Introduction to C++ Inheritance

This handout is a near-verbatim copy of Chapter 14 from this year's CS106B/X course reader. Those who've taken CS106X recently were taught this material, but CS106B skipped over it, and because the chapter is new to the reader as of autumn 2006, those who took CS106 prior were denied inheritance coverage as well. Eric Roberts wrote the first version of the chapter, and Julie Zelenski updated everything last summer to use C++ inheritance. My edits are minimal—all I've done is updated the examples to make use of STL containers as opposed to CS106-specific ones.

CS106B/X spend a good amount of time on binary search trees, because they serve to explain 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 programming 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.* 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 purposes of this handout, 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 our version, all data values are assumed to be of type **`int`**, which simplifies the structure of the interpreter considerably.

Overview of the interpreter

The primary goal of this larger example 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(int argc, char *argv[])
{
   while (true) {
      Expression *exp = ReadExp();
      int value = exp->eval();
      cout << value << endl;
   }

   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 **`Expression`** object. The task of evaluating the expression falls to the **`eval`** member function, 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.

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 **`getline`**.
2. *Lexical analysis.* The lexical analysis phase consists of dividing user input 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 an object-oriented version of **`streamtokenizer`** type you've seen in CS107. (The details of the **`streamtokenizer`** are being downplayed here, since the mechanics of tokenizing aren't all that interesting and isn't our focus.)
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, as shown in here:

```
/**
 * File: interp.cc
 * ---------------
 * This program simulates the top level of a programming
 * language interpreter.  The program reads an expression,
 * evaluates the expression, and displays the result.
 */

static const string kNewLineCharacters("\n\r");
int main(int argc, char *argv[])
{
   map<string, int> varTable;
   streamtokenizer st(cin, kNewLineCharacters, true);

   while (true) {
      cout << "=> ";
      Expression *exp = ParseExp(st);
      if (exp == NULL) break;
      cout << exp->toString() << " evaluates to "
           << exp->eval(varTable) << endl;
   }

   return 0;
}
```

A sample run of the interpreter might look like this:

```
=> x = 6↲
(x = 6) evaluates to 6
=> y = 10↲
(y = 10) evaluates to 10
=> 2 * x + 3 * y↲
((2 * x) + (3 * y)) evaluates to 42
=> 2 * (x + 3) * y↲
((2 * (x + 3)) * y) evaluates 180
=> quit↲
```

As the sample run illustrates, the interpreter allows assignment to variables and adheres to C++'s precedence conventions by evaluating multiplication before addition.

Although the code for the **`main`** program is straightforward, you still have some unfinished business. First, you need to think about exactly what expressions are and how to represent them as objects and how to implement their member functions. Then, you have to implement the **`ParseExp`** function. Because each of these problems involves some subtlety, completing the interpreter will take up the remainder of the handout.

Understanding the abstract structure of expressions

Your first task in completing the interpreter is to understand the concept of an expression and how that concept can be represented as an object. As is often the case when you are thinking about a programming abstraction, it makes sense to begin with the insights you have acquired about expressions from your experience as a C++ programmer. For example, you know that the lines

```
0
2 * 11
3 * (a + b + c)
x = x + 1
```

represent legal expressions in C++. At the same time, you also know that the lines

```
2 * (x - y
17 k
```

are not expressions; the first has unbalanced parentheses, and the second is missing an operator. An important part of understanding expressions is articulating what constitutes an expression so that you can differentiate legal expressions from malformed ones.

A recursive definition of expressions

As it happens, the best way to define the structure of a legal expression is to adopt a recursive perspective. A sequence of symbols is an expression if it has one of the following forms:

1. An integer constant
2. A variable name
3. An expression enclosed in parentheses
4. A sequence of two expressions separated by an operator

The first two possibilities represent the simple cases for the recursive definition. The remaining possibilities, however, define an expression in terms of simpler ones.

To see how you might apply this recursive definition, consider the following sequence of symbols:

```
y = 3 * (x + 1)
```

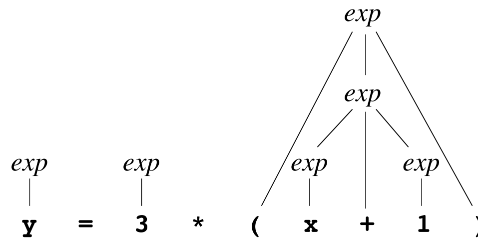
Does this sequence constitute an expression? You know from experience that the answer is yes, but you can use the recursive definition of an expression to justify that answer. The integer constants 3 and 1 are expressions according to rule #1. Similarly, the variable names **x** and **y** are expressions as specified by rule #2. Thus, you already know that the expressions marked by the symbol *exp* in the following diagram are expressions, as defined by the simple-case rules:

```
exp     exp           exp     exp
 |       |             |       |
 y   =   3   *   (     x   +   1   )
```

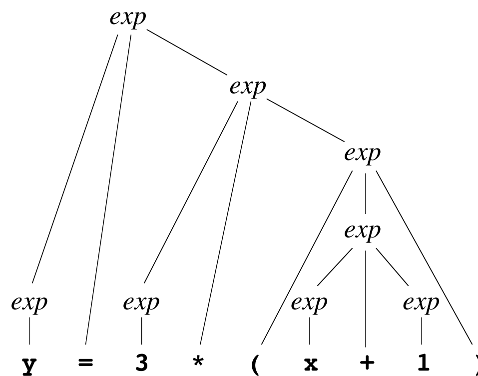
At this point, you can start to apply the recursive rules. Given that **x** and **1** are both expressions, you can tell that the string of symbols **x + 1** is an expression by applying rule #4, because it consists of two expressions separated by an operator. You can record this observation in the diagram by adding a new expression marker tied to the parts of the expression that match the rule, as shown:

```
                                exp
                               / | \
exp     exp           exp      |   exp
 |       |             |       |    |
 y   =   3   *   (     x   +   1   )
```

