TAWK: a Simple Interpreter in C++

The data-encapsulation features of C++ let you create more sophisticated programs without adding complexity. Bruce's "tiny" AWK implementation illustrates this and more.

By Bruce Eckel, Dr. Dobb's Journal
May 01, 1989
URL:http://www.ddj.com/architect/184408136

Bruce Eckel is a C++ consultant and owner of Eisys Consulting. He has been writing for Micro Cornucopia for two and a half years. This article is adapted from his book Using C++ (Osborne/ McGraw-Hill, 1989). Bruce may be contacted at Eisys Consulting, 501 N. 36th St., Ste. 163, Seattle, WA 98103.

Most microcomputer database management systems (DBMSs) read and write records in a "comma-separated ASCII" format. This is probably an artifact from the days when Basic (which uses that format) was the only common tongue on microcomputers. Comma-separated ASCII files are useful not only because they allow the records from one DBMS to be moved to another, but also because they can be manipulated by using programming languages.

While Basic automatically reads and writes these records, other languages must be programmed to do so. In C++, this tedious task can be encapsulated into several classes; the user of the class doesn't need to worry about the details. In the first part of this article, two classes are created. The first, class field, reads a single quoted and comma-separated field and makes an object from that field. The second, class csascii, opens a comma-separated ASCII file and reads records (as arrays of field objects) one at a time, until the file ends. A simple application that uses the classes to search through a data-base file for a last name is presented.

Database files must often be manipulated or output in an organized way as a "report." It becomes tedious and problematic to write and compile code for each different report since nonprogrammers must often design reports. A common solution to a problem such as this is the creation of a "scripting language" specifically tailored to the task at hand. The second part of this article is the creation of a simple language that outputs the records (to standard output) in a comma-separated ASCII file according to a script in a separate file.

The program is called TAWK for "tiny awk," since the problem it solves is vaguely reminiscent of the "awk" pattern-matching language found on Unix (versions have also been created for DOS). It demonstrates one of the thornier problems in computer science: parsing and executing a programming language. The data-encapsulation features of C++ prove most useful here, and a recursive-descent technique is used to read arbitrarily long fields and records.

The code was developed and tested on a DOS system. It compiles with Zortech C++ or the Glockenspiel/Advantage translator used with Microsoft C. The programs should also work on Unix, because all

library calls are ANSI C and the only class used that is not defined here is the streams class (which is included with every C++ package). The simple screen-manipulation commands (clear screen, reverse video) assume an ANSI terminal or a PC with ANSI.SYS loaded.

Object-Oriented Terminology

When discussing object-oriented programming, it is helpful to review some of the terminology. Object-oriented programming means "sending messages to objects." In traditional languages, data is declared and functions act directly on the data. In object-oriented programming, objects are created, messages are sent to the objects, and the objects decide what to do with the message (in other words, how to act on their internal data).

An object is an entity with internal state (data) and external operations, called member functions in C++. "Sending a message" means calling one of these member functions.

An important reason for organizing a program into distinct objects is the concept of encapsulation. When data is encapsulated in an object, it is hidden away (private) and can only be accessed by way of a clearly defined interface. Only class member functions and friend functions may modify private data. Data encapsulation can clarify code by combining data in a single package with specific legal operations (member functions). Data encapsulation is also useful in preventing bugs -- a working class doesn't break simply because it is used in a new program.

Virtual Functions

Object-oriented purists will notice this program does not use late binding (by way of C++ virtual functions) and thus is not object-oriented in the Smalltalk sense. When a message is sent to an object in Smalltalk, the object always decides what to do with the message (that is, the specific function to call) at run time, so the function address isn't bound to the function call until the call is actually made. Because most compilers bind function calls during compilation, run-time binding is often called late binding.

C++ always performs binding at compile time, unless the programmer specifically instructs the compiler to wait until run time by using the virtual keyword. This feature allows subclasses (all inherited from the same base class) to have identical function calls that are executed differently. A collection of generic objects (all of the same base class) can be maintained and all the "legal" messages for the base class may be sent to any of the objects in that collection. The object figures out what to do with the message according to what specific subclass it is. This is object-oriented programming in its true sense.

Most problems can benefit from the data-encapsulation features of C++. It seems, however, that not every problem demands virtual functions. The project presented here is one of those cases. For an example of the use of virtual functions, see my article "Building MicroCAD" in the November/December 1988 issue of Micro Cornucopia (also available as part of C++ source code library disk #1, available from Eisys Consulting for $15).

Reading C++ Code

If you are a C programmer, here's a simple way to think about C++ while you are reading the code for this article: Objects look like structures, with a few bells and whistles. One bell is that you can hide some of the structure members--members are automatically hidden (private) unless you explicitly state they are public. A whistle is the ability to include functions as members of the struct. Members of a class (a user-defined type) are accessed just as you would access members of a struct--with a dot (or an arrow, if you have a pointer to an object). One more whistle is that the programmer can define the way the objects are initialized (by using the constructor) when they come into scope, and cleaned up (by using the destructor) when they go out of scope.

Example 1 shows a tiny class to introduce you to the basics of C++ programming. class declarations are generally contained in header files with the extension .hxx. Definitions are generally contained in files with the extension .cxx. The AT&T Unix C++ chose the unfortunate extension of .C for definitions, and .h for declarations. This is fine on Unix, which is case-sensitive, but causes problems while in DOS. Walter Bright's Zortech C++ compiler originally used .cpp. He later modified it to allow .cxx, which is the style the Glockenspiel translator (previously marketed by Lifeboat as Advantage C++) uses. I use the .cxx format because it works with both products.

