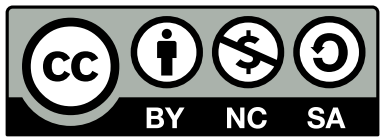
INF1135
Programmation procédurale
Chapitre 5 — Fonctions

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Rappel et motivation

- Vu précédemment : `void f(void)` — une fonction qui fait **toujours le même travail**
- Comment convertir **plusieurs** températures en Fahrenheit ?

Sans fonction adaptée

```
double f1 = 0.0 * 9.0 / 5.0 + 32.0;
double f2 = 25.0 * 9.0 / 5.0 + 32.0;
double f3 = t * 9.0 / 5.0 + 32.0;
```

- La **formule** est recopiée à chaque utilisation
- Une erreur dans une copie est difficile à repérer

Ce qu'il manque

Transmettre une valeur à la fonction, et **obtenir** un résultat en retour.

Exemple : conversion de températures

Fichier `conversion.c` :

```c
#include <stdio.h>

double celsius_vers_fahrenheit(double celsius) {
  return celsius * 9.0 / 5.0 + 32.0;
}

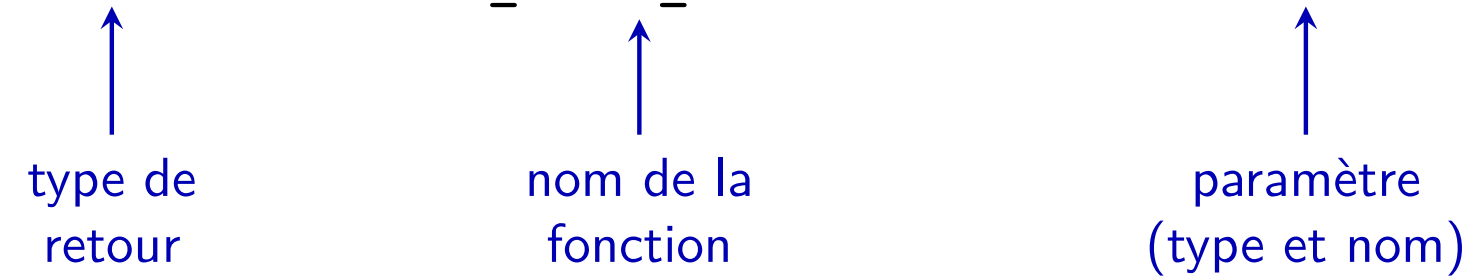
int main(void) {
  printf("%.1f C = %.1f F\n", 0.0, celsius_vers_fahrenheit(0.0));
  printf("%.1f C = %.1f F\n", 25.0, celsius_vers_fahrenheit(25.0));
  double t = -40.0;
  double f = celsius_vers_fahrenheit(t);
  printf("%.1f C = %.1f F\n", t, f);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o conversion conversion.c
$ ./conversion
0.0 C = 32.0 F
25.0 C = 77.0 F
-40.0 C = -40.0 F
```

Paramètres

Anatomie

```
double celsius_vers_fahrenheit(double celsius)
```



- **Paramètre** : variable déclarée entre les parenthèses de la **définition**, avec son type
- **Argument** : valeur fournie à l'**appel** — une constante, une variable ou une expression
- Plusieurs paramètres : séparés par des virgules, chacun avec son **propre type** — `int maximum(int a, int b)`, et non `int maximum(int a, b)`
- Les arguments sont associés aux paramètres **dans l'ordre** : `maximum(x + 1, 10)` donne `a = x + 1`, `b = 10`
- Si les types diffèrent, l'argument est **converti** implicitement : `celsius_vers_fahrenheit(25)` reçoit `25.0`

Valeur de retour

Exemple

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

- `return expression;` : termine **immédiatement** la fonction et renvoie la valeur à l'appelant
- La valeur est convertie dans le **type de retour** déclaré
- Plusieurs `return` possibles : le premier atteint gagne
- L'appel est une **expression** : `int m = maximum(x, y) + 1;`
- Fonction `void` : aucune valeur, mais `return;` permet d'en sortir plus tôt

Et `main` ?

`main` retourne un `int` : le **code de sortie** du programme, vu précédemment (`echo $?`).

