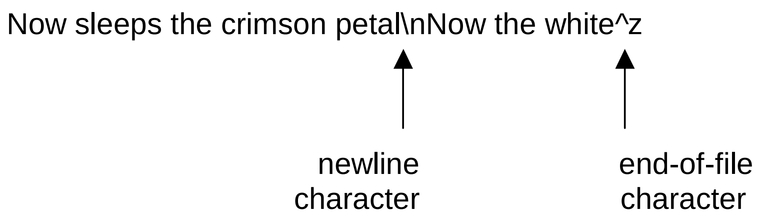
9 Text Files

9.1 Introduction

In chapters seven and eight we looked at files with fixed-length records. Now we look at files with variable-length records. The problem with variable-length records is knowing where one record ends and the next one begins. One method is to use a special character to mark the end of a record. We see how this might be done with a file of text.

A file of text is a sequence of characters structured into lines. A line is terminated with either a newline character or the end of file character. For example



The end-of-file character is usually ^z in MSDOS systems and ^d on Unix systems, where ^ represents the control (Ctrl) key.

Program 9.1 shown below retrieves lines of text from a file (such as a C program) and writes them out on the printer.

```
/* program 9.1 - prints text files. */

#include <stdio.h>
#include <stdlib.h>
#define printerDevice "LPT1"

int main()
{
  char diskFileName[BUFSIZ], line[BUFSIZ];
  FILE *text;
  FILE *printer = fopen(printerDevice, "w");

  printf("Name of file to print? ");
  gets(diskFileName);
  text = fopen(diskFileName, "r");

  if (text == NULL) {
    printf("Program halted: unable to find %s to print.\n",
              diskFileName);
    exit(EXIT_FAILURE);
  }
  while (fgets(line, BUFSIZ, text) != NULL)
    fprintf(printer, "%s", line);
  fclose(text);
  fclose(printer);
  return 0;
}
```

The line

```
fgets(line, BUFSIZ, text);
```

says retrieve characters from the file named *text* and place them in the variable *line* until either *BUFSIZ - 1* characters have been read or the newline character has been retrieved or the end of the file *text* has been reached; then append NULL to the character sequence stored in *line*. The newline and end-of-file characters retrieved from the file are discarded. *fgets* returns NULL if the end of the file is reached. So

```
while (fgets(line, BUFSIZ, text) != NULL)
```

makes repeated calls to *fgets* until the end of the file is reached. *fgets* works just like *gets* does, except that *gets* obtains text from the keyboard while *fgets* obtains text from a named file.

fprintf works just like *printf* does, except that the output is to the named file (*text*, in this example) and not the screen.

9.2 Command Line Arguments

When program 9.1 is run, it requests the user to input the name of the file to be printed. What we would like to do is supply the name of the file to be printed at the time we run the file print program. For example, suppose the file print program was named *pprint* (for pretty print) and the file to be printed was named *prog91.c*, then we would like to issue the command

pprint prog91.c

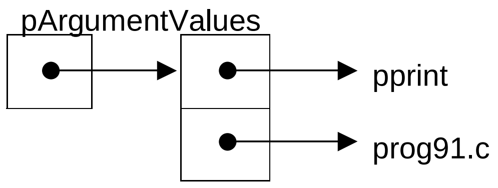
at the operating system prompt. We would like the string *prog91.c* to be input into the *pprint* program. The following program shows how this may be done.

```
/* pprint - prints text files, uses command line arguments. */

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

#define printerDevice "LPT1"

void printTextFile(char diskFileName[]);
/* Writes contents of diskFileName in printer. */
```

```
int main(int argumentCount, char *pArgumentValues[])
{
  if (argumentCount != 2) {
    printf("Usage: pprint filename.ext\n");
    exit(EXIT_FAILURE);
  }
  printTextFile(pArgumentValues[1]);
  return 0;
}

void printTextFile(char diskFileName[])
{
  char line[BUFSIZ];
  FILE *printer = fopen(printerDevice, "w");
  FILE *text = fopen(diskFileName, "r");
  if (text == NULL) {
    printf("Program halted: unable to find %s\n",
                diskFileName);
    exit(EXIT_FAILURE);
  }
  while (fgets(line, BUFSIZ, text) != NULL)
    fprintf(printer, "%s", line);
  fclose(text);
  fclose(printer);
}
```