Example 1: A C++ class

```
class tiny {
  // private stuff here (this is a comment)
  int i;
public: // public stuff here:
 print () { // an "in-line" function
   printf ("i = %d\n", i);
 }
 tiny (int j); // constructors have the class name
 tiny() {} // destructors use a tilde
}; // classes end with a brace and a semicolon

tiny::tiny (int j) { // non inline definition
  i = j;
}

main() {
  tiny A(2); // implicit constructor call
  // A.i = 30; // error! private member
  A.print (); // calling a member function
  // implicit destructor call at end of scope
  }
```

The Streams Class

The streams class used here is an extremely useful class developed by Bjarne Stroustrup (the inventor of the language) to handle input/output. It defaults to standard input and standard output (the cin and cout objects, automatically defined when you include the stream.hxx header file), but can also be used to read and write files. A buffer can even be made into a stream object, and the same operations can be performed on that object. The most complete written reference available for the streams class is chapter 8 of Stroustrup's The C++ Programming Language (Addison-Wesley, 1986). This chapter is not exactly an exhaustive example of streams. One of the beauties of C++ is that you always have access to a description (often admittedly terse) of the operations available for that particular class -- the header file. By studying the header file, you can often get ideas for new ways to use an object. Zortech C++ also has library source code available, which includes valuable comments on the use of certain functions (that's how I figured out many features). Output in streams is accomplished with the operator you know from C as left shift. C++ allows you to overload functions and operators to give them different meanings depending on their arguments. When left shift is used with a stream object, it means "put this stuff out to the stream." Example 2 lists a short program to show the use of streams. Notice how streams allow you to string together a series of output statements.

Example 2: The use of streams

```
# include <stream.hxx> // cout automatically defined
main() {
  cout << "Hello, world!\n" << "I am"
       << 6 << "today!\n";
```

Recursive Descent

A recursive descent algorithm is useful if you don't know how long or complicated a statement will be when you start looking at it. In programming languages, for example, recursive descent parsers are often used in expression evaluation, because expressions can contain other expressions. In this project, the expressions aren't particularly complicated, but we don't know how long a string of text is going to be. A central function is used when scanning an expression using recursive descent. This function munches along and absorbs input until it runs into a delimiter that indicates a change in the type of input (white space, for example, or a number). At this point, it might call another function to eat the white space or to get a string of digits and turn it into a number. Then, if the expression is finished, it will just return. If the expression isn't finished (and here's the tricky part), it calls itself (that is, it recurses). Every time it encounters a new expression within the one it's evaluating, it recurses to evaluate the expression. When solving more complex problems (such as a programming language), a set of functions is used. Each function may call any of the others during expression evaluation. At some point, the evaluation must bottom out. When this happens, the function performs some termination activities and then returns. As the stack unwinds from all the recursive calls, the tail end of each function call performs some operation to store the information it was able to glean, and then it returns. When the function finally completes, the expression has been evaluated. Recursive descent is used in three places in this project. The field class, which creates an object containing a single quote-delimited field, has a recursive function field::getfield() (shown in Listing Two) to read one character at a time, keeping track of the number of characters encountered, until the end of the field. When the closing quotation mark is encountered, memory is allocated for exactly the right number of characters and the function returns. As it unwinds, characters are placed in the object's data buffer. Using recursive descent means no restrictions are imposed on the field size (other than stack space). The token class uses recursive descent in a more sophisticated way. When a token object is created by handing it an input stream (by way of the constructor function token::token(istream & input)), it reads the input stream until it has scanned a complete token. When the constructor completes, a new token has been created. A token is a group of symbols that represent a single concept. A C++ compiler uses a large number of tokens: { means begin a scope, for means start a for loop, foo means a variable. TAWK has a much smaller number of tokens. All tokens in TAWK are delimited by the "@" sign, which starts a new command. When "@" is encountered, it is pushed back onto the input stream (for use in the next token object) and the current token is completed. The central recursive-descent function for token is token::get_token(), shown in Listing Seven. The class parse_array builds an array of tokens by recursively calling parse_array::build_array() (shown in Listing Seven). This function makes a new token, then looks at the token to decide what to do next. The two programs (LOOKUP and TAWK) are built from several classes. Each of these classes will be examined.

The Class Field

The declaration of the field class is shown in Listing One and the definitions are in Listing Two. The field object doesn't control opening or closing files. It is simply handed an istream from which it takes its input. If it finds the end of input, it just makes an internal note (by setting its end_of_file flag) and returns. It's up to the caller to check for end-of-file with the function field::eof(). The operator<<() is overloaded so that a field object may be put to a stream output object. When this occurs, the data field is copied to the output. The field constructor field::field (istream & instream) initializes all the variables to zero and sets the member istream * input equal to instream. This allows field::getfield() to treat input as a global variable and to simply get the next character. The last thing the constructor does is call the recursive-descent function field::getfield(), which recurses until it reaches the end of the field. When the constructor finishes, the field is complete. The function field::getfield() reads a character from the input stream. If it isn't an end-of-file character, it checks for terminators, which include a comma if not enclosed by quotation marks (determined by a special flag infield) or a carriage return, which delimits the entire record. If no terminator is found, the function counts the current character and calls itself to get the next character. If a terminator is found, memory is allocated to hold the string (using the C++ dynamic-memory allocation keyword new) and the string terminator \ 0 is inserted. As the function returns from

calling itself, each character is inserted, from right to left, into the buffer. Memory is not always allocated for a field. The constructor for a field object sets the data pointer to zero. If memory is never allocated, the destructor will delete a null pointer, which is defined to have no effect.

