Introduction au C

CM2 - IG3

Marc Hartley
marc.hartley@umontpellier.fr

Polytech Montpellier

February 6, 2025

Overview

Les pointeurs

```
??? get_heures_et_minutes(int nombre_de_minutes) {
    int heures = nombre_de_minutes / 60;
    int minutes = nombre_de_minutes % 60;
    return ???
}

int main() {
    int temps_passe = 90; // (90 minutes = 1h30)

    int heures = ???;
    int minutes = ???;

    get_heures_et_minutes(temps_passe);

    printf("%d h %d min \n", heures, minutes);
    return 0;
}
```

En C, une fonction ne peut retourner qu'une seule valeur...

Les pointeurs

```c
void get_heures_et_minutes(int nombre_de_minutes) {
    int heures = nombre_de_minutes / 60;
    int minutes = nombre_de_minutes % 60;
    // Pas de retour, pas besoin
}

int main() {
    int temps_passe = 90; // (90 minutes = 1h30)

    int heures = ???;
    int minutes = ???;

    get_heures_et_minutes(temps_passe);

    printf("%d h %d min \n", heures, minutes);
    return 0;
}
```

En C, une fonction ne peut retourner qu'une seule valeur... On ne peut pas se fier à un `return`...

Les pointeurs

```c
void get_heures_et_minutes(int nombre_de_minutes, int heures, int minutes) {
    heures = nombre_de_minutes / 60;
    minutes = nombre_de_minutes % 60;
    // Pas de retour
}

int main() {
    int temps_passe = 90; // (90 minutes = 1h30)

    int heures;
    int minutes;

    get_heures_et_minutes(temps_passe, heures, minutes);

    printf("%d h %d min \n", heures, minutes); // Affiche "0 h 90 min"
    return 0;
}
```

Une variable donnée en "copie" n'est pas modifiée par la fonction...

Les pointeurs

```c
void get_heures_et_minutes(int nombre_de_minutes, int* heures, int* minutes) {
    *heures = nombre_de_minutes / 60;
    *minutes = nombre_de_minutes % 60;
    // Pas de retour
}

int main() {
    int temps_passe = 90; // (90 minutes = 1h30)

    int heures;
    int minutes;

    get_heures_et_minutes(temps_passe, &heures, &minutes);

    printf("%d h %d min \n", heures, minutes); // Affiche "1 h 30 min" !!
    return 0;
}
```

Les pointeurs autorisent à modifier notre variable !

Les pointeurs – Le scanf

```
int scanf_naif(char* format_de_saisie) {
    // Appels système pour récuperer l'entrée utilisateur
    int ma_variable = ...; // Un peu de magie pour transformer en entier
    return ma_variable
}

int main() {
    printf("Donner un nombre : ");
    int nombre = scanf_naif("%d");
    printf("Vous avez choisi %d", nombre);
    return 0;
}
```

Les pointeurs – Le scanf

```c
float scanf_naif(char* format_de_saisie) {
    // Appels système pour récuperer l'entrée utilisateur
    float ma_variable = ...; // Un peu de magie pour transformer en float
    return ma_variable
}

int main() {
    printf("Donner un flottant : ");
    float nombre = scanf_naif("%f");
    printf("Vous avez choisi %f", nombre);
    return 0;
}
```

On a du changer la *signature* de la fonction.

Les pointeurs – Le scanf

```c
void scanf_naif(char* format_de_saisie, float* ma_variable) {
    // Appels système pour récuperer l'entrée utilisateur
    *ma_variable =  (float) ...; // Un peu de magie pour transformer en float
    return ma_variable
}

int main() {
    float nombre;
    printf("Donner un flottant : ");
    scanf_naif("%f", &nombre);
    printf("Vous avez choisi %f", nombre);
    return 0;
}
```

Avant de trouver une bonne solution, on va passer par l'utilisation de pointeurs.

`(float)` est une *conversion de type* pour préciser que la case mémoire doit être organisée sous la forme d'un `float`.

