Interfaces*

• Objective: To discuss the considerations that must be addressed when designing an interface. To illustrate these design issues with a simple yet useful example.

• “The essence of design is to balance competing goals and constraints. Although there are 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.”

*The examples in these slides come from Brian W. Kernighan and Rob Pike, “The Practice of Programming”, Addison-Wesley, 1999.
Themes

• Interfaces: what services and access are provided?
  – The interface is in effect a contract between the supplier and the 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.
Themes

• Information Hiding: What information is visible and what is private?
  – An interface must provide straightforward access to the components while hiding the details of the implementation so they can be changed without affecting users.
Themes

• Resource Management: who is responsible for managing memory and other resources?
  – Here, the main problems are allocating and freeing storage, and managing shared copies of information.
Themes

• Error handling: who detects errors, who reports them, and how? When an error is detected, what recovery is attempted?
Topics

- CSV - Comma separated values
- CSV - Prototype library
- CSV - A library for others
- CSV - A C++ implementation
Topics

• Interface Principles
  1. Hide implementation details
  2. Choose a small orthogonal set of primitives
  3. Don't reach behind the user's back
  4. Do the same thing the same way everywhere

• Resource Management
  1. Free a resource in the same layer that allocated it

• Abort, Retry, Fail?
  1. Detect errors at a low level, handle them at a high level
  2. Use exceptions only for exceptional situations
Comma Separated Values (CSV)

• A natural and widely used format for tabular data
  – Each row is a line of text
  – The fields on each line are separated by commas
  – Text format used by many spreadsheets

• Example
  – “Good Student”,CS265,”Advanced Programming Tools and Techniques”,A
  – “Bad Student”,CS265,”Advanced Programming Tools and Techniques”,D
Prototype Library

• “plan to throw one away; you will, anyhow” – Frederick Brooks
• “It is 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.”
• First version
  – ignore many of the difficulties
  – complete enough to be useful and to gain some familiarity with the problem
Prototype

```
char buf[200];  /* input line buffer */
char *field[20]; /* fields */

/* csvgetline: read and parse line, return field count */
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;
}
```
Prototype

/* 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;
}

/* 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;
}
Decisions Made in Prototype

- Doesn’t handle long input lines or lots of fields
- Lines terminated by newlines
- Fields are separated by commas and surrounding quotes removed – no embedded quotes
- Input line not preserved – overwritten when creating fields
- No data saved from one input line to the next
- Access to fields through global variable (does not prevent access beyond last field)
Decisions Made in Prototype

- Global variables make code unsuitable to multi-threaded environment or interleaved calls
- Caller must open and close files
- Input and splitting are inextricably linked
- The return value is the number of fields (each line must be split to compute this value)
- Each decision is interwoven into the code. There is no way to change any of these properties without changing the code
Specification

• Fields are separated by commas
• A field may be enclosed in double-quotes
• A quoted field may contain commas but not newlines
• A quoted field may contain double-quotes, represented by “”
• Fields may be empty; “” and empty string both represent an empty field
• Leading and trailing white space is preserved
Specification

- `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
Specification

- **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.
  - fields 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.
Specification

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

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 */
Internal Variables

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 */
Internal Functions

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

/* reset: set variables back to starting values */
static void reset(void)

/* split: split line into fields */
static int split(void)

/* advquoted: quoted field; return pointer to next separator */
static char *advquoted(char *p)
Example

- ab,"cd","e""f",,"g,h"
  - line = `ab,"cd","e""f",,"g,h"
  - field[0] = `ab'
  - field[1] = `cd'
  - field[2] = `e"f'
  - field[3] = `'
  - field[4] = `g,h'

```
| a | b | 0 | c | d | 0 | 0 | e | f | 0 | 0 | 0 | 0 | g | h | 0 |
```

```plaintext
1 2 3 4 5
```
Main Program

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#include "csv.h"

/* 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;
}
/* csvgetline: get one line, grow as needed */
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;
        }
        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;
/ * 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;
}
endofline

/* 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;
}
split

/* 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;
}
advquoted

/docs/advquoted/advquoted.c

/* 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;
}
/* 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;
}
C++ Version

• Class CSV
• Use C++ strings (handles storage management issues)
• Use C++ vectors to store fields
• Use class mechanism for information hiding
Public Interface

class Csv {  // read and parse comma-separated values

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

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

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);
};
/ **Main Program**

```c
// 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;
}
```