4

Interfaces

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

Robert Frost, *Mending Wall*

The essence 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 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.

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.
- 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 2 we looked at the individual pieces—the data structures—from which a system is built. In Chapter 3, 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 interfaces that frustrate and impede programmers every day.

4.1 Comma-Separated Values

Comma-separated values, or CSV, is the term for a natural and widely used representation for tabular 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","Lines 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, 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	86-1/4	+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 to invoke a browser, wait, watch a barrage of advertisements, type a list of stocks, wait, wait, wait, then watch another barrage, all to get a few numbers. To process the numbers further requires even more interaction; selecting the ''Download Spreadsheet Format'' link retrieves a file that contains much the same 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 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 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 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=sl1d1t1c1ohgv"

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 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 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.

4.2 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 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 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 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 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 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 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 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.

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 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 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 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, since the ''library'' is small and only a prototype anyway. Imagine, however, that after sitting on the shelf 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 trouble that surfaces years later. This scenario 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.

4.3 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 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 line

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 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 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 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 f;
        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.

char *csvfield(int n);
    fields are numbered from 0.
    returns n-th field from last line read by csvgetline;
        returns NULL if n < 0 or beyond last field.
    fields are separated by commas.
    fields may be surrounded by "..."; such quotes are removed;
        within "...", "" is replaced by " and comma is not a separator.
    in unquoted fields, quotes are regular characters.
    there can be an arbitrary number of fields of any length;
        returns NULL if memory limit exceeded.
    field must be treated as read-only storage;
        caller must make a copy to preserve or change contents.
    behavior undefined if called before csvgetline is called.
```

```
int csvnfield(void);
    returns number of fields on last line read by csvgetline.
    behavior undefined if called before csvgetline is called.
```

This specification still leaves open questions. For example, what values should be returned by `csvfield` and `csvnfield` if they are called after `csvgetline` has encountered `EOF`? How should ill-formed fields be handled? Nailing down all such puzzles is difficult even for a tiny system, and very challenging for a large one, though it is important to try. One often doesn't discover oversights and omissions until implementation is underway.

The rest of this section contains a new implementation of `csvgetline` that matches the specification. The library is broken into two files, a header `csv.h` that contains the function declarations that represent the public part of the interface, and an implementation file `csv.c` that contains the code. Users include `csv.h` in their source code and link their compiled code with the compiled version of `csv.c`; the source need never be visible.

Here is the header file:

```
/* csv.h: interface for csv library */

extern char *csvgetline(FILE *f); /* read next input line */
extern char *csvfield(int n);     /* return field n */
extern int csvnfield(void);       /* return number of fields */
```

The internal variables that store text and the internal functions like `split` are declared `static` so they are visible only within the file that contains them. This is the simplest way to hide information in a C program.

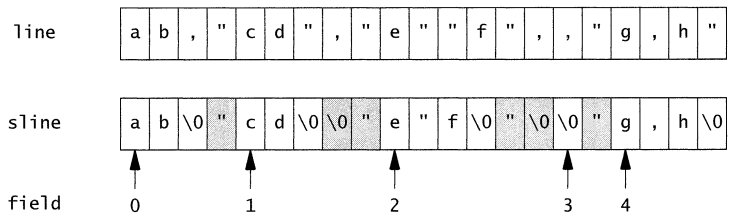
```
enum { NOMEM = -2 };          /* out of memory signal */

static char *line    = NULL;  /* input chars */
static char *sline   = NULL;  /* line copy used by split */
static int  maxline  = 0;     /* size of line[] and sline[] */
static char **field  = NULL;  /* field pointers */
static int  maxfield = 0;     /* size of field[] */
static int  nfield   = 0;     /* number of fields in field[] */

static char fieldsep[] = ","; /* field separator chars */
```

The variables are initialized statically as well. These initial values are used to test whether to create or grow arrays.

These declarations describe a simple data structure. The `line` array holds the input line; the `sline` array is created by copying characters from `line` and terminating each field. The `field` array points to entries in `sline`. This diagram shows the state of these three arrays after the input line `ab,"cd","e""f",,"g,h"` has been processed. Shaded elements in `sline` are not part of any field.



Here is the function `csvgetline` itself:

```
/* csvgetline:  get one line, grow as needed */
/* sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
char *csvgetline(FILE *fin)
{
    int i, c;
    char *newl, *news;

