

05A : Fondements du C

420-3A6-ST : Programmation système

Dany Gagnon

Cégep de Saint-Jean-sur-Richelieu

Mercredi 23 septembre 2026

Table des matières

- ① Programme C et compilation
- ② Variables, types et opérateurs
- ③ Structures de contrôle
- ④ Fonctions
- ⑤ Tableaux, adresses et chaînes
- ⑥ Entrées, sorties et mémoire
- ⑦ Exercices

Contenu et conventions

- Variables, types, opérateurs et structures de contrôle.
- Fonctions, tableaux, pointeurs et chaînes de caractères.
- Entrées/sorties, allocation et premiers programmes sur les fichiers.

Nous utilisons C17. Les exemples sont compilés avec GCC sous Linux. Les fragments de code supposent que les en-têtes indiqués ont été inclus.

Structure d'un programme C

```
#include <stdio.h>
```

```
int main(void)
{
    printf("Bonjour\n");
    return 0;
}
```

- `#include <stdio.h>` fournit notamment la déclaration de `printf`.
- `main` est la fonction appelée au démarrage du programme C.
- `void` indique ici que la fonction ne reçoit aucun paramètre.
- `return 0;` signale une fin normale au système.

Compilation et exécution

Enregistrer le programme précédent dans `bonjour.c`.

```
gcc -std=c17 -Wall -Wextra -Wpedantic bonjour.c -o bonjour  
./bonjour  
# Bonjour
```

- `-std=c17` sélectionne la version du langage utilisée dans le cours.
- `-Wall -Wextra -Wpedantic` active des avertissements.
- `-o bonjour` nomme le fichier exécutable produit.
- `./bonjour` exécute le fichier situé dans le répertoire courant.

Instructions, blocs et commentaires

```
int total = 0;           // commentaire jusqu'a la fin de la ligne
/* Un commentaire peut occuper
   plusieurs lignes. */
if (total == 0) {
    total = 3;
    printf("%d\n", total);
}
```

- Une déclaration ou une instruction simple se termine par ;.
- Les accolades { } délimitent un bloc.
- Les parenthèses entourent les arguments d'un appel ou la condition d'un `if`.
- L'indentation reflète les blocs; elle ne change pas leur sens.

Directives du préprocesseur

```
#include <stdio.h>  
#define CAPACITE 32
```

```
char ligne[CAPACITE];
```

- `#include` insère le contenu d'un en-tête avant la compilation.
- `#define` remplace ici le nom `CAPACITE` par `32`.
- Une macro n'est pas une variable : elle n'a pas de type ni de stockage.
- Ces directives ne se terminent pas par un point-virgule.

Déclaration et initialisation

Une variable est déclarée avec un type et un nom. L'initialisation fournit sa première valeur.

```
int compteur = 0;  
double moyenne = 12.5;  
char reponse = 'o';
```

```
compteur = 4;    /* affectation d'une nouvelle valeur */  
compteur++;     /* ajoute 1 */
```

- Une variable doit être déclarée avant son utilisation.
- En C17, une déclaration peut apparaître au milieu d'un bloc.
- Une variable locale automatique non initialisée a une valeur indéterminée : il faut lui affecter une valeur avant de la lire.

Types de base et formats d'affichage

Type	Exemple	Format de <code>printf</code>
<code>int</code>	<code>int total = 12;</code>	<code>%d</code>
<code>unsigned int</code>	<code>unsigned int n = 12U;</code>	<code>%u</code>
<code>double</code>	<code>double moyenne = 2.5;</code>	<code>%f</code>
<code>char</code>	<code>char lettre = 'R';</code>	<code>%c</code>
<code>size_t</code>	<code>size_t taille = sizeof total;</code>	<code>%zu</code>

`sizeof` mesure une taille en octets. Les tailles des types dépendent de l'implémentation ; on ne les déduit pas de leur nom.

```
printf("%zu octets\n", sizeof(int));
```

Booléens et constantes

```
#include <stdbool.h>
const int seuil = 10;
bool accepte = false;
accepte = (seuil > 0);
```

- En C17, `<stdbool.h>` fournit `bool`, `true` et `false`.
- Dans une condition, 0 est faux et toute valeur non nulle est vraie.
- `const` empêche de modifier l'objet par ce nom après son initialisation.

Portée des variables

La portée détermine où un nom peut être utilisé.

```
int seuil = 10;
if (seuil > 0) {
    int double_seuil = seuil * 2;
    printf("%d\n", double_seuil);
}
printf("%d\n", seuil);
/* double_seuil n'est plus visible ici */
```

Une variable déclarée dans un bloc est visible après sa déclaration et jusqu'à la fin de ce bloc, y compris dans ses sous-blocs.

