Lecture : C++ Programming Basics

This lecture introduces three such fundamentals: basic program construction, variables, and input/output (I/O). It also touches on a variety of other language features, including comments, arithmetic operators, the increment operator, data conversion, and library functions.

Basic Program Construction

Let's look at a very simple C++ program. This program is called FIRST, so its source file is `FIRST.CPP`. It simply prints a sentence on the screen. Here it is:

```
#include <iostream>
using namespace std;

int main()
    {
    cout << "Every age has a language of its own\n";
    return 0;
    }
```

- The FIRST program consists almost entirely of a single function called **main()**.
- The only parts of this program that are not part of the function are the first two lines—the ones that start with **#include** and **using**.
- The parentheses following the word main are the distinguishing feature of a function.
- The word **int** preceding the function name indicates that this particular function has a return value of type **int**.
- The *body* of a function is surrounded by *braces* (sometimes called *curly brackets*). These braces play the same role as the BEGIN and END keywords in some other languages.
- The program *statement* is the fundamental unit of C++ programming. There are two statements

  in the FIRST program: the line

  ```
  cout << "Every age has a language of its own\n";
  ```

and the return statement

```
return 0;
```

- The phrase in quotation marks, "`Every age has a language of its own\n`", is an example of a *string constant*.
- The '\n' character at the end of the string constant is an example of an escape sequence. It causes the next text output to be displayed on a new line.
- A sign (#) called a *preprocessor directive*. Recall that program statements are instructions to the computer to do something, such as adding two numbers or printing a sentence. A preprocessor directive, on the other hand, is an instruction to the compiler.
- The preprocessor directive `#include` tells the compiler to add the source file `IOSTREAM` to the `FIRST.CPP` source file before compiling.
- The directive

  ```
  using namespace std;
  ```

  says that all the program statements that follow are within the `std namespace`. Various program components such as cout are declared within this namespace. If we didn't use the using directive, we would need to add the `std` name to many program elements. For example, in the FIRST

  program we'd need to say

  ```
  std::cout << "Every age has a language of its own.";
  ```

  To avoid adding std:: dozens of times in programs we use the `using` directive instead.

Always Start with main()

When you run a C++ program, the first statement executed will be at the beginning of a function called **main().** The program may consist of many functions, classes, and other program elements, but on startup, control always goes to **main().** If there is no function called **main()** in your program, an error will be reported when you run the program.

The **main()** function may also contain calls to other standalone functions. This is shown in Figure 1.

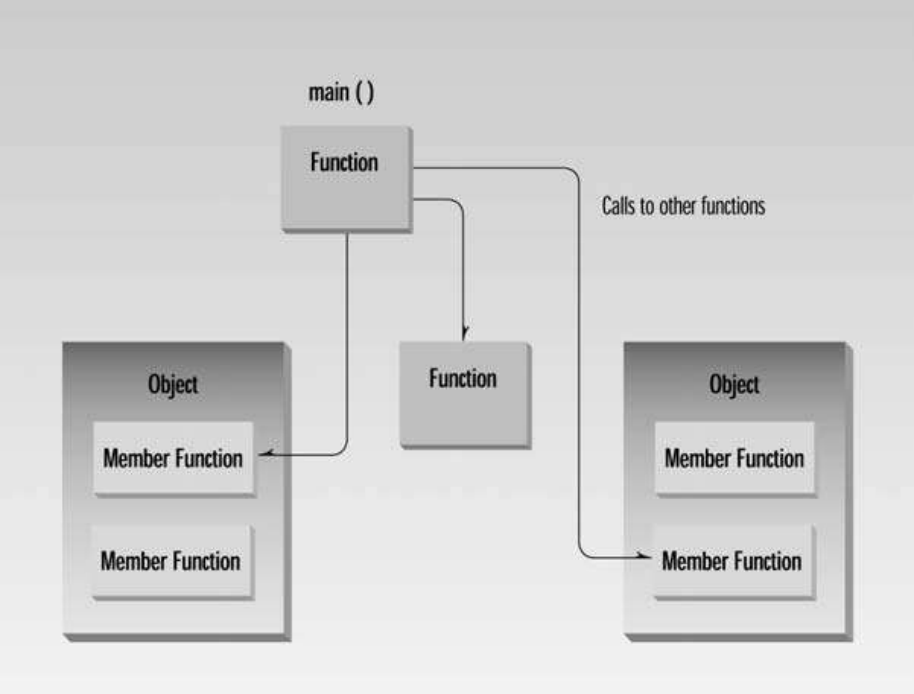

FIGURE 1 *Objects, functions, and main().*

Output Using cout

As you have seen, the statement

```
cout << "Every age has a language of its own\n";
```

