Faculty of Computer Science, Dalhousie University *26-Nov-2018*

CSCI 2132 — Software Development

Lecture 32: Dynamically Allocated Arrays

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Merge Sort with Linked List example finished
- Git and GitLab, comparison to Subversion (svn)
- File Manipulation in C
  - streams and files

Slide notes:

File Pointers

- In C, streams are accessed through *file pointers*; e.g.:
  `FILE *fp;`
- Defined in `stdio.h`
- When a file is open, a file pointer has address of a file descriptor structure
- Two types of files: text and binary
- Text files:
  - newline character may be treated specially
  - may have a special marker byte at the end
- Binary files: raw access

In C, streams are accessed through file pointers. These pointers are variables of type `FILE *`, and to use them we need to include `stdio.h`. Before we learn how to manipulate files, it is important to learn that there are two types of files: text files and binary files. The notion of lines applies to text files only. Thus a newline character is treated as a special character as it marks the end of a line. In a binary file, all bytes are treated equally. Another difference is how a value is stored. For example, we wish to store the integer 12345 in a file. If we store it in a text file, then it will be stored as five characters: '1', '2', '3', '4' and '5'. This requires 5 bytes. If we store it in a binary file, its binary representation will be stored in `sizeof(int)` bytes.

25.2 Opening and Closing a File

Slide notes:

Opening a File

- Using function `fopen`:
  ```
  FILE* fopen(const char *filename,
              const char *mode);
  ```
- Modes for text files:
  - `"r"` to open for reading,
  - `"w"` to open for writing (deletes old contents),
  - `"a"` to open for appending,
  - `"r+"` reading and writing, starts at beginning,
  - `"w+"` reading and writing, deletes old contents,
  - `"a+"` reading and writing, writes at the end position.

Before we read or write a file, we need open it. To open a file, we use the function `fopen`. Its prototype is:

```
FILE* fopen(const char *filename, const char *mode);
```

In this prototype, filename is the absolute or relative pathname of a file to be opened. The mode specifies what operations we intend to perform on the file. It can have the following three values:

- `"r"` to open for reading,
- `"w"` to open for writing. The file does not need to exist. If it exists, the old contents will be deleted before writing.
- `"a"` to open for appending. If the file does not exist, a new file will be started.
- `"r+"` reading and writing, starts at beginning,
- `"w+"` reading and writing, deletes old contents,
- `"a+"` reading and writing, writes at the end position.

To open a binary file, we need to add a '`b`' at the end or just before '`+`' in a mode; i.e., we have modes:

- `"rb"` to open for reading,
- `"wb"` to open for writing,
- `"ab"` to open for appending,
- `"r+b"` or `"rb+"` reading and writing, starts at beginning,
- `"w+b"` or `"wb+"` reading and writing, deletes old contents,
- `"a+b"` or `"ab+"` reading and writing, writes at the end position.

A Unix system and Unix-like systems (e.g., Linux) do not really make any difference between a text file and a binary file. This distinction exists for the system such as DOS, Windows, and old-style Mac, in which the the newline character '`\n`' needs to be interpreted differently.

If this function fails to open a file, it will return a NULL pointer. Otherwise it returns a file pointer using which we can access the file.

Closing a File

When we no longer need a file, we should close it. We can use the following function:

```
int fclose(FILE* fp);
```

This function returns 0 if it successfully closes the file that the file pointer fp is associated with. Otherwise, it returns EOF.

25.3 Formatted I/O with a File

Now let us learn some functions that we can use to read and write text files. One set of such functions is used for formatted I/O. It is easy to learn them as long as we understand that `printf` and `scanf` are special cases of the following two functions:

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

The only difference is that `fprintf` and `fscanf` have a parameter that is a file pointer, so that they will write to or read from a file.

Thus, `printf(...)` is equivalent to `fprintf(stdout, ...)`, and `scanf(...)` is equivalent to `fscanf(stdin, ...)`. We can also print error messages to `stderr` as Unix utilities typically do. This can be achieved using `fprintf(stderr, ...)`.

Let us use what we have learned to write "hello, world" to a file:

```
FILE *fp;
fp = fopen("hello.txt", "w");
if (fp == NULL) {
   fprintf(stderr, "Cannot open hello.txt");
   exit(EXIT_FAILURE);
}

fprintf(fp, "hello, world\n");
fclose(fp);
```

25.4 Character I/O with a File

There is another set of functions that can be used to read or write a file character by character. The output function is:

```
int putc(int c, FILE *stream);
```

The function writes the character `c` to `stream`. If a write error occurs, it returns EOF. Otherwise, the character written is returned. Thus, `putchar(c)` is equivalent to `putc(c` stdout),.

