Chapitre 7 : Les Fichiers

Objectifs pédagogiques :

- Comprendre la différence entre fichiers texte et binaires
- Maîtriser les opérations de base : ouverture, lecture, écriture, fermeture
- Savoir choisir le mode adapté à chaque situation
- Appliquer ces notions à des problèmes concrets (sauvegarde de mesures, configuration, résultats)

1. Introduction aux fichiers

1.1. Pourquoi utiliser des fichiers ?

En automatique, nous avons constamment besoin de :

- Sauvegarder des mesures pour analyse ultérieure
- Lire des paramètres de configuration au démarrage
- Enregistrer des résultats de simulation
- Échanger des données avec d'autres logiciels (Excel, Matlab, Python)
- Journaliser l'évolution d'un système (logging)

1.2. Types de fichiers

Mode	Caractéristiques	Utilisation
Texte	Données lisibles par un humain, séparateurs explicites	Configuration, échange, petits volumes
Binaire	Données au format machine, compacts, rapides	Gros volumes, données brutes, sauvegarde d'état

2. Fichiers texte en C (approche historique)

En C, on utilise `FILE*` et les fonctions `fprintf/fscanf`. Cette approche fonctionne aussi en C++.

2.1. Ouverture et fermeture

```
#include <cstdio>  // Equivalent C de <stdio.h>
using namespace std;

int main() {
    FILE* fichier = nullptr;

    // Ouverture en écriture ("w" = write)
    fichier = fopen("mesures.txt", "w");

    if (fichier == nullptr) {
        printf("Erreur : impossible d'ouvrir le fichier\n");
        return 1;
```

```
    }

    // Écriture
    fprintf(fichier, "Bonjour fichier !\n");
    fprintf(fichier, "Ceci est une ligne.\n");

    // Fermeture (TRÈS IMPORTANT !)
    fclose(fichier);

    return 0;
}
```

2.2. Modes d'ouverture

Mode	Description	Si fichier existe	Si fichier n'existe pas
`"r"`	Lecture seule	OK	Erreur
`"w"`	Écriture seule	Efface le contenu	Crée
`"a"`	Ajout (append)	Conserve, écrit à la fin	Crée
`"r+"`	Lecture/écriture	OK	Erreur
`"w+"`	Lecture/écriture	Efface	Crée
`"a+"`	Lecture/ajout	Conserve	Crée

2.3. Écriture formatée

```
#include <cstdio>
using namespace std;

int main() {
    FILE* fichier = fopen("donnees.txt", "w");

    if (fichier) {
        // Différents types de données
        int compteur = 42;
        double temperature = 23.5;
        char etat[] = "OK";
```

```
        fprintf(fichier, "Compteur: %d\n", compteur);
        fprintf(fichier, "Température: %.2f °C\n", temperature);
        fprintf(fichier, "État: %s\n", etat);

        // Écriture en tableau
        for (int i = 0; i < 5; i++) {
            fprintf(fichier, "%d\t%.1f\n", i, i * 10.5);
        }

        fclose(fichier);
    }

    return 0;
}
```

2.4. Lecture formatée

```
#include <cstdio>
using namespace std;

int main() {
    FILE* fichier = fopen("donnees.txt", "r");

    if (fichier) {
        int compteur;
        double temperature;
        char etat[10];  // Attention : taille fixe !

        // Lecture formatée
        fscanf(fichier, "Compteur: %d\n", &compteur);
        fscanf(fichier, "Température: %lf °C\n", &temperature);
        fscanf(fichier, "État: %s\n", etat);

        printf("Lu : compteur=%d, temp=%.2f, état=%s\n",
               compteur, temperature, etat);

        // Lecture de tableau
        printf("\nTableau lu :\n");
        int i;
        double val;
        while (fscanf(fichier, "%d\t%lf\n", &i, &val) == 2) {
            printf("  %d -> %.1f\n", i, val);
        }
```

```
        fclose(fichier);
    }

    return 0;
}
```

2.5. Problèmes avec l'approche C

- Gestion manuelle : il faut penser à `fclose()`
- Buffer overflow : avec `fscanf` et les chaînes
- Types non vérifiés : `%d` pour int, `%lf` pour double, etc.
- Erreurs silencieuses : pas d'exceptions

3. Fichiers en C++ : la manière moderne

La bibliothèque `<fstream>` (file stream) fournit une interface orientée objet, plus sûre et plus naturelle en C++.