Les pointeurs – Le scanf

```c
void scanf_naif(char* format_de_saisie, void* ma_variable) {
    // Appels système pour récuperer l'entrée utilisateur
    if (format_de_saisie est égale à "%d") {
        *ma_variable = (int) ...; // Un peu de magie pour transformer en entier
    } else if (format_de_sortie est égal à "%f") {
        *ma_variable = (float) ...; // Magie pour un float
    }
}
int main() {
    int unInt;
    float unFloat;
    printf("Donner un int puis un float : ");
    scanf_naif("%d", &unInt); // Le contenu de unInt est int
    scanf_naif("%f", &unFloat); // Le contenu de unFloat est un float
    printf("Vous avez choisi %d et %f", unInt, unFloat);
    return 0;
}
```

`void*` est un pointeur de type "indéfini" mais, comme tout pointeur, il pointe juste vers une adresse.

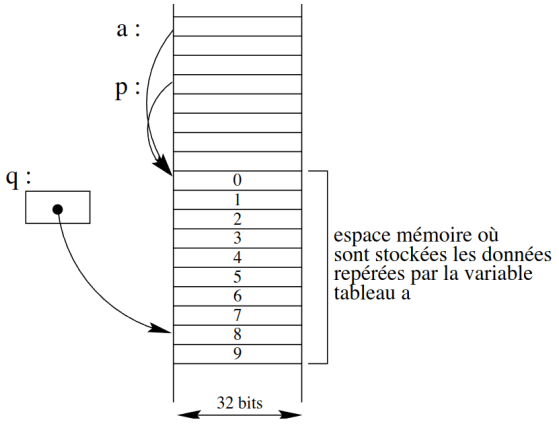
Les pointeurs – Le scanf

```
void scanf_naif(char* format_de_saisie, void* ma_variable) {
    // Appels système pour récuperer l'entrée utilisateur
    if (format_de_saisie est égale à "%d") {
        *((int*) ma_variable) = (int) ...; // Un peu de magie pour transformer en entier
    } else if (format_de_sortie est égal à "%f") {
        *((float*) ma_variable) = (float) ...; // Magie pour un float
    }
}
int main() {
    int unInt;
    float unFloat;
    printf("Donner un int puis un float : ");
    scanf_naif("%d", &unInt); // Le contenu de unInt est int
    scanf_naif("%f", &unFloat); // Le contenu de unFloat est un float
    printf("Vous avez choisi %d et %f", unInt, unFloat);
    return 0;
}
```

On doit préciser au programme que l'on souhaite stocker la donnée sous forme d'un `int` ou `float` avec un cast.

Les pointeurs

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int *p;
int *q;
p = a;
q = a + 8; // Equivalent à q = &(a[8])
```



Allouer et libérer la mémoire

Puisqu'un tableau n'est rien d'autre qu'un espace mémoire où l'on stocke des données repérées par un pointeur, on peut grâce à des fonctions d'allocation mémoire créer dynamiquement des tableaux. Le prototype de ces fonctions est :

```
#include <stdlib.h>
void* malloc(size_t nombre_d_octets_a_allouer);
void* calloc(size_t nombre_d_octets_a_allouer);
```

Ainsi la syntaxe générale pour allouer un tableau de n éléments de type type est :

```
T=(type *) malloc(n*sizeof(type));
```

Allouer et libérer la mémoire

Tout espace mémoire alloué dynamiquement doit être libéré par l'utilisateur.

On utilise pour cela la fonction

```
void* free(void* p);
```

Allouer et libérer la mémoire

Exemple de création d'un tableau de 10 entiers :

```
void main (void)
{
    int* p;
    int n = 10;
    p = (int *) malloc(n * sizeof(int));
    for(int i = 0; i < n; i++){
        p[i] = i;
    }
    printf("p[0] = %d, p[1] = %d, ...\n", p[0], p[1]);

    free(p);
}
```

n aurait pu être calculé, ou bien être saisi au clavier. C'est là l'intérêt de l'allocation dynamique de mémoire.

Comparer deux tableaux

```c
int tab1[3] = {1, 2, 3};
int tab2[3] = {1, 2, 3};

if (tab1 == tab2) {
    printf("Les tableaux sont identiques \n");
} else {
    printf("Les tableaux sont différents \n");
}
```

`tab1` et `tab2` sont deux adresses différentes, donc le code affichera toujours "Les tableaux sont différents" !

Comparer deux tableaux

```c
int tab1[3] = {1, 2, 3};
int tab2[3] = {1, 2, 3};

bool toutIdentique = true;
for (int i = 0; i < 3; i++) {
    if (tab1[i] != tab2[i]) {
        toutIdentique = false;
        break;
    }
}
if (toutIdentique) {
    printf("Les tableaux sont identiques \n");
} else {
    printf("Les tableaux sont différents \n");
}
```

Comparer deux tableaux