    if (line == NULL) {          /* allocate on first call */
        maxline = maxfield = 1;
        line = (char *) malloc(maxline);
        sline = (char *) malloc(maxline);
        field = (char **) malloc(maxfield*sizeof(field[0]));
        if (line == NULL || sline == NULL || field == NULL) {
            reset();
            return NULL;         /* out of memory */
        }
    }
    for (i=0; (c=getc(fin))!=EOF && !endofline(fin,c); i++) {
        if (i >= maxline-1) {    /* grow line */
            maxline *= 2;        /* double current size */
            newl = (char *) realloc(line, maxline);
            news = (char *) realloc(sline, maxline);
            if (newl == NULL || news == NULL) {
                reset();
                return NULL;     /* out of memory */
            }
            line = newl;
            sline = news;
        }
        line[i] = c;
    }
    line[i] = '\0';
    if (split() == NOMEM) {
        reset();
        return NULL;             /* out of memory */
    }
    return (c == EOF && i == 0) ? NULL : line;
}
```

An incoming line is accumulated in `line`, which is grown as necessary by a call to `realloc`; the size is doubled on each growth, as in Section 2.6. The `sline` array is

kept the same size as `line`; `csvgetline` calls `split` to create the field pointers in a separate array `field`, which is also grown as needed.

As is our custom, we start the arrays very small and grow them on demand, to guarantee that the array-growing code is exercised. If allocation fails, we call `reset` to restore the globals to their starting state, so a subsequent call to `csvgetline` has a chance of succeeding:

```
/* reset: set variables back to starting values */
static void reset(void)
{
    free(line); /* free(NULL) permitted by ANSI C */
    free(sline);
    free(field);
    line = NULL;
    sline = NULL;
    field = NULL;
    maxline = maxfield = nfield = 0;
}
```

The `endofline` function handles the problem that an input line may be terminated by a carriage return, a newline, both, or even `EOF`:

```
/* endofline: check for and consume \r, \n, \r\n, or EOF */
static int endofline(FILE *fin, int c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        c = getc(fin);
        if (c != '\n' && c != EOF)
            ungetc(c, fin); /* read too far; put c back */
    }
    return eol;
}
```

A separate function is necessary, since the standard input functions do not handle the rich variety of perverse formats encountered in real inputs.

Our prototype used `strtok` to find the next token by searching for a separator character, normally a comma, but this made it impossible to handle quoted commas. A major change in the implementation of `split` is necessary, though its interface need not change. Consider these input lines:

```
"",,""
,"",
,,
```

Each line has three empty fields. Making sure that `split` parses them and other odd inputs correctly complicates it significantly, an example of how special cases and boundary conditions can come to dominate a program.

```
/* split: split line into fields */
static int split(void)
{
    char *p, **newf;
    char *sepp; /* pointer to temporary separator character */
    int sepc;   /* temporary separator character */

    nfield = 0;
    if (line[0] == '\0')
        return 0;
    strcpy(sline, line);
    p = sline;

    do {
        if (nfield >= maxfield) {
            maxfield *= 2;          /* double current size */
            newf = (char **) realloc(field,
                        maxfield * sizeof(field[0]));
            if (newf == NULL)
                return NOMEM;
            field = newf;
        }
        if (*p == '"')
            sepp = advquoted(++p);  /* skip initial quote */
        else
            sepp = p + strcspn(p, fieldsep);
        sepc = sepp[0];
        sepp[0] = '\0';             /* terminate field */
        field[nfield++] = p;
        p = sepp + 1;
    } while (sepc == ',');

    return nfield;
}
```

The loop grows the array of field pointers if necessary, then calls one of two other functions to locate and process the next field. If the field begins with a quote, `advquoted` finds the field and returns a pointer to the separator that ends the field. Otherwise, to find the next comma we use the library function `strcspn(p,s)`, which searches a string `p` for the next occurrence of any character in string `s`; it returns the number of characters skipped over.

Quotes within a field are represented by two adjacent quotes, so `advquoted` squeezes those into a single one; it also removes the quotes that surround the field. Some complexity is added by an attempt to cope with plausible inputs that don't match the specification, such as `"abc"def`. In such cases, we append whatever follows the second quote until the next separator as part of this field. Microsoft Excel appears to use a similar algorithm.

```
/* advquoted: quoted field; return pointer to next separator */
static char *advquoted(char *p)
{
    int i, j;

    for (i = j = 0; p[j] != '\0'; i++, j++) {
        if (p[j] == '"' && p[++j] != '"') {
            /* copy up to next separator or \0 */
            int k = strcspn(p+j, fieldsep);
            memmove(p+i, p+j, k);
            i += k;
            j += k;
            break;
        }
        p[i] = p[j];
    }
    p[i] = '\0';
    return p + j;
}
```

Since the input line is already split, `csvfield` and `csvnfield` are trivial:

```
/* csvfield:  return pointer to n-th field */
char *csvfield(int n)
{
    if (n < 0 || n >= nfield)
        return NULL;
    return field[n];
}

/* csvnfield:  return number of fields */
int csvnfield(void)
{
    return nfield;
}
```

Finally, we can modify the test driver to exercise this version of the library; since it keeps a copy of the input line, which the prototype does not, it can print the original line before printing the fields:

