FILE I/O

CS 23200

Big Picture

- ☑ Developing programs on *nix computers
- □ C Language
  - ☑ Familiar aspects of C (variables, operators, basic I/O, control flow, functions)
  - ☑ Pointers
  - ☑ Structures and related constructs
  - □ File operations
  - □ Standard library functions
  - □ Multi-file programs
- □ *nix tools

Outline

- Manipulating I/O from the shell
  - Redirection
  - Piping
- C functions for I/O from standard in/out
- C functions for general file I/O
  - Formatted I/O
  - Unformatted text I/O
  - Binary files
  - File positioning

Standard In, Out, Error

- Standard input
  - Typically the stream of keyboard input from the terminal
- Standard output
  - Programs often print information to the standard output stream (using printf, e.g.)
  - Typically displayed to the terminal
- Standard error
  - Programs print error messages to the standard error stream
  - Also typically displayed to the terminal

Input Redirection

- What if we want to automate standard input?
  - Putting theBomb passwords in a text file, e.g.
- Use the input redirection operator (<) in the shell

```
[shell prompt $]  ./myProgram <inputFile.txt
```

- Will take the contents of inputFile.txt as if they were entered on the keyboard

Input Redirection

```
[shell $]  ./myProgram <inputFile.txt
```

- Note that myProgram doesn't know the redirection has happened
  - <inputFile.txt is not a command line argument
  - The <inputFile.txt is processed by the shell before the program starts

Output Redirection

- Instead of displaying output on the screen, can redirect it to a file

```
[shell $]  ./myProgram >consoleOutput.txt
```

- To redirect standard error as well:

```
[shell $]  ./myProgram >consoleOutput.txt  2>&1
```

sends stdout to consoleOutput.txt

redirects stderr (2) into stdout (1)

- ">>" append the results to the file

Piping

- Several *nix tools do processing on standard input and write results to standard output
- Can use the output of one program as the input of another program using piping

```
[shell $]  ./prog1 | ./prog2
```

  - Standard output of prog1 is the standard input of prog2
- Example: checking your primes project

```
[shell $] ./primes 10 20 | ./checkPrimes 10 20
```

tee

- tee is a *nix program that lets you capture output to a file and print it to the screen
- Usage:

```
./myProg arg1 arg2 | tee consoleOutput.txt
```

- tee takes the standard input stream and copies it to a file and to the screen

Outline

- Manipulating I/O from the shell
  - Redirection
  - Piping
- **C functions for I/O from standard in/out**
- C functions for general file I/O
  - Formatted I/O
  - Unformatted text I/O
  - Binary files
  - File positioning

Single-character I/O

- Input: `int getchar(void);`
  - Returns the next unprocessed character from standard input
  - Returning int permits returning EOF, which indicates end-of-file

```
int c;
c = getchar();
if(c != EOF){
  ...
```

Single-character I/O

- Output: `int putchar(int c);`
  - Writes c to standard output
  - Normally, returns c
  - On error, returns EOF

```
const char *str = "A string to write \n";
const int n = strlen(str);
int i;

for(i=0; i<n; i++){
  if(putchar(str[i]) == EOF){
    /* do error handling */
    ...
  }
}
```

Exercise: Single-character I/O

Exercise 7-1 (adapted) from K&R:

- Write a program that takes one command-line argument: "L" or "U"
  - If "L" the program will convert to lower case
  - If "U" the program will convert to upper case
- The program should read from standard input, doing the case conversion before echoing back to standard output
- Use "tolower(c)" and "toupper(c)"

Solution: Single-character I/O

```
#include <stdio.h>

int main(int argc, char** argv){
   int c, goLower;
   if(argc < 2){
      fprintf(stderr, "USAGE: %s [lower|upper]\n", argv[0]);
      return -1;
   }
   goLower = !strcmp("lower", argv[1]);
   c = getchar();
   while( c != EOF ){
      if (goLower)
         putchar(tolower(c));
      else
         putchar(toupper(c));
      c = getchar();
   }
   return 0;
}
```

Good ol' printf

- Use printf to print to standard output
- printf takes a format string and some arguments

```
printf("%s is a string.\n"
       "%d is an int.\n"
       "%f is a double.\n",
       "\"This\"", -78, 54.2);
```

