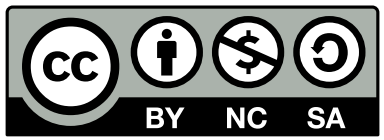
INF1135
Programmation procédurale
Chapitre 2 — Bases du C

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Types entiers

Type	Taille typique [1]	Plage
`char`	1 octet	-128 à 127 ou 0 à 255
`short`	2 octets	$-32\,768$ à $32\,767$
`int`	4 octets	$\approx \pm 2{,}1 \times 10^9$
`long`	8 octets	$\approx \pm 9{,}2 \times 10^{18}$
`long long`	8 octets	idem `long`

- `unsigned` : élimine les valeurs négatives, double la plage positive
- `signed` : par défaut, sauf pour `char` seul, signé ou non selon la plateforme (signé sur x86-64, non signé sur ARM)
- Tailles **dépendantes de la plateforme**

1. Valeurs typiques sous Linux 64 bits.

Exemple : afficher des entiers

Fichier `integers.c` :

```c
#include <stdio.h>

int main(void) {
  int temperature = -12;
  unsigned int compteur = 4000000000;
  short altitude = -300;
  unsigned short frequence = 65000;
  long population = 8100000000;
  long long distance = 900000000000000000;

  printf("%d\n", temperature);
  printf("%u\n", compteur);
  printf("%hd, %hu\n", altitude, frequence);
  printf("%ld\n", population);
  printf("%lld\n", distance);
  printf("%d%%\n", 75);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=
    c23 -o integers
    integers.c
$ ./integers
-12
4000000000
-300, 65000
8100000000
900000000000000000
75%
```

Afficher des entiers avec `printf`

`printf(format, valeur1, valeur2, ...)` : chaque **spécificateur** (`%...`) du format est remplacé par une valeur.

Type	Signé	Non signé
`int`	`%d` ou `%i`	`%u`
`short`	`%hd`	`%hu`
`long`	`%ld`	`%lu`
`long long`	`%lld`	`%llu`

- Modificateurs de longueur : `h` (`short`), `l` (`long`), `ll` (`long long`)
- `%%` : affiche le caractère `%`
- Spécificateur qui ne correspond pas au type : comportement **indéfini**, signalé par `-Wall`

Types entiers de taille fixe (<stdint.h>)

Type	Taille	Plage
`int8_t` / `uint8_t`	1 octet	-128 à 127 / 0 à 255
`int16_t` / `uint16_t`	2 octets	$-32\,768$ à $32\,767$ / 0 à $65\,535$
`int32_t` / `uint32_t`	4 octets	$\approx \pm 2{,}1 \times 10^9$ / 0 à $\approx 4{,}3 \times 10^9$
`int64_t` / `uint64_t`	8 octets	$\approx \pm 9{,}2 \times 10^{18}$ / 0 à $\approx 1{,}8 \times 10^{19}$

- `int`, `long`, etc. : taille **variable** selon la plateforme
- Registres matériels : taille fixe requise — sinon corruption des bits voisins
- Protocoles : trames en octets, tailles fixes — sinon mauvaise interprétation
- Fichiers binaires : structure identique en lecture/écriture
- Microcontrôleurs : registres souvent `uint8_t`, `uint16_t`, `uint32_t`

Exemple : afficher un entier de taille fixe

Fichier `inttypes.c` :

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

int main(void) {
  int32_t x = 42;
  uint8_t y = 200;
  printf("x = %" PRId32 ", y = %" PRIu8 "\n", x, y);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o inttypes inttypes.c
$ ./inttypes
x = 42, y = 200
```

- Leur format `printf` dépend de la plateforme : `<inttypes.h>` fournit des macros, `PRId32`, `PRIu8`, etc.
- `"x = %" PRId32 ", y = %"` : chaînes littérales **concaténées** par le compilateur

Type caractère

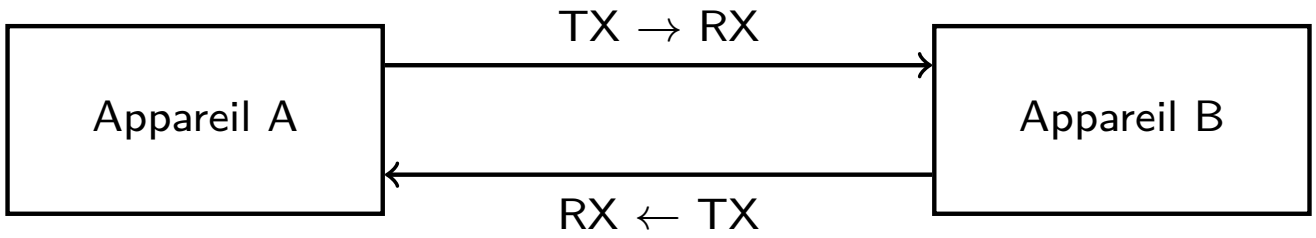
```c
#include <stdio.h>

