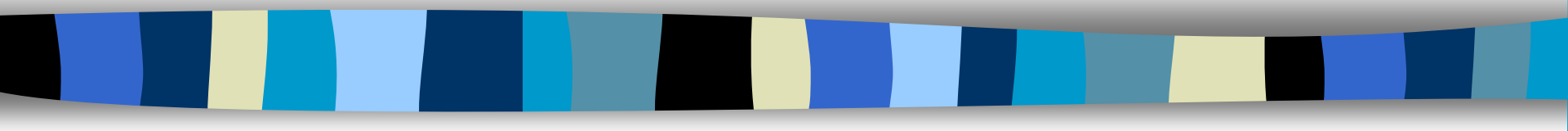


chapitre 5- Flots et Fichiers



Ressource R1.01 : Initiation au développement - Partie 2

Institut Universitaire de Technologie de Bayonne – Pays Basque
BUT Informatique – Semestre 1 - P. Dagorret


Plan

1.- Introduction aux fichiers	3
Notion de fichier	3
Exemples de fichiers	6
Catégorisation des fichiers selon leur nature	7
Catégorisation des fichiers selon leur contenu	3
Organisation logique	9
Mode d'accès	16
Organisation logique & Mode d'accès - Conclusion	21
2.- Programmer avec des fichiers	25
Éléments nécessaires	25
Primitives nécessaires	27
Graphe de précedence entre les primitives	28
Mode d'ouverture	29
3.- Fichiers et C++	30
Flots d'E/S conversationnels <code>cin</code> et <code>cout</code>	30
Généralisation : flots d'E/S	31
Bibliothèques C++ pour la gestion des flots	33
Bibliothèque <code>fstream</code> pour la gestion de flots associés à des fichiers	34
4.- TAD UnFichierTexte	35
5.- TAD UnFichierTexte – Déclarer, Associer, Ouvrir, Fermer	36
Déclarer une variable fichier de texte	36
Associer	37
Ouvrir	38
Fermer	40
Exercice récapitulatif	41
6.- TAD UnFichierTexte – Écrire dans un fichier texte	43
7.- TAD UnFichierTexte – Lire dans un fichier texte	45
Lecture ligne à ligne	46
Lecture mot à mot	48
Lecture caractère à caractère	50
8.- TAD UnFichierTexte – Autres primitives	52
9.- TAD UnFichierTexte – Structure d'un programme traitant un fichier texte	56
Algorithmes de parcours séquentiels	57
Exercice récapitulatif	58

1.- Introduction aux fichiers –

Notion de fichier

□ Notion de fichier

- Conservation d'informations en **Mémoire Secondaire** :
 - disques, bandes magnétiques, cassettes, disques compacts, DVD,...
 - pérennité : tant que cette mémoire n'est pas endommagée ou effacée
 - grande capacité de stockage par rapport à la **Mémoire Centrale**
- Identification d'un fichier :
 - par son **nom système** = le nom du fichier sur le disque
 - par des **attributs** (date de création, taille, icône, extension,...)
- En informatique, 2 points de vues
 - Point de vue Système d'exploitation qui s'intéresse à l'organisation des blocs d'informations sur le disque
 - Allocation d'espace disque, organisation des blocs
 - Transfert des blocs d'octets entre disque et mémoire centrale
 - Gestion des répertoires
 - Optimisation de l'espace mémoire
 - Optimisation des temps d'accès aux blocs d'octets
 -  – Point de vue du **programmeur**,
qui **utilise les fichiers depuis ses programmes**

1.- Introduction aux fichiers –

Notion de fichier

– Point de vue Système d'exploitation



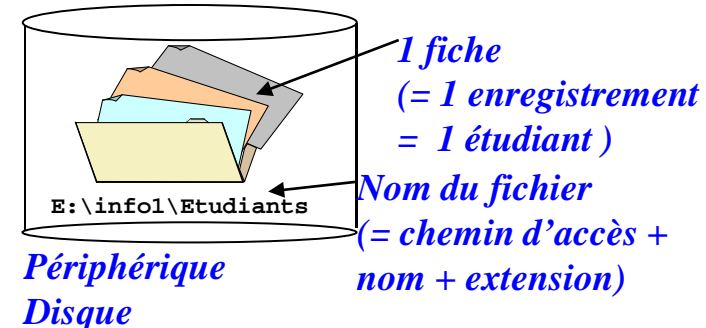
– Point de vue de l'utilisateur **programmeur**

- Un fichier est une **structure de données** composée d'éléments (**fiches** / **enregistrements** / **articles**) dont le nombre n'est pas connu *a priori*, et résidant en **mémoire secondaire**
 - Exemples : fichier d'étudiants, fichier texte, fichier de produits, fichier image
- Une **fiche** / **enregistrement** / **article** est l'ensemble des informations qui décrivent un **élément** du fichier.
- Une fiche / enregistrement peut être elle-même une information simple ou **composée**. Lorsqu'elle est composée, ses composants sont appelés **champs** / **attributs**
 - Exemples :
1 fiche étudiant décrite par {num, nom, prénom, dateNaissance, adresse} ; 1 fiche = 1 nombre ;
1 fiche produit décrite par {codeProduit, libellé, prixU, qtéEnStock} ; 1 fiche = 1 ligne de texte ;
- Un **champ** / **attribut** est une information simple ou composée
 - Exemple : nom = string; adresse = {numRue, nomRue, CodePostal, nomVille}
- Un ou plusieurs champs peuvent jouer le rôle de **clé**
- Les fichiers peuvent être **utilisés** / **créés** depuis un programme (par ex. C++) via le **Système de Gestion de Fichiers** (SGF) = partie/module du système d'exploitation qui se charge du stockage et de la manipulation de fichiers.
- L'**accès** aux enregistrements d'un fichier se fait en fournissant au Système de Gestion de Fichiers : le nom du fichier, la référence de l'enregistrement souhaité
- Il existe différentes **méthodes d'accès** aux enregistrements d'un fichier

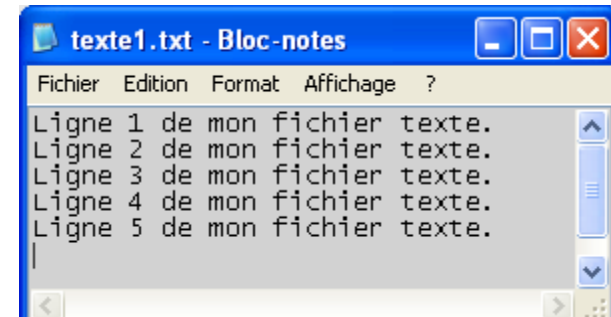
1.- Introduction aux fichiers – Exemples de fichiers

❑ Exemples de fichiers

- *Exemple 1 : fichier des étudiants*
= Ensemble de fiches.
L'élément de base = 1 fiche
= 1 enregistrement = description **structurée**
d'**un** étudiant (num étudiant, nom, prénom, ...)



- *Exemple 2 : fichier de texte*
Fichier composé de lignes de texte séparées par un caractère spécial représentant la finDeLigne (EOL).
L'unité de base (*fiche=enregistrement*) = 1 ligne.



- *Exemple 3 : fichier pluviométrique*
Hauteurs pluviométriques (= hauteur de pluie tombée en mm) captées 2 fois par jour chaque jour de l'année et enregistrées dans un *fichier de nombres* éditable, pour un traitement statistique ultérieur.
L'unité de base (*fiche=enregistrement*) = 1 entier.
Les entiers sont séparés par des séparateurs (espace ou tabulation).

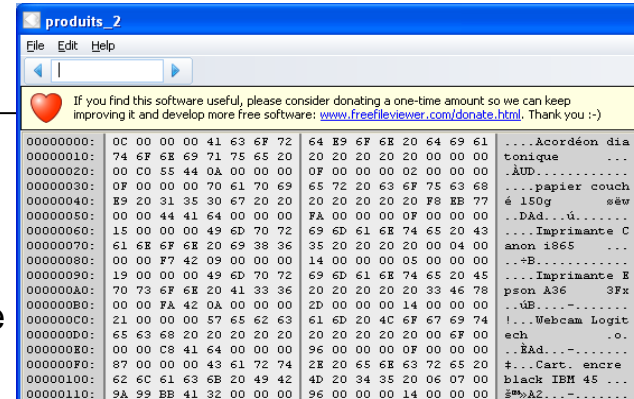
Fichier	Edition	Format	Affichage	?					
2	20	15	3	25	10	11	15	0	8
12	12	10	9	7	5	20	2	5	9
8	5	7	6	12	11	15	20	12	11

1.- Introduction aux fichiers – Exemples de fichiers

– Exemple 4 : fichier de produits

Exemple de fichier de produits en vente par un magasin.

L'unité de base = *fiche* = *enregistrement* est structurée = {codeProduit, libellé, prixU, qtéEnStock}.

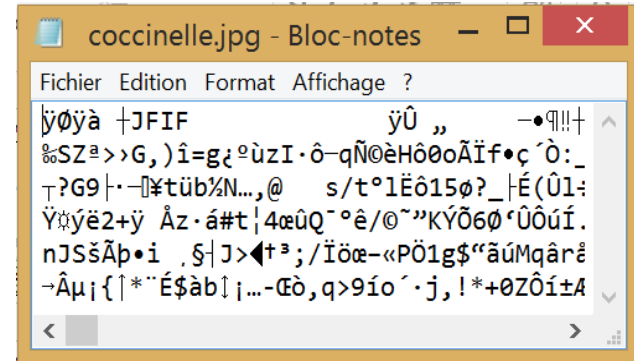


00000000:	0C 00 00 00 41 63 6F 72	64 E9 6F 6E 20 64 69 61Accordéon dia
00000010:	74 6F 6E 69 71 75 65 20	20 20 20 20 20 00 00 00	tonique
00000020:	00 C0 55 44 0A 00 00 00	0F 00 00 00 02 00 00 00	..AUD.....
00000030:	0F 00 00 00 70 61 70 69	65 72 20 63 6F 75 63 68papier couch
00000040:	E9 20 31 35 30 67 20 20	20 20 20 20 20 F8 BB 77	é 150gsèr
00000050:	00 00 44 41 64 00 00 00	FA 00 00 00 0F 00 00 00	...DAd...ù.....
00000060:	15 00 00 00 49 6D 70 72	69 6D 61 6E 74 65 20 43Imprimante C
00000070:	61 6E 6F 6E 20 69 38 36	35 20 20 20 20 00 04 00	anon 1865
00000080:	00 00 F7 42 09 00 00 00	14 00 00 00 05 00 00 00	...+B.....
00000090:	19 00 00 00 49 6D 70 72	69 6D 61 6E 74 65 20 45Imprimante E
000000A0:	70 73 6F 6E 20 41 33 36	20 20 20 20 20 33 46 78	pson A36 3Fx
000000B0:	00 00 FA 42 0A 00 00 00	2D 00 00 00 14 00 00 00	...dB.....
000000C0:	21 00 00 00 57 65 62 63	61 6D 20 4C 6F 67 69 74	!...Webcam Logit
000000D0:	65 63 68 20 20 20 20 20	20 20 20 20 20 00 6F 00	echo.
000000E0:	00 00 C8 41 64 00 00 00	96 00 00 00 0F 00 00 00	...ÈAd.....
000000F0:	87 00 00 00 43 61 72 74	2E 20 65 6E 63 72 65 20	+...Cart. encre
00000100:	62 6C 61 69 6B 20 49 42	4D 20 34 35 20 06 07 00	black IBM 45
00000110:	9A 99 BB 41 32 00 00 00	96 00 00 00 14 00 00 00	##A2.....