Erreurs fréquentes

Oublier un `return`

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

```
$ gcc -Wall -Wextra -std=c23 -c noreturn.c
noreturn.c: In function 'maximum':
noreturn.c:3:1: warning: control reaches end of non-void
function [-Wreturn-type]
    3 | }
      | ^
```

Si `a <= b`, la valeur retournée est **indéfinie**.

- **Ignorer** la valeur retournée : `celsius_vers_fahrenheit(25.0);` compile, mais le résultat est perdu
- Mauvais **nombre** d'arguments : erreur `too many/too few arguments`
- Confondre **afficher** (`printf`) et **retourner** (`return`) : une fonction de calcul devrait retourner son résultat, et laisser l'appelant décider quoi en faire

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Appeler une fonction définie plus bas

Fichier `noproto.c` :

```c
#include <stdio.h>

int main(void) {
  printf("%.1f\n", carre(3.0));
  return 0;
}

double carre(double x) {
  return x * x;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o noproto noproto.c
noproto.c: In function 'main':
noproto.c:4:20: warning: implicit declaration of
      function 'carre'
[-Wimplicit-function-declaration]
    4 |   printf("%.1f\n", carre(3.0));
      |                     ^~~~~
noproto.c: At top level:
noproto.c:8:8: error: conflicting types for 'carre';
have 'double(double)'
    8 | double carre(double x) {
      |        ^~~~~
noproto.c:4:20: note: previous implicit declaration of
      'carre'
with type 'int()'
```

- Le compilateur lit le fichier **de haut en bas** : à la ligne 4, il ne connaît pas encore `carre`
- Il **devine** alors `int carre()`, ce qui contredit la définition qui suit
- La norme **interdit** la déclaration implicite depuis C99, mais exige seulement un **diagnostic** : GCC se contentait d'un avertissement, il en fait une **erreur** depuis la version 14

Déclaration vs définition

Fichier `proto.c` :

```c
#include <stdio.h>

double carre(double x);   // prototype

int main(void) {
  printf("%.1f\n", carre(3.0));
  return 0;
}

double carre(double x) {  // definition
  return x * x;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o proto
      proto.c
$ ./proto
9.0
```

- **Prototype** (déclaration) : type de retour, nom et types des paramètres, suivis de `;`
- **Définition** : la même signature (**identique**), suivie du **corps**
- Le prototype suffit au compilateur pour **vérifier** chaque appel : nombre et types des arguments, type du résultat
- Les noms des paramètres sont **facultatifs** dans le prototype (`double carre(double);`), mais aident la lecture

Organisation d'un fichier

Gabarit

```c
#include <stdio.h>

// ---------- Prototypes ----------

double carre(double x);
int maximum(int a, int b);

// ---------- Definitions ---------

double carre(double x) {
  ...
}

int maximum(int a, int b) {
  ...
}

// ------ Programme principal -----

int main(void) {
  ...
}
```

- **Prototypes** en haut : vue d'ensemble des fonctions du fichier, comme une table des matières
- **Définitions** ensuite : leur ordre n'a plus d'importance, chacune peut appeler les autres
- `main` à la **fin** : le point d'entrée du programme est facile à trouver
- Chaque partie est annoncée par un **commentaire de section**
- `#include <stdio.h>` insère les **prototypes** de `printf`, `scanf`, etc. — on reviendra sur ces fichiers `.h`

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Une fonction peut-elle modifier ses arguments ?

Fichier `byvalue.c` :

```c
#include <stdio.h>

void doubler(int x) {
  x = 2 * x;
  printf("dans doubler : x = %d\n", x);
}

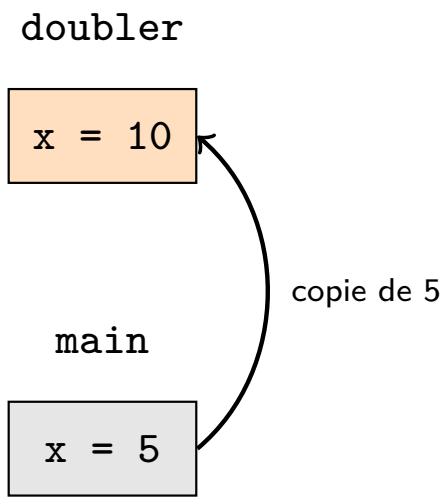
void echanger(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
  printf("dans echanger : a = %d, b = %d\n", a, b);
}

int main(void) {
  int x = 5;
  doubler(x);
  printf("dans main : x = %d\n", x);
  int a = 1, b = 2;
  echanger(a, b);
  printf("dans main : a = %d, b = %d\n", a, b);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o byvalue
      byvalue.c
$ ./byvalue
dans doubler : x = 10
dans main : x = 5
dans echanger : a = 2, b = 1
dans main : a = 1, b = 2
```