int main(void) {
  char c = 'A';
  printf("c = '%c' (code %d)\n", c, c);
  c = c + 1;
  printf("c = '%c' (code %d)\n", c, c);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o chars chars.c
$ ./chars
c = 'A' (code 65)
c = 'B' (code 66)
```

- char : **1 octet** — en réalité un **petit entier**
- Encodage **ASCII** : `'A'` vaut 65, `'a'` vaut 97, `'0'` vaut 48
- Littéral `'x'` : constante de type `int`
- char : `signed` ou `unsigned` selon la plateforme

Communication série UART



- **UART** = *Universal Asynchronous Receiver/Transmitter*
- Communication série, un **octet** à la fois
- Deux fils : **TX** (transmission) et **RX** (réception)
- **Asynchrone** : pas d'horloge partagée
- *Baud rate* (ex. 9600, 115200) : nombre de bits par seconde — débit convenu à l'avance
- Chaque octet est encadré d'un **bit de start** et d'un (ou plusieurs) **bit(s) de stop**
- Chaque octet transmis = une valeur `char`

Représentation binaire des entiers

- Tout entier en mémoire : suite de **bits** $(0/1)$
- n bits $\rightarrow 2^n$ valeurs possibles
- `unsigned` : binaire direct, de 0 à $2^n - 1$

`signed` : complément à deux

- Bit de poids fort : **signe** (0 positif, 1 négatif)
- Négation : inverser les bits, puis $+1$
- Plage : de -2^{n-1} à $2^{n-1} - 1$

Valeur	**Binaire (8 bits)**	**Hexadécimal**
5	`0000 0101`	`0x05`
-5	`1111 1011`	`0xFB`
-1	`1111 1111`	`0xFF`
-128	`1000 0000`	`0x80`

Exemple : décimal, hexadécimal, binaire

Fichier `binary.c` :

```c
#include <stdio.h>

int main(void) {
  signed char values[] = {5, -5, -1, -128};
  for (int i = 0; i < 4; i++) {
    signed char v = values[i];
    printf("%4d  0x%02hhX  %08hhb\n", v, v, v);
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o binary
    binary.c
$ ./binary
   5  0x05  00000101
  -5  0xFB  11111011
  -1  0xFF  11111111
-128  0x80  10000000
```

- `%x`/`%X` : hexadécimal (minuscules/majuscules), `%o` : octal, `%b` : binaire (C23) ; la valeur est lue comme **non signée**
- `%#x`, `%#o` : ajoutent le préfixe de la base (`0xff`, `0377`)
- Mêmes modificateurs de longueur que pour `%d` : `%lx`, `%llb`, … ; ici, `hh` : tronque à 8 bits avant conversion

Types à virgule flottante

Type	Taille typique	Précision
`float`	4 octets	≈ 7 chiffres significatifs
`double`	8 octets	≈ 15–16 chiffres significatifs
`long double`	souvent 16 octets	précision étendue

Représentation binaire

Représentation en **binaire** (norme IEEE 754) — certaines valeurs décimales (ex. `0.1`) non représentables exactement.

Dans ce cours

`double` par défaut pour les calculs flottants.

Norme IEEE 754

- Nombre $=$ signe $\times$ mantisse $\times 2^{\text{exposant}}$
- 3 champs : **signe** (1 bit), **exposant**, **mantisse**

Type	**Signe**	**Exposant**	**Mantisse**
`float` (32 bits)	1 bit	8 bits	23 bits
`double` (64 bits)	1 bit	11 bits	52 bits

Conséquences

- Précision **finie** : arrondi à chaque opération
- Valeurs spéciales : +0/−0, $\pm\infty$, `NaN`
- Comparaison d'égalité (==) risquée sur les flottants

Conversion vers `float` : exemple détaillé

Convertir $13{,}375$ en `float` :

- Partie entière : $13 = (1101)_2$
- Partie fractionnaire $0{,}375$: multiplier par 2, retirer la partie entière à chaque étape

Calcul	Bit	Reste
$0{,}375 \times 2 = 0{,}75$	0	$0{,}75$
$0{,}75 \times 2 = 1{,}5$	1	$0{,}5$
$0{,}5 \times 2 = 1{,}0$	1	0

- Donc $0{,}375 = (0{,}011)_2$, et $13{,}375 = (1101{,}011)_2 = (1{,}101011)_2 \times 2^3$
- Exposant **stocké** = exposant réel + **biais** (127 pour `float`)
- Ici : $3 + 127 = 130 = (1000\,0010)_2$
- Mantisse : bits après le « 1, », complétés de 0 jusqu'à 23 bits

Valeur	Signe	Exposant	Mantisse
$13{,}375$	0	1000 0010	1010 1100 0000 0000 0000 000

Exemple : représentation `float`

Valeur	Signe	Exposant	Mantisse
$6{,}5$	0	1000 0001	1010 0000 0000 0000 0000 000
$+0$	0	0000 0000	0000 0000 0000 0000 0000 000
-0	1	0000 0000	0000 0000 0000 0000 0000 000
$+\infty$	0	1111 1111	0000 0000 0000 0000 0000 000
$-\infty$	1	1111 1111	0000 0000 0000 0000 0000 000
`NaN`	–	1111 1111	1000 0000 0000 0000 0000 000

- $6{,}5 = (110{,}1)_2 = (1{,}101)_2 \times 2^2$
- Ici : $2 + 127 = 129 = (1000\,0001)_2$
- Exposant 0 + mantisse 0 : **zéro** (signé)
- Exposant max + mantisse 0 : ∞ (signé)
- Exposant max + mantisse $\neq 0$: `NaN` (signe **quelconque**)

Exemple : afficher 0,1

Fichier `float01.c` :

