Overview

Computer programs process data to provide information. The job of the programmer is to properly organize data for storage and use. Most data used by programs is stored in either variables or constants.

A ***variable*** holds data that can change while the program is running. A ***constant*** is used to store data that remains the same throughout the program's execution.

CHAPTER 4, SECTION 1

Variable Types

There are more than a dozen types of variables in C++ that can store numbers and characters. Some variables are for storing integers (whole numbers). Other variables are for floating-point numbers (real numbers).

You may recall from math courses that an integer is a positive or negative whole number, such as -2, 4, and 5133. Real numbers can be whole numbers or decimals and can be either positive or negative, such as 1.99, -2.5, 3.14159, and 4.

Data types are of little concern in the real world. When programming in C++, however, you must select a type of variable, called a ***data type***, that best fits the nature of the data itself. Let's now examine the data types that are available for C++ variables.

INTEGER TYPES

When you are working with either positive or negative whole numbers, you should use integer data types for your variables. There are several integer data types available in C++. Selecting which integer data type to use is the next step.

Table 4-1 lists the variable data types that are for storing integers. Notice the range of values that each type can hold. For example, any value from -32,768 to 32,767 can be stored in a variable if the *int* data type is chosen. If you need to store a value outside of that specific range, you must choose a different data type.

INTEGER DATA TYPES	MINIMUM RANGE OF VALUES	MINIMUM NUMBER OF BYTES OCCUPIED
char	–128 to 127	1
unsigned char	0 to 255	1
int	–32,768 to 32,767	2
short	–32,768 to 32,767	2
unsigned int	0 to 65,535	2
long	–2,147,483,648 to 2,147,483,647	4
unsigned long	0 to 4,294,967,295	4

TABLE 4-1

Note

The number of bytes occupied by the data types may vary depending on your compiler, especially in regard to the int type. Your compiler may use four bytes for an int, in which case the range for an int is -2,147,483,648 to 2,147,483,647. In cases where a four-byte int is used, the short type should be two bytes long.

Notice the range of values for ***unsigned*** data types in Table 4-1. An unsigned variable can store only positive numbers. For example, if you were to be storing the weights of trucks in variables, an unsigned data type might be a good choice. A truck can't weigh less than zero.

Why would you want to use the int type when the long type has a bigger range? The answer is that you *can* use the long type when an int would do, but there is more to consider. Notice the third column of Table 4-1. The variables with the larger ranges require more of the computer's memory. In addition, it often takes the computer longer to access the data types that require more memory. Having all of these data types gives the programmer the ability to use only what is necessary for each variable, decrease memory usage, and increase speed.

Speed

You have already seen program speed mentioned a few times in this book, and you will see it again. Right now it may seem like the computer runs so quickly that the difference in processing speed between different data types is of little consequence. But in more complex programs, a little increase in speed can go a long way.

Consider what would happen if you could increase the speed at which a program compares database records by a tenth of a second per record. If the program has to compare 20,000 records, you have reduced the time required to compare the records by over half an hour! Even though the time saved by using the most efficient data type is much less than a tenth of a second, you can still see how small improvements in processing time add up.

On the Net

To learn more about how integers are stored in RAM, see http://www.ProgramCPP.com. See topic 4.1.1.

CHARACTERS

Recall from Chapter 2 that in the computer, characters are stored as numbers, and that the ASCII codes are what the computer uses to assign numbers to the characters. Because the computer considers characters to be numbers, they are stored in an integer data type named *char*.

Each variable of the char data type can hold only one character. In order to store words or sentences, you must *string* characters together. A group of characters put together to make a word or more is called a ***string***. Sometimes you will hear a string referred to as an array of characters. You will learn more about characters and strings in a later chapter.

Note

Pronounce char like "care." Remember, it is short for character.

FLOATING-POINT TYPES

Integer variables are inappropriate for certain types of data. For example, tasks as common as working with money call for using floating-point numbers. Just as there is more than one type of integer, there is more than one type of floating-point variable.

