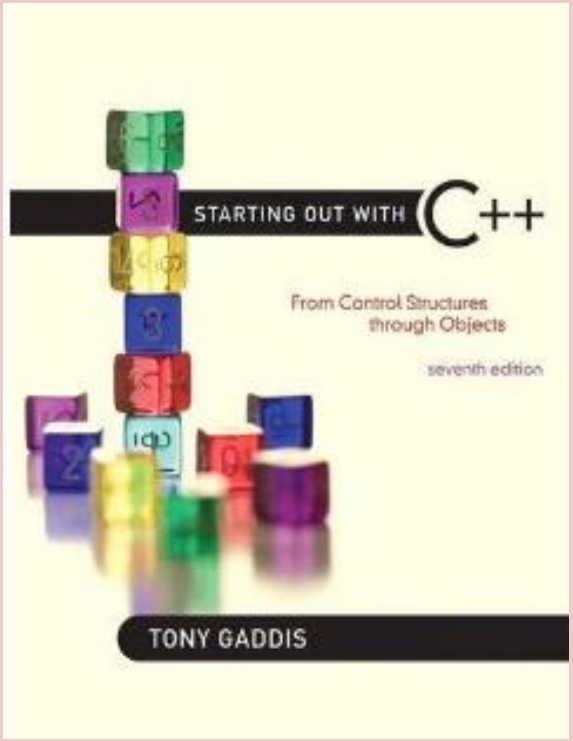
Software Design & Programming I



Starting Out with C++ (From Control Structures through Objects) 7th Edition
Written by: Tony Gaddis
Pearson - Addison Wesley
ISBN: 13-978-0-132-57625-3

Chapter 3

Introduction to C++

The cin Object

cin can be used to read data typed at the keyboard.

Just as C++ provides the cout object to produce console output, it provides an object named cin that is used to read console input. (You can think of the word cin as meaning **c**onsole **in**put.) Program 3-1 shows cin being used to read values input by the user.

Sample Program (cin)

Program 3-1

```
1  // This program calculates and displays the area of a rectangle.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int length, width, area;
8
9     cout << "This program calculates the area of a rectangle.\n";
10
11    // Have the user input the rectangle's length and width
12    cout << "What is the length of the rectangle? ";
13    cin  >> length;
14    cout << "What is the width of the rectangle? ";
15    cin  >> width;
16
17    // Compute and display the area
18    area = length * width;
19    cout << "The area of the rectangle is " << area << endl;
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
This program calculates the area of a rectangle.
What is the length of the rectangle? 10[Enter]
What is the width of the rectangle? 20[Enter]
The area of the rectangle is 200.
```

cin Object

Instead of calculating the area of one rectangle, this program can be used to compute the area of any rectangle. The values that are stored in the length and width variables are entered by the user when the program is running. Look at lines 12 and 13.

```
cout << "What is the length of the rectangle? ";
cin >> length;
```

In line 12 cout is used to display the question “What is the length of the rectangle?” This is called a *prompt*. It lets the user know that an input is expected and prompts them as to what must be entered. When cin will be used to get input from the user, it should always be preceded by a prompt.

cin Object

Gathering input from the user is normally a two-step process:

1. Use cout to display a prompt on the screen.
2. Use cin to read a value from the keyboard.

The prompt should ask the user a question, or tell the user to enter a specific value. For example, the code we just examined from Program 3-1 displays the following prompt:

What is the length of the rectangle?

This tells the user to enter the rectangle's length. After the prompt displays, the program uses cin to read a value from the keyboard and store it in the length variable.

cin Object

Notice that the << and >> operators appear to point in the direction that data is flowing. It may help to think of them as arrows. In a statement that uses cout, the << operator always points toward cout, as shown here. This indicates that data is flowing from a variable or a literal to the cout object.

cout << "What is the length of the rectangle? ";

cout ← "What is the length of the rectangle? ";

In a statement that uses cin, the >> operator always points toward the variable receiving the value. This indicates that data is flowing from the cin object to a variable.

cin >> length;

cin → length;

cin Object

The cin object causes a program to wait until data is typed at the keyboard and the [Enter] key is pressed. No other lines will be executed until cin gets its input.