No commas; these string literals are concatenated by the compiler

Good ol' printf

- Format strings are often string literals

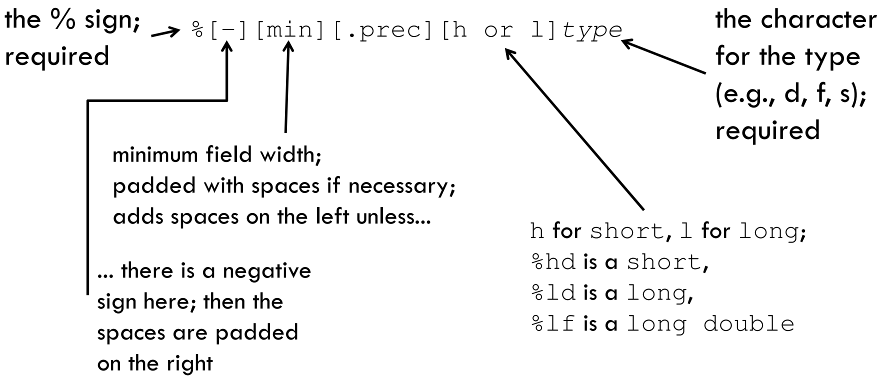
```
double x = -5.4;
printf("%.12f", x);
```

- But they can also be string variables

```
const char *dblFormat = "%.12f";
double x;
...
printf(dblFormat, x);
```

Advanced printf

- □ printf format specifiers are very flexible, allowing lots of output formats



the % sign; required

`%[-][min][.prec][h or l]type`

the character for the type (e.g., d, f, s); required

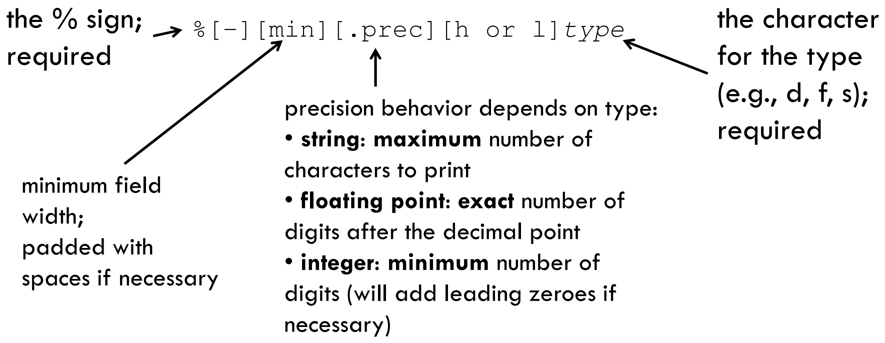
minimum field width; padded with spaces if necessary; adds spaces on the left unless...

... there is a negative sign here; then the spaces are padded on the right

`h` for `short`, `l` for `long`; `%hd` is a `short`, `%ld` is a `long`, `%lf` is a `long double`

Advanced printf

- □ printf format specifiers are very flexible, allowing lots of output formats



the % sign; required

`%[-][min][.prec][h or l]type`

the character for the type (e.g., d, f, s); required

minimum field width; padded with spaces if necessary

precision behavior depends on type:
- **string: maximum** number of characters to print
- **floating point: exact** number of digits after the decimal point
- **integer: minimum** number of digits (will add leading zeroes if necessary)

Fixed-width Fields

- □ Fixed-width fields are a common goal

```
Jane      Doe             12       3.864
George    Smith         7349     278.100
```

not

```
Jane Doe 12 3.864
George Smith 7349 278.1
```

Fixed-width Fields

```
Jane      Doe             12     3.864
George    Smith         7349   278.100
```

- □ Potential problem 1: data is too narrow
  - Solution: use minimum width to pad with spaces

```
printf("%-8s %-8s %8d %8.3f\n", ...);
```

Why the negative signs?
Why the .3 for the floating point?