```
/* csvtest main: test CSV library */
int main(void)
{
    int i;
    char *line;

    while ((line = csvgetline(stdin)) != NULL) {
        printf("line = '%s'\n", line);
        for (i = 0; i < csvnfield(); i++)
            printf("field[%d] = '%s'\n", i, csvfield(i));
    }
    return 0;
}
```

This completes our C version. It handles arbitrarily large inputs and does something sensible even with perverse data. The price is that it is more than four times as long as the first prototype and some of the code is intricate. Such expansion of size and complexity is a typical result of moving from prototype to production.

Exercise 4-1. There are several degrees of laziness for field-splitting; among the possibilities are to split all at once but only when some field is requested, to split only the field requested, or to split up to the field requested. Enumerate possibilities, assess their potential difficulty and benefits, then write them and measure their speeds. □

Exercise 4-2. Add a facility so separators can be changed (a) to an arbitrary class of characters; (b) to different separators for different fields; (c) to a regular expression (see Chapter 9). What should the interface look like? □

Exercise 4-3. We chose to use the static initialization provided by C as the basis of a one-time switch: if a pointer is `NULL` on entry, initialization is performed. Another possibility is to require the user to call an explicit initialization function, which could include suggested initial sizes for arrays. Implement a version that combines the best of both. What is the role of `reset` in your implementation? □

Exercise 4-4. Design and implement a library for creating CSV-formatted data. The simplest version might take an array of strings and print them with quotes and commas. A more sophisticated version might use a format string analogous to `printf`. Look at Chapter 9 for some suggestions on notation. □

4.4 A C++ Implementation

In this section we will write a C++ version of the CSV library to address some of the remaining limitations of the C version. This will entail some changes to the specification, of which the most important is that the functions will handle C++ strings instead of C character arrays. The use of C++ strings will automatically resolve some of the storage management issues, since the library functions will manage the memory for us. In particular, the field routines will return strings that can be modified by the caller, a more flexible design than the previous version.

A class `Csv` defines the public face, while neatly hiding the variables and functions of the implementation. Since a class object contains all the state for an instance, we can instantiate multiple `Csv` variables; each is independent of the others so multiple CSV input streams can operate at the same time.

```
class Csv { // read and parse comma-separated values
    // sample input: "LU",86.25,"11/4/1998","2:19PM",+4.0625

  public:
    Csv(istream& fin = cin, string sep = ",") :
        fin(fin), fieldsep(sep) {}

    int getline(string&);
    string getfield(int n);
    int getnfield() const { return nfield; }

  private:
    istream& fin;           // input file pointer
    string line;            // input line
    vector<string> field;   // field strings
    int nfield;             // number of fields
    string fieldsep;        // separator characters

    int split();
    int endofline(char);
    int advplain(const string& line, string& fld, int);
    int advquoted(const string& line, string& fld, int);
};
```

Default parameters for the constructor are defined so a default `Csv` object will read from the standard input stream and use the normal field separator; either can be replaced with explicit values.

To manage strings, the class uses the standard C++ `string` and `vector` classes rather than C-style strings. There is no non-existent state for a `string`: "empty" means only that the length is zero, and there is no equivalent of `NULL`, so we can't use that as an end of file signal. Thus `Csv::getline` returns the input line through an argument by reference, reserving the function value itself for end of file and error reports.

```
// getline: get one line, grow as needed
int Csv::getline(string& str)
{
    char c;

    for (line = ""; fin.get(c) && !endofline(c); )
        line += c;
    split();
    str = line;
    return !fin.eof();
}
```

The += operator is overloaded to append a character to a string.

Minor changes are needed in `endofline`. Again, we have to read the input a character at a time, since none of the standard input routines can handle the variety of inputs.

```
// endofline: check for and consume \r, \n, \r\n, or EOF
int Csv::endofline(char c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        fin.get(c);
        if (!fin.eof() && c != '\n')
            fin.putback(c); // read too far
    }
    return eol;
}
```

Here is the new version of `split`:

```
// split: split line into fields
int Csv::split()
{
    string fld;
    int i, j;

    nfield = 0;
    if (line.length() == 0)
        return 0;
    i = 0;

    do {
        if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i);  // skip quote
        else
            j = advplain(line, fld, i);
        if (nfield >= field.size())
            field.push_back(fld);
        else
            field[nfield] = fld;
        nfield++;
        i = j + 1;
    } while (j < line.length());

    return nfield;
}
```

Since `strcspn` doesn't work on C++ strings, we must change both `split` and `advquoted`. The new version of `advquoted` uses the C++ standard function `find_first_of` to locate the next occurrence of a separator character. The call `s.find_first_of(fieldsep,j)` searches the string `s` for the first instance of any character in `fieldsep` that occurs at or after position `j`. If it fails to find an instance, it returns an index beyond the end of the string, so we must bring it back within range. The inner `for` loop that follows appends characters up to the separator to the field being accumulated in `fld`.

```
// advquoted: quoted field; return index of next separator
int Csv::advquoted(const string& s, string& fld, int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
            int k = s.find_first_of(fieldsep, j);
            if (k > s.length()) // no separator found
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
        fld += s[j];
    }
    return j;
}
```

The function `find_first_of` is also used in a new function `advplain`, which advances over a plain unquoted field. Again, this change is required because C string functions like `strcspn` cannot be applied to C++ strings, which are an entirely different data type.

```
// advplain: unquoted field; return index of next separator
int Csv::advplain(const string& s, string& fld, int i)
{
    int j;

    j = s.find_first_of(fieldsep, i); // look for separator
    if (j > s.length())               // none found
        j = s.length();
    fld = string(s, i, j-i);
    return j;
}
```

As before, `Csv::getfield` is trivial, while `Csv::getnfield` is so short that it is implemented in the class definition.