```
bool comparer(int* tab1, int* tab2, int n) {
    bool toutIdentique = true;
    for (int i = 0; i < n; i++) {
        if (tab1[i] != tab2[i]) {
            toutIdentique = false;
            break;
        }
    }
    return toutIdentique;
}

int main() {
    int tab1[3] = {1, 2, 3};
    int tab2[3] = {1, 2, 3};
    if (comparer(tab1, tab2, 3) == true) {
        printf("Les tableaux sont identiques \n");
    } else {
        printf("Les tableaux sont différents \n");
    }
}
```

Comparer deux chaines de caractères

```c
#include <string.h>
int main() {
    char nom1[10] = "Polytech";
    char nom2[10] = "Polytech";
    // Équivalent à {'P', 'o', 'l', 'y', 't', 'e', 'c', 'h', '\0', '\0'}

    if (strcmp(tab1, tab2) == 0) {
        printf("Les chaines sont identiques \n");
    } else {
        printf("Les chaines sont différents \n");
    }
}
```

Les chaines de caractères sont des tableaux de `char`, donc on ne peut pas les comparer "simplement" avec ==, on doit utiliser la fonction `strcmp()` ("**str**ing **comp**arison").

Pointeurs de fonction

```
void modifierTab(int* tab, int n, int (*operation)(int)) {
    for (int i = 0; i < n; i++) {
        tab[i] = operation(tab[i]);
    }
}
```

Un pointeur de fonction est défini de cette façon :

```
<type_de_retour> (*nom_du_pointeur)(<type_parametre_1>,
                <type_parametres_2>, ..., <type_parametre_n>);
```

Pointeurs de fonction

```c
void modifierTab(int* tab, int n, int (*operation)(int)) {
    for (int i = 0; i < n; i++) {
        tab[i] = operation(tab[i]);
    }
}
int doubler(int x) {
    return x * 2;
}
int auCarre(int x) {
    return x * x;
}

int main() {
    int tableau1[5] = {0, 1, 2, 3, 4};
    int tableau2[5] = {0, 1, 2, 3, 4};

    modifierTab(tableau1, 5, doubler); // tableau1 = {0, 2, 4, 6, 8}
    modifierTab(tableau2, 5, auCarre); // tableau2 = {0, 1, 4, 9, 16}
}
```

Structures

```c
struct Etudiant {
    char nom[50];
    int age;
    float note;
};

int main() {
    // On précise qu'on utilise une structure avec "struct"
    struct Etudiant etudiant1 = {"Alice", 20, 15.5};

    printf("Nom: %s\n", etudiant1.nom);
    printf("Age: %d ans\n", etudiant1.age);
    printf("Note: %.1f/20\n", etudiant1.note);

    return 0;
}
```

Structures

```c
// Raccourci 1 : typedef et struct
struct EtudiantStruct {
    char nom[50];
    int age;
    float note;
};

typedef struct EtudiantStruct Etudiant;

int main() {
    Etudiant etudiant1 = {"Alice", 20, 15.5};
    // Equivalent à "struct EtudiantStruct etudiant1 = {"Alice", 20, 15.5};"

    printf("Nom: %s\n", etudiant1.nom);
    printf("Age: %d ans\n", etudiant1.age);
    printf("Note: %.1f/20\n", etudiant1.note);

    return 0;
}
```

Utiliser `typedef` nous évite d'avoir à réécrire `struct` partout dans notre code.

Structures

```c
// Raccourci 2 : typedef et struct d'un coup
typedef struct {
    char nom[50];
    int age;
    float note;
} Etudiant;

int main() {
    Etudiant etudiant1 = {"Alice", 20, 15.5};

    printf("Nom: %s\n", etudiant1.nom);
    printf("Age: %d ans\n", etudiant1.age);
    printf("Note: %.1f/20\n", etudiant1.note);

    return 0;
}
```

Utiliser `typedef` nous évite d'avoir à réécrire `struct` partout dans notre code.

Lire et écrire dans un fichier

Toutes les fonctions pour lire et écrire sont identique à lire et écrire dans la console, avec un "f" devant (pour "File")