- `x` est doublé dans `doubler`… mais pas dans `main`
- `a` et `b` sont échangés dans `echanger`… mais pas dans `main`

Copie des arguments



- Chaque paramètre est une **variable locale** de la fonction
- À l'appel, elle est initialisée avec une **copie** de la valeur de l'argument
- Modifier le paramètre modifie la **copie**, jamais l'original
- À la fin de l'appel, la copie **disparaît**

Lien avec ce qu'on a vu

Les paramètres suivent les mêmes règles que les variables locales : **portée** limitée à la fonction, durée de vie **automatique**. Même si les noms sont identiques (`x`, `a` et `b` dans les fonctions et dans `main`), ce sont des variables **distinctes**.

Plus tard

- Pour qu'une fonction modifie une variable de l'appelant, il faut lui transmettre l'**adresse** de cette variable : le **passage par adresse**
- On y reviendra avec les **pointeurs**

On l'a déjà fait !

```
scanf("%d", &n);
```

`scanf` doit **modifier** `n` : on lui passe donc son adresse `&n`, et non une copie de sa valeur.

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Pourquoi documenter une fonction ?

- Le prototype indique **comment** appeler une fonction, pas **ce qu'elle fait** ni **dans quelles conditions**
- On veut pouvoir **utiliser** une fonction sans lire son corps : un collègue, un correcteur, ou vous-même dans six mois
- Documenter le **quoi**, pas le **comment** : le code montre déjà le comment

Docstring

Commentaire de documentation placé juste avant une fonction (ou son prototype). Le terme vient de l'anglais *documentation string*.

Exemple : fonctions documentées

```c
/**
 * Convertit une temperature de degres Celsius en degres
 * Fahrenheit.
 *
 * @param celsius  La temperature en degres Celsius
 * @return         La temperature en degres Fahrenheit
 */
double celsius_vers_fahrenheit(double celsius);

/**
 * Calcule la factorielle d'un entier.
 *
 * L'entier n doit etre entre 0 et 20 : au-dela de 20, le
 * resultat ne tient pas dans un long.
 *
 * @param n  L'entier dont on calcule la factorielle
 * @return   n! = 1 * 2 * ... * n, avec 0! = 1
 */
long factorielle(int n);
```

Format exigé dans le cours

Pour le moment, **toute fonction** doit être précédée d'une *docstring* `/** ... */` contenant, dans l'ordre :

1. Une **phrase** résumant ce que fait la fonction
2. *(optionnel)* Un ou plusieurs **paragraphes** de détails : valeurs admises, cas particuliers, etc.
3. `@param` : description de **chaque** paramètre
4. `@return` : description de la valeur retournée (absent pour une fonction `void`)

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Une fonction qui s'appelle elle-même

Définition mathématique de la factorielle

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

- La définition de $n!$ utilise $(n-1)!$: elle est **récursive**
- En C, une fonction peut s'**appeler elle-même** : la traduction est directe

Traduction en C

```
long factorielle(int n) {
  if (n == 0) return 1;                // cas de base
  return n * factorielle(n - 1);       // cas recursif
}
```

Exemple : factorielle

Fichier `factorial.c` :

```c
#include <stdio.h>

long factorielle(int n) {
  if (n == 0) return 1;             // cas de base
  return n * factorielle(n - 1);    // cas recursif
}

int main(void) {
  printf("3! = %ld\n", factorielle(3));
  printf("10! = %ld\n", factorielle(10));
  printf("20! = %ld\n", factorielle(20));
  printf("21! = %ld\n", factorielle(21));
  printf("22! = %ld\n", factorielle(22));
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o factorial factorial.c
$ ./factorial
3! = 6
10! = 3628800
20! = 2432902008176640000
```

