CHAPTER 2

Common Syntax and Semantic Errors

2.1 CHAPTER OBJECTIVES

- To understand the fundamental characteristics of syntax and semantic errors
- To be able to identify specific common syntax and semantic errors frequently encountered by beginning programmers
- To be able to interpret a syntax warning
- To be able to apply appropriate techniques to correct these common errors

2.2 SYNTAX ERRORS

A *syntax error* is a violation of the syntax, or grammatical rules, of a natural language or a programming language. The following sentence contains an error of English syntax:

```
I is going to the concert tonight.
```

If we write or say this sentence, other English speakers will know that we have used incorrect grammar, however they will still understand what we *mean*. Programming languages are not so forgiving, however. If we write a line of C++ code containing a syntax error, the compiler does *not* know what we mean. A syntax error is called a *fatal compilation error*, because the compiler cannot translate a C++ program into executable code if even one syntax error is present.

2.2.1 Syntax Errors: Summary of Important Points

- *How are they detected?* The compiler detects them when you try to compile your program.
- *Why do they occur?* The syntax rules of C++ have been violated.
- *Is there object-code generated?* No, so you cannot run the program.
- Solution: Find the line(s) containing the syntax error(s) using the compiler's flagged lines and error messages; using your textbook or other C++ reference as a guide, correct them.
- Remember, frequently, a syntax error occurs not in the line flagged by your compiler, but in some line *above* that line; it is often the previous line, but not necessarily.

2.2.2 Examples: Common Syntax Errors

Some syntax errors are very common, especially for beginning programmers, and the examples that follow should help you identify and correct many syntax errors in whatever program you are currently working on. The syntax diagrams in your C++ textbook or a C++ reference book should be your ultimate guide in correcting these types of errors.

Different compilers report syntax errors in different formats. For these examples, we will assume that the compiler displays errors for a C++ program named "myprog.cpp" in the following way:

```
Syntax Error: <description of error>
Line <line number here> of program myprog.cpp
```

The first line indicates that a syntax error message is being displayed and then gives a brief description of what the compiler thinks the error is. The second line will give the line number on which the compiler has identified the error.

Compilers that provide a graphical user interface (GUI) using windows and various graphical items to display information for you may display all such error messages in one window (which we will assume for our discussion here), or they may simply list the program with erroneous lines highlighted or pointed to by an arrow or other graphic, with written error messages shown off to the side. In any event, error messages displayed by different compilers generally are very similar.

An important note about compilers: Modern compilers typically are very accurate in identifying syntax errors and will help you enormously in correcting your code. However, compilers often present two difficult problems for new programmers: (1) they frequently can miss reporting an actual error on one line but get "thrown off track," then report errors on subsequent lines that are not truly errors; the compiler may then also display error messages which are incorrect; and (2) after encountering one true syntax error, compil-

ers often generate many incorrect syntax error messages; again, the compiler has been "thrown off track" by a particular error. Why does this occur? Basically, because a compiler is a very complex and sophisticated language-processing program, and no computer program can analyze any language as well as a human being can at this point in time.

What, then, is your best strategy for eliminating syntax errors?

- Display the current list of syntax errors (print it if you like)
- Start at the first error listed, try to correct it, and then re-compile your program; sometimes many errors will drop out after one error is fixed
- If you are having trouble with a particular error listed for a specific line, yet you are 100% sure that line is correct, then search for a syntax error in the lines ABOVE that line, starting with the line immediately preceeding the line under consideration, and working backwards; usually the actual error will be found in a line close to the line flagged, though not always
- Repeat this process until all errors are eliminated

Specific examples follow.

Missing Semicolon In the C++ code that follows, three declarations are given. Line numbers (chosen in all examples arbitrarily) are shown to the left of each line.

```
5    int num;
6    float value
7    double bigNum;
```

A C++ compiler would generate an error something like the following:

```
Syntax Error: semi-colon expected
Line 6 of program myprog.cpp
```

To fix this error, simply add a semicolon after the identifier value, as in

```
6    float value;
```

Undeclared Variable Name - version 1 If the preceding code were compiled and included an assignment statement, as in

```
5    int num;
6    float value
7    double bigNum;
8    bigNum = num + value;
```

we would see the following additional error message:

```
Syntax Error: undeclared identifier "bigNum"
Line 8 of program myprog.cpp
```

This is a situation in which there is actually no syntax error on the line flagged, and the real error occurs on a line above it. Line 8 is totally correct. If we correct the problem in line 6, the error reported for line 8 will drop out the next time we compile the program.