The parenthesized quantity can now be identified as an expression according to rule #3, which results in the following diagram:



By applying rule #4 two more times to take care of the remaining operators, you can show that the entire set of characters is indeed an expression, as follows:



As you can see, this diagram forms a tree. A tree that demonstrates how a sequence of input symbols fits the syntactic rules of a programming language is called a **parse tree.**

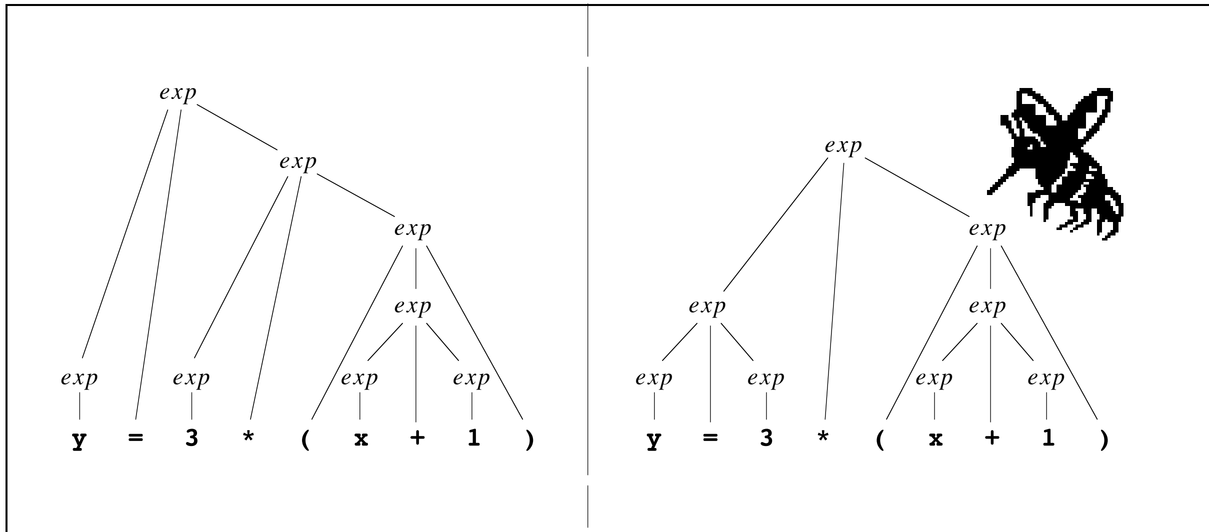
Ambiguity

Generating a parse tree from a sequence of symbols requires a certain amount of caution. Given the four rules for expressions outlined in the preceding section, you can form more than one parse tree for the expression

```
y = 3 * (x + 1)
```

Although the tree structure shown at the end of the last section presumably represents what the programmer intended, it is just as valid to argue that `y = 3` is an expression according to rule #4, and that the entire expression therefore consists of the expression `y = 3`, followed by a multiplication sign, followed by the expression `(x + 1)`. This argument ultimately reaches the same conclusion about whether the input line represents an expression, but generates a different parse tree. Both parse trees are shown in a diagram on the next page. The parse tree on the left is the one generated in the last section and corresponds to what a C++ programmer means by that expression. The parse tree on the right represents a legal application of the expression rules but reflects an incorrect ordering of the operations, given C++'s rules of precedence.

Intended parse tree and a legal but incorrect alternative



The problem with the second parse tree is that it ignores the mathematical rule specifying that multiplication is performed before assignment. The recursive definition of an expression indicates only that a sequence of two expressions separated by an operator is an expression; it says nothing about the relative precedence of the different operators and therefore admits both the intended and unintended interpretations. Because it allows multiple interpretations of the same string, the informal definition of expression given in the preceding section is said to be **ambiguous.** To resolve the ambiguity, the parsing algorithm must include some mechanism for determining the order in which operators are applied.

The question of how to resolve the ambiguity in an expression during the parsing phase is discussed in the section on "Parsing an expression" later in this handout. At the moment, the point of introducing parse trees is to provide some insight into how you might represent an expression as a data structure. To this end, it is extremely important to make the following observation about the parse trees in the diagram below: the trees themselves are not ambiguous. The structure of each parse tree explicitly represents the structure of the expression. The ambiguity exists only in deciding how to generate the parse tree from the original string of constants, variables, and operators. Once you have the correct parse tree, its structure contains everything you need to understand the order in which the operators need to be applied.

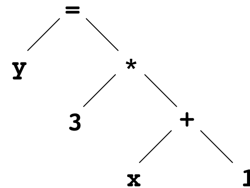
Expression trees

In fact, parse trees contain more information than you need in the evaluation phase. Parentheses are useful in determining how to generate the parse tree but play no role in the evaluation of an expression once its structure is known. If your concern is simply to find the value of an expression, you do not need to include parentheses within the structure. This observation allows you to simplify a complete parse tree into an abstract

structure called an **expression tree** that is more appropriate to the evaluation phase. In the expression tree, nodes in the parse tree that represent parenthesized sub-expressions are eliminated. Moreover, it is convenient to drop the **exp** labels from the tree and instead mark each node in the tree with the appropriate operator symbol. For example, the intended interpretation of the expression

```
y = 3 * (x + 1)
```

corresponds to the following expression tree:



The structure of an expression tree is similar in many ways to the binary search tree, but there are also some important differences. In the binary search tree, every node had the same structure. In an expression tree, there are three different types of nodes, as follows:

1. *Integer expressions* represent integer constants, such as 3 and 1 in the example tree.
2. *Identifier expressions* represent the names of variables and are presumably represented internally by a string.
3. *ŏmpound expressions* represent the application of an operator to two operands, each of which is an arbitrary expression tree.