The Class csascii

The csascii (for comma-separated-AS-CII) class is shown in Listings Three (the declaration) and Four (the definition). The constructor opens the input file, counts the number of fields in a record, and closes the file. It then creates an array of pointers to field objects, reopens the file and reads in the first record. Every time csascii::next() is called, a new record is read until the end of the file. The operator[]() is overloaded so the individual fields may be selected from each record. This function checks to ensure that the index is within bounds. The method of opening files should be examined here. The line istream infile(new filebuf>open (filename, input)); is a succinct way to create a buffer and open a file. The new filebuf creates a filebuf object (necessary to open a file as an istream) on the free store and returns a pointer to this object. The pointer is used to call a member function, filebuf::open(). The pointer is also handed to the constructor of istream to create an object called infile. This is a clever piece of code, and nice for quick programming -- I got it from the Glockenspiel/Advantage manual, so I suspect it's something John Carolan cooked up. Unfortunately, it isn't robust unless you know that the file exists. If the file doesn't exist on DOS machines, the system locks up. A more robust way to open the files in this program is to replace the previous code with the code in Example 3. Notice that in csascii::csascii(), the file is closed implicitly by putting braces around the first clause in the constructor where the fields are counted. When the istream object goes out of scope, the file is closed. This is the only purpose for putting the braces there. Anytime you want to control the destruction of a local variable, simply put it in braces.

Example 3: Opening files

```
"Ball", "Mike", "Oregon Software C++ Compiler"
"Bright", "Walter", "Zortech C++ Compiler"
"Carolan", "John", "Glockenspiel C++ Translator"
"Stroustrup", "Bjarne", "AT&T, C++ Creator"
"Tiemann", "Michael", "Free Software Foundation C++ Compiler"
```

Testing Field and csascii

Listing Five is a short program to show the use of class csascii. The csascii object file is created by giving it the name of the comma-separated ASCII file PPQUICK.ASC. (See Example 4 for a sample file.) Then the records are read one at a time and field 0 is compared to the first argument on the command line (presumably the last name of the persons in the database). When a record is found, it is displayed on the screen (notice the use of the ANSI screen-control codes). A flag called found is set to indicate the least one record is found. When no more matches occur, the program knows to exit (it is assumed the file has been sorted by the database manager).

Example 4: A sample comma-separated ASCII file PPQUICK.ASC

```
filebuf fl;
if (fl. open(argv [1], input) == 0) {
   cout << "cannot open" << argv[1] << "\n";
   exit (1);
}
istream infile (&fl);
```

The ANSI C library function strcmp() has been used here for compatibility. To ignore uppercase or lowercase in the comparisons, Microsoft C provides strcmpi() and Zortech provides strcmpl(). Notice how easy it is to use a class once it has been created. One of the advantages of C++ is the ease of use of library functions. (That is,

when library functions become available!)

TAWK

Table 1 provides the complete syntax for the TAWK language. You can see that each TAWK conmmand consists of an "@" sign and a single character (in the case of @() and @<>, the commands are @(and @< and the) and > are used by the function that reads the number, to find the end).

Table 1: The TAWK syntax

```
TAWK: A Tiny database processor, vaguely like AWK

usage: tawk tawkfile csafile
where: csafile contains comma-separated ASCII records.  Each field in a
       record is contained in quotes, and each record is delimited by a
       newline.  These are standard records that can be generated by the
       Basic language and most database management systems.

       tawkfile is a file that contains formatting commands.  Each
       record in the csafile is read and fields in the record are
       printed out according to the formatting commands in the
       tawkfile.  Everything in the tawkfile (characters, spaces,
       newlines) is printed literally except for the following:

       @(n)            Print field number n; @(3) prints field 3 of the
                       current record.  The first field in a record is
                       field 0.

       @<n>            Print an ascii character number n; @<27> prints
                       the escape character

       @!              This line is a comment until the end of the line

       @?nn@:          (then statements) @ (else statements) @.  An
                       if-then-else conditional.  If field nn is not
                       empty, the then statements are executed, otherwise
                       the else statements are executed.  A conditional
                       must have all three parts, but the statements may
                       be empty.  Conditionals can be nested.

       @Preamble or    When a tawkfile is begun, all statements until
       @P or @p        @main are considered to be part of the
                       preamble.  The preamble is only executed once, at
                       the beginning of the program.  The preamble must
                       be strictly text; it cannot contain field numbers
                       or conditionals.  The @preamble statement is
                       optional; @preamble is assumed until @main.

       @main           The main section is executed once for each record
       or @M or @m     in the  file.  All statements between
                       @main and @conclusion are part of the main.
                       section.  @main may contain field numbers and
                       conditionals.  The @main statement is required.

       @conclusion     The conclusion is executed after the last record
       or @C or @c     in the database file is read and the file is
                       closed. The conclusion,
                       like the preamble, may only contain text.  All
                       other characters on the same line as @preamble,
                       @main, or @conclusion are ignored.  The
                       @conclusion statement is required.
```

```
        @end            This must be at the end of the tawkfile

        @@              Print an @ sign

Example tawkfile:
@! A comment, which isn't printed
@! The @preamble is optional, but promotes understanding
@main

      This is field 1:@(1)
      This is field 10:@(10)
      @?4@:@(4) @Field 4 is empty @.
      print an escape: @<27>
      Re-generate comma-separated ASCII record:
```

The execution of a tawkscript parallels the compilation or interpretation of other programming languages. The tawkscript is parsed into arrays of tokens when the program starts up. An execution routine steps through the arrays and performs actions based on the tokens to run the tawkscript. Listing Six is the declaration for class token and class parse_array. Listing Seven contains the definitions. Listing Eight is the main() function for TAWK. In Listing Eight the tawkscript is parsed into three different parse_arrays, one each for the @preamble, @main, and @conclusion. These arrays are executed using the database file as input.