Opérateurs arithmétiques et comparaisons

Opérateurs	Opération
<code>+</code> <code>-</code> <code>*</code> <code>/</code>	addition, soustraction, multiplication, division
<code>%</code>	reste d'une division entière
<code><</code> <code><=</code> <code>></code> <code>>=</code>	comparaison d'ordre
<code>==</code> <code>!=</code>	égalité, différence
<code>=</code>	affectation
<code>+=</code> <code>-=</code>	calcul suivi d'une affectation

`total += valeur` équivaut ici à `total = total + valeur`. Une comparaison produit l'entier 0 ou 1. Attention : `x = 3` modifie `x`; `x == 3` compare sa valeur à 3.

Division entière et conversions

```
int somme = 7;  
int nombre = 2;  
double a = somme / nombre;           /* 3.0 */  
double b = (double)somme / nombre;  /* 3.5 */  
double c = (double)(somme / nombre); /* 3.0 */
```

- Deux opérandes entiers donnent une division entière, tronquée vers zéro.
- Convertir avant la division change le calcul.
- Affecter le résultat à un `double` ne refait pas une division déjà effectuée.

Question : quelle valeur est affectée ?

```
int a = 9 / 4;  
int b = 9 % 4;  
double c = 9 / 4;  
double d = 9.0 / 4;  
int e = 2.9;
```

Donner la valeur de chaque variable. Dans quel cas la conversion fait-elle perdre la partie fractionnaire ? Pourquoi `c` et `d` diffèrent-elles ?

Opérateurs logiques et court-circuit

```
if (nombre != 0 && somme / nombre > 2) {  
    puts("Moyenne entiere superieure a 2");  
}  
if (choix == 'o' || choix == 'O') {  
    puts("Oui");  
}
```

`&&` exige deux conditions vraies; `||` en exige au moins une; `!` inverse une condition.

L'évaluation va de gauche à droite et s'arrête dès que le résultat est connu. Dans le premier test, la division n'a pas lieu si `nombre` vaut 0.

Instruction if / else

```
if (note >= 60) {  
    puts("Reussite");  
} else if (note >= 50) {  
    puts("Reprise");  
} else {  
    puts("Echec");  
}
```

- Les conditions sont testées dans l'ordre.
- Le premier bloc dont la condition est vraie est exécuté.
- Le `else` final traite les autres cas; il est facultatif.

Boucle for

```
int total = 0;
for (int i = 1; i <= 4; i++) {
    total += i;
}
printf("%d\n", total);  /* 10 */
```

- ❶ `int i = 1` : initialisation, exécutée une fois.
- ❷ `i <= 4` : condition évaluée avant chaque tour.
- ❸ `total += i` : corps exécuté si la condition est vraie.
- ❹ `i++` : progression, puis retour au test.

La variable `i` déclarée dans le `for` n'est plus visible après la boucle.

Boucles while et do / while

```
int restant = 3;
while (restant > 0) {
    printf("%d\n", restant);
    restant--;
}
```

Le test précède chaque tour. Si `restant` vaut 0 au départ, le corps ne s'exécute jamais. Sans la décrémentation, la boucle ne progresse plus.

Une boucle `do { ... } while (condition);` teste à la fin et effectue toujours un premier tour.

Instructions break, continue et return

```
for (int n = 1; n <= 8; n++) {  
    if (n == 6) break;  
    if (n % 2 == 0) continue;  
    printf("%d ", n);  
}
```

- `break` quitte la boucle la plus proche.
- `continue` passe au tour suivant ; dans un `for`, il exécute d'abord la progression.
- `return` quitte toute la fonction, même depuis une boucle.

Question : quelles valeurs sont affichées ? La valeur 6 est-elle affichée ?

Instruction switch

```
switch (commande) {  
  case 'a':  
    puts("Ajouter");  
    break;  
  case 'q':  
    return 0;  
  default:  
    puts("Commande inconnue");  
    break;  
}
```

Chaque `case` désigne une constante entière. Sans `break` ou `return`, l'exécution continue dans le cas suivant. Pour comparer du texte, on utilisera `strcmp`, pas `switch`.

Définition d'une fonction

```
int maximum(int a, int b)
{
    if (a > b) {
        return a;
    }
    return b;
}
```

- Le premier `int` est le type de retour.
- `a` et `b` sont les paramètres, chacun déclaré avec son type.
- `return` termine l'appel et fournit la valeur du résultat.
- Une fonction de type `void` ne retourne pas de valeur.