Undeclared Variable Name - version 2 What about the following?

```
5    int num;
6    float value;
7    double bigNum;
8    bignum = num + value;
```

We would see the error message

```
Syntax Error: undeclared identifier "bignum"
Line 8 of program myprog.cpp
```

This is a different problem; in this case, an error actually exists on line 8. The lowercase `n` in `bignum` must be changed to an uppercase `N`, or else the variable name does not match its declaration. Remember, in C++ declarations, lowercase letters are different from uppercase letters.

Undeclared Variable Name - version 3 Missing Reference to Namespace Consider the following program:

```
1    #include <iostream>
2    int main ( )
3    {
4         cout << "Hello World!!!";
5         return 0;
6    }
```

A compiler will generate an error message like this:

```
Syntax Error: undeclared identifer "cout"
Line 4 of program myprog.cpp
```

The problem is that `cout` is defined in a `namespace` named `std`. To correct this error, we need only add the following line, right after line 1 in the preceding program:

```
using namespace std;
```

Unmatched Parentheses Given the code

```
5    result = (firstVal - secondVal / factor;
```

the compiler would generate an error message like

```
Syntax Error: ')' expected
Line 5 of program myprog.cpp
```

We could correct this error with

```
5    result = (firstVal - secondVal) / factor;
```

Note that similar syntax errors can occur with unmatched braces, `{` and `}`.

Unterminated Strings It is easy to forget the last double quote in a string, as in

```
21    const string ERROR_MESSAGE = "bad data entered!;
```

or

```
45    cout << "Execution Terminated << endl;
```

Both will provoke the compiler to print something similar to

```
Syntax Error: illegal string constant
```

Both can be fixed by adding the terminating double quote to the string:

```
21    const string ERROR_MESSAGE = "bad data entered!";
```

or

```
45    cout << "Execution Terminated" << endl;
```

Left-Hand Side of Assignment does not Contain an L-Value Look at the following statements, where the intent of the assignment statement in line 7 is to calculate `x * y` and store the result in `product`:

```
6   double x = 2.0, y = 3.1415, product;
7   x * y = product;
```

Many C++ compilers will print an error message like

```
Syntax Error: not an l-value
Line 7 of program myprog.cpp
```

But what does this mean? An "l-value" roughly refers to a value that specifies the address of a location in memory where something can be stored. We can think of it as the Left side of an assignment, and what we know about the syntax of this statement is that the left-hand side must contain the name of a *variable*. Nothing else is valid in this position. Therefore, we correct this error using

```
7   product = x * y;
```

Value-Returning Function has no Return Statement A function declared with a return type must contain a return statement. Consider the following function, which is supposed to round its float parameter up or down appropriately:

```
int RoundFloat (float floatToRound)
{
      int roundedValue;
      roundedValue = int (floatToRound + 0.5);
}
```

This function calculates a correct result in the local variable `roundedValue`, but it never returns this result. Some compilers will merely generate a warning message for this function (to be discussed later in this chapter), others will generate a syntax error. In any event, the error can be corrected by adding a `return` statement, as in

```
int RoundFloat (float floatToRound)
{
      int roundedValue;
      roundedValue = int (floatToRound + 0.5);
      return (roundedValue);

}
```

Can't Convert int* to int** (and similar errors) Many times we need to pass an array to a function. Let's suppose we have the following declarations:

```
const int NUM_SCORES = 100;
typedef int ScoreList [NUM_SCORES];
void ProcessScores (ScoreList inputScores [NUM_SCORES]);
ScoreList scores;
```

Now let's assume we call the function `ProcessScores` and pass the array the variable `scores` like this:

```
10  ProcessScores (scores);
```

A compiler will generate an error message something like

```
Syntax Error: can't convert int* to int**
Line 10 of program myprog.cpp
```

What is wrong here? The data type of the argument `scores` and the data type of the corresponding parameter `inputScores` do *not* match. Array `scores` is a one-dimensional array of `int`. In the prototype we wrote earlier, `inputScores` is a two-dimensional array of `int`. The corrected version of the prototype for the function is

```
void ProcessScores (ScoreList inputScores);
```

Remember, the data types of arguments and parameters must generally match in three places—a function's prototype, its heading, and calls to that function. C++ does perform some type conversions automatically, but as part of defensive programming and good style, it is generally not wise to use data types that do not match.