The Class token

Each token must be a particular type. The kind of information a token contains depends on what type it is. In TAWK, the possible token types are as follows: a field number (for printing out a field or testing if a field is empty in an if statement), a string (simple text including nonprintable characters), parts of a conditional statement (if, else, and endif), or a phase change (which indicates a transition from @preamble to @main or @main to @conclusion). Because a phase change is never executed but is simply used to terminate the creation of a parse_array, it isn't a token in the same sense, but some form of communication was necessary and this seemed the cleanest. The different types of tokens and phases are enumerated in the tokentype and phase declarations. The phase information is kept by the main program, but each token contains a tokentype identifier. Because a token can never be a field number and a string at the same time, the data container in a token is combined into an anonymous union (which is like a regular union only it has no name). The union is used to save space. A token also contains information about the level of the if statement. Because if statements can be nested, each token that is an if, else, or endif must have information about the nesting level. If the conditional evaluates to false (that is, the field is empty), the interpreter must hunt through tokens in the parse_array until it finds the else statement at the same level, and continue executing statements from there. While token::get_token() is performing its recursive-descent scanning, it calls several other functions, which are made private because they aren't needed by the user. token::get_next() gets a character and tests for end-of-file (which is an error condition, because an @end statement should always terminate the tawkfile). token::get_value() is used for the @() and @<> statements. token::dumpline() is called for comments. Listing Seven starts with a number of global variables that are declared static. This means they cannot be accessed outside the file (this use of the static keyword is called file static). When the constructor is called, it establishes the source of input characters (token-stream), sets the length of the string (which has been read so far) to zero, and begins the descent by calling token::get_token(). The following are three possibilities in token::get_token():

1. The next character in the input stream is an @ and the length is zero. This means you are at the beginning of a command and the next character will determine what the command is. In this case, a large switch statement is executed.

2. The next character is an @ and the length is not zero. This means you are in the middle of a string and a command is starting. In this case, the @ is pushed back on the input stream (for use by the next token), space is allocated for the string, and the unwinding of the stack is started with a return.

3. The next character is not an @. This means it must be plain text. In this case, token::get_token() calls itself to get more characters.

The Class parse_array

The class parse_array is a container class, because it is only used to contain objects of another class (token). There is no way to know how many tokens a parse_array will contain, so the recursive approach is used again. The constructor initializes some variables and calls the recursive function parse_array::build_array(), which keeps getting tokens and calling itself until a phase change or the end of the input (an @end statement). At this point, it allocates space to hold all the tokens (which it has been counting during the descent) and ascends, storing a token on each function return. The individual tokens in a parse_array can be selected by using brackets ([]) because the bracket operator has been overloaded in parse_array: :operator[](). Because token has a stream function defined, tokens can be put directly to cout.

Executing a TAWKscript

Listing Eight shows the main() function for TAWK. After the command-line arguments are checked, the tawkfile is opened and three parse_arrays are created: one for the @preamble, one for @main, and one for the @conclusion. The second command-line argument is used to create a csascii object. At the beginning and end of the script execution, the preamble and conclusion parse_arrays are simply sent to standard output (cout). Because they can only contain text, no other action is necessary. The central loop executes the statements in the @main phase for each record the csascii object reads from the database file. After a record is read, the type of each token in parse_array Amain is used in a switch statement to choose the proper action. Strings are sent to cout and fieldnumbers send the selected field to cout. If an if statement, if the selected field is empty in the current record, the parse_array index is incremented until the else token at the same level is found. If the field is not empty, no action is taken (the following statements are executed). When an else is encountered, it means the if evaluated to true, so the else clause is skipped over until the endif of the same level is found. Listing Nine is a make-file to make all the examples in this project.

Example TAWKscripts

Listings Ten and Eleven show examples of tawkscripts. Listing Ten reformats a file with six fields into one with five fields, combining the last two fields. If both of the last two fields are not empty, a space is inserted between them. Listing Eleven shows the usefulness of the preamble and conclusion. It creates a tiny telephone list (which I carry in my wallet) on an HP LaserJet printer. The preamble and conclusion are used to send special control codes to the printer. The use of nested if-then-else statements is shown here: If field 3 exists, it is printed followed by a carriage return and a test to see if field 4 exists, which is printed with a linefeed if it does (nothing happens if it isn't). If field 3 doesn't exist, field 4 is tested and printed with a linefeed (otherwise only a linefeed is printed). When everything is completed a reset is sent to the LaserJet. If you want a further challenge, try adding a goto system to TAWK. You will need to create a label command and a goto command. gotos can be executed from if-then-else statements.