Prototype et appel

```
#include <stdio.h>
int maximum(int a, int b); /* prototype */

int main(void)
{
    int resultat = maximum(4, 9);
    printf("%d\n", resultat);
    return 0;
}
/* La definition de maximum peut etre placee ensuite. */
```

Le prototype déclare les types attendus avant l'appel. Ici, les arguments 4 et 9 deviennent les valeurs des paramètres a et b.

Passage des paramètres par valeur

```
int doubler(int valeur)
{
    valeur = valeur * 2;
    return valeur;
}
```

/ Dans main : */*

```
int n = 4;
int resultat = doubler(n);
printf("%d %d\n", n, resultat);
```

Question : que vaut `n` après l'appel ? Qu'affiche `printf` ? Quelle affectation faudrait-il écrire pour remplacer `n` par son double ?

Déclaration et parcours d'un tableau

Un tableau contient un nombre fixé d'éléments de même type, stockés de façon contiguë.

```
int mesures[] = {3, 8, 5};
size_t nombre = sizeof mesures / sizeof mesures[0];
for (size_t i = 0; i < nombre; i++) {
    printf("%d\n", mesures[i]);
}
```

Les indices valides sont 0, 1 et 2. L'accès `mesures[3]` sort du tableau : son comportement est indéfini, même si le programme compile.

Le calcul avec `sizeof` fonctionne ici sur le tableau lui-même.

Pointeurs : adresse et déréférencement

Un pointeur contient l'adresse d'un objet du type indiqué.

```
int total = 4;
int *p = &total;
*p = 7;
printf("%d\n", total); /* 7 */
```

- `&total` donne l'adresse de `total`.
- `p` conserve cette adresse; `*p` désigne l'entier à cette adresse.
- Déclarer un pointeur ne crée pas l'objet qu'il pourrait désigner.
- Il faut un objet encore vivant avant de déréférencer un pointeur.

Tableau passé à une fonction

```
int somme(const int valeurs[], size_t nombre)
{
    int total = 0;
    for (size_t i = 0; i < nombre; i++) {
        total += valeurs[i];
    }
    return total;
}

/* appel : somme(mesures, 3) */
```

Dans un paramètre, `const int valeurs[]` est ajusté en `const int *valeurs`. Le tableau n'est pas copié et sa longueur n'est pas transmise automatiquement. `sizeof valeurs` mesure le pointeur.

Chaînes de caractères

Une chaîne C est une suite de caractères terminée par un octet nul.

```
char mot[8] = "velo";  
/* 'v' 'e' 'l' 'o' '\0' ... */  
printf("%s\n", mot);  
printf("%zu %zu\n", strlen(mot), sizeof mot); /* 4 8 */
```

- 'v' est un caractère; "velo" est une chaîne.
- '\0' marque la fin; ce n'est ni '0' ni le saut de ligne.
- `strlen`, déclarée dans `<string.h>`, compte les octets avant cette fin.
- La capacité comprend la place réservée à l'octet nul.

Question : longueur, capacité et indices

```
char mot[8] = "velo";  
char *p = mot;
```

- ❶ Que valent `strlen(mot)` et `sizeof mot` ?
- ❷ À quel indice se trouve l'octet nul qui termine le texte ?
- ❸ L'accès `mot[8]` est-il valide ?
- ❹ Est-ce que `sizeof p` permet de retrouver la capacité du tableau ?

Comparaison de chaînes : strcmp

```
#include <string.h>
char reponse[] = "oui";
if (strcmp(reponse, "oui") == 0) {
    puts("Accord");
}
```

`strcmp` retourne 0 pour deux contenus égaux, une valeur négative ou positive sinon. `==` appliqué à deux pointeurs compare les adresses.

"oui", "Oui" et "oui\n" sont trois chaînes différentes.

Lecture d'une ligne : fgets

`fgets(tampon, capacite, flux)` lit du texte en limitant le nombre d'octets écrits.

```
#include <stdio.h>
int main(void)
{
    char ligne[32];
    if (fgets(ligne, sizeof ligne, stdin) == NULL) {
        return 1;
    }
    printf("%s", ligne);
    return 0;
}
```

`fgets` lit au plus 31 octets, puis ajoute l'octet nul. Les espaces sont conservés, ainsi que le saut de ligne s'il entre dans le tampon. Le retour vaut `NULL` en cas d'erreur ou de fin d'entrée sans donnée, sinon l'adresse du tampon.