Cas de base et cas récursif

Toute fonction récursive a besoin de

- Au moins un **cas de base** : résultat calculé **sans** appel récursif
- Un **cas récursif** : appel sur un problème **plus petit**, qui se rapproche du cas de base

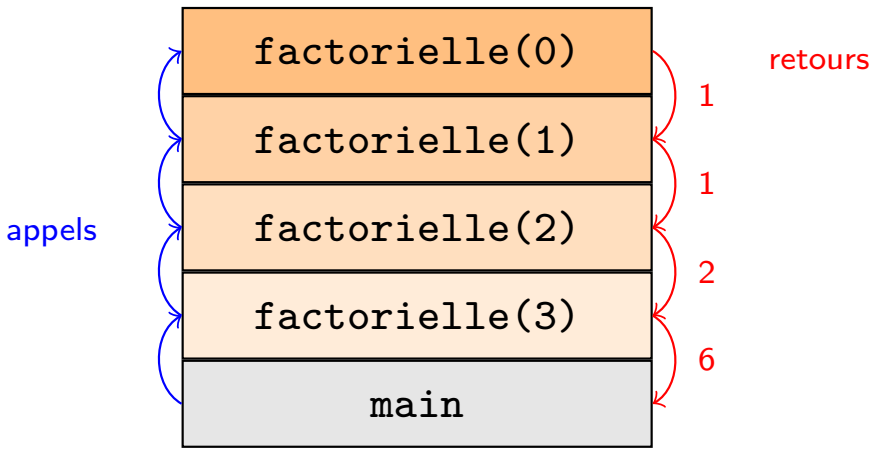
Récursion infinie

Si le cas de base n'est jamais atteint, les appels s'accumulent jusqu'à épuiser la mémoire réservée aux appels :

```
factorielle(-1)  ->  factorielle(-2)  ->  factorielle(-3)  ->  ...
Segmentation fault (core dumped)
```

D'où l'importance d'indiquer dans la documentation que `n` doit être positif ou nul.

Pile d'appels



- Chaque appel reçoit son propre **cadre** sur la **pile** : ses paramètres et ses variables locales
- Quatre variables `n` distinctes coexistent : 3, 2, 1, 0
- Le dernier appel commencé est le premier terminé
- Au retour, le cadre est **retiré** de la pile

Lien avec ce qu'on a vu

C'est la **pile** de l'organisation de la mémoire : les variables automatiques y naissent à l'appel et y meurent au retour.

Exemple : suite de Fibonacci

Définition

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ si } n \geq 2$$

Fichier `fibonacci.c` :

```c
#include <stdio.h>

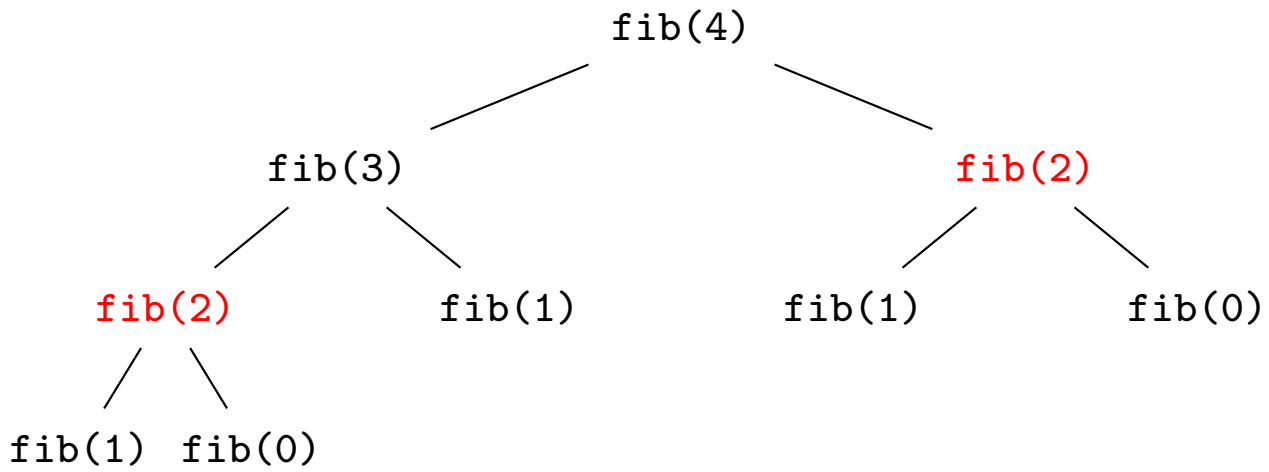
long fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
  for (int n = 0; n <= 45; n++) {
    printf("fibonacci(%d) = %ld\n", n, fibonacci(n));
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o fibonacci
    fibonacci.c
$ ./fibonacci
0 1 1 2 3 5 8 13 21 34 55
fibonacci(40) = 102334155
```

