Variables Defined at Point of Use

The FAHREN program has several new wrinkles besides its input capability. Look closely at the listing. Where is the variable `ctemp` defined? Not at the beginning of the program, but in the next-to-the-last line, where it's used to store the result of the arithmetic operation. As we noted earlier, you can define variables throughout a program, not just at the beginning. (Many languages, including C, require all variables to be defined before the first executable statement.)

Defining variables where they are used can make the listing easier to understand, since you don't need to refer repeatedly to the start of the listing to find the variable definitions. However, the practice should be used with discretion. Variables that are used in many places in a function are better defined at the start of the function.

Cascading <<

The insertion operator `<<` is used repeatedly in the second `cout` statement in FAHREN. This is perfectly legal. The program first sends the phrase Equivalent in Celsius is: to `cout`, then it sends the value of `ctemp`, and finally the newline character `'\n'`.

The extraction operator `>>` can be cascaded with `cin` in the same way, allowing the user to enter a series of values. However, this capability is not used so often, since it eliminates the opportunity to prompt the user between inputs.

Expressions

Any arrangement of variables, constants, and operators that specifies a computation is called an expression. Thus, `alpha+12` and `(alpha-37)*beta/2` are expressions. When the computations specified in the expression are performed, the result is usually a value. Thus if `alpha` is 7, the first expression shown has the value 19.

Parts of expressions may also be expressions. In the second example, `alpha-37` and `beta/2` are expressions. Even single variables and constants, like `alpha` and `37`, are considered to be expressions.

Note that expressions aren't the same as statements. Statements tell the compiler to do something and terminate with a semicolon, while expressions specify a computation. There can be several expressions in a statement.

Precedence

Note the parentheses in the expression

```
(ftemp-32) * 5 / 9
```

Without the parentheses, the multiplication would be carried out first, since * has higher priority than -. With the parentheses, the subtraction is done first, then the multiplication, since all operations inside parentheses are carried out first. What about the precedence of the * and / signs? When two arithmetic operators have the same precedence, the one on the left is executed first, so in this case the multiplication will be carried out next, then the division. Precedence and parentheses are normally applied this same way in algebra and in other computer languages, so their use probably seems quite natural. However, precedence is an important topic in C++. We'll return to it later when we introduce different kinds of operators.

Floating Point Types

We've talked about type `int` and type `char`, both of which represent numbers as integers—that is, numbers without a fractional part. Now let's examine a different way of storing numbers—as floating-point variables.

Floating-point variables represent numbers with a decimal place—like 3.1415927, 0.0000625, and –10.2. They have both an integer part, to the left of the decimal point, and a fractional part, to the right. Floating-point variables represent what mathematicians call real numbers, which are used for measurable quantities such as distance, area, and temperature. They typically have a fractional part.

There are three kinds of floating-point variables in C++: type `float`, type `double`, and type `long double`. Let's start with the smallest of these, type `float`.

Type `float`

Type `float` stores numbers in the range of about 3.4×10^{-38} to 3.4×10^{38}, with a precision of seven digits. It occupies 4 bytes (32 bits) in memory, as shown in Figure 2.6.

The following example program prompts the user to type in a floating-point number representing the radius of a circle. It then calculates and displays the circle's area.

```
// circarea.cpp
// demonstrates floating point variables
#include <iostream>                     //for cout, etc.
using namespace std;

int main()
   {
   float rad;                           //variable of type float
   const float PI = 3.14159F;           //type const float

   cout << "Enter radius of circle: ";  //prompt
   cin >> rad;                          //get radius
```

```
float area = PI * rad * rad;          //find area
cout << "Area is " << area << endl; //display answer
return 0;
}
```