The operator << is called the insertion or put to operator. It directs the contents of the variable on its right to the object on its left. In FIRST it directs the string constant "`Every age has a language of its own\n`" to `cout`, which sends it to the display. Figure 2 shows the result of using cout and the insertion operator <<.

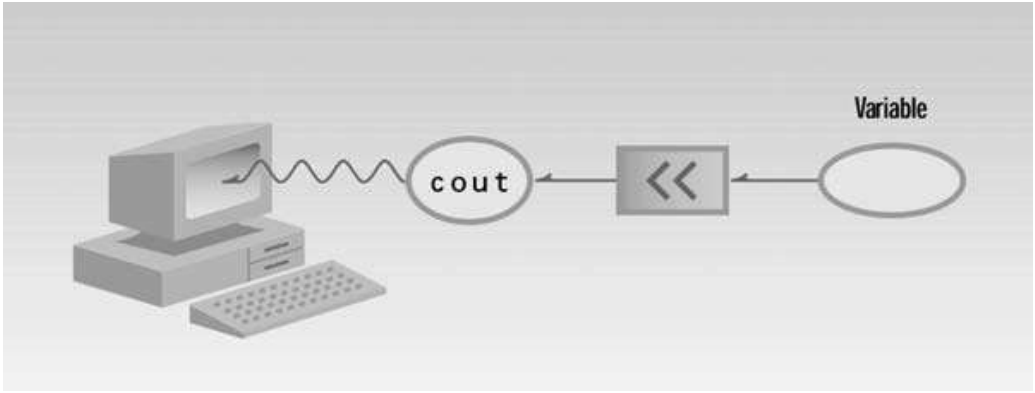

FIGURE 2
Output with cout.

Comments

Comments are an important part of any program. They help the person writing a program, and anyone else who must read the source file, understand what's going on. The compiler ignores comments, so they do not add to the file size or execution time of the executable program.

Comments start with a double slash symbol (//) and terminate at the end of the line.

Let's rewrite our FIRST program, incorporating comments into our source file. We'll call the new program COMMENTS:

```
// comments.cpp
// demonstrates comments
#include <iostream>              //preprocessor directive
using namespace std;             //"using" directive

int main()                       //function name "main"
   {                             //start function body
   cout << "Every age has a language of its own\n";  //statement
   return 0;                     //statement
   }                             //end function body
```

Integer Variables

- Variables are the most fundamental part of any language.
- A variable has a symbolic name and can be given a variety of values.
- Variables are located in particular places in the computer's memory. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. an `int` occupies 4 bytes (which is 32 bits) of memory. This allows an `int` to hold numbers in the range from –2,147,483,648 to 2,147,483,647. Figure 3 shows an integer variable in memory.