– Exemple 5 : fichier image

Un fichier d'image est composé d'octets terminés par un caractère finDeFichier (EOF)

L'unité de base (*fiche*=*enregistrement*) = 1 octet.



– Exemple 6 : fichier de configuration

Composé d'informations hétérogènes définissant **les réglages** d'une application ou d'un service :

n° licence, date installation, durée du contrat,...).

Il n'y a pas d'homogénéité des contenus

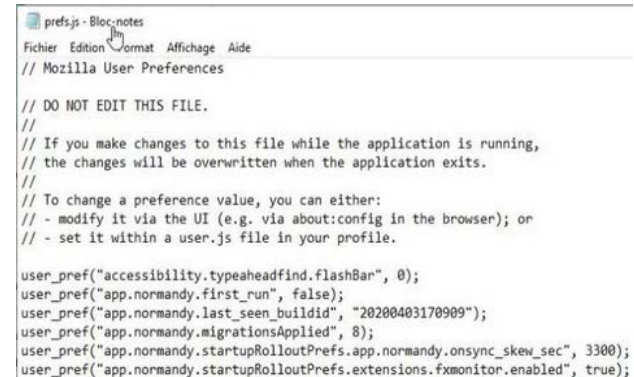
→ **c'est en général un fichier de texte**

Exemples :

...fichier **.htaccess** = configuration d'un serveur web (Apache) relativement aux répertoires et sous-répertoires du site web

...fichiers **user.js** et **prefs.js** = fichiers de configuration de Mozilla Firefox.

Il stockent et permettent de modifier la configuration de l'utilisateur.



```
// DO NOT EDIT THIS FILE.
//
// If you make changes to this file while the application is running,
// the changes will be overwritten when the application exits.
//
// To change a preference value, you can either:
// - modify it via the UI (e.g. via about:config in the browser); or
// - set it within a user.js file in your profile.

user_pref("accessibility.typeaheadfind.flashBar", 0);
user_pref("app.normandy.first_run", false);
user_pref("app.normandy.last_seen_buildid", "20200403170909");
user_pref("app.normandy.migrationsApplied", 8);
user_pref("app.normandy.startupRolloutPrefs.app.normandy.onsync_skew_sec", 3300);
user_pref("app.normandy.startupRolloutPrefs.extensions.fxmonitor.enabled", true);
```

1.- Introduction aux fichiers – Catégorisation des fichiers

❑ Catégorisation des fichiers par rapport à leur **nature** :

- les fichiers (de) **texte** (text files), ou fichiers imprimables/*éditables* :
 - contenu = caractères accolés, lettres et nombres, implémentés par un code – UNICODE - ASCII (LATIN-1, dont les 128 premiers caractères sont ceux du **code ASCII**). Exemple : 35 -> 63C 65C (63C = code ASCII de '3', 65C = code ASCII de '5')
 - susceptibles d'être lus par l'œil d'un humain, imprimés, ou édités avec un éditeur de textes (BlocNotes).
 - contiennent des **caractères**, éventuellement organisés en **lignes** (ou en **mots**)
 - chaque ligne est terminée par un caractère **fin_de_ligne**
 - les mots sont séparés par des **séparateurs** (tabulation, espace,...)
 - après dernière ligne (dernier mot), **fin_de_fichier**
- les fichiers **binaires** (binary files) :
 - leur contenu est stocké dans un format interne.
Exemple : 35 -> 43 ($35_{10} = 43_8$)
 - ne peuvent être décodés/manipulés que par des **programmes ad-hoc** (= spécialement conçus pour)

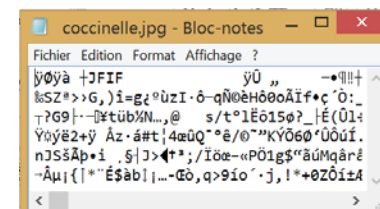


```
prefs.js - Bloc-notes
Fichier Edition Format Affichage Aide
// Mozilla User Preferences

// DO NOT EDIT THIS FILE.
//
// If you make changes to this file while the application is running,
// the changes will be overwritten when the application exits.
//
// To change a preference value, you can either:
// - modify it via the UI (e.g. via about:config in the browser); or
// - set it within a user.js file in your profile.

user_pref("accessibility.typeaheadfind.flashBar", 0);
user_pref("app.normandy.first_run", false);
user_pref("app.normandy.last_seen_buildid", "20200403170909");
user_pref("app.normandy.migrationsApplied", 8);
user_pref("app.normandy.startupRolloutPrefs.app.normandy.onsync_skew_sec", 3300);
user_pref("app.normandy.startupRolloutPrefs.extensions.fxmonitor.enabled", true);
```

Nécessite IrfanView



1.- Introduction aux fichiers –

Catégorisation des fichiers

❑ Catégorisation des fichiers par rapport à leur **contenu** :

Le suffixe ajouté au nom du fichier donne des informations sur le contenu du fichier :

- `fichier.h` / `fichier.cpp` : programme source écrit en C++ (fichier texte)
- `ficher.pas` : programme source écrit en Pascal (fichier texte)
- `ficher.bin` : fichier binaire exécutable
- `ficher.lib` : fichier bibliothèque (library)
- `ficher.dll` : fichier bibliothèque (library) (fichier texte mais compilé)
- `ficher.dat` : fichier de données (fichier texte)



1.- Introduction aux fichiers – Organisation logique

❑ Organisation logique d'un fichier

Elle explicite ***la façon dont les enregistrements sont organisés les uns par rapport aux autres*** dans le fichier.

Les grands types d'organisations logique d'un fichier sont :

- l'organisation **séquentielle**,
- l'organisation **relative**,
- l'organisation **séquentielle indexée**. (Origine : langage COBOL)
- l'organisation **aléatoire**

L'organisation d'un fichier induit un mode d'accès à ses enregistrements

1.- Introduction aux fichiers –

Organisation logique

❑ Fichier à **organisation séquentielle**

- Ensemble **ordonné** (éventuellement vide) d'éléments de même type rangés de manière contiguë (adjacente – les uns à côté des autres) sur un support physique permanent.



- L'ordre de rangement des enregistrements dans le fichier correspond à **l'ordre de création** des enregistrements
- Sur la plupart des ordinateurs l'entrée-clavier et la sortie-écran sont assimilés à des fichiers séquentiels.
- Supports physiques associés :
Bande magnétique, disque magnétique, disque optique
- L'organisation séquentielle convient aux fichiers textes et binaires

1.- Introduction aux fichiers – Organisation logique

❑ Fichier à organisation **séquentielle**

– Exemples typiques d'utilisation :

- échanges d'informations entre programmes (via fichier xml)
- mise à jour d'un fichier **permanent** à partir d'un fichier de **mouvements**

Principe :

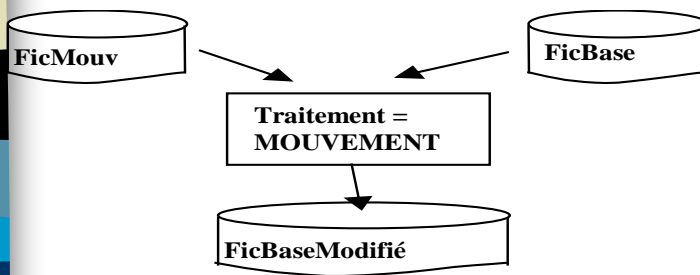


Illustration :

Fichier *fAncien* : 2a | 5b | 6c | 8d | 9e | 15f | 20g |

Fichier des mouvements *fMouvements* :

C3x | M5y | S6 | M9z | 9e |

Légende : codes des mouvements

M : modification ;

S : suppression ;

C : création.

Nouveau fichier *fNouveau* :

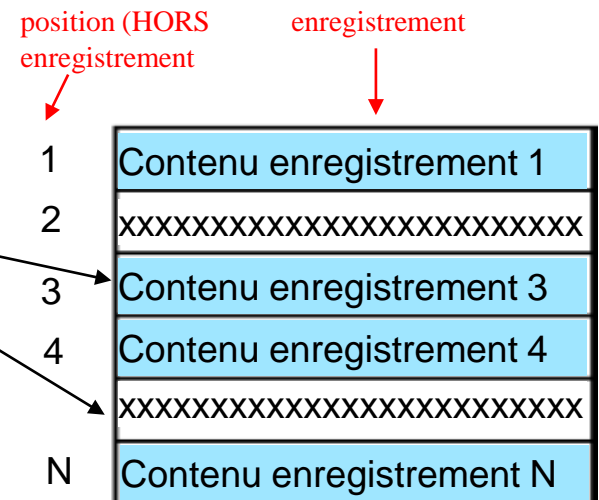
2a | 3x | 5y | 8d | 9z | 15f | 20g |

Le fichier 'mouvement' contient un 'lot' / ensemble de transactions / opérations à réaliser sur le fichier permanent.

1.- Introduction aux fichiers – Organisation logique

❑ Fichier à **organisation relative**

- Les enregistrements sont identifiés par un numéro d'ordre (ordre **relatif** par rapport au début du fichier) démarrant à 1.
- L'enregistrement de numéro d'ordre **n** occupe la $n^{\text{ième}}$ position dans le fichier.
- Chaque 'case' numérotée, contient un enregistrement ou rien : il peut y avoir des 'trous'
- Assimilables à des tableaux sur disque
- Supports physiques associés :
Disques magnétiques, disques optiques
- Les fichiers relatifs sont de type texte ou binaire **mais** utilisation limitée de l'accès direct pour les fichier texte



1.- Introduction aux fichiers – Mode d'accès

□ **Mode d'accès** à un fichier (= aux enregistrements du fichier)

Il désigne *la façon dont le système d'exploitation accède aux enregistrements de ce fichier.*

Les modes d'accès disponibles dépendent de l'organisation logique du fichier.