When the user enters characters from the keyboard, they are temporarily placed in an area of memory called the *input buffer*, or *keyboard buffer*. cin automatically converts this data to the data type of the variable it is to be stored in. If the user types 10, it is read as the characters ‘1’ and ‘0’, but cin is smart enough to know this will have to be converted to the int value 10 before it is stored in length.

cin Object

If the user enters a floating-point number like 10.7, however, there is a problem. cin knows such a value cannot be stored in an integer variable, so it stops reading when it gets to the decimal point, leaving the decimal point and the rest of the digits in the input buffer. This can cause a problem when the next value is read in.

cin Object (floating-point Entered for integer)

Program 3-2

```
1  // This program illustrates what can happen when a
2  // floating-point number is entered for an integer variable.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int intNumber;
9     double floatNumber;
10
11    cout << "Input a number. ";
12    cin  >> intNumber;
13    cout << "Input a second number.\n";
14    cin  >> floatNumber;
15    cout << "You entered: " << intNumber
16         << " and " << floatNumber << endl;
17
18    return 0;
19 }
```

Program Output with Example Input Shown in Bold

```
Input a number. 12.3[Enter]
Input a second number.
You entered: 12 and 0.3
```

cin Object (Examples)

Entering Multiple Values

```
cin >> length >> width;
```

Enter the length and width of the rectangle separated by a space.

10 20[Enter]

- - - - - - - - - - - - - - - - - - -

Entering Multiple Values of Different Data Types

```
cin >> whole >> fractional >> letter;
```

Enter an integer, a double, and a character:

4 5.7 b[Enter]

cin Object

As you can see in the previous sample output, the values are stored in the order entered in their respective variables.

But what if the user had entered the values in the wrong order?

Entering Multiple Values of Different Data Types

```
cin >> whole >> fractional >> letter;
```

```
Enter an integer, a double, and a character:
```

5.7 4 b[Enter]

Because the data was not entered in the specified order, there is a complete mix-up of what value is stored for each variable.

cin Object

The cin statement reads **5** for int variable whole, **.7** for double variable fractional, and **4** for char variable letter. The character b is left in the input buffer. For a program to function correctly it is important that the user enter data values in the order the program expects to receive them and that a floating-point number not be entered when an integer is expected.

Mathematical Expressions

C++ allows you to construct complex mathematical expressions using multiple operators and grouping symbols.

An expression is a programming statement that has a value. Usually, an expression consists of an operator and its operands. Look at the following statement:

sum = 21 + 3;

Since 21 + 3 has a value, it is an expression. Its value, 24, is stored in the variable sum. Expressions do not have to be in the form of mathematical operations. In the following statement, 3 is an expression.

number = 3;

Mathematical Expressions

Although some instructors prefer that you not perform mathematical operations within a cout statement, it is possible to do so.

Enclosed in parenthesis

```
16
17      // Compute and display the decimal value
18      cout << "The decimal value is "<< (numerator / denominator) << endl;
```

NOTE: When sending an expression that includes an operator to cout, it is always a good idea to put parentheses around the expression. Some operators will yield unexpected results otherwise.

Operator Precedence

It is possible to build mathematical expressions with several operators. The following

statement assigns the sum of 17, x, 21, and y to the variable answer.

answer = 17 + x + 21 + y;

Some expressions are not that straightforward, however. Consider the following statement:

outcome = 12 + 6 / 3;

What value will be stored in outcome? **14** or **6**

The answer is 14 because the division operator has higher precedence than the addition operator. This is exactly the same as the operator precedence found in algebra.

Operator Precedence

Table 3-1 Precedence of Arithmetic Operators (Highest to Lowest)

( )			Expressions within parentheses are evaluated first
-		unary	Negation of a value, e.g., -6
*	/	%	binary	Multiplication, division, and modulus
+	-		binary	Addition and subtraction

The multiplication, division, and modulus operators have the same precedence. Same is true for addition and subtraction.

Table 3-2 Some Expressions

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`4 + 17 % 2 - 1`	4
`6 - 3 * 2 + 7 - 1`	6

Associativity

Associativity is the order in which an operator works with its operands. Associativity is either left to right or right to left. The associativity of the division operator is left to right, so it divides the operand on its left by the operand on its right. Table 3-3 shows the arithmetic operators and their associativity.