3.1. Flux de fichiers

```cpp
#include <iostream>
#include <fstream>   // Pour les fichiers
#include <string>
using namespace std;

int main() {
    // Écriture : ofstream = output file stream
    ofstream fichier_sortie("mesures.txt");

    if (!fichier_sortie.is_open()) {
        cerr << "Erreur : impossible d'ouvrir le fichier en écriture" << endl;
        return 1;
    }

    // On utilise << comme avec cout
    fichier_sortie << "=== Mesures du système ===" << endl;
    fichier_sortie << "Temps (s)\tValeur" << endl;

    for (int i = 0; i < 5; i++) {
        fichier_sortie << i * 0.1 << "\t\t" << i * 2.5 << endl;
    }

    fichier_sortie.close();  // Optionnel (le destructeur le fait automatiquement)
```

```
    // Lecture : ifstream = input file stream
    ifstream fichier_entree("mesures.txt");

    if (!fichier_entree) {  // Autre façon de tester
        cerr << "Erreur : impossible d'ouvrir le fichier en lecture" << endl;
        return 1;
    }

    string ligne;
    while (getline(fichier_entree, ligne)) {  // Lit ligne par ligne
        cout << ligne << endl;
    }

    // Pas besoin de close explicitement (destructeur appelé automatiquement)

    return 0;
}
```

3.2. Les différentes classes

Classe	Rôle	Analogue à
`ifstream`	Lecture seule	`cin`
`ofstream`	Écriture seule	`cout`
`fstream`	Lecture et écriture	-

3.3. Modes d'ouverture en C++

```
#include <fstream>
using namespace std;

int main() {
    // Différents modes d'ouverture
    ofstream f1("fichier.txt");                  // Écriture (efface)
    ofstream f2("fichier.txt", ios::app);        // Ajout (append)
    ofstream f3("fichier.txt", ios::ate);        // À la fin mais peut écrire ailleurs

    ifstream f4("fichier.txt");                   // Lecture

    fstream f5("fichier.txt", ios::in | ios::out);  // Lecture + écriture

    // Modes binaires
```

```
    ofstream f6("donnees.bin", ios::binary);        // Écriture binaire
    ifstream f7("donnees.bin", ios::binary);        // Lecture binaire

    return 0;
}
```

3.4. Vérification d'erreurs

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream fichier("important.txt");

    if (!fichier) {
        cerr << "Erreur : fichier non trouvé" << endl;
        return 1;
    }

    // Autres tests
    if (fichier.fail()) {
        cerr << "Erreur lors de la lecture" << endl;
    }

    if (fichier.eof()) {
        cout << "Fin de fichier atteinte" << endl;
    }

    // Bonne pratique : tester après chaque opération
    double valeur;
    fichier >> valeur;