The following is the input function:

```
int getc(FILE *stream);
```

This function reads a character from stream and returns the character read. When the end of file is reached, EOF is returned. Thus `getchar()` is equivalent to `getc(stdin)`.

Now let us see an example. Previously we learned a Unix command `wc`, which computes the number of characters, words and lines in a file. We will implement here one of its features: computing the number of characters in a file. The filename is given as a command-line argument. A fill-in-the-blanks program is given in `~prof2132/public/countchar.c` and is also included below:

```
/* Program: countchar.c */
#include <stdio.h>
```

```
int main(int argc, char* argv[]) {
  FILE *fp;
  int count = 0;

  if (argc != 2) {
    fprintf(stderr, "usage: %s file\n", __________ );
    return 1;
  }

  if ((fp = fopen( _______________ )) == NULL) {

    fprintf(stderr, "cannot open file %s\n", _________ );
    return 1;
  }

  while (getc(fp) != EOF)
    count++;

  printf("There are %d characters.\n", count);

  fclose(fp);
  return 0;
}
```

25.5 Reading and Writing Blocks to Files and Other Functions

More Read and Write Functions

- Reading end-of-file indicator:
  `int feof(FILE *stream);`
- Reading and writing blocks of data:

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

- Note: we can ignore keyword `restrict`

File Positioning

- Reading and setting file position:

```
void rewind(FILE *stream);
long int ftell(FILE *stream);
int fseek(FILE *stream, long int offset,
          int whence);
```

- `whence` can be `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`
- Additional functions (better for very large files):

```
int fgetpos(FILE *restrict stream,
            fpos_t *restrict pos);
int fsetpos(FILE *stream,
```

```
                const fpos_t *pos);
```

When working with file data, we sometimes have a choice of writing it in a textual file or a binary file. For example, we can write a number using `fwrite` and read it using `fread` from a binary file, or we can write it using `fprintf` and read it using `fscanf` from a textual file. Some advantages of a binary representation is that we typically save on disk space because such representation is more compact, and we do not loose on precision in floating-point numbers for example. Advantages of textual representation is that it is more clear and easy to inspect externally, and it is generally more portable since it does not depend on internal memory representation, which may be different on different computing platforms. One difference is such representation is for example big-endian vs. little-endian representation.

Let us consider the following example programs:

Example: Writing and Reading Double in Binary

```
/* Program: writedouble.c */
#include <stdio.h>

int main() {
  FILE *fp;
  double d;
  printf("Enter a double: "); scanf("%lf", &d);
  printf("Saving double in tmp1.\n");
  fp = fopen("tmp1", "wb");
  fwrite(&d, sizeof(double), 1, fp);
  close(fp);
  return 1;
}
```

In the above program, we declare a file pointer `fp`, read a floating-point number into the variable '`d`' of type double, and then we save this variable into the file named '`tmp1`'. The file needs to be declared as a binary file, open for writing, which will delete any previous contents. The main writing function call:

```
  fwrite(&d, sizeof(double), 1, fp);
```

uses the address of the variable `d` as the first parameter, followed by the number of bytes that needs to be written: `sizeof(double)`. The next parameter '`1`' is the number of such blocks of memory, which is only one, and finally we have the file pointer `fp`. This call will write out the memory image of the variable `d` exactly as stored in memory to the file '`tmp1`' on disk. At the end, we close the file.

The next program is used to read the same file:

```
/* Program: readdouble.c */
#include <stdio.h>

int main() {
  FILE *fp;
  double d;
  int c;
  printf("Reading double from tmp1:\n");
  fp = fopen("tmp1", "rb");
  fread(&d, sizeof(double), 1, fp);
  close(fp);
  printf("Read: %lf\n", d);
```

```
  printf("Bit layout:\n");
  rewind(fp);
  while(EOF != (c = getc(fp))) {
    int b = 1 << 7;
    for (; b != 0; b >>= 1)
      putchar( (b & c) ? '1' : '0' );
  }
  putchar('\n');

  return 1;
}
```

Although we could read the value of the floating-point number again into a variable of type double, and get the same value, we choose to read it byte by byte and print out the bit patters of those bytes. Each byte of the file is read into the variable '`c`' and the bit mask '`b`' is used to examine the bits of these bytes and print them out. This is one way to examine the binary representation of any floating-point number of type 'double'.