1 Interfaces

Before I build a wall I'd ask to know
What I was walling in or walling out,
And to whom I was like to give offense.
Something there is that doesn't love a wall.
That wants it down.

Robert Frost, *Mending Wall*

The essence (精髓) of design is to balance competing goals and constraints. Although there may be many tradeoffs when one is writing a small self-contained system, the ramifications (分叉) of particular choices remain within the system and affect only the individual programmer. But when code is to be used by others, decisions have wider repercussions (反响).

Among the issues to be worked out in a design are

- Interfaces: what services and access are provided? The interface is in effect a contract between supplier and customer. The desire is to provide services that are uniform and convenient, with enough functionality to be easy to use but not so much as to become unwieldy (笨拙).
- Information hiding: what information is visible and what is private? An interface must provide straightforward access to the components while hiding details of the implementation so they can be changed without affecting users.
- Resource management: who is responsible for managing memory and other limited resources? Here, the main problems are allocating and freeing storage, and managing shared copies of information.
- Error handling: who detects errors, who reports them, and how? When an error is detected, what recovery is attempted?

In Chapter **??** we looked at the individual pieces -- the data structures -- from which a system is built. In Chapter **??**, we looked at how to combine those into a small program. The topic now turns to the interfaces between components that might come from different sources. In this chapter we illustrate interface design by building a library of functions and data structures for a common task. Along the way, we will present some principles of design. Typically there are an enormous number of decisions to be made, but most are

made almost unconsciously. Without these principles, the result is often the sort of haphazard (无计划的) interfaces that frustrate and impede (妨碍) programmers every day.

1.1 Comma-Separated Values

Comma-separated values, or ***CSV***, is the term for a natural and widely used representation for tabular (表格式的) data. Each row of a table is a line of text; the fields on each line are separated by commas. The table at the end of the previous chapter might begin this way in CSV format:

,"250MHz","400MHz","Line of"
,"R10000","Pentium II","source code"
C,0.36 sec,0.30 sec,150
Java,4.9,9.2,105

This format is read and written by programs such as spreadsheets; not coincidentally (巧合), it also appears on web pages for services such as stock price quotations. A popular web page for stock quotes presents a display like this:

Symbol	Last Trade		Change		Volume
LU	2:19PM	86-114	+4-1/16	+4.94%	5,804,800
T	2:19PM	60-11/16	-1-3/16	-1.92%	2,468,000
MSFT	2:24PM	106-9/16	+1-3/8	+1.31%	11,474,900

Download spreadsheet Format

Retrieving numbers by interacting with a web browser is effective but time-consuming. It's a nuisance (讨厌的) to invoke a browser, wait, watch a barrage (弹幕) of advertisements, type a list of stocks, wait, wait, wait, then watch another barrage, all to get a few numbers. To process the Spreadsheet Format" link retrieves a file that contains much the same (几乎相同) information in lines of CSV data like these (edited to fit):

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625,
        83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
        62.375,62.625,60.4375,2468000
"MSFT",106.5625,"11/4/1998","2:24PM",+1.375,
        105.8125,107.3125,105.5625,11474900
```

Conspicuous (显著的) by its absence in this process is the principle of letting the machine do the work. Browsers let your computer access data on a remote server, but it would be more convenient to retrieve the data without forced interaction. Underneath (在.. 下面) all the button-pushing is a purely textual procedure -- the browser reads some HTML, you type some text, the browser sends that to a server and reads some HTML back. With the right tools and language, it's easy to retrieve the information automatically. Here's a program in the language Tcl to access the stock quote web site and retrieve CSV data in the format above, preceded (开始) by a few header lines:

```
# getquotes.tcl: stock prices for Lucent, AT&T, Microsoft
set so [socket quote.yahoo.com 80]  ;# connect to server
set q "/d/quotes.csv?s=LU+T+MSFT&f=sl1d1t1ohgv"
puts $so "GET $q HTTP/1.0\r\n\r\n"  ;# send request
flush $so
```

