Chapter 3

COMPLETING THE BASICS

In the last chapter, we explored how results are displayed and how numerical data are stored and processed using variables and assignment statements. In this chapter, we complete our introduction to the basics of C++ by presenting additional processing and input capabilities.

3.1 ASSIGNMENT OPERATORS

Assignment operator (=) is used for assignment a value to a variable and for performing computations.

Assignment statement has the syntax:

variable = *expression*;

Expression is any combination of constants, variables, and function calls that can be evaluated to yield a result.

Example:

```
length = 25;
cMyCar = "Mercedes";
sum = 3 + 7;
newtotal = 18.3*amount;
slope = (y2 – y1)/(x2 – x1)
```

The order of events when the computer executes an assignment statement is

- Evaluate the expression on the right hand side of the assignment operator.
- Store the resultant value of the expression in the variable on the left hand side of the assignment operator.

Note:

1. It's important to note that the equal sign in C++ does not have the same meaning as an equal sign in mathematics.
2. Each time a new value is stored in a variable, the old one is overwritten.

Example 3.1.1

```
// This program calculates the volume of a cylinder,
//      given its radius and height
#include <iostream.h>
int main()
{
  float radius, height, volume;
```

```
  radius = 2.5;
  height = 16.0;
  volume = 3.1416 * radius * radius * height;
  cout << "The volume of the cylinder is " << volume << endl;
  return 0;
}
```

The output of the above program:

```
The volume of the cylinder is 314.16
```

We can write *multiple assignments*, such as a = b = c = 25;. Because the assignment operator has a right-to-left associativity, the final evaluation proceeds in the sequence

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

Data Type Conversion across Assignment Operator

Note that data type conversion can take place across assignment operators, that is, the value of the expression on the right side of the assignment operator is converted to the data type of the variable to the left side of the assignment operator.

For example, if *temp* is an integer variable, the assignment *temp* = 25.89 causes the integer value 25 to be stored in the integer variable *temp*.

Assignment Variations

C++ also use a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign. For example, to add 4 to the variable *x* and assign *x* to the result, you say: x += 4. Table 3.1 illustrates assignment operator and all assignment variations.

Table 3.1 Assignment Operators

Operator	Example	Meaning
=	iNum1 = iNum2	
+=	iNum1 += iNum2	iNum1 = iNum1 + iNum2
-=	iNum1 -= iNum2	iNum1 = iNum1 - iNum2
*=	iNum1 *= iNum2	iNum1 = iNum1 * iNum2
/=	iNum1 /= iNum2	iNum1 = iNum1 / iNum2
%=	iNum1 %= iNum2	iNum1 = iNum1 % iNum2

Assignment statements such as sum += 10 or its equivalent, sum = sum + 10, are very common in C++ programming.

Increment and decrement operators

For the special case in which a variable is either increased or decreased by 1, C++ provides two unary operators: *increment operator* and *decrement operator*.

Operator	Description
++	Increase an operand by a value of one
--	Decrease an operand by a value of one

The *increment* (++) and *decrement* (--) unary operators can be used as prefix or postfix operators to increase or decrease value.

A *prefix* operator is placed *before* a variable and returns the value of the operand *after* the operation is performed.

A *postfix* operator is placed *after* a variable and returns the value of the operand *before* the operation is performed.

Prefix and postfix operators have different effects when used in a statement

```
b = ++a;
```

will first increase the value of a to 6, and then assign that new value to b. It is equivalent to

```
a = a +1;
b = a;
```

On the other hand, execution of the statement

```
b = a++;
```

will first assign the value of 5 to b, and then increase the value of a to 6. It is now equivalent to

```
b = a;
a = a + 1;
```

The decrement operators are used in a similar way.

Example 3.1.2

```
// Preincrementing and postincrementing
#include <iostream.h>
int main()
{
  int c;
```

```
  c = 5;
  cout << c << endl;         // print 5
  cout << c++ << endl;       // print 5 then postincrement
  cout << c << endl << endl; // print 6

  c = 5;
  cout << c << endl;         // print 5
  cout << ++c << endl;       // preincrement then print 6
  cout << c << endl;         // print 6

  return 0;                  // successful termination
}
```

The output of the above program:

```
5
5
6

5
6
6
```

3.2 FORMATTING NUMBER FOR PROGRAM OUTPUT

Besides displaying correct results, a program should present its results attractively with good formats.

