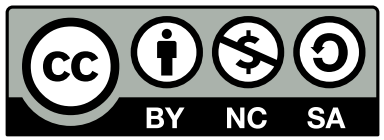
INF1135
Programmation procédurale
Chapitre 3 — Structures de contrôle

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Structures conditionnelles
2. Boucles
3. Tests et débogage

Plan

1. Structures conditionnelles
2. Boucles
3. Tests et débogage

Exemple : if / else if / else

Fichier ifelse.c :

```c
#include <stdio.h>

int main(void) {
  int note = 78;
  if (note >= 90) {
    printf("mention: excellent\n");
  } else if (note >= 75) {
    printf("mention: tres bien\n");
  } else if (note >= 60) {
    printf("mention: bien\n");
  } else {
    printf("mention: insuffisant\n");
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o ifelse ifelse.c
$ ./ifelse
mention: tres bien
```

if / else if / else

Syntaxe

```
if (condition1) {
   ...
} else if (condition2) {
   ...
} else {
   ...
}
```

- Bloc `if` : exécuté si la condition est **vraie** ($\neq 0$), par exemple un `bool` : `if (est_led_allume)`
- `else if` : évalué seulement si les conditions précédentes sont fausses
- `else` : optionnel — exécuté si **aucune** condition n'est vraie
- Première condition vraie : **gagne** — les suivantes ne sont pas évaluées
- Accolades `{}` : nécessaires si plus d'une instruction

Exemple : switch

Fichier switch.c :

```c
#include <stdio.h>

int main(void) {
  int jour = 3;
  switch (jour) {
    case 1:
      printf("lundi\n");
      break;
    case 2:
      printf("mardi\n");
      break;
    case 3:
      printf("mercredi\n");
      break;
    case 6:
    case 7:
      printf("fin de semaine\n");
      break;
    default:
      printf("jour invalide\n");
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o switch switch.c
$ ./switch
mercredi
```

- case 6/case 7 regroupés : même traitement, un seul break

switch / case / default / break

Syntaxe

```
switch (expression) {
  case valeur1: ...; break;
  case valeur2: ...; break;
  default: ...
}
```

- `expression` : type **entier** (`int`, `char`, ...)
- `case valeur` : comparaison d'**égalité** avec `expression`
- `break` : sort du `switch` — sinon, **continue** dans le cas suivant
- `default` : optionnel — si aucun `case` ne correspond

Piège fréquent : passage au cas suivant

Oublier un `break` → les instructions du cas suivant sont aussi exécutées.

Exemple : opérateur ternaire

Fichier `ternary.c` :

```c
#include <stdio.h>

int main(void) {
  int a = 12, b = 7;
  int max = (a > b) ? a : b;
  printf("max(%d, %d) = %d\n", a, b, max);

  int nb_fichiers = 1;
  printf("%d fichier%s copie%s\n", nb_fichiers,
         (nb_fichiers > 1) ? "s" : "", (nb_fichiers > 1) ? "s" : "");

  int nb_erreurs = 3;
  printf("%d erreur%s\n", nb_erreurs, (nb_erreurs > 1) ? "s" : "");
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o ternary ternary.c
$ ./ternary
max(12, 7) = 12
1 fichier copie
3 erreurs
```

Opérateur ternaire ? :

```
condition ? valeur_si_vraie : valeur_si_fausse
```

- Seul opérateur **ternaire** du langage (3 opérandes)
- C'est une **expression**, pas une instruction : elle retourne une valeur, utilisable dans une affectation ou un `printf`
- Équivalent compact à un `if`/`else` qui ne fait qu'affecter une valeur
- Lisible pour des cas simples

Exemple

```
int max = (a > b) ? a : b;
```

équivaut à

```
int max; if (a > b) { max = a; } else { max = b; }
```

Plan

1. Structures conditionnelles
2. Boucles
3. Tests et débogage

Exemple : while vs do-while

Fichier whiledo.c :

```c
#include <stdio.h>

int main(void) {
  printf("while (n = 5, condition n > 10):\n");
  int n = 5;
  while (n > 10) {
    printf("  n = %d\n", n);
    n--;
  }
  printf("  (rien affiche)\n");

  printf("do-while (n = 5, condition n > 10):\n");
  n = 5;
  do {
    printf("  n = %d\n", n);
    n--;
  } while (n > 10);

  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o whiledo whiledo.c
$ ./whiledo
while (n = 5, condition n > 10):
  (rien affiche)
do-while (n = 5, condition n > 10):
  n = 5
```

Boucle `while`

Syntaxe

```
while (condition) {
   ...
}
```

- Condition évaluée **avant** chaque itération
- Corps exécuté **tant que** la condition est vraie
- Condition fausse dès le départ → corps **jamais** exécuté
- Boucle **infinie** si la condition ne devient jamais fausse

Piège fréquent

Oublier de faire progresser une variable de la condition → boucle infinie.