==> la liste des opérations disponibles pour 'manipuler' un fichier depuis un programme dépend de l'organisation logique du fichier.

L'**accès** à un enregistrement peut se faire :

- **séquentiellement** : à partir de l'enregistrement précédent
- **directement** en donnant la **position** d'un enregistrement
- **selon une clé** : l'adresse sur le disque dépend d'un champ particulier de l'enregistrement (clé primaire) :
 - Accès séquentiel indexé, où la correspondance entre clé et adresse est réalisée d'un **index**
 - Accès calculé, si la correspondance est réalisée par le biais d'une **fonction de hashage**.

1.- Introduction aux fichiers – Mode d'accès

□ Principe **d'accès séquentiel**

– Il est basé sur 3 mécanismes :

- mécanisme d'accès au premier élément du fichier,
- mécanisme mettant en œuvre la relation de succession existant entre les éléments et définie par : "tout élément, sauf le dernier, a un successeur dans le fichier",
- mécanisme indiquant quel est le dernier élément (il n'aura pas de successeur).

=> l'accès à l'élément situé au rang (n) se fera à partir de l'élément situé au rang (n-1).

– Primitives d'Entrée/Sortie les plus répandues

- **LireSuivant** : primitive d'entrée MS→MC transférant en MC l'enregistrement suivant de l'enregistrement courant,
- **ÉcrireSuivant** : primitive de sortie MC→MS transférant le contenu d'une fiche en MS, à l'emplacement suivant celui de l'enregistrement courant
- **RéécrireCourant** : primitive de sortie MC→MS transférant le contenu d'une fiche en MS, à l'emplacement de l'enregistrement courant

– Définition : **enregistrement courant**

C'est l'enregistrement ayant fait l'objet de la dernière opération d'Entrée/Sortie dans le fichier.

1.- Introduction aux fichiers –

Mode d'accès

❑ Principe **d'accès direct**

L'accès aux enregistrements peut se faire directement (sans avoir à passer par les autres enregistrements du fichier) grâce à la **position relative** occupée par chaque enregistrement dans le fichier.

Primitives d'Entrée/Sortie les plus répandues

- **LireFicheNuméro** : primitive d'entrée MS→MC transférant en MC l'enregistrement dont la position relative est signifiée par un paramètre de la primitive,
- **ÉcrireFicheNuméro** : primitive de sortie MC→MS transférant le contenu d'une fiche en MS, à la position relative du fichier désignée par un paramètre de la primitive
- **SupprimerFicheNuméro** : supprime du fichier la fiche dont la position relative est désignée par un paramètre de la primitive. Il s'agit d'une suppression 'logique', à savoir que l'emplacement n'est pas détruit, mais rendu inaccessible.

1.- Introduction aux fichiers –

Organisation logique & Mode d'accès

- ❑ **Lien** entre **organisation logique** et **mode d'accès** à un fichier

L'organisation d'un fichier **induit** un mode d'accès à ses éléments :

		Accès		
		séquentiel	direct	par clé
Organisation	séquentielle	x		
	relative	x	x	
	séqu. Indexée	x		x
	aléatoire			x



1.- Introduction aux fichiers –

Conclusion : Organisations Relative versus Séquentielle

- ❑ À titre de conclusion, comparons :

Fichier à **Organisation relative**

versus

Fichier à **Organisation séquentielle**



1.- Introduction aux fichiers –

Conclusion : Organisations Relative versus Séquentielle

❑ Fichier à **Organisation relative**

- Deux type d'accès possible :
 - accès **séquentiel** (comme les fichiers séquentiels)
 - accès **direct**, si l'on connaît le numéro d'ordre (position relative) de l'enregistrement
- Avantages de l'accès direct
 - rapidité de recherche
 - lecture et écriture n'importe où dans le fichier

1.- Introduction aux fichiers –

Conclusion : Organisations Relative versus Séquentielle

❑ Fichier à **Organisation Séquentielle**

– Avantages :

- simples, universels (surtout les fichier texte !)
- rapidité d'accès à l'enregistrement suivant
- efficaces dans les cas où les traitements nécessitent de lire ou d'écrire tous (ou une grande partie) les enregistrements du fichier.

– Inconvénients :

- l'accès à un enregistrement suppose la lecture de tous les enregistrements qui le précèdent.
- la mise à jour d'un fichier séquentiel nécessite (presque) toujours la construction d'un autre fichier.
- Exemple : l'insertion d'un enregistrement (

Bruyère	Marie
---------	-------

) nécessite la recopie de tout le fichier dans un second fichier. (Idem pour la suppression) :

<u>Albiztur</u>	Adeline
<u>Bélascaïn</u>	Roger
Bologne	Marie
<u>Dalmau</u>	Marc

<u>Albiztur</u>	Adeline
<u>Bélascaïn</u>	Roger
Bologne	Marie
<u>Bruyère</u>	Marie
<u>Dalmau</u>	Marc

2.- Programmer avec des fichiers – Éléments de programme nécessaires

❑ Éléments d'un programme nécessaires à l'écriture de programmes utilisant des fichiers

- **Nom logique** du fichier

variable du programme

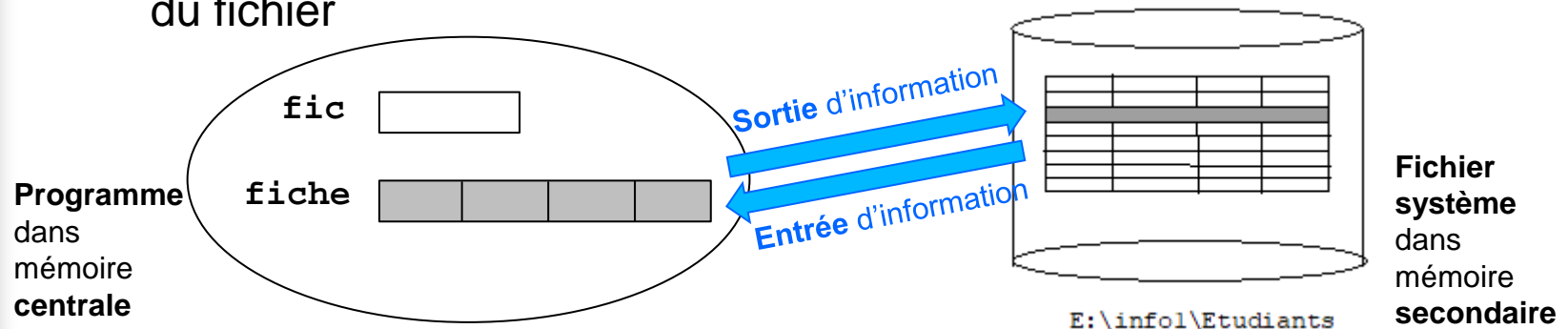
Elle **représente** le fichier dans le programme (le fichier ne bouge pas de la mémoire secondaire)

- **Fiche**

variable du programme

C'est l'**unité minimale de transfert d'information** entre le fichier et le programme.

Elle a donc la même structure (→ même type) que les enregistrements du fichier



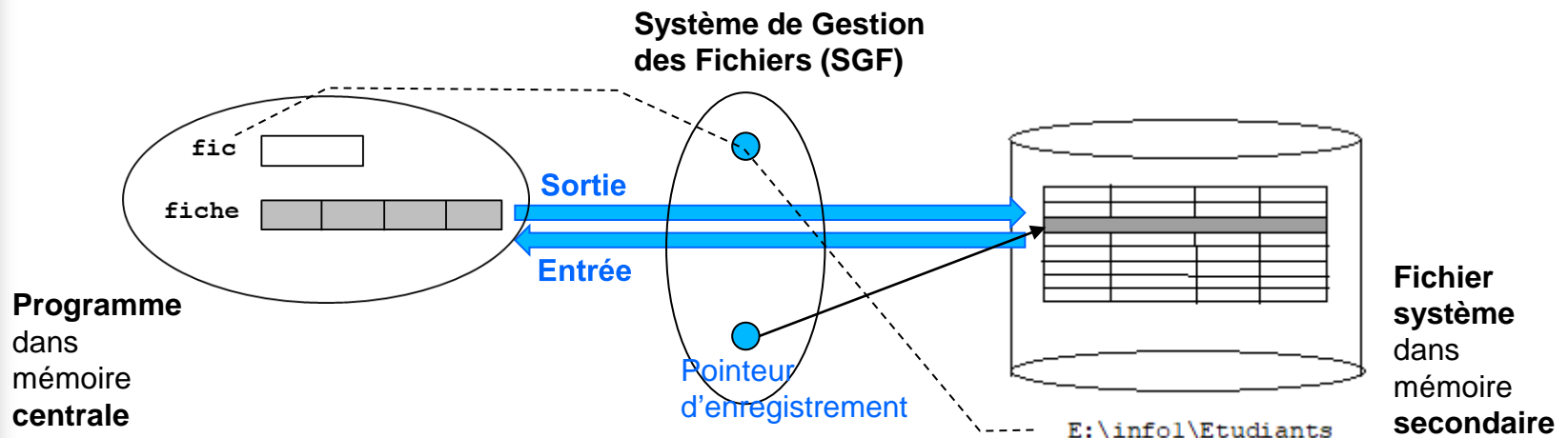
2.- Programmer avec des fichiers – Éléments de programme nécessaires

❑ Fonctionnement des Entrées/Sorties d'informations durant l'exécution du programme

Grâce à des **primitives spécifiques** du programme, le *Système de Gestion de Fichiers* (SGF) :

- effectue le lien entre le **nom logique** et le **nom système** du fichier (= le nom du fichier sur le disque, chemin d'accès inclus).
- effectue les opérations d'Entrée/Sortie (= Entrée ou Sortie d'information) sur le fichier.

Pour ce faire, il dispose d'un **pointeur d'enregistrement** indiquant à tout moment quel est l'**enregistrement courant** du fichier, c'est à dire l'enregistrement du fichier sur lequel a eu lieu la dernière opération d'E/S.