Fixed-width Fields

```
Jane     Doe            12    3.864
George   Smith        7349  278.100
Rumpelstiltskin Jones    1816473825 37857.357
Alexander Johnson       139 574839.320
```

- Potential problem 2: data is too wide
  - Solution depends on type
  - String: use precision to specify maximum width

```
printf("%-8.8s %-8.8s %8d %8.3f\n", ...);
```

```
Rumpelst Jones    1816473825 37857.357
Alexande Johnson       139 574839.320
```

Fixed-width Fields

```
printf("%-8.8s %-8.8s %8d %8.3f\n", ...);
```

```
Rumpelst Jones    1816473825 37857.357
Alexande Johnson       139 574839.320
```

- Potential problem 2: data is too wide
  - No way to specify maximum width for numerical types
  - Must set the field width based on the maximum length value you will be printing
  - Not very robust
  - Should check values before printf if formatting is crucial

```
if(num >= 100000000 || num <= -10000000){
   /* num is too wide for an 8-character field */
   /* print an error message */
   ...
```

printf exercise

```
struct scoreInfo{
   char* name;
   int id;
   double score;
};
```

```
...
const int n = 200;
struct scoreInfo scores[n];
...
/* scores is initialized */
...
```

- Write code to print the data in `scores`
  - Use one line for each item in the array
  - Use alignment so the data for a given field always starts at the same offset

printf exercise: solution

```
struct scoreInfo{
   char* name;
   int id;
   double score;
};
```

```
...
const int n = 200;
struct scoreInfo scores[n];
...
/* scores is initialized */
...
```

```
int i;
for(i=0; i<n; i++){
   printf("%9d %-20.20s %8.3f\n",
            scores[i].id, scores[i].name,
            scores[i].score);
}
```

Formatted Input

- scanf is the input counterpart to printf
  - printf prints output to standard out
  - scanf grabs input from standard in

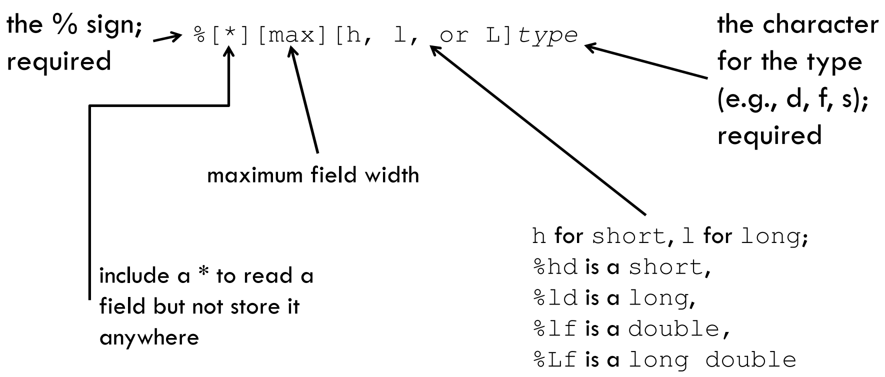
```
int intOne, intTwo;
double dblOne;

printf("Enter two integers and a floating point:\n");
scanf("%d %d %lf", &intOne, &intTwo, &dblOne);
```

use %lf for double,
%f for float

pass **addresses** for storing the input

scanf Format Specifiers

- scanf format specifiers are slightly different from printf's



scanf and Whitespace

- Blanks or tabs in format string tell scanf to gobble up as much white space (newlines, spaces, tabs, etc.) as it can

```
scanf("%d              %d", ...);
/* same as */
scanf("%d %d", ...);
```

- scanf will gobble up white space before parsing a field

```
/*  above examples are the same as */
scanf("%d%d", ...);
```

  - Use %c specifier or getchar() to get whitespace characters

- scanf("%d %d", ...) will correctly process...
  - 234   789
  - 234 789
  - 234
    789

scanf

- %s reads in characters until a whitespace

- Ordinary characters (not whitespace or format specifiers) should match the input

```
char name[9];
scanf("name: %8s", name);
```

succeeds on:

```
name: John
```

fails on:

```
Name: John
```

scanf

- NEVER use %s in scanf without specifying a maximum width
  - This would allow the user to crash your program (or execute their own code) by specifying a long string
- scanf returns the number of successfully parsed fields
  - Stops processing input after the first failure
- Remember: scanf arguments should be pointers (i.e., addresses)

scanf exercise

- A user enters 3 integers per line for n lines
- Compute and print the column sums
  - e.g., the sum of the first integer on each line is the first column sum

scanf exercise solution

- Remember: pass ADDRESSES to scanf

```
const int numCols = 3;
int sums[3] = {0,0,0};
int lineNum, colNum;

for(lineNum=0; lineNum<n; lineNum++){
   int vals[numCols];
   scanf("%d %d %d%*c", vals, vals+1, vals+2);
   for(colNum=0; colNum<numCols; colNum++){
      sums[colNum] += vals[colNum];
   }
}
printf("%d %d %d\n", sums[0], sums[1], sums[2]);
```

http://stackoverflow.com/questions/3744776/simple-c-scanf-does-not-work

Outline

- Manipulating I/O from the shell
  - Redirection
  - Piping
- C functions for I/O from standard in/out
- **C functions for general file I/O**
  - **Formatted I/O**
  - **Unformatted text I/O**
  - **Binary files**
  - **File positioning**