Deux appels récursifs par appel... et `fibonacci(40)` prend environ **une seconde** !

Fibonacci : arbre des appels

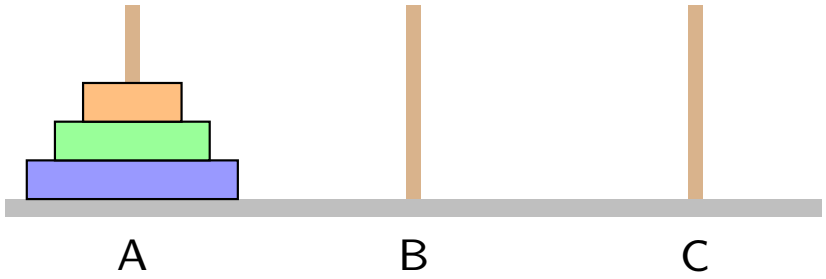


- `fib(2)` est calculé **deux fois**, `fib(1)` trois fois : les mêmes calculs sont **répétés**
- `fib(4)` : 9 appels ; `fib(40)` : plus de **331 millions** d'appels
- En général : $2F_{n+1} - 1$ appels, qui croissent aussi vite que la suite elle-même

Leçon

Une solution récursive peut être **élégante** et pourtant très **inefficace**.

Exemple : tours de Hanoï



Problème

Déplacer les n disques de la tige `A` vers la tige `C` :

- un seul disque à la fois
- jamais un disque sur un plus petit

Idée récursive

Pour déplacer n disques de `A` vers `C` :

1. déplacer les $n-1$ disques du dessus de `A` vers `B` (récursion)
2. déplacer le plus grand disque de `A` vers `C`
3. déplacer les $n-1$ disques de `B` vers `C` (récursion)

Cas de base : $n = 0$, il n'y a rien à faire.

Exemple : tours de Hanoï en C

Fichier `hanoi.c` :

```c
#include <stdio.h>

/**
 * Affiche les deplacements qui resolvent les tours de Hanoi.
 *
 * Les n disques du dessus de la tige de depart sont deplaces
 * vers la tige d'arrivee, en passant par la tige intermediaire.
 * L'entier n doit etre positif ou nul.
 *
 * @param n     Le nombre de disques a deplacer
 * @param de    La tige de depart
 * @param vers  La tige d'arrivee
 * @param via   La tige intermediaire
 */
void hanoi(int n, char de, char vers, char via) {
  if (n == 0) return;
  hanoi(n - 1, de, via, vers);
  printf("disque %d : %c -> %c\n", n, de, vers);
  hanoi(n - 1, via, vers, de);
}

int main(void) {
  int n;
  printf("Nombre de disques: ");
  scanf("%d", &n);
  hanoi(n, 'A', 'C', 'B');
  return 0;
}
```

Exemple : tours de Hanoï, exécutions

```
$ gcc -Wall -Wextra -std=c23 -o hanoi hanoi.c
```

```
$ ./hanoi
Nombre de disques: 0
$ ./hanoi
Nombre de disques: 1
disque 1 : A -> C
$ ./hanoi
Nombre de disques: 2
disque 1 : A -> B
disque 2 : A -> C
disque 1 : B -> C
```

```
$ ./hanoi
Nombre de disques: 3
disque 1 : A -> C
disque 2 : A -> B
disque 1 : C -> B
disque 3 : A -> C
disque 1 : B -> A
disque 2 : B -> C
disque 1 : A -> C
```