Table 3-3 Associativity of Arithmetic Operators

Operator	Associativity
(unary negation) -	Right to left
* / %	Left to right
+ -	Left to right

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the sum of a plus b is divided by 4.

average = (a + b) / 4;

Without the parentheses b would be divided by 4 before adding a to the result. Table 3-4 shows more expressions and their values.

Table 3-4 More Arithmetic Expressions

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(4 + 17) % 2 - 1`	0
`(6 - 3) * (2 + 7) / 3`	9

Converting Algebraic Expressions to Programming Statements

In algebra it is not always necessary to use an operator for multiplication. C++, however, requires an operator for any mathematical operation. Table 3-5 shows some algebraic expressions that perform multiplication and the equivalent C++ expressions.

Table 3-5 Algebraic and C++ Multiplication Expressions

Algebraic Expression	Operation	C++ Equivalent
6B	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
4xy	4 times x times y	`4 * x * y`

Converting Algebraic Expressions to Programming Statements

When converting some algebraic expressions to C++, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following expression:

$$x = \frac{a + b}{c}$$

To convert this to a C++ statement, a + b will have to be enclosed in parentheses:

x = (a + b) / c;

Converting Algebraic Expressions to Programming Statements

Table 3-6 Algebraic and C++ Expressions

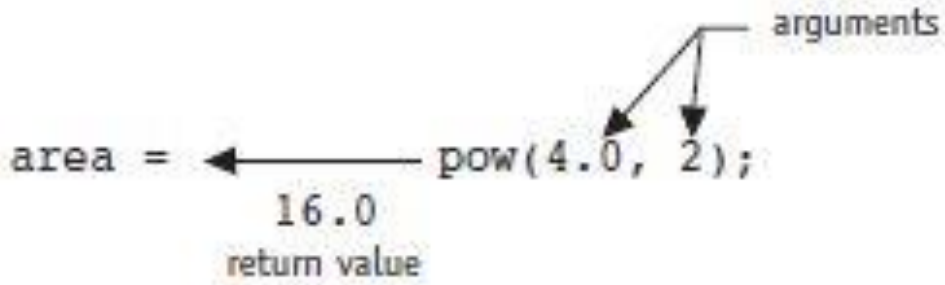
Algebraic Expression	C++ Expression
$y = 3\frac{x}{2}$	`y = x / 2 * 3;`
$z = 3bc + 4$	`z = 3 * b * c + 4;`
$a = \frac{3x + 2}{4a - 1}$	`a = (3 * x + 2) / (4 * a - 1)`

Unlike many programming languages, C++ does not have an exponent operator. Raising a number to a power requires the use of a library function. One of the library functions is called pow, and its purpose is to raise a number to a power. Here is an example of how it's used:

area = pow(4.0, 2); **Equivalent to: 4^2**

Converting Algebraic Expressions to Programming Statements

Figure 3-1



The statement contains a call to the pow function. The numbers inside the parentheses are arguments. Arguments are information being sent to the function. The pow function always raises the first argument to the power of the second argument. In this example, 4.0 is raised to the power of 2. The result is returned from the function and used in the statement where the function call appears. The pow function expects floating-point arguments.

Class Exercise

Expression	Value
6 + 3 * 5	
12 / 2 - 4	
9 + 14 * 2 - 6	
5 + 19 % 3 - 1	
(6 + 2) * 3	
14 / (11 - 4)	
9 + 12 * (8 - 3)	
(6 + 17) % 2 - 1	
(9 - 3) * (6 + 9) / 3	

Implicit Type Conversion

When an operator's operands are of different data types, C++ will automatically convert them to the same data type. This can affect the results of mathematical expressions.

If a floating-point value is assigned to an int variable, what value will the variable receive? If an int is multiplied by a float, what data type will the result be? What if a double is divided by an unsigned int? Is there any way of predicting what will happen in these instances? The answer is yes. C++ follows a set of rules when performing mathematical operations on variables of different data types. It's helpful to understand these rules to prevent subtle errors from creeping into your programs.

Implicit Type Conversion

Table 3-7 Data Type Ranking

long double
double
float
unsigned long
long
unsigned int
int
unsigned short
short
char

When C++ is working with an operator, it strives to convert the operands to the same type. This implicit, or automatic, conversion is known as type coercion. When a value is converted to a higher data type, it is said to be promoted. To demote a value means to convert it to a lower data type.