General File I/O

- General pattern
  - Open a file
    - Returns a FILE*
    - FILE* identifies the file in subsequent operations (read, write, close)
    - "Opening" can include creating the file if it doesn't exist
  - Read/write from the file
  - Close the file

fopen

- FILE* fopen(char *filename, char *mode);
  - filename can be a relative or absolute path
  - mode
    - "r" for read
    - "w" for write
    - "a" for append
    - "r+" for read and write
    - BE CAREFUL: "w" will create a blank file with the given name, overwriting an existing file
  - Returns NULL on error
    - ALWAYS check for NULL
    - More on error handling later

fclose

- int fclose(FILE *stream);
  - Closes the file indicated by stream
  - Returns 0 upon success, EOF upon error

Reading and Writing

- Different sets of functions for reading and writing
- One set: fprintf and fscanf
  - Work like printf and scanf
  - One additional argument: first argument is the FILE*

- printf and scanf are actually special cases of fprintf and fscanf
  - Standard input, standard output, and standard error each have global FILE* variables
  - `stdin, stdout, stderr`

fprintf Example

```
int main(int argc, char* argv[]){
   FILE *fp;
   if(argc < 2){
      fprintf(stderr, "Missing file name.\n");
      return -1;
   }
   fp = fopen(argv[1], "w");
   if(fp == NULL){
      fprintf(stderr, "Could not create file (%s).\n",
                  argv[1]);
      return -2;
   }
   fprintf(fp, "This is a number: %d\n", 17);
   fprintf(fp, "Another number: %8.4f\n", 3.141592);
   fclose(fp);
   return 0;
}
```

fscanf Example

- Let "matrix.txt" be a file with the following format:

```
number of rows
number of columns
data, one line for each row


2
3
3.42 -47.0 9.3
0.0 8.38 -374.2
```

```
const char* filename = "matrix.txt";
int numRows, numCols, res, row, col;
double **matrix;
FILE *fp = fopen(filename, "r");
if(fp == NULL){
   fprintf(stderr, "Could not open file (%s).\n", filename);
   return -1;
}
res = fscanf(fp, "%d", &numRows);
if(res != 1){
   fprintf(stderr, "File format error.\n");
   return -2;
}
res = fscanf(fp, "%d", &numCols);
... /* check res and create matrix */
for(row=0; row<numRows; row++){
   for(col=0; col<numCols; col++){
      fscanf(fp, "%lf", &(matrix[row][col]));
   }
}
fclose(fp);
```

Exercises

- Two programs:
  - Write your birthday to a file "birthday.txt"
  - Read your birthday from the file you just wrote
    - Should have month, date, year variables

A Solution for Writing

```
int main(){
   const char *filename = "birthday.txt";
   FILE *fp = fopen(filename, "w");

   if(fp == NULL){
      fprintf(stderr, "Could not open file (%s)\n",
            filename);
      return -1;
   }

   fprintf(fp, "%d/%d/%d\n", 3, 28, 1928);
   fclose(fp);

   return 0;
}
```

A Solution for Reading

```
int main(){
   const char *filename = "birthday.txt";
   FILE *fp = fopen(filename, "r");
   int month, date, year, res;
   if(fp == NULL){
      fprintf(stderr, "Could not open file (%s)\n",
            filename);
      return -1;
   }
   res = fscanf(fp, "%d/%d/%d\n", &month, &date, &year);
   if(res != 3){
      fprintf(stderr, "File format error.\n");
      return -2;
   }
   fclose(fp);
   return 0;
}
```

Another Example

- □ Calculate the average of students’ scores
- □ Scores are stored in a text file
- □ Format in the file:

  student_name score

- □ get_average.c

Landscape of I/O Functions

	Formatted	Unformatted	Binary
Input	**fscanf** (scanf is a special case)	single char: **fgetc** whole line: **fgets**	**fread**
Output	**fprintf** (printf is a special case)	single char: **fputc** more: **fputs**	**fwrite**

