Unit 3: Files

Programming 2

Degree in Computer Engineering
University of Alicante
2023-2024

Introduction

- Data we have worked with so far is stored in the main memory of the computer (*RAM*)
- The size of the main memory is limited (a few Gigabytes)
- All data is deleted when the program ends (*volatile memory*)

- *Files* are the way in which C++ allows accessing to the information stored in disk (*secondary storage*)
- Files are dynamic structures: their size may vary during the execution of the program according to the data they store
- There are two types of files depending on how the information is stored inside: *text files* and *binary files*

- There are two ways to access a file:
    - *Sequential access*: reads/writes the elements of the file in order, starting from the beginning and one after the other

    ... 1 2 3 4 5 6 7 8 ...
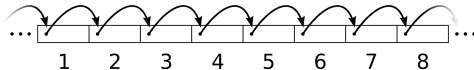
    - *Direct access* (or *random*): Read/write directly in any position of the file, without traversing the previous ones

    1 3 7 2 8 6 4 5
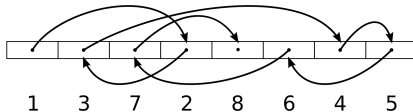
Text files

Definition (1/2)

- *Text files* are also called *files with format*
- A text file stores the information as a character sequence
- For example, the integer value `19` is saved in a text file as the characters `1` and `9`
- Examples of text files: C++ source code, a web page (HTML) or a file created with a text editor
- The most common read/write mode used in text files is the sequential access

Definition (2/2)

- Text files contain only printable characters: those whose ASCII code is greater or equal to 32
- Each character is assigned an ASCII code for its storage in memory:

Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car	Dec	Hex	Car
0	00	NUL	32	20	SPC	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C		
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Declaration of variables

- Files are just another data type in C++
- The `fstream` library has to be included in the code to work with files:

```
#include <fstream>
```

- There are three basic data types to work with files, depending on what you want to do with them:*

```cpp
ifstream fileRead; // Read only
ofstream fileWrite; // Write only
fstream fileReadWrite; // Read and write
```

*It is unusual to use the `fstream` type with text files

- A file variable (*logical file*) must be associated with a real file in the system (*physical file*) in order to be able to read/write in it
- The file must be opened with `open` to establish the relationship between the variable and the physical file:

```cpp
ifstream file; // Read from file
file.open("myFile.txt");
// Now we can read from "myFile.txt"
```

- The file name can be passed as a character array or as a `string`:*

```cpp
char fileName[]="myFile.txt";
file.open(fileName);
```

*Using `string` is available from C++ 2011 version onward

- A second parameter can be passed to `open` specifying the file's *opening mode*:
  - Read: `ios::in`
  - Write: `ios::out`
  - Read/write: `ios::in | ios::out`
  - Append: `ios::out | ios::app`

```cpp
ifstream fr;
ofstream fw;
// Open for reading only
fr.open("myFile.txt",ios::in);
// Open for adding information at the end
fw.open("myFile.txt",ios::out|ios::app);
```

- If a file that already exists is opened for writing (`ios::out`) all its contents are deleted
- If the file is opened with `ios:app` the content is not deleted, and the new information is added at the end
- If the file does not exist, a new one is created with an initial size of `0`

- By default, `ifstream` is opened for reading and `ofstream` for writing
- The file can be opened at declaration:

```cpp
ifstream fr("myFile.txt"); // By default ios::in
ofstream fw("myFile.txt"); // By default ios::out
```

- Before reading/writing, check with `is_open` whether the file was correctly opened (`true`) or not (`false`)
- After finishing working with a file, it must be released with `close`:

```cpp
ifstream fr("myFile.txt");
if(fr.is_open()){
    // Now we can work with the file
    ...
    fr.close(); // Close the file
}
else // Show opening error
```

- Reading a file allows retrieving information saved in disk and loading it in the main memory of the computer
- The operator >> can be used to read from files as we did previously with `cin` to read from keyboard
- Loop to read a file character by character:

```cpp
ifstream fr("myFile.txt")
if(fr.is_open()){
  char c; // Could be int, float, ...
  while(fr >> c){ // Read while there are characters
    cout << c;
  }
  fr.close()
}
else{
  cout << "Error opening file" << endl;
}
```

- The operator >> discards blanks in the file, as when reading from `cin`
- The `get` function can be used to read character by character without discarding blanks:

```cpp
ifstream fr("myFile.txt");
if(fr.is_open()){
  char c;
  while(fr.get(c)){
    cout << c;
  }
  fr.close();
}
else{
  cout << "Error opening file" << endl;
}
```

- The operator >> can be used to read files containing different data types
- For example, given a file that contains in each line a string and two integers (e.g. `Hello 1032 124`):

```cpp
ifstream fr("myFile.txt");
if(fr.is_open()){
  string s;
  int num1,num2;
  while(fr >> s){ // Read the string
    fr >> num1; // Read the first integer
    fr >> num2; // Read the second integer
    cout << s << "," << num1 << "," << num2 << endl;
  }
  fr.close()
}
...
```

