Using C++ Math Operators and Precedence

If you are dreading this chapter because you don't like math—relax, C++ does all your math for you! It is a misconception that you have to be good at math to understand how to program computers. In fact, programming practice assumes the opposite is true! Your computer is your "slave," to follow your instructions, and to do all the calculations for you. This chapter explains how C++ computes by introducing you to

- Primary math operators
- Order of operator precedence
- Assignment statements
- Mixed data type calculations
- Type casting

Many people who dislike math actually enjoy learning how the computer handles it. After learning the math operators and a few simple ways in which C++ uses them, you should feel comfortable using calculations in your programs. Computers are fast, and they can perform math operations much faster than you can!

C++'s Primary Math Operators

A C++ *math operator* is a symbol used for adding, subtracting, multiplying, dividing, and other operations. C++ operators are not always mathematical in nature, but many are. Table 8.1 lists these operator symbols and their primary meanings.

Table 8.1. C++ primary operators.

Symbol	*Meaning*
`*`	Multiplication
`/`	Division and Integer Division
`%`	Modulus or Remainder
`+`	Addition
`-`	Subtraction

Most of these operators work in the familiar way you expect them to. Multiplication, addition, and subtraction produce the same results (and the division operator *usually* does) as those produced with a calculator. Table 8.2 illustrates four of these simple operators.

Table 8.2. Typical operator results.

Formula	*Result*
`4 * 2`	8
`64 / 4`	16
`80 - 15`	65
`12 + 9`	21

Table 8.2 contains examples of *binary operations* performed with the four operators. Don’t confuse binary operations with *binary numbers*. When an operator is used between two literals, variables, or a combination of both, it is called a *binary operator* because it operates using two values. When you use these operators (when assigning their results to variables, for example), it does not matter in C++ whether you add spaces to the operators or not.

> **CAUTION:** For multiplication, use the asterisk (*), *not* an `x` as you might normally do. An `x` cannot be used as the multiplication sign because C++ uses `x` as a variable name. C++ interprets `x` as the value of a variable called `x`.

The Unary Operators

A *unary operator* operates on, or affects, a single value. For instance, you can assign a variable a positive or negative number by using a unary + or –.

Examples

1. The following section of code assigns four variables a positive or a negative number. The plus and minus signs are all unary because they are not used between two values.

   The variable `a` *is assigned a negative* `25` *value.*
   The variable `b` *is assigned a positive* `25` *value.*
   The variable `c` *is assigned a negative* `a` *value.*
   The variable `d` *is assigned a positive* `b` *value.*

```
a = -25;// Assign ‘a’ a negative 25.
b = +25;// Assign ‘b’ a positive 25 (+ is not needed).
c = -a; // Assign ‘c’ the negative of ‘a’ (-25).
d = +b; // Assign ‘d’ the positive of ‘b’ (25, + not needed).
```

2. You generally do not have to use the unary plus sign. C++ assumes a number or variable is positive, even if it has no plus sign. The following four statements are equivalent to the previous four, except they do not contain plus signs.

```
a = -25;   // Assign ‘a’ a negative 25.
b =  25;   // Assign ‘b’ a positive 25.
c = -a;    // Assign ‘c’ the negative of ‘a’ (-25).
d =  b;    // Assign ‘d’ the positive of ‘b’ (25).
```

3. The unary negative comes in handy when you want to negate a single number or variable. The negative of a negative is positive. Therefore, the following short program assigns a negative number (using the unary –) to a variable, then prints the negative of that same variable. Because it had a negative number to begin with, the `cout` produces a positive result.

```
// Filename: C8NEG.CPP
// The negative of a variable that contains a negative value.
#include <iostream.h>
main()
{
   signed int temp=-12;  // ‘signed’ is not needed because
                         //      it is the default.
   cout << -temp << “\n”;  // Produces a 12 on-screen.

   return 0;
}
```

The variable declaration does not need the *signed* prefix, because all integer variables are signed by default.