```c
#include <stdio.h>

int main(void) {
  float x = 0.1f;
  printf("%.1f\n", x);
  printf("%.10f\n", x);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o float01 float01.c
$ ./float01
0.1
0.1000000015
```

- Avec `%.1f` : arrondi, le défaut **passe inaperçu**
- Avec `%.10f` : la valeur **réellement stockée** apparaît
- $0{,}1000000015 \neq 0{,}1$, mais c'est la valeur `float` la plus proche

Exemple : `0,1` non représentable

Valeur	**Signe**	**Exposant**	**Mantisse**
$\approx 9{,}9999994 \times 10^{-2}$	0	0111 1011	1001 1001 1001 1001 1001 100
$0{,}1$ (réel)		*non représentable*	
$\approx 1{,}00000001 \times 10^{-1}$	0	0111 1011	1001 1001 1001 1001 1001 101

- $0{,}1 = (1{,}1001100110011\ldots)_2 \times 2^{-4}$: périodique infini
- Les deux mantisses ne diffèrent que par le **dernier bit**
- `float(0.1)` : la valeur représentable la **plus proche** de $0{,}1$
- Écart entre les deux : $2^{-4-23} = 2^{-27} \approx 7{,}45 \times 10^{-9}$

Exemple : type booléen

Fichier `bool.c` :

```c
#include <stdio.h>

int main(void) {
  bool est_led_allume = false;
  bool est_actif = true;
  printf("est_led_allume = %d\n", est_led_allume);
  printf("est_actif      = %d\n", est_actif);

  est_led_allume = 42;  // toute valeur non nulle devient true
  printf("est_led_allume = %d\n", est_led_allume);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o bool bool.c
$ ./bool
est_led_allume = 0
est_actif      = 1
est_led_allume = 1
```

Type booléen

- `bool` : une **valeur de vérité**, `true` (vrai) ou `false` (faux)
- `bool`, `true` et `false` sont des **mots-clés** depuis C23
- Occupe en général **1 octet**
- Pas de format dédié pour `printf` : `%d` affiche `0` (faux) ou `1` (vrai)
- Toute valeur **non nulle** affectée à un `bool` devient `true` ; `0` devient `false`

Avant C23

Il fallait `#include <stdbool.h>`. Beaucoup de code existant utilise encore un `int` valant `0` ou `1` : on le rencontrera souvent.

Bon usage

Un nom qui commence par un verbe à l'indicatif et se lit comme une affirmation : `est_valide`, `est_led_allume`, `a_termine`.

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Constantes littérales : entiers

Fichier `litint.c` :

```c
#include <stdio.h>

int main(void) {
  int dec = 42, oct = 052, hex = 0x2A, bin = 0b101010;
  unsigned long big = 42UL;
  printf("%d %d %d %d %lu\n", dec, oct, hex, bin, big);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o litint litint.c
$ ./litint
42 42 42 42 42
```

- Décimal : aucun préfixe (premier chiffre $\neq 0$)
- Octal : préfixe `0` ; hexadécimal : préfixe `0x`/`0X` ; binaire : préfixe `0b`/`0B` (C23)
- Suffixes combinables : `u`/`U` (`unsigned`), `l`/`L` (`long`), `ll`/`LL` (`long long`)

Constantes littérales : flottants

Fichier `litflt.c` :

```
#include <stdio.h>

int main(void) {
  double pi = 3.14, avogadro = 6.02e23;
  float ratio = 1.0f;
  printf("%.2f %.2e %.1f\n", pi, avogadro, ratio);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o litflt litflt.c
$ ./litflt
3.14 6.02e+23 1.0
```

- Flottant : notation décimale (`3.14`) ou scientifique (`6.02e23`)
- Suffixe `f`/`F` : `float` ; `l`/`L` : `long double` ; sinon `double`

Constantes littérales : caractères et chaînes

Fichier `litchar.c` :

```c
#include <stdio.h>

int main(void) {
  char letter = 'A', newline = '\n';
  char greeting[] = "Hello";
  printf("%c %s", letter, greeting);
  putchar(newline);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o litchar litchar.c
$ ./litchar
A Hello
```

- Caractère : entre apostrophes, échappement avec \
- Chaîne : entre guillemets
- `char greeting[]` : **tableau** de caractères qui contient la chaîne (on va y revenir)

Constantes symboliques

Fichier `constants.c` :

```
#include <stdio.h>

#define LED_PIN 13

int main(void) {
  const double v_ref = 3.3;
  printf("LED_PIN = %d\n", LED_PIN);
  printf("v_ref   = %.1f\n", v_ref);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o constants constants.c
$ ./constants
LED_PIN = 13
v_ref   = 3.3
```

- `#define` — substitution textuelle par le préprocesseur, sans type
- `const` — variable typée, vérifiée par le compilateur

Quelques constantes de la bibliothèque standard

Fichier `stdlimits.c` :

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main(void) {
  printf("int         : [%d, %d]\n", INT_MIN, INT_MAX); // <limits.h>
  printf("DBL_EPSILON : %.3e\n", DBL_EPSILON);          // <float.h>
  printf("EOF         : %d\n", EOF);                    // <stdio.h>
  return 0;
}
```