2.- Programmer avec des fichiers – Primitives spécifiques nécessaires

□ Instructions spécifiques de gestion / manipulation de fichiers

Le langage de programmation doit disposer des primitives suivantes :

– Primitives de **préparation** du fichier aux Entrées / Sorties d'Information

- **Associer** relie le fichier logique au fichier système
- **Ouvrir** le fichier dans un certain **mode**
rend le fichier disponible aux opérations autorisées dans ce mode
- **Fermer** rend le fichier **IN**disponible à toute autre opération d'E/S

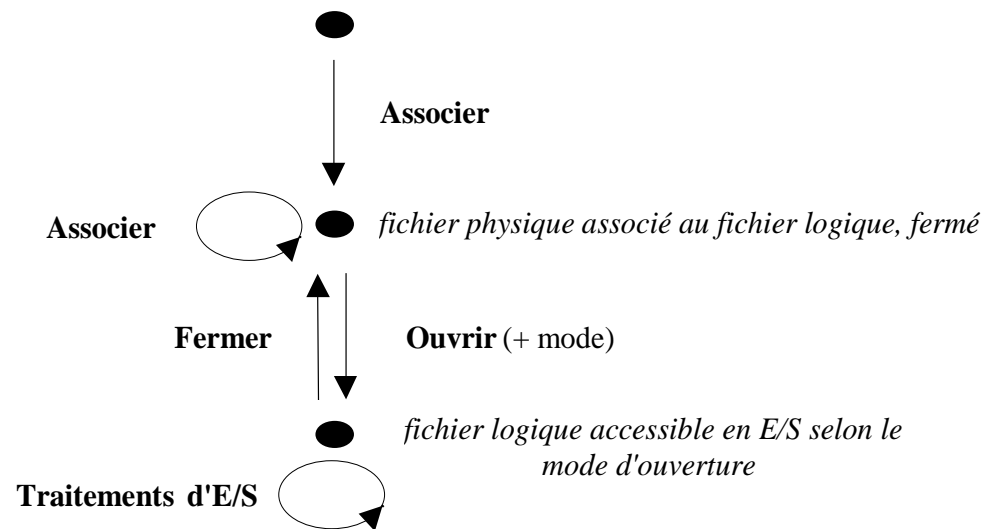
– Primitives d'**Entrée et de Sorties** d'Information

Elles se chargent des **échanges d'informations** entre l'**enregistrement courant** (du fichier système) et la **variable *fiche*** du programme, **dans le respect du **mode**** d'échange préalablement précisé.

- **Lire** transfère le contenu de l'enregistrement courant du fichier **vers** la mémoire centrale, dans la variable ***fiche***,
puis fait avancer le pointeur d'enregistrement
- **Écrire** transfère le contenu de la variable ***fiche*** **vers** le fichier, à l'endroit pointé par le pointeur d'enregistrement,
puis fait avancer le pointeur d'enregistrement
- **Réécrire** **place** le pointeur d'enregistrement à l'endroit de la dernière opération d'Entrée / Sortie,
puis transfère le contenu de la variable ***fiche*** **vers** le fichier, à l'endroit pointé par le pointeur d'enregistrement
= met à jour l'enregistrement courant

2.- Programmer avec des fichiers – Graphe de précedence entre primitives

- ❑ **Graphe de précedence** des primitives de gestion d'un fichier
 - Il décrit l'ordre dans lequel peuvent/doivent être réalisées les primitives disponibles sur les fichiers



- Exemples
 - Séquences **autorisées** :
 - associer, ouvrir, E/S, E/S, E/S, fermer associer, ouvrir, fermer
 - associer, ouvrir, E/S, fermer, associer, ouvrir, E/S, E/S..., E/S, fermer
 - Séquences **non autorisées** :
 - associer, E/S, E/S, E/S, fermer ouvrir, E/S, fermer

2.- Programmer avec des fichiers – Mode d'ouverture d'un fichier

❑ **Mode d'ouverture** d'un fichier

– Il détermine le type d'actions qui seront faites sur le fichier :

- Ouvert en mode **Consultation**:

- seules des lectures seront autorisées sur le fichier
- on parlera de **fichier d'Entrée**

- Ouvert en mode **Écriture** :

- opérations d'ajout de nouvelles fiches autorisées
- on parlera de **fichier de Sortie**
- variantes possibles
 - **Création** : S'il n'existait pas lors de l'ouverture, le fichier est créé vide. S'il existait, son contenu antérieur à l'ouverture est effacé et le fichier est vide suite à l'ouverture.
 - **Extension** : S'il n'existait pas lors de l'ouverture, le fichier est créé vide. S'il existait, son contenu antérieur à l'ouverture est préservé suite à l'ouverture et de nouvelles fiches seront rajoutées.

- Ouvert en mode **Consultation et Écriture simultanées** (ou mode **Modification**):

- autorisation de lire ET écrire → cad de **modifier** des fiches existantes

3.- Fichiers et C++ :

Flots E/S conversationnels `cin` et `cout`

❑ Entrées-Sorties conversationnelles : les flots `cin` et `cout`

- Le noyau du langage C++ ne définit pas la manière dont les informations sont écrites sur l'écran ou lues depuis le clavier.
- Ces E/S conversationnelles (pour échanger avec l'utilisateur humain) sont gérées de manière séparée par la **bibliothèque** `iostream` (Input/Output Stream)
- La séparation E/S (bibliothèque) <--> noyau du langage aide à la modularité et à la **portabilité** du langage, et donc à la qualité du programme écrit.
- La bibliothèque `iostream` gère des échanges d'informations *textuels* (*) entre :
 - le programme et l'écran
 - le clavier et le programme

comme **une suite séquentielle d'octets**, passant par deux canaux, un pour les sorties d'informations, un pour les entrées d'informations.

(*) pourquoi textuels ?

3.- Fichiers et C++ : Flots d'E/S

□ Généralisation : Flots d'Entrée-Sortie

- Un flot peut être considéré comme un **canal** :
 - recevant séquentiellement de l'information du programme qui **sort** du programme : **flot de sortie**
 - fournissant séquentiellement de l'information en **entrée** du programme : **flot d'entrée**
- Avantages des flots :

Le **transit** des informations entre programme et périphériques d'E/S est **encapsulé** (enveloppé) dans le flot :

 - Quand le flot est créé, le programme échange **avec le flot**, et c'est ce dernier qui règle les détails des échanges avec le périphérique d'E/S.
 - La gestion du transit est donc simplifiée pour le programme
- Opérateurs **>>** et **<<**

Il servent à assurer le transfert de l'information ainsi que son éventuel formatage, entre le programme et le flot
- Un flot peut être connecté à un périphérique (d'E/S) ou à un disque

3.- Fichiers et C++ : Flots d'E/S

❑ Entrées / Sorties tamponnées

Les E/S sur certains périphériques (disques) sont relativement lentes et freinent souvent l'exécution d'un programme ==> usage de **tampons**

Tampon / buffer :

- Zone mémoire (en mémoire centrale) vers laquelle le contenu du flux est momentanément enregistré (on parle de contenu tamponné ou '*bufferisé*'), par exemple, dans l'attente d'être transféré vers un périphérique
- L'écriture vers le périphérique se fera en une seule fois lorsque le tampon sera plein
==> le programme n'est pas ralenti par l'attente du remplissage de la zone tampon
- Possibilité de 'purger' le tampon sans attendre qu'il soit plein
- Danger : perte des données qui sont dans le tampon (avant écriture sur le disque) lors d'un 'plantage' du programme

3.- Fichiers et C++ :

Flots d'E/S

❑ Bibliothèques pour la gestion des flots

- Bibliothèque `iostream` : gère les flux d'E/S conversationnels (clavier -> programme -> écran) **de manière séquentielle**.

Elle met à disposition plusieurs flots prédéfinis :

- `cin` : gère les entrées en provenance de l'entrée standard : le clavier
- `cout` : gère les sorties vers la sortie standard : l'écran
- `cerr` : gère les sorties vers l'unité d'erreur standard : l'écran. Ces informations ne sont pas bufferisées, elles sont directement affichées.
- `clog` : gère aussi les sorties vers l'unité d'erreur standard : l'écran, mais les informations sont bufferisées.

- Bibliothèque `fstream` : gère les flots d'E/S avec les **fichiers (File)**.

- Ces 2 bibliothèques sont à inclure dans nos programmes (`#include`)

```
#include <iostream>
#include <fstream>      // à rajouter pour traiter des fichiers

using namespace std;
int main()
{
    ...
    return 0;
}
```

3.- Fichiers et C++ :

Flots d'E/S : bibliothèque `fstream`

- **Bibliothèque `fstream` gérant les flots d'E/S connectés à des fichiers**
 - Elle permet de gérer des fichiers textes **et** des fichiers binaires
 - Elle autorise :
 - Un accès séquentiel dans les fichiers texte
 - Un accès séquentiel **et** direct dans les fichiers binaires
 - L'usage de la bibliothèque `fstream` supposant la maîtrise de la programmation objet, nous allons aborder la manipulation des fichiers de manière simplifiée :
 - Nous n'aborderons que les fichiers texte (accès séquentiel seul)
 - Nous utiliserons un ensemble **restreint** de primitives de gestion de fichiers texte
 - Nous utiliserons le style de programmation procédural (non objet) habituel utilisé depuis le début de la ressource R1.01
- ➔ Ce type et ensemble de primitive ont été regroupés dans un type abstrait de données simplifié nommé `UnFichierTexte`

4.- Fichiers texte avec C++ et le TAD `UnFichierTexte` :

Type Abstrait de Données(*) `UnFichierTexte`

- ❑ Il permet une gestion simplifiée de fichiers de texte dans le cadre de ce cours.
- ❑ Il met à disposition :
 - Un **type** `UnFichierTexte` permettant de déclarer des **fichiers de texte (à accès séquentiel)**

Pour information, sa structure est la suivante :

```
enum UnModeOuverture {consultation, creation, extension};
struct UnFichierTexte
{
    string nom;                                // nom système du fichier
    UnModeOuverture modeOuverture;             // précise le mode d'ouverture : consultation,
    bool modeOuvertureDefini;
    fstream donnees;                            // la variable fichier de la bibliothèque
                                              // de référence fstream
} ;
```

- Un ensemble de **primitives** permettant de manipuler des fichiers de type `UnFichierTexte`
- ❑ Ce TAD est disponible dans `fichierTexte.h` et `fichierTexte.cpp`

5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ Déclaration d'une variable fichier texte :

```
#include <iostream>
#include "fichierTexte.h"

using namespace std;
int main()
{

    UnFichierTexte monFichier ;

return 0;
}
```

- la variable `monFichier` correspond au **nom logique** du fichier
- on peut expliciter dans le nom logique les opérations qui seront effectuées sur le fichier :

```
UnFichierTexte monFicE ; // pour fichier d'Entrée
UnFichierTexte monFicS ; // pour fichier de Sortie
```

5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ **Associer** un nom système à un fichier (à un nom logique)

- Entête primitive

```
void associer ( UnFichierTexte& f,  
               string nom);
```

Effet

Relie le **nom logique** d'un fichier (*variable `f` représentant le fichier dans le programme*) à son **nom système** (*variable `nom` représentant le nom du fichier sur le disque*)

- Exceptions levées
néant
- Exemple d'appel

```
string nomSysE = "dec-2019" ;    // dans répertoire courant (*)  
string nomSysS = "..\dec-2019.txt" ; // avec chemin d'accès  
  
associer (monFicE, nomSysE);  
associer (monFicS, nomSysS);  
associer (monFichier, "..\dec-2019.txt") ;
```