Implicit Type Conversion

Specific rules that govern the evaluation of mathematical expressions:

Rule 1: char, short, and unsigned short are automatically promoted to int. (*Anytime these data types are used in a mathematical expression, they are automatically promoted to an int.*)

Rule 2: When an operator works with two values of different data types, the lower-ranking value is promoted to the type of the higher-ranking value.

Assume that years is an int and interestRate is a double:

years * interestRate

Before the multiplication takes place, the value in years will be promoted to a double.

Implicit Type Conversion

Rule 3: When the final value of an expression is assigned to a variable, it will be converted to the data type of that variable.

- If the variable receiving the value is of a lower data type than the value it is receiving, the value will be demoted to the type of the variable.
- If the variable's data type does not have enough storage space to hold the value, part of the value will be lost, and the variable could receive an inaccurate result.
- If the variable receiving the value is an integer and the value being assigned to it is a floating-point number, the value will be truncated before being stored in the variable.

Implicit Type Conversion

```
int x;
double y = 3.75;
x = y;                    // x is assigned 3
                          // y remains 3.75
```

If the variable receiving the value has a higher data type than the value being assigned to it, there is no problem.

In the following statement, assume that area is a long int, while length and width are both int:

area = length * width;

Because length and width are both an int, they will not be converted to any other data type. The result of the multiplication, however, will be converted to long so it can be stored in area.

Explicit Type Conversion

Type casting allows you to explicitly perform data type conversion.

A type cast expression lets you manually promote or demote a value. The general format of a type cast expression is

static_cast<DataType>(Value)

where Value is a variable or literal value that you wish to convert and DataType is the data type you wish to convert it to. Here is an example of code that uses a type cast expression:

```
double number = 3.7;
int val;
val = static_cast<int>(number);
```

number is converted to int and the fraction is truncated

Explicit Type Conversion

Program 3-7

```
1  // This program uses a type cast to avoid an integer division.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int    books,
8            months;
9     double booksPerMonth;
10
11    // Get user inputs
12    cout << "How many books do you plan to read? ";
13    cin  >> books;
14    cout << "How many months will it take you to read them? ";
15    cin  >> months;
16
17    // Compute and display books read per month
18    booksPerMonth = static_cast<double>(books) / months;
19    cout << "That is " << booksPerMonth << " books per month.\n";
20    return 0;
21 }
```

books is converted to a double before it is used in the division operation.

Program Output with Example Input Shown in Bold

```
How many books do you plan to read? 30[Enter]
How many months will it take you to read them? 7[Enter]
That is 4.28571 books per month.
```

The type cast expression is used to prevent integer division from taking place; allowing **booksPerMonth** to display the factional value.

Explicit Type Conversion

WARNING! In Program 3-7, the following statement would still have resulted in integer division:

booksPerMonth = static_cast<double>(books / months);

The result of the expression books / months is 4. When 4 is converted to a double, it is 4.0. To prevent the integer division from taking place, one of the operands should be converted to a double prior to the division operation. This forces C++ to automatically convert the value of the other operand to a double.

In the above statement, books will be divided by months using integer division, then the results will be converted to double.

Explicit Type Conversion

Program 3-8

```
1  // This program prints a character from its ASCII code.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number = 65;
8
9     // Display the value of the number variable
10    cout << number << endl;
11
12    // Use a type cast to display the value of number
13    // converted to the char data type
14    cout << static_cast<char>(number) << endl;
15    return 0;
16 }
```

Program Output

```
65
A
```

The type cast expression converts the value in number to the char data type before sending it to cout.

C-style and Prestandard C++ Type Cast Expressions

Examples of Prefix Notation

```
cout << (int) 2.6;                  // Displays integer 2
intVal = (int)number;               // Assigns intVal the value of
                                    // number, converted to an int
booksPerMonth =                     // Converts a copy of the value
  (double)books / months;           // stored in books to a double
                                    // before performing the
                                    // division operation
```

Examples of Functional Notation (Prestandard C++)

```
cout << int(2.6);
intVal = int(number);
booksPerMonth = double(books) / months;
```

The static_cast expression is recommended by the ANSI standard for this type of data type conversion

Overflow and Underflow

When a value cannot fit in the number of bits provided by a variable's data type, overflow or underflow occurs.

