Informatics

Ingeniería en Electrónica y Automática Industrial

Input and Output With Files

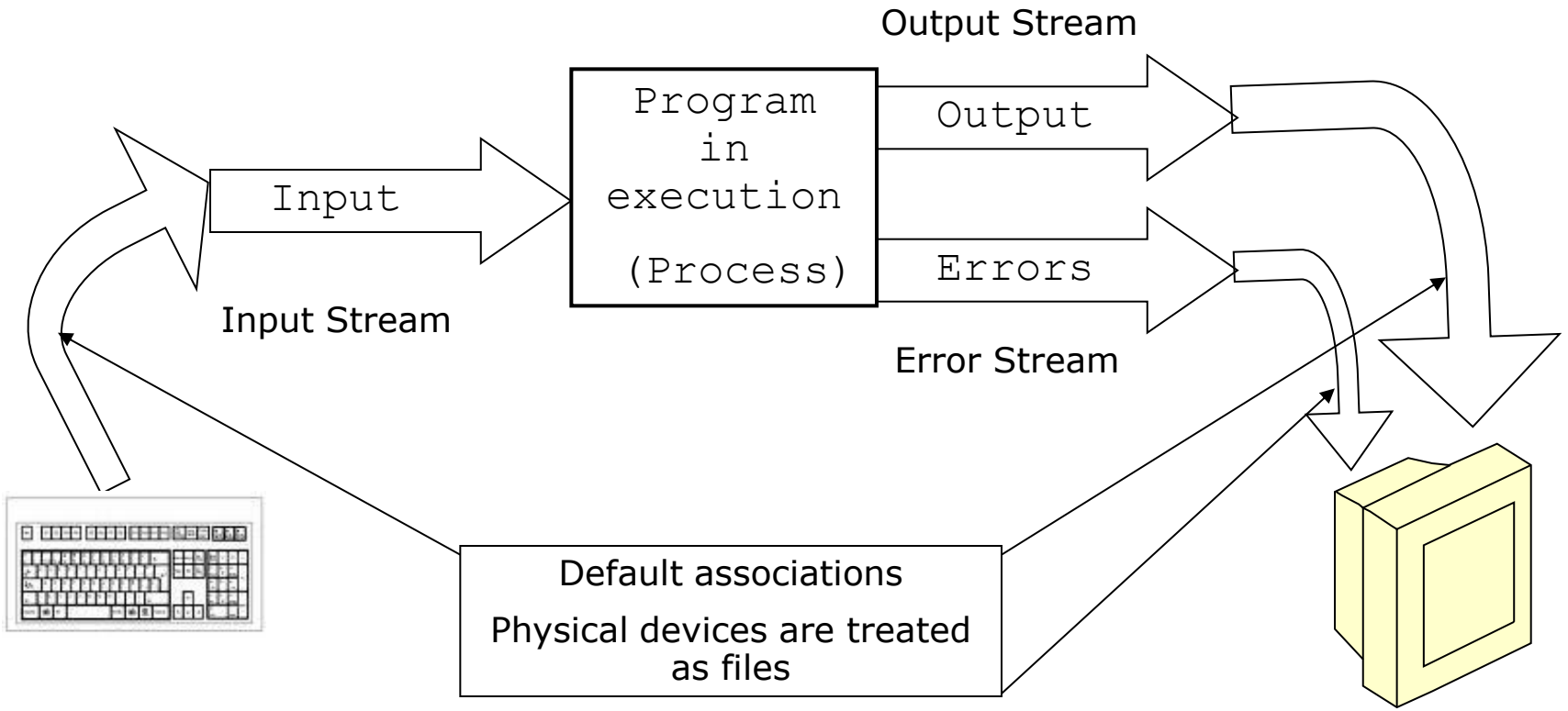
Input and Output With Files in C Language

- Files and streams in C language
- Opening and closing files
- Text input/output
  - Characters
  - Text
- Binary data input/output
- Formatted Input/Output in Files
- File positioning: `fseek()`
- Other operations over files
  - Function `ftell()`
  - Function `rewind()`
  - Function `remove()`
  - Function `fflush()`
  - Function `tmpfile()`

Files and streams in C language (I)

- Storing information requires a system for **I/O with files**
  - Independent of the physical device
  - Implemented with generic, powerful and flexible functions
  - Two general tools exist: ***Files*** and ***Streams***
- **File**: Sequence of bytes stored/sent in/to some device (hard disc, printer, keyboard, controller, screen…)
- **Stream**: Abstract element over which every I/O operation is performed to make them easier for the programmer
  - It works as intermediary between programs and files.
  - Physically is a part of the memory working as a *buffer*
  - Three special streams exist:
    - `Stdin` ***Standard Input*** *Stream* associated to the keyboard
    - `Stdout` ***Standard Output*** *Stream* associated to the screen
    - `Stderr` ***Standard Error*** *Stream* associated to the screen