4. If you want to subtract the negative of a variable, make sure you put a space before the unary minus sign. For example, the following line:

```
new_temp + new_temp- -inversion_factor;
```

temporarily negates the `inversion_factor` and subtracts that negated value from `new_temp`.

Division and Modulus

The division sign, `/`, and the modulus operator, `%`, might behave in ways unfamiliar to you. They're as easy to use, however, as the other operators you have just seen.

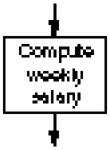
The modulus (%) computes remainders in division.

The forward slash (`/`) is always used for division. However, it produces an integer called `divide` if integer values (literals, variables, or a combination of both) appear on both sides of the slash. If there is a remainder, C++ discards it.

The percent sign (`%`) produces a *modulus*, or a *remainder*, of an integer division. It requires that integers be on both sides of the symbol, or it does not work.

Examples

1. Suppose you want to compute your weekly pay. The following program asks for your yearly pay, divides it by 52, and prints the results to two decimal places.



```
// Filename: C8DIV.CPP
// Displays user's weekly pay.
#include <stdio.h>
main()
{
   float weekly, yearly;
   printf(“What is your annual pay? “);   // Prompt user.
   scanf(“%f”, &yearly);

   weekly = yearly/52;  // Computes the weekly pay.
   printf(“\n\nYour weekly pay is $%.2f”, weekly);
   return 0;
}
```

Because a floating-point number is used in the division, C++ produces a floating-point result. Here is a sample output from such a program:

```
What is your annual pay? 38000.00
Your weekly pay is $730.77
```

Because this program used `scanf()` and `printf()` (to keep you familiar with both ways of performing input and output), the stdio.h header file is included rather than iostream.h.

2. Integer division does not round its results. If you divide two integers and the answer is not a whole number, C++ ignores the fractional part. The following `printf()`s help show this. The output that results from each `printf()` appears in the comment to the right of each line.

```
printf(“%d \n”, 10/2);      // 5  (no remainder)
printf(“%d \n”, 300/100);   // 3  (no remainder)
printf(“%d \n”, 10/3);      // 3  (discarded remainder)
printf(“%d \n”, 300/165);   // 1  (discarded remainder)
```

The Order of Precedence

Understanding the math operators is the first of two steps toward understanding C++ calculations. You must also understand the *order of precedence*. The order of precedence (sometimes called the *math hierarchy* or *order of operators*) determines exactly how C++ computes formulas. The precedence of operators is exactly the same concept you learned in high school algebra courses. (Don’t worry, this is the easy part of algebra!) To see how the order of precedence works, try to determine the result of the following simple calculation:

```
2 + 3 * 2
```

If you said 10, you are not alone; many people respond with 10. However, 10 is correct only if you interpret the formula from the left. What if you calculated the multiplication first? If you took the value of `3 * 2` and got an answer of 6, then added the 2, you receive an answer of 8—which is exactly the same answer that C++ computes (and happens to be the correct way)!

C++ performs multiplication, division, and modulus *before* addition and subtraction.

C++ always performs multiplication, division, and modulus first, then addition and subtraction. Table 8.3 shows the order of the operators you have seen so far. Of course, there are many more levels to C++’s precedence table of operators than the ones shown in Table 8.3. Unlike most computer languages, C++ has 20 levels of precedence. Appendix D, “C++ Precedence Table,” contains the complete precedence table. Notice in this appendix that multiplication, division, and modulus reside on level 8, one level higher than level 9's addition and subtraction. In the next few chapters, you learn how to use the remainder of this precedence table in your C++ programs.

Table 8.3. Order of precedence for primary operators.

Order	*Operator*
First	Multiplication, division, modulus remainder (`*`, `/`, `%`)
Second	Addition, subtraction (+, -)

Examples

1. It is easy to follow C++'s order of operators if you follow the intermediate results one at a time. The three calculations in Figure 8.1 show you how to do this.