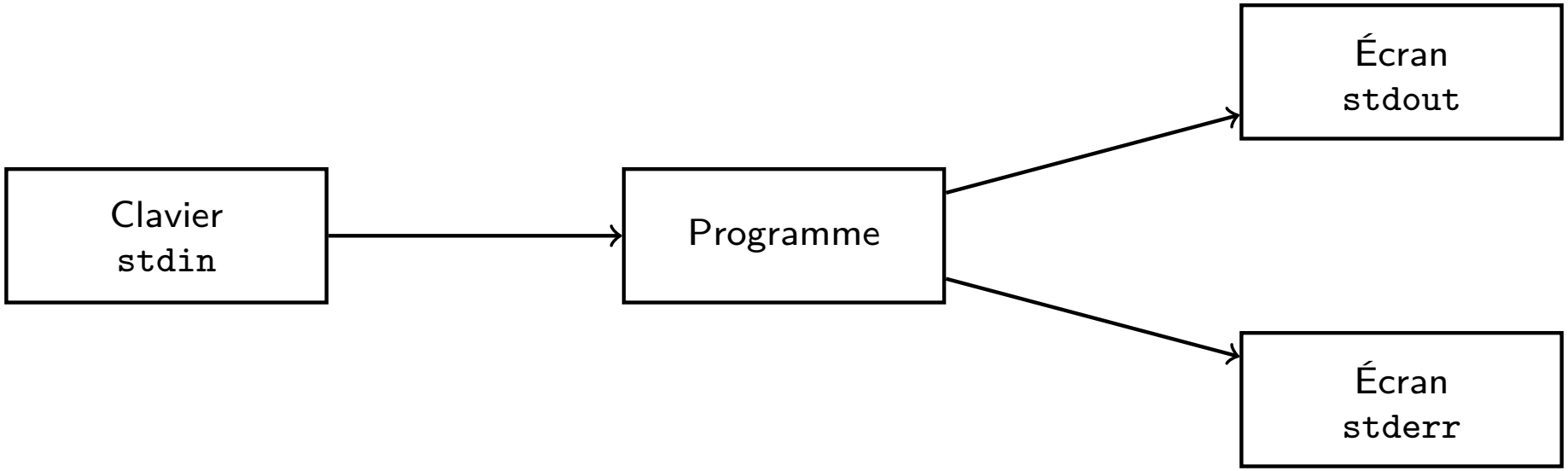
```
$ gcc -Wall -Wextra -std=c23 -o stdlimits stdlimits.c
$ ./stdlimits
int         : [-2147483648, 2147483647]
DBL_EPSILON : 2.220e-16
EOF         : -1
```

- Constantes définies dans les en-têtes de la bibliothèque standard
- Portables : valeurs garanties correctes, peu importe la plateforme

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Un programme et son environnement



- `stdin`, `stdout`, `stderr` : trois flux ouverts automatiquement
- `stdin` : entrée standard, lue par `scanf`, `getchar`, ...
- `stdout` : sortie standard, écrite par `printf`, `putchar`, ...
- `stderr` : sortie d'erreur, pour les messages d'erreur
- Redirection (shell) : `prog < entree.txt > sortie.txt`

Affichage avec `printf` : rappel

Spécificateurs vus précédemment :

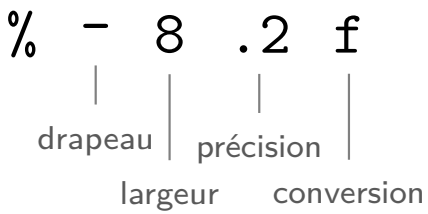
Spécificateur	**Affiche**
`%d` (ou `%i`), `%u`	un `int`, un `unsigned int`
`%hd`, `%ld`, `%lld`, ...	un `short`, un `long`, un `long long`, ...
`%x`, `%X`, `%o`, `%b`	un entier en base 16, 8 ou 2 (`#` : avec préfixe)
`%c`	un caractère (`char`)
`%f`, `%e`	un `double`, en notation décimale ou scientifique
`%s`	une chaîne de caractères
`%%`	le caractère %

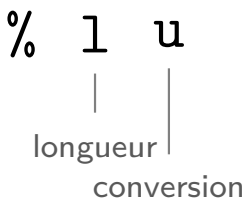
Anatomie d'un spécificateur

`%[drapeaux][largeur][.précision][longueur]conversion`

Seuls le % et la **conversion** sont obligatoires ; les autres parties, facultatives, respectent cet ordre.

Partie	**Rôle**	**Exemples**
Drapeaux	alignement, signe, zéros, préfixe	`-`, `+`, `0`, `#`
Largeur	nombre **minimal** de caractères	`8`, `10`
Précision	décimales, longueur max. d'une chaîne, …	`.2`, `.3`
Longueur	taille du type (`short`, `long`, …)	`h`, `l`, `ll`
Conversion	type de la valeur, façon de l'écrire	`d`, `u`, `x`, `f`, `s`, …





Largeur, précision et drapeaux

Largeur : `%8.2f`, `%10s`

- Nombre **minimal** de caractères affichés ; complété par des espaces à gauche
- Jamais de troncature : une valeur plus longue s'affiche en entier
- `*` : largeur fournie en argument, `printf("%*d", 6, n)`