Question : le saut de ligne fait-il partie du texte ?

```
char ligne[8];  
if (fgets(ligne, sizeof ligne, stdin) != NULL) {  
    printf("%zu\n", strlen(ligne));  
    printf("%d\n", strcmp(ligne, "abc") == 0);  
}
```

L'utilisateur tape `abc`, puis Entrée. Quels sont les deux résultats affichés ? Dessiner les caractères placés dans le tableau, jusqu'à l'octet nul.

Exemple : lecture jusqu'à un mot d'arrêt

```
char ligne[32];
while (fgets(ligne, sizeof ligne, stdin) != NULL) {
    ligne[strcspn(ligne, "\n")] = '\0';
    if (strcmp(ligne, "fin") == 0) {
        puts("Lecture terminee");
        return 0;
    }
}
return 1;
```

Ce fragment appartient à `main`, avec `stdio.h` et `string.h`. Il suppose des lignes d'au plus 30 octets de texte : le texte, le saut de ligne et l'octet nul tiennent ensemble dans 32 octets.

Allocation dynamique : malloc et free

`malloc(n)` réserve `n` octets et retourne leur adresse, ou `NULL` en cas d'échec. `free` libère le bloc obtenu.

```
/* stdio.h et stdlib.h; fragment dans main */
const int capacite = 32;
char *ligne = malloc((size_t)capacite);
if (ligne == NULL) return 1;
if (fgets(ligne, capacite, stdin) == NULL) {
    free(ligne);
    return 1;
}
printf("%s", ligne);
free(ligne);
return 0;
```

La mémoire obtenue n'est pas initialisée. `sizeof ligne` mesure le pointeur, pas l'allocation. Après `free`, aucun accès au bloc n'est permis.

Arguments du programme : argc et argv

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s chemin\n", argv[0]);
        return 1;
    }
    printf("Chemin reçu : %s\n", argv[1]);
    return 0;
}
```

`./outil "rapport final.txt"` fournit un argument après le nom du programme. `argc` vaut 2; `argv[1]` contient le chemin entier.

`stdin` reçoit les données tapées ou redirigées. `stderr` reçoit les diagnostics; le code de retour indique le succès ou l'échec au shell.

Écriture dans un fichier : fopen, fputs, fclose

```
FILE *sortie = fopen(argv[1], "w");
if (sortie == NULL) { perror(argv[1]); return 1; }
int statut = 0;
if (fputs("Resultat disponible\n", sortie) == EOF) {
    statut = 1;
}
if (fclose(sortie) == EOF) statut = 1;
return statut;
```

Après vérification des arguments, ce fragment utilise `<stdio.h>`. Le mode "w" crée le fichier ou efface son contenu existant; "r" ouvre un fichier existant en lecture.

Lecture d'un fichier : fgetc et EOF

```
/* entree est un FILE * ouvert et verifie */  
int octet;  
while ((octet = fgetc(entree)) != EOF) {  
    putchar(octet);  
}  
int statut = 0;  
if (ferror(entree)) statut = 1;  
fclose(entree);  
return statut;
```

`fgetc` retourne un octet ou `EOF`; on conserve son résultat dans un `int`, pas dans un `char`. `ferror` distingue une erreur d'une fin normale. Ici, les écritures sur le terminal sont supposées réussir.

Exercices : tableaux et lecture de lignes

- ❶ Écrire une fonction qui compte les valeurs supérieures à un seuil dans un tableau d'entiers. Recevoir aussi le nombre de cases.
- ❷ Pour `{3, 8, 5}` et le seuil 4, prévoir puis vérifier le résultat 2.
- ❸ Écrire un second programme qui lit des lignes jusqu'à la réponse exacte `oui`, puis affiche `Accord` et retourne 0.
- ❹ Retourner 1 si l'entrée se termine avant cette réponse. Les lignes d'essai contiennent au plus 30 octets de texte.

Compilation et cas de test

```
gcc -std=c17 -Wall -Wextra -Wpedantic accord.c -o accord
printf 'non\noui\n' | ./accord
echo "$?"                                # 0
printf 'Oui\nouistiti\n' | ./accord
echo "$?"                                # 1
printf 'oui' | ./accord                  # sans saut de ligne final
./accord < /dev/null                     # entree vide
```

Pour le tableau : essayer zéro élément, aucune valeur au-dessus du seuil, toutes les valeurs au-dessus et une valeur égale au seuil.