Files and streams in C language (II)



Files and streams in C language (III)

- In DOS/Windows a file can be opened in two ways:
  - *Text mode:* bytes are considered to be ASCII codes
    - End of line is `'\n'` (in ASCII `CR`: Carriage return)
    - When writing, `CR+LF` (Intro) is converted to `'\n'`
  - *Binary mode*: bytes are considered to be binary code
- In Unix/Linux there is no such distinction

Opening and closing files (I)

- To access any file it is necessary a **file descriptor**
  - Declaration:

```
FILE *pfile;
```

  - `FILE` is a constant defined in `stdio.h`
  - The descriptor `pfile` points to a buffer that will contain all the information about the file
  - It is used in any operation with the file
  - It must be declared before use
  - It is initialized when opening a file without error

Opening and closing files (II)

- Before any operation the file must be **opened** with `fopen()`

```
FILE *pfile
pfile = fopen("filename", "mode");
```

  - `fopen` receives two character chains
    - First one with the file's name (including access path)
    - Second one with the opening mode
  - It returns:
    - The descriptor `pfile` that points to an structure that contains all information about the file: name, size, attributes….
    - The `NULL` descriptor in case of error

Opening and closing files (III)

FILE OPENING MODES IN C			
Opening modes	**String**		**Observations**
	Text file	**Binary file**	
Open to read	"r"	"rb"	If it does not exist, error is produced
Create to write	"w"	"wb"	If it exist, content is lost
Open or create to append	"a"	"ab"	If it does not exist, is created
Open to read and/or write	"r+"	"rb+"	It must exist
Create to read and/or write	"w+"	"wb+"	If it exist, content is lost
Open or create to append and/or read	"a+"	"ab+"	If it does not exist, is created

Opening and closing files (IV)

- When the program finishes normally all open files are closed by the OS but
- To prevent from abnormal termination, it is recommended to close all files in the program with

```
fclose(pfile);
```

  - It receives as argument de file descriptor `pfile`
  - It returns
    - An integer with ‘0’ value if normal closing
    - `EOF` in case of error
- All information of a non-propertly closed file is lost

Opening and closing files (V)

- Example:

```
FILE *pf;                                   /* descriptor */
if ((pf=fopen("myadata/draft.txt","w+")) == NULL)
{
    puts("\nFile can't be created");
    exit(0);
}
else printf("\nFile has been opened");

/* ... Program ...  */

fclose(pf);                                 /* File is closed */
```

Opening and closing files (VI)

- The **end of file** is indicated with the special character **EOF**, defined in `stdio.h`
  - It is the last byte of the file
  - When bytes are read with `fgetc()`, `EOF` might be not distinguished as last character so,
  - Function `feof()` of `stdio.h` returns a value different from cero (true) when `EOF` is read

```
while (!feof(pfile))
{
      /* operarions with the open file */
}
```

Text input/output (I)

- Functions to Read/Write ONE character (`stdio.h`)
  - `int fgetc(FILE *pfile);`
    - Reads a character from the file whose descriptor `pfile` receives
    - Returns the read character in an integer or `EOF` in case of error
  - `int fputc(int char, FILE *pfile);`
    - Writes a character in the file whose descriptor receives
    - Receives as arguments
      - `char`: The character to write
      - `pfile`: The file descriptor
    - Returns `EOF` in case of error

Text input/output (II)

- Example:

```
FILE *pf1, *pf2;
char letter;
pf1 = fopen(“read.txt”, “r”);
letter = fgetc(pf1);  /* Reads 1 character */
pf2 = fopen (“write.txt”, “w”);
fputc(letter, pf2);  /* Writes 1 character */
fclose(pf1);
fclose(pf2);
```

Text input/output(III)

- Functions to Read/Write strings (`stdio.h`)
  - `char * fgets(char *cad, int numchar, FILE *pf);`
    - It receives as arguments
      - `cad`: Pointer to where the string will be stored
      - `numchar-1`: Number of character to read (`'\0'` is added)
      - `pf`: file descriptor
    - It returns a pointer to the chain or NULL in case of error
    - The character `\n` is the last one read if found
  - `int fputs(char *pstring, FILE *pf);`
    - It receives as arguments
      - `pstring`: A pointer to the string to be written
      - `pf`: File descriptor
    - It returns the last written character or `EOF` in case of error

Text input/output(IV)

- Example:

```
FILE *pf1, *pf2;
char rea[50];
char writ[]="Message to keep in the file";
int num=50-1;
pf1 = fopen("read.txt", "r");
fgets(rea, num, pf1);
   /* Reads a string of 49 chars from read.txt */
pf2 = fopen ("write.txt", "w");
fputs(writ, pf2);
/*Writes the string "Message to …" in write.txt*/
fclose(pf1);
fclose(pf2);
```