Précision : `%.2f`, `%.3s`, `%.4d`

- `%f`, `%e` : nombre de décimales (6 par défaut), avec arrondi
- `%s` : nombre **maximal** de caractères (la chaîne est tronquée)
- Entiers : nombre minimal de chiffres, complété par des zéros

Drapeaux

`-` : aligné à gauche ; `0` : complété par des zéros ; `+` : signe toujours affiché ; `#` : préfixe de base

Exemple : largeur et précision

Fichier `printf.c` :

```c
#include <stdio.h>

int main(void) {
  int n = 42;
  double pi = 3.14159;
  char s[] = "bonjour";

  printf("[%5d][%-5d][%05d][%+d]\n", n, n, n, n);
  printf("[%f][%.2f][%8.2f][%-8.2f]\n", pi, pi, pi, pi);
  printf("[%10s][%-10s][%.3s]\n", s, s, s);
  printf("[%.4d][%*d]\n", n, 6, n);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o printf printf.c
$ ./printf
[   42][42   ][00042][+42]
[3.141590][3.14][    3.14][3.14    ]
[   bonjour][bonjour   ][bon]
[0042][    42]
```

Autres fonctions de sortie

Fonction	Rôle
`putchar(c)`	écrit un caractère sur `stdout`
`puts(s)`	écrit une chaîne, suivie d'un `\n`
`fprintf(stderr, ...)`	écrit sur la sortie d'erreur, comme `printf`
`snprintf(buf, n, ...)`	écrit dans une chaîne, comme `printf`

Exemple : lecture au clavier

Fichier `io.c` :

```c
#include <stdio.h>

int main(void) {
  int age;
  double height;
  printf("Age: ");
  scanf("%d", &age);
  printf("Height: ");
  scanf("%lf", &height);
  printf("Age=%d, height=%.2f\n", age, height);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o io io.c
$ ./io
Age: 25
Height: 1.8
Age=25, height=1.80
```

Lecture avec `scanf` (`stdin`)

- Mêmes spécificateurs de format que `printf`
- `&` obligatoire devant les variables (adresse où écrire)

→ Exception : les chaînes (`char[]`) — déjà des adresses

- Valeur de retour : nombre d'éléments lus avec succès

Pièges courants

- Oublier `&` → avertissement `-Wformat` à la compilation ; si ignoré, comportement indéfini (souvent un crash)
- `%s` sans limite de taille → dépassement de tampon
- Toujours **vérifier la valeur de retour** de `scanf`

Exemple : fgets

Fichier fgets.c :

```c
#include <stdio.h>

#define TAILLE 20

int main(void) {
  char name[TAILLE];
  printf("Nom: ");
  fgets(name, TAILLE, stdin);
  printf("Bonjour, %s", name);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o fgets fgets.c
$ ./fgets
Nom: Alice
Bonjour, Alice
```

Autres fonctions d'entrée

Fonction	Rôle
`getchar()`	lit un caractère depuis `stdin`
`fgets(buf, n, stdin)`	lit une ligne (au plus `n-1` caractères)

- `fgets` conserve le `\n` final, s'il y a la place
- Contrairement à `scanf("%s")`, `fgets` ne déborde jamais : la taille est fournie

À éviter

`gets()` : **retiré** du standard (C11) — dépassement de tampon garanti. Utiliser `fgets()` à la place.

Exemple : la sortie d'erreur

Fichier `messages.c` :

```c
#include <stdio.h>

int main(void) {
  printf("result: 42\n");
  fprintf(stderr, "warning: estimated value\n");
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o messages messages.c
$ ./messages
result: 42
warning: estimated value
$ ./messages > out.txt
warning: estimated value
$ cat out.txt
result: 42
$ ./messages 2> err.txt
result: 42
$ cat err.txt
warning: estimated value
```

La sortie d'erreur `stderr`

- `fprintf(stderr, ...)` : comme `printf`, mais écrit sur `stderr`
- Au terminal, `stdout` et `stderr` s'affichent au même endroit, mais restent deux flux **distincts**
- `>` redirige seulement `stdout` : les messages d'erreur restent visibles
- `2>` redirige `stderr`
- `stderr` n'est pas mise en mémoire tampon : un message est affiché immédiatement, même si le programme plante juste après

Bon usage

Résultats sur `stdout` ; erreurs, avertissements et diagnostics sur `stderr`.

Exemple : code de retour

Fichier `exitcode.c` :

```c
#include <stdio.h>

int main(void) {
  int n;
  scanf("%d", &n);
  if (n < 0) {
    fprintf(stderr, "error: negative number\n");
    return 1;
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o exitcode exitcode.c
$ ./exitcode
5
$ echo $?
0
$ ./exitcode
-3
error: negative number
$ echo $?
1
```

Code de retour

- Valeur retournée par `main` : le **code de retour** du programme

→ 0 : succès ; toute autre valeur : erreur

- Consultable dans le terminal avec `echo $?`
- Exploité par les scripts et les outils, qui s'arrêtent si une commande échoue
- En cas d'erreur, on écrit un message sur `stderr` **et** on retourne un code non nul

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Opérateurs arithmétiques

- `+`, `-`, `*`, `/`, `%` (modulo)
- `%` s'applique uniquement aux **entiers**

Division entière

`7 / 2` = 3 (troncature vers zéro), pas `3.5` !