Extra for Experts

Making Floating-Point More Efficient

Earlier you read that floating-point numbers are not as efficient as integers. This is true. But many computer manufacturers include a floating-point unit (FPU) to help with the problem. A ***floating-point unit*** *is a processor that works with floating-point numbers. FPUs are often called* ***math coprocessors.*** *Microprocessors with an FPU can perform calculations with floating-point numbers much more quickly than a microprocessor without an FPU. Some FPUs are on a chip separate from the microprocessor and some are built into the microprocessor chip.*

On the Net

To learn more about math coprocessors in modern microprocessors and how dramatically floating-point operations are increased by an FPU, see http://www.ProgramCPP.com. See topic 4.1.2.

Table 4-2 lists the three floating-point data types and their range of values. The range of floating-point data types are more complicated than the range of integers. Selecting an appropriate floating-point type is based upon both the range of values and the required decimal precision.

TABLE 4-2

FLOATING POINT DATA TYPES	APPROXIMATE RANGE OF VALUES	DIGITS OF PRECISION	NUMBER OF BYTES OCCUPIED
float	3.4×10^{-38} to 3.4×10^{38}	7	4
double	1.7×10^{-308} to 1.7×10^{308}	15	8
long double	3.4×10^{-4932} to 1.1×10^{4932}	19	10

Note

The information in Table 4-2 may vary among compilers. Check your compiler's manual for exact data type ranges and bytes occupied for both integers and floating-point numbers.

When you are choosing a floating-point data type, first look to see how many digits of precision are necessary to store the value you need to store. For example, if you need to store π as 3.1415926535897, 14 digits of precision are required. Therefore you should use the double type. You should also verify that your value will fit within the range of values the type supports. But unless you

are dealing with very large or very small numbers, the range is not usually as important an issue as the precision.

Let's look at some examples of values and what data types would be appropriate for the values.

Dollar amounts in the range $-99,999.99 to $99,999.99 can be handled with a variable of type *float*. A variable of type *double* can store dollar amounts in the range $-9,999,999,999,999.99 to $9,999,999,999,999.99.

The number 5.98×10^{24} kg, which happens to be the mass of the Earth, can be stored in a variable of type float because the number is within the range of values and requires only three digits of precision.

On the Net

To learn more about how floating-point numbers are stored in RAM, see http://www.ProgramCPP.com. See topic 4.1.3.

BOOLEAN VARIABLES

A ***Boolean variable*** is a variable which can have only two possible values. One of the values represents true (or some other form of the affirmative) and the other value represents false (or some other form of the negative). Boolean variables are very useful in programming to store information such as whether an answer is yes or no, whether a report has been printed or not, or whether a device is currently on or off.

On the Net

Boolean variables are named in honor of George Boole, an English mathematician who lived in the 1800s. Boole created a system called ***Boolean algebra****, which is a study of operations that take place using variables with the values true and false. For more information about George Boole and Boolean algebra see http://www.ProgramCPP.com. See topic 4.1.4.*

Some C++ compilers do not support a Boolean variable. Others have a data type **`bool`** which can be used to declare Boolean variables. If your compiler does not support the **`bool`** data type, you can use the **`bool.h`** header file on your work disk to make the feature available in your programs. Your instructor can help you access this header file. Later in this course, you will use the **`bool`** data type and examine the header file which makes it work.

SECTION 4.1 QUESTIONS

1. What integer data type is necessary to store the value 4,199,999,999?
2. Why is it important to use data types that store your data efficiently?
3. What range of values can be stored in an unsigned int variable?
4. What is a string?
5. What floating-point data type provides the most digits of precision?

CHAPTER 4, SECTION 2

Using Variables

You are now going to have an opportunity to put your knowledge to work and select the right variable for the job. You must first indicate to the compiler what kind of variable you want and what you want to name it. Then it is ready to use.

DECLARING AND NAMING VARIABLES

Indicating to the compiler what type of variable you want and what you want to call it is called ***declaring*** the variable.

DECLARING VARIABLES

You must declare a variable before you can use it. The C++ statement declaring a variable must include the data type followed by the name you wish to call the variable, and a semicolon. An integer variable named `i` is declared in Figure 4-1.