Boucle `do-while`

Syntaxe

```
do {
   ...
} while (condition);
```

- Corps exécuté **avant** de tester la condition
- Corps exécuté **au moins une fois**, peu importe la condition
- Utile quand une action doit se produire avant de pouvoir vérifier la condition (ex. lire une entrée avant de la valider)
- Point-virgule **obligatoire** après `while(condition)`

Exemple : for

Fichier forloop.c :

```c
#include <stdio.h>

int main(void) {
  int somme = 0;
  for (int i = 1; i <= 10; i++) {
    somme += i;
  }
  printf("somme = %d\n", somme);

  for (int i = 10; i >= 1; i -= 2) {
    printf("%d ", i);
  }
  printf("\n");
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o forloop forloop.c
$ ./forloop
somme = 55
10 8 6 4 2
```

Boucle `for`

Syntaxe

```
for (initialisation; condition; increment) {
   ...
}
```

- `initialisation` : exécutée **une fois**, avant la première itération
- `condition` : évaluée avant chaque itération (comme `while`)
- `increment` : exécuté après chaque itération, avant de retester la condition
- Les trois parties sont **optionnelles** — `for (;;)` = boucle infinie
- Variable déclarée dans le `for` (`for (int i = 0; ...)`) : portée limitée à la boucle

Exemple : break et continue

Fichier breakcontinue.c :

```c
#include <stdio.h>

int main(void) {
  printf("break: ");
  for (int i = 1; i <= 10; i++) {
    if (i == 6) break;
    printf("%d ", i);
  }
  printf("\n");

  printf("continue: ");
  for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) continue;
    printf("%d ", i);
  }
  printf("\n");
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o breakcontinue breakcontinue.c
$ ./breakcontinue
break: 1 2 3 4 5
continue: 1 3 5 7 9
```

`break` et `continue`

- `break` : sort **immédiatement** de la boucle (ou du `switch`)
- `continue` : passe **immédiatement** à l'itération suivante — saute le reste du corps
- Dans un `for` : `continue` exécute quand même l'**increment** avant de retester la condition
- Applicables à `while`, `do-while` et `for`

À utiliser avec parcimonie

Trop de `break`/`continue` $\rightarrow$ flot de contrôle difficile à suivre.
Souvent, une condition bien formulée les rend inutiles.

Exemple : boucles imbriquées

Fichier `nested.c` :

```c
#include <stdio.h>

int main(void) {
  for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
      printf("%2d ", i * j);
    }
    printf("\n");
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o nested nested.c
$ ./nested
 1  2  3
 2  4  6
 3  6  9
```

Boucles imbriquées

- Une boucle peut contenir une autre boucle dans son corps
- Boucle **externe** : une itération = une exécution **complète** de la boucle interne
- Coût : n itérations externes $\times$ m itérations internes $= n \times m$ répétitions du corps
- `break`/`continue` : n'affectent que la boucle **la plus proche**

Utilisations typiques

- Parcours d'une grille ou d'une matrice (lignes $\times$ colonnes)
- Tables (ex. table de multiplication)
- Comparaison de toutes les paires d'éléments

Exemple : que vaut `total` ?

Fichier `triangle.c` :

```
1  #include <stdio.h>
2
3  int main(void) {
4    int total = 0;
5    for (int i = 1; i <= 3; i++) {
6      for (int j = 1; j <= i; j++) {
7        total += j;
8      }
9    }
10   printf("total = %d\n", total);
11   return 0;
12 }
```

Question

Sans compiler ni exécuter le programme, que va-t-il afficher ?

Exemple : trace à la main (à compléter)

total	i	i <= 3	j	j <= i

Exemple : trace à la main (solution)

total	i	i <= 3	j	j <= i
0	–	–	–	–
0	1	vrai		
1	1		1	vrai
1	1		2	**faux**
1	2	vrai		
2	2		1	vrai
4	2		2	vrai
4	2		3	**faux**
4	3	vrai		
5	3		1	vrai
7	3		2	vrai
10	3		3	vrai
10	3		4	**faux**
10	4	**faux**	–	–

- Une **colonne** par variable et par condition
- Une **ligne** par test d'une condition
- Un trait horizontal à chaque nouveau tour de la boucle **externe**
- Dernière ligne : `i <= 3` est faux, la boucle externe se termine

```
$ ./triangle
total = 10
```

Trace à la main

- **Trace à la main** : exécuter le programme **sur papier**, instruction par instruction, en notant l'état des variables dans un tableau
- Colonnes : les variables qui changent, et les **conditions** évaluées
- Nouvelle ligne dès qu'une valeur change ou qu'une condition est évaluée
- Suivre le code **tel qu'il est écrit**, pas tel qu'on pense l'avoir écrit

Utile pour

- **Prédire** le résultat d'un programme avant de l'exécuter
- Vérifier les **bornes** d'une boucle : première et dernière itération, condition qui devient fausse
- Comprendre un programme écrit par quelqu'un d'autre