- Les rôles des tiges **changent** d'un appel à l'autre : ce sont des paramètres
- $2^n - 1$ déplacements : 1, 3, 7, ...
- Solution itérative possible, mais bien moins évidente

Itératif vs récursif

```
long factorielle(int n) {
  long resultat = 1;
  for (int i = 2; i <= n; i++) {
    resultat *= i;
  }
  return resultat;
}
```

```
long fibonacci(int n) {
  long a = 0, b = 1;
  for (int i = 0; i < n; i++) {
    long suivant = a + b;
    a = b;
    b = suivant;
  }
  return a;
}
```

Récursif

+ Proche de la définition mathématique, souvent plus **court** et plus **lisible**
– Coût de chaque appel ; profondeur limitée par la taille de la pile

Itératif

+ Généralement plus **rapide**, aucune limite de profondeur
– Parfois beaucoup plus difficile à écrire (Hanoï !)

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Nombre variable d'arguments

- `printf` accepte **un nombre variable** d'arguments :

```
printf("bonjour\n");
printf("%d\n", n);
printf("%d + %d = %d\n", a, b, a + b);
```

- Fonction **variadique** : au moins un paramètre fixe, suivi de `...`

Prototypes

```
int printf(const char *format, ...);
int somme(int n, ...);
```

Le type `const char *` désigne une chaîne de caractères — on y reviendra.

Exemple : somme d'entiers

Fichier `variadic.c` :

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

int somme(int n, ...) {
  va_list args;
  va_start(args, n);
  int total = 0;
  for (int i = 0; i < n; i++) {
    total += va_arg(args, int);
  }
  va_end(args);
  return total;
}

int main(void) {
  printf("%d\n", somme(3, 10, 20, 30));
  printf("%d\n", somme(5, 1, 2, 3, 4, 5));
  printf("%d\n", somme(0));
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o
      variadic variadic.c
$ ./variadic
60
15
0
```

- `n` : nombre d'arguments supplémentaires
- `#include <stdarg.h>` obligatoire

<stdarg.h>

- `va_list args;` : variable qui parcourt les arguments supplémentaires
- `va_start(args, n);` : commence le parcours, juste après le dernier paramètre fixe `n`
- `va_arg(args, int)` : retourne l'argument suivant, en supposant qu'il est de type `int`
- `va_end(args);` : termine le parcours (obligatoire)

La fonction ne sait pas combien d'arguments elle reçoit

Il faut le lui indiquer, grâce aux paramètres fixes :

- un **compte** explicite : `somme(3, 10, 20, 30)`
- une **valeur sentinelle** qui marque la fin : `somme_positifs(10, 20, 30, -1)`
- une chaîne de **format** : `printf` compte les `%`

Pièges

- **Aucune vérification** du nombre ni du type des arguments supplémentaires, ni à la compilation, ni à l'exécution :

```
somme(3, 10, 20);      // 3e argument absent
somme(3, 1.5, 2, 3);   // 1.5 lu comme int
```

  Résultat **imprévisible**, sans aucun avertissement.

- **Promotions** : les arguments supplémentaires sont convertis **avant** l'appel
  - `char` et `short` deviennent `int`
  - `float` devient `double`

  → écrire `va_arg(args, double)`, jamais `va_arg(args, float)`

Bonne pratique

Réserver les fonctions variadiques aux cas où le nombre d'arguments varie **vraiment**. Sinon, préférer des paramètres fixes : le compilateur peut alors vérifier chaque appel.

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Exemple : tester une factorielle itérative

On veut tester `factorielle` **avant** de savoir comment l'écrire : une définition minimale suffit pour compiler.

