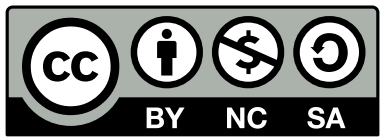
INF1135
Programmation procédurale
Chapitre 4 — Variables

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Pourquoi des fonctions ?

- Jusqu'ici : tout le programme dans `main` — long, répétitif, difficile à lire
- **Fonction** : suite d'instructions **nommée**, qu'on peut exécuter à la demande
- **Découper** un programme en petites tâches, chacune avec un nom explicite
- **Réutiliser** : écrire une fois, appeler autant de fois que nécessaire
- Déjà utilisées sans le savoir : `printf`, `scanf`… et `main` elle-même !

Dans ce chapitre

Seulement les fonctions **sans paramètre** et **sans valeur de retour** — on y reviendra plus tard.

Exemple : fonctions sans paramètre

Fichier `functions.c` :

```c
#include <stdio.h>

void afficher_ligne(void) {
  printf("--------------------\n");
}

void afficher_entete(void) {
  afficher_ligne();
  printf("  Mesures du capteur\n");
  afficher_ligne();
}

int main(void) {
  printf("debut de main\n");
  afficher_entete();
  printf("tension : 5 V\n");
  afficher_ligne();
  printf("fin de main\n");
  return 0;
}
```

Exemple : exécution

```
$ gcc -Wall -Wextra -std=c23 -o functions functions.c
$ ./functions
debut de main
--------------------
  Mesures du capteur
--------------------
tension : 5 V
--------------------
fin de main
```

- Les deux premières lignes pointillées : les deux appels à `afficher_ligne` faits **dans** `afficher_entete`
- La dernière : l'appel fait directement dans `main`
- Les lignes s'affichent dans l'ordre des **appels**, pas dans l'ordre où les fonctions sont écrites dans le fichier

Définir et appeler une fonction

Définition

```
void afficher_ligne(void) {
  printf("--------------------\n");
}
```

Appel

```
afficher_ligne();
```

- `void` devant le nom : la fonction ne retourne **aucune valeur**
- `(void)` après le nom : la fonction ne prend **aucun paramètre**
- Corps entre `{ }` : exécuté à chaque **appel**
- Parenthèses **obligatoires** à l'appel, même vides
- Définir la fonction **avant** (plus haut que) son premier appel

Flot d'exécution

- Le programme commence toujours par `main`
- À l'**appel** : l'exécution **saute** au début de la fonction
- À la fin du corps : l'exécution **revient** juste après l'appel
- Une fonction peut en appeler une autre : les retours se font dans l'ordre **inverse** des appels

Plus tard

- **Paramètres** : transmettre des valeurs à une fonction
- **Valeur de retour** : obtenir un résultat avec `return`
- **Prototypes** : appeler une fonction définie plus bas ou ailleurs
- **Récursion** : une fonction qui s'appelle elle-même

Pour la suite de ce chapitre

Chaque fonction est un **bloc** : ses variables suivent les règles de portée et de durée de vie qui suivent.

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Exemple : portée et blocs imbriqués

Fichier `scope.c` :

```c
#include <stdio.h>

int x = 1;  // portee globale

int main(void) {
  printf("globale : x=%d\n", x);
  int x = 2;  // portee locale, masque la globale
  printf("locale  : x=%d\n", x);
  {
    int x = 3;  // portee de bloc, masque la locale
    int y = 10;
    printf("bloc    : x=%d, y=%d\n", x, y);
  }
  printf("apres   : x=%d\n", x);
  // printf("%d\n", y);  // erreur : y hors de portee
  for (int i = 0; i < 2; i++) {
    printf("boucle  : i=%d\n", i);
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o scope scope.c
$ ./scope
globale : x=1
locale  : x=2
bloc    : x=3, y=10
apres   : x=2
boucle  : i=0
boucle  : i=1
```

Notion de portée

- **Portée** (*scope*) : région du code source où un identificateur est **visible**, c'est-à-dire utilisable par son nom
- Déterminée **à la compilation**, selon l'endroit où la variable est **déclarée**
- Hors de sa portée, une variable n'existe pas pour le compilateur → erreur `undeclared`

Trois niveaux dans ce cours

→ **Locale** : déclarée dans une fonction

→ **De bloc** : déclarée dans un bloc `{ }`

→ **Globale** : déclarée hors de toute fonction

À ne pas confondre

Portée (**où** le nom est visible) $\neq$ durée de vie (**quand** la variable existe en mémoire).