Résultat flottant : convertir au moins un opérande (`(double) 7 / 2`).

Modulo et nombres négatifs

Signe de `%` : suit le **dividende** — `-7 % 2` = `-1`.

En pratique

Signal périodique (PWM, clignotement DEL) : `compteur % periode`.

Exemple : seuil d'alarme

Fichier `alarm.c` :

```c
#include <stdio.h>

#define TEMP_MAX  80

int main(void) {
  int temperature = 85;
  int alarm = (temperature > TEMP_MAX) || (temperature < 0);
  printf("temperature=%d, alarm=%d\n", temperature, alarm);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o alarm alarm.c
$ ./alarm
temperature=85, alarm=1
```

- Capteur hors plage $\rightarrow$ `||` déclenche l'alarme
- Résultat `0`/`1` : directement utilisable comme drapeau

Opérateurs relationnels et logiques

Relationnels (résultat : 0 ou 1)

`<`, `>`, `<=`, `>=`, `==`, `!=`

Logiques

`&&` (et), `||` (ou), `!` (non)

- `0` : **faux** — tout le reste : **vrai**
- Le résultat se range dans un `bool` : `bool est_majeur = age >= 18;`
- **Évaluation en court-circuit** : `&&` et `||` arrêtent dès que le résultat est connu

Piège fréquent

`if (x = 5)` (affectation) au lieu de `if (x == 5)` (comparaison) — compile sans erreur !

Opérateurs bit à bit

Fichier `operators.c` :

```c
#include <stdio.h>

int main(void) {
  unsigned int a = 12;  // 1100
  unsigned int b = 10;  // 1010
  printf("a & b  = %u\n", a & b);
  printf("a | b  = %u\n", a | b);
  printf("a ^ b  = %u\n", a ^ b);
  printf("a << 1 = %u\n", a << 1);
  printf("a >> 1 = %u\n", a >> 1);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o operators operators.c
$ ./operators
a & b  = 8
a | b  = 14
a ^ b  = 6
a << 1 = 24
a >> 1 = 6
```

- &, |, ^, ~, <<, >> — utiles pour les **masques**

Exemple : registre GPIO

Registre **GPIO** [2] — fichier `gpio.c` :

```c
#include <stdio.h>

#define LED_PIN  3
#define LED_MASK (1 << LED_PIN)

int main(void) {
  unsigned char port = 0x00;
  port |= LED_MASK;
  printf("set:    0x%02X\n", port);
  port &= ~LED_MASK;
  printf("clear:  0x%02X\n", port);
  port ^= LED_MASK;
  printf("toggle: 0x%02X\n", port);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o gpio gpio.c
$ ./gpio
set:    0x08
clear:  0x00
toggle: 0x08
```

- Activer/désactiver/inverser : `|=`, `&= ~`, `^=`

2. *General-Purpose Input/Output* — broche du microcontrôleur configurable en entrée ou en sortie numérique.

Opérateurs d'affectation composés

```
a += b   équivaut à a = a + b
a -= b   équivaut à a = a - b
a *= b   équivaut à a = a * b
a /= b   équivaut à a = a / b
a %= b   équivaut à a = a % b
```

Incrémentation et décrémentation

- `a++` / `++a` équivalent à `a += 1`
- `a--` / `--a` équivalent à `a -= 1`

→ Préfixe vs postfixe : diffèrent dans la **valeur de l'expression**, pas dans l'effet sur `a`

À éviter

Combiner plusieurs effets de bord sur la même variable dans une expression (ex. `a = a++ + ++a;`) — comportement indéfini.

Exemple : l'opérateur `sizeof`

Fichier `types.c` :

```c
#include <stdio.h>

int main(void) {
  double tension = 3.3;
  printf("char   : %zu octet(s)\n", sizeof(char));
  printf("bool   : %zu octet(s)\n", sizeof(bool));
  printf("int    : %zu octet(s)\n", sizeof(int));
  printf("long   : %zu octet(s)\n", sizeof(long));
  printf("double : %zu octet(s)\n", sizeof(double));
  printf("tension: %zu octet(s)\n", sizeof tension);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o types
      types.c
$ ./types
char   : 1 octet(s)
bool   : 1 octet(s)
int    : 4 octet(s)
long   : 8 octet(s)
double : 8 octet(s)
tension: 8 octet(s)
```

- On retrouve les tailles typiques des types vus précédemment
- `sizeof` s'applique à un **type** (entre parenthèses) ou à une **variable**

L'opérateur `sizeof`

- `sizeof(type)` ou `sizeof variable` : taille en **octets**
- C'est un **opérateur**, et non une fonction : la taille est connue dès la **compilation**
- Résultat de type `size_t` (entier non signé), affiché avec %zu
- Utile pour éviter d'écrire des tailles « en dur » :

```
char nom[TAILLE];
fgets(nom, sizeof nom, stdin);  // suit TAILLE
```

Portabilité

Les tailles dépendent de la plateforme (sauf `sizeof(char)`, toujours 1) : ne jamais les supposer, les demander à `sizeof`.