Conclusion

The main() program for TAWK is actually quite small for what it does. Because the details are hidden in the csascii and parse_array objects, you can imagine creating a much more sophisticated program without losing control of the complexity. This is typical of C++. Indeed, it was designed to allow one programmer to handle the same amount of code that previously required several programmers. The compiler supports the creation of large projects by hiding initialization and cleanup, and by enforcing the correct use of user-defined types.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 (Calif. residents add sales tax) to Dr. Dobb's Journal, 501 Galveston Dr., Redwood City, CA 94063; or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

Products Mentioned

Zortech C++ -- Zortech Inc. 1165 Massachusetts Ave. Arlington, MA 02174 800-848-8408 $149.95 Glockenspiel C++ -- Glockenspiel (Available for DOS and OS/2) 2 Haven Ave. Port Washington, NY 11050 800-462-4374 $495 Includes Glockenspiel C++ for DOS, Glockenspiel C++ for OS/2, CommonView for MS Windows, CommonView for OS/2 PM, and source code for CommonView. Microsoft C 5.1 --Microsoft (For use with the Glockenspiel translator. Discounts available from other retailers) Box 97017 Redmond, WA 98073-9717 206-882-8080 $450 _TAWK, A Simple Interpreter in C++_ by Bruce Eckel **[LISTING ONE]**

```
// FIELD.HXX: used by csascii class to build a single field.
// Fields are collected by csascii to create a record.
// by Bruce Eckel,
#include <stream.hxx>

class field { // one field in a comma-separated ASCII record
  istream * input; // where to get the data
  char * data;
  int length, fsize;
  int end_of_file;  // flag to indicate the end of file happened
  void getfield();  // recursive function to read in a field;
            // treats data, length & input as globals
  int infield; // flag used by getfield() to determine whether
        // it's inside a quoted field
 public:
  field(istream & instream);
  ~field();
  friend ostream& operator<<(ostream &s, field & f) {
    s << f.data;
    return s;
  }
  int eof() { return end_of_file; }  // to check for end
  int size() { return fsize;}
  int last_length() {return length; }
  char * string() { return data; }
};
```

[LISTING TWO]

```
// FIELD.CXX: definitions for class field
// A "recursive descent" scanning scheme is used because field
// length is always unknown.
// by Bruce Eckel
#include "field.hxx"
```

```
field::field(istream & instream) {
  input = &instream;
  length = 0;
  end_of_file = 0; // set flag to say "we're not at the end"
  infield = 0; // set flag to say "we're not inside a field"
  data = (char *)0; // to show no memory has been allocated
  getfield();  // recursively get characters until end of field
}

field::~field() {
  delete data;  // if no memory has been allocated,
  // data = (char *)0 so this will have no effect.
}

// A Comma-separated ASCII field is contained in quotes to allow
// commas within the field; these quotes must be stripped out
void field::getfield() {
  char c;
  // This happens when DEscending:
  if((input->get(c)).eof() ) {
    end_of_file++;  // just say we reached the end...
    return;
  }
  else  // watch out for the Unix vs. DOS LF/CR problem here:
    if (((c != ',') || infield) && (c != '\n')) {
      if ( (c != '"') && (c != '\r')) // watch for quotes or CR
        length++;  // no quotes -- count this character
      else {
        if ( c == '"')
          infield = !infield;  // if we weren't inside a field
          // and a quote was encountered, we are now inside
          // a field.  If we were inside a field and a quote
          // was found, we're out of the field.
        c = 0; // a quote or CR; mark it so it isn't included
      }
      getfield();  // recursively get characters in field
      // after returning from function call, we jump past
      // the following "else" part to finish the recursion
    }
    else { // This happens once, when the terminator is found:
      fsize = length;  // remember how long the string is
      data = new char[length + 1]; // space for null terminator
      data[length] = '\0';  // highest index is "length"
        // when you allocate an array of length + 1
      length--;  // notice we don't insert the delimiter
      // Now the first "if" statement evaluates to TRUE and
      // the function rises back up.
      return;
    }
  // This happens when Ascending:
  if ( c ) // if it wasn't a quote or CR,
    data[length--] = c;  // put chars in as we rise back up...
}
```

[LISTING THREE]

```
// CSASCII.HXX: class to manipulate comma-separated ASCII
```

```
// database files.
//by Bruce Eckel
#include <stream.hxx>
#include "field.hxx"

class csascii {  // manipulates comma-separated ascii files,
// generated by most database management systems (generated and
// used by the BASIC programming language).  Each field
// is separated by a comma; records are separated by newlines.
  int fieldcount;
  field ** data; // an array to hold the entire record
  istream * datafile; // file with comma separated ASCII input
  int readrecord(); // private function to read a record
 public:
  csascii( char * filename );  // Open file, get first record
  ~csascii(); // destructor
  int next(); // get next record, return 0 when EOF
  field & operator[](int index); // select a field
  int number_of_fields() { return fieldcount; }
};
```