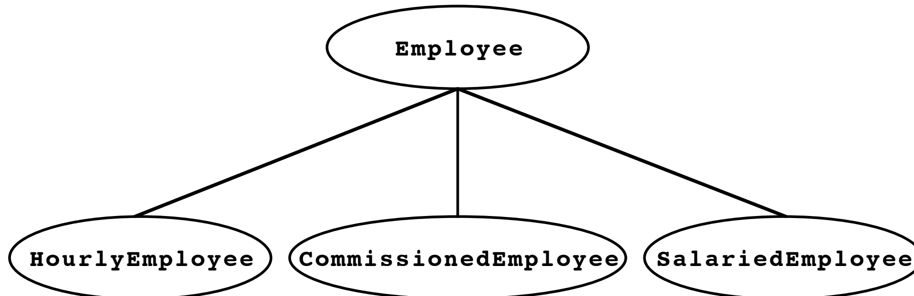
Each of these types corresponds to one of the rules in the recursive formulation of an expression. The `Expression` class must make it possible for clients to work with expressions of all three types. Similarly, the underlying implementation must somehow make it possible for different node structures to coexist within the tree. To represent such a structure, you need to define a representation for expressions that allows them to have different structures depending on their type. An integer expression, for example, must include the value of the integer as part of its internal structure. An identifier expression must include the name of the identifier. A compound node must include the operator along with the left and right sub-expressions. Defining a single abstract type that allows expression to take on these different underlying structures requires you to learn a new aspect of C++'s type system, which is introduced in the next section.

Class hierarchies and inheritance

Object-oriented languages like C++ allow you define hierarchical relationships among classes. Whenever you have a class that provides some of the functionality you need for a particular application, you can define new classes that are derived from the original class, but which specialize its behavior in some way. The derived classes are known as

subclasses of the original class, which in turn becomes the **superclass** for each of its subclasses.

As an example, suppose that you have been charged with designing an object-oriented payroll system for a company. You might begin by defining a general class called **`Employee`**, which encapsulates the information about an individual worker along with methods that implement operations required for the payroll system. These operations could include simple member functions like **`getName`**, which returns the name of an employee, along with more complicated member functions like **`computePay`**, which calculates the pay for an employee based on data stored within each **`Employee`** object. In many companies, however, employees fall into several different classes that are similar in certain respects but different in others. For example, a company might have hourly employees, commissioned employees, and salaried employees on the same payroll. In such companies, it might make sense to define subclasses for each employee category as illustrated by the following diagram:



Each of the classes **`HourlyEmployee`**, **`CommissionedEmployee`**, and **`SalariedEmployee`** is a subclass of the more general **`Employee`** class, which acts as their common superclass.

By default, each subclass **inherits** the behavior its superclass, which means that the member functions and internal data structure of the superclass are also available to its subclasses. In cases in which the behavior of a subclass needs to differ from its superclass, the designer of the subclass can define entirely new member functions for that subclass or **override** existing member functions with modified ones. In the payroll example, all three subclasses will presumably inherit the **`getName`** member function from the **`Employee`** superclass. All employees, after all, have a name. On the other hand, it probably makes sense to write separate **`computePay`** member functions for each subclass, because the computation is likely to be different in each case.

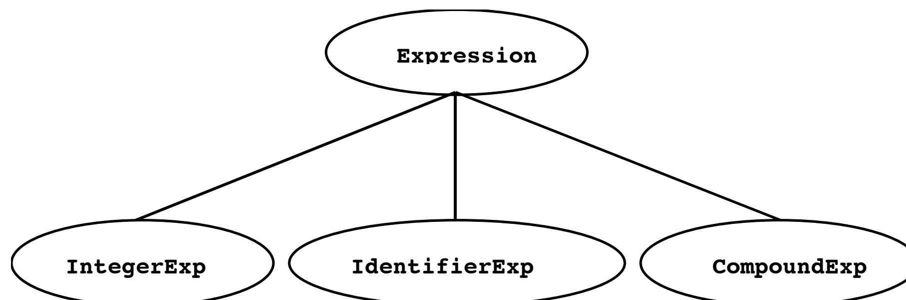
The relationship between the subclasses and the superclass goes beyond just the convenience of allowing the common implementation to be shared between the classes. The subclass has an **is-a** relationship with the superclass; that is, a **`SalariedEmployee`** is an **`Employee.`** This means that in any context where an **`Employee`** object is used, a **`SalariedEmployee`** can be substituted instead. Anything that a client can do with an

`Employee` object (i.e. using the features available in the public interface) can be equivalently done to a **`SalariedEmployee`** object. This powerful **subtyping** relationship makes it possible for client code to be written in terms of the generic **`Employee`** object type and any specialization required for a specific kind of employee is handled by the subclass implementation.

Defining an inheritance hierarchy for expressions

As noted earlier, there are three different types of expression nodes that can make up an expression tree. An integer expression requires different storage and is evaluated differently than an identifier or compound expression. Yet all three of these types of expressions need to be able to co-exist within an expression tree and need to behave similarly from an abstract perspective.

An inheritance hierarchy is an appropriate way to represent the different types of expression trees. At the top of the hierarchy will be the **`Expression`** class that specifies the features that will be common to all types of expressions. The **`Expression`** class has three subclasses, one for each specific type of expression.



