Chapter 14
Expression Trees

"What's twice eleven?" I said to Pooh.
("Twice what?" said Pooh to Me.)
"I think *it ought to be twenty-two."*
"Just what I think myself," said Pooh.

— A. A. Milne, "Us Two," *Now We Are Six*, 1927

Objectives

- To appreciate how you can use trees to interpret expressions in a programming language.
- To recognize the recursive structure of expressions and understand how you can represent that structure in C++.
- To learn how to use C++ inheritance to structure a set of related classes.
- To understand the process of parsing the text representation of an expression into its internal form.
- To be able to write simple recursive functions that manipulate expression trees.

Chapter 13 focused on binary search trees because they provide a simple context for explaining how trees work. Trees occur in many other programming contexts as well. In particular, trees often show up in the implementation of compilers because they are ideal for representing the hierarchical structure of a program. By exploring this topic in some detail, you will learn quite a bit, not only about trees, but also about the compilation process itself. Understanding how compilers work removes some of the mystery surrounding programming and makes it easier to understand the process as a whole.

Unfortunately, designing a complete compiler is far too complex to serve as a useful illustration. Typical commercial compilers require many person-years of programming, much of which is beyond the scope of this text. Even so, it is possible to give you a sense of how they work—and, in particular, of how trees fit into the process—by making the following simplifications:

- *Having you build an interpreter instead of a compiler.* As described in the section on "What is C++?" in Chapter 1, a compiler translates a program into machine-language instructions that the computer can then execute directly. Although it has much in common with a compiler, an **interpreter** never actually translates the source code into machine language but simply performs the operations necessary to achieve the effect of the compiled program. Interpreters are generally easier to write, but have the disadvantage that interpreted programs tend to run much more slowly than their compiled counterparts.
- *Focusing only on the problem of evaluating arithmetic expressions.* A full-scale language translator for a modern programming language—whether a compiler or an interpreter—must be able to process control statements, function calls, type definitions, and many other language constructs. Most of the fundamental techniques used in language translation, however, are illustrated in the seemingly simple task of translating arithmetic expressions. For the purpose of this chapter, arithmetic expressions will be limited to constants and variables combined using the operators +, –, *, /, and = (assignment). As in C++, parentheses may be used to define the order of operations, which is otherwise determined by applying precedence rules.
- *Limiting the types used in expressions to integers.* Modern programming languages like C++ allow expressions to manipulate data of many different types. In this chapter, all data values are assumed to be of type `int`, which simplifies the structure of the interpreter considerably.

14.1 Overview of the interpreter

The goal of this chapter is to show you how to design a program that accepts arithmetic expressions from the user and then displays the results of evaluating those expressions. The basic operation of the interpreter is therefore to execute the following steps repeatedly as part of a loop in the main program:

1. Read in an expression from the user and translate it into an appropriate internal form.
2. Evaluate the expression to produce an integer result.
3. Print the result of the evaluation on the console.

This iterated process is characteristic of interpreters and is called a **read-eval-print loop.**

At this level of abstraction, the code for the read-eval-print interpreter is extremely simple. Although the final version of the program will include a little more code than is shown here, the following main program captures the essence of the interpreter:

```
int main() {
   while (true) {
      expressionT exp = ReadExp();
      int value = exp->eval();
      cout << value << endl;
      delete exp;
   }
   return 0;
}
```

As you can see, the idealized structure of the main program is simply a loop that calls functions to accomplish each phase of the read-eval-print loop. In this formulation, the task of reading an expression is indicated by a call to the function **ReadExp**, which will be replaced in subsequent versions of the interpreter program with a somewhat longer sequence of statements. Conceptually, **ReadExp** is responsible for reading an expression from the user and converting it into its internal representation, which takes the form of an **expressionT** value. The task of evaluating the expression falls to the **eval** method, which returns the integer you get if you apply all the operators in the expression in the appropriate order. The print phase of the read-eval-print loop is accomplished by a simple stream insertion to displays the result.

At this point, you don't yet have any detailed sense of what the **expressionT** type is or how it is represented. From the appearance of the -> operator in the line

```
int value = exp->eval();
```

you can infer that **expressionT** is a pointer to some type that has a method called **eval**. The existence of an **eval** method, moreover, tells you that **expressionT** type points to objects of some class, but the details are as yet undefined. That, of course, is how it should be. As a client of the expression package, you are less concerned with how expressions are implemented than you are with how to use them. For the moment, it is best to think of **expressionT** as an abstract type for which you just happen to use pointer syntax to refer to its methods. The reasons underlying that decision will come in time.

The operation of the **ReadExp** function consists of the three following steps:

1. *Input*. The input phase consists of reading in a line of text from the user, which can be accomplished with a simple call to the **GetLine** function from **simpio.h**.
2. *Lexical analysis*. The lexical analysis phase consists of dividing the input line into individual units called *tokens*, each of which represents a single logical entity, such as an integer constant, an operator, or a variable name. Fortunately, all the facilities required to implement lexical analysis are provided by the **Scanner** class introduced in Chapter 4.
3. *Parsing*. Once the line has been broken down into its component tokens, the parsing phase consists of determining whether the individual tokens represent a legal expression and, if so, what the structure of that expression is. To do so, the parser must determine how to construct a valid parse tree from the individual tokens in the input.

It would be easy enough to implement **ReadExp** as a single function that combined these steps. In many applications, however, having a **ReadExp** function is not really what you want. Keeping the individual phases of **ReadExp** separate gives you more flexibility in designing the interpreter structure. The complete implementation of the main module for the interpreter therefore includes explicit code for each of the three phases. In addition, the evaluator will need access to information about the state of the evaluation