Unformatted Text I/O

- Single characters
  - int fgetc(FILE *stream);
    - Returns next character from stream (EOF if end of stream)
  - int fputc(int c, FILE *stream);
    - Writes c to stream
    - Returns the character written, or EOF for error
- Peeking
  - Might want to stop processing if the next character satisfies some condition
  - fgetc to look at the character, then ungetc to put it back
  - int ungetc(int c, FILE *stream);
  - Only guaranteed for one character

Unformatted Text I/O

- More than single characters
  - char* fgets(char *s, int n, FILE *stream);
    - Reads characters into s until...
      - A '\n' is reached
      - or n-1 characters have been read
    - s will always be null-terminated
    - s might contain a newline '\n'
    - Returns s, or NULL if end-of-file error occurs
  - int fputs(const char *s, FILE *stream);
    - Writes s to stream
    - s need not contain a newline
    - Returns EOF for error

A Common File Input Pattern

- Loop over lines while fgets does not return NULL
  - Process the line that fgets just read
    - Often use sscanf to extract int's or double's

```
int sscanf(const char* source, const char* format,
                ...);
```

- Works like fscanf
- Processes the string `source` (instead of input from a file)

The Reasoning

- Why use fgets and sscanf instead of using fscanf directly?
  - With fgets, an erroneous line is isolated from other lines
  - The whitespace issue
    - Expecting 3 int's per line: `fscanf(fp, "%d %d %d\n", ...);`
    - If one line is missing an int, this will grab the first int from the next line as the third int
  - Can look through the line multiple times if needed
- Disadvantage: lose the automatic position advancement within the line
  - Will discuss how to deal with this later

Example

- Get users inputs for a list of integers
- Calculate the sum and print it
- Keep looping until user type the empty string
- Number of integers can be varying

- calculate_sum.c

An Exercise

- The file "data.txt" contains the following information:
  - First line: the number of rows that follow
  - Each subsequent line: a list of whitespace-delimited integers
    - Each list can be a different length
- Example: each line contains Twitter followers (integer ids) for one person
- Read in the data and store each row in its own array of int's (for later processing)

An Exercise

- First line: the number of rows that follow
- Other lines: lists of integers
    - Each list can be a different length
- Read in the data and store each row in an array of int's (for later processing)

- **The process for approaching any problem:**
  - What are the big steps?
    - Then break each of those down into smaller steps
  - What data structures will I need for those steps?
  - Often, the first answers to these questions will need to be revised as you work out the details

The Steps

- Open the file and verify the file pointer
- Read the number of lines
- Create an array with one element for each line
- Loop over the lines
  - Read in the line
  - Determine how many int's there are
  - Allocate space for those int's
  - Read the int's into that space
- Close the file

We don't currently have the tools to do these, but we can still write the other code

The Data Structure

- □ What information will we need from the file?

```
struct arrayOfVectors{
  int numRows;

  /* an array of size numRows */
  int *rowSizes;

  /* an array of size numRows;
     theData[i] points to an array
     of int's of size rowSizes[i] */
  int **theData;
};
```

```
struct arrayOfVectors* readData(const char* filename){
  struct arrayOfVectors* pResult = NULL;
  int lineIndex, res;
  const int bufSize = 10000;
  char line[bufSize];
  FILE *fp = NULL;

  pResult = malloc(sizeof(struct arrayOfVectors));
  if(pResult == NULL){
    fprintf(stderr, "Could not allocate memory.\n");
    return NULL;
  }

  fp = fopen(filename, "r");
  if(fp == NULL){
    fprintf(stderr, "Could not open the file (%s).\n",
       filename);
    free(pResult);
    return NULL;
  }
```

```
res = fscanf(fp, "%d", &pResult->numRows);
if(res != 1 || pResult->numRows > 1000000 ||
        pResult->numRows <= 0){
  fprintf(stderr, "Invalid number of lines.\n");
  fclose(fp);
  free(pResult);
  return NULL;
}
pResult->theData = malloc(sizeof(int*) * pResult->numRows);
if(pResult->theData == NULL){
  fprintf(stderr, "Could not allocate memory.\n");
  fclose(fp);
  free(pResult);
  return NULL;
}
pResult->rowSizes = malloc(sizeof(int) * pResult->numRows);
if(pResult->rowSizes == NULL) {
  fprintf(stderr, "Could not allocate memory.\n");
  free(pResult->theData);
  fclose(fp);
  free(pResult);
  return NULL;
}
```

Noticing a Pattern?