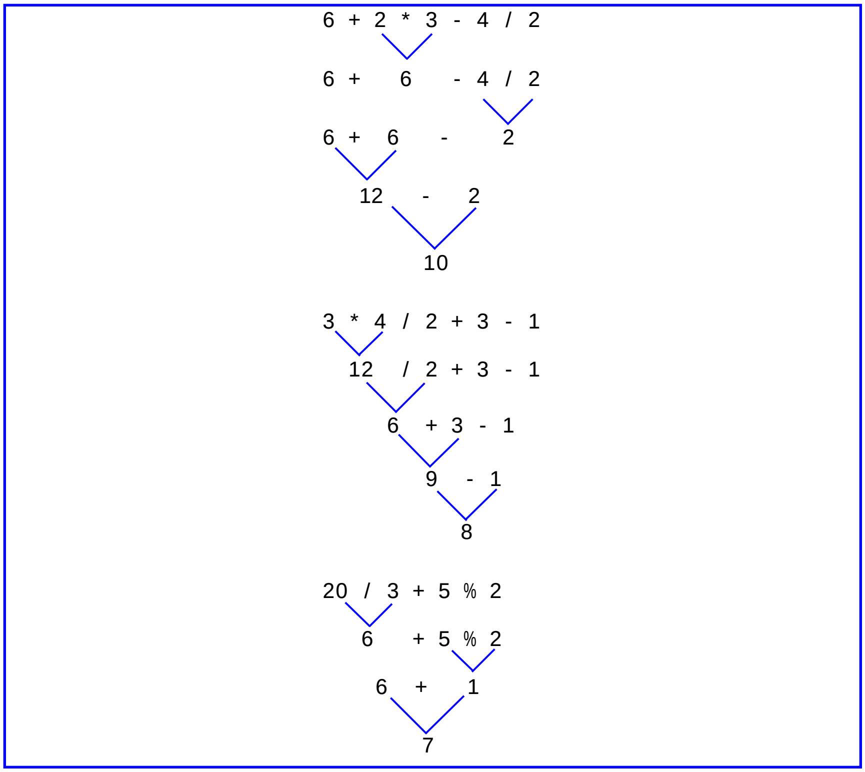


Figure 8.1. C++'s order of operators with lines indicating precedence.

2. Looking back at the order of precedence table, you might notice that multiplication, division, and modulus are on the same level. This implies there is no hierarchy on that level. If more than one of these operators appear in a calculation, C++ performs the math from the left. The same is true of addition and subtraction—C++ performs the operation on the extreme left first.

   Figure 8.2 illustrates an example showing this process.

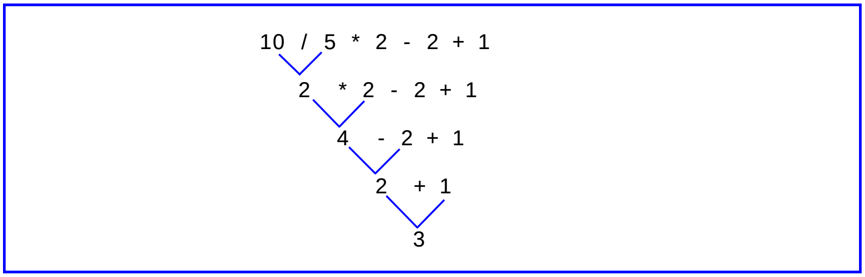


Figure 8.2. C++'s order of operators from the left, with lines indicating precedence.

Because the division appears to the left of the multiplication, it is computed first.

You now should be able to follow the order of these C++ operators. You don't have to worry about the math because C++ does the actual work. However, you should understand this order of operators so you know how to structure your calculations. Now that you have mastered this order, it's time to learn how you can override it with parentheses!