** précisez*

5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ Ouvrir un fichier

– Entête primitive

```
void ouvrir ( UnFichierTexte& f,  
             UnModeOuverture mode );
```

Effet

- en mode `consultation`:
 - Rend le fichier de nom logique `f` disponible pour un accès en lecture.
 - Le pointeur d'enregistrement se positionne en début de fichier pour commencer la lecture, et avance *automatiquement* après la lecture de chaque enregistrement.
- en mode `creation` :
 - S'il n'existait avant l'ouverture, le fichier de nom logique `f` est créé vide, puis est rendu disponible pour y écrire.
 - S'il existait, son contenu antérieur à l'ouverture est effacé et le fichier est vide suite à l'ouverture.
 - Toute information nouvelle ajoutée sera ajoutée en fin de fichier
- en mode `extension` :
 - S'il n'existait pas lors de l'ouverture, le fichier de nom logique `f` est créé vide.
 - S'il existait, son contenu antérieur à l'ouverture est préservé, et le fichier est rendu disponible pour y écrire.
 - Toute information nouvelle ajoutée sera ajoutée en fin de fichier

5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ Ouvrir un fichier

– Exceptions levées

- `erreurDeStatut` : si le fichier est déjà ouvert
- `erreurDeNomOuUsage` : si le fichier n'a pu être ouvert
 - `DeNom`, si le nom est illégal ou le fichier inexistant,
 - `DUsage`, si les protections du fichiers rendent l'opération illégale

– Exemple d'appel

```
ouvrir(monFicE, consultation);  
ouvrir(monFicS, creation);  
ouvrir(monFichier, extension);
```

5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ **Fermer un fichier**

- Entête primitive

```
void fermer ( UnFichierTexte& f );
```

Effet

Rend le mode d'ouverture du fichier indéfini, et donc le fichier **indisponible** pour toute opération d'Entrée/Sortie

- Exceptions levées

- `erreurDeStatut` : si le fichier n'est pas ouvert
- `erreurInconnue` : si la fermeture n'a pu se faire (fichier corrompu...)

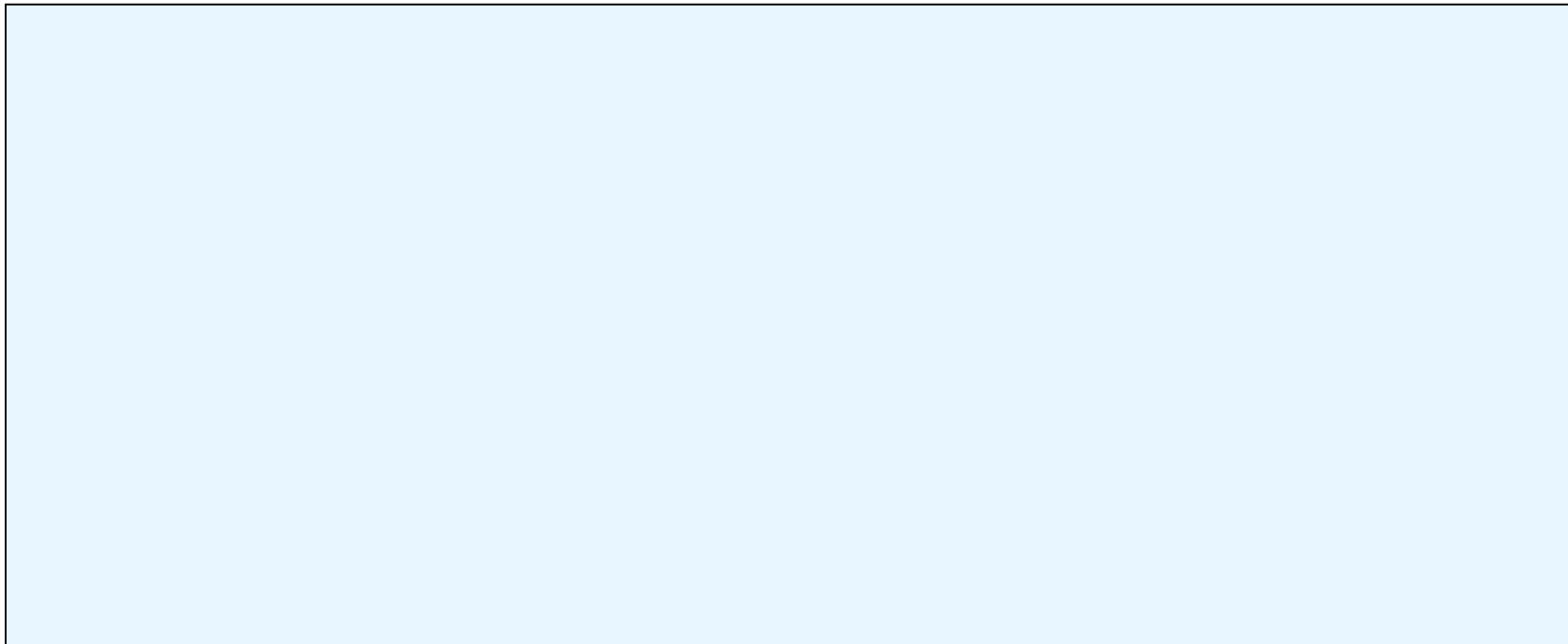
- Exemple d'appel

```
fermer (monFicE);  
fermer (monFicS);  
fermer (monFichier);
```



5.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Déclarer, Associer, Ouvrir, Fermer

❑ Exercice récapitulatif 1

- Écrire un programme complet qui crée un fichier vide, de nom "rien.txt", dans le répertoire courant du programme
- Code :



- Résultat attendu

Nom	Taille	Type
 ExpleRecapitulatifCreationFichierVide.exe	947 Ko	Application
 rien.txt	0 Ko	Document texte

5.- Fichiers texte avec C++ et le TAD **UnFichierTexte** : Déclarer, Associer, Ouvrir, Fermer

❑ Exercice récapitulatif 1

- Écrire un programme complet qui crée un fichier vide, de nom "rien.txt"

```
#include <iostream>
#include "fichierTexte.h"
using namespace std;

int main()
{
    UnFichierTexte monFichier ;          // nom logique du fichier

    associer (monFichier, "rien.txt"); // associer nom logique et nom système

    ouvrir (monFichier, creation);       // ouverture du fichier

    fermer(monFichier);                  // fermeture du fichier

    return 0;
}
```

- Résultat produit

Nom	Taille	Type
ExpleRecapitulatifCreationFichierVide.exe	947 Ko	Application
rien.txt	0 Ko	Document texte

6.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Écrire dans un fichier texte

- ❑ **Plusieurs primitives d'écriture**, dont certaines *surchargées*
 - Entête primitives

```
void ecrire ( UnFichierTexte& f, string item);  
void ecrire ( UnFichierTexte& f, char item);  
void ecrire ( UnFichierTexte& f, int item);  
void ecrire ( UnFichierTexte& f, float item);  
void ecrire ( UnFichierTexte& f, bool item);  
void ecrireLigne ( UnFichierTexte& f, string item);
```

Effet :

- Pour toutes les primitives : dans les modes *creation* et *extension*, le contenu du paramètre `item` est enregistré en fin du fichier ayant pour nom logique `f`
- Primitive *ecrireLigne* : le paramètre `item` est enregistré en fin de fichier de nom logique `f`, suivi d'un caractère finDeLigne (`'\n'`) – hexa:0D0A
- Exceptions levées
 - *erreurDeMode* : si le mode d'ouverture du fichier interdit l'écriture
 - *erreurInconnue* : si l'écriture n'a pu se faire ou s'est mal déroulée (plus d'espace disque, fichier corrompu...)

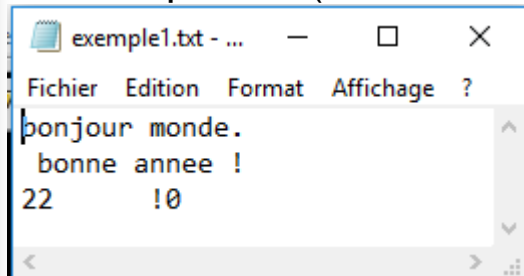
6.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Écrire dans un fichier texte

❑ Plusieurs primitives d'écriture

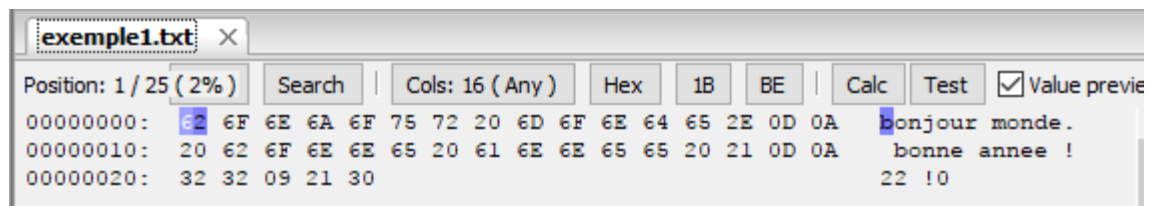
– Exemple 1 – appels des primitives `ecrire`

```
char tab = '\\t';  
char finDeLigne = '\\n';  
string chaineBonneAnnee = "bonne annee " ;  
  
// on suppose le fichier ouvert en création ou extension  
ecrire (monFichier, "bonjour ");  
ecrireLigne (monFichier, "monde.");  
ecrire (monFichier, ' ');  
ecrire (monFichier, chaineBonneAnnee);  
ecrire (monFichier, '!');  
ecrire (monFichier, finDeLigne);  
ecrire (monFichier, 22);  
ecrire (monFichier, tab);  
ecrire (monFichier, '!');  
ecrire (monFichier, false);
```

Résultat produit (fichier texte)



Vue hexadécimale du fichier texte



7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

- 3 primitives, en fonction de la manière de lire :

- **lecture *ligne à ligne***

Extraction du fichier de chaînes (lignes) séparées un caractère **délimiteur**
`finDeLigne` (`'\n'`) – hexa:0D0A

- **lecture *mot à mot***

- Extraction du fichier une série de caractères séparés par un (ou plusieurs) **délimiteur(s)**, en appliquant le formatage inverse de la primitive **ecrire**, c'est-à-dire en interprétant le type de ce qui est lu (nombre entier, réel, chaîne, caractère, booléen).
- Les délimiteurs par défaut utilisés sont :
 - les « espaces blancs » servent de séparateurs/délimiteurs.
Est considéré comme « espace blanc » : espace, tabulations horizontale `'\t'` – hexa:09 - et verticale `'\v'`, fin de ligne `'\n'` et saut de page `'\f'`
 - toute lecture commence par sauter ces séparateurs/délimiteurs s'ils existent

Conséquence : les délimiteurs ne peuvent pas être lus en tant que caractères avec cette primitive de lecture. Utiliser une autre opération de lecture.

