Chapitre 3 : Structures conditionnelles et boucles

Objectifs pédagogiques :

- Maîtriser les structures conditionnelles pour créer des programmes qui prennent des décisions
- Savoir utiliser les boucles pour répéter des calculs (simulations, échantillonnage)
- Combiner conditions et boucles pour implémenter des algorithmes typiques de l'automatique
- Éviter les pièges classiques (boucles infinies, conditions jamais remplies)

1. Introduction

En automatique, nous avons constamment besoin de prendre des décisions et de répéter des calculs :

- Décisions : "Si la température dépasse le seuil, déclencher l'alarme"
- Répétitions : "Pour chaque pas de temps de 0 à 10 secondes, calculer la réponse du système"

Les structures conditionnelles et les boucles sont les outils qui permettent d'implémenter ces comportements.

2. Structures conditionnelles

2.1. La structure `if`

La forme la plus simple pour prendre une décision.

Syntaxe :

```
if (condition) {
    // Instructions exécutées si condition est vraie
}
```

Exemple :

```
double temperature = 85.5;

if (temperature > 100.0) {
    cout << "ALARME : Surchauffe détectée !" << endl;
    // Déclencher ventilation, arrêt d'urgence, etc.
}
```

Règle fondamentale : La condition doit être une expression booléenne (vraie ou fausse). En C++, 0 est considéré comme faux, toute valeur non nulle est vraie.

2.2. La structure `if-else`

Pour gérer les deux cas (vrai et faux).

Syntaxe :

```
if (condition) {
    // Instructions si condition vraie
} else {
    // Instructions si condition fausse
}
```

Exemple :

```
// Exemple : Détection de mouvement
double vitesse = 12.5;  // m/s

if (vitesse > 0) {
    cout << "Le système est en mouvement" << endl;
    cout << "Vitesse : " << vitesse << " m/s" << endl;
} else {
    cout << "Le système est à l'arrêt" << endl;
}
```

2.3. La structure `if-else if-else`

Pour gérer plusieurs cas mutuellement exclusifs.

Syntaxe :

```
if (condition1) {
    // Cas 1
} else if (condition2) {
    // Cas 2
} else if (condition3) {
    // Cas 3
} else {
    // Tous les autres cas
}
```

Exemple :

```
// Exemple : Contrôle de vitesse d'un moteur
double vitesse_consigne = 1500;  // tr/min

if (vitesse_consigne < 0) {
    cout << "ERREUR : Vitesse négative" << endl;
} else if (vitesse_consigne == 0) {
    cout << "Moteur à l'arrêt" << endl;
```

```
} else if (vitesse_consigne <= 1000) {
    cout << "Régime basse vitesse" << endl;
} else if (vitesse_consigne <= 3000) {
    cout << "Régime nominal" << endl;
} else {
    cout << "ATTENTION : Surrégime !" << endl;
}
```

2.4. La structure `switch`

Alternative à if-else if quand on teste une variable discrète (entier ou caractère).

Syntaxe :

```
switch (expression) {
    case valeur1:
        // Instructions pour valeur1
        break;
    case valeur2:
        // Instructions pour valeur2
        break;
    default:
        // Instructions pour tous les autres cas
}
```

Exemple :

```
// Exemple : Machine d'états simple pour un feu tricolore
int etat_feu = 2;  // 1=rouge, 2=orange, 3=vert

switch (etat_feu) {
    case 1:
        cout << "Feu ROUGE - Arrêt obligatoire" << endl;
        break;
    case 2:
        cout << "Feu ORANGE - Préparez-vous à démarrer/arrêter" << endl;
        break;
    case 3:
        cout << "Feu VERT - Passage autorisé" << endl;
        break;
    default:
        cout << "État inconnu - PANNE" << endl;
}
```

Points importants :

- Le break est crucial ! Sans lui, l'exécution "tombe" dans le cas suivant (appelé "fallthrough").
- default est optionnel mais recommandé.

2.5. Conditions et opérateurs logiques

Parfois, il est nécessaire de combiner les conditions avec les opérateurs logiques pour satisfaire une condition complexe.