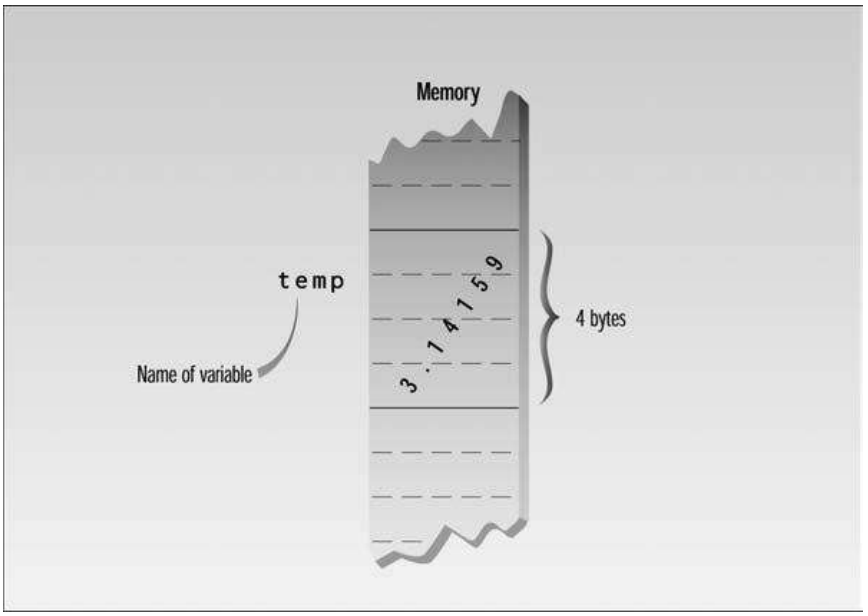

FIGURE 2.6
Variable of type `float` in memory.

Here's a sample interaction with the program:

```
Enter radius of circle: 0.5
Area is 0.785398
```

This is the area in square feet of a 12-inch LP record (which has a radius of 0.5 feet). At one time this was an important quantity for manufacturers of vinyl.

Type `double` and `long double`

The larger floating point types, `double` and `long double`, are similar to `float` except that they require more memory space and provide a wider range of values and more precision. Type `double` requires 8 bytes of storage and handles numbers in the range from 1.7×10^{-308} to 1.7×10^{308} with a precision of 15 digits. Type `long double` is compiler-dependent but is often the same as `double`. Type `double` is shown in Figure 2.7.

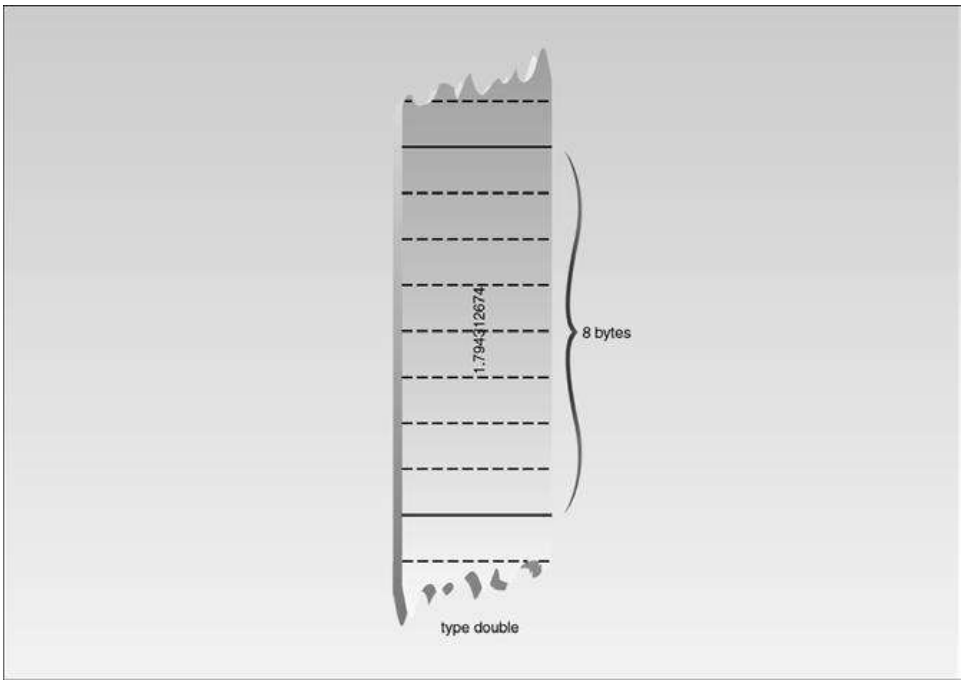

FIGURE 2.7
Variable of type `double`.

Floating-Point Constants

The number 3.14159F in CIRCAREA is an example of a floating-point constant. The decimal point signals that it is a floating-point constant, and not an integer, and the F specifies that it's type `float`, rather than `double` or `long double`. The number is written in normal decimal notation. You don't need a suffix letter with constants of type `double`; it's the default. With type `long double`, use the letter L.