Now you should consider what features must be exported by in the public interface of the **`Expression`** class. The code being used by the interpreter in expects that an expression is capable of producing a string representation of itself using **`toString`** and returning an integer result from the **`eval`** member function. The prototype for **`toString`** is simple enough, but **`eval`** requires a bit more careful study. An expression is evaluated in a particular context, which establishes the values for variables used within an expression. A compound assignment expression allows assigning a value to a variable and an identifier expression needs to be able to look up a variable and return its value. To do so, you need to have some mechanism through which you can associate a variable name with a value, just as a compiler would do. A compiler maintains a *symbol table* where an identifier name is mapped to its associated information. The **`map`** container from the STL provides just the right tool for implementing such a symbol table. This table is passed by reference to the **`eval`** member functions, so that values for variables referred to in the expression can be accessed or updated in the table as needed.

Another important issue for the **Expression** class interface is that the member functions of the superclass need to be declared using the C++ keyword **virtual**. The **virtual** keyword is applied to a member function to inform the compiler that this member function can be overridden by a subclass. The **virtual** keyword ensures that the member function is invoked using the dynamic run-time type of the object instead of relying on the static compile-time type. For example, consider this code fragment

```
Expression *exp;
// exp initialized to point to new object of an Expression subclass
cout << exp->eval(table);
```

In the above code, **exp** has a compile-time type of **Expression*** whereas the dynamic type might be **IntegerExp*** or **CompoundExp***. When invoking the **eval** member function, the compile-time type dictates that it should use the version of **eval** that is implemented in the **Expression** class. However, you want the overridden version of **eval** that is defined in the specific subclass to be used. By tagging the **eval** member function with the virtual keyword, you are indicating that the member function should be chosen based on the dynamic type of the object. This **dynamic dispatch** is typically desired for any class that is intended to be subclasses so you will typically mark every member function within such a class with the virtual keyword.

All classes that are subtypes of **Expression**—integers, identifiers, and compound expressions—are able to evaluate themselves and the superclass **Expression** declares the common prototype for the **eval** method. At the same time, it isn't possible to evaluate an expression unless you know what type of expression it is. You can evaluate an integer or an identifier easily enough, but you can't evaluate a generic expression without more information. Therefore the member function **eval** in the **Expression** class is marked as **pure virtual**, which means that there is no default implementation provided by the superclass, the implementation must be supplied by the subclass in an overridden version of the member function.

If you think about this problem, you'll soon realize that the **Expression** class is somewhat different from the other classes in this hierarchy. You can't have an **Expression** object that is not also a member of one of its subclasses. It never makes sense to construct an **Expression** object in its own right. Whenever you want to create an expression, you simply construct an object of the appropriate subclass. Classes, like **Expression**, that are never constructed are called **abstract classes.** In C++, you indicate that a class is abstract by including at least one pure virtual member function in the class interface.

Defining the interface for the Expression subclasses

In C++, the inheritance relationship for a class is declared in the class header like this

```
class IntegerExp : public Expression  {
```

The above class header declares the new class **`IntegerExp`** to be a public subclass of the **`Expression`** class. Being a public subclass means that all of the public features of the **`Expression`** class are inherited and public in the **`IntegerExp`** class. This establishes the subtyping relationship that an **`IntegerExp`** is an **`Expression`** and perhaps more, which means an **`IntegerExp`** object can be substituted wherever an **`Expression`** object is expected.

Each concrete Expression subclass must provide the implementation for the two pure virtual member functions declared in the superclass: **`toString`** and **`eval.`** Each expression subclass, whether it be an integer constant, an identifier, or a compound expression, will have its own specific way of implementing these member functions, but must provide that functionality using the exact prototype specified by the superclass.

Each subclass also declares its own constructor that depends on the expression type. To construct an integer expression, for example, you need to know the value of the integer constant. To construct a compound expression, you need to specify the operator along with the left and right sub-expressions.