Using Parentheses

If you want to override the order of precedence, you can add parentheses to the calculation. The parentheses actually reside on a level above the multiplication, division, and modulus in the precedence table. In other words, any calculation in parentheses—whether it is addition, subtraction, division, or whatever—is always calculated before the rest of the line. The other calculations are then performed in their normal operator order.

Parentheses override the usual order of math.

The first formula in this chapter, `2 + 3 * 2`, produced an 8 because the multiplication was performed before addition. However, by adding parentheses around the addition, as in `(2 + 3) * 2`, the answer becomes 10.

In the precedence table shown in Appendix D, "C++ Precedence Table," the parentheses reside on level 3. Because they are higher than the other levels, the parentheses take precedence over multiplication, division, and all other operators.

Examples

1. The calculations shown in Figure 8.3 illustrate how parentheses override the regular order of operators. These are the same three formulas shown in the previous section, but their results are calculated differently because the parentheses override the normal order of operators.

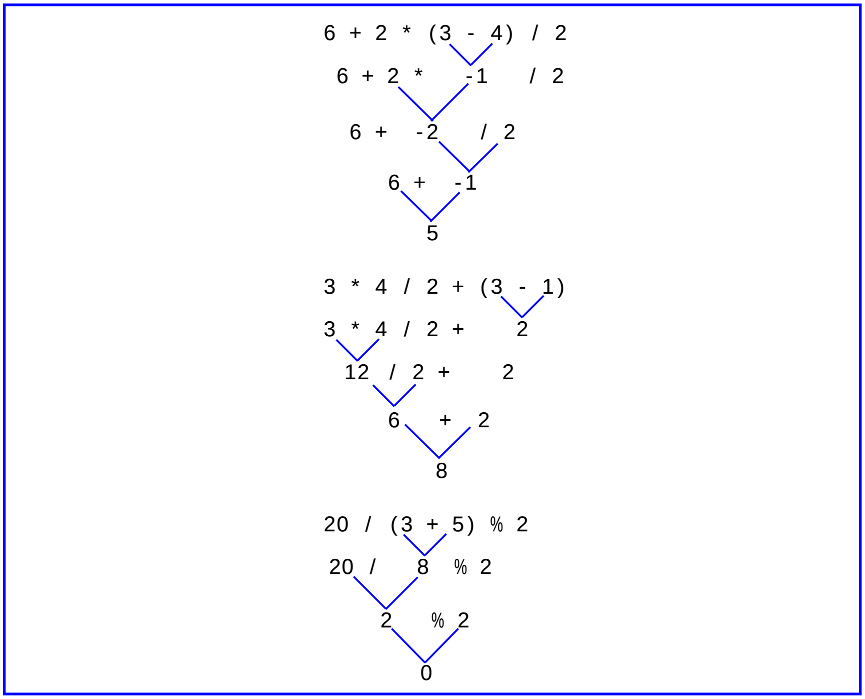


Figure 8.3. Example of parentheses as the highest precedence level with lines indicating precedence.

2. If an expression contains parentheses-within-parentheses, C++ evaluates the innermost parentheses first. The expressions in Figure 8.4 illustrate this.

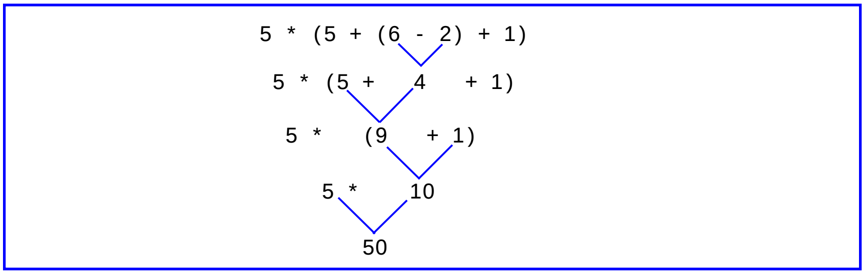


Figure 8.4. Precedence example of parentheses-within-parentheses with lines indicating precedence.

