Le C pour le TP canon

Physique, 2026-2027

Jonathan Lemus, Orestis Malaspinas

2026

Informatique et Systèmes de Communication, HEPIA

Compiler et lancer

Du code source au programme

- Un programme C est d'abord un **fichier texte** : `main.c`, le **code source**.
- L'ordinateur ne sait pas l'exécuter tel quel : un **compilateur** le traduit en **exécutable** (du langage machine).

- Pour le TP, on ne lance pas le compilateur à la main : on utilise `make`.

Compiler avec make

Dans un terminal, depuis le dossier du TP :

```
cd Canon_ball
make           # compile: crée bin/canon
make run       # compile si besoin, puis lance
make clean     # efface les fichiers compilés
```

- Le fichier `Makefile` est fourni : il contient toutes les options, rien à y modifier.
- Après chaque modification de `main.c`, relancer `make run`.

Lire les messages du compilateur

```
main.c:42:5: error: 'px' undeclared
main.c:57:12: warning: unused variable 'vy_0'
```

- `main.c:42:5` : le fichier, la **ligne** (42) et la colonne (5).
- **error** : aucun exécutable n'est produit, il faut corriger.
- **warning** : l'exécutable est produit, mais c'est souvent un vrai bug. On les corrige aussi.
- Toujours corriger la **première** erreur d'abord : les suivantes en découlent souvent.

Structure d'un programme

Le programme minimal

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

int main() {
    // un commentaire: ignoré par le compilateur
    printf("Bonjour\n");
    return EXIT_SUCCESS;
}
```

- Le programme commence toujours au début de `main`.
- Chaque **instruction** se termine par un point-virgule `;`.
- Les accolades `{ }` regroupent des instructions en un **bloc**.
- `return EXIT_SUCCESS;` termine le programme : tout s'est bien passé.
- Pour ce TP tous les includes se trouvent dans le squelette `main.c`.

Variables et types

Qu'est-ce qu'une variable ?

Une variable est une case mémoire qui a un **nom**, un **type** et une **valeur**.

```
double v_0;           // déclaration
v_0 = 4.0;            // affectation
double x_0 = -1.0;    // déclaration et initialisation
```

- Une variable doit être déclarée **avant** d'être utilisée.
- Une variable déclarée dans un bloc `{ }` n'existe que dans ce bloc.
- Un nom contient des lettres, des chiffres et `_` : pas d'espace, pas d'accent, et il ne commence pas par un chiffre.
- Une variable non initialisée contient une valeur **quelconque** : toujours l'initialiser.

Le signe = n'est pas une égalité

`=` veut dire : **calcule la partie droite, puis range le résultat dans la variable de gauche**.

```
double t = 0.0;
t = t + DELTA_T;    // t vaut maintenant 1e-5
t += DELTA_T;       // raccourci: même chose
```

- En maths, $t = t + \Delta t$ est faux. En C, c'est une **mise à jour** de `t`.
- Pour **comparer** deux valeurs, on utilise `==` (voir plus loin).

Constantes avec #define

#define

```c
#define GRAVITY (-9.81)
#define DELTA_T (1e-5)
#define MAX_T   (100.0)
```

- Avant la compilation, chaque `GRAVITY` du code est **remplacé** par `(-9.81)`.
- Par convention, les noms sont en **MAJUSCULES**.
- Pas de `=` et pas de `;` à la fin de la ligne.
- Tous les réglages du TP sont en haut du fichier : on les change à un seul endroit.

Pourquoi des parenthèses ?

```
#define LARGEUR 1.0 + 1.0
double d = 2.0 * LARGEUR;    // 2.0 * 1.0 + 1.0 = 3.0 !
```

```
#define LARGEUR (1.0 + 1.0)
double d = 2.0 * LARGEUR;    // 2.0 * (1.0 + 1.0) = 4.0
```

- Le remplacement est **textuel** : les priorités des opérateurs s'appliquent ensuite.
- Règle : toujours entourer la valeur d'un `#define` de parenthèses.

Calculer

Les opérateurs arithmétiques

`+`, `-`, `*`, `/` avec les priorités habituelles, et des parenthèses pour forcer l'ordre.

```
double aire = M_PI * r * r;
double moyenne = (a + b) / 2.0;
```

- Il n'y a **pas** d'opérateur puissance : v_0^2 s'écrit `v_0 * v_0`.
- Le symbole `^` existe en C, mais ce n'est **pas** une puissance : `2 ^ 3` vaut 1 !

Piège : la division entière

Si les deux nombres sont entiers, `/` donne un **entier** : la partie décimale est perdue.

```
double a = 1 / 2;          // a vaut 0.0 !
double b = 1.0 / 2;        // b vaut 0.5
double c = 0.5 * t * t;    // le plus simple
```

- Avec `1 / 2 * g * t * t`, le résultat vaut toujours 0 : le boulet ne tombe jamais.
- Règle : dans les calculs physiques, écrire les nombres avec un point : `2.0`, `0.5`.

Le modulo %

`a % b` donne le **reste** de la division entière de `a` par `b` (entiers seulement).