Illegal Function Overloading This is another common error message generated when the data types of arguments and parameters in a function's prototype, its heading, and calls to that function, do not all match. As discussed in the previous example, it is typically best to utilize matching data types in all of these three program elements. Note that this error message may be generated when you are trying to call a a C++ *library function*, as well as when you try to call functions you have written yourself.

2.3 SYNTAX WARNINGS

From the discussion thus far, you know that a syntax error is a *fatal compilation error*—that is, the compiler cannot translate your program into executable code. Compilers also can generate syntax *warning* messages, which are not fatal errors, and are often very helpful in the debugging process. A syntax warning is displayed when the compiler has found that the syntax of some part of your code is valid, but it is potentially erroneous anyway. Compiler writers, being programmers themselves, are familiar with common programming errors and usually build some error checking of this type into the compiler. Let's look at a few examples.

2.3.1 Syntax Warnings: Summary of Important Points

- *How are they detected?* The compiler detects them when you compile your program.
- *Why do they occur?* The syntax rules of C++ have *not* been violated, but the compiler writers have built in special error checking for certain common programming errors, and the compiler has found a possible error of this type.

- *Is there object-code generated?* Yes, so you can run the program.
- *Solution*: Find the line(s) containing the syntax warning(s), and check them very carefully to see if you think they contain true errors.
- Remember, a syntax warning should *always* be taken seriously, because there is probably a real error in your code if the compiler issues a warning message.

2.3.2 Examples: Common Syntax Warnings

Using "=" when "==" is Intended Given the if statement

```
20    if (num = 100)
21        cout << "num equals 100";
22    else
23        cout << "num is not 100";
```

many compilers would display something like the following warning message:

```
Syntax Warning: assignment operator used in if expression
Line 20 of program myprog.cpp
```

What is the problem with this code? You may have noticed that the equal sign used in the `if` expression is actually an assignment operator, not the relational operator, which tests for equality. In this code, `num` is set to 100 because of the assignment, and the expression `num = 100` is *always* true, because the value of the expression is actually 100. The corrected code for this example would be

```
20   if (num == 100)
```

Loop has no Body Consider the following `for` loop, which is supposed to print out the numbers between 1 and 10, along with their squares:

```
9    int lcv;
10   for (lcv = 1; lcv <= 10; lcv++);
11   cout << lcv << " " << lcv * lcv
12        << endl;
```

This loop, if run, would print only

```
11     121
```

When we try to compile this loop, a good compiler would generate the following warning message:

```
Syntax Warning: for loop has no body
Line 10 of program myprog.cpp
```

We would correct this code by removing the erroneous semicolon, as follows:

```
10    for (int lcv = 1; lcv <= 10; lcv++)
```

Uninitialized Variable Consider the following code, which is supposed to read characters until the user hits `return`:

```
10    char ch;
11    while (ch != '\n')
12         cin >> ch;
```

When we try to compile this version, many compilers would generate the following warning message:

```
Syntax Warning: variable "ch" is uninitialized
Line 11 of program myprog.cpp
```

In this code, the variable `ch` is tested for a value when it has not received a value yet. This code, as it is, will not execute correctly. We can fix this problem using a priming read, as follows:

```
10    char ch;
11    cin >> ch;
12    while (ch != '\n')
13         cin >> ch;
```

All of the errors discussed here which generate warning messages are what we call *semantic* errors, to be discussed in the next section.