The essential point is the way main is written. Here, main has two parameters, *argumentCount* and **pArgumentValues[]*.

pArgumentValues points to an array of pointers to strings. For example, if at the operating system prompt we entered *pprint prog91.c*, then *pArgumentValues* contains the address of an array of pointers, where each pointer contains the address of a string entered.



pArgumentValues[0] points to *pprint*. *pArgumentValues[1]* points to *prog91.c*. These strings can be used within a program and are known as program parameters.

argumentCount contains the number of strings in the array to which *pArgumentValues* points. If the user uses *pprint* incorrectly, by not supplying it with the name of a file to be printed for example, then *argumentCount* does not contain the value 2 and the user is informed how to use *pprint* correctly.

Exercise 9.1

1 Write a program which will print out program text files. There should be a margin of about one inch all round the printed text on each page. If a line of text is too long to fit across the page, then the line should be truncated and two exclamation marks should be printed. Each page should have its own title and page number. You could use program pprint as a starting point. Note: if you use an integrated programming environment, then you might need to exit from the environment to run the program.

9.3 File Handling Program Structure

Every file is managed by the same few processes: open, close, retrieve, write and error handling. If we confine, to a single function, all the logic necessary to open a file, retrieve data from it and then close it, then we can use the same function with minimal modification to retrieve data from any other file. Let us see how this might be done.

We can think of the file as being in one of three states: either unopened (that is, never been opened in the current program run) or open (ready for retrieving records) or closed (because the end of the file has been reached).

If the file is unopened, then we want to open it and set its state to open.

If the file is open, then we want to retrieve a record. If the retrieval fails, because the end of the file has been reached for example, then we want to close the file and set its state to closed.

A file, once closed, cannot be opened again in the current program run. Here is how we might write the function.

```
void retrieveLineFromTextFile(char diskFileName[],
                            char lineOfText[], int *pFinished)
{
  enum { false, true };
  typedef enum { unopened, open, closed } FileState;

  static FILE *file;
  static FileState fileState = unopened;

  *pFinished = false;
  switch (fileState) {
  case unopened:
    file = fopen(diskFileName, "r");
    if (file == NULL) {
      printf("Unable to open %s\n", diskFileName);
      exit(EXIT_FAILURE);
    }
    else
      fileState = open;  /* fall through */
  case open:
    if (fgets(lineOfText, BUFSIZ, file) == NULL) {
      fclose(file);
      *pFinished = true;
      fileState = closed;
    }
    break;
  case closed:
    break;
  default:
    printf("retrieveLineFromTextFile: this should not
               happen.\n");
    exit(EXIT_FAILURE);
  }
}
```

When the function is called for the first time, the file state is *unopened*. An attempt is made to open the file. If successful then the file state is *open* and we fall through to the next case.

We attempt to retrieve a record with *fgets*. Let us suppose we are successful. We return to the caller.

On the next call to the function, the value stored in *fileState* is still open. This is the result of declaring the type *FileState* as *static*. Variables whose storage class is defined as *static* remain in existence from one function call to the next. Without the static storage class specifier, the variable *fileState* would be re-created and re-initialised with *unopened* every time the function is called.

So the file state is *open* and we attempt to retrieve a record. If we fail, because the end of the file has been reached for example, then we close the file, set **pFinished* to *true* and the file state to *closed*. We need to communicate to other functions the fact that the end of the file has been reached; this is the purpose of the *int *pFinished* parameter.
If we make a further call to *retrieveLineFromTextFile*, the file state is *closed* and no further action is specified.