The code below shows the interface for the **`Expression`** abstract superclass and its three subclasses. All **`Expression`** objects are immutable, in the sense that an **`Expression`** object, once created, will never change. Although clients are free to embed existing expressions in larger ones, the interface offers no facilities for changing the components of any existing expression. Using an immutable type to represent expressions helps enforce the separation between the implementation of the **`Expression`** class and its clients. Because those clients are prohibited from making changes in the underlying representation, they are unable to change the internal structure in a way that violates the requirements for expression trees.

```
/**
 * Class: Expression
 * -----------------
 * This class is used to represent the abstract notion of an
 * expression, such as one you might encounter in a C++ program.
 * The abstract Expression class has three concrete subclasses:
 *
 *  1. IntegerExp: an integer constant
 *  2. IdentifierExp: string representing an identifier
 *  3. CompoundExp: two expressions combined by an operator
 *
 * The Expression class defines the interface common to all Expression
 * objects and each subclass provides its own specific implementation
 * of the common interface.
 */

class Expression
{
  public:

   /**
    * Constructor: Expression
    * -----------------------
    * The base class constructor is merely a no-op.  The subclasses
    * should provide their own constructors.
    */

   Expression();

   /**
    * Destructor: ~Expression
    * Usage: delete exp;
    * ------------------
    * The destructor deallocates the storage for this expression.
    * It must be declared virtual to ensure that the correct subclass
    * destructor is called when deleting an expression.
    */

   virtual ~Expression();

   /**
    * Member function: eval
    * Usage: result = exp->eval(table);
    * ---------------------------------
    * This member function evaluates this expression and returns its
    * value.  It is declared virtual to ensure the appropriate subclass
    * version is used when evaluating an expression. The "= 0" after the
    * prototype indicates that this member function is "pure virtual"
    * and the code for the function must be supplied by the subclass.
    */
   virtual int eval(map<string, int>& varTable) = 0;
```

```
  /**
   * Member function: toString
   * Usage: str = exp->toString();
   * -----------------------------
   * This member function returns a string representation of this
   * expression. The "= 0" after the prototype indicates that this
   * member function is "pure virtual" and the code for the function
   * must be supplied by the subclass.
   */

  virtual string toString() = 0;

 private:
  // for simplicity, disallow deep copying of Expressions and subclasses
  Expression(const Expression& exp);
  void operator=(const Expression& rhs);
};

class IntegerExp: public Expression
{
 public:

  /**
   * Constructor: IntegerExp
   * Usage: Expression *exp = new IntegerExp(10);
   * --------------------------------------------
   * The constructor initializes a new integer constant expression
   * to the given value.
   */

  IntegerExp(int val);

  /**
   * Member function: eval
   * Usage: result = exp->eval(table);
   * ---------------------------------
   * This member function returns the value of the integer constant
   * represented by this expression.
   */

  virtual int eval(map<string, int>& varTable);

  /**
   * Member function: toString
   * Usage: str = exp->toString();
   * -----------------------------
   * This member function returns a string representation of the
   * the integer constant represented by this expression.
   */

  virtual string toString();

 private:
  int value;
};
```

```
class IdentifierExp : public Expression
{
  public:

   /**
    * Constructor: IdentifierExp
    * Usage: Expression *exp = new IdentifierExp("count");
    * ---------------------------------------------------
    * The constructor initializes a new identifier expression
    * for the variable named by name.
    */

   IdentifierExp(string name);

   /**
    * Member function: eval
    * Usage: result = exp->eval(table);
    * ---------------------------------
    * This member function returns the value of the identifier
    * represented by this expression by looking up the name
    * in the table and returning its assigned value. An error
    * is raised if the identifier is not found in the table.
    */

   virtual int eval(map<string, int>& varTable);

   /**
    * Member function: toString
    * Usage: str = exp->toString();
    * -----------------------------
    * This member function returns the identifier name represented
    * by this expression.
    */

   virtual string toString();

  private:
      string id;
};

class CompoundExp: public Expression
{
  public:

   /**
    * Constructor: CompoundExp
    * Usage: Expression *exp = new CompoundExp('+', e1, e2);
    * -----------------------------------------------------
    * The constructor initializes a new compound expression
    * which is composed of the operator (op) and the left and
    * right subexpression (lhs and rhs).
    */

   CompoundExp(char op, Expression *lhs, Expression *rhs);
```

```
      /**
       * Destructor: CompoundExp
       * Usage: delete exp;
       * ------------------
       * The destructor deallocates all storage associated with
       * this compound expression which includes recursively
       * deallocating its subexpressions.
       */

      virtual ~CompoundExp();

      /**
       * Member function: eval
       * Usage: result = exp->eval(table);
       * ---------------------------------
       * This member function returns the value of this expression
       * by recursively evaluating the left and right subexpressions
       * and joining the results using op.
       */

      virtual int eval(map<string, int>& varTable);

      /**
       * Member function: toString
       * Usage: str = exp->toString();
       * -----------------------------
       * This member function returns a string representation of this
       * compound expression.
       */

      virtual string toString();

     private:
      char op;
      Expression *lhs, *rhs;
};
```

As written, the **Expression** classes export constructors, string conversion, and evaluation functions. There are, however, other operations on expressions that you might at first think belong in this interface. For example, the main program for the interpreter calls the function **ParseExp**, which are in some sense part of the behavior of the expression type. This observation raises the question of whether the **Expression** interface should export those functions as well.

Although **ParseExp** must be defined somewhere in the code, exporting it through the **Expression** interface may not be the best design strategy. In a full-scale interpreter, the parser requires a significant amount of code—enough to warrant making this phase a complete module in its own right. In the stripped-down version of the interpreter presented in this chapter, the code is much smaller. Even so, it makes sense to partition the phases of the interpreter into separate modules for the following reasons:

1. *The resulting modular decomposition resembles more closely the structure you would tend to encounter in practice.* Full-scale interpreters are divided into separate modules;

following this convention even in our restricted example clarifies how the pieces fit together.

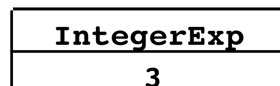
2. *ȟe program ȉll be easier to maintain as you add features.* Getting the module structure right early in the implementation of a large system makes it easier for that system to evolve smoothly over time. If you start with a single module and later discover that the program is growing too large, it usually takes more work to separate the modules than it would have earlier in the program evolution.
3. *šing separate module for the parser makes it easier to substitute neˇimplementations.* One of the principal advantages of using a modular design is that doing so makes it easier to substitute one implementation of an interface for another. For example, the section on "Parsing" later in this chapter defines two different implementations of the **`ParseExp`** function. If **`ParseExp`** is exported by the **`Expression`** interface, it is more difficult to substitute a new implementation than it would be if **`ParseExp`** were exported from a separate module.

For these reasons, the **`Expression`** interface exports only the types needed to represent expressions, along with the constructor and evaluation functions. The **`ParseExp`** function is exported by a separate module.

Implementing the Expression classes

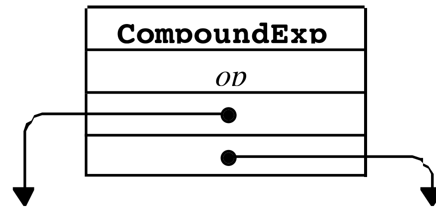
The abstract **`Expression`** superclass declares no data members. This is appropriate, as there is no data that is common to all expression types. Each specific subclass has its own unique storage requirements—an integer expression needs to store an integer constant, a compound expression stores pointers to its sub-expressions, and so on. Each subclass declares those specific data members that are required for its particular expression type.