Binary data Input/Output (I)

- To **read** binary data: `fread()` (in `stdio.h`)

```
unsigned fread(void *pdat, unsigned numbytes,
               unsigned numdat, FILE *pfile);
```

- To **write** binary data: `fwrite()` (in `stdio.h`)

```
unsigned fwrite(void *pdat, unsigned numbytes,
                unsigned numdat, FILE *pfile);
```

  - They return the number of read/written data
  - They receive
    - `pdat`: A pointer to the read/written data
    - `numbytes`: number of bytes that each data occupies (`sizeof`)
    - `numdat`: Total number of data
    - `pfile`: File descriptor

Binary data Input/Output (II)

- Example:

```
FILE *pf;
float value1=3.5, value2;
pf=fopen (“file.dat”, “a+”);
fwrite(&value1, sizeof(value1), 1, pf); /*Writes*/
fread(&value2, sizeof(float), 1, pf);   /*Reads */
fclose(pf)
```

Formatted Input/Output in Files (I)

- `fprintf()` y `fcanf()` in `stdio.h` are analogous to `printf()` and `scanf()` but using a file descriptor
- `int fprintf(FILE *pf, char *format, arglist);`
- `int fscanf(FILE *pf, char *format, arglist);`
  - They receive
    - `pf`: File descriptor
    - `format`: A string that specifies formats
    - `arglist`: Arguments to be written/read
  - They return
    - `fprintf()` returns the number of written bytes
    - `fscanf()` returns the number of read bytes or EOF

Formatted Input/Output in Files (II)

- Example:

```
FILE *pf;
int i = 100;
char c = 'C';
float f = 1.234;
pf = fopen("myfile.dat", "w+");
fprintf(pf, "%d %c %f", i, c, f);
                          /* Writes in the file */
fscanf(pf, "%d %c %f", &i, &c, &f);
                           /* Read from the file */
fclose(pf);
```

File positioning: `fseek()` (I)

- With `fseek()` the program can **access directly any position in the file** (random vs sequential access)
- `FILE` pointer points to an structure created by the OS to control operations over the file
  - It incudes a **read/write pointer** that contains the current position to read/write
  - When opening a file this pointer points to the beginning of the file (except if open to append)
- Therefore `fseek()` allows to read/write in any position of the file by setting the value of the read/write pointer

File positioning: `fseek()` (II)

```
int fseek(FILE *pf, long offset, int origin);
```

- It returns ‘true’ if success (right movement) or NULL otherwise
- It receives
  - `pf`: File descriptor
  - `origin`: Initial reference point. Some references defined in `stdio.h` can be taken:
    - `SEEK_SET`: beginning of the file
    - `SEEK_CUR`: current position
    - `SEEK_END` : end of the file
  - `offset`: Value to add to `origin` to obtain the new position

File positioning: `fseek()` (III)

- Example:

```
FILE * pFile;
pFile = fopen ("example.txt" , "w");
fputs ("This is an apple." , pFile);
fseek (pFile , 9 , SEEK_SET);
fputs (" sam" , pFile);
fclose (pFile);
```

After execution, example.txt will contain:

```
"This is an sample."
```

Other operations over files (I)

- Function `ftell()`

```
long ftell(FILE *pf);
```

  - Returns a long integer with the position of the write/read pointer with respect to the origin of the file.
  - Receives the file descriptor `pf`
  - Defined in `stdio.h`

Other operations over files (II)

- Function `rewind()`

```
void rewind(FILE *pf);
```

  - Initilializes read/write pointer the beginning of the file
  - Does not return anything
  - It receives the file descriptor
  - Defined in `stdio.h`

Other operations over files (III)

- Function `remove()`

```
int remove(char *filename);
```

  - It removes the file pointed by `filename`
  - Returns 0 if success and -1 if error
    - In case of error, global variable `errno`, defined in `errno.h` will indicate the kind of error
  - Defined in `stdio.h`

Other operations over files (IV)

- Function `fflush()`

```
int fflush(FILE  *pf);
```

  - It empties I/O buffers associated to the descriptor `pf`
  - Returns 0 if success and `NULL` if error
  - Defined in `stdio.h`
  - Very used to erase keyboard buffer and when working with printers

Other operations over files (V)

- Function `tmpfile()`

```
FILE * tmpfile(void));
```

  - It creates a temporal file that is automatically removed when the file is closed or the program ends.
  - The temporal file is created in "`w+`" mode
  - It returns a file descriptor to the temporal file (or null pointer if cannot be created)
  - Defined in `stdio.h`