Plan

1. Structures conditionnelles
2. Boucles
3. Tests et débogage

Exemple : conversion d'une note en lettre

Spécification

Lire une note entière entre 0 et 100 et afficher sa lettre : `A` si ≥ 90, `B` si ≥ 80, `C` si ≥ 70, `D` si ≥ 60, `E` sinon. Une note hors de l'intervalle est **invalide**.

```c
#include <stdio.h>

int main(void) {
  int note;
  char lettre;
  scanf("%d", &note);
  if (note > 90) {
    lettre = 'A';
  } else if (note > 80) {
    lettre = 'B';
  } else if (note > 70) {
    lettre = 'C';
  } else if (note > 60) {
    lettre = 'D';
  } else {
    lettre = 'E';
  }
  printf("%d -> %c\n", note, lettre);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o grades grades.c
$ echo 85 | ./grades
85 -> B
$ echo 72 | ./grades
72 -> C
```

Le programme compile sans avertissement et donne les bonnes réponses... Est-il **correct** ?

Exemple : quelques entrées de plus

```
$ echo 90 | ./grades
90 -> B
$ echo 105 | ./grades
105 -> A
$ echo -3 | ./grades
-3 -> E
```

- 90 : devrait donner `A` — > au lieu de >= dans **toutes** les conditions
- 105 : note **invalide** acceptée sans message
- -3 : note **invalide** convertie en `E`

Constat

Les entrées essayées pendant le développement étaient toutes **typiques**. Les erreurs se cachent aux **bornes** et hors des **hypothèses**.

Exemple : version corrigée

```c
#include <stdio.h>

int main(void) {
  int note;
  char lettre;
  scanf("%d", &note);
  if (note < 0 || note > 100) {
    printf("%d -> note invalide\n", note);
    return 1;
  }
  if (note >= 90) {
    lettre = 'A';
  } else if (note >= 80) {
    lettre = 'B';
  } else if (note >= 70) {
    lettre = 'C';
  } else if (note >= 60) {
    lettre = 'D';
  } else {
    lettre = 'E';
  }
  printf("%d -> %c\n", note, lettre);
  return 0;
}
```

```
$ echo 90 | ./grades-fixed
90 -> A
$ echo 89 | ./grades-fixed
89 -> B
$ echo 100 | ./grades-fixed
100 -> A
$ echo 0 | ./grades-fixed
0 -> E
$ echo 105 | ./grades-fixed
105 -> note invalide
$ echo -3 | ./grades-fixed
-3 -> note invalide
```

- Validation de l'intervalle **avant** la conversion
- >= dans toutes les conditions
- Bornes vérifiées de **part et d'autre** : 90/89, 100/105, 0/−3

Question

Comment choisir **systématiquement** les entrées à essayer ?

Notion de cas de test

- **Cas de test** : entrée précise + sortie **attendue**, utilisée pour vérifier un programme
- Écrire les cas de test **avant ou pendant** l'écriture du code — pas seulement après
- Un programme qui **compile** n'est pas toujours **correct**
- Tester **plusieurs** entrées, pas seulement celle utilisée pendant le développement

Dans ce cours

Chaque structure de contrôle introduit de nouveaux **chemins** possibles dans le programme — chacun mérite d'être testé.

Cas normaux, cas limites, cas d'erreur

Cas normaux

Entrées typiques, respectant les hypothèses du programme.

Cas limites

Bornes des conditions : exactement à l'endroit où une condition bascule.

- → Ex. : `if (n >= 0)` — tester `n = 0`, `n = -1`
- → Boucles : 0 itération, 1 itération, dernière itération

Cas d'erreur

Entrées invalides ou inattendues — le programme doit réagir **sans planter**.

- → Ex. : division par zéro, entrée non numérique, tableau vide

Exemple : suivre le flot de contrôle

Fichier `trace.c` :

```c
#include <stdio.h>

int main(void) {
  int tab[] = {4, 8, 15, 16, 23, 42};
  int taille = 6, cible = 16, trouve = -1;
  for (int i = 0; i < taille; i++) {
    printf("trace: i=%d, tab[i]=%d\n", i, tab[i]);
    if (tab[i] == cible) {
      trouve = i;
      break;
    }
  }
  if (trouve >= 0) {
    printf("cible %d trouvee a l'indice %d\n", cible, trouve);
  } else {
    printf("cible %d absente\n", cible);
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o trace trace.c
$ ./trace
trace: i=0, tab[i]=4
trace: i=1, tab[i]=8
trace: i=2, tab[i]=15
trace: i=3, tab[i]=16
cible 16 trouvee a l'indice 3
```

Débogage : traces d'exécution

- **Trace d'exécution** : afficher, au fil de l'exécution, les valeurs des variables et le chemin emprunté
- Utile pour repérer : une condition qui ne se déclenche jamais (ou trop souvent), une boucle qui ne progresse pas, une valeur inattendue
- Réflexe : `printf` **temporaire** avant/dans/après la structure suspecte

Gabarit de trace

```
printf("trace: i=%d, x=%d\n", i, x);
```

Stratégie

1. Isoler la structure de contrôle suspecte, ajouter des traces autour et dans celle-ci
2. Comparer le chemin observé au chemin **attendu**, puis retirer les traces