Portée locale

Exemple

```
int main(void) {
   int n = 5;          // visible a partir d'ici...
   printf("%d\n", n);
   return 0;
}                      // ... jusqu'ici
```

- Variable **locale** : déclarée à l'intérieur d'une fonction
- Visible de sa **déclaration** jusqu'à la **fin** de la fonction
- Invisible depuis les autres fonctions : deux fonctions peuvent avoir chacune leur propre `n`, sans conflit
- Utilisation **avant** la déclaration : erreur de compilation

Portée de bloc

Exemple

```
if (n > 0) {
   int carre = n * n;    // visible dans le bloc
   printf("%d\n", carre);
}
// carre n'existe plus ici
for (int i = 0; i < n; i++) {
   ...                   // i visible dans la boucle
}
// i n'existe plus ici
```

- **Bloc** : instructions entre { et } (`if`, `while`, `for`, ou bloc seul)
- Visible de sa déclaration jusqu'à l'accolade **fermante**
- Bonne pratique : déclarer une variable dans le **plus petit bloc** où elle est utilisée

Portée globale

Exemple

```c
#include <stdio.h>

int compteur = 0;    // globale

int main(void) {
   compteur++;          // accessible partout dans le fichier
   printf("%d\n", compteur);
   return 0;
}
```

- Variable **globale** : déclarée **hors** de toute fonction, généralement en haut du fichier
- Visible de sa déclaration jusqu'à la **fin du fichier**, dans **toutes** les fonctions qui suivent
- Initialisée automatiquement à 0 si aucune valeur n'est donnée

Masquage

- Une déclaration dans un bloc interne peut réutiliser un nom déjà déclaré dans un bloc englobant
- La déclaration la plus **interne** gagne : elle **masque** (*shadows*) l'autre jusqu'à la fin du bloc
- La variable masquée n'est pas modifiée — elle est seulement **inaccessible** par son nom

Source de bogues

Le masquage est légal, mais rarement voulu. L'option `-Wshadow` le signale :

```
$ gcc -Wall -Wextra -Wshadow -std=c23 -c shadow.c
shadow.c:3:7: warning: declaration of 'x' shadows a global
declaration [-Wshadow]
    3 |   int x = 2;
      |       ^
```

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Notion de durée de vie

- **Durée de vie** : période de l'**exécution** pendant laquelle une variable occupe réellement de la mémoire
- Avant et après : aucune mémoire réservée — la valeur est perdue
- Propriété **dynamique** (à l'exécution), contrairement à la portée (à la compilation)

Trois durées de vie

→ **Automatique** : le temps d'un bloc

→ **Statique** : toute l'exécution du programme

→ **Dynamique** : contrôlée explicitement par le programmeur

Durée de vie automatique

- Cas **par défaut** des variables locales et de bloc
- **Créée** à l'entrée dans le bloc, **détruite** à la sortie
- Réservée sur la **pile** (*stack*) : allocation et libération très rapides, gérées par le compilateur
- Chaque entrée dans le bloc crée une **nouvelle** variable : la valeur de l'itération ou de l'appel précédent n'est pas conservée

Initialisation

Une variable automatique **non initialisée** contient une valeur **indéterminée**. Toujours initialiser avant de lire !

Durée de vie statique

- Variable créée **avant** l'exécution de `main` et détruite à la **fin** du programme
- Concerne : les variables **globales** et les variables locales déclarées `static`
- Initialisée **une seule fois**, avant le démarrage
- Sans initialisation explicite : valeur `0` garantie
- Placée dans une zone de mémoire dédiée (**données statiques**), distincte de la pile

Contrainte

L'initialisation doit être une **expression constante** (ex. `static int n = 10;`, mais pas `static int n = f();`).