```c
FILE* file;

file = fopen("example.txt", "w");
if (file == NULL) {
    perror("Erreur lors de l'ouverture du fichier");
    return 1;
}

fprintf(file, "Hello, Polytech!\n");
fprintf(file, "Ceci est un exemple de fichier texte.\n");
fclose(file);
```

On utilise un pointeur vers un `FILE` (!)
On ouvre un fichier avec `fopen` et on le referme avec `fclose`.
On écrit avec `fprintf(ptr_fichier, mon_texte)`.

Lire et écrire dans un fichier

```
file = fopen(nom_du_fichier, mode_d_ouverture);
```

Le mode d'ouverture précise si on souhaite

- "r"ead juste pour la lecture, "r+" pour lecture/écriture. (Le fichier doit exister)
- "w"rite pour réécrire le contenu, "w+" pour lecture/écriture.
- "a"ppend ajouter du contenu à la fin du fichier, "a+" pour le mode lecture/écriture.

Lire et écrire dans un fichier

```
fprintf(ptr_fichier, texte_format, variable1, variable2, ...);
fprintf(file, "%d h %d min : x = %f", heures, minutes, x);
```

Utilisation identique à `printf()` mais on précise dans quel fichier on écrit.

Behind the scenes, `printf()` appelle `fprintf(stdout, ...)`, avec `stdout` un pointeur vers l'affichage de votre terminal.

On peut écrire du texte simple (sans "format" pour afficher des variables) avec `fputs(mon_texte, ptr_fichier)` et donc `puts(mon_texte)` pour écrire dans le terminal.

Lire et écrire dans un fichier

```
fscanf(ptr_fichier, texte_format, &variable1, &variable2, ...);
fscanf(file, "%d", x);
```

Utilisation identique à `scanf()` mais on précise dans quel fichier on lit.
Plus de précisions sur l'utilisation du `scanf()` par la suite..!

Lire et écrire dans un fichier

```
fgets(buffer, longueur_max, ptr_fichier);
char nom[20];
fgets(nom, 20, ptr_fichier);
printf("Le nom : %s ! \n", nom);
```

fgets() récupère le contenu d'une ligne (avec une longueur max)

Attention, la documentation précise que s'il y a un retour à la ligne dans le fichier, il est aussi ajouté au *buffer* (le texte de sortie).
Dans l'exemple précédent, si le fichier contient :

```
Alice
Bob
```

Cela va afficher :

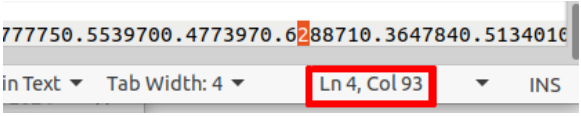
```
Le nom : Alice
!
```

On peut retirer ce `'\n'` avec

```
nom[strcspn(nom, "\n")] = '\0';
```

Lire et écrire dans un fichier

La lecture et écriture dans un fichier se fait à partir d'un *curseur* (un position), exactement comme si vous écriviez "à la main".



Au début, votre *curseur* est au début si vous ouvrez en mode "r" ou "r+" (ou "w" et "w+" car le fichier est vidé) ou à la fin si vous ouvrez en mode "a" ou "a+".

Le curseur se déplace à chaque lecture ou écriture

Lire et écrire dans un fichier

```c
char mot[100];
FILE* f = fopen("test.txt", "r+");
// Fichier initial: "Hello World"

printf("Position du curseur : %ld\n", ftell(f)); // Affiche "0"
fgets(mot, 8, f);
printf("%s\n", mot);                             // Affiche "Hello "
printf("Position du curseur : %ld\n", ftell(f)); // Affiche "7"
fputs("_", f); // Change le 7e caractère en "_"
printf("Position du curseur : %ld\n", ftell(f)); // Affiche "8"
fgets(mot, 8, f);
printf("%s\n", mot);                             // Affiche "rld\n"
printf("Position du curseur : %ld\n", ftell(f)); // Affiche "12"
```

`ftell(fichier)` nous dit à quelle position on en est.

Attention, l'écriture (ici, fputs) *remplace* le texte existant ! Le fichier devient "Hello W_rld"

Lire et écrire dans un fichier

```c
char mot[100];
FILE* f = fopen("test.txt", "r+");
// Fichier initial: "Hello World"

printf("Position du curseur : %ld\n", ftell(f)); // Affiche "0"
fgets(mot, 8, f);
printf("Position du curseur : %ld\n", ftell(f)); // Affiche "7"
fgets(mot, 8, f);
printf("Position du curseur : %ld\n", ftell(f)); // Affiche "12"
                                  // Pourtant, 7 + 7 = 14 < 12
if (feof(f)) {
    printf("Le curseur est arrivé à la fin du fichier");
}
```

On peut savoir si on a atteint la fin de notre fichier avec la fonction `feof()` ("File : End Of File")

Lire et écrire dans un fichier