    if (fichier.fail()) {
        cerr << "Erreur : format incorrect" << endl;
    }

    return 0;
}
```

4. Fichiers binaires

4.1. Pourquoi le binaire ?

Critère	Texte	Binaire
Taille	Grande	Petite
Vitesse	Plus lente (conversion)	Plus rapide
Lisibilité	Lisible par humain	Illisible
Précision	Peut perdre en conversion	Exacte
Portabilité	Excellente	Dépend de l'architecture

4.2. Écriture binaire

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    // Données à sauvegarder
    double mesures[5] = {12.5, 23.7, 18.9, 21.3, 19.8};
    int nb_mesures = 5;
    char commentaire[] = "Mesures du 25/02/2024";

    // Écriture binaire
    ofstream fichier("donnees.bin", ios::binary);

    if (!fichier) {
        cerr << "Erreur d'ouverture" << endl;
        return 1;
    }

    // Écriture brute des données
    fichier.write(reinterpret_cast<char*>(&nb_mesures), sizeof(nb_mesures));
    fichier.write(commentaire, sizeof(commentaire));
    fichier.write(reinterpret_cast<char*>(mesures), sizeof(mesures));

    // Alternative : écrire élément par élément
    // for (int i = 0; i < nb_mesures; i++) {
    //     fichier.write(reinterpret_cast<char*>(&mesures[i]), sizeof(double));
    // }

    cout << "Données sauvegardées en binaire" << endl;
```

```
    return 0;
}
```

4.3. Lecture binaire

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    // Lecture binaire
    ifstream fichier("donnees.bin", ios::binary);

    if (!fichier) {
        cerr << "Erreur d'ouverture" << endl;
        return 1;
    }

    // Lecture dans l'ordre d'écriture
    int nb_mesures_lues;
    char commentaire_lu[50];
    double mesures_lues[5];

    fichier.read(reinterpret_cast<char*>(&nb_mesures_lues), sizeof(nb_mesures_lues));
    fichier.read(commentaire_lu, sizeof(commentaire_lu));
    fichier.read(reinterpret_cast<char*>(mesures_lues), sizeof(mesures_lues));

    // Vérification
    if (!fichier) {
        cerr << "Erreur lors de la lecture" << endl;
        return 1;
    }

    // Affichage
    cout << "Nombre de mesures : " << nb_mesures_lues << endl;
    cout << "Commentaire : " << commentaire_lu << endl;
    cout << "Mesures : ";
    for (int i = 0; i < nb_mesures_lues; i++) {
        cout << mesures_lues[i] << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

4.4. Positionnement dans le fichier

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    // Création d'un fichier binaire
    ofstream fout("test.bin", ios::binary);

    if (fout) {
        double data[10] = {0,1,2,3,4,5,6,7,8,9};
        fout.write(reinterpret_cast<char*>(data), sizeof(data));
        fout.close();
    }

    // Lecture à une position spécifique
    ifstream fin("test.bin", ios::binary);

    if (fin) {
        // Aller au 5ème élément (index 4)
        fin.seekg(4 * sizeof(double), ios::beg);  // seekg = seek get

        double valeur;
        fin.read(reinterpret_cast<char*>(&valeur), sizeof(double));
        cout << "5ème élément : " << valeur << endl;  // Devrait être 4

        // Position actuelle
        streampos pos = fin.tellg();
        cout << "Position : " << pos << " octets" << endl;

        // Retour au début
        fin.seekg(0, ios::beg);