Durée de vie dynamique

- Mémoire réservée **à la demande**, pendant l'exécution
- Existe jusqu'à ce qu'elle soit **explicitement** libérée
- Réservée dans le **tas** (*heap*)
- Utile quand la taille ou la durée d'utilisation ne sont connues qu'à l'exécution

Aperçu (pas à maîtriser pour l'instant)

```
int *p = malloc(100 * sizeof(int));  // reservation
...
free(p);                             // liberation
```

Notions vues plus tard

`int *` : **pointeurs** — `malloc`/`free` : **gestion de la mémoire**.

Portée vs durée de vie

Variable	**Portée**	**Durée de vie**	**Défaut**
Locale	bloc	automatique	indéterminée
Locale `static`	bloc	statique	0
Globale	fichier	statique	0
Dynamique	—	dynamique	indéterminée

- Les deux notions sont **indépendantes** : une variable locale `static` a une petite portée mais une longue durée de vie
- Une zone dynamique n'a pas de nom : on y accède par son **adresse**

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Vue d'ensemble

- **Classe de stockage** : mot-clé placé devant une déclaration pour préciser la durée de vie et/ou la visibilité
- Au plus **une** classe de stockage par déclaration

Mot-clé	**Effet**
`auto`	durée de vie automatique (défaut des locales)
`static`	durée de vie statique / visibilité limitée au fichier
`extern`	déclare une variable définie ailleurs
`register`	suggère un registre du processeur

En pratique

`static` est de loin le plus utilisé ; `extern` devient utile avec la compilation séparée.

auto

Exemple

```
auto int n = 5;    // equivalent a: int n = 5;
```

- Classe par **défaut** des variables locales et de bloc
- Donc **redondant** : le mot-clé n'est presque jamais écrit
- Interdit au niveau global

Attention : C23

Depuis C23, `auto` sans type demande au compilateur de **déduire** le type : `auto x = 3.5;` déclare un `double`. Même sens qu'en C++.

static

Sur une variable locale

```
void f(void) {
  static int appels = 0;  // cree une seule fois
  appels++;
}
```

Portée inchangée (le bloc), mais durée de vie **statique** : la valeur est **conservée** entre les appels.

Sur une variable globale

```
static int total = 0;      // globale privee au fichier
```

Durée de vie inchangée, mais visibilité limitée au **fichier** .c courant — invisible depuis les autres fichiers du programme.

extern

Exemple

```
extern int compteur;    // declaration: existe ailleurs

int main(void) {
  compteur++;
  return 0;
}

int compteur = 0;       // definition: reserve la memoire
```

- **Déclaration** : annonce le nom et le type — aucune mémoire réservée
- **Définition** : réserve la mémoire — une **seule** par variable
- Utile surtout avec la **compilation séparée** (chapitre sur la modularité)

register

- Suggère au compilateur de garder la variable dans un **registre** du processeur plutôt qu'en mémoire
- Simple **indication** : les compilateurs modernes optimisent mieux seuls et l'ignorent
- Conséquence qui demeure : impossible d'obtenir l'**adresse** de la variable

```
$ gcc -std=c23 -c reg.c
reg.c:4:3: error: address of register variable 'r' requested
    4 |   printf("%p\n", (void *) &r);
      |   ^~~~~~
```

Dans ce cours

À connaître pour lire du code ancien — inutile d'en écrire.

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Exemple : compteur d'appels

Fichier `counter.c` :

```c
#include <stdio.h>

void compter(void) {
  int automatique = 0;
  static int statique = 0;
  automatique++;
  statique++;
  printf("automatique=%d, statique=%d\n", automatique, statique);
}

int main(void) {
  for (int i = 0; i < 3; i++) {
    compter();
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o counter counter.c
$ ./counter
automatique=1, statique=1
automatique=1, statique=2
automatique=1, statique=3
```

Comportement entre appels

- Variable locale `static` : **visible** seulement dans sa fonction, mais **existe** pendant tout le programme
- Initialisation effectuée **une seule fois**, avant le premier appel — pas à chaque appel
- Chaque appel retrouve la valeur laissée par l'appel **précédent**

Rappel

Chaque appel d'une fonction **entre** dans son bloc : ses variables automatiques sont **recréées**, et leur valeur précédente est perdue. Une variable `static` échappe à cette règle.

Cas d'utilisation

Compter

Nombre d'appels d'une fonction, nombre de mesures traitées, identifiants uniques croissants.