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

main()
{
  int i;     // declare i as an integer

  i = 2;
  cout << i << '\n';
  return 0;
}
```

FIGURE 4-1
This program declares *i* as an integer.

Table 4-3 shows that declaring variables for other data types is just as easy as the example in Figure 4-1.

DATA TYPE	EXAMPLE C++ DECLARATION STATEMENT
char	`char Grade;`
unsigned char	`unsigned char T5;`
int	`int DaysInMonth;`
short	`short temperature;`
unsigned int	`unsigned int Age_in_dog_years;`
long	`long PopulationChange;`
unsigned long	`unsigned long j;`
float	`float CostPerUnit;`
double	`double Distance;`
long double	`long double x;`

TABLE 4-3

C++ will allow you to declare a variable anywhere in the program as long as the variable is declared before you use it. However, you should get into the habit of declaring all variables at the top of the function. Declaring variables at the top of the function makes for better organized code, makes the variables easy to locate, and helps you plan for the variables you will need.

NAMING VARIABLES

The names of variables in C++ are typically referred to as ***identifiers***. Notice how the variable names in Table 4-3 are very descriptive and consider how they might help the programmer recall the variable's purpose. You are encouraged to use the same technique. For example, a variable that holds a bank balance could be called **`balance`**, or the circumference of a circle could be stored in a variable named **`circumference`**. The following are rules for creating identifiers.

- Identifiers must start with a letter or an underscore (_). You should, however, avoid using identifiers that begin with underscores because the language's internal identifiers often begin with underscores. By avoiding the use of underscores as the first character, you will ensure that your identifier remains out of conflict with C++'s internal identifiers.
- As long as the first character is a letter, you can use letters or numerals in the rest of the identifier.
- Use a name that makes the purpose of the variable clear, but avoid making it unnecessarily long. Most C++ compilers will recognize only the first 31 or 32 characters.
- There can be no spaces in identifiers. A good way to create a multi-word identifier is to use an underscore between the words, for example **`last_name`**.
- The following words, called ***keywords***, must NOT be used as identifiers because they are part of the C++ language. Your compiler may have additional keywords not listed here.

`asm`	`delete`	`if`	`return`	`try`
`auto`	`do`	`inline`	`short`	`typedef`
`break`	`double`	`int`	`signed`	`union`
`case`	`else`	`long`	`sizeof`	`unsigned`
`catch`	`enum`	`new`	`static`	`virtual`
`char`	`extern`	`operator`	`struct`	`void`
`class`	`float`	`private`	`switch`	`volatile`
`const`	`for`	`protected`	`template`	`while`
`continue`	`friend`	`public`	`this`	
`default`	`goto`	`register`	`throw`	

Recall from the previous chapter that C++ is case sensitive. The capitalization you use when the variable is declared must be used each time the variable is accessed. For example, **`total`** is not the same identifier as **`Total`**.

Table 4-4 gives some examples of illegal identifiers.

TABLE 4-4

IMPROPER C++ VARIABLE NAMES	WHY ILLEGAL
`Miles per gallon`	Spaces are not allowed
`register`	register is a keyword
`4Sale`	Identifiers cannot begin with numerals

DECLARING MULTIPLE VARIABLES IN A STATEMENT

You can declare more than one variable in a single statement as long as all of the variables are of the same type. For example, if your program requires three variables of type float, all three variables could be declared by placing commas between the variables like this:

```
float x, y, z;
```

INITIALIZING VARIABLES

The compiler assigns a location in memory to a variable when it is declared. However, a value already exists in the space reserved for your variable. A random value could have been stored when the computer was turned on, or the location could retain data from a program that ran earlier. Regardless, the memory location now belongs to your program and you must specify the initial value to be stored in the location. This process is known as initializing.

Extra for Experts

Each byte of memory is filled with a value upon turning on your computer. The compiler sets up memory locations for your variables, and you initialize them. When your program is through with a variable, that memory location is once again made available to be used by a different variable or for another purpose. The value you last stored in the variable will remain in that memory location until it is assigned another value.

To ***initialize*** a variable, you simply assign it a value. In C++, the equal sign (=) is used to assign a value to a variable. In Figure 4-2, the variables `i` and `j` are initialized to the values of 3 and 2 respectively. Notice that the variable `k` has yet to be initialized because it is to be assigned the sum of `i` and `j`.

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

  // initialize i, j, and k
  i = 3;
  j = 2;
  k = i + j;
  return 0;
}
```

FIGURE 4-2
This program declares three integers and assigns values to them.