Stream Manipulators

Stream manipulator functions are special stream functions that change certain characteristics of the input and output.

The main advantage of using manipulator functions is they facilitate the formatting of the input and output streams.

- *setw()* The *setw()* stands for *set width*. This manipulator is used to specify the minimum number of the character positions on the output field a variable will consume.

- *setprecision()* The *setprecision()* is used to control the number of digits of an output stream display of a floating point value. *Setprecision(2)* means 2 digits of precision to the right of the decimal point.

To carry out the operations of these manipulators in a user program, you must include the header file <*iomanip.h*>.

Example 3.2.1

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;
   return 0;
}
```

The output of the above program:

```
  6
 18
124
- - -
148
```

Example:

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

cause the printout

```
|    25.670|
```

- *setiosflags* This manipulator is used to control different input and output settings.
  setioflag(ios::fixed) means the output field will use conventional fixed-point decimal notation.
  setiosflag(ios::showpoint) means the output field will show the decimal point for floating point number.
  setiosflag(ios::scientific) means the output field will use exponential notation.

Note: In the absence of the *ios::fixed* flag, a floating point number is displayed with a default of 6 significant digits. If the integral part of the number requires more than 6 digits, the display will be in exponential notation.

Table 3.2 illustrates some other format flags for use with *setiosflags().*

Table 3.2 Some commonly used format flags going with setiosflags().

Flag	Meaning
ios::showpos	display a leading + sign when the number is positive.
ios::dec	display in decimal format
ios::oct	display in octal format
ios::left	left-justify output
ios::right	right-justify output
ios::hex	display in hexadecimal format

Example 3.2.2

```
// This program will illustrate output conversions
#include <iostream.h>
#include <iomanip.h>
int main()
{
  cout << "The decimal (base 10) value of 15 is " << 15 << endl
       << "The octal (base 8) value of 15 is "
       << setiosflags(ios::oct) << 15 << endl
       << "The hexadecimal (base 16) value of 15 is "
       << setiosflags(ios::hex) << 15 << endl;
  return 0;
  }
```

The output of the above program:

```
The decimal (base 10) value of 15 is 15
The octal (base 8) value of 15 is 17
The hexadecimal (base 16) value of 15 is f
```

To designate an *octal* integer constant, the number must have a leading **0**. *Hexadecimal* number are denoted using a leading **0x**.

Example 3.2.3

```
// Octal and hexadecimal integer constant
#include <iostream.h>

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

The output of the above program:

```
The decimal value of 025 is 21
The decimal value of 0x37 is 55
```

3.3 USING MATHEMATICAL LIBRARY FUNCTIONS

Although addition, subtraction, multiplication and division are easily accomplished using C++'s arithmetic operators, no such operators exist for finding the square root of a number or determining trigonometric values. To facilitate such calculations, C++ provides standard library functions that can be included in a program.

Functions are normally called by writing the name of the function, followed by a left parenthesis, followed by the *argument* (or a comma-separated list of arguments) of the function, followed by a right parenthesis. For example, a programmer desiring to calculate and print the square root of 900.0 might write:

```
cout << sqrt(900.0);
```

When this statement is executed, the math library function *sqrt* is called to calculate the square root of the number contained in the parentheses (900.0). The number 900.0 is the argument of the *sqrt* function. The preceding statement would print 30. The *sqrt* function takes an argument of type *double* and returns a result of type *double*.

If your program uses mathematic function *sqrt()*, it should have the preprocessor command *#include<math.h>* in the beginning of the program. This makes a mathematical library accessible. Table 3.3 lists some commonly used mathematical functions provided in C++.

Except *abs(a)*, the functions all take an argument of type *double* and return a value of type *double*.