```
    puts [read $so]                         ;# read & print reply
```

The cryptic sequence `f=...` that follows the ticker symbols is an undocumented control string, analogous to the first argument of `printf`, that determines what values to retrieve. By experiment, we determined that `s` identifies the stock symbol, `l1` the last price, `c1` the change since yesterday, and so on. What's important isn't the details, which are subject to (可以) change anyway, but the possibility of automation: retrieving the desired information and converting it into the form we need without any human intervention. We can let the machine do the work.

It typically takes a fraction of a second to run `getquotes` , far less than interacting with a browser. Once we have the data, we will want to process it further, Data formats like CSV work best if there are convenient libraries for converting to and from the format, perhaps allied with (联合) some auxiliary processing such as numerical conversions. But we do not know of an existing public library to handle CSV, so we will write one ourselves.

In the next few sections, we will build three versions of a library to read CSV data and convert it into an internal representation. Along the way, we'll talk about issues that arise when designing software that must work with other software. For example, there does not appear to be a standard definition of CSV, so the implementation cannot be based on a precise specification, a common situation in the design of interfaces.

1.2 A Prototype Library

We are unlikely to get the design of a library or interface right on the first attempt. As Fred Brooks once wrote, "plan to throw one away; you will, anyhow." Brooks was writing about large systems but the idea is relevant for any substantial (有内容的) piece of software. It's not usually until you've built and used a version of the program that you understand the issues well enough to get the design right.

In this spirit, we will approach (接近) the construction of a library for CSV by building one to throw away, a prototype. Our first version will ignore many of the difficulties of a thoroughly engineered library, but will be complete enough to be useful and to let us gain some familiarity with the problem.

Our starting point is a function `csvgetline` that reads one line of CSV data from a file into a buffer, splits it into fields in an array, removes quotes, and returns the number of fields. Over the years, we have written similar code in almost every language we know, so it's a familiar task. Here is a prototype version in C; we've marked it as questionable because it is just a prototype:

```
?   char    buf[200];   /* input line buffer */
?   char    *field[20]; /* fields */
?
?   /* csvgetline: read and parse line, return field count */
?   /* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
?   int csvgetline(FILE *fin)
?   {
?       int     nfield;
?       char    *p, *q;
?
?       if (fgets(buf, sizeof(buf), fin) == NULL)
?           return -1;
?       nfield = 0;
?       for (q = buf; (p=strtok(q, ",\n\r")) != NULL; q = NULL)
```

```
?           field[nfield++] = unquote(p);
?       return nfield;
?   }
```

The comment at the top of the function includes an example of the input format that the program accepts; such comments are helpful for programs that parse messy (散乱的) input.

The CSV format is too complicated to be parsed easily by `scanf` so we use the C standard library function `strtok`. Each call of `strtok(p, s)` returns a pointer to the first token within `p` consisting of characters not in `s`; `strtok` terminates the token by overwriting the following character of the original string with a null byte. On the first call, `strtok`'s first argument is the string to scan; subsequent calls use `NULL` to indicate that scanning should resume where it left off (停止) in the previous call. This is a poor interface. Because `strtok` stores a variable in a secret place between calls, only one sequence of calls may be active at one time; unrelated interleaved (交错的) calls will interfere with each other.

Our function `unquote` removes the leading and trailing quotes that appear in the sample input above. It does not handle nested quotes, however, so although sufficient for a prototype, it's not general.

```
?   /* unquote: remove leading and trailing quote */
?   char *unquote(char *p)
?   {
?       if (p[0] == '"') {
?           if (p[strlen(p)-1] == '"')
?               p[strlen(p)-1] = '\0';
?           p++;
?       }
?       return p;
?   }
```

A simple test program helps verify that `csvgetline` works:

```
?   /* csvtest main: test csvgetline function */
?   int main(void)
?   {
?       int i, nf;
?
?       while ((nf = csvgetline(stdin)) != -1)
?           for (i = 0; i < nf; i++)
?               printf("field[%d] = '%s'\n", i, field[i]);;
?           return 0;
?   }
```

The `printf` encloses the fields in matching single quotes, which demarcate (区分) them and help to reveal bugs that handle white space incorrectly.

We can now run this on the output produced by `getquotes.tcl`:

```
% getquotes.tcl | csvtest
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7] = '85.0625'
```

```
field[8] = '2888600'
field[0] = 'T'
field[1] = '61.0625'
...
```

(We have edited out the HTTP header lines.)

Now we have a prototype that seems to work on data of the sort we showed above. But it might be prudent (谨慎的) to try it on something else as well, especially if we plan to let others use it. We found another web site that downloads stock quotes and obtained a file of similar information but in a different form: carriage returns (`\r`) rather than newlines to separate records, and no terminating carriage return at the end of the file. We've edited and formatted it to fit on the page:

```
"Ticker","Price","Change","Open","Prev Close","Day High",
    "Day Low","52 Week High","52 Week Low","Dividend",
    "Yield","Volume","Average Volume","P/E"
"LU",86.313,-0.188,86.000,86.500,86.438,85.063,108.50,
    36.18,0.16,0.1,2946700,9675000,N/A
"T",61,125,0.938,60.375,60.188,61.125,60.000,68.50,
    46.50,1.32,2.1,3061000,4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
    119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

With this input, our prototype failed miserably (非常不幸地).