Reading by lines

- The `getline` function can be used to read a whole line of a file, just as when reading from `cin`:

```cpp
ifstream fr("myFile.txt");
if(fr.is_open()){
  string s;
  while(getline(fr,s)){
    cout << s << endl;
  }
  fr.close();
}
else{
  cout << "Error opening file" << endl;
}
```

Detecting the end of file

- The `eof` method indicates whether the end of the file has been reached
- This occurs when there are no more data to read:

```
ifstream fr;
...
while(!fr.eof()){
  // Read using any of the previous methods
}
```

- When trying to read data outside the file, the methods return `true`
- After reading the last valid data in the file, the method still returns `false`
- It is necessary to do an additional reading to make `eof` to return `true`

Writing with operator <<

- The operator << can be used to write to file, similarly to how we use `cout` to write on screen:

```cpp
ofstream fw("myFile.txt");
if(fw.is_open()){
    int num=10;
    string s="Hello, world";
    fw << "An integer: " << num << endl;
    fw << "A string: " << s << endl;
    fw.close();
}
else{
    cout << "Error opening file" << endl;
}
```

Exercise 1 Write a program that reads a file `file.txt` and prints on screen the lines of the file containing the string `Hello`.

Exercise 2

Write a program that reads a file `file.txt` and writes in another file `FILE.TXT` the content of the input file with all its letters in uppercase.

Example:

file.txt	FILE.TXT
Hello, world.	HELLO, WORLD.
How are you?	HOW ARE YOU?
Bye, bye...	BYE, BYE...

Exercise 3

Write a program that reads two text files, f1.txt and f2.txt, writing on screen the lines that differ in each file, inserting < if the line belongs to f1.txt and > if it belongs to f2.txt.

Example:

f1.txt	f2.txt
hello, world.	hello, world.
how are you?	how is it going?
bye, bye...	bye, bye...

The output should read:

```
< how are you?
> how is it going?
```

Exercise 4

Develop a function `endFile` that receives two parameters: the first one is a positive integer `n` and the second one is the name of a text file. The function must show on screen the last `n` lines of the file.

Example:

```
endFile(3,"strings.txt")

with several words
oneword
maaaany words, many, many...
```

Exercise 4 (continue)

There are two solutions:

1. Using brute force: read the file to count the number of lines and then read it again to write the final n lines. Problem: what if there are 100000000000000 lines?
2. Use an array of strings that always stores the n last lines of the file (although at the beginning there are less than n lines)

Exercise 5

Consider two text files, `f1.txt` and `f2.txt`, in which each line is a series of numbers, separated by `:`. Each line is sorted by the first number, from lowest to highest, in the two files. Write a program that reads the two files, line by line, and writes in a file `f3.txt` the lines common to both files.

Example:

| f1.txt	f2.txt	f3.txt
10:4543:23	10:334:110	10:4543:23:334:110
15:1:234:67	12:222:222	15:1:234:67:881:44
17:188:22	15:881:44	20:111:22:454:313
20:111:22	20:454:313	

Binary files

Definition (1/2)

- Also called *files without format*
- They store information in the same way as it is stored in the main memory of the computer
- For example, the integer (char) value `19` is saved to file as the sequence `00010011`
- Functions used to read and write binary files are different to those used for text files
- Both sequential and direct access are commonly used with binary files
- Reading and writing is faster than with text files (there is no conversion to character data)
- Binary files usually require less disk space than their equivalent text files

- It is very usual to store each item in the file using a `struct`:

```
struct TStudent{
  char name[100];
  int group;
  float avgMark;
};
```

- Using direct access, the n record of the file can be accessed without having to read the previous $n-1$ records

TStudent	TStudent	TStudent	⋯	TStudent	TStudent
0	1	2		n	n+1
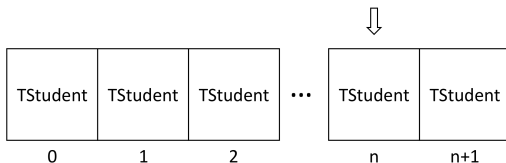
- Variables are declared as in text files:

```
#include <fstream> // Required to work with files

ifstream fileRead; // Read only
ofstream fileWrite; // Write only
fstream fileReadWrite; // Read and write
```

Opening and closing

- Binary files must be opened setting the opening mode to
  ios::binary:
  - Read: ios::in | ios::binary
  - Write: ios::out | ios::binary
  - Read/write: ios::in | ios::out | ios::binary
  - Append: ios::out | ios::app | ios::binary

```
ifstream fileRead;
ofstream fileWrite;
// Open to read only in binary mode
fileRead.open("myFile.dat",ios::in | ios::binary);
// Open to write only in binary mode
fileWrite.open("myFile.dat",ios::out | ios::binary);
// Abbreviated form
fstream fileReadWrite("myFile.dat",ios::binary)
```