You can also write floating-point constants using exponential notation. Exponential notation is a way of writing large numbers without having to write out a lot of zeros. For example, 1,000,000,000 can be written as 1.0E9 in exponential notation. Similarly, 1234.56 would be written 1.23456E3. (This is the same as 1.23456 times 10^3.) The number following the E is called the exponent. It indicates how many places the decimal point must be moved to change the number to ordinary decimal notation.

The exponent can be positive or negative. The exponential number 6.35239E–5 is equivalent to 0.0000635239 in decimal notation. This is the same as 6.35239 times 10^{-5}.

The const Qualifier

Besides demonstrating variables of type `float`, the CIRCAREA example also introduces the qualifier `const`. It's used in the statement

```
const float PI = 3.14159F;  //type const float
```

The keyword `const` (for constant) precedes the data type of a variable. It specifies that the value of a variable will not change throughout the program. Any attempt to alter the value of a variable defined with this qualifier will elicit an error message from the compiler.

The qualifier `const` ensures that your program does not inadvertently alter a variable that you intended to be a constant, such as the value of PI in CIRCAREA. It also reminds anyone reading the listing that the variable is not intended to change. The `const` modifier can apply to other entities besides simple variables. We'll learn more about this as we go along.

The #define Directive

Although the construction is not recommended in C++, constants can also be specified using the preprocessor directive `#define`. This directive sets up an equivalence between an identifier and a text phrase. For example, the line

```
#define PI 3.14159
```

appearing at the beginning of your program specifies that the identifier `PI` will be replaced by the text `3.14159` throughout the program. This construction has long been popular in C. However, you can't specify the data type of the constant using `#define`, which can lead to program bugs; so even in C `#define` has been superseded by `const` used with normal variables. However, you may encounter this construction in older programs.

Type bool

For completeness we should mention type `bool` here, although it won't be important until we discuss relational operators in the next chapter.

We've seen that variables of type `int` can have billions of possible values, and those of type `char` can have 256. Variables of type `bool` can have only two possible values: `true` and `false`. In theory a `bool` type requires only one bit (not byte) of storage, but in practice compilers often store them as bytes because a byte can be quickly accessed, while an individual bit must be extracted from a byte, which requires additional time.

As we'll see, type `bool` is most commonly used to hold the results of comparisons. Is `alpha` less than `beta`? If so, a `bool` value is given the value `true`; if not, it's given the value `false`.

Type `bool` gets its name from George Boole, a 19th century English mathematician who invented the concept of using logical operators with true-or-false values. Thus such true/false values are often called Boolean values.

The setw Manipulator

We've mentioned that manipulators are operators used with the insertion operator (`<<`) to modify—or manipulate—the way data is displayed. We've already seen the `endl` manipulator; now we'll look at another one: `setw`, which changes the field width of output.

You can think of each value displayed by `cout` as occupying a field: an imaginary box with a certain width. The default field is just wide enough to hold the value. That is, the integer 567 will occupy a field three characters wide, and the string "`pajamas`" will occupy a field seven characters wide. However, in certain situations this may not lead to optimal results. Here's an example. The WIDTH1 program prints the names of three cities in one column, and their populations in another.

```
// width1.cpp
// demonstrates need for setw manipulator
#include <iostream>
using namespace std;

int main()
   {
   long pop1=2425785, pop2=47, pop3=9761;

   cout << "LOCATION " << "POP." << endl
        << "Portcity " << pop1 << endl
        << "Hightown " << pop2 << endl
        << "Lowville " << pop3 << endl;
   return 0;
   }
```

Here's the output from this program:

```
LOCATION POP.
Portcity 2425785
Hightown 47
Lowville 9761
```

Unfortunately, this format makes it hard to compare the numbers; it would be better if they lined up to the right. Also, we had to insert spaces into the names of the cities to separate them from the numbers. This is an inconvenience.