3. The following program produces an incorrect result, even though it looks as if it will work. See if you can spot the error!

Comments to identify your program.
Include the header file iostream.h so `cout` *works.*
Declare the variables `avg`, `grade1`, `grade2`, *and* `grade3` *as floating-point variables.*
The variable `avg` *becomes equal to* `grade3` *divided by 3.0 plus* `grade2` *plus* `grade1`.
Print to the screen `The average is` *and the average of the three grade variables.*
Return to the operating system.

```
// Filename: C8AVG1.CPP
// Compute the average of three grades.
#include <iostream.h>
main()
{
   float avg, grade1, grade2, grade3;

   grade1 = 87.5;
   grade2 = 92.4;
   grade3 = 79.6;
   avg = grade1 + grade2 + grade3 / 3.0;
   cout << “The average is “ << avg << “\n”;
   return 0;
}
```

The problem is that division is performed first. Therefore, the third grade is divided by 3.0 first, then the other two grades are added to that result. To correct this problem, you simply have to add one set of parentheses, as shown in the following:

```
// Filename: C8AVG2.CPP
// Compute the average of three grades.
#include <iostream.h>
main()
{
   float avg, grade1, grade2, grade3;

   grade1 = 87.5;
   grade2 = 92.4;
   grade3 = 79.6;

   avg = (grade1 + grade2 + grade3) / 3.0;
   cout << “The average is “ << avg << “\n”;
   return 0;
}
```

TIP: Use plenty of parentheses in your C++ programs to clarify the order of operators, even when you don’t have to override their default order. Using parentheses makes the calculations easier to understand later, when you might have to modify the program.

Shorter Is Not Always Better

When you program computers for a living, it is much more important to write programs that are easy to understand than programs that are short or include tricky calculations.

Maintainability is the computer industry's word for the changing and updating of programs previously written in a simple style. The business world is changing rapidly, and the programs companies have used for years must often be updated to reflect this changing environment. Businesses do not always have the resources to write programs from scratch, so they usually modify the ones they have.

Years ago when computer hardware was much more expensive, and when computer memories were much smaller, it was important to write small programs, which often meant relying on clever, individualized tricks and shortcuts. Unfortunately, such programs are often difficult to revise, especially if the original programmers leave and someone else (you!) must modify the original code.

Companies are realizing the importance of spending time to write programs that are easy to modify and that do not rely on tricks, or "quick and dirty" routines that are hard to follow. You can be a much more valuable programmer if you write clean programs with ample white space, frequent remarks, and straightforward code. Use parentheses in formulas if it makes the formulas clearer, and use variables for storing results in case you need the same answer later in the program. Break long calculations into several smaller ones.

Throughout the remainder of this book, you can read tips on writing maintainable programs. You and your colleagues will appreciate these tips when you incorporate them in your own C++ programs.

The Assignment Statements

In C++, the assignment operator, `=`, behaves differently from what you might be used to in other languages. So far, you have used it to assign values to variables, which is consistent with its use in most other programming languages.

However, the assignment operator also can be used in other ways, such as multiple assignment statements and compound assignments, as the following sections illustrate.

Multiple Assignments

If two or more equal signs appear in an expression, each performs an assignment. This fact introduces a new aspect of the precedence order you should understand. Consider the following expression:

```
a=b=c=d=e=100;
```

This might at first seem confusing, especially if you know other computer languages. To C++, the equal sign always means: Assign the value on the right to the variable on the left. This right-to-left order is described in Appendix D’s precedence table. The third column in the table is labeled *Associativity*, which describes the direction of the operation. The assignment operator associates from the right, whereas some of the other C++ operators associate from the left.

Because the assignment associates from the right, the previous expression assigns `100` to the variable named `e`. This assignment produces a value, `100`, for the expression. In C++, all expressions produce values, typically the result of assignments. Therefore, `100` is assigned to the variable `d`. The value, `100`, is assigned to `c`, then to `b`, and finally to `a`. The old values of these variables are replaced by `100` after the statement finishes.