```
int fseek(fichier, offset, source);

// Positionner le curseur au début du fichier
fseek(fichier, 0, SEEK_SET);
// Avancer le curseur de 100 octets depuis le début
fseek(fichier, 100, SEEK_SET);
// Reculer le curseur de 50 octets depuis la position actuelle
fseek(fichier, -50, SEEK_CUR);
// Se positionner à la fin du fichier
fseek(fichier, 0, SEEK_END);
```

`fseek()` déplace le curseur à une certaine position par rapport à une source. La source peut être :

- SEEK_SET : place le curseur *n* caractères après le début du fichier.
- SEEK_CUR : déplace le curseur *n* caractères en avant ou en arrière.
- SEEK_END : place le curseur *n* caractères avant la fin du fichier.

Lire et écrire dans un fichier binaire

Petite nuance pour les fichiers binaires

```c
FILE* file;

file = fopen("example.bin", "wb");
int numbers[] = {1, 2, 3, 4, 5};
int number, i;

if (file == NULL) {
    perror("Erreur lors de l'ouverture du fichier");
    return 1;
}

fwrite(numbers, sizeof(int), 5, file);

fclose(file);
```

On utilise un pointeur vers un `FILE` (!)
On ouvre un fichier avec `fopen` et on le referme avec `fclose`.
On écrit maintenant avec `fwrite()` et on lit avec `fread()`.

Lire et écrire dans un fichier binaire

```
file = fopen(nom_du_fichier, mode_d_ouverture);
```

On ajoute un "b" pour préciser "binaire" :

- "rb"ead juste pour la lecture, "rb+" pour lecture/écriture. (Le fichier doit exister)
- "wb"rite pour réécrire le contenu, "wb+" pour lecture/écriture.
- "ab"ppend ajouter du contenu à la fin du fichier, "ab+" pour le mode lecture/écriture.

Lire et écrire dans un fichier binaire

```
fwrite(ptr_vers_mes_donnees, taille_de_chaque_donnee, nb_donnees, ptr_fichier);

float mon_tableau[3] = {3.1415f, 2.7182f, 1.4142f};
fwrite(mon_tableau, sizeof(float), 3, fichier);

char c = 'H';
fwrite(&c, sizeof(char), 1, fichier);
```

On copie directement le contenu de la mémoire RAM vers le contenu du fichier binaire.

Lire et écrire dans un fichier binaire

```
fread(ptr_vers_mes_donnees, taille_de_chaque_donnee, nb_donnees, ptr_fichier);

float* mon_tablea = (float*) malloc (3 * sizeof(float));
fread(mon_tableau, sizeof(float), 3, fichier);

char c;
fread(&c, sizeof(char), 1, fichier);
```

On copie directement le du fichier binaire vers la mémoire RAM.

Lire et écrire dans un fichier

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

int main() {
        int n = 100000000;
        float* texte = (float*) malloc (n * sizeof(float));
        for (int i = 0; i < n; i++) {
                texte[i] = (float)rand()/RAND_MAX;
        }
        FILE* f1 = fopen("test.txt", "w+");
        clock_t t0 = clock();
        for (int i = 0; i < n; i++) {
                fprintf(f1, "%f", texte[i]);
        }
        clock_t t1 = clock();
        fclose(f1);
        FILE* f2 = fopen("test.bin", "wb+");
        clock_t t2 = clock();
        fwrite(texte, sizeof(texte[0]), n, f2);
        clock_t t3 = clock();
        fclose(f2);
        long int timeNormal = (t1 - t0) * 1000 / CLOCKS_PER_SEC;
        long int timeBinary = (t3 - t2) * 1000 / CLOCKS_PER_SEC;
        printf("Normal : %ldms \n", timeNormal);
        printf("Binaire: %ldms \n", timeBinary);
        printf("Speedup: x%f \n", (float)(timeNormal) / (float)(timeBinary));
        return 0;
}
```

```
Normal : 17861ms
Binaire: 376ms
Speedup: x47.502659

test.txt : 800 Mo
test.bin : 400 Mo
```

Modularité

Ai-je besoin de préciser qu'il faut toujours séparer son code en un maximum de fonctions et