To illustrate what we have looked at so far in this chapter, Figure 2.1 shows the syntax errors and warnings window generated by the Metrowerks CodeWarrior Professional Release 5 compiler for a specific program. Looking at this figure, notice in particular how errors and warnings are flagged with different graphics elements. Figure 2.2 shows the window generated by the Microsoft Visual C++ 6 compiler for the same program. Notice how the two compilers produce similar, but not identical, error reports. Other compilers will also produce similar error and warning messages.

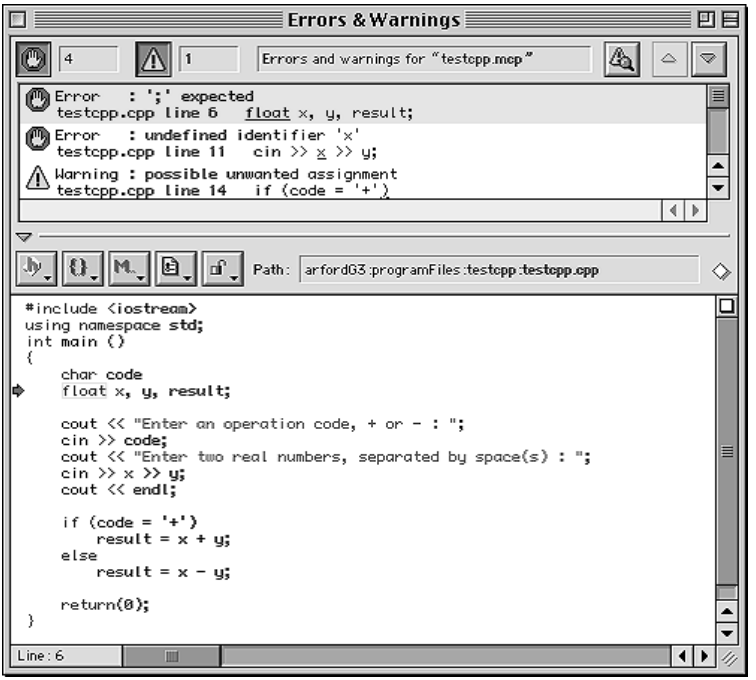

Figure 2.1

Example Program in Metrowerks CodeWarrior Syntax Errors and Warnings Window

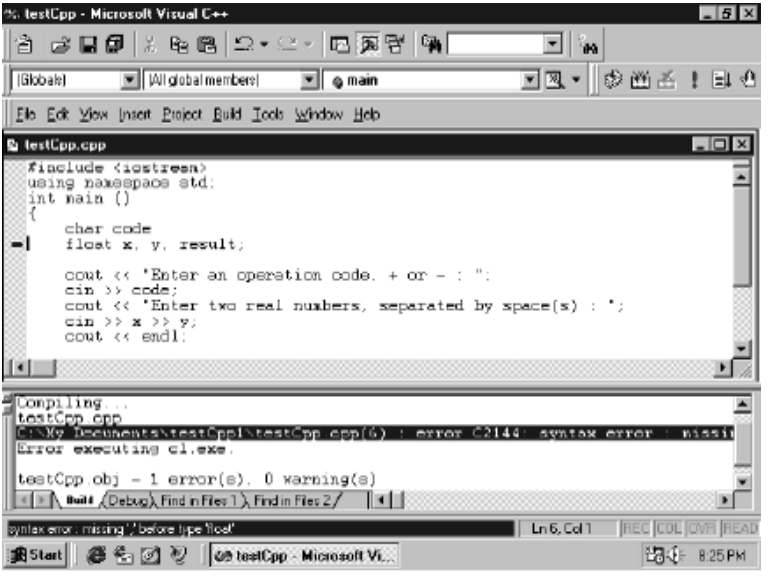

Figure 2.2

Example Program in Microsoft Visual C++ Syntax Errors and Warnings Window

2.4 SEMANTIC ERRORS

A *semantic error* is a violation of the rules of *meaning* of a natural language or a programming language. The following sentence contains an error of English semantics:

```
My refrigerator just drove a car to Chicago.
```

If we write or say this sentence, other English speakers may begin to wonder about our sanity, but they will nevertheless know that our syntax is perfectly correct! Since a compiler only checks for correct use of syntax, it is not able to evaluate whether or not we have written code whose *meaning* is correct. Semantic errors are much harder to detect and correct than syntax errors, and they are also more common.