Because C++ does not automatically set variables to zero before you use them, you might want to do so before you use the variables with a single assignment statement. The following section of variable declarations and initializations is performed using multiple assignment statements.

```
main()
{
   int ctr, num_emp, num_dep;
   float sales, salary, amount;

   ctr=num_emp=num_dep=0;
   sales=salary=amount=0;
   // Rest of program follows.
```

In C++, you can include the assignment statement almost anywhere in a program, even in another calculation. For example, consider this statement:

```
value = 5 + (r = 9 - c);
```

which is a perfectly legal C++ statement. The assignment operator resides on the first level of the precedence table, and always produces a value. Because its associativity is from the right, the `r` is assigned `9 - c` because the equal sign on the extreme right is evaluated first. The subexpression (`r = 9 - c`) produces a value (and places that value in `r`), which is then added to `5` before storing the answer in `value`.

Example

Because C++ does not initialize variables to zero before you use them, you might want to include a multiple assignment operator to do so before using the variables. The following section of code ensures that all variables are initialized before the rest of the program uses them.

```
main()
{
    int num_emp, dependents, age;
    float salary, hr_rate, taxrate;

    // Initialize all variables to zero.
    num_emp=dependents=age=hours=0;
    salary=hr_rate=taxrate=0.0;

    // Rest of program follows.
```

Compound Assignments

Many times in programming, you might want to update the value of a variable. In other words, you have to take a variable’s current value, add or multiply that value by an expression, then reassign it to the original variable. The following assignment statement demonstrates this process:

```
salary=salary*1.2;
```

This expression multiplies the old value of `salary` by 1.2 (in effect, raising the value in `salary` by 20 percent), then reassigns it to `salary`. C++ provides several operators, called *compound operators*, that you can use any time the same variable appears on both sides of the equal sign. The compound operators are shown in Table 8.4.

Table 8.4. C++'s compound operators.

Operator	*Example*	*Equivalent*
`+=`	`bonus+=500;`	`bonus=bonus+500;`
`-=`	`budget-=50;`	`budget=budget-50;`
`*=`	`salary*=1.2;`	`salary=salary*1.2;`
`/=`	`factor/=.50;`	`factor=factor/.50;`
`%=`	`daynum%=7;`	`daynum=daynum%7;`

The compound operators are low in the precedence table. They typically are evaluated last or near-last.

Examples

1. You have been storing your factory's production amount in a variable called `prod_amt`, and your supervisor has just informed you that a new addition has to be applied to the production value. You could code this update in a statement, as follows:

   ```
   prod_amt = prod_amt + 2.6;  // Add 2.6 to current production.
   ```

   Instead of using this formula, use C++'s compound addition operator by coding it like this:

   ```
   prod_amt += 2.6;  // Add 2.6 to current production.
   ```

2. Suppose you are a high school teacher who wants to raise your students' grades. You gave a test that was too difficult, and the grades were not what you expected. If you had stored each of the student's grades in variables named `grade1`, `grade2`, `grade3`, and so on, you can update the grades in a program with the following section of compound assignments.

```
grade1*=1.1;     // Increase each student's grade by 10.
percent.
grade2*=1.1;
grade3*=1.1;
// Rest of grade changes follow.
```

3. The precedence of the compound operators requires important consideration when you decide how to code compound assignments. Notice from Appendix D, “C++ Precedence Table,” that the compound operators are on level 19, much lower than the regular math operators. This means you must be careful how you interpret them.