```c
#include <stdio.h>

// ---------- Prototypes ----------

/**
 * Calcule la factorielle d'un entier
 * positif ou nul.
 *
 * @param n  L'entier
 * @return   n! = 1 * 2 * ... * n
 */
long factorielle(int n);

/**
 * Compare factorielle(n) a la valeur
 * attendue et affiche le verdict.
 *
 * @param n        L'entree du test
 * @param attendu  Le resultat attendu
 * @return         true si reussi
 */
bool tester(int n, long attendu);
```

```c
// ---------- Definitions ---------

long factorielle(int n) {
  return 0;  // a completer
}

bool tester(int n, long attendu) {
  long obtenu = factorielle(n);
  bool ok = obtenu == attendu;
  printf("%-5s : %d! = %ld (attendu %ld)\n",
         ok ? "OK" : "ECHEC", n, obtenu, attendu);
  return ok;
}

// ------ Programme principal -----

int main(void) {
  int reussis = 0;
  reussis += tester(0, 1);
  reussis += tester(1, 1);
  reussis += tester(2, 2);
  reussis += tester(5, 120);
  reussis += tester(10, 3628800);
  printf("%d/5 tests reussis\n", reussis);
  return reussis == 5 ? 0 : 1;
}
```

Exécution des tests

```
$ gcc -Wall -Wextra -std=c23 -o testfactorial testfactorial.c
testfactorial.c: In function 'factorielle':
testfactorial.c:26:22: warning: unused parameter 'n'
$ ./testfactorial
ECHEC : 0! = 0 (attendu 1)
ECHEC : 1! = 0 (attendu 1)
ECHEC : 2! = 0 (attendu 2)
ECHEC : 5! = 0 (attendu 120)
ECHEC : 10! = 0 (attendu 3628800)
0/5 tests reussis
$ echo $?
1
```

- Tous les tests **échouent** : normal, `factorielle` n'est pas encore écrite
- L'avertissement est attendu : `n` n'est pas encore utilisé
- Chaque appel à `tester` : une entrée et le résultat **attendu**
- `tester` retourne un `bool` : dans `reussis += ...`, `true` compte pour 1 et `false` pour 0
- Le programme affiche un **bilan** et retourne 1 en cas d'échec

Une première tentative

On remplace la définition minimale :

```
long factorielle(int n) {
  long resultat = 1;
  for (int i = 2; i < n; i++) {
    resultat *= i;
  }
  return resultat;
}
```

On relance le **même** programme de test :

```
$ gcc -Wall -Wextra -std=c23 -o testfactorial testfactorial.c
$ ./testfactorial
OK    : 0! = 1 (attendu 1)
OK    : 1! = 1 (attendu 1)
ECHEC : 2! = 1 (attendu 2)
ECHEC : 5! = 24 (attendu 120)
ECHEC : 10! = 362880 (attendu 3628800)
2/5 tests reussis
$ echo $?
1
```

Les cas 0 et 1 **passent**... mais pas les suivants.

Analyse

- `factorielle(5)` donne $24 = 4!$: il manque le **dernier** facteur
- Coupable : la condition de la boucle

```
for (int i = 2; i < n; i++)     // incorrect
for (int i = 2; i <= n; i++)    // correct
```

Constat

Avec seulement les tests 0 et 1, l'erreur passait **inaperçue** : il faut des cas **normaux** et des cas **limites** (ici 2, première valeur où la boucle s'exécute).

Après correction

```
long factorielle(int n) {
  long resultat = 1;
  for (int i = 2; i <= n; i++) {
    resultat *= i;
  }
  return resultat;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o testfactorial testfactorial.c
$ ./testfactorial
OK    : 0! = 1 (attendu 1)
OK    : 1! = 1 (attendu 1)
OK    : 2! = 2 (attendu 2)
OK    : 5! = 120 (attendu 120)
OK    : 10! = 3628800 (attendu 3628800)
5/5 tests reussis
$ echo $?
0
```

Les **mêmes** tests réussissent tous : code de sortie 0.

Principe

- **Test unitaire** : vérifie **une seule** fonction, isolée du reste du programme
- Pour chaque cas : appeler la fonction avec une entrée connue, puis **comparer** le résultat obtenu au résultat attendu
- Choisir les cas comme vu précédemment : cas **normaux**, **limites** et **d'erreur**

Pourquoi des fonctions qui retournent une valeur ?

Une fonction qui **retourne** son résultat se teste facilement ; une fonction qui ne fait qu'**afficher** doit être vérifiée à l'œil.

Automatiser

- Relancer **tous** les tests après **chaque** modification : une correction peut en briser une autre
- Code de sortie non nul en cas d'échec : exploitable par un script ou un `Makefile`
- On y reviendra : code de test dans un fichier séparé, vérifications avec `assert`

Plan

1. Paramètres et valeur de retour
2. Prototypes
3. Passage par valeur
4. Documentation
5. Récursion
6. Fonctions variadiques
7. Tests unitaires
8. Débogage avec gdb

Exemple : pas à pas dans `factorielle`