When there are semantic errors in a C++ program, the compiler *does* translate the program into executable code. Most of the time, semantic errors do NOT generate compiler warnings. When the program is run, however, it does not work correctly.

2.4.1 Summary of Important Points

- *How are they detected?* Semantic errors usually are detected by the programmer or user of the program, often while reading output and finding it is incorrect.
- *Why do they occur?* The syntax rules of C++ have been correctly followed, but the meaning of the code is incorrect (e.g., faulty algorithms, algorithms not translated into C++ correctly, values calculated or input erroneously, flow of control is wrong, etc.).
- *Is there object-code generated?* Yes, so you can run the program.
- Solution: Find the line(s) containing the semantic error(s), using extra print statements, hand tracing, or an interactive debugger if needed, and correct them.
- Remember, semantic errors are undoubtedly the most common type of error, and they are the hardest to find and correct, so prevention in the form of good design and defensive programming are often your best tools.

2.4.2 Examples: Common Semantic Errors

Infinite Loop An infinite loop, which repeats indefinitely, is often created when a programmer writes a loop in which the expression tested never becomes false. For example, consider the following code, which is intended to read a lower case 'y' or 'n' from the user:

```
char response;
cout << "Please enter (y)es or (n)o -> ";
cin >> response;
while ((response != 'y') || (response != 'n'))
{
    cout << "Please try again. Enter (y)es or (n)o -> ";
    cin >> response;
}
```

The expression

```
(response != 'y') || response != 'n')
```

is always true, regardless of what input is entered by the user. If the user enters a 'y', then the first part of the expression is false, but the second part of the expression is true. Thus, the entire expression is true because of the OR operation. Because of this property, the loop is infinite. The following expression is the corrected version, which merely substitutes AND for OR and is false when the user enters either a 'y' or an 'n', allowing the loop to exit:

```
(response != 'y') && (response != 'n')
```

Misunderstanding of Operator Precedence Misunderstanding of the operator precedence rules often can lead to expressions that evaluate data incorrectly. Consider the following code, which is intended to calculate the miles per gallon achieved by a car:

```
    float startMileage,
          endMileage,
          gallonsUsed,
          milesPerGallon;
cout << "Please enter the starting mileage, the ending"
     << " mileage, and the number of gallons used by the"
     << " car -> ";
cin >> startMileage >> endMileage >> gallonsUsed;
milesPerGallon = endMileage - startMileage / gallonsUsed;
```

Since division has higher precedence than subtraction, the final assignment statement evaluates incorrectly. The corrected version is as follows:

```
milesPerGallon = (endMileage - startMileage) / gallonsUsed;
```

Programmers often make liberal use of parentheses to prevent this kind of error in the first place, as part of defensive programming.

Dangling Else The "dangling else" is a very common and subtle error in the flow of control of a nested `if` statement. Remember, compilers ignore indentation, and pair an *else* clause with the most recent unmatched *then* clause. The following code is intended to print out the word "both" if the `bool` variables `relative` and `my friend` are both true, and print out "neither" if both are false.

```
if (relative)
    if (my friend)
        cout << "both";
else
    cout << "neither";
```

When this code is run, it prints "both" correctly when both `bool` variables are true. However, if both variables are false, it prints nothing. Further, when `relative` is true, but `friend` is false, it prints "neither!" There are many ways to fix or rewrite this code, but let's consider one very common correction technique here. The following code forces the compiler to pair the `else` with the first `if`, instead of with the second `if`, and therefore works correctly:

```
if (relative)

{
    if (my friend)
        cout << "both";
}
else
    cout << "neither";
```

Off-By-One Error The off-by-one error generally describes a loop that iterates one fewer or one more time than is correct. Consider the following code, which is supposed to read in 50 values from the user, and keep a running total:

```
const int NUM_VALUES = 50;
int lcv,
    someValue,
    total = 0;
for (lcv = 1; lcv < NUM_VALUES; lcv++)
{
    cout << "Enter an integer ->";
    cin >> someValue;
    total = total + someValue;
}
```