We designed our prototype after examining one data source, and we tested it originally only on data from that same source. Thus we shouldn't be surprised when the first encounter with a different source reveals gross (恶劣的) failings. Long input lines, many fields, and unexpected or missing separators all cause trouble. This fragile prototype might serve for personal use or to demonstrate the feasibility of an approach, but no more than that. It's time to rethink the design before we try another implementation.

We made a large number of decisions, both implicit and explicit, in the prototype. Here are some of the choices that were made, not always in the best way for a general-purpose library. Each raises an issue that needs more careful attention.

- The prototype doesn't handle long input lines or lots of fields. It can give wrong answers or crash because it doesn't even check for overflows, let alone return sensible values in case of errors.
- The input is assumed to consist of lines terminated by newlines.
- Fields are separated by commas and surrounding quotes are removed. There is no provision (准备) for embedded quotes or commas. The input line is not preserved; it is overwritten by the process of creating fields.
- No data is saved from one input line to the next: if something is to be remembered, a copy must be made.
- Access to the fields is through a global variable, the `field` array, which is shared by `csvgetline` and functions that call it; there is no control over access to the `field` contents or the pointers. There is also no attempt to prevent access beyond the last field.
- The global variables make the design unsuitable for a multi-threaded environment or even for two sequences of interleaved calls.

- The caller must open and close files explicitly; `csvgetline` reads only from open files.
- Input and splitting are inextricably (无法避免地) linked: each call reads a line and splits it into fields, regardless of whether the application needs that service.
- The return value is the number of fields on the line; each line must be split to compute this value. There is also no way to distinguish errors from end of file.
- There is no way to change any of these properties without changing the code.

This long yet incomplete list illustrates some of the possible design tradeoffs. Each decision is woven through the code. That's fine for a quick job, like parsing one fixed format from a known source. But what if the format changes, or a comma appears within a quoted string, or the server produces a long line or a lot of fields?

It may seem easy to cope (应付), since the "library" is small and only a prototype anyway. Imagine, however, that after sitting on the shelf (被搁置的) for a few months or years the code becomes part of a larger program whose specification changes over time. How will `csvgetline` adapt? If that program is used by others, the quick choices made in the original design may spell (招致) trouble that surfaces years later. This scenario (情节) is representative of the history of many bad interfaces. It is a sad fact that a lot of quick and dirty code ends up in widely-used software, where it remains dirty and often not as quick as it should have been anyway.

1.3 A Library for Others

Using what we learned from the prototype, we now want to build a library worthy of general use. The most obvious requirement is that we must make `csvgetline` more robust so it will handle long lines or many fields; it must also be more careful in the parsing of fields.

To create an interface that others can use, we must consider the issues listed at the beginning of this chapter: interfaces, information hiding, resource management, and error handling. The interplay (互相作用) among these strongly affects the design. Our separation of these issues is a bit arbitrary, since they are interrelated.

Interface. We decided on three basic operations:

`char *csvgetline(FILE *)`: read a new CSV line
`char *csvfield(int n)`: return the n-th field of the current line
`int csvnfield(void)`: return the number of fields on the current

What function value should `csvgetline` return? It is desirable to return as much useful information as convenient, which suggests returning the number of fields, as in the prototype. But then the number of fields must be computed even if the fields aren't being used. Another possible value is the input line length, which is affected by whether the trailing newline is preserved. After several experiments, we decided that `csvgetline` will return a pointer to the original line of input, or `NULL` if end of file has been reached.

We will remove the newline at the end of the line returned by `csvgetline`, since it can easily be restored if necessary.

The definition of a field is complicated; we have tried to match what we observe empirically (经验的) in spreadsheets and other programs. A field is a sequence of zero or more characters. Fields are separated by

commas. Leading and trailing blanks are preserved. A field may be enclosed in double-quote characters, in which case it may contain commas. A quoted field may contain double-quote characters, which are represented by a doubled double-quote; the CSV field `"x""y"` defines the string `x"y`. Fields may be empty; a field specified as "" is empty, and identical to one specified by adjacent commas.