```
// getfield: return n-th field
string Csv::getfield(int n)
{
    if (n < 0 || n >= nfield)
        return "";
    else
        return field[n];
}
```

Our test program is a simple variant of the earlier one:

```
// Csvtest main: test Csv class
int main(void)
{
    string line;
    Csv csv;

    while (csv.getline(line) != 0) {
        cout << "line = '" << line <<"'\n";
        for (int i = 0; i < csv.getnfield(); i++)
            cout << "field[" << i << "] = '"
                 << csv.getfield(i) << "'\n";
    }
    return 0;
}
```

The usage is different than with the C version, though only in a minor way. Depending on the compiler, the C++ version is anywhere from 40 percent to four times slower than the C version on a large input file of 30,000 lines with about 25 fields per line. As we saw when comparing versions of `markov`, this variability is a reflection on library maturity. The C++ source program is about 20 percent shorter.

Exercise 4-5. Enhance the C++ implementation to overload subscripting with `operator[]` so that fields can be accessed as `csv[i]`. □

Exercise 4-6. Write a Java version of the CSV library, then compare the three implementations for clarity, robustness, and speed. □

Exercise 4-7. Repackage the C++ version of the CSV code as an STL iterator. □

Exercise 4-8. The C++ version permits multiple independent `Csv` instances to operate concurrently without interfering, a benefit of encapsulating all the state in an object that can be instantiated multiple times. Modify the C version to achieve the same effect by replacing the global data structures with structures that are allocated and initialized by an explicit `csvnew` function. □

4.5 Interface Principles

In the previous sections we were working out the details of an interface, which is the detailed boundary between code that provides a service and code that uses it. An interface defines what some body of code does for its users, how the functions and perhaps data members can be used by the rest of the program. Our CSV interface provides three functions—read a line, get a field, and return the number of fields—which are the only operations that can be performed.

To prosper, an interface must be well suited for its task—simple, general, regular, predictable, robust—and it must adapt gracefully as its users and its implementation

change. Good interfaces follow a set of principles. These are not independent or even consistent, but they help us describe what happens across the boundary between two pieces of software.

Hide implementation details. The implementation behind the interface should be hidden from the rest of the program so it can be changed without affecting or breaking anything. There are several terms for this kind of organizing principle; information hiding, encapsulation, abstraction, modularization, and the like all refer to related ideas. An interface should hide details of the implementation that are irrelevant to the client (user) of the interface. Details that are invisible can be changed without affecting the client, perhaps to extend the interface, make it more efficient, or even replace its implementation altogether.

The basic libraries of most programming languages provide familiar examples, though not always especially well-designed ones. The C standard I/O library is among the best known: a couple of dozen functions that open, close, read, write, and otherwise manipulate files. The implementation of file I/O is hidden behind a data type `FILE*`, whose properties one might be able to see (because they are often spelled out in `<stdio.h>`) but should not exploit.

If the header file does not include the actual structure declaration, just the name of the structure, this is sometimes called an *opaque type*, since its properties are not visible and all operations take place through a pointer to whatever real object lurks behind.

Avoid global variables; wherever possible it is better to pass references to all data through function arguments.

We strongly recommend against publicly visible data in all forms; it is too hard to maintain consistency of values if users can change variables at will. Function interfaces make it easier to enforce access rules, but this principle is often violated. The predefined I/O streams like `stdin` and `stdout` are almost always defined as elements of a global array of `FILE` structures:

```
extern FILE    __iob[_NFILE];
#define stdin  (&__iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

This makes the implementation completely visible; it also means that one can't assign to `stdin`, `stdout` or `stderr`, even though they look like variables. The peculiar name `__iob` uses the ANSI C convention of two leading underscores for private names that must be visible, which makes the names less likely to conflict with names in a program.

Classes in C++ and Java are better mechanisms for hiding information; they are central to the proper use of those languages. The container classes of the C++ Standard Template Library that we used in Chapter 3 carry this even further: aside from some performance guarantees there is no information about implementation, and library creators can use any mechanism they like.

Choose a small orthogonal set of primitives. An interface should provide as much functionality as necessary but no more, and the functions should not overlap excessively in their capabilities. Having lots of functions may make the library easier to use—whatever one needs is there for the taking. But a large interface is harder to write and maintain, and sheer size may make it hard to learn and use as well. ''Application program interfaces'' or APIs are sometimes so huge that no mortal can be expected to master them.

In the interest of convenience, some interfaces provide multiple ways of doing the same thing, a tendency that should be resisted. The C standard I/O library provides at least four different functions that will write a single character to an output stream:

```
char c;
putc(c, fp);
fputc(c, fp);
fprintf(fp, "%c", c);
fwrite(&c, sizeof(char), 1, fp);
```

If the stream is `stdout`, there are several more possibilities. These are convenient, but not all are necessary.