Just as a bucket will overflow if you try to put more water in it than it can hold, a variable will experience a similar problem if you try to store a value in it that requires more bits than it has available. Let's look at an example. Suppose a short int that uses 2 bytes of memory has the following value stored in it.

Overflow and Underflow

This is the binary representation of 32,767, the largest value that will fit in this data type. Without going into the details of how negative numbers are stored, it is helpful to understand that for integer data types that store both positive and negative numbers, a number with a 0 in the high order (i.e., leftmost) bit is interpreted as a positive number and a number with a 1 in the high order bit is interpreted as a negative number. If 1 is added to the value stored above, the variable will now be holding the following bit pattern.

Overflow and Underflow

But this is not 32,768. It is interpreted as a negative number instead, which was not what was intended. A binary 1 has “flowed” into the high bit position. This is called overflow.

Likewise, when an integer variable is holding the value at the far end of its data type’s negative range and 1 is subtracted from it, the 1 in its high order bit will become a 0 and the resulting number will be interpreted as a positive number. This is another example of overflow.

In addition to overflow, floating-point values can also experience underflow. This occurs when a value is too close to zero, so small that more digits of precision are needed to express it than can be stored in the variable holding it.

Overflow and Underflow

Program 3-10

```
1  // This program demonstrates overflow and underflow.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      // Set intVar to the maximum value a short int can hold
8      short intVar = 32767;
9
10     // Set floatVar to a number too small to fit in a float
11     float floatVar = 3.0E-47;
12
13     // Display intVar
14     cout << "Original value of intVar     " << intVar << endl;
15
16     // Add 1 to intVar to make it overflow
17     intVar = intVar + 1;
18     cout << "intVar after overflow        " << intVar << endl;
19
20     // Subtract 1 from intVar to make it overflow again
21     intVar = intVar - 1;
22     cout << "intVar after 2nd overflow    " << intVar << endl;
23
24     // Display floatVar
25     cout << "Value of very tiny floatVar  " << floatVar;
26     return 0}
27 }
```

Program Output

```
Original value of intVar    32767
intVar after overflow      -32768
intVar after 2nd overflow   32767
Value of very tiny floatVar 0
```

Overflow and Underflow

Some systems display an error message when an overflow or underflow occurs. But, most do not. The variable simply holds an incorrect value now and the program keeps running. Therefore, it is important to select a data type for each variable that has enough bits

to hold the values you will store in it.

Named Constants

Constants may be given names that symbolically represent them in a program.

Let's assume this statement appears in a banking program that calculates data pertaining to loans. In such a program, two potential problems arise. First, **it is not clear to anyone other than the original programmer what 0.129 is**. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this **number is used in other calculations throughout the program and must be changed periodically**.

Named Constants

Both of these problems can be addressed by using named constants. A named constant, also called a constant variable, is like a variable, but its content is read-only and cannot be changed while the program is running. Here is a definition of a named constant:

```
const double INTEREST_RATE = 0.129;
```

Incorrect usage:

```
const double INTEREST_RATE;          // illegal
INTEREST_RATE = 0.129;               // illegal
```

When a named constant is defined it must be initialized with a value. It cannot be defined and then later assigned a value with an assignment statement.

Named Constants (Advantages)

1. Make programs more self-documenting.

```
const double INTEREST_RATE = 0.129;
newAmount = balance * INTEREST_RATE;
```

2. Widespread changes can easily be made to the program.

No matter how many places the interest rate is used in the program, if the rate changes the programmer only has to change one line of code—the statement that defines and initializes the named constant.

It is important to realize the difference between constant variables created with the key word const and constants created with the #define directive. Constant variables are defined like regular variables. They have a data type and a specific storage location in memory. They are like regular variables in every way except that you cannot change their value while the program is running.

The #define Directive

The older C-style method of creating named constants is with the #define preprocessor directive. Although it is preferable to use the **const** modifier, there are programs with the #define directive still in use. In addition, the #define directive has other uses, so it is important to understand.

NOTE: no semi colon

Ex:

```
#define  PI 3.14159   // PI is "defined" to be 3.14159
```

Constants created with the #define directive are not variables at all, but text substitutions. Each occurrence of the named constant in your source code is removed and the value of the constant is written in its place when it is sent to the compiler.

Multiple and Combined Assignment

Multiple assignment means to assign the same value to several variables with one statement.