EXERCISE 4-1 DECLARING VARIABLES

1. Enter the following program into a blank editor screen:

```
// IDECLARE.CPP
// Example of variable declaration.

#include <iostream.h>

main()
{
  int i;              // declare i as an integer
  i = 0;              // initialize i to 0
  cout << i << '\n';
  return 0;
}
```

2. Save the source code file as *IDECLARE.CPP*.
3. Compile and run the program. The program should print the number 0 on your screen. If no errors are encountered, leave the program on your screen. If errors are found, check the source code for keyboarding errors and compile again.
4. Change the initialization statement to initialize the value of `i` to -40 and run again. The number -40 is shown on your screen. Save the source code again and leave the source code file open for the next exercise.

Assigning a floating-point value to a variable works the way you probably expect, except when you need to use ***exponential notation.*** You may have used exponential notation and called it scientific notation. In exponential notation, very large or very small numbers are represented with a fractional part (called the mantissa) and an exponent. Use an *e* to signify exponential notation. Just place an *e* in the number to separate the mantissa from the exponent. Below are some examples of statements that initialize floating-point variables.

```
x = 2.5;
ElectronGFactor = 1.0011596567;
Radius_of_Earth = 6.378164e6;    // radius of Earth at equator
Mass_of_Electron = 9.109e-31;    // 9.109 x 10^-31 kilograms
```

EXERCISE 4-2 DECLARING AND INITIALIZING FLOATING-POINT VARIABLES

1. Modify the program on your screen to match the program below.

```
// IDECLARE.CPP
// Example of variable declaration.

#include <iostream.h>

main()
{
```

```
    float x, Radius_of_Earth, Mass_of_Electron;
    int i;                  // declare i as an integer
    i = 0;                  // initialize i to 0
    x = 2.5;
    Radius_of_Earth = 6.378164e6;
    Mass_of_Electron = 9.109e-31;
    cout << i << '\n';
    cout << x << '\n';
    cout << Radius_of_Earth << '\n';
    cout << Mass_of_Electron << '\n';
    return 0;
}
```

2. Compile and run the program. The integer and three floating-point values should print to the screen.
3. When the program runs successfully, close the source code file.

SECTION 4.2 QUESTIONS

1. Write a statement to declare an integer named **age** as an unsigned char.
2. What are the words called that cannot be used as identifiers because they are part of the C++ language?
3. Why can't "first name" be used as an identifier?
4. Write a statement that declares four int data type variables **i, j, k,** and **l** in a single statement.
5. What character is used to assign a value to a variable?

CHAPTER 4, SECTION 3

Constants

In C++, a constant holds data that remains the same as the program runs. Constants allow you to give a name to a value that is used several times in a program so that the value can be more easily used. For example, if you use the value of π (3.14159) several times in your program, you can assign the value 3.14159 to the name **PI**. Then, each time you need the value 3.14159, you need only use the name **PI**.

Constants are defined in a manner that is similar to the way you define a variable. You still must select a data type and give the constant a name. The difference is you tell the compiler that the data is a constant using the *const* keyword and assign a value all in the same statement.

The statement below declares **PI** as a constant.

```
const float PI = 3.14159;
```

Any valid identifier name can be used to name a constant. The same rules apply as with variables. Traditionally, uppercase letters have been used when

naming constants. Lowercase letters are generally used with variable names. Therefore, uppercase letters help distinguish constants from variables. Some C++ programmers think lowercase letters should be used for constants as well as variables. In this book, we will use uppercase letters for constants because it will help you quickly identify constants in programs. Just be aware that you may see programs elsewhere that use lowercase letters for constants.

EXERCISE 4-3 USING CONSTANTS

1. Enter the following program. Save the source code as *CIRCLE.CPP*.

```
// CIRCLE.CPP
// Example of using a constant.

#include <iostream.h>

main()
{
  const float PI = 3.14159;     // declare PI as a constant
  float circumference, radius;

  // Ask user for the radius of a circle
  cout << "What is the radius of the circle? ";
  cin  >> radius;

  circumference = 2 * PI * radius;  // calculate circumference

  // Output the circle's circumference
  cout << "The circle's circumference is ";
  cout << circumference << '\n';
  return 0;
}
```

2. Compile and run the program. Enter 4 as the radius of the circle. The program will return 25.132721 as the circumference.
3. An error message is generated if you add the following line at the end of the program. Add the line before the closing brace (}).

```
PI = 2.5;
```

Remember that the value of a constant remains the same while the program is running.