Priorité et associativité

Priorité	Opérateurs	Associativité
Plus élevée	`()`, `[]`	gauche à droite
	`!`, `~`, `++`, `--`, `(type)`, `sizeof`	droite à gauche
	`*`, `/`, `%`	gauche à droite
	`+`, `-`	gauche à droite
	`<<`, `>>`	gauche à droite
	`<`, `<=`, `>`, `>=`	gauche à droite
	`==`, `!=`	gauche à droite
	`&`, `^`, `|`	gauche à droite
	`&&`, `||`	gauche à droite
	`=`, `+=`, `-=`, ...	droite à gauche
Moins élevée	`,`	gauche à droite

Piège classique

`flags & MASK == 1` → `==` avant `&` ! Écrire `(flags & MASK) == 1`.

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Conversions explicites (*cast*)

Syntaxe : `(type) expression`

Fichier `cast.c` :

```c
#include <stdio.h>

int main(void) {
  int total = 7;
  int n = 2;
  printf("total / n           = %d\n", total / n);
  printf("(double) total / n = %.2f\n", (double) total / n);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o cast cast.c
$ ./cast
total / n           = 3
(double) total / n = 3.50
```

- Le *cast* force la conversion **avant** le calcul
- Utile pour forcer une division flottante, ou tronquer volontairement

Conversions implicites

Promotion entière

Types plus petits que `int` (`char`, `short`) : **promus en `int`** avant calcul.

Conversions arithmétiques usuelles

Expression mixte : opérande de type **le plus large** impose son type à l'autre :

- `int + double` → `double`
- `int + long` → `long`
- `int + unsigned int` → `unsigned int`

Piège : signé/non signé

`int + unsigned int` convertit l'`int` en `unsigned` — une valeur négative devient un grand nombre positif ! Ex. : un ADC retourne `unsigned int`.

Exemple : capteur et ADC




```c
#include <stdio.h>
#define ADC_MAX  1023
#define VREF     3.3

int main(void) {
  int raw = 614;
  double voltage = (double) raw / ADC_MAX * VREF;
  printf("raw=%d, voltage=%.2fV\n", raw, voltage);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o adc adc.c
$ ./adc
raw=614, voltage=1.98V
```

Débogage : pièges de conversion

- Conversion inattendue : résultat souvent **plausible mais faux** — sans plantage ni avertissement
- Réflexe : **afficher la valeur** des variables suspectes avec `printf`

Exemple

```
printf("debug: total=%d, n=%d, total/n=%d\n",
       total, n, total / n);
```

Stratégie

1. Identifier l'expression suspecte
2. Afficher sa valeur (et celle de ses opérandes)
3. Ajouter les *casts* nécessaires

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Pourquoi `make` ?

À chaque modification, il faut retaper :

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

- Commande **longue** : facile d'oublier une option
- Un laboratoire ou un TP contient souvent **plusieurs** programmes
- Recompiler ce qui n'a pas changé : une perte de temps

`make`

- Lit un fichier nommé `Makefile`, qui décrit **comment** produire chaque fichier
- Exécute seulement les commandes **nécessaires**
- Une seule commande à retenir : `make`

Exemple : un premier Makefile

Fichier Makefile :

```
all: hello

hello: hello.c
  gcc -Wall -Wextra -std=c23 -o hello hello.c

clean:
  rm -f hello

.PHONY: all clean
```

```
$ make
gcc -Wall -Wextra -std=
    c23 -o hello hello.c
$ ./hello
Hello, world!
$ make
make: Nothing to be done
    for 'all'.
```

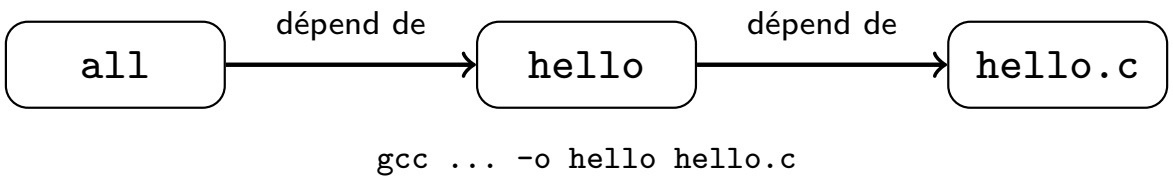
- make affiche chaque commande avant de l'exécuter
- Deuxième make : **rien à faire**, hello est à jour

Anatomie d'une règle

Syntaxe

```
cible: dépendances
  [TAB] commande(s)
```

- **Cible** : le fichier à produire (`hello`)
- **Dépendances** : les fichiers nécessaires pour le produire (`hello.c`)
- **Commandes** : comment le produire ; chaque ligne commence par une **tabulation**



Erreur fréquente

Des espaces au lieu d'une tabulation devant une commande. Configurer l'éditeur pour conserver les tabulations dans les `Makefile`.

Exemple : modifier un fichier source

```
$ touch hello.c
$ make
gcc -Wall -Wextra -std=c23 -o hello
    hello.c
$ make
make: Nothing to be done for 'all'.
$ make clean
rm -f hello
$ make
gcc -Wall -Wextra -std=c23 -o hello
    hello.c
```

- `touch hello.c` : met à jour la date de modification, comme si on avait modifié et enregistré le fichier
- `hello.c` est maintenant **plus récent** que `hello` : `make` recompile
- `make clean` : supprime l'exécutable
- `hello` n'existe plus : `make` recompile

Quand `make` exécute-t-il une règle ?