[LISTING FOUR]

```
// CSASCII.CXX: function definitions for comma-separated
// ascii database manipulation class
// by Bruce Eckel,
#include "csascii.hxx"

int csascii::readrecord() {
  for (int fieldnum = 0; fieldnum < fieldcount; fieldnum++ ) {
    data[fieldnum] = new field(*datafile);
    if (data[fieldnum]->eof()) return 0;
  }
  return 1;
}

csascii::csascii( char * filename ) {
  char c;
  fieldcount = 0;
  int quote = 0;
  // first, determine the number of fields in a record:
  {
    // See text for dangers of opening files this way:
    istream infile(new filebuf->open(filename, input));
    while(infile.get(c), c != '\n') {
      // keep track of being inside a quoted string:
      if (c == '"') quote = !quote;
      // fields are delimited by unquoted commas:
      if ( c == ',' && !quote)
        fieldcount++;
    }
  }  // infile goes out of scope; file closed
  fieldcount++; // last field terminated by newline, not comma
  // an array of field pointers:
  data = new field * [ fieldcount ];
  // re-open at start; dynamically allocate so it isn't scoped:
  datafile = new istream(new filebuf->open(filename, input));
  readrecord();
```

```
}

csascii::~csascii() {
  delete data;
  delete datafile; // calls istream destructor to close file
}

int csascii::next() {
  for (int i = 0; i < fieldcount; i++ )
    delete data[i];  // free all the data storage
  return readrecord(); // 0 when end of file
}

field & csascii::operator[](int index) {
  if (index >= fieldcount) {
    cerr << "index too large for number of fields in record\n";
    exit(1);
  }
  return *(data[index]);
}
```

[LISTING FIVE]

```
// LOOKUP.CXX: simple use of csascii to find name in a database
// by Bruce Eckel,
#include "csascii.hxx"
#include <string.h>

main(int argc, char ** argv) {
  if (argc < 2) {
    cerr << "usage: lookup lastname\n";
    exit(1);
  }
  // This puts the database file in the root directory:
  csascii file("\\ppquick.asc"); // create object & open file
  int found = 0;  // indicates one record was found
  do {
    if (strcmp(file[0].string(),argv[1]) == 0) {
      found++;  // found one.  File is sorted, so if we stop
      // finding them, quit instead of wasting time.
      cout << chr(27) << "[2J"; // ANSI clear screen
      for (int i = 0; i < file.number_of_fields(); i++)
        cout << file[i] << "\n";
      cout << chr(27) << "[7m" << "press any key" <<
        chr(27) << "[0m";
      if( getch() == 27) break;
    }  else if (found) exit(0);  // quit if that was the last
  } while (file.next());
}
```

[LISTING SIX]

```
// PARSE.HXX: class to parse a tawk script file.  Creates
// a structure which can be used at run-time to "execute"
// the tawk script.
// by Bruce Eckel,
#include <stream.hxx>

// types of tokens the scanner can find:
enum tokentype {
  fieldnumber, string, if_, else_, endif_, phase_change
};

// preamble and conclusion of the tawk script are only executed
// once, while main is executed once for every data record
enum phase { preamble, tmain, conclusion};

class token {
  tokentype ttype;
  union {  // an "anonymous union"
    int fieldnum;  // if type is a fieldnumber
    unsigned char * literal; // if type is a string
  };
  int if_level;  // if this is an if_, then_, or else_
  // private functions:
  void get_token();  // recursive descent scanner
  // Functions to help in scanning:
  void getnext(char & c); // used by get_token();
  unsigned char get_value(char delimiter, char * msg);
  void dumpline(); // for @! comments
  void error(char * msg = "", char * msg2 = "");
 public:
  token(istream & input);
  ~token();
  friend ostream & operator<<(ostream &s, token &t);
  int field_number() { return fieldnum; }
  int token_type() { return ttype; }
  int nesting_level() { return if_level;}
};

// The following is called a "container class," since its sole
// purpose is to hold a list of objects (tokens, in this case):
class parse_array {
  token ** tokenarray; // an array of token pointers
  istream * parse_stream;
  int token_count;
  int end; // the size of the array
  phase p_section; // of the program (preamble, etc.)
  void build_array(); // another recursive function
 public:
  parse_array(istream & input);
  ~parse_array();
  int size() { return end; } // how big is it?
  token & operator[](int index); // select a token
  phase section() { return p_section; }
};
```