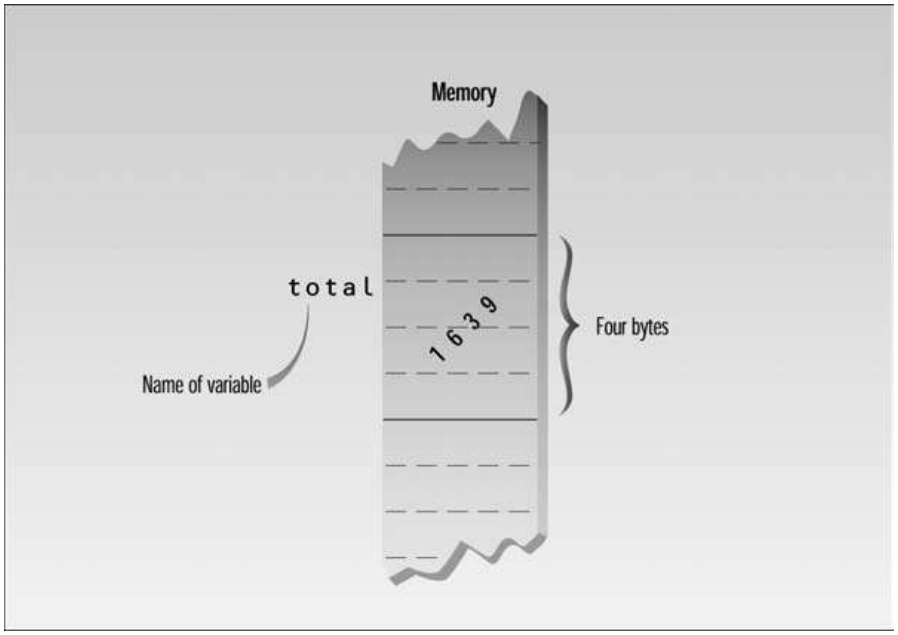

FIGURE 3
Variable of type int *in memory.*

- Most popular languages use the same general variable types, such as integers, floating-point numbers, and characters, so you are probably already familiar with the ideas behind them.

Here's a program that defines and uses several variables of type `int`:

```
// intvars.cpp
// demonstrates integer variables
#include <iostream>
using namespace std;

int main()
   {
   int var1;               //define var1
   int var2;               //define var2
    var1 = 20;              //assign value to var1
    var2 = var1 + 10;       //assign value to var2
    cout << "var1+10 is ";  //output text
    cout << var2 << endl;   //output value of var2
    return 0;
    }
```

- The statements
  ```
  int var1;
  int var2;
  ```

define two integer variables, var1 and var2. The keyword `int` signals the type of variable. These statements, which are called *declarations*, must terminate with a semicolon, like other program statements.
A *declaration* introduces a variable's name (such as `var1`) into a program and specifies its type (such as `int`).

Variable Names

- The program INTVARS uses variables named var1 and var2. The names given to variables are called *identifiers*.
- What are the rules for writing identifiers?

1. You can use upper- and lowercase letters, and the digits from 1 to 9.
2. You can also use the underscore (_).
3. The first character must be a letter or underscore.
4. Identifiers can be as long as you like, but most compilers will only recognize the first few hundred characters.
5. The compiler distinguishes between upper- and lowercase letters, so Var is not the same as var or VAR.
6. You can't use a C++ keyword as a variable name. A *keyword* is a predefined word with a special meaning. `int, class, if, and while` are examples of keywords.
7. A variable's name should make clear to anyone reading the listing the variable's purpose and how it is used.

Assignment Statements

- The statements
```
var1 = 20;
var2 = var1 + 10;
```
assign values to the two variables. The equal sign (=), as you might guess, causes the value on the right to be assigned to the variable on the left.
- The number 20 is an *integer constant*. Constants don't change during the course of the program.
- In the second program line shown here, the plus sign (+) adds the value of var1 and 10, in which 10 is another constant. The result of this addition is then assigned to var2.

The endl Manipulator

- The last cout statement in the INTVARS program ends with an unfamiliar word: `endl.`
- It has the same effect as sending the '\n' character, but is somewhat clearer. Strictly speaking, endl (unlike '\n') also causes the output buffer to be

flushed, but this happens invisibly so for most purposes the two are equivalent.

Other Integer Types

- There are several numerical integer types besides type `int`. The two most common types are `long` and `short`. (Strictly speaking type `char` is an integer type as well, but we'll cover it separately.)
- Type `long` always occupies four bytes, which is the same as type `int` on 32-bit Windows systems. Thus it has the same range, from –2,147,483,648 to 2,147,483,647. However, if your program may need to run on a **16-bit** system such as MS-DOS, or on older versions of Windows, specifying type long will guarantee a four-bit integer type.
  If you want to create a constant of type long, use the letter L following the numerical value, as in

```
longvar = 7678L; // assigns long constant 7678 to longvar
```

- On all systems type `short` occupies two bytes, giving it a range of –32,768 to 32,767.