What happens when this code is executed? The user is prompted 49 times, and

49 values are read in and summed. The following small change in the `for` loop heading fixes this problem:

```
for (lcv = 1; lcv <= NUM_VALUES; lcv++)
```

Or, alternatively, we can write

```
for (lcv = 0; lcv < NUM_VALUES; lcv++)
```

Code inside a Loop that does not Belong There The following code, adapted from the previous example, is intended to read in a series of 50 numbers entered by the user, calculate the running total, and print out both the final total and the average value of all the numbers:

```
const int NUM_VALUES = 50;
int lcv,
    someValue,
    total = 0,
    average;
for (lcv = 1; lcv <= NUM_VALUES; lcv++)
{
    cout << "Enter an integer ->";
    cin >> someValue;
    total = total + someValue;
    average = total / NUM_VALUES;
    cout << "Total is: " << total << endl;
    cout << "Average is: " << average;
}
```

When this code is executed, it produces a disturbing result! The output is very long, because the total and average are both printed out 50 times. Moreover, they are correct only the last time they are printed. What is wrong? The last three lines inside the loop do not belong there, they must be placed outside the loop and after it for the code to work correctly. The corrected version of this code is as follows:

```
for (lcv = 1; lcv <= NUM_VALUES; lcv++)
{
    cout << "Enter an integer ->";
    cin >> someValue;
    total = total + someValue;
}
average = total / NUM_VALUES;
cout << "Total is: " << total << endl;
cout << "Average is: " << average;
```

Not Using a Compound Statement When One is Required The following code is intended to read in a text file and echoprint its contents (for this example, assume that `infile` is the file variable name and the file has just been opened successfully):

```
char ch;
infile.get (ch);
while (infile)            // while input stream has not failed
    cout << ch;           // echoprint the current character
    infile.get (ch);      // read the next character
```

The programmer has indented this code to show that both the `cout` and the `get` calls are inside the loop. However, the compiler interprets it differently. When executed, this code reads one character from the file (assuming the file does contain at least one character) and then enters an infinite loop, printing that same character over and over indefinitely. The following code contains the addition of a compound statement, and thus is correct:

```
infile.get (ch);
while (infile)          // while input stream has not failed
{
    cout << ch;         // echoprint the current character
    infile.get (ch);    // read the next character
}
```

Array Index Bounds Error The following code is intended to read in seven temperature values, one for each day of the week, and load them into a seven-element array.

```
const int NUM_DAYS = 7;
int temperatures[NUM_DAYS];
int count;
for (count = 1; count <= NUM_DAYS; count++)
{
    cout << "Enter a temperature: ";
    cin >> temperatures [count];
}
```

Let's say that we execute this code as it is written. Many problems may result, because the code contains a fundamental error—the array's correct indices range from 0 through 6, while the array indices used in the `for` loop range from 1 through 7. The array element `temperatures[0]` is never used. The attempt to access an array element with `temperatures[7]` is wrong, because `temperatures[7]` does not exist.

The results of reading in the array with this loop and then attempting to use the array later in the same program are somewhat unpredictable and are partly dependent on what indices are used later on. The program may even crash (a runtime error). We may find that an error message such as "Bus Error" or "Segmentation Fault" is printed at the time of the crash. Alternatively, the program may keep running, and the value of the count may be overwritten by whatever value the user types in for the last array element, which causes other unpredictable behavior. The fundamental problem is that when invalid array indices are used, a C++ program will access memory locations that are not valid portions of the array. C++ does not prevent such array index errors by performing what is called "bounds checking." To avoid this problem, you must be very careful to write your code so that only valid indices are used. We can correct the code above by using this conventional `for` loop control for such an array:

```
for (count = 0; count < NUM_DAYS; count++)
```

A Final Note

This concludes our discussion of syntax errors and warnings, and semantic errors. While working on your programming assignments on a computer, you may wish to keep this book with you as a helpful guide in determining what exactly the compiler is trying to tell you when you see one of its sometimes obscure error messages. In addition, by studying the examples of all of the common errors described in this chapter, you may be able to prevent many of them from occurring in the first place, as you will have the knowledge and the tools to avoid them altogether.