only making small changes in each successive image. Hidden-surface removal algorithms can delete lines and surfaces from unseen parts of an image. Colorizing can give a colorless image a wide range of colors.

Using computer-aided design (CAD) systems, images can be elaborately detailed and easily changed and manipulated, allowing them to be observed from any angle. Researchers use computer graphics for such purposes as producing three-dimensional perspectives of the human brain or studying how pollutants interact with certain meteorological phenomena. Besides the exploration of 3-D possibilities in a virtual reality, there is also an increasing application for education. For example, flight simulators help pilots and astronauts in training, and visualizations of complex mathematical equations have assisted students in learning.

Few industries, however, have taken advantage of the computer's ability to create and manipulate graphic images the way the entertainment industry has. Video games offer increasingly realistic play, and computers create special effects for movies.

Improved capabilities of personal computers and their programs have allowed consumers access to powerful graphics tools. Paint programs enable users to make a drawing and then apply

color to it. Draw programs allow the creation of two- and three-dimensional images. Animation tools let users make action sequences automatically. Photographs, drawings, and video clips can be digitized and introduced into many graphics programs. Often computer users have collections of stored images called clip art, which can be used in the documents and images created by the user.

As users become familiar with computer graphics, their uses and applications will grow. For example, the charting abilities of spreadsheet software allow a person using a personal computer to create sophisticated charts and graphs. Other software helps computer users produce complicated graphics such as three-dimensional animation. With a good understanding of computer graphics and the methods and terms involved in creating graphics, users have an almost infinite range of possibilities.

Steve Ferrera is a Junior at Fenwick High School, Oak Park, Illinois.