   For example, suppose you want to update the value of a `sales` variable with this formula:

   ```
   4-factor+bonus
   ```

   You can update the `sales` variable with the following statement:

   ```
   sales = *4 - factor + bonus;
   ```

   This statement adds the quantity `4-factor+bonus` to `sales`. Due to operator precedence, this statement is not the same as the following one:

   ```
   sales = sales *4 - factor + bonus;
   ```

   Because the `*=` operator is much lower in the precedence table than `*` or `-`, it is performed last, and with right-to-left associativity. Therefore, the following are equivalent, from a precedence viewpoint:

   ```
   sales *= 4 - factor + bonus;
   ```

   and

   ```
   sales = sales * (4 - factor + bonus);
   ```

Mixing Data Types in Calculations

You can mix data types in C++. Adding an integer and a floating-point value is mixing data types. C++ generally converts the smaller of the two types into the other. For instance, if you add a double to an integer, C++ first converts the integer into a double value, then performs the calculation. This method produces the most accurate result possible. The automatic conversion of data types is only temporary; the converted value is back in its original data type as soon as the expression is finished.

C++ attempts to convert the smaller data type to the larger one in a mixed data-type expression.

If C++ converted two different data types to the smaller value's type, the higher-precision value is *truncated,* or shortened, and accuracy is lost. For example, in the following short program, the floating-point value of `sales` is added to an integer called `bonus`. Before C++ computes the answer, it converts `bonus` to floating-point, which results in a floating-point answer.

```
// Filename: C8DATA.CPP
// Demonstrate mixed data type in an expression.
#include <stdio.h>
main()
{
   int bonus=50;
   float salary=1400.50;
   float total;

   total=salary+bonus;   // bonus becomes floating-point
                         // but only temporarily.
   printf("The total is %.2f", total);
   return 0;
}
```

Type Casting

Most of the time, you don't have to worry about C++'s automatic conversion of data types. However, problems can occur if you mix unsigned variables with variables of other data types. Due to differences in computer architecture, unsigned variables do not always convert to the larger data type. This can result in loss of accuracy, and even incorrect results.

You can override C++'s default conversions by specifying your own temporary type change. This process is called *type casting.* When you type cast, you temporarily change a variable's data type from its declared data type to a new one. There are two formats of the type cast. They are

```
(data type) expression
```

and

```
data type(expression)
```

where `data type` can be any valid C++ data type, such as `int` or `float`, and the `expression` can be a variable, literal, or an expression that combines both. The following code temporarily type casts the integer variable `age` into a double floating-point variable, so it can be multiplied by the double floating-point `factor`. Both formats of the type cast are illustrated.

The variable `age_factor` *is assigned the value of the variable* `age` *(now treated like a double floating-point variable) multiplied by the variable* `factor`.

```
age_factor = (double)age * factor;    // Temporarily change age
                                      // to double.
```

The second way of type casting adds the parentheses around the variable rather than the data type, as so:

```
age_factor = double(age) * factor;    // Temporarily change age
                                      // to double.
```

NOTE: Type casting by adding the parentheses around the expression and not the data type is new to C++. C programmers do not have the option—they must put the data type in parentheses. The second method “feels” like a function call and seems to be more natural for this language. Therefore, becoming familiar with the second method will clarify your code.

Examples

1. Suppose you want to verify the interest calculation used by your bank on a loan. The interest rate is 15.5 percent, stored as .155 in a floating-point variable. The amount of interest you owe is computed by multiplying the interest rate by the amount of the loan balance, then multiplying that by the number of days in the year since the loan originated. The following program finds the daily interest rate by dividing the annual interest rate by 365, the number of days in a year. C++ must convert the integer 365 to a floating-point literal automatically, because it is used in combination with a floating-point variable.

```
// Filename: C8INT1.CPP
// Calculate interest on a loan.
#include <stdio.h>
main()
{
   int days=45;  // Days since loan origination.
   float principle = 3500.00; // Original loan amount
   float interest_rate=0.155;  // Annual interest rate
   float daily_interest;   // Daily interest rate

   daily_interest=interest_rate/365; // Compute floating-
                                     // point value.

   // Because days is int, it too is converted to float.
   daily_interest = principle * daily_interest * days;
   principle+=daily_interest;//Update principle with interest.
   printf(“The balance you owe is %.2f\n”, principle);
   return 0;
}
```