Exemple :

```
// Combinaison d'opérateurs logiques
double temperature = 85.5;
double pression = 2.3;
bool urgence_arret = false;

// Vérification de sécurité multiple
if (temperature > 100.0 || pression > 3.0 || urgence_arret) {
    cout << "ARRET D'URGENCE" << endl;
    // Couper alimentation, actionner freins, etc.
}
```

3. Les boucles

Les boucles permettent de répéter des instructions. En automatique, on les utilise pour :

- Simuler l'évolution d'un système dans le temps
- Parcourir des tableaux de mesures
- Itérer jusqu'à convergence d'un algorithme

3.1. La boucle `while`

Répète tant qu'une condition est vraie. Idéal quand on ne connaît pas à l'avance le nombre d'itérations.

Syntaxe :

```
while (condition) {
    // Instructions répétées tant que condition est vraie
}
```

Exemple :

```
// Exemple : Attente d'une condition (simulée)
double temperature = 20.0;
double consigne = 100.0;
int temps = 0;

cout << "Chauffage en cours..." << endl;
while (temperature < consigne) {
    temperature += 5.0;  // Chauffe de 5° par unité de temps
    temps++;
    cout << "t = " << temps << "s : T = " << temperature << "°C" << endl;
}
cout << "Température atteinte après " << temps << " secondes" << endl;
```

3.2. La boucle `do-while`

Similaire à while, mais la condition est testée APRÈS l'exécution. Garantit au moins une itération.

Syntaxe :

```
do {
    // Instructions exécutées au moins une fois
} while (condition);
```

Exemple :

```
// Exemple : Saisie utilisateur avec validation
double tension;
bool tension_valide;

do {
    cout << "Entrez la tension (0-24V) : ";
    cin >> tension;

    tension_valide = (tension >= 0 && tension <= 24);

    if (!tension_valide) {
        cout << "ERREUR : Tension hors plage. Recommencez." << endl;
    }
} while (!tension_valide);

cout << "Tension enregistrée : " << tension << " V" << endl;
```

3.3. La boucle **for**

Idéale quand on connaît le nombre d'itérations à l'avance. C'est la boucle la plus utilisée en calcul scientifique.

Syntaxe :

```
for (initialisation; condition; incrémentation) {
    // Instructions répétées
}
```

Exemple :

```
// Exemple : Simulation temporelle
double dt = 0.1;  // pas de temps (s)
double t_max = 2.0;  // temps total (s)
int nb_pas = static_cast<int>(t_max / dt);

cout << "Simulation sur " << nb_pas << " pas de temps" << endl;
```

```
for (int i = 0; i < nb_pas; i++) {
    double t = i * dt;
    double y = 2.0 * (1 - exp(-t / 0.5));  // Réponse indicielle d'un 1er ordre
    cout << "t = " << t << " s, y = " << y << endl;
}
```

4. Instructions de contrôle supplémentaires

4.1. `break` : Sortie anticipée d'une boucle

```
// Recherche d'une valeur avec arrêt anticipé
double seuil_alarme = 100.0;
double mesures[5] = {45.2, 78.5, 102.3, 67.8, 89.1};
bool alarme_declenchee = false;

for (int i = 0; i < 5; i++) {
    cout << "Mesure " << i << " : " << mesures[i] << endl;

    if (mesures[i] > seuil_alarme) {
        cout << "ALARME détectée ! Arrêt de l'analyse." << endl;
        alarme_declenchee = true;
        break;  // Sort immédiatement de la boucle
    }
}

if (!alarme_declenchee) {
    cout << "Aucune alarme détectée." << endl;
}
```

4.2. `continue` : Passer à l'itération suivante

```
// Traitement des mesures en ignorant les valeurs aberrantes
double mesures[8] = {12.3, 45.6, -999.9, 78.9, 23.4, -999.9, 56.7, 89.0};
double somme = 0.0;
int nb_valides = 0;

for (int i = 0; i < 8; i++) {
    if (mesures[i] < -900) {  // Valeur aberrante (code d'erreur)
        cout << "Mesure " << i << " aberrante, ignorée" << endl;
        continue;  // Passe directement à i++ suivant
    }

    somme += mesures[i];
    nb_valides++;
```

```
    cout << "Mesure " << i << " valide: " << mesures[i] << endl;
}

cout << "Moyenne sur " << nb_valides << " mesures: " << (somme / nb_valides) << endl;
```