- `make cible` : traite la règle de `cible` ; `make` seul traite la **première** règle du fichier (d'où `all` en tête)
- Avant tout, `make` traite les règles des **dépendances**, de proche en proche
- Les commandes sont exécutées seulement si :
  - la cible **n'existe pas**, ou
  - une dépendance est **plus récente** que la cible (date de modification)

Messages qui ne sont pas des erreurs

`Nothing to be done for 'all'` et `'hello' is up to date` : tout est déjà à jour.

Attention

Une dépendance oubliée dans la règle : `make` ne voit pas sa modification et ne recompile pas.

Cibles fictives : .PHONY

```
all: hello

hello: hello.c
  gcc -Wall -Wextra -std=c23 -o hello hello.c

clean:
  rm -f hello

.PHONY: all clean
```

- `all` et `clean` ne sont pas des fichiers à produire, mais des **noms d'actions**
- `.PHONY` : déclare ces cibles **fictives** ; leurs commandes sont toujours exécutées, même si un fichier du même nom existe
- Conventions :
  - `all` : première règle, produit tous les programmes
  - `clean` : supprime les fichiers **générés**, jamais les sources

Exemple : plusieurs programmes

Fichier `Makefile` :

```
all: hello salut

hello: hello.c
  gcc -Wall -Wextra -std=c23 -o hello hello.c

salut: salut.c
  gcc -Wall -Wextra -std=c23 -o salut salut.c

clean:
  rm -f hello salut

.PHONY: all clean
```

```
$ make
gcc -Wall -Wextra -std=
    c23 -o hello hello.c
gcc -Wall -Wextra -std=
    c23 -o salut salut.c
$ touch salut.c
$ make
gcc -Wall -Wextra -std=
    c23 -o salut salut.c
$ make -n clean
rm -f hello salut
```

- `all` dépend des **deux** programmes
- Après `touch salut.c`, seul `salut` est recompilé
- `make -n` : affiche les commandes **sans** les exécuter

Erreurs fréquentes

Espaces au lieu d'une tabulation

```
$ make
Makefile:4: *** missing separator.  Stop.
```

Dépendance introuvable (ici, `helo.c` au lieu de `hello.c`)

```
$ make
make: *** No rule to make target 'helo.c', needed by 'hello'.  Stop.
```

- Le numéro de ligne (`Makefile:4`) indique où chercher
- Le fichier doit s'appeler exactement `Makefile`, sinon : `No targets specified and no makefile found`

Résumé

Commande	Effet
`make`	traite la première règle (`all`)
`make hello`	traite la règle de `hello`
`make clean`	supprime les fichiers générés
`make -n`	affiche les commandes sans les exécuter

Un `Makefile` typique

1. `all`, qui dépend de tous les programmes
2. Une règle par programme : `prog: prog.c`
3. `clean`
4. `.PHONY: all clean`

Plan

1. Types de base
2. Constantes
3. Flux standards et code de retour
4. Opérateurs
5. Conversions de types
6. Automatiser la compilation : `make`
7. Bonnes pratiques

Conventions de style

Dans le cours :

- **Indentation** : 2 espaces — cohérence avant tout
- **Variables** : `snake_case` (`num_elements`, `max_value`)
- **Constantes** : `MAJUSCULES` (`MAX_SIZE`)
- **Fonctions** : `snake_case` (`compute_sum`, `is_contained`)
- **Longueur des lignes** : 80 caractères max
- **Accolades** : ouverte en fin de ligne

Principe

Cohérence du projet prime sur la convention choisie.

Commentaires

Deux styles de commentaires en C :

```
// Single-line comment (since C99)

/* Multi-line
   comment */
```

- Documenter les **types** et les **fonctions** (on va y revenir)
- Un **bon identifiant** se passe souvent de commentaire
- Éviter de **paraphraser** le code (on va y revenir)
- Attention aux commentaires **obsolètes** (pire que l'absence)

Interpréter les messages du compilateur

Fichier `error.c` :

```
1  #include <stddef.h>
2
3  int main(void) {
4    printf("Hello, world!\n");
5    return 0;
6  }
```

```
$ gcc -Wall -Wextra -std=c23 -o error error.c
error.c:4:3: error: implicit declaration of 'printf'
```

- **Ligne** : 4, **colonne** : 3 — pas d'`#include <stdio.h>`
- `error` — bloque la compilation
- **Déclaration implicite** : `printf` inconnu du compilateur (type de retour, paramètres attendus)

Un exemple d'avertissement

Fichier `warning.c` :

```c
#include <stdio.h>

int main(void) {
    int x = 42;
    printf("%f\n", x);
    return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o warning warning.c
warning.c:5:12: warning: '%f' wants 'double', got 'int'
```

- **Ligne** : 5, **colonne** : 12 — %f attend un double, x est un int
- warning — la compilation **réussit quand même**

→ Programme incorrect — peut planter à l'exécution

- **Toujours corriger les avertissements !**

Débogage

Affichage

À l'aide de `printf` ou `fprintf` :

```
1  #include <stdio.h>
2
3  int main(void) {
4      int x = 42;
5      fprintf(stderr, "DEBUG: x = %d\n", x);
6      return 0;
7  }
```

- `stderr` — sortie d'erreur, non bufferisée

Débogueur

- gdb = GNU Debugger
- Vu en laboratoire