Here's a variation of this program, WIDTH2, that uses the `setw` manipulator to eliminate these problems by specifying field widths for the names and the numbers:

```
// width2.cpp
// demonstrates setw manipulator
#include <iostream>
#include <iomanip>      // for setw
using namespace std;

int main()
   {
   long pop1=2425785, pop2=47, pop3=9761;

   cout << setw(8) << "LOCATION" << setw(12)
        << "POPULATION" << endl
        << setw(8) << "Portcity" << setw(12) << pop1 << endl
        << setw(8) << "Hightown" << setw(12) << pop2 << endl
        << setw(8) << "Lowville" << setw(12) << pop3 << endl;
   return 0;
   }
```

The `setw` manipulator causes the number (or string) that follows it in the stream to be printed within a field `n` characters wide, where `n` is the argument to `setw(n)`. The value is right-justified within the field. Figure 2.8 shows how this looks. Type `long` is used for the population figures, which prevents a potential overflow problem on systems that use 2-byte integer types, in which the largest integer value is 32,767.

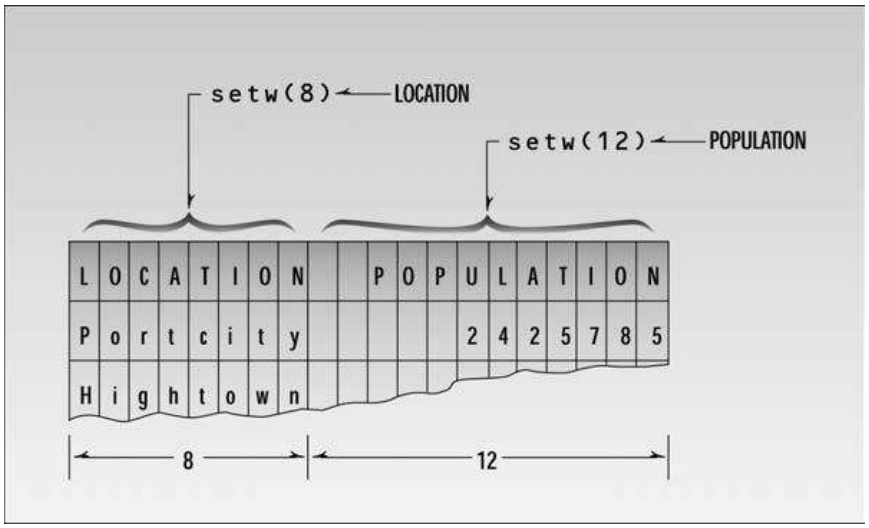

FIGURE 2.8
Field widths and `setw`.

Here's the output of WIDTH2:

```
LOCATION  POPULATION
Portcity     2425785
Hightown          47
Lowville        9761
```

Cascading the Insertion Operator

Note that there's only one `cout` statement in WIDTH1 and WIDTH2, although it's written on multiple lines. In doing this, we take advantage of the fact that the compiler ignores whitespace, and that the insertion operator can be cascaded. The effect is the same as using four separate statements, each beginning with `cout`.

Multiple Definitions

We initialized the variables `pop1`, `pop2`, and `pop3` to specific values at the same time we defined them. This is similar to the way we initialized `char` variables in the CHARVARS example. Here, however, we've defined and initialized all three variables on one line, using the same `long` keyword and separating the variable names with commas. This saves space where a number of variables are all the same type.

The IOMANIP Header File

The declarations for the manipulators (except `endl`) are not in the usual IOSTREAM header file, but in a separate header file called IOMANIP. When you use these manipulators you must `#include` this header file in your program, as we do in the WIDTH2 example.

Variable Type Summary

Our program examples so far have used four data types—`int`, `char`, `float`, and `long`. In addition we've mentioned types `bool`, `short`, `double`, and `long double`. Let's pause now to summarize these data types. Table 2.2 shows the keyword used to define the type, the numerical range the type can accommodate, the digits of precision (in the case of floating-point numbers), and the bytes of memory occupied in a 32-bit environment.

TABLE 2.2 Basic C++ Variable Types

	Numerical Range		Digits of	Bytes of
Keyword	Low	High	Precision	Memory
`bool`	false	true	n/a	1
`char`	–128	127	n/a	1
`short`	–32,768	32,767	n/a	2

TABLE 2.2 Continued

Keyword	Numerical Range Low	High	Digits of Precision	Bytes of Memory
`int`	–2,147,483,648	2,147,483,647	n/a	4
`long`	–2,147,483,648	2,147,483,647	n/a	4
`float`	3.4×10^{-38}	3.4×10^{38}	7	4
`double`	1.7×10^{-308}	1.7×10^{308}	15	8

unsigned Data Types

By eliminating the sign of the character and integer types, you can change their range to start at 0 and include only positive numbers. This allows them to represent numbers twice as big as the signed type. Table 2.3 shows the unsigned versions.