- As with text files, it can be checked whether the file is open with
  is_open and release it with close

- The function `read` is used to read data from binary files
- This function receives two parameters: the first one indicates where is saved the information read from the file, the second one indicates the amount of information (number of bytes) to be read:*

```cpp
TStudent student;
ifstream file;

file.open("myFile.dat",ios::in | ios::binary);
if(file.is_open()){
  // In each iteration a TStudent record is read
  while (file.read((char *)&student, sizeof(TStudent))){
    // Show the name and mark of each student
    cout << student.name << ": " << student.mark << endl;
  }
  file.close();
}
```

*The function `sizeof` can be used to get the number of bytes occupied by any data type

Reading (2/3)

- The n-th record from the file can be read without having to read the previous n-1 records (direct access)
- The function seekg allows positioning the reading window at a specific point in the file
- This function receives two parameters: the first one indicates how many bytes we want to skip, the second one indicates the reference point to make the jump

```cpp
// There is a file with TStudent records
ifstream file("myFile.dat",ios::binary);
TStudent student;
...
// We can read directly the third record
// Skip the first two records
file.seekg(2*sizeof(TStudent),ios::beg);
// Now we can read the third record
file.read((char *)&student,sizeof(TStudent));
...
```

Reading (3/3)

- Possible reference points:
  - ios::beg: from the beginning of the file
  - ios::cur: from the current position
  - ios::end: from the end of the file
- If the first parameter of seekg is a negative number, the reading window moves towards the beginning of the file:

```
ifstream file("myFile.dat",ios::binary);
TStudent student;
...
file.seekg(-1*sizeof(TStudent),ios::end);
// Read the last record in the file
file.read((char *)&student,sizeof(TStudent));
...
```

- The function write is used to write to a binary file
- This function receives two parameters: the first one indicates where the information to be written to the file is stored, the second one indicates the amount of information (number of bytes) to be written:
- The syntax is very similar to that of read:

```
ofstream file("myFile.dat",ios::binary);
TStudent student;

if(file.is_open())
{
  strcpy(student.name,"John Doe");
  student.avgMark=7.8;
  student.group=5;

  file.write((const char *)&student,sizeof(TStudent));
  file.close();
}
```

- As when reading, the n-th record from the file can be written without having to write in the previous $n-1$ records
- The function seekp allows positioning the writing window at a specific point in the file (seekg is for reading)
- Parameters are the same as for seekg:

```cpp
ofstream file("myFile.dat",ios::binary);
TStudent student;
...
// Positioning to write in the third record
file.seekp(2*sizeof(TStudent),ios::beg);
file.write((const char *)&student,sizeof(TStudent));
...
```

- If the position searched with seekp does not exist in the file, the file is "enlarged" to write the data

- Character arrays must be used instead of string to store strings in a binary file
- The problem with string is that it is a variable size data type, thus records including this type would have different sizes
- When using character arrays, it might be necessary to trim the string to fit in the record before saving it to file:

```
const int SIZE=20;
char str[SIZE];
string s;
...
strncpy(str,s.c_str(),SIZE-1); // Maximum 19 chars
str[SIZE-1]='\0';
```

- The current position (in bytes) of the reading window can be obtained with the `tellg` function, and the writing window with `tellp`
- This can be used, for example, to calculate the number of records in a file:

```
ifstream file("myFile.dat",ios::binary);
// Set the reading window at the end of the file
file.seekg(0,ios::end);
// Calculate the number of TStudent records in the file
cout << file.tellg()/sizeof(TStudent) << endl;
```

Exercise 6

Given a binary file students.dat that stores records with the following information for each student:

- id: 10 characters array
- surname: 40 characters array
- name: 20 characters array
- group: integer

Write a program that prints on screen the id of all the students belonging to group 7.

Extension: write a program that exchanges the students in groups 4 and 8 (groups range from 1 to 10).

Exercise 7

Given the file `students.dat` from the previous exercise, write a program that converts the name and surname of the fifth student to uppercase, and then writes it again to file.

Exercise 8

Write a program that creates a file `students.dat` from the data stored in a text file `students.txt` in which each field (`id`, `surname`, etc.) is in a different line. Consider that the `id`, name and surname may be longer than the length specified for the binary file, and may need to be trimmed.

Exercise 9

Write a program that automatically assigns students to 10 possible groups. Each student is assigned the group that matches the last number on their `id` (if the `id` ends in 0, the group 10 is assigned). The student data is stored in a file `students.dat` with the same structure as in previous exercises.

Assigning groups must be done by reading the file only once, not storing it into the main memory. At each step, the information corresponding to a student is read, the group is calculated, and the record is stored at the same position.