To reinforce your understanding of how **`Expression`** objects are stored, you can visualize how the concrete expression structure is represented inside the computer's memory. The representation of an **`Expression`** object depends on its specific subclass. You can diagram the structure of an expression tree by considering the three classes independently. An **`IntegerExp`** object simply stores an integer value, shown here as it would exist for the integer 3:



An **`IdentifierExp`** object stores a string representing a variable name, as illustrated here for the variable **`x`**:

In the case of a **`CompoundExp`** object, it stores the binary operator along with two pointers which indicate the left and right sub-expressions:

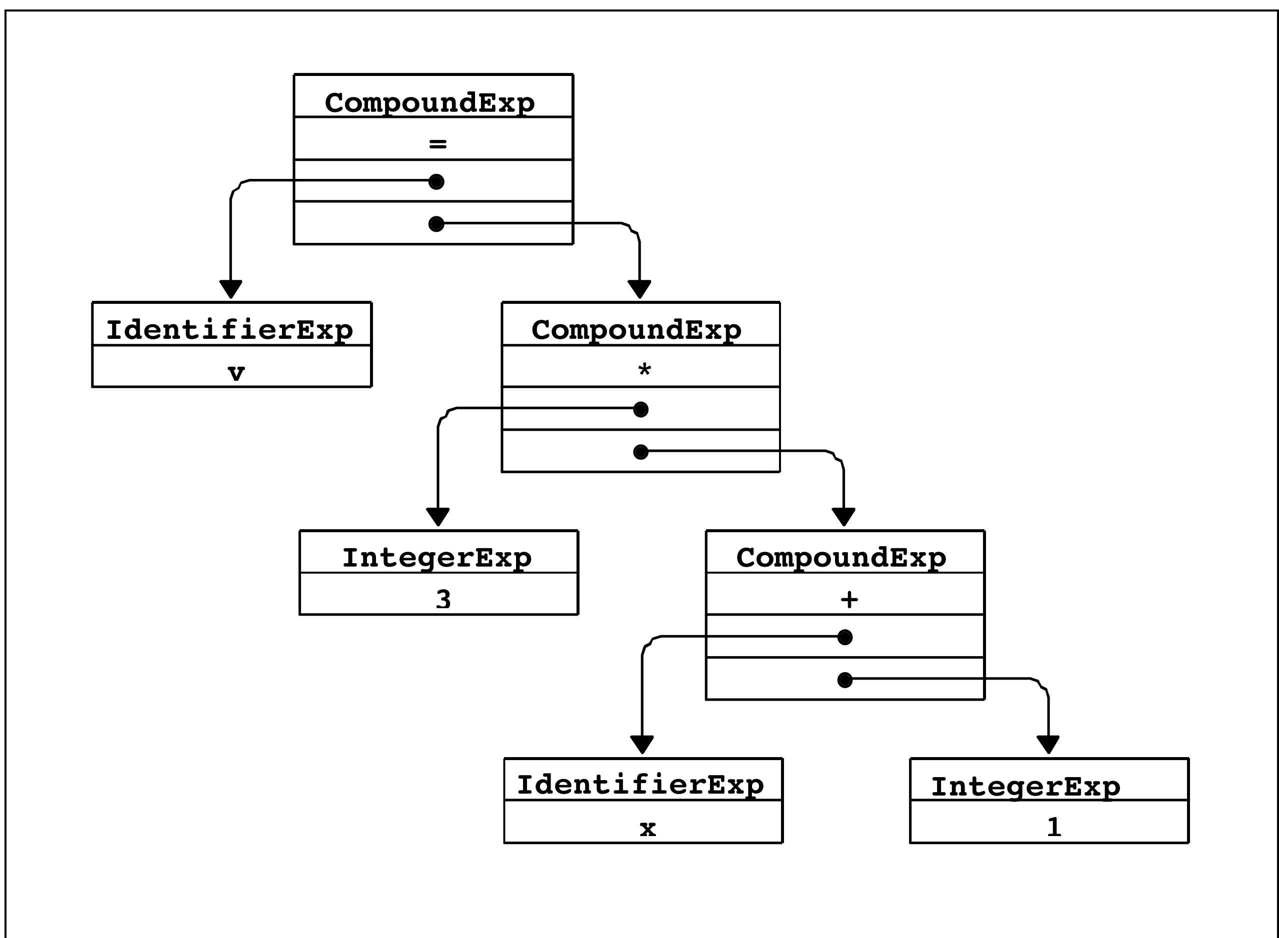


Because compound nodes contain sub-expressions that can themselves be compound nodes, expression trees can grow to an arbitrary level of complexity. The diagram below illustrates the internal data structure for the expression

```
y = 3 * (x + 1)
```

which includes three operators and therefore requires three compound nodes. Although the parentheses do not appear explicitly in the expression tree, its structure correctly reflects the desired order of operations.

Representation of the expression `y = 3 * (x + 1)`



Implementing the member functions

The member functions in the expression classes are quite easy to implement. Each subclass provides a constructor that takes in appropriate arguments and initializes the data members. The implementation of the **toString** member function uses the information from the data members to return a string representation of the expression.

The only remaining task to implement the evaluation member function. Each subclass has its own strategy for evaluating an expression.

Integer expressions are the easiest. The value of an expression of an integer expression is simply the value of the integer stored in that node. Thus, the **IntegerExp::eval** member function looks like

```
int IntegerExp::eval(map<string, int>& varTable)
{
   return value;
}
```

Note that even though an **IntegerExp** does not use the parameter **varTable**, it is required in the prototype for **eval** member function so that it exactly matches the prototype given in the **Expression** superclass.