- **lecture *caractère à caractère***

La lecture ne s'arrête que lorsqu'elle trouve la `finDeFichier`

7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

□ Lecture *ligne à ligne*

```
void lireLigne ( UnFichierTexte& f,  
                string& chaine,  
                bool& finFichier);
```

Effet

- Disponible pour le mode `consultation`,
- 2 cas de retour
 - S'il y a une ligne suivante (chaîne suivie d'un caractère finDeLigne ou finDeFichier) :
 - son contenu est affecté au paramètre `chaîne` (**sans** le caractère délimiteur)
 - le paramètre `finFichier` est retourné à Faux
 - S'il n'y a pas de ligne suivante (uniquement finDeFichier trouvée) :
 - le paramètre `chaîne` n'est pas modifié
 - le paramètre `finFichier` est retourné à Vrai
- Lorsque le caractère finDeLigne est trouvée, il n'est pas ajouté dans la chaîne `chaîne` et la prochaine opération de lecture dans le fichier démarrera après lui.

À retenir :

La primitive `lireLigne` est une **tentative de lecture**, qui peut réussir ou échouer.
La réussite ou échec est exprimée dans le paramètre résultat booléen `finFichier`

7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

❑ Lecture *ligne à ligne*

– Exemple 2 – appel de la primitive `lireLigne`

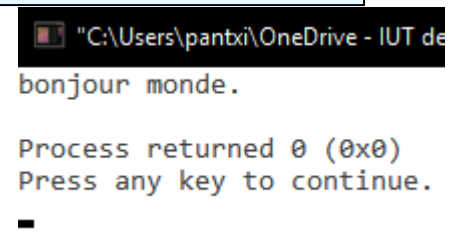
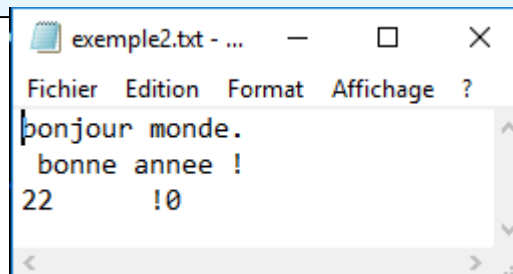
```
UnFichierTexte monFicE;  
string maLigne;  
bool fdf;  
  
// ... on suppose le fichier ouvert en consultation  
  
// monFicE >> tentative de lecture >> fdf, [maLigne]  
lireLigne (monFicE, maLigne, fdf);  
if (fdf)  
{ // ce qu'il faut faire si la lecture a échoué car fin de fichier atteinte  
  cout << "fin de fichier atteinte" << endl;  
}  
else  
{ // ce qu'il faut faire avec le contenu de maLigne  
  cout << maLigne;  
}
```

Pour ce fichier texte

Résultat d'exécution

❑ Exception levées

- `erreurDeMode`
- `erreurInconnue`



7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

❑ Lecture *mot à mot*

Plusieurs primitives *surchargées*

```
void lireMot ( UnFichierTexte& f, string& item, bool& finFichier);  
void lireMot ( UnFichierTexte& f, int& item, bool& finFichier);  
void lireMot ( UnFichierTexte& f, float& item, bool& finFichier);  
void lireMot ( UnFichierTexte& f, bool& item, bool& finFichier);
```

Effet :

- Pour toutes les primitives : disponible pour le mode *consultation*,
- 2 cas de retour
 - S'il y a un mot suivant (chaîne suivie d'un délimiteur ou d'une finDeLigne) :
 - son contenu est affecté au paramètre *item* (sans le caractère délimiteur)
 - le paramètre *finFichier* est retourné à Faux
 - S'il n'y a pas de mot (uniquement finDeFichier trouvée) :
 - le paramètre *item* n'est pas modifié
 - le paramètre *finFichier* est retourné à Vrai
- Lorsque le caractère délimiteur est trouvé, la prochaine opération de lecture dans le fichier démarrera après lui.

À retenir :

La primitive *lireMot* est une **tentative de lecture**, qui peut réussir ou échouer.
La réussite ou échec est exprimée dans le paramètre résultat booléen *finFichier*

7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

❑ Lecture *mot à mot*

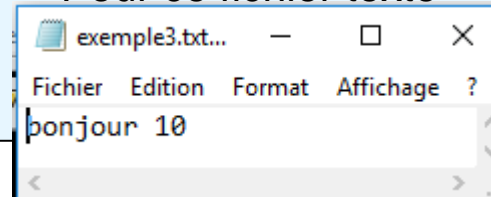
– Exemple 3 – appels de de la primitive `lireMot`

```
UnFichierTexte monFichier ; // nom logique du fichier
int monNbre ;      string maChaine ;      bool fdf;

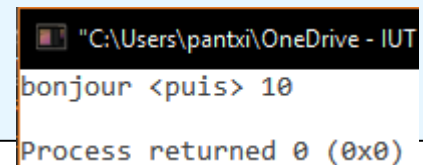
// créer et peupler le fichier
associer (monFichier, "exemple3.txt"); ouvrir (monFichier, creation);
ecrire (monFichier, "bonjour"); ecrire (monFichier, '\t'); ecrire (monFichier, 10);
fermer (monFichier);

// consulter le fichier
ouvrir (monFichier, consultation);
lireMot (monFichier, maChaine, fdf);
if (!fdf)
{ lireMot (monFichier, monNbre, fdf);
  if (!fdf)
    { cout << maChaine << " <puis> " << monNbre << endl ; }
  else { //... }
}
fermer(monFichier);
```

Pour ce fichier texte



Résultat d'exécution



❑ Exception levées

- **erreurDeMode** : si le mode d'ouverture interdit la lecture,
- **erreurInconnue** : si la lecture s'est mal déroulée (type de l'item non adéquat à la nature de l'information lue, fichier corrompu...)

7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte

❑ Lecture *caractère à caractère*

```
void lireCar ( UnFichierTexte& f, char& item, bool& finFichier);
```

Effet

- Disponible pour le mode `consultation`,
- 2 cas de retour
 - S'il y a un caractère suivant :
 - son contenu est affecté au paramètre `item`
 - le paramètre `finFichier` est retourné à Faux
 - S'il n'y a pas de caractère suivant (uniquement `finDeFichier` trouvée) :
 - le paramètre `item` n'est pas modifié
 - le paramètre `finFichier` est retourné à Vrai

À retenir :

La primitive `lireCar` est une **tentative de lecture**, qui peut réussir ou échouer.
La réussite ou échec est exprimée dans le paramètre résultat booléen `finFichier`

❑ Exception levées

- `erreurDeMode` : si le mode d'ouverture interdit la lecture,
- `erreurInconnue` : si la lecture s'est mal déroulée (fichier corrompu...)

7.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Lire dans un fichier texte



Attention !
Éviter le mélange de lectures

mot à mot* et *ligne à ligne

- Car les délimiteurs de mots et de lignes sont différents
- **A notre niveau**, nous ne traiterons pas la lecture 'mélangée' de ces 2 types d'enregistrements

8.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Autres primitives

❑ Observateur `estOuvert`

- Entête primitive

```
bool estOuvert (UnFichierTexte& f);
```

Effet :

Retourne

- `vrai` si le fichier de nom logique `f` est ouvert,
- `faux` dans tous les autres cas

- Exceptions levées
néant

- Exemple d'appel

```
if (estOuvert(monFichier))  
    { cout << "fichier ouvert" << endl ; }  
else  
    { cout << "fichier non prêt pour des E/S" << endl ; }
```

8.- Fichiers texte avec C++ et le TAD **UnFichierTexte** : Autres primitives

❑ Observateur **nomSysteme**

- Entête primitive

```
string nomSysteme (UnFichierTexte& f);
```

Effet :

- si le fichier de nom logique *f* est associé à un fichier sur disque, retourne une chaîne de caractères contenant le nom système du fichier,
- Sinon retourne une chaîne vide

- Exceptions levées
néant

- Exemple d'appel

```
if (estOuvert(monFichier))  
    { cout << "fichier " << nomSysteme (monFichier) << " ouvert" << endl ; }  
else  
    { cout << "fichier " << nomSysteme (monFichier)  
      << " non prêt pour des E/S" << endl ;  
    }
```


8.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Autres primitives

❑ Modifier le nom système d'un fichier

– Entête primitive

```
void renommer (UnFichierTexte& f, string nouveauNomSys);
```

Effet :

Opération effectuée *directement sur le fichier* situé sur le disque, sans utilisation d'un flot C++.

- si `nouveauNomSys` précise une nouvelle localisation –répertoire- du fichier de nom logique `f`, cad une valeur différente de `nomSysteme(f)`, le système déplace le fichier vers le nouvel emplacement
- si `nouveauNomSys` fait référence à un fichier existant, la primitive peut échouer ou écraser le fichier existant, cela dépend du système d'exploitation

Le fichier ***doit être fermé*** pour que l'opération réussisse.

– Exceptions levées

- `erreurInconnue` : si l'opération a échoué, quelle que soit l'erreur

– Exemple d'appel

```
fermer(monFichier);  
renommer(monFichier, "old_"+ nomSysteme(monFichier));
```

8.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Autres primitives

❑ Supprimer un fichier sur le disque

- Entête primitive

```
void supprimer (UnFichierTexte& f);
```

Effet :

Opération effectuée *directement sur le fichier* situé sur le disque, sans utilisation d'un flot C++.

Supprime physiquement, c'est-à-dire sur le disque, le fichier associé au nom logique `f`.

Le fichier ***doit être fermé*** pour que l'opération réussisse.