Table 3.3 Some mathematical built-in functions

Function Name	Description	Return Value
abs(a)	Absolute value	Same data type as argument
log(a)	Natural logarithm	double
sin(a)	sine of a (a in radians)	double
cos(a)	cosine of a (a in radians)	double
tan(a)	tangent of a (a in radians)	double
log10(a)	common log (base 10) of a	double
pow(a1,a2)	a1 raised to the a2 power	double
exp(a)	e^a	double
sqrt(a)	square root of a	double

Example 3.3.1

```
// this program calculates the area of a triangle
// given its three sides
#include <iostream.h>
#include <math.h>
int main()
{
   double a,b,c, s;
   a = 3;
   b = 4;
   c = 5;
   s = (a+b+c)/2.0;
   area = sqrt(s*(s-a)*(s-b)*(s-c));
   cout << "The area of the triangle =  " << area << endl;
   return 0;
}
```

The output of the above program:

```
The area of the triangle =  6.0
```

Casts

We have already seen the conversion of an operand's data type within mixed-mode expressions and across assignment operators. In addition to these implicit data type conversions that are automatically made within mixed-mode expressions and assignment, C++ also provides for explicit user-specified type conversion. This method of type conversion is called *casting.* The word *cast* is used in the sense of "casting into a mold."

Casting or *type casting*, copies the value contained in a variable of one data type into a variable of another data type.

The C++ syntax for casting variables is

variable = new_type(old_variable);

where the *new_type* portion is the keyword representing the type to which you want to cast the variable.

Example:

```
int iNum = 100;
float fNum;
fNum = float (inum);
```

If you do not explicitly cast a variable of one data type to another data type, then C++ will try to automatically perform the cast for you.

3.4 PROGRAM INPUT USING THE cin OBJECT

So far, our programs have been limited in the sense that all their data must be defined within the program source code.

We will now learn how to write programs which enable data to be entered via the keyboard, while the program is running.

Such programs can be made to operate upon different data every time they run, making them much more flexible and useful.

Standard Input Stream

The *cin* object reads in information from the keyboard via the standard input stream.

The *extraction operator* (>>) retrieves information from the input stream.

When the statement *cin >> num1;* is encountered, the computer stops program execution and accepts data from the keyboard. The user responds by typing an integer (or float) and then pressing the *Enter* key (sometimes called the *Return* key) to send the number to the computer. When a data item is typed, the *cin* object stores the integer (or float) into the variable listed after the >> operator.

The *cin* and *cout* stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it is often called *conversational computing* or *interactive computing*.

Example 3.4.1

```
#include <iostream.h>
int main()
{
  int integer1, integer2, sum;          // declaration

  cout << "Enter first integer\n";  // prompt
  cin >> integer1;                 // read an integer
  cout << "Enter second integer\n"; // prompt
  cin >> integer2;                 // read an integer
  sum = integer1 + integer2;
  cout << "Sum is " << sum << endl;
  return 0;                        // indicate that program ended successfully
}
```

The output of the above program:

```
Enter the first integer
45
Enter the second integer
72
Sum is  117
```

Example 3.4.2

```
#include <iostream.h>
int main()
{
  int integer1, integer2, sum;          // declaration

  cout << "Enter two integers\n";  // prompt
  cin >> integer1 >> integer2;              // read two integers
  sum = integer1 + integer2;
  cout << "Sum is " << sum << endl;
  return 0;
}
```

The output of the above program:

```
Enter two integers
45 72
Sum is  117
```

3.5 SYMBOLIC CONSTANTS

C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed. The qualifier *const* tells the compiler that a name represents a constant. Any data type, built-in or user-defined, may be defined as *const*. If you define something as *const* and then attempt to modify it, the compiler will generate an error. To define a constant in a program, we use *const* declaration qualifier.

Example:

```
const float PI = 3.1416;
const double SALESTAX = 0.05;
const int MAXNUM = 100;
```

Once declared, a constant can be used in any C++ statement in place of the number it represents.

Example 3.5.1

```
// this program calculates the circumference of a circle
// given its radius
```

```
#include <iostream.h>

int main()
{
  const float PI = 3.1416
  float radius, circumference;

  radius = 2.0;
  circumference = 2.0 * PI * radius;
  cout << "The circumference of the circle is "
       << circumference << endl;

  return 0;
}
```

The output of the above program:

```
The circumference of the circle is  12.5664
```