[LISTING SEVEN]

```
// PARSE.CXX: class parse function definitions
// by Bruce Eckel,
#include "csascii.hxx"
#include "parse.hxx"
#include <ctype.h>
#include <stdlib.h>

// The following are "file static," which means no one outside
// this file can know about them.  This is the meaning when a
// global variable is declared "static."
static istream * tokenstream;
static int length; // to remember size of string
static int line_number = 1;  // line counting for errors
static int if_counter = 0; // monitors "if" statement nesting
static phase program_section = preamble;  // ... until @main
static int end_of_file = 0; // zero means not end of file

token::token(istream & input) {
  // initialize values and start the descent
  tokenstream = &input;
  length = 0;
  get_token();  // recursively get characters to end of token
}

token::~token() { // delete heap if any has been allocated:
  if (ttype == string)
    delete literal;
}

void token::error(char * msg, char * msg2) {
  cerr << "token error on line " << line_number << ": " <<
      msg << " " << msg2 << "\n";
  exit(1);
}

ostream & operator<<(ostream &s, token &t) {
  switch (t.ttype) {
    case string:
      s << (char *)t.literal;
      break;
    case fieldnumber: // only for testing
      s << " fieldnumber: " << t.fieldnum << "\n";
  }
  return s;
}

// Get a character from the tokenstream, checking for
// end-of-file and newlines
void token::getnext(char & c) {
  if(end_of_file)
    error("attempt to read after @end statement\n",
       "missing @conclusion ?");
  if((tokenstream->get(c)).eof() )
    error("@end statement missing");
  if (c == '\n')
    line_number++; // keep track of the line count
}

// See text for description of tokens
void token::get_token() {
  char c;
  // This happens when DEscending:
  getnext(c);
  if ( c == '@') {
```

```
if (length == 0) { // length 0 means start of token
  getnext(c);
  switch(c) {
    case '!': // comment line
      dumpline(); // dump the comment
      get_token(); // get a real token
      break;
    case 'p' : case 'P' : // preamble statement
      if ( program_section != preamble )
        error("only one preamble allowed");
      dumpline(); // just for looks, ignore it
      get_token(); // get a real token
      break;
    case 'm' : case 'M' : // start of main loop
      dumpline(); // toss rest of line
      program_section = tmain;
      ttype = phase_change;
      return; // very simple token
    case 'c' : case 'C' : // start conclusion
      dumpline();
      program_section = conclusion;
      ttype = phase_change;
      return; // very simple token
    case 'e' : case 'E': // end statement
      end_of_file++;  // set flag
      ttype = fieldnumber; // so destructor doesn't
                // delete free store for this token.
      if (if_counter)
        error("unclosed 'if' statement(s)");
      return;
    case '(' :
      if ( program_section == preamble ||
        program_section == conclusion )
       error("@() not allowed in preamble or conclusion");
      fieldnum = get_value(')',"@()");
      ttype = fieldnumber;
      // This is a complete token, so quit
      return;
    case '<' :
      c = get_value('>',"@<>");
      length++;
      get_token(); // get more...
      break;
    case '?' : // beginning of an "if" statement
      if ( program_section == preamble ||
        program_section == conclusion )
       error("@? not allowed in preamble or conclusion");
      fieldnum = get_value('@',"@?@");
      ttype = if_;
      getnext(c);  // just eat the colon
      if(c != ':')
        error("@? must be followed by @: (then)");
      if_level = ++if_counter;  // for nesting
      return;
    case '~' : // the "else" part of an "if" statement
      ttype = else_;
      if_level = if_counter;
      return;
    case '.' : // "endif" terminator of an "if" statement
      ttype = endif_;
      if_level = if_counter--;
      if(if_counter < 0)
        error("incorrect nesting of if-then-else clauses");
      return;
    case '@' : // two '@' in a row mean print an '@'
      length++;  // just leave '@' as the value of c
```

```
          get_token();
          break;
        default:
          error("'@' must be followed by:",
          "'(', '<', '?',':','~','.','p','m','c' or '@'");
      }
    } else { // an '@' in the middle of a string; terminate
      // the string.  Putback() is part of the stream class.
      // It is only safe to put one character back on the input
      tokenstream->putback(c); // to be used by the next token
      // allocate space, put the null in and return up the stack
      literal = new unsigned char[length + 1]; // space for '\0'
      literal[length--] = '\0'; // string delimiter
      ttype = string; // what kind of token this is
      return; // back up the stack
    }
  } else { // not an '@', must be plain text
    length++;
    get_token();
  }
  // This occurs on the "tail" of the recursion:
  literal[length--] = c;  // put chars in as we rise back up...
}

// This function is used by get_token when it encounters a @(
// or a @< to get a number until it finds "delimiter."
// If an error occurs, msg is used to notify the user what
// kind of statement it is.
unsigned char token::get_value(char delimiter, char * msg) {
  char c;
  char buf[5];
  int i = 0;
  while(getnext(c), c != delimiter) {
    if (!isdigit(c))
      error("must use only digits inside", msg);
    buf[i++] = c;
  }
  buf[i] = 0;
  return atoi(buf);
}

void token::dumpline() { // called when '@!' encountered
  char c;
  while(getnext(c), c != '\n')
    ; // just eat characters until newline
}

// Since there's no way to know how big a parse_array is
// going to be until the entire tawkfile has been tokenized,
// the recursive approach is again used:

parse_array::parse_array(istream & input) {
  parse_stream = &input;
  token_count = 0;
  p_section = program_section; // so we know at run-time
  build_array();
}

void parse_array::build_array() {
  token * tk = new token(*parse_stream);
  if( ! end_of_file && tk->token_type() != phase_change) {
    // normal token, not end of file or phase change:
    token_count++;
    // recursively get tokens until eof or phase change:
    build_array();
  } else { // end of file or phase change
```

```
    // only done once per object:
    // allocate memory and return up the stack
    tokenarray = new token * [end = token_count];
    if(token_count) token_count--; // only if non-zero
    return;
  }
  tokenarray[token_count--] = tk;  // performed on the "tail"
}


parse_array::~parse_array() {
  for (int i = 0; i < end; i++)
    delete tokenarray[i];
  delete tokenarray;
}

token & parse_array::operator[](int index) {
  if ( index >= end ) {
    cerr << "parse_array error: index " << index
      << " out of bounds\n";
    exit(1);
  }
  return *tokenarray[index];
}
```