The next to consider is that of identifiers. To evaluate an identifier expression, you look up the variable in the variable table and return the associated value as shown here:

```
int IdentifierExp::eval(map<string, int>& varTable)
{
   map<string, int>::const_iterator found = varTable.find(id);
   if (found == varTable.end()) {
      cerr << id << " is undefined." << endl;
      exit(1);
   }

   return found->second;
}
```

The last case you need to consider is that of compound expressions. A compound expression consists of an operator and two sub-expressions, but you must differentiate two subcases: the arithmetic operators (**+**, **–**, *****, and **/**) and the assignment operator (**=**).

For the arithmetic operators, all you have to do is evaluate the left and right sub-expressions recursively and then apply the appropriate operation. For assignment, you need to evaluate the right-hand side and then store that value into the variable table for the identifier on the left-hand side.

The implementation of the full **Expression** hierarchy is presented here:

```
/**
 * File: exp.cpp
 * -------------
 * This file implements the Expression class hierarchy.  The public
 * member functions are standard constructor and eval functions
 * that require no individual documentation.
 */

Expression::Expression() {}
Expression::~Expression() {}

IntegerExp::IntegerExp(int val)
{
   value = val;
}

string IntegerExp::toString()
{
   ostringstream oss;
   oss << value;
   return oss.str();
}

int IntegerExp::eval(map<string, int>& varTable)
{
   return value;
}

IdentifierExp::IdentifierExp(string name) : id(name) {}

string IdentifierExp::toString()
{
   return id;
}

int IdentifierExp::eval(map<string, int>& varTable)
{
   map<string, int>::const_iterator found = varTable.find(id);
   if (found == varTable.end()) {
      cerr << id << " is undefined." << endl;
      exit(1);
   }
   return found->second;
}

CompoundExp::CompoundExp(char op, Expression *l, Expression *r)
{
   op = oper;
   lhs = l;
   rhs = r;
}

CompoundExp::~CompoundExp()
{
   delete lhs;
   delete rhs;
}
```

```
string CompoundExp::toString()
{
   return '(' + lhs->toString() + ' ' + op + ' '
              + rhs->toString() + ')';
}

int CompoundExp::eval(Map<int> &varTable)
{
   if (op == '=')
   {
      int val = rhs->eval(varTable);
      varTable[lhs->toString()] = val;
      return val;
   }

   int left = lhs->eval(varTable);
   int right = rhs->eval(varTable);
   switch (op) {
      case '+': return left + right;
      case '-': return left - right;
      case '*': return left * right;
      case '/': return left / right;
   }

   cerr << "Illegal operator: '" << op << "'" << endl;
   exit(1);
   return 0;   // never gets here, but compiler may not be able to tell
}
```

Parsing an expression

The problem of building the appropriate parse tree from a stream of tokens is not an easy one. To a large extent, the underlying theory necessary to build a efficient parser lies beyond the scope of this text. Even so, it is possible to make some headway on the problem and solve it for the limited case of arithmetic expressions.

Parsing and grammars

In the early days of programming languages, programmers implemented the parsing phase of a compiler without thinking very hard about the nature of the process. As a result, early parsing programs were difficult to write and even harder to debug. In the 1960s, however, computer scientists studied the problem of parsing from a more theoretical perspective, which simplified it greatly. Today, a computer scientist who has taken a course on compilers can write a parser for a programming language with very little work. In fact, most parsers can be generated automatically from a simple specification of the language for which they are intended. In the field of computer science, parsing is one of the areas in which it is easiest to see the profound impact of theory on practice. Without the theoretical work necessary to simplify the problem, programming languages would have made far less headway than they have.

The essential theoretical insight necessary to simplify parsing is actually borrowed from linguistics. Like human languages, programming languages have rules of syntax that define the grammatical structure of the language. Moreover, because programming

languages are much more regular in structure than human languages, it is usually easy to describe the syntactic structure of a programming language in a precise form called a **grammar.** In the context of a programming language, a grammar consists of a set of rules that show how a particular language construct can be derived from simpler ones.

If you start with the English rules for expression formation, it is not hard to write down a grammar for the simple expressions used in this chapter. Partly because it simplifies things a little in the parser, it helps to incorporate the notion of a term into the parser as any single unit that can appear as an operand to a larger expression. For example, constants and variables are clearly terms. Moreover, an expression in parentheses acts as a single unit and can therefore also be regarded as a term. Thus, a term is one of the following possibilities:

- An integer constant
- A variable
- An expression in parentheses

An expression is then either of the following:

- A term
- Two expressions separated by an operator

This informal definition can be translated directly into the following grammar, presented in what programmers call **BNF,** which stands for Backus-Naur form after its inventors John Backus and Peter Naur:

E → T
E → E *op* E

T → *integer*
T → *identifier*
T → (E)

In the grammar, uppercase letters like E and T are called **nonterminal symbols** and stand for an abstract linguistic class, such as an expression or a term. The specific punctuation marks and the italicized words represent the **terminal symbols,** which are those that appear in the token stream. Explicit terminal symbols, such as the parentheses in the last rule, must appear in the input exactly as written. The italicized words represent placeholders for tokens that fit their general description. Thus, the notation *integer* stands for any string of digits returned by the scanner as a token. Each terminal corresponds to exactly one token in the scanner stream. Nonterminals typically correspond to an entire sequence of tokens.

Like the informal rules for defining expressions presented in the section on "A recursive definition of expressions" earlier in the chapter, grammars can be used to generate parse trees. Just like those rules, this grammar is ambiguous as written and can generate several different parse trees for the same sequence of tokens. Once again, the problem is that the grammar does not incorporate any knowledge of the precedence of the operators and is therefore not immediately useful in constructing a parser.