        fin.close();
    }

    return 0;
}
```

5. Lecture de fichiers de configuration

5.1. Format simple clé=valeur

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <map>
using namespace std;

class Config {
private:
    map<string, string> parametres;

public:
    bool charger(const string& nom_fichier) {
        ifstream fichier(nom_fichier);

        if (!fichier) {
            return false;
        }

        string ligne;
        int ligne_num = 0;

        while (getline(fichier, ligne)) {
            ligne_num++;

            // Ignorer les commentaires et lignes vides
            if (ligne.empty() || ligne[0] == '#') {
                continue;
            }

            // Chercher le séparateur '='
            size_t pos = ligne.find('=');
            if (pos == string::npos) {
                cerr << "Ligne " << ligne_num << " : format incorrect (pas de '=')" << endl;
                continue;
            }

            // Extraire clé et valeur
            string cle = ligne.substr(0, pos);
            string valeur = ligne.substr(pos + 1);
```

```
            // Nettoyer les espaces
            cle.erase(0, cle.find_first_not_of(" \t"));
            cle.erase(cle.find_last_not_of(" \t") + 1);

            valeur.erase(0, valeur.find_first_not_of(" \t"));
            valeur.erase(valeur.find_last_not_of(" \t") + 1);

            parametres[cle] = valeur;
        }

        return true;
    }

    template<typename T>
    T get(const string& cle, T defaut = T()) {
        auto it = parametres.find(cle);
        if (it == parametres.end()) {
            return defaut;
        }

        stringstream ss(it->second);
        T valeur;
        ss >> valeur;
        return valeur;
    }

    void afficher() {
        cout << "=== Configuration ===" << endl;
        for (const auto& p : parametres) {
            cout << p.first << " = " << p.second << endl;
        }
    }
};

int main() {
    // Créer un fichier de configuration exemple
    ofstream config_exemple("systeme.cfg");
    config_exemple << "# Configuration du système\n";
    config_exemple << "tau = 0.5\n";
    config_exemple << "K = 2.0\n";
    config_exemple << "mode = auto\n";
    config_exemple << "seuil_alarme = 100.5\n";
```

```
    config_exemple << "nb_capteurs = 4\n";
    config_exemple.close();

    // Lire la configuration
    Config config;
    if (config.charger("systeme.cfg")) {
        config.afficher();

        // Utiliser les paramètres
        double tau = config.get<double>("tau", 1.0);
        double K = config.get<double>("K", 1.0);
        string mode = config.get<string>("mode", "manuel");
        int nb_capteurs = config.get<int>("nb_capteurs", 1);
        double seuil = config.get<double>("seuil_alarme", 50.0);

        cout << "\nParamètres utilisés :" << endl;
        cout << "tau = " << tau << endl;
        cout << "K = " << K << endl;
        cout << "mode = " << mode << endl;
        cout << "nb_capteurs = " << nb_capteurs << endl;
        cout << "seuil_alarme = " << seuil << endl;
    }

    return 0;
}
```

5.2. Format CSV (Comma Separated Values)

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
using namespace std;

vector<vector<double>> lire_csv(const string& nom_fichier, char separateur = ',') {
    vector<vector<double>> donnees;
    ifstream fichier(nom_fichier);

    if (!fichier) {
        cerr << "Impossible d'ouvrir " << nom_fichier << endl;
        return donnees;
    }
```

```
    string ligne;
    int ligne_num = 0;

    while (getline(fichier, ligne)) {
        ligne_num++;

        // Ignorer les en-têtes (première ligne)
        if (ligne_num == 1 && ligne.find("temps") != string::npos) {
            continue;
        }

        vector<double> ligne_donnees;
        stringstream ss(ligne);
        string cellule;

        while (getline(ss, cellule, separateur)) {
            try {
                double valeur = stod(cellule);
                ligne_donnees.push_back(valeur);
            } catch (...) {
                // Ignorer les valeurs non numériques
            }
        }

        if (!ligne_donnees.empty()) {
            donnees.push_back(ligne_donnees);
        }
    }

    return donnees;
}

void ecrire_csv(const string& nom_fichier,
                const vector<double>& temps,
                const vector<double>& signaux[],
                int nb_signaux,
                const string noms[],
                char separateur = ',') {

    ofstream fichier(nom_fichier);

    if (!fichier) {
```

```
        cerr << "Impossible de créer " << nom_fichier << endl;
        return;
    }

    // En-tête
    fichier << "temps";
    for (int i = 0; i < nb_signaux; i++) {
        fichier << separateur << noms[i];
    }
    fichier << endl;