Narrow interfaces are to be preferred to wide ones, at least until one has strong evidence that more functions are needed. Do one thing, and do it well. Don't add to an interface just because it's possible to do so, and don't fix the interface when it's the implementation that's broken. For instance, rather than having `memcpy` for speed and `memmove` for safety, it would be better to have one function that was always safe, and fast when it could be.

Don't reach behind the user's back. A library function should not write secret files and variables or change global data, and it should be circumspect about modifying data in its caller. The `strtok` function fails several of these criteria. It is a bit of a surprise that `strtok` writes null bytes into the middle of its input string. Its use of the null pointer as a signal to pick up where it left off last time implies secret data held between calls, a likely source of bugs, and it precludes concurrent uses of the function. A better design would provide a single function that tokenizes an input string. For similar reasons, our second C version can't be used for two input streams; see Exercise 4-8.

The use of one interface should not demand another one just for the convenience of the interface designer or implementer. Instead, make the interface self-contained, or failing that, be explicit about what external services are required. Otherwise, you place a maintenance burden on the client. An obvious example is the pain of managing huge lists of header files in C and C++ source; header files can be thousands of lines long and include dozens of other headers.

Do the same thing the same way everywhere. Consistency and regularity are important. Related things should be achieved by related means. The basic `str...` functions in the C library are easy to use without documentation because they all behave about the same: data flows from right to left, the same direction as in an assignment

statement, and they all return the resulting string. On the other hand, in the C Standard I/O library it is hard to predict the order of arguments to functions. Some have the `FILE*` argument first, some last; others have various orders for size and number of elements. The algorithms for STL containers present a very uniform interface, so it is easy to predict how to use an unfamiliar function.

External consistency, behaving like something else, is also a goal. For example, the `mem...` functions were designed after the `str...` functions in C, but borrowed their style. The standard I/O functions `fread` and `fwrite` would be easier to remember if they looked like the `read` and `write` functions they were based on. Unix command-line options are introduced by a minus sign, but a given option letter may mean completely different things, even between related programs.

If wildcards like the `*` in `*.exe` are all expanded by a command interpreter, behavior is uniform. If they are expanded by individual programs, non-uniform behavior is likely. Web browsers take a single mouse click to follow a link, but other applications take two clicks to start a program or follow a link; the result is that many people automatically click twice regardless.

These principles are easier to follow in some environments than others, but they still stand. For instance, it's hard to hide implementation details in C, but a good programmer will not exploit them, because to do so makes the details part of the interface and violates the principle of information hiding. Comments in header files, names with special forms (such as `__iob`), and so on are ways of encouraging good behavior when it can't be enforced.

No matter what, there is a limit to how well we can do in designing an interface. Even the best interfaces of today may eventually become the problems of tomorrow, but good design can push tomorrow off a while longer.

4.6 Resource Management

One of the most difficult problems in designing the interface for a library (or a class or a package) is to manage resources that are owned by the library or that are shared by the library and those who call it. The most obvious such resource is memory—who is responsible for allocating and freeing storage?—but other shared resources include open files and the state of variables whose values are of common interest. Roughly, the issues fall into the categories of initialization, maintaining state, sharing and copying, and cleaning up.

The prototype of our CSV package used static initialization to set the initial values for pointers, counts, and the like. But this choice is limiting since it prevents restarting the routines in their initial state once one of the functions has been called. An alternative is to provide an initialization function that sets all internal values to the correct initial values. This permits restarting, but relies on the user to call it explicitly. The `reset` function in the second version could be made public for this purpose.

In C++ and Java, constructors are used to initialize data members of classes. Properly defined constructors ensure that all data members are initialized and that there is no way to create an uninitialized class object. A group of constructors can support various kinds of initializers; we might provide `Csv` with one constructor that takes a file name and another that takes an input stream.

What about copies of information managed by a library, such as the input lines and fields? Our C `csvgetline` program provides direct access to the input strings (line and fields) by returning pointers to them. This unrestricted access has several drawbacks. It's possible for the user to overwrite memory so as to render other information invalid; for example, an expression like

```
strcpy(csvfield(1), csvfield(2));
```

could fail in a variety of ways, most likely by overwriting the beginning of field 2 if field 2 is longer than field 1. The user of the library must make a copy of any information to be preserved beyond the next call to `csvgetline`; in the following sequence, the pointer might well be invalid at the end if the second `csvgetline` causes a reallocation of its line buffer.

```
char *p;

csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
/* p could be invalid here */
```

The C++ version is safer because the strings are copies that can be changed at will.

Java uses references to refer to objects, that is, any entity other than one of the basic types like `int`. This is more efficient than making a copy, but one can be fooled into thinking that a reference is a copy; we had a bug like that in an early version of our Java `markov` program and this issue is a perennial source of bugs involving strings in C. Clone methods provide a way to make a copy when necessary.

The other side of initialization or construction is finalization or destruction—cleaning up and recovering resources when some entity is no longer needed. This is particularly important for memory, since a program that fails to recover unused memory will eventually run out. Much modern software is embarrassingly prone to this fault. Related problems occur when open files are to be closed: if data is being buffered, the buffer may have to be flushed (and its memory reclaimed). For standard C library functions, flushing happens automatically when the program terminates normally, but it must otherwise be programmed. The C and C++ standard function `atexit` provides a way to get control just before a program terminates normally; interface implementers can use this facility to schedule cleanup.