[LISTING EIGHT]

```
// TAWK.CXX: parses a tawk script and reads an ascii file;
// generates results according to the tawk script.
// by Bruce Eckel,
#include "csascii.hxx"
#include "parse.hxx"

main (int argc, char * argv[]) {
  int screen = 0;  // flag set true if screen output desired
  if (argc < 3) {
    cerr << "usage: tawk tawkfile datafile\n" <<
        "trailing -s pages output to screen";
    exit(1);
  }
  if (argc == 4) {
    if (argv[3][0] != '-') {
      cerr << "must use '-' before trailing flag\n";
      exit(1);
    } else
    if (argv[3][1] != 's') {
      cerr << "'s' is only trailing flag allowed";
      exit(1);
    } else
      screen++; // set screen output flag true
  }
  istream tawkfile(new filebuf->open(argv[1], input));
  parse_array Apreamble(tawkfile);  // the @preamble
  parse_array Amain(tawkfile);  // the @main section
  parse_array Aconclusion(tawkfile); // the @conclusion
  csascii datafile(argv[2]); // make a comma-separated ASCII
                             // object from the second arg
  // ------ @preamble  ------
```

```
  for (int i = 0; i < Apreamble.size(); i++)
    cout << Apreamble[i]; // preamble can only contain strings
  if(screen) {
    // ANSI reverse video sequence:
    cout << chr(27) << "[7m" << "press any key" <<
      chr(27) << "[0m";
    getch();
  }
  // ------ The Central Loop (@main) -------
  do {  // for each record in the data file
    if(screen) cout << chr(27) << "[2J"; // ANSI clear screen
    for(int i = 0; i < Amain.size(); i++) {
      switch(Amain[i].token_type()) {
        case fieldnumber:
          cout << datafile[Amain[i].field_number()];
          break;
        case string:
          cout << Amain[i];
          break;
        case if_:
          int fn = Amain[i].field_number();
          if (datafile[fn].size() == 0) { // conditional false
            int level = Amain[i].nesting_level();
            // find the "else" statement on the same level:
            while ( !(Amain[i].token_type() == else_
                && Amain[i].nesting_level() == level))
                  i++;
          } // conditional true -- just continue
          break;
        case else_: // an "if" conditional was true so skip
          // all the statements in the "else" clause
          int level = Amain[i].nesting_level();
          // find the "endif" statement on the same level:
          while ( !(Amain[i].token_type() == endif_
              && Amain[i].nesting_level() == level))
                i++;
          break;
        case endif_: // after performing the "else" clause
          break; // ignore it; only used to find the end
          // of the conditional when "if" is true.
        default: // should never happen (caught in parsing)
          cerr << "unknown statement encountered at run-time\n";
          exit(1);
      }
    }
    if(screen) {
      cout << chr(27) << "[7m" <<
        "press a key (ESC quits)" << chr(27) << "[0m";
      if( getch() == 27) break;
    }
  } while (datafile.next()); // matches do { ...
  //  ------ @conclusion ------
  for ( i = 0; i < Aconclusion.size(); i++)
    cout << Aconclusion[i]; //conclusion contains only strings
}
```

[LISTING NINE]

```
# makefile for tawk.exe & lookup.exe
# Zortech C++:
CPP = ztc
# Glockenspiel C++ w/ MSC 4:
#CPP = ccxx !4

all: tawk.exe lookup.exe

tawk.exe : tawk.obj parse.obj csascii.obj field.obj
        $(CPP)  tawk.obj parse.obj csascii.obj field.obj

lookup.exe : lookup.cxx csascii.obj field.obj
        $(CPP) lookup.cxx csascii.obj field.obj

tawk.obj : tawk.cxx parse.hxx csascii.hxx field.hxx
        $(CPP)  -c tawk.cxx

parse.obj : parse.cxx parse.hxx
        $(CPP)  -c parse.cxx

csascii.obj : csascii.cxx csascii.hxx field.hxx
        $(CPP)  -c csascii.cxx

field.obj : field.cxx field.hxx
        $(CPP)  -c field.cxx
```

[LISTING TEN]

```
@! REFORM.TWK
@! A tawk script to reformat a comma-separated ASCII file
@! with 6 fields.  This creates a new CS-ASCII file with
@! fields 4 and 5 combined.
@main
"@(0)","@(1)","@(2)","@(3)","@(4)@?4@: @~@.@(5)"
@conclusion
@end
```

[LISTING ELEVEN]

```
@! WALLET.TWK
@! Tawkfile to create a tiny phone listing for a wallet
@! on a Hewlett-Packard Laserjet-compatible printer
@! From a comma-separated ASCII file generated by a DBMS
@preamble
@<27>&l5C@! approximately 10 lines per inch
@<27>(s16.66H@! small typeface, built into Laserjet
@main
@! last, first, (area code) phone1
```

```
@(0),@(1)(@(2))@?3@:@(3)
@ phone2, if it exists
@?4@:@(4)
@~@.@~@?4@:@(4)
@~
@.@.@conclusion
@<27>E @! Reset the Laserjet
@end
```

[EXAMPLE 1]

```
class tiny {
  // private stuff here (this is a comment)
  int i;
 public: // public stuff here:
  print() { // an "in-line" function
    printf("i = %d\n",i);
  }
  tiny(int j); // constructors have the class name
  ~tiny() {} // destructors use a tilde
};  // classes end with a brace and a semicolon

tiny::tiny(int j) { // non inline definition
  i = j;
}

main() {
  tiny A(2);  // implicit constructor call
  // A.i = 30; // error! private member
  A.print(); // calling a member function
  // implicit destructor call at end of scope
}
```

[EXAMPLE 2]

```
#include <stream.hxx> // cout automatically defined
main() {
  cout << "Hello, world!\n" << "I am "
       << 6 << "today!\n";
}
```

[EXAMPLE 3]

```
filebuf f1;
if (f1.open(argv[1],input) == 0) {
  cout << "cannot open " << argv[1] << "\n";
  exit(1);
}
istream infile(&f1);
```

[EXAMPLE 4]

```
"Ball","Mike","Oregon Software C++ Compiler"
"Bright","Walter","Zortech C++ Compiler"
"Carolan","John","Glockenspiel C++ Translator"
```

```
"Stroustrup","Bjarne","AT&T, C++ Creator"
"Tiemann","Michael","Free Software Foundation C++ Compiler"
```