4. Compile the program again to see the error generated.
5. Delete the line causing the error and compile the program again.

The compiler prohibits the assignment of another value to a constant after the declaration statement. If you fail to initialize the constant in the declaration statement, however, whatever value is in the memory location remains assigned to the constant throughout the execution of the program.

A good reason to use constants in a large program is that it gives you the ability to easily change the value of the constant in one place in the program. For example, suppose you have a program that needs the sales tax rate in several

places. If you declare a constant named **TAX_RATE**, when the tax rate changes you have to change the constant only where it is declared. Every place in the program that uses the **TAX_RATE** constant is now going to use the new value.

SECTION 4.3 QUESTIONS

1. What is a constant?
2. What keyword is used to declare a constant in C++?
3. Write a declaration statement to create a constant named **PI** with 16 digit precision and use 3.141592653589793 for π. *Hint:* Choose your data type carefully.
4. Write a constant declaration statement to create a constant for the number of feet in a mile (5,280).
5. When is it appropriate to use constants?

KEY TERMS

Boolean variable
constant
data type
declaring
exponential notation
floating-point unit
identifier
initialize
keyword
math coprocessor
string
unsigned
variable

SUMMARY

- Most data is stored in either variables or constants.
- There are several types of variables. Some are for integer data and some are for floating-point data.
- Integer data types are selected based on the range of values you need to store. Some integer data types are unsigned, meaning they can store only positive numbers.
- Characters are stored in the computer as numbers. The char data type can store one character of data.
- Floating-point data types are selected based on the range of values and the required precision.
- Boolean variables are variables which can have only two possible values: true or false.
- Variables must be declared before they are used. Variables should also be initialized to clear any random values that may be in the memory location. When a variable is declared, it must be given a legal name called an identifier.

- Constants are declared in a way similar to variables. The const keyword tells the compiler that the data is a constant. The constant must be assigned a value in the declaration statement.

PROJECTS

PROJECT 4-1

1. Enter, compile, and run the following program. Save the source code file as *DATATYPE.CPP*.

```
// DATATYPE.CPP
// Examples of variable declaration and
// initialization.

#include <iostream.h>

main()
{
  // declare a constant for the square root of two
  const double SQUARE_ROOT_OF_TWO = 1.414214;

  int i;              // declare i as an integer
  long j;             // j as a long integer
  unsigned long k;    // k as an unsigned long integer
  float n;            // n as a floating point number

  i = 3;              // initialize i to 3
  j = -2048111;       // j to -2,048,111
  k = 4000000001;     // k to 4,000,000,001
  n = 1.887;          // n to 1.887

  // output constant and variables to screen
  cout << SQUARE_ROOT_OF_TWO << '\n';
  cout << i << '\n';
  cout << j << '\n';
  cout << k << '\n';
  cout << n << '\n';
  return 0;
}
```

2. Add declarations using appropriate identifiers for the values below. Declare *e*, the speed of light, and the speed of sound as constants. Initialize the variables. Use any identifier you want for those values that give you no indication as to their purpose.

100	*e* (2.7182818)
–1000	Speed of light (3.00×10^8 m/s)
–40,000	Speed of sound (340.292 m/s)
40,000	

3. Print the new values to the screen.
4. Save, compile, and run. Correct any errors you have made.
5. Close the source code file.

PROJECT 4-2

Write a program that declares two constants (A and B). Initialize A = 1 an B = 2.2. Next, declare an int named C and a float named D. Initialize C = A and D = B. Write statements to print C and D to the screen.

PROJECT 4-3

Use your compiler's manuals or on-line help to determine the ranges and sizes of the data types available in your compiler. Compare your findings to Tables 4-1 and 4-2. If your compiler differs from the tables, record those differences. Also record any additional data types not listed in the tables.