Fields are numbered from zero. What if the user asks for a non-existent field by calling `csvfield(-1)` or `csvfield(100000)`? We could return `""` (the empty string) because this can be printed and compared; programs that process variable numbers of fields would not have to take special precautions to deal with non-existent ones. But that choice provides no way to distinguish empty from non-existent. A second choice would be to print an error message or even abort; we will discuss shortly why this is not desirable. We decided to return `NULL`, the conventional value for a non-existent string in C.

Information hiding. The library will impose no limits on input line length or number of fields. To achieve this, either the caller must provide the memory or the callee (the library) must allocate it. The caller of the library function `fgets` passes in an array and a maximum size. If the line is longer than the buffer, it is broken into pieces. This behavior is unsatisfactory for the CSV interface, so our library will allocate memory as it discovers that more is needed.

Thus only `csvgetline` knows about memory management; nothing about the way that it organizes memory is accessible from outside. The best way to provide that isolation is through a function interface: `csvgetline` reads the next line, no matter how big, `csvfield(n)` returns a pointer to the bytes of the `n`-th field of the current line, and `csvnfield` returns the number of fields on the current line.

We will have to grow memory as longer lines or more fields arrive. Details of how that is done are hidden in the csv functions; no other part of the program knows how this works, for instance whether the library uses small arrays that grow, or very large arrays, or something completely different. Nor does the interface reveal when memory is freed.

If the user calls only `csvgetline`, there's no need to split into fields; lines can be split on demand. Whether field-splitting is eager (done right away when the line is read) or lazy (done only when a field or count is needed) or very lazy (only the requested field is split) is another implementation detail hidden from the user.

Resource management. We must decide who is responsible for shared information. Does `csvgetline` return the original data or make a copy? We decided that the return value of `csvgetline` is a pointer to the original input, which will be overwritten when the next line is read. Fields will be built in a copy of the input line, and `csvfield` will return a pointer to the field within the copy. With this arrangement, the user must make another copy if a particular line or field is to be saved or changed, and it is the user's responsibility to release that storage when it is no longer needed.

Who opens and closes the input file? Whoever opens an input file should do the corresponding close: matching tasks should be done at the same level or place. We will assume that `csvgetline` is called with a `FILE` pointer to an already-open file that the caller will close when processing is complete.

Managing the resources shared or passed across the boundary between a library and its callers is a difficult task, and there are often sound (合理的) but conflicting reasons to prefer various design choices. Errors and misunderstandings about the shared responsibilities are a frequent source of bugs.

Error handling. Because `csvgetline` returns `NULL`, there is no good way to distinguish end of file from an error like running out of memory; similarly, access to a non-existent field causes no error. By analogy with `ferror`, we could add another function `csvgeterror` to the interface to report the most recent error, but for simplicity we will leave it out of this version.

As a principle, library routines should not just die when an error occurs; error status should be returned to the caller for appropriate action. Nor should they print messages or pop up dialog boxes, since they may be running in an environment where a message would interfere with something else. Error handling is a topic worth a separate discussion of its own, later in this chapter.

Specification. The choices made above should be collected in one place as a specification of the services that `csvgetline` provides and how it is to be used. In a large project, the specification precedes the implementation, because specifiers and implementers are usually different people and may be in different organizations. In practice, however, work often proceeds in parallel, with specification and code evolving together, although sometimes the "specification" is written only after the fact to describe approximately what the code does.

The best approach is to write the specification early and revise it as we learn from the ongoing implementation. The more accurate and careful a specification is, the more likely that the resulting program will work well. Even for personal programs, it is valuable to prepare a reasonably thorough specification because it encourages consideration of alternatives and records the choices made.

For our purposes, the specification would include function prototypes and a detailed prescription (命令) of behavior, responsibilities and assumptions:

> Fields are separated by commas.
> A field may be enclosed in double-quote characters "...".
> A quoted field may contain commas but not newlines.
> A quoted field may contain double-quote characters ", represented by
> "".
> Fields may be empty; "" and an empty string both represent an empty
> field.
> Leading and trailing white space is preserved.
>
> ```
> char *csvgetline(FILE *f);
> ```
>
> reads one line from open input file; assumes that input lines are terminated by `\r`, `\n`, `\r\n`, or `EOF`.
>
> returns pointer to line, with terminator removed, or `NULL` if `EOF` occurred.
>
> line may be of arbitrary length; returns `NULL` if memory limit exceeded.
>
> line must be treated as read-only storage; caller must make a copy to preserve or change contents.