```
7 % 3       // 1
40 % 20     // 0
41 % 20     // 1
```

- `n % SHOW_ITER == 0` est vrai une fois sur `SHOW_ITER` : cela sert à n'afficher qu'une partie des itérations.

Comparaisons et logique

- Comparer : `<`, `<=`, `>`, `>=`, `==` (égal), `!=` (différent).
- Combiner : `&&` (ET), `||` (OU), `!` (NON).
- Le résultat est `true` ou `false`.

```
bool trop_loin = (x > 1.0) || (x < -1.0);
bool continuer = !done && t < MAX_T;
```

- Piège : `if (x = 0)` **affecte** 0 à `x` au lieu de comparer. Il faut écrire `==`.
- Comparer deux `double` avec `==` est risqué à cause des arrondis : préférer `<` ou `<=`.

Bibliothèque mathématique

math.h

```c
double s = sin(angle);      // angle en radians
double c = cos(angle);
double a = asin(0.5);       // vaut M_PI / 6
double r = sqrt(2.0);       // racine carrée
double n = round(2.6);      // 3.0
double f = fabs(-9.81);     // valeur absolue: 9.81
```

- `M_PI` est la constante π.
- La bibliothèque mathématique est déjà liée par le `Makefile`.

Radians et degrés

- `sin`, `cos` et `asin` travaillent **en radians** : $\alpha_{rad} = \alpha_{deg} \cdot \pi/180$.

```c
double angle_deg = 30.0;
double angle_rad = angle_deg * M_PI / 180.0;
printf("sin(30 degrés) = %g\n", sin(angle_rad));
```

- `asin(x)` n'existe que pour $-1 \leq x \leq 1$. Sinon, le résultat vaut `nan` (*not a number*) : pour le canon, la cible est hors de portée.
- Un `nan` contamine tous les calculs suivants : affichez l'angle pour le vérifier.

Structures de contrôle

if / else

```c
if (v_0 <= 0.0) {
    printf("Vitesse invalide\n");
} else if (v_0 > 100.0) {
    printf("Vitesse trop grande\n");
} else {
    printf("Vitesse correcte\n");
}
```

- Un bloc n'est exécuté que si sa condition est vraie.
- `else if` n'est testé que si les conditions précédentes étaient fausses.
- Toujours mettre les accolades, même pour une seule instruction.
- Dans le TP : la cible est-elle touchée ? Le boulet est-il au sol ?

La boucle for

```c
for (int i = 0; i < 5; i += 1) {
    printf("%d\n", i);   // affiche 0 1 2 3 4
}
```

La parenthèse contient trois parties séparées par `;` :

1. `int i = 0` : exécuté **une fois**, au début ;
2. `i < 5` : testé **avant chaque tour** ; s'il est faux, la boucle s'arrête ;
3. `i += 1` : exécuté **à la fin** de chaque tour.

La boucle de simulation

```
bool done = false;       // fenêtre fermée?
bool finished = false;   // boulet arrivé?
for (double t = 0; !done && !finished && t < MAX_T;
     t += DELTA_T) {
    // 1. nouvelle position du boulet
    // 2. cible touchée? sol touché?
    // 3. affichage, de temps en temps
}
```

- Le temps avance par pas : $t_{n+1} = t_n + \Delta t$.
- La boucle s'arrête dès qu'une des trois conditions devient fausse.
- `done` et `finished` sont des **drapeaux** : des `bool` qu'on met à `true` pour arrêter.

Fonctions

Définir une fonction

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

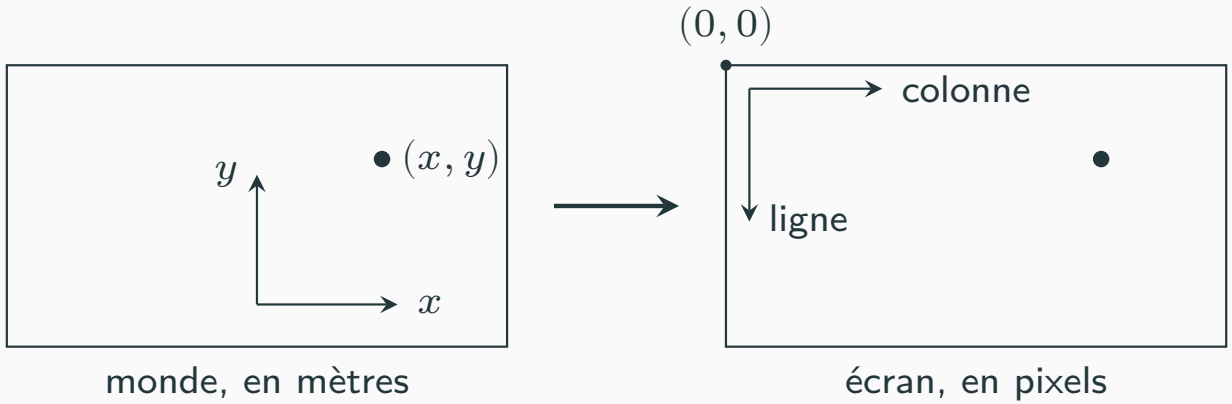
- `double` avant le nom : le type de la valeur **retournée**.
- `(double x)` : les **paramètres**, chacun avec son type.
- `return` : renvoie le résultat et quitte la fonction.
- Une fonction se définit **en dehors** de `main`, et **avant** d'être utilisée.

Appeler une fonction