Free a resource in the same layer that allocated it. One way to control resource allocation and reclamation is to have the same library, package, or interface that allocates

a resource be responsible for freeing it. Another way of saying this is that the allocation state of a resource should not change across the interface. Our CSV libraries read data from files that have already been opened, so they leave them open when they are done. The caller of the library needs to close the files.

C++ constructors and destructors help enforce this rule. When a class instance goes out of scope or is explicitly destroyed, the destructor is called; it can flush buffers, recover memory, reset values, and do whatever else is necessary. Java does not provide an equivalent mechanism. Although it is possible to define a finalization method for a class, there is no assurance that it will run at all, let alone at a particular time, so cleanup actions cannot be guaranteed to occur, although it is often reasonable to assume they will.

Java does provide considerable help with memory management because it has built-in *garbage collection.* As a program runs, it allocates new objects. There is no way to deallocate them explicitly, but the run-time system keeps track of which objects are still in use and which are not, and periodically returns unused ones to the available memory pool.

There are a variety of techniques for garbage collection. Some schemes keep track of the number of uses of each object, its *reference count*, and free an object when its reference count goes to zero. This technique can be used explicitly in C and C++ to manage shared objects. Other algorithms periodically follow a trail from the allocation pool to all referenced objects. Objects that are found this way are still in use; objects that are not referred to by any other object are not in use and can be reclaimed.

The existence of automatic garbage collection does *not* mean that there are no memory-management issues in a design. We still have to determine whether interfaces return references to shared objects or copies of them, and this affects the entire program. Nor is garbage collection free—there is overhead to maintain information and to reclaim unused memory, and collection may happen at unpredictable times.

All of these problems become more complicated if a library is to be used in an environment where more than one thread of control can be executing its routines at the same time, as in a multi-threaded Java program.

To avoid problems, it is necessary to write code that is *reentrant*, which means that it works regardless of the number of simultaneous executions. Reentrant code will avoid global variables, static local variables, and any other variable that could be modified while another thread is using it. The key to good multi-thread design is to separate the components so they share nothing except through well-defined interfaces. Libraries that inadvertently expose variables to sharing destroy the model. (In a multi-thread program, `strtok` is a disaster, as are other functions in the C library that store values in internal static memory.) If variables might be shared, they must be protected by some kind of locking mechanism to ensure that only one thread at a time accesses them. Classes are a big help here because they provide a focus for discussing sharing and locking models. Synchronized methods in Java provide a way for one thread to lock an entire class or instance of a class against simultaneous modifica-

tion by some other thread; synchronized blocks permit only one thread at a time to execute a section of code.

Multi-threading adds significant complexity to programming issues, and is too big a topic for us to discuss in detail here.

4.7 Abort, Retry, Fail?

In the previous chapters we used functions like `eprintf` and `estrdup` to handle errors by displaying a message before terminating execution. For example, `eprintf` behaves like `fprintf(stderr,...)`, but exits the program with an error status after reporting the error. It uses the `<stdarg.h>` header and the `vfprintf` library routine to print the arguments represented by the `...` in the prototype. The `stdarg` library must be initialized by a call to `va_start` and terminated by `va_end`. We will use more of this interface in Chapter 9.

```
#include <stdarg.h>
#include <string.h>
#include <errno.h>

/* eprintf: print error message and exit */
void eprintf(char *fmt, ...)
{
    va_list args;

    fflush(stdout);
    if (progname() != NULL)
        fprintf(stderr, "%s: ", progname());

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);

    if (fmt[0] != '\0' && fmt[strlen(fmt)-1] == ':')
        fprintf(stderr, " %s", strerror(errno));
    fprintf(stderr, "\n");
    exit(2); /* conventional value for failed execution */
}
```

If the format argument ends with a colon, `eprintf` calls the standard C function `strerror`, which returns a string containing any additional system error information that might be available. We also wrote `weprintf`, similar to `eprintf`, that displays a warning but does not exit. The `printf`-like interface is convenient for building up strings that might be printed or displayed in a dialog box.

Similarly, `estrdup` tries to make a copy of a string, and exits with a message (via `eprintf`) if it runs out of memory:

```
/* estrdup: duplicate a string, report if error */
char *estrdup(char *s)
{
    char *t;

    t = (char *) malloc(strlen(s)+1);
    if (t == NULL)
        eprintf("estrdup(\"%.20s\") failed:", s);
    strcpy(t, s);
    return t;
}
```

and `emalloc` provides a similar service for calls to `malloc`:

```
/* emalloc: malloc and report if error */
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL)
        eprintf("malloc of %u bytes failed:", n);
    return p;
}
```