Parsing without precedence

Before considering how it might be possible to add precedence to the grammar, it helps to think about circumventing this problem in a simpler way. What if there were no precedence in the language? Would that make parsing easier? Throwing away precedence is not as crazy an idea as it might seem. In the 1960s, Ken Iverson designed a language called APL (which is an abbreviation for *Programming Language*), which is still in use today. Instead of using standard mathematical rules of precedence, APL operators all have equal precedence and are executed in strictly right-to-left order. Thus, the expression

```
2 * x + y
```

is interpreted in APL as if it had been written

```
2 * (x + y)
```

which is exactly the opposite of the conventional mathematical interpretation. To recover the conventional meaning, you would have to write

```
(2 * x) + y
```

in APL. This style of precedence is called **Iversonian precedence** after its inventor.

The problem of parsing turns out to be much easier for languages that use Iversonian precedence, mostly because, in them, the grammar for expressions can be written in a form that is both unambiguous and simple to parse:

E → T
E → T *op* E

T → *integer*
T → *identifier*
T → (E)

This grammar is almost the same as the ambiguous grammar presented in the preceding section. The only difference is the rule

E → T *op* E

which specifies that the left-hand operand to any operator must be a simple term.

Writing a parser based on the Iversonian expression grammar requires little more than a direct translation of the grammar into code. For each of the nonterminal symbols, you write a function that follows the structure of the grammar. For example, the task of reading an expression is assigned to a function called **ReadE**, whose structure follows the rules for expressions. To parse either of the two expression forms, the **ReadE** function must first call the function **ReadT** to read a term and then check to see whether the next token is an operator. If it is, **ReadE** calls itself recursively to read the expression following the operator and creates a compound expression node from the parts. If the token is not an operator, **ReadE** calls **saveToken** to put that token back in the input being scanned where it will be read at a higher level of the recursive structure. In much the same way, the **ReadT** function implements the rules in the grammar that define a term. The code for **ReadT** begins by reading a token and determining whether it represents an integer, an identifier, or a parenthesized expression. If it does, **ReadT** returns the corresponding expression. If the token does not correspond to any of these possibilities, the expression is illegal.

Parsers that are structured as a collection of functions that call themselves recursively in a fashion guided by a grammar are called **recursive-descent parsers.** A complete implementation of a recursive-descent parser for expressions with Iversonian precedence appears below. The real work is done by the mutually recursive functions **ReadE** and **ReadT**. The **ParseExp** function itself simply calls **ReadE** to read the expression and then checks to see that there are no extra tokens on the input line.

```
/**
 * Implementation notes: ParseExp
 * ------------------------------
 * This function calls ReadE to read an expression and then
 * checks to make sure no tokens are left over.
 */

Expression *ParseExp(streamtokenizer& st)
{
   if (!st.hasMoreToken()) return NULL;  // end-of-stream reached
   Expression *exp = ReadE(st);
   if (st.hasMoreTokens()) {
      cerr << "Extra tokens encountered while parsing an expression." << endl;
      cerr << "Exiting gracefully" << endl;
      return NULL;
   }

   return (exp);
}

/**
 * Implementation notes: ReadE
 * Usage: exp = ReadE(scanner);
 * ---------------------------
 * This function reads the next expression from the streamtokenizer by
 * matching the input to one of the following grammatical rules:
```

```
 *
 *       E  ->  T
 *       E  ->  T op E
 *
 * Both right-hand sides start with T, so the code can begin by
 * calling ReadT.  If the next token is an operator, the code
 * creates a compound expression from the term, the operator,
 * and the expression after the operator.
 */

Expression *ReadE(streamtokenizer& st)
{
   Expression *exp = ReadT(st);
   string token = st.nextToken();

   if (IsOperator(token)) {
      Expression *lhs = exp; // given better name to exp
      Expression *rhs = ReadE(st);
      exp =  new CompoundExp(token[0], lhs, rhs);
   } else {
      st.saveToken(token);
   }

   return exp;
}

/**
 * Function: ReadT
 * Usage: exp = ReadT(scanner);
 * ----------------------------
 * This function reads a single term from the streamtokenizer by matching
 * the input to one of the following grammatical rules:
 *
 *       T  ->  integer
 *       T  ->  identifier
 *       T  ->  ( E )
 *
 * In each case, the first token identifies the appropriate rule.
 */

Expression *ReadT(streamtokenizer& st)
{
   Expression *exp;

   string token = st.nextToken();
   if (isdigit(token[0])) {
      istringstream iss(token);
      int value;
      iss >> value;
      exp = new IntegerExp(value);
   } else if (isalpha(token[0])) {
      exp = new IdentifierExp(token);
   } else if (token == "(") {
      exp = ReadE(st);
      if (st.nextToken() != ")") {
         cerr << "Unbalanced parenthesis encountered." << endl;
         exit(1);
      }
   } else {
```

```
      cerr << "Illegal term in expression." << endl;
      exit(1);
   }

   return exp;
}

/**
 * Function: IsOperator
 * Usage: if (IsOperator(token)) . . .
 * -----------------------------------
 * This function returns true if the token is a legal operator.
 */

bool IsOperator(string token)
{
   if (token.length() != 1) return false;
   switch (token[0]) {
      case '+': case '-': case '*': case '/': case '=':
         return true;
      default:
         return false;
   }
}
```

If all of this seems like magic, you should go through each step in the operation of the parser on a simple expression of your own choosing. As with many recursive functions, the code for the parser is simple even though the effect is profound.