    // Données
    for (size_t i = 0; i < temps.size(); i++) {
        fichier << temps[i];
        for (int j = 0; j < nb_signaux; j++) {
            fichier << separateur << signaux[j][i];
        }
        fichier << endl;
    }

    cout << "Fichier CSV créé : " << nom_fichier << endl;
}

int main() {
    // Génération de données exemple
    vector<double> temps(100);
    vector<double> signal1(100);
    vector<double> signal2(100);

    for (int i = 0; i < 100; i++) {
        temps[i] = i * 0.01;
        signal1[i] = 10.0 * sin(2 * M_PI * 2.0 * temps[i]);
        signal2[i] = 5.0 * cos(2 * M_PI * 1.0 * temps[i]);
    }

    // Écriture CSV
    vector<double>* signaux[] = {&signal1, &signal2};
    string noms[] = {"sinus_2Hz", "cosinus_1Hz"};
    ecrire_csv("donnees.csv", temps, signaux, 2, noms);

    // Lecture CSV
    auto data = lire_csv("donnees.csv");
    cout << "\nDonnées lues : " << data.size() << " lignes" << endl;
```

```
    if (!data.empty()) {
        cout << "Première ligne : ";
        for (double val : data[0]) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}
```

6. Journalisation (logging)

6.1. Journal simple

```
#include <iostream>
#include <fstream>
#include <ctime>
using namespace std;

class Logger {
private:
    ofstream fichier;
    string niveau_str[4] = {"DEBUG", "INFO", "WARN", "ERROR"};

public:
    enum Niveau { DEBUG, INFO, WARN, ERROR };

    Logger(const string& nom_fichier) {
        fichier.open(nom_fichier, ios::app);  // Mode append
        if (fichier) {
            time_t now = time(nullptr);
            fichier << "=== Journal démarré le " << ctime(&now);
            fichier << "================================\n" << endl;
        }
    }

    ~Logger() {
        if (fichier) {
            time_t now = time(nullptr);
            fichier << "\n=== Journal terminé le " << ctime(&now);
        }
    }
```

```
    void log(Niveau niveau, const string& message) {
        if (!fichier) return;

        time_t now = time(nullptr);
        struct tm* tm_info = localtime(&now);
        char timestamp[20];
        strftime(timestamp, 20, "%H:%M:%S", tm_info);

        fichier << "[" << timestamp << "] "
                << "[" << niveau_str[niveau] << "] "
                << message << endl;

        // Aussi à l'écran pour les erreurs
        if (niveau >= WARN) {
            cerr << "[" << niveau_str[niveau] << "] " << message << endl;
        }
    }
};

int main() {
    Logger journal("systeme.log");

    journal.log(Logger::INFO, "Démarrage du système");

    double temperature = 85.5;
    journal.log(Logger::DEBUG, "Température = " + to_string(temperature));

    if (temperature > 100.0) {
        journal.log(Logger::ERROR, "Surchauffe détectée !");
    } else {
        journal.log(Logger::INFO, "Température normale");
    }

    for (int i = 0; i < 3; i++) {
        journal.log(Logger::DEBUG, "Itération " + to_string(i));
    }

    journal.log(Logger::INFO, "Arrêt du système");

    return 0;
}
```

7. Synthèse

Opération	Texte (C)	Texte (C++)	Binaire (C++)
Ouverture	`fopen(nom, "w")`	`ofstream f(nom)`	`ofstream f(nom, ios::binary)`
Écriture	`fprintf(f, "%d", x)`	`f << x`	`f.write(&x, sizeof(x))`
Lecture	`fscanf(f, "%d", &x)`	`f >> x`	`f.read(&x, sizeof(x))`
Ligne	`fgets(buf, size, f)`	`getline(f, str)`	-
Fermeture	`fclose(f)`	Automatique	Automatique
Test erreur	`f == NULL`	`!f` ou `f.fail()`	`!f` ou `f.fail()`

Règles d'or :

- Toujours vérifier que l'ouverture a réussi
- Fermer les fichiers (ou laisser le destructeur le faire)
- Pour les fichiers texte, utilisez des séparateurs clairs (tabulation, virgule)
- Pour les fichiers binaires, documentez le format
- N'oubliez pas ios::binary pour les fichiers !