A matching header file called `eprintf.h` declares these functions:

```
/* eprintf.h: error wrapper functions */
extern  void    eprintf(char *, ...);
extern  void    weprintf(char *, ...);
extern  char    *estrdup(char *);
extern  void    *emalloc(size_t);
extern  void    *erealloc(void *, size_t);
extern  char    *progname(void);
extern  void    setprogname(char *);
```

This header is included in any file that calls one of the error functions. Each error message also includes the name of the program if it has been set by the caller; this is set and retrieved by the trivial functions `setprogname` and `progname`, declared in the header file and defined in the source file with `eprintf`:

```
static char *name = NULL;  /* program name for messages */

/* setprogname: set stored name of program */
void setprogname(char *str)
{
    name = estrdup(str);
}

/* progname: return stored name of program */
char *progname(void)
{
    return name;
}
```

Typical usage looks like this:

```
int main(int argc, char *argv[])
{
    setprogname("markov");
    ...
    f = fopen(argv[i], "r");
    if (f == NULL)
        eprintf("can't open %s:", argv[i]);
    ...
}
```

which prints output like this:

```
markov: can't open psalm.txt: No such file or directory
```

We find these wrapper functions convenient for our own programming, since they unify error handling and their very existence encourages us to catch errors instead of ignoring them. There is nothing special about our design, however, and you might prefer some variant for your own programs.

Suppose that rather than writing functions for our own use, we are creating a library for others to use in their programs. What should a function in that library do if an unrecoverable error occurs? The functions we wrote earlier in this chapter display a message and die. This is acceptable behavior for many programs, especially small stand-alone tools and applications. For other programs, however, quitting is wrong since it prevents the rest of the program from attempting any recovery; for instance, a word processor must recover from errors so it does not lose the document that you are typing. In some situations a library routine should not even display a message, since the program may be running in an environment where a message will interfere with displayed data or disappear without a trace. A useful alternative is to record diagnostic output in an explicit ''log file,'' where it can be monitored independently.

Detect errors at a low level, handle them at a high level. As a general principle, errors should be detected at as low a level as possible, but handled at a high level. In most cases, the caller should determine how to handle an error, not the callee. Library routines can help in this by failing gracefully; that reasoning led us to return `NULL` for a non-existent field rather than aborting. Similarly, `csvgetline` returns `NULL` no matter how many times it is called after the first end of file.

Appropriate return values are not always obvious, as we saw in the earlier discussion about what `csvgetline` should return. We want to return as much useful information as possible, but in a form that is easy for the rest of the program to use. In C, C++ and Java, that means returning something as the function value, and perhaps other values through reference (pointer) arguments. Many library functions rely on the ability to distinguish normal values from error values. Input functions like `getchar` return a `char` for valid data, and some non-`char` value like `EOF` for end of file or error.

This mechanism doesn't work if the function's legal return values take up all possible values. For example a mathematical function like `log` can return any floating-point number. In IEEE floating point, a special value called `NaN` (''not a number'') indicates an error and can be returned as an error signal.

Some languages, such as Perl and Tcl, provide a low-cost way to group two or more values into a *tuple*. In such languages, a function value and any error state can be easily returned together. The C++ STL provides a `pair` data type that can also be used in this way.

It is desirable to distinguish various exceptional values like end of file and error states if possible, rather than lumping them together into a single value. If the values can't readily be separated, another option is to return a single ''exception'' value and provide another function that returns more detail about the last error.

This is the approach used in Unix and in the C standard library, where many system calls and library functions return `-1` but also set a global variable called `errno` that encodes the specific error; `strerror` returns a string associated with the error number. On our system, this program:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

/* errno main: test errno */
int main(void)
{
    double f;

    errno = 0;  /* clear error state */
    f = log(-1.23);
    printf("%f %d %s\n", f, errno, strerror(errno));
    return 0;
}
```

prints

```
nan0x10000000 33 Domain error
```

As shown, `errno` must be cleared first; then if an error occurs, `errno` will be set to a non-zero value.

Use exceptions only for exceptional situations. Some languages provide *exceptions* to catch unusual situations and recover from them; they provide an alternate flow of control when something bad happens. Exceptions should not be used for handling expected return values. Reading from a file will eventually produce an end of file; this should be handled with a return value, not by an exception.

In Java, one writes