Initialiser une seule fois

Effectuer un travail coûteux au premier appel seulement (ex. calibration d'un capteur), puis mémoriser qu'il a été fait.

Mémoriser un état

Garder la dernière valeur lue pour détecter un changement (ex. front montant d'un signal).

Limite

L'état est **unique** et **caché** : une fonction avec une variable `static` ne se comporte pas toujours de la même façon pour les mêmes arguments — plus difficile à tester.

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Variables globales

Exemple

```c
#include <stdio.h>

double tension = 5.0;          // globale initialisee
int nb_mesures;                // globale: vaut 0

void mesurer(void) {
  nb_mesures++;                // modifiable ici...
}

int main(void) {
  mesurer();
  printf("%d\n", nb_mesures);  // ... et lisible ici
  return 0;
}
```

Variables globales : explications

- Déclarées **hors** de toute fonction, en haut du fichier
- `tension` : initialisée explicitement à `5.0`
- `nb_mesures` : aucune valeur donnée → vaut `0` (garanti pour une globale)
- Durée de vie **statique** : existent pendant toute l'exécution
- Portée : tout le fichier, à partir de la déclaration

Conséquence

`mesurer` **modifie** `nb_mesures` et `main` la **lit**, sans que la variable soit transmise d'une fonction à l'autre : toutes les fonctions partagent la même variable.

Exemple : effet de bord caché

Fichier `globalbug.c` :

```c
#include <stdio.h>

int i;  // globale partagee par tout le fichier

void afficher_ligne(void) {
  for (i = 0; i < 5; i++) {
    printf("*");
  }
  printf("\n");
}

int main(void) {
  for (i = 0; i < 3; i++) {  // on veut 3 lignes...
    afficher_ligne();
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o globalbug globalbug.c
$ ./globalbug
*****
```

- Une seule ligne au lieu de trois !
- `afficher_ligne` laisse `i` à 5 $\rightarrow$ la boucle de `main` s'arrête
- Correction : `for (int i = 0; ...)` dans chaque boucle

Dangers des variables globales

- **Effets de bord cachés** : n'importe quelle fonction peut modifier la variable, sans que cela paraisse à l'appel
- **Couplage** : les fonctions dépendent d'un état partagé — impossible de les comprendre ou de les réutiliser isolément
- **Tests** difficiles : le résultat d'une fonction dépend de ce qui a été exécuté avant
- **Débogage** difficile : une valeur erronée peut venir de n'importe où dans le programme
- **Collisions de noms** et masquage involontaire

Dans le cours

Ne jamais utiliser de variable globale.

Bonnes pratiques

- Préférer les variables **locales**, déclarées dans le plus petit bloc possible
- Transmettre les données par **paramètres** et **valeurs de retour** aux fonctions (plus tard)
- Constantes partagées : `#define` ou globale `const` — aucune modification possible
- Si une globale modifiable est vraiment nécessaire :
  - → la déclarer `static` pour la limiter au fichier
  - → lui donner un nom **explicite** (jamais `i`, `n`, `tmp`)
  - → documenter qui la lit et qui la modifie

Plan

1. Fonctions
2. Portée
3. Durée de vie
4. Classes de stockage
5. Variables locales statiques
6. Variables globales
7. Adresses mémoire

Exemple : observer les adresses

Fichier `addresses.c` :

```c
#include <stdio.h>

int globale = 1;

int main(void) {
  static int statique = 2;
  int locale = 3;
  double reel = 4.0;
  printf("globale  : %p\n", (void *) &globale);
  printf("statique : %p\n", (void *) &statique);
  printf("locale   : %p\n", (void *) &locale);
  printf("reel     : %p\n", (void *) &reel);
  printf("taille   : %zu et %zu octets\n", sizeof locale, sizeof reel);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o addresses addresses.c
$ ./addresses
globale  : 0x59a431050010
statique : 0x59a431050014
locale   : 0x7ffef621bf7c
reel     : 0x7ffef621bf80
taille   : 4 et 8 octets
```

- `globale` et `statique` : voisines ; `locale` et `reel` : très loin, ailleurs
- Adresses **différentes** à chaque exécution