C++ allows you to assign a value to multiple variables at once. If a program has several variables, such as a, b, c, and d, and each variable needs to be assigned a value, such as 12, the following statement may be constructed:

a = b = c = d = 12;

The value 12 will be assigned to each variable listed in the statement. This works because the assignment operations are carried out from right to left. First 12 is assigned to d. Then d's value, now a 12, is assigned to c. Then c's value is assigned to b, and finally b's value is assigned to a.

Combined Assignment Operators

The table below shows the common way of writing expressions.

Table 3-8 Assignment Statements that Change a Variable's Value (Assume x = 6)

Statement	What It Does	Value of x After the Statement
`x = x + 4;`	Adds 4 to x	10
`x = x - 3;`	Subtracts 3 from x	3
`x = x * 10;`	Multiplies x by 10	60
`x = x / 2;`	Divides x by 2	3
`x = x % 4`	Makes x the remainder of x / 4	2

Combined Assignment Operators

Quite often programs have assignment statements of the following form:

number = number + 1;

The expression on the right side of the assignment operator gives the value of number plus 1. The result is then assigned to number, replacing the value that was previously stored there. Effectively, this statement adds 1 to number. In a similar fashion, the following statement subtracts 5 from number.

number = number – 5;

Note that in both examples, the same variable name appears on both sides of the assignment operator.

Combined Assignment Operators

Because these types of operations are so common in programming, C++ offers a special set of operators designed specifically for these jobs.

Table 3-9 Combined Assignment Operators

Operator	Example Usage	Equivalent To
+=	`x += 5;`	`x = x + 5;`
-=	`y -= 2;`	`y = y - 2;`
*=	`z *= 10;`	`z = z * 10;`
/=	`a /= b;`	`a = a / b;`
%=	`c %= 3;`	`c = c % 3;`

result *= a + 5; is equivalent to

result = result * (a + 5); not **result = result * a + 5;**

which is different.

Formatting Output

cout provides ways to format data as it is being displayed. This affects the way data appears on the screen.

The same data can be printed or displayed in several different ways. For example, all of the following numbers have the same value, although they look different:

720 720.0 720.00000000 7.2e+2 +720.0

The way a value is printed is called its formatting. The cout object has a standard way of formatting variables of each data type. Sometimes, however, you need more control over the way data is displayed.

Formatting Output

When outputting numbers, the numbers may not line up in columns. This is because some of the numbers, such as 5 and 7, occupy one position on the screen, while others occupy two or three positions. cout uses just the number of spaces needed to print each number.

To remedy this, cout offers a way of specifying the minimum number of spaces to use for each number. A stream manipulator, setw, can be used to establish print fields of a specified width. Here is an example of how it is used:

```
value = 23;
cout << setw(5) << value;
```

Formatting Output

The number inside the parentheses after the word setw specifies the field width for the value immediately following it. The field width is the minimum number of character positions, or spaces, on the screen to print the value in. In our example, the number 23 will be displayed in a field of five spaces.

To further clarify how this works, look at the following statements:

```
value = 23;
cout << "(" << setw(5) << value << ")";
```

This will produce the following output:

```
(   23)
```

Because the number did not use the entire field, cout filled the extra three positions with blank spaces.

Formatting Output

Program 3-15

```
1  // This program uses setw to display three rows of numbers so they al
2  #include <iostream>
3  #include <iomanip>          // Header file needed to use setw
4  using namespace std;
5
6  int main()
7  {
8     int num1 = 2897, num2 = 5,    num3 = 837,
9         num4 = 34,   num5 = 7,    num6 = 1623,
10        num7 = 390,  num8 = 3456, num9 = 12;
11
12    // Display the first row of numbers
13    cout << setw(6) << num1 << setw(6) << num2 << setw(6) << num3 << endl;
14
15    // Display the second row of numbers
16    cout << setw(6) << num4 << setw(6) << num5 << setw(6) << num6 << endl;
17
18    // Display the third row of numbers
19    cout << setw(6) << num7 << setw(6) << num8 << setw(6) << num9 << endl;
20
21    return 0;
22 }
```

This header file is needed to use setw()

Without using setw()

Program Output

```
  2897     5   837
    34     7  1623
   390  3456    12
```

Program Output

```
2897  5  837
34  7  1623
390  3456  12
```