The function to manage the printer works in a similar way. It is shown below in program 9.3.

```
/* program 9.3 - prints text files, uses functions. */
#include <stdio.h>
#include <stdlib.h>

void retrieveLineFromTextFile(char diskFileName[],
                           char lineOfText[], int *pFinished);
/* Post-condition: *pFinished' = true if the end of the file
                   diskFilename is reached, otherwise
                   lineOfText' contains a line of text. */

void writeLineToPrinter(char lineOfText[], int finished);
/* Writes lineOfText to the printer if finished is not true.*/

int main()
{
  char line[BUFSIZ], diskFileName[BUFSIZ];
  int finished;
  printf("Name of text file to print? ");
  gets(diskFileName);
  retrieveLineFromTextFile(diskFileName, line, &finished);
  while (!finished) {
    writeLineToPrinter(line, finished);
    retrieveLineFromTextFile(diskFileName, line, &finished);
  }
  return 0;
}
```

```
void retrieveLineFromTextFile(char diskFileName[],
                              char lineOfText[], int *pFinished)
{
  enum { false, true };
  typedef enum { unopened, open, closed } FileState;
  static FILE *file;
  static FileState fileState = unopened;

  *pFinished = false;

  switch (fileState) {
  case unopened:
    file = fopen(diskFileName, "r");
    if (file == NULL) {
      printf("Unable to open %s\n", diskFileName);
      exit(EXIT_FAILURE);
    }
    else
      fileState = open;  /* fall through */
  case open:
    if (fgets(lineOfText, BUFSIZ, file) == NULL) {
      fclose(file);
      *pFinished = true;
      fileState = closed;
    }
    break;
  case closed:
    break;
  default:
    printf("retrieveLineFromTextFile: this should not
               happen.\n");
    exit(EXIT_FAILURE);
  }
}

void writeLineToPrinter(char lineOfText[], int finished)
{
  typedef enum { unopened, open, closed } FileState;
  static FILE *printer;
  static FileState fileState = unopened;
  switch (fileState) {
  case unopened:
    printer = fopen("LPT1", "w");
    if (printer == NULL) {
      printf("Unable to open printer.\n");
      exit(EXIT_FAILURE);
    }
    else
      fileState = open;  /* fall through */
  case open:
    if (!finished)
      fprintf(printer, "%s", lineOfText);
    else {
      fclose(printer);
      fileState = closed;
    }
    break;
  case closed:
    break;
  default:
    printf("writeLineToPrinter: this should not happen.\n");
    exit(EXIT_FAILURE);
    break;
  }
}
```

writeLineToPrinter needs to know when to close the printer file. This is why *finished* is a parameter to the function. Notice the role of *finished* in *main*.

Notice also that *main* has nothing to do with opening files and then closing them. This is all taken care of in the retrieve and write functions. Very neat.

9.4 Processing Text Files

It is sometimes convenient to create files using an ordinary text editor (such as the one you use to create or amend your programs). For example, suppose the following data on people who owe money was stored in a text file

```
147309 Cartwright £19999.99
247609 Jones £20000.00
965235 Stevenson £20000.01
```

Each line in the file has the format *referenceNumber name debt*. We can write programs which use this data. Suppose we want to create a file of debtors who owe more than, say, £19999.99. We need to isolate the debt from a line of text in order to decide whether the line should go in the new file. For example, we need to isolate 19999.99 from

```
147309 Cartwright £19999.99
```

We could write our own function from scratch to do the job. Or we could use the *strtok* function from the string library.

strtok divides a string into a set of words known as tokens. Its template is

```
char *strtok(char *string, const string *separators);
```

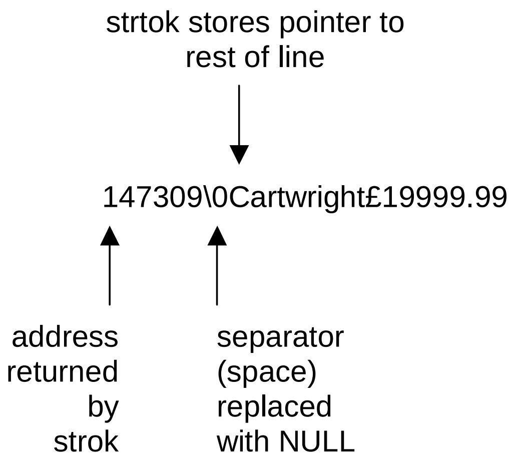
An example of a call to *strtok* is

```
char line[] = "147309 Cartwright £19999.99";
char wordSeparators[] = " ,£";
char *word;

word = strtok(line, wordSeparators);
```

wordSeparators is a string that contains the symbols which separate one word from the next. Here the explicit word separator characters are space, comma and the £ sign; the implicit word separator is the NULL end-of-string character.

word is a pointer to *char*; that is, it contains the start address of a sequence of characters held in memory.