```
String fname = "someFileName";
try {
    FileInputStream in = new FileInputStream(fname);
    int c;
    while ((c = in.read()) != -1)
        System.out.print((char) c);
    in.close();
} catch (FileNotFoundException e) {
    System.err.println(fname + " not found");
} catch (IOException e) {
    System.err.println("IOException: " + e);
    e.printStackTrace();
}
```

The loop reads characters until end of file, an expected event that is signaled by a return value of `-1` from `read`. If the file can't be opened, that raises an exception, however, rather than setting the input stream to `null` as would be done in C or C++. Finally, if some other I/O error happens in the `try` block, it is also exceptional, and it is caught by the `IOException` clause.

Exceptions are often overused. Because they distort the flow of control, they can lead to convoluted constructions that are prone to bugs. It is hardly exceptional to fail to open a file; generating an exception in this case strikes us as over-engineering. Exceptions are best reserved for truly unexpected events, such as file systems filling up or floating-point errors.

For C programs, the pair of functions `setjmp` and `longjmp` provide a much lower-level service upon which an exception mechanism can be built, but they are sufficiently arcane that we won't go into them here.

What about recovery of resources when an error occurs? Should a library attempt a recovery when something goes wrong? Not usually, but it might do a service by making sure that it leaves information in as clean and harmless a state as possible. Certainly unused storage should be reclaimed. If variables might be still accessible, they should be set to sensible values. A common source of bugs is trying to use a pointer that points to freed storage. If error-handling code sets pointers to zero after freeing what they point to, this won't go undetected. The `reset` function in the second version of the CSV library was an attempt to address these issues. In general, aim to keep the library usable after an error has occurred.

4.8 User Interfaces

Thus far we have talked mainly about interfaces among the components of a program or between programs. But there is another important kind of interface, between a program and its human users.

Most of the example programs in this book are text-based, so their user interfaces tend to be straightforward. As we discussed in the previous section, errors should be

detected and reported, and recovery attempted where it makes sense. Error output should include all available information and should be as meaningful as possible out of context; a diagnostic should not say

```
estrdup failed
```

when it could say

```
markov: estrdup("Derrida") failed: Memory limit reached
```

It costs nothing to add the extra information as we did in `estrdup`, and it may help a user to identify a problem or provide valid input.

Programs should display information about proper usage when an error is made, as shown in functions like

```
/* usage: print usage message and exit */
void usage(void)
{
    fprintf(stderr, "usage: %s [-d] [-n nwords]"
        " [-s seed] [files ...]\n", progname());
    exit(2);
}
```

The program name identifies the source of the message, which is especially important if this is part of a larger process. If a program presents a message that just says `syntax error` or `estrdup failed`, the user might have no idea who said it.

The text of error messages, prompts, and dialog boxes should state the form of valid input. Don't say that a parameter is too large; report the valid range of values. When possible, the text should be valid input itself, such as the full command line with the parameter set properly. In addition to steering users toward proper use, such output can be captured in a file or by a mouse sweep and then used to run some further process. This points out a weakness of dialog boxes: their contents are hard to grab for later use.

One effective way to create a good user interface for input is by designing a specialized language for setting parameters, controlling actions, and so on; a good notation can make a program easy to use while it helps organize an implementation. Language-based interfaces are the subject of Chapter 9.

Defensive programming, that is, making sure that a program is invulnerable to bad input, is important both for protecting users against themselves and also as a security mechanism. This is discussed more in Chapter 6, which talks about program testing.

For most people, graphical interfaces are *the* user interface for their computers. Graphical user interfaces are a huge topic, so we will say only a few things that are germane to this book. First, graphical interfaces are hard to create and make ''right'' since their suitability and success depend strongly on human behavior and expectations. Second, as a practical matter, if a system has a user interface, there is usually more code to handle user interaction than there is in whatever algorithms do the work.

Nevertheless, familiar principles apply to both the external design and the internal implementation of user interface software. From the user's standpoint, style issues like simplicity, clarity, regularity, uniformity, familiarity, and restraint all contribute to an interface that is easy to use; the absence of such properties usually goes along with unpleasant or awkward interfaces.

Uniformity and regularity are desirable, including consistent use of terms, units, formats, layouts, fonts, colors, sizes, and all the other options that a graphical system makes available. How many different English words are used to exit from a program or close a window? The choices range from Abandon to control-Z, with at least a dozen between. This inconsistency is confusing to a native speaker and baffling for others.

Within graphics code, interfaces are particularly important, since these systems are large, complicated, and driven by a very different input model than scanning sequential text. Object-oriented programming excels at graphical user interfaces, since it provides a way to encapsulate all the state and behaviors of windows, using inheritance to combine similarities in base classes while separating differences in derived classes.

Supplementary Reading

Although a few of its technical details are now dated, *The Mythical Man Month*, by Frederick P. Brooks, Jr. (Addison-Wesley, 1975; Anniversary Edition 1995), is delightful reading and contains insights about software development that are as valuable today as when it was originally published.

Almost every book on programming has something useful to say about interface design. One practical book based on hard-won experience is *Large-Scale C++ Software Design* by John Lakos (Addison-Wesley, 1996), which discusses how to build and manage truly large C++ programs. David Hanson's *C Interfaces and Implementations* (Addison-Wesley, 1997) is a good treatment for C programs.

Steve McConnell's *Rapid Development* (Microsoft Press, 1996) is an excellent description of how to build software in teams, with an emphasis on the role of prototyping.

There are several interesting books on the design of graphical user interfaces, with a variety of different perspectives. We suggest *Designing Visual Interfaces: Communication Oriented Techniques* by Kevin Mullet and Darrell Sano (Prentice Hall, 1995), *Designing the User Interface: Strategies for Effective Human-Computer Interaction* by Ben Shneiderman (3rd edition, Addison-Wesley, 1997), *About Face: The Essentials of User Interface Design* by Alan Cooper (IDG, 1995), and *User Interface Design* by Harold Thimbleby (Addison-Wesley, 1990).