- Exceptions levées

- `erreurInconnue` : si l'opération a échoué, quelle que soit l'erreur

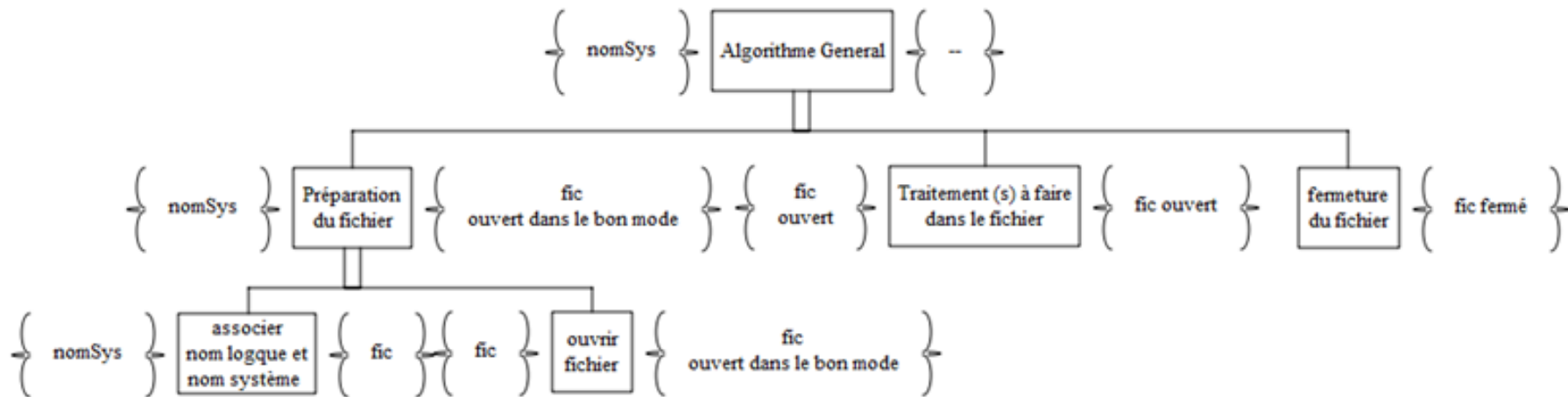
- Exemple d'appel

```
fermer(monFichier);  
supprimer(monFichier);
```

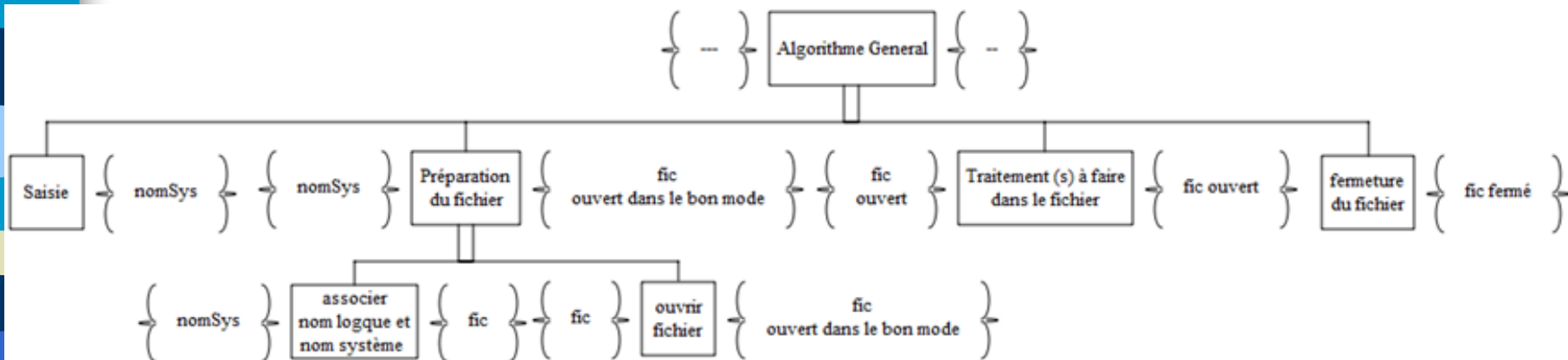
9.- Fichiers texte avec C++ et le TAD `UnFichierTexte` :

Structure d'un programme traitant un fichier texte

❑ **Forme générale** d'un algorithme traitant un fichier texte



ou bien



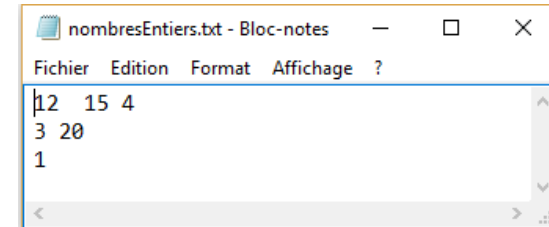
9.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Algorithmes de parcours **séquentiels**

- ❑ **Les fichiers de texte sont séquentiels**
- ❑ Les algorithmes de parcours de fichiers de texte obéissent aux modèles d'algorithmes séquentiels
 - ❑ Parcours séquentiel complet avec traitement obligatoire
 - ❑ Parcours séquentiel complet avec traitement conditionnel
 - ❑ Recherche séquentielle de première occurrence
 - ❑ (Parcours séquentiel parallèle de 2 fichiers triés selon un critère)
- ❑ Les 3 mécanismes de mise en œuvre d'un accès séquentiel aux éléments d'un fichier de texte sont disponibles via les primitives mise à disposition :
 - ❑ Accès au premier élément : primitive **ouvrir**
 - ❑ Accès à l'élément suivant : primitives **lireMot / lireLigne / lireCar**
 - ❑ Mécanisme de détection de la fin de la structure : indicateur **finDeFichier**

9.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Algorithmes de traitement **séquentiels**

❑ Exercice récapitulatif 2

1. Écrire l'algorithme d'un programme qui affiche à l'écran le contenu du fichier de texte (nom système = "nombresEntiers.txt"), constitué de nombre entiers, ainsi que le nombre total de valeurs lues dans le fichier.

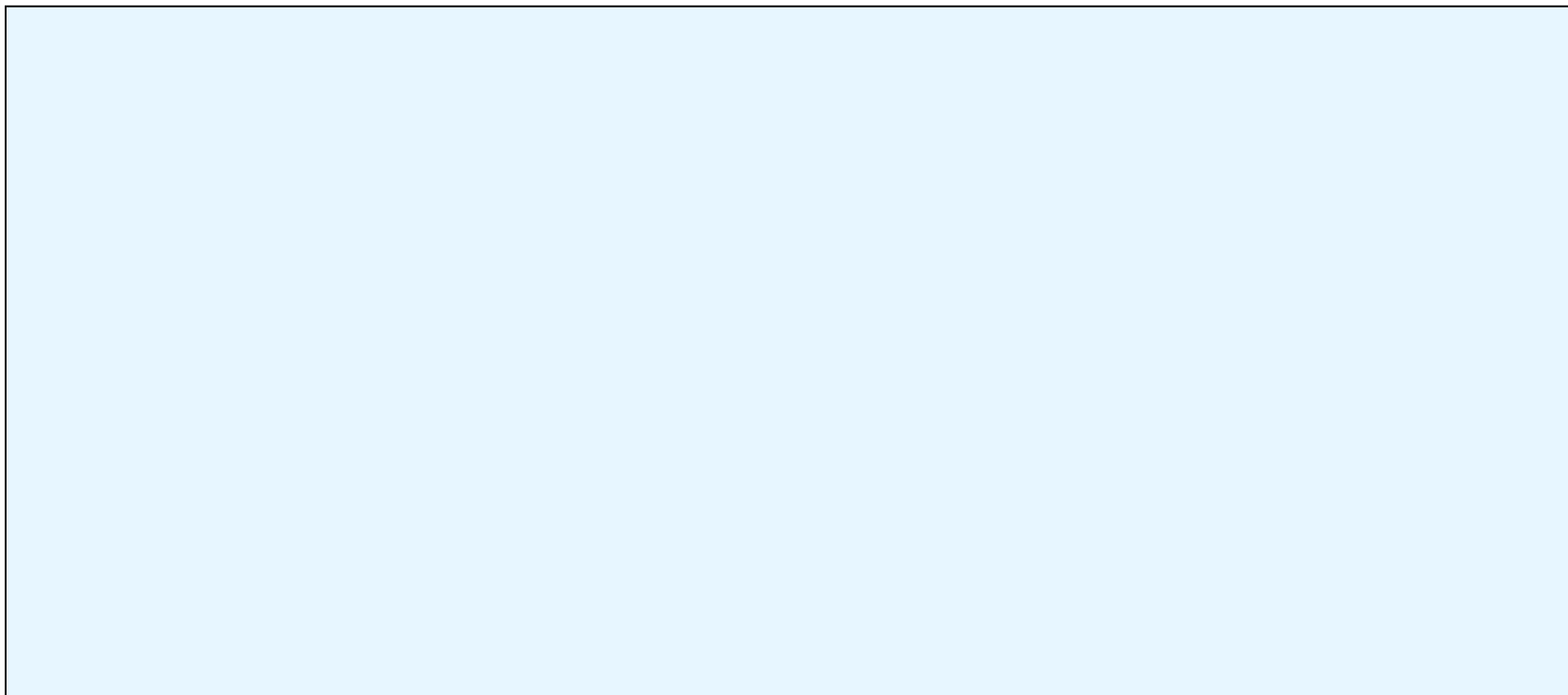


```
nombresEntiers.txt - Bloc-notes
Fichier  Edition  Format  Affichage  ?
12 15 4
3 20
1
```

9.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Algorithmes de traitement **séquentiels**

❑ Exercice récapitulatif 2

2. Écrire un programme complet correspondant à l'exercice précédent. Le fichier de texte contient des entiers et a pour nom système : "nombresEntiers.txt".
 - Code :



- Résultat attendu :

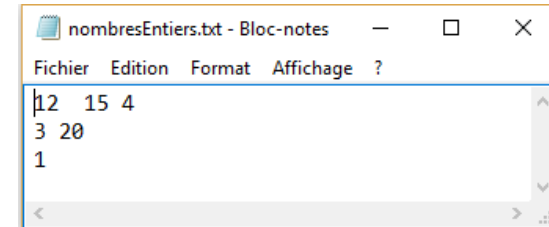
```
12 15 4 3 20 1
Le fichier comporte 6 nombres entiers.

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

9.- Fichiers texte avec C++ et le TAD `UnFichierTexte` : Algorithmes de traitement **séquentiels**

❑ Exercice récapitulatif 2

1. Écrire l'algorithme d'un programme qui affiche à l'écran le contenu du fichier de texte (nom système = "nombresEntiers.txt"), constitué de nombre entiers, ainsi que le nombre total de valeurs lues dans le fichier.



A screenshot of a Notepad window titled "nombresEntiers.txt - Bloc-notes". The window has a menu bar with "Fichier", "Edition", "Format", "Affichage", and "?". The text area contains three lines of integers: "12 15 4", "3 20", and "1".

```
nombresEntiers.txt - Bloc-notes
Fichier  Edition  Format  Affichage  ?
12 15 4
3 20
1
```

```

#include <iostream>

#include "fichierTexte.h"
using namespace std;

int main()
{
    UnFichierTexte ficE;          // nom logique du fichier
    string nomSysFicE = "nombresEntiers.txt";          // nom système du fichier
    int nombre;          // valeur courante lue dans le fichier,
    unsigned int nbre ;          // nbre de valeurs comptées au cours du parcours
    bool fdf ;          // indicateur de fin de fichier

    // association et ouverture du fichier
    associer(ficE, nomSysFicE); ouvrir(ficE, consultation);
    // Parcours séquentiel complet pour consultation
    nbre = 0;
    while (true)
    {
        // tentative de lecture
        lireMot(ficE, nombre, fdf);
        if (fdf)
        { break; }
        // traitement si pas d'erreur de lecture
        = nbre + 1 ;
        cout << nombre << " " ;

    }
    fermer(ficE);

    // nbre >> affichage ecran >> (ecran)
    cout << endl << "Le fichier comporte " << nbre << " nombres entiers." << endl;
    return 0;
}

```


chapitre 5.- Flots et Fichiers

Ressource R1.01 : Initiation au développement - Partie 2

Merci pour votre attention !