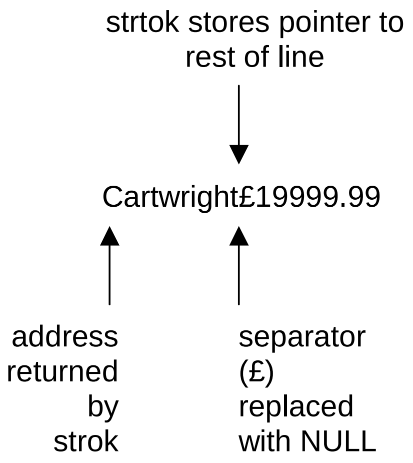
strtok looks through line for a character which is not in *wordSeparators*. If it finds one then it is the first character in the first word in line. Then *strtok* looks through line for a character which is in *wordSeparators*. If it finds one then it is replaced with the NULL character to mark the end of the first word. *strtok* stores a pointer to the rest of the line and returns the address of the first word.


strtok stores pointer to rest of line
147309\0Cartwright£19999.99
address returned by strok
separator (space) replaced with NULL

If we subsequently make the call

```
word = strtok(NULL, wordSeparators);
```

strtok looks through the rest of the line for a character which is in *wordSeparators*. If it finds one then it is replaced with the NULL character to mark the end of the next word. *strtok* stores a pointer to the rest of the line and returns the address of the next word.

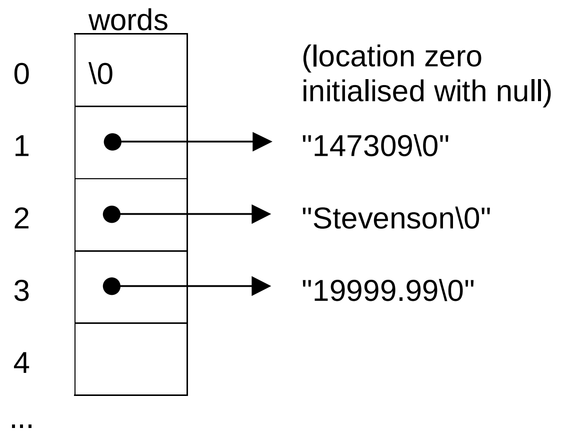

strtok stores pointer to rest of line
Cartwright£19999.99
address returned by strok
separator (£) replaced with NULL

If, on looking through *line strtok* cannot find a character which is in *wordSeparators,* then it returns NULL.

Since *strtok* modifies its first string argument value, we are obliged to provide a copy of *line* for *strtok* to work with. This is what we have done in the function *debtFromLine* shown below.

```
char* debtFromLine(char aLine[])
{
  int wordCount = 0;
  char wordSeparators[] = " ,£";
  char *words[BUFSIZ] = { NULL };
  char *line = (char*)malloc(strlen(aLine) + 1);
  char *word;

  strcpy(line, aLine);
  word = strtok(line, wordSeparators);
  while (word != NULL) {
    wordCount++;
    words[wordCount] = word;
    word = strtok(NULL, wordSeparators);
  }
  return words[3];
}
```

What we have done here is to store pointers to each word in line.