```
$ gcc -Wall -Wextra -g -std=c23 -o factorial factorial.c
$ gdb -q factorial
Reading symbols from factorial...
(gdb) break main
Breakpoint 1 at 0x118b: file factorial.c, line 9.
(gdb) run
Breakpoint 1, main () at factorial.c:9
9         printf("3! = %ld\n", factorielle(3));
(gdb) next
3! = 6
10        printf("10! = %ld\n", factorielle(10));
(gdb) step
factorielle (n=10) at factorial.c:4
4         if (n == 0) return 1;           // cas de base
(gdb) print n
$1 = 10
(gdb) next
5         return n * factorielle(n - 1);  // cas recursif
(gdb) step
factorielle (n=9) at factorial.c:4
4         if (n == 0) return 1;           // cas de base
(gdb) finish
Run till exit from #0  factorielle (n=9) at factorial.c:4
0x0000555555555179 in factorielle (n=10) at factorial.c:5
5         return n * factorielle(n - 1);  // cas recursif
Value returned is $2 = 362880
```

Lancer gdb et points d'arrêt

- Compiler avec `-g` : ajoute les **informations de débogage** (noms, numéros de ligne)
- Lancer : `gdb -q programme`
- **Point d'arrêt** : endroit où l'exécution s'interrompt, pour examiner l'état du programme

Commandes

`break factorielle`	arrêt à l'entrée de la fonction
`break 10`	arrêt à la ligne 10
`break factorielle if n == 0`	arrêt seulement si la condition est vraie
`run`	lance le programme
`continue`	reprend jusqu'au prochain point d'arrêt
`delete`	supprime les points d'arrêt
`quit`	quitte gdb

Chaque commande a une abréviation : `b`, `r`, `c`, `q`, ...

Exécution pas à pas

Commandes

`next` (`n`)	exécute la ligne courante, **sans entrer** dans les appels
`step` (`s`)	exécute la ligne courante, en **entrant** dans les appels
`finish`	termine la fonction courante et affiche sa valeur de retour
`print n` (`p n`)	affiche la valeur d'une variable ou d'une expression
`info locals`	affiche toutes les variables locales

- `next` : on fait **confiance** à la fonction appelée
- `step` : on la **soupçonne**, et on veut voir ce qu'elle fait
- La ligne affichée est la **prochaine** à exécuter, pas celle qui vient de l'être

Pile d'appels : backtrace

```
$ gdb -q factorial
Reading symbols from factorial...
(gdb) break factorielle if n == 0
Breakpoint 1 at 0x1159: file factorial.c, line 4.
(gdb) run
Breakpoint 1, factorielle (n=0) at factorial.c:4
4         if (n == 0) return 1;           // cas de base
(gdb) backtrace
#0  factorielle (n=0) at factorial.c:4
#1  0x0000555555555179 in factorielle (n=1) at factorial.c:5
#2  0x0000555555555179 in factorielle (n=2) at factorial.c:5
#3  0x0000555555555179 in factorielle (n=3) at factorial.c:5
#4  0x0000555555555195 in main () at factorial.c:9
(gdb) frame 2
#2  0x0000555555555179 in factorielle (n=2) at factorial.c:5
5         return n * factorielle(n - 1);  // cas recursif
(gdb) print n
$1 = 2
```

- `backtrace` (`bt`) : liste les cadres de la pile, du plus récent (#0) jusqu'à `main`
- `frame 2` : se place dans le cadre #2 pour y examiner les variables (`up`/`down` pour se déplacer d'un cadre)
- Indispensable pour savoir **comment** on est arrivé à un plantage