The output of this program follows:

```
The balance you owe is 3566.88
```

2. Instead of having C++ perform the conversion, you might want to type cast all mixed expressions to ensure they convert to your liking. Here is the same program as in the first example, except type casts are used to convert the integer literals to floating-points before they are used.

```
// Filename: C8INT2.CPP
// Calculate interest on a loan using type casting.
#include <stdio.h>
main()
{
   int days=45;  // Days since loan origination.
   float principle = 3500.00;   // Original loan amount
   float interest_rate=0.155;   // Annual interest rate
   float daily_interest;        // Daily interest rate

   daily_interest=interest_rate/float(365);  // Type cast days
                                             // to float.

   // Because days is integer, convert it to float also.
   daily_interest = principle * daily_interest * float(days);
   principle+=daily_interest;// Update principle with interest.
   printf(“The balance you owe is %.2f”, principle);
   return 0;
}
```

The output from this program is exactly the same as the previous one.

Review Questions

The answers to the review questions are in Appendix B.

1. What is the result for each of the following expressions?

   a. `1 + 2 * 4 / 2`

   b. `(1 + 2) * 4 / 2`

   c. `1 + 2 * (4 / 2)`

2. What is the result for each of the following expressions?

   a. `9 % 2 + 1`

   b. `(1 + (10 - (2 + 2)))`

3. Convert each of the following formulas into its C++ assignment equivalents.

   a. $$a = \frac{3 + 3}{4 + 4}$$

   b. $x = (a - b)*(a - c)2$

   c. $$f = \frac{a2}{b3}$$

   d. $$d = \frac{(8 - x2)}{(x - 9)} - \frac{(4 * 2 - 1)}{x3}$$

4. Write a short program that prints the area of a circle, when its radius equals 4 and π equals 3.14159. (*Hint:* The area of a circle is computed by π `*` radius^2.)

5. Write the assignment and `printf()` statements that print the remainder of 100/3.

Review Exercises

1. Write a program that prints each of the first eight powers of 2 (21, 22, 23,...28). Please write comments and include your name at the top of the program. Print string literals that describe each answer printed. The first two lines of your output should look like this:

```
2 raised to the first power is 2
2 raised to the second power is 4
```

2. Change C8PAY.CPP so it computes and prints a bonus of 15 percent of the gross pay. Taxes are not to be taken out of the bonus. After printing the four variables, `gross_pay`, `tax_rate`, `bonus`, and `gross_pay`, print a check on-screen that looks like a printed check. Add string literals so it prints the check-holder and put your name as the payer at the bottom of the check.
3. Store the weights and ages of three people in variables. Print a table, with titles, of the weights and ages. At the bottom of the table, print the averages.
4. Assume that a video store employee works 50 hours. He is paid $4.50 for the first 40 hours, time-and-a-half (1.5 times the regular pay rate) for the first five hours over 40, and double-time pay for all hours over 45. Assuming a 28 percent tax rate, write a program that prints his gross pay, taxes, and net pay to the screen. Label each amount with appropriate titles (using string literals) and add appropriate comments in the program.

Summary

You now understand C++'s primary math operators and the importance of the precedence table. Parentheses group operations so they can override the default precedence levels. Unlike some other programming languages, every operator in C++ has a meaning, no matter where it appears in an expression. This fact enables you to use the assignment operator (the equal sign) in the middle of other expressions.

When you perform math with C++, you also must be aware of how C++ interprets data types, especially when you mix them in the same expression. Of course, you can temporarily type cast a variable or literal so you can override its default data type.

This chapter has introduced you to a part of the book concerned with C++ operators. The following two chapters (Chapter 9, “Relational Operators,” and Chapter 10, “Logical Operators”) extend this introduction to include relational and logical operators. They enable you to compare data and compute accordingly.