- □ One line of processing
- □ Check for errors
  - ◘ Error handling
- □ Next line of processing
- □ Check for errors
  - ◘ Error handling
- □ ...

- □ While this can get tedious, it's the only way to ensure secure, robust code

```
struct arrayOfVectors* readData(const char* filename){
  struct arrayOfVectors* pResult = NULL;
  int lineIndex, res;
  const int bufSize = 10000;
  char line[bufSize];
  FILE *fp = NULL;
   ...
  for(lineIndex=0; lineIndex<pResult->numRows; lineIndex++){
    fgets(line, bufSize, fp);

    /* by using fgets, we have a string to pass
       to createIntArray, which can manipulate it
       as necessary (in contrast to using fscanf) */
    pResult->theData[lineIndex] =
      createIntArray(line, pResult->rowSizes + lineIndex);
  }

  fclose(fp);
  return pResult;
}
```

Landscape of I/O Functions

	Formatted	Unformatted	Binary
Input	**fscanf** (scanf is a special case)	single char: **fgetc** whole line: **fgets**	**fread**
Output	**fprintf** (printf is a special case)	single char: **fputc** more: **fputs**	**fwrite**

Binary File I/O

- Using binary file I/O will make your code system-dependent
  - Might reduce file size

```
size_t fread(void *ptr, size_t size,
                  size_t nobj, FILE *stream);
size_t fwrite(const void *ptr, size_t size,
                  size_t nobj, FILE *stream);
```

- Returns number of objects read/written

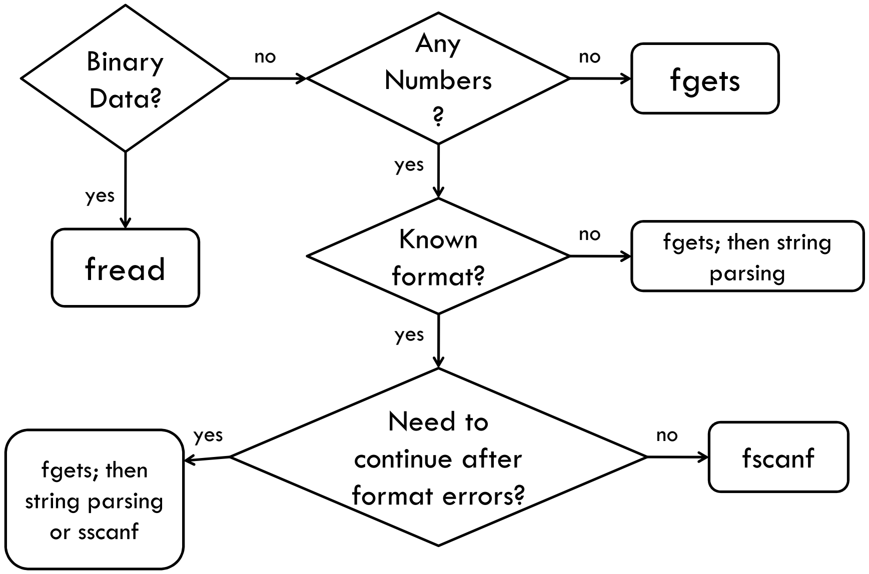
File Positioning

- The functions so far start at the beginning of the file and read/write in order
- What if you need to go back or skip ahead?
- int fseek(FILE *stream, long offset, int origin);
  - Moves the file position
  - origin:
    - SEEK_SET : beginning of file
    - SEEK_CUR : current position
    - SEEK_END : end of file
  - Returns non-zero on error

Summary

- □ Manipulating I/O from the shell
  - Redirection
  - Piping
- □ C functions for I/O from standard in/out
- □ C functions for general file I/O
  - Formatted I/O
  - Unformatted text I/O
  - Binary files
  - File positioning

Choosing an Input Method



Binary Data?

no

yes

fread

Any Numbers ?

no

fgets

yes

Known format?

no

fgets; then string parsing

yes

Need to continue after format errors?

yes

fgets; then string parsing or sscanf

no

fscanf