```

1  #ifndef FICHIERSTEXTE_H
2  #define FICHIERSTEXTE_H
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8
9  enum UnModeOuverture {consultation, creation, extension};
10 struct UnFichierTexte
11 {
12     string nom; // Nom système du fichier
13     UnModeOuverture modeOuverture; // Précise le mode d'ouverture : consultation,
        creation, ...
14     bool modeOuvertureDefini; // Indique si le mode d'ouverture du fichier est
        défini ou pas
15     fstream donnees; // les données du fichier
16 };
17
18 void associer ( UnFichierTexte& f,
19               string nom);
20 /* relie le NOM LOGIQUE du fichier (ou "fichier logique") à son NOM SYSTEME
21 -- (ou "fichier physique")
22 -- : ne peut occasionner aucune erreur
23 */
24
25 void ouvrir ( UnFichierTexte& f,
26              UnModeOuverture mode);
27 /*rend le fichier disponible pour les Entrées/Sorties autorisées
28 -- : peut occasionner erreurDeStatut si le fichier est déjà ouvert,
29 -- erreurDeNomOuUsage : le fichier n'a pu être ouvert,
30 -- * DeNom, si le nom est illégal ou le fichier inexistant,
31 -- * DUsage, si les protections du fichiers rendent
32 l'opération illégale
33 */
34
35 void fermer ( UnFichierTexte& f);
36 /*rend le mode du fichier indéfini, et donc le fichier indisponible pour toutes
37 -- les Entrées/Sorties
38 -- : peut occasionner erreurDeStatut si le fichier n'est pas ouvert
39 -- : erreurInconnue si la fermeture n'a pu se faire (fichier
        corrompu...)
40 */
41
42 void lireLigne ( UnFichierTexte& f,
43                 string& chaine,
44                 bool& finFichier);
45 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
46 -- S'il y a une ligne suivante (chaine suivie d'un caractère FIN_DE_LIGNE ou
        FIN_DE_FICHIER) :
47 -- son contenu est affecté au paramètre chaine
48 -- le paramètre finFichier est retourné à Faux
49 -- S'il n'y a pas de ligne suivante (uniquement le caractère FIN_DE_FICHIER
        trouvé):
50 -- le paramètre chaine n'est pas modifié
51 -- le paramètre finFichier est retourné à Vrai
52 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
53 -- erreurInconnue si la consultation s'est mal déroulée
54 (fichier corrompu...)
55 */
56
57 void lireCar ( UnFichierTexte& f,
58               char& item,
59               bool& finFichier);
60 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
61 -- S'il y a un item suivant :
62 -- son contenu est affecté au paramètre item
63 -- le paramètre finFichier est retourné à Faux
64 -- S'il n'y a pas d'item (fin de fichier trouvée):
65 -- le paramètre item n'est pas modifié
66 -- le paramètre finFichier est retourné à Vrai

```

```

65     -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
66     --     erreurInconnue si la consultation s'est mal déroulée
    (fichier corrompu...)
67 */
68
69 void lireMot ( UnFichierTexte& f,
70             string& item,
71             bool& finFichier);
72 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
73 --     S'il y a un item suivant (item suivi d'un caractère séparateur) :
74 --     son contenu est affecté au paramètre item
75 --     le paramètre finFichier est retourné à Faux
76 --     S'il n'y a pas d'item (fin de fichier trouvée):
77 --     le paramètre item n'est pas modifié
78 --     le paramètre finFichier est retourné à Vrai
79 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
80 --     erreurInconnue si la consultation s'est mal déroulée
    (fichier corrompu...)
81 */
82
83 void lireMot ( UnFichierTexte& f,
84             int& item,
85             bool& finFichier);
86 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
87 --     S'il y a un item suivant (item suivi d'un caractère séparateur) :
88 --     son contenu est affecté au paramètre item
89 --     le paramètre finFichier est retourné à Faux
90 --     S'il n'y a pas d'item (fin de fichier trouvée):
91 --     le paramètre item n'est pas modifié
92 --     le paramètre finFichier est retourné à Vrai
93 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
94 --     erreurInconnue si la consultation s'est mal déroulée
    (fichier corrompu...)
95 */
96
97 void lireMot ( UnFichierTexte& f,
98             float& item,
99             bool& finFichier);
100 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
101 --     S'il y a un item suivant (item suivi d'un caractère séparateur) :
102 --     son contenu est affecté au paramètre item
103 --     le paramètre finFichier est retourné à Faux
104 --     S'il n'y a pas d'item (fin de fichier trouvée):
105 --     le paramètre item n'est pas modifié
106 --     le paramètre finFichier est retourné à Vrai
107 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
108 --     erreurInconnue si la consultation s'est mal déroulée
    (fichier corrompu...)
109 */
110
111 void lireMot ( UnFichierTexte& f,
112             bool& item,
113             bool& finFichier);
114 /*pour les modes consultation (et Modification NON FAIT), 2 cas de retour
115 --     S'il y a un item suivant (item suivi d'un caractère séparateur) :
116 --     son contenu est affecté au paramètre item
117 --     le paramètre finFichier est retourné à Faux
118 --     S'il n'y a pas d'item (fin de fichier trouvée):
119 --     le paramètre item n'est pas modifié
120 --     le paramètre finFichier est retourné à Vrai
121 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit la consultation,
122 --     erreurInconnue si la consultation s'est mal déroulée
    (fichier corrompu...)
123 */
124
125 void ecrireLigne ( UnFichierTexte& f,
126                 string item);
127 /*pour les modes creation et extension, le contenu du paramètre item
128 -- est enregistré en fin de fichier, suivi d'un caractère FIN_DE_LIGNE
129 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
130 --     erreurInconnue si l'écriture s'est mal déroulée (plus

```

```

    d'espace disque, fichier corrompu...)
131 */
132
133 void ecrire ( UnFichierTexte& f,
134             string item);
135 /*pour les modes creation et extension, le contenu du paramètre item
136 -- est enregistré en fin de fichier
137 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
138 -- erreurInconnue si l'écriture s'est mal déroulée (plus
    d'espace disque, fichier corrompu...)
139 */
140
141 void ecrire ( UnFichierTexte& f,
142             char* item);
143 /*pour les modes creation et extension, le contenu du paramètre item
144 -- est enregistré en fin de fichier
145 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
146 -- erreurInconnue si l'écriture s'est mal déroulée (plus
    d'espace disque, fichier corrompu...)
147
148 ATTENTION
149 Cette version surchargée de la procédure est nécessaire pour pouvoir prendre en
    charge l'appel :
150 ecrire(monFichier, "bonjour");
151     car, par défaut, les constantes littérales chaînes sont considérées comme des
    bool.
152     car C++ considère la constante string littérale comme un char* ... puis
    comme bool (cf. document stackOverflow ci-joint)
153
154 Si on ne met pas à disposition cette procédure, il faudrait forcer le typage de la
    constante littérale string au moment
155 de l'appel de la procédure ecrire( UnFichierTexte& f, string item) de la manière
    suivante :
156 ecrire (monFichier, (string) "bonjour");
157
158 Ce problème ne se pose pas lorsque la chaine se trouve dans une variable string
    préalablement déclarée.
159 string maChaine = "bonjour";
160 ecrire (monFichier, maChaine); ... fait appel à la bonne procédure surchargée
    ayant le paramètre string item.
161
162 */
163
164 void ecrire ( UnFichierTexte& f,
165             char item);
166 /*pour les modes creation et extension, le contenu du paramètre chaine
167 -- est enregistré en fin de fichier
168 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
169 -- erreurInconnue si l'écriture s'est mal déroulée (plus
    d'espace disque, fichier corrompu...)
170 */
171
172 void ecrire ( UnFichierTexte& f,
173             int item);
174 /*pour les modes creation et extension, le contenu du paramètre item
175 -- est enregistré en fin de fichier
176 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
177 -- erreurInconnue si l'écriture s'est mal déroulée (plus
    d'espace disque, fichier corrompu...)
178 */
179
180 void ecrire ( UnFichierTexte& f,
181             float item);
182 /*pour les modes creation et extension, le contenu du paramètre item
183 -- est enregistré en fin de fichier
184 -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
185 -- erreurInconnue si l'écriture s'est mal déroulée (plus
    d'espace disque, fichier corrompu...)
186 */
187
188 void ecrire ( UnFichierTexte& f,

```

```

189         bool item);
190     /*pour les modes creation et extension, le contenu du paramètre item
191     -- est enregistré en fin de fichier
192     -- : peut occasionner erreurDeMode si le mode d'ouverture interdit l'écriture,
193     --      erreurInconnue si l'écriture s'est mal déroulée (plus
194     d'espace disque, fichier corrompu...)
195     */
196     bool estOuvert (UnFichierTexte& f);
197     /* retourne VRAI si le fichier est ouvert, FAUX sinon */
198
199     string nomSysteme (UnFichierTexte& f);
200     /* retourne le nom du fichier sur le disque */
201
202     void renommer (UnFichierTexte& f, string nouveauNom);
203     /* Change le nom du fichier f.nom par nouveauNom, où f est le nom logique d'un
204     fichier texte.
205     Opération effectuée directement sur le fichier, sans utilisation de flot C++.
206     Si f.nom et nouveauNom précisent différentes localisations (répertoires),
207     le système déplace le fichier vers le nouvel emplacement.
208     Si nouveauNom fait référence à un fichier existant, la fonction peut échouer ou
209     écraser
210     le fichier existant, cela dépend du système d'exploitation.
211     Le fichier f doit être *fermé* pour que l'opération réussisse.
212     -- : peut occasionner erreurInconnue si l'opération échoue
213     */
214
215     void supprimer (UnFichierTexte& f) ;
216     /* Supprime le fichier de nom système associé au fichier logique f.
217     Opération effectuée directement sur le fichier, sans utilisation de flot C++.
218     Le fichier doit être *fermé* pour que l'opération réussisse.
219     -- : peut occasionner erreurInconnue si l'opération échoue
220     */
221
222     /**Exceptions
223     erreurDeStatut, erreurDeMode, erreurDeNomOuUsage, erreurInconnue;
224     erreurDUsage (pour primitive réécrire non implémentée)
225     */
226
227 #endif //FICHIERSTEXTE_H

```