```
double dx = px - px_target;
double d2 = carre(dx) + carre(py - py_target);
```

- Les arguments sont **copiés** dans les paramètres : la fonction ne peut pas modifier `dx`.
- Une fonction qui ne renvoie rien a le type `void`, par exemple `void gfx_present(...)`.

Du monde (mètres) à l'écran (pixels)



- On calcule en **mètres**, dans la boîte `WORLD_X_MIN` … `WORLD_Y_MAX`.
- On convertit en **pixels** uniquement pour dessiner.
- L'axe vertical est **retourné** : à l'écran, la ligne 0 est en haut.

Les fonctions de conversion

Ces fonctions sont à recopier dans `main.c`, à la place des fonctions vides du squelette (étape 4) :

```
uint32_t x_to_column(double x) {
    return (uint32_t)round((x - WORLD_X_MIN) * SCALE_X);
}
uint32_t y_to_row(double y) {
    return SCREEN_HEIGHT - 1
        - (uint32_t)round((y - WORLD_Y_MIN) * SCALE_Y);
}
uint32_t length_to_pixels(double length) {
    return (uint32_t)round(length * SCALE_X);
}
```

- `x - WORLD_X_MIN` : distance au bord gauche, en mètres. Fois `SCALE_X`, le nombre de pixels par mètre : la même distance en pixels.
- `SCREEN_HEIGHT - 1 - ...` : retourne l'axe vertical, pour que la ligne 0 soit en haut.

Utiliser les fonctions de conversion

```
uint32_t col = x_to_column(px);      // mètres -> colonne
uint32_t row = y_to_row(py);         // mètres -> ligne
uint32_t r = length_to_pixels(RB);   // rayon en pixels
```

- On garde `px`, `py` et `RB` en mètres pour la physique.
- On ne convertit qu'au moment de dessiner, juste avant d'appeler une fonction gfx.

L'API gfx

API gfx : le dessin

```c
// un pixel
void gfx_putpixel(struct gfx_context_t *ctxt,
    uint32_t column, uint32_t row, uint32_t color);
// un segment entre deux pixels
void gfx_draw_line(struct gfx_context_t *ctxt,
    uint32_t column0, uint32_t row0,
    uint32_t column1, uint32_t row1, uint32_t color);
// un disque plein de centre (c_column, c_row), de rayon r
void gfx_draw_full_circle(struct gfx_context_t *ctxt,
    uint32_t c_column, uint32_t c_row,
    uint32_t r, uint32_t color);
// la touche pressée, ou 0 si aucune (non bloquant)
SDL_Keycode gfx_keypressed();
```

- Positions et rayons en **pixels**, $(0, 0)$ en haut à gauche.
- Couleurs : `0xRRGGBB`, ou `COLOR_BLACK`, `COLOR_RED`, ... (voir `gfx/gfx.h`).

N'afficher qu'une itération sur SHOW_ITER

- Avec DELTA_T $= 10^{-5}$ s, il faut 100 000 itérations pour simuler 1 seconde.
- Afficher chaque itération serait beaucoup trop lent : on n'affiche qu'une itération sur SHOW_ITER.

```
if (((int)round(t / DELTA_T)) % SHOW_ITER == 0) {
    // effacer, dessiner, présenter, quit_signal
}
```

- `t / DELTA_T` est le numéro de l'itération, un `double` : on l'arrondit, puis on le convertit en `int` pour pouvoir utiliser `%`.
- Ce test est fourni dans le squelette.

Afficher du texte

printf

```
printf("Cible touchée!\n");
printf("angle = %g degrés\n", angle_deg);
printf("x = %f m, y = %f m\n", px, py);
printf("image numéro %d\n", n);
```

- `\n` passe à la ligne suivante.
- `%g` et `%f` sont remplacés par un `double`, `%d` par un `int`.
- Les valeurs sont données après le texte, **dans l'ordre** des `%`.
- Très utile pour **déboguer** : afficher une valeur pour vérifier un calcul.

Récapitulatif

Quelles notions pour quelle étape ?

Dans le TP	Notions de C
compiler et lancer	`make`, `make run`, messages d'erreur
étape 1 : cible, rayons, test de touche	`#define`, `double`, `if`, `bool`, logique
étape 2 : accélération	variables, `const`
étape 3 : angle de tir	`math.h`, radians, division
étape 4 : trajectoire et affichage	boucle `for`, conversions, API gfx, `printf`
étape 5 : Verlet	ordre des affectations, `if`/`else`

Conseils

- Compiler **souvent**, après chaque petite modification.
- Vérifier les formules sur papier avant de les coder.
- Afficher les valeurs intermédiaires avec `printf`.
- Ne pas ignorer les warnings : ils signalent souvent un vrai bug.
- Toujours calculer en mètres et en secondes ; convertir en pixels au dernier moment.