Program 9.4 creates a file of debtors who owe more than £19999.99 from the original file of debtors.

```
/* program 9.4 - processes debtors file to create a new file
                 of debtors who owe more than £19999.99 */

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

char* debtFromLine(char line[]);
/* Post-condition: returns debt from a line of text in */
/*                 debtor.dat.                         */
```

```
double doubleFromString(char string[]);
/* returns double number equivalent of string. */

void retrieveLineFromTextFile(char diskFileName[],
                              char lineOfText[], int *pFinished);
/* Post-condition: *pFinished' = true if the end of the file
                   diskFilename is reached, otherwise
                   lineOfText' contains a line of text. */

void writeLineToTextFile(char diskFileName[],
                         char lineOfText[], int finished);
/* Post-condition: lineOfText is written to file diskFileName
                   if finished = 0, otherwise if finished = 1,
                   closes diskFileName.   */

int main()
{
  char line[BUFSIZ];
  int finished;
  char* debt;
  char debtorsDiskFileName[] = "debtors.dat";
  char bigDebtorsDiskFileName[] = "bigdebts.dat";

  retrieveLineFromTextFile(debtorsDiskFileName, line,
             &finished);
  while (!finished) {
    debt = debtFromLine(line);
    if (doubleFromString(debt) >= 20000.00)
      writeLineToTextFile(bigDebtorsDiskFileName, line,
             finished);
    retrieveLineFromTextFile(debtorsDiskFileName, line,
             &finished);
  }
  return 0;
}

char* debtFromLine(char aLine[])
{
  int wordCount = 0;
  char wordSeparators[] = " ,£";
  char *words[BUFSIZ] = { NULL };
  char *line = (char*)malloc(strlen(aLine) + 1);
  char *word;

  strcpy(line, aLine);

  word = strtok(line, wordSeparators);
  while (word != NULL) {
    wordCount++;
    words[wordCount] = word;
    word = strtok(NULL, wordSeparators);
  }
  return words[3];
}
```

```
double doubleFromString(char string[])
{
  double number = 0.0;

  sscanf(string, "%lf", &number);
  return number;
}

void retrieveLineFromTextFile(char diskFileName[],
                            char lineOfText[], int *pFinished)
{
  enum { false, true };
  typedef enum { unopened, open, closed } FileState;

  static FILE *file;
  static FileState fileState = unopened;

  *pFinished = false;
  switch (fileState) {
  case unopened:
    file = fopen(diskFileName, "r");
    if (file == NULL) {
      printf("Unable to open %s\n", diskFileName);
      exit(EXIT_FAILURE);
    }
    else
      fileState = open;  /* fall through */
  case open:
    if (fgets(lineOfText, BUFSIZ, file) == NULL) {
      fclose(file);
      *pFinished = true;
      fileState = closed;
    }
    break;
  case closed:
    break;
  default:
    printf("retrieveLineFromTextFile: this should not
              happen.\n");
    exit(EXIT_FAILURE);
  }
}

void writeLineToTextFile(char diskFileName[],
                         char lineOfText[], int finished)
{
  enum { false, true };
  typedef enum { unopened, open, closed } FileState;

  static FILE *file;
  static FileState fileState = unopened;

  switch (fileState) {
  case unopened:
    file = fopen(diskFileName, "w");
    if (file == NULL) {
      printf("Unable to open %s\n", diskFileName);
      exit(EXIT_FAILURE);
    }
    else
      fileState = open;  /* fall through */
  case open:
    if (!finished)
      fprintf(file, "%s", lineOfText);
    else {
      fclose(file);
      fileState = closed;
    }
    break;
  case closed:
    break;
  default:
    printf("retrieveLineFromTextFile: this should not
               happen.\n");
    exit(EXIT_FAILURE);
  }
}
```

After running this program the file *bigdebts.dat* contains

```
247609 Jones £20000.00
965235 Stevenson £20000.01
```

9.6 Programming Principles

Place all the operations - open, close, input, output - which deal with one file - into one function. If a file handling error occurs, then the problem lies in the one function.

When designing and writing programs, focus on the data values involved. Consider providing functions which operate on these values, operations such as input, output and creation. Then use these functions to create other (higher level) functions or programs.

If you need to write a function, see if somebody has already done - if they have then use it. If you find a function which nearly does what you want, then adapt it. There is no need to re-invent what has already been invented.

Exercise 9.2

1 Word processors usually embed formatting commands in the text. Unfortunately, different word processors have different formatting commands; this means that files created with one word processor may not be amended with another word processor. Write a program which will copy any file created by a word processor into a text file without the formatting commands. (The text file may then be formatted by any other word processor.)