Character Variables

- Type `char` stores integers that range in value from –128 to 127. Variables of this type occupy only 1 byte (eight bits) of memory.
- they are much more commonly used to store ASCII characters.
- Character constants use single quotation marks around a character, like 'a' and 'b'. (Note that this differs from string constants, which use double quotation marks.) When the C++ compiler encounters such a character constant, it translates it into the corresponding ASCII code. The constant 'a' appearing in a program, for example, will be translated into 97, as shown in Figure 4.
- Character variables can be assigned character constants as values.

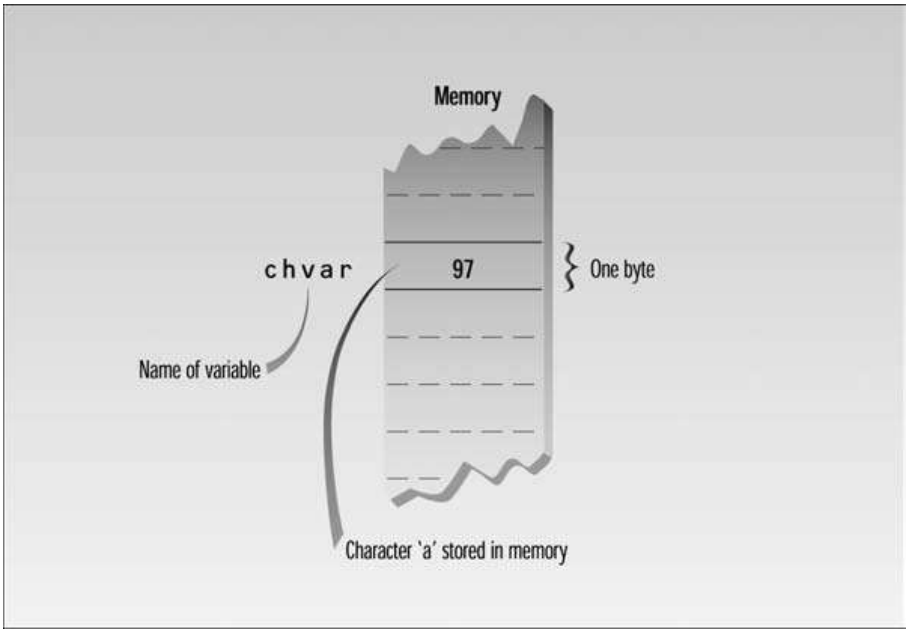

FIGURE 4
Variable of type char *in memory.*

- The following program shows some examples of character constants and variables.

```
// charvars.cpp
// demonstrates character variables
#include <iostream>        //for cout, etc.
using namespace std;

int main()
   {
   char charvar1 = 'A';    //define char variable as character
   char charvar2 = '\t';   //define char variable as tab

   cout << charvar1;       //display character
   cout << charvar2;       //display character
   charvar1 = 'B';         //set char variable to char constant
   cout << charvar1;       //display character
   cout << '\n';           //display newline character
   return 0;
   }
```

The CHARVARS program prints the value of `charvar1` (‘A’) and the value of `charvar2` (a tab). It then sets charvar1 to a new value (‘B’), prints that, and finally prints the newline. The output looks like this:

```
A       B
```

Escape Sequences

- The character constants, '\t' and '\n', are example of an *escape sequence*.
- A tab causes printing to continue at the next tab stop.
- '\n', is sent directly to `cout` in the last line of the program.
- Table 1 shows a list of common escape sequences.

TABLE 1 Common Escape Sequences

Escape Sequence	*Character*
`\ a`	Bell (beep)
`\ b`	Backspace
`\ f`	Formfeed
`\ n`	Newline
`\ r`	Return
`\ t`	Tab
`\ \`	Backslash
`\ '`	Single quotation mark
`\ "`	Double quotation marks
`\ xdd`	Hexadecimal notation

- Here's an example of a quoted phrase in a string constant:

```
cout << "\"Run, Spot, run,\" she said.";
```

This translates to

```
"Run, Spot, run," she said.
```

- You can use the '`\xdd`' representation, where each d stands for a hexadecimal digit. For example, you'll find such a character listed as decimal number 178, which is hexadecimal number B2 in the ASCII table. This character would be represented by the character constant '`\xB2`'.