TABLE 2.3 Unsigned Integer Types

Keyword	Numerical Range Low	High	Bytes of Memory
`unsigned char`	0	255	1
`unsigned short`	0	65,535	2
`unsigned int`	0	4,294,967,295	4
`unsigned long`	0	4,294,967,295	4

The `unsigned` types are used when the quantities represented are always positive—such as when representing a count of something—or when the positive range of the signed types is not quite large enough.

To change an integer type to an unsigned type, precede the data type keyword with the keyword `unsigned`. For example, an unsigned variable of type `char` would be defined as

```
unsigned char ucharvar;
```

Exceeding the range of signed types can lead to obscure program bugs. In certain (probably rare) situations such bugs can be eliminated by using unsigned types. For example, the following program stores the constant 1,500,000,000 (1.5 billion) both as an `int` in `signedVar` and as an `unsigned int` in `unsignVar`.

```
// signtest.cpp
// tests signed and unsigned integers
#include <iostream>
```

```
using namespace std;

int main()
   {
   int signedVar = 1500000000;           //signed
   unsigned int unsignVar = 1500000000;  //unsigned

   signedVar = (signedVar * 2) / 3; //calculation exceeds range
   unsignVar = (unsignVar * 2) / 3;  //calculation within range

   cout << "signedVar = " << signedVar << endl; //wrong
   cout << "unsignVar = " << unsignVar << endl; //OK
   return 0;
   }
```

The program multiplies both variables by 2, then divides them by 3. Although the result is smaller than the original number, the intermediate calculation is larger than the original number. This is a common situation, but it can lead to trouble. In SIGNTEST we expect that two-thirds the original value, or 1,000,000,000, will be restored to both variables. Unfortunately, in `signedVar` the multiplication created a result—3,000,000,000—that exceeded the range of the `int` variable (–2,147,483,648 to 2,147,483,647). Here's the output:

```
signedVar = -431,655,765
unsignVar = 1,000,000,000
```

The signed variable now displays an incorrect answer, while the unsigned variable, which is large enough to hold the intermediate result of the multiplication, records the result correctly. The moral is this: Be careful that all values generated in your program are within the range of the variables that hold them. (The results will be different on 16-bit or 64-bit computers, which use different numbers of bytes for type `int`.)

Type Conversion

C++, like C, is more forgiving than some languages in the way it treats expressions involving several different data types. As an example, consider the MIXED program:

```
// mixed.cpp
// shows mixed expressions
#include <iostream>
using namespace std;

int main()
   {
   int count = 7;
   float avgWeight = 155.5F;
```

```
double totalWeight = count * avgWeight;
cout << "totalWeight=" << totalWeight << endl;
return 0;
}
```

Here a variable of type `int` is multiplied by a variable of type `float` to yield a result of type `double`. This program compiles without error; the compiler considers it normal that you want to multiply (or perform any other arithmetic operation on) numbers of different types.

Not all languages are this relaxed. Some don't permit mixed expressions, and would flag the line that performs the arithmetic in MIXED as an error. Such languages assume that when you mix types you're making a mistake, and they try to save you from yourself. C++ and C, however, assume that you must have a good reason for doing what you're doing, and they help carry out your intentions. This is one reason for the popularity of C++ and C. They give you more freedom. Of course, with more freedom, there's also more opportunity for you to make a mistake.

Automatic Conversions

Let's consider what happens when the compiler confronts such mixed-type expressions as the one in MIXED. Types are considered "higher" or "lower," based roughly on the order shown in Table 2.4.

TABLE 2.4 Order of Data Types

Data Type	Order
`long double`	Highest
`double`	
`float`	
`long`	
`int`	
`short`	
`char`	Lowest

The arithmetic operators such as `+` and `*` like to operate on two operands of the same type. When two operands of different types are encountered in the same expression, the lower-type variable is converted to the type of the higher-type variable. Thus in MIXED, the `int` value of `count` is converted to type `float` and stored in a temporary variable before being multiplied by the `float` variable `avgWeight`. The result (still of type `float`) is then converted to `double` so that it can be assigned to the `double` variable `totalWeight`. This process is shown in Figure 2.9.