Des adresses qui changent : l'ASLR

```
$ ./addresses
globale   : 0x62294b588010
statique  : 0x62294b588014
locale    : 0x7ffebbb408cc
reel      : 0x7ffebbb408d0
$ ./addresses
globale   : 0x5b744f91c010
statique  : 0x5b744f91c014
locale    : 0x7ffd6ab38bbc
reel      : 0x7ffd6ab38bc0
```

- **ASLR** (*Address Space Layout Randomization*) : à chaque lancement, le système d'exploitation place la pile, les données et le code à des adresses de départ **aléatoires**
- Les **écarts** à l'intérieur d'une zone ne changent pas : `statique` est toujours 4 octets après `globale`

Pourquoi ?

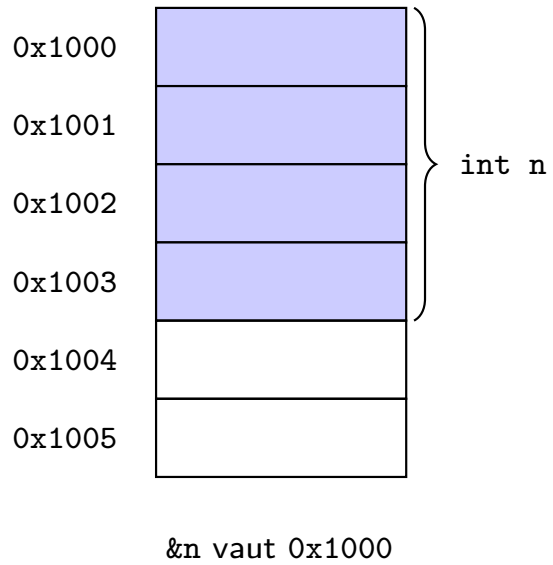
Une attaque exploitant une faille mémoire doit souvent connaître l'adresse exacte d'une donnée ou d'une fonction. Des adresses imprévisibles rendent ces attaques beaucoup plus difficiles.

À savoir

Sous gdb, l'ASLR est **désactivé** par défaut : les adresses y sont identiques d'une exécution à l'autre.

Notion d'adresse

- Mémoire : grande suite d'**octets** numérotés
- **Adresse** : numéro d'un octet
- Une variable occupe `sizeof` octets consécutifs
- Son adresse : celle de son **premier** octet
- Adresses habituellement écrites en **hexadécimal**



Opérateur &

- `&x` : **adresse** de la variable `x`
- Déjà rencontré avec `scanf("%d", &n)` : `scanf` a besoin de savoir **où** écrire la valeur lue
- Affichage avec `printf` et le format %p, en convertissant en `(void *)`

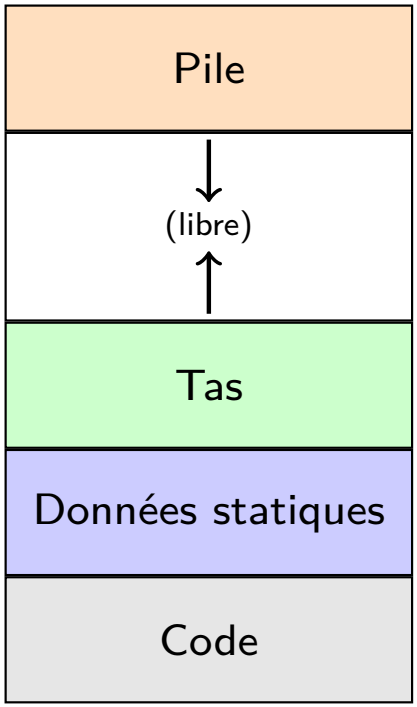
Exemple

```
int n = 42;
printf("valeur  : %d\n", n);
printf("adresse : %p\n", (void *) &n);
```

Attention

& unaire (adresse) $\neq$ & binaire (ET bit à bit) $\neq$ && (ET logique).

Organisation de la mémoire



- **Pile** : variables automatiques — grandit et rétrécit au fil des blocs et des appels
- **Tas** : mémoire dynamique (`malloc`)
- **Données statiques** : globales et locales `static`
- **Code** : instructions du programme

Dans `addresses.c`

`0x7ffe...` → pile

`0x59a4...` → données statiques