- **Input with `cin`**

  let's see how a program accomplishes input. The next example program asks the user for a temperature in degrees Fahrenheit, converts it to Celsius, and displays the result. It uses integer variables.

```
// fahren.cpp
// demonstrates cin, newline
#include <iostream>
using namespace std;

int main()
   {
   int ftemp;  //for temperature in fahrenheit

   cout << "Enter temperature in fahrenheit: ";
   cin >> ftemp;
   int ctemp = (ftemp-32) * 5 / 9;
   cout << "Equivalent in Celsius is: " << ctemp << '\n';
   return 0;
   }
```

- The statement

```
cin >> ftemp;
```

causes the program to wait for the user to type in a number.

- The >> is the *extraction* or *get from* operator. It takes the value from the stream object on its left and places it in the variable on its right.
- Here's some sample interaction with the program:

```
Enter temperature in fahrenheit: 212
Equivalent in Celsius is: 100
```

- Figure 5 shows input using `cin` and the extraction operator >>.

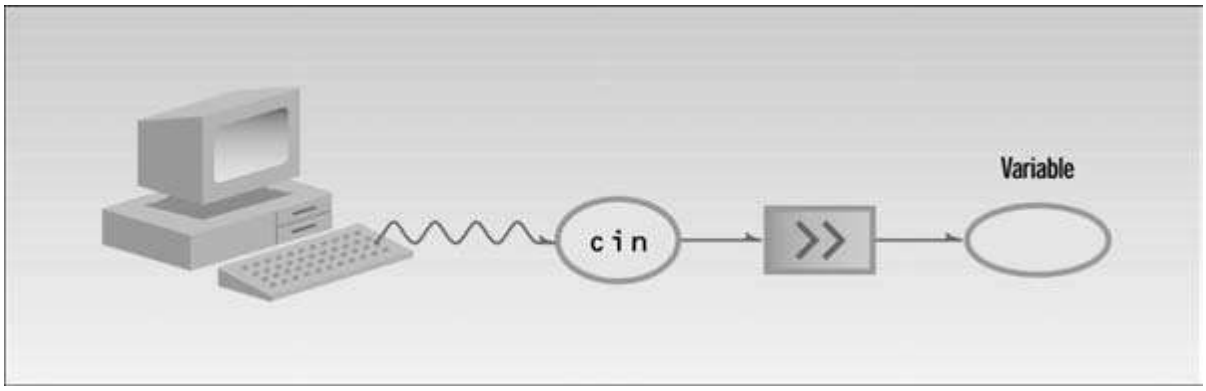

FIGURE 5
Input with `cin`.

- 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.

Floating Point Types

- 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*.
- There are three kinds of floating-point variables in C++: type float, type double, and type long double.

Type float

- Type float stores numbers in the range of about 3.4x10–38 to 3.4x1038, with a precision of seven digits. It occupies 4 bytes (32 bits) in memory.

  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;
}
```

Here's a sample interaction with the program:

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

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 1.7x10–308 to 1.7x10308 with a precision of 15 digits. Type long double is compiler-dependent but is often the same as double.

Floating-Point Constants

- The number 3.14159F in CIRCAREA is an example of a floating-point constant.
- 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. 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 times10^3.) 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}.

Type bool

- 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.
- 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 (<<) and `endl` manipulators; now we'll look at another one: `setw`, which changes the field width of output.
- 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.

The 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;
   }
```

Here's the output of WIDTH2:

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

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 8 shows how this looks.

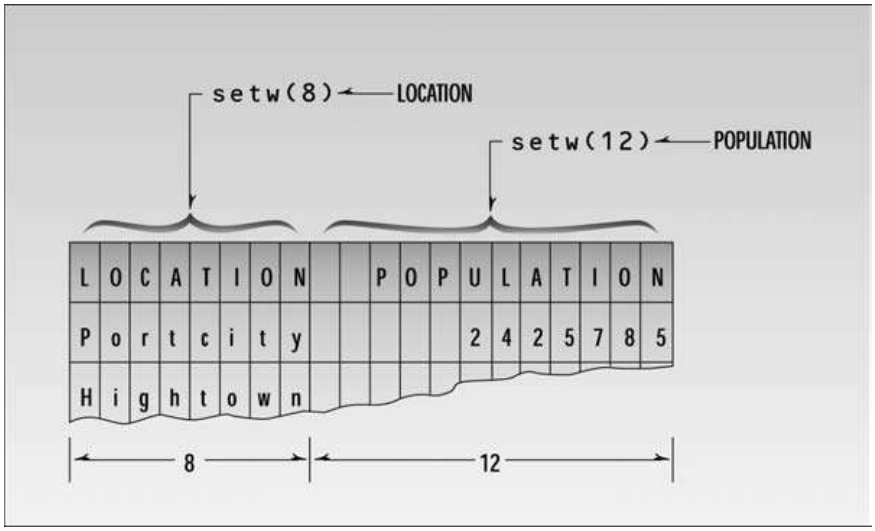

FIGURE 8
Field widths and setw.

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 3 shows the unsigned versions.

TABLE 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

- Exceeding the range of signed types can lead to obscure program bugs. 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;
   }
```

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
```

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.

- **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 4.

TABLE 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 9.

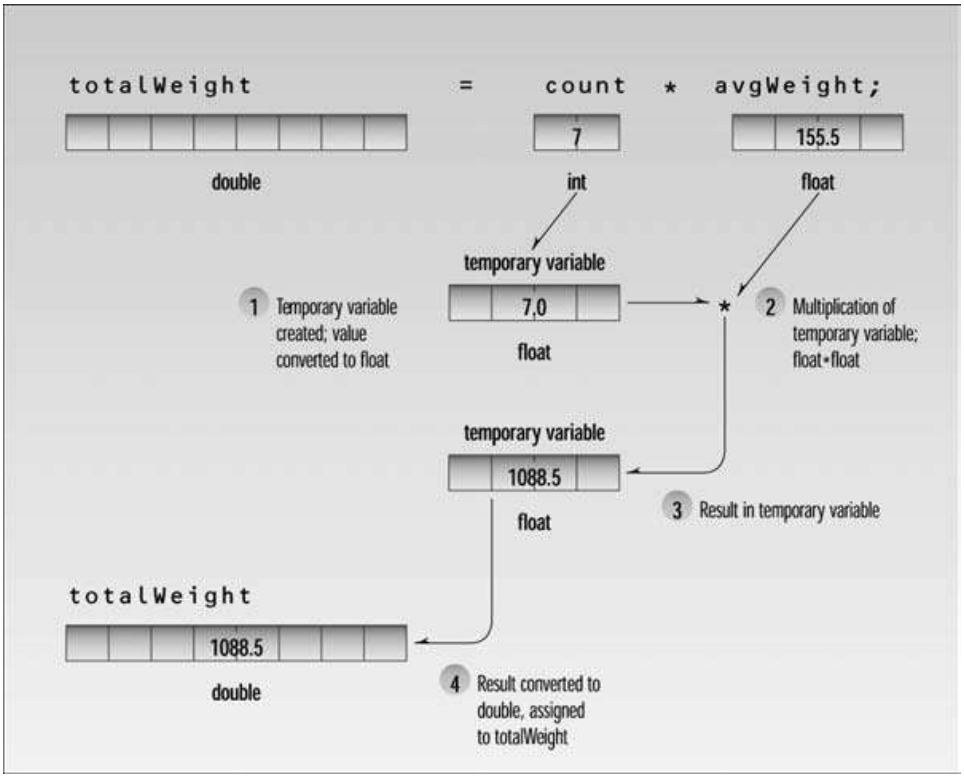

FIGURE 9
Data conversion.

- When we start to use objects, we will in effect be defining our own data types. We may want to use these new data types in mixed expressions, just as we use normal variables in mixed expressions. When this is the case, we must be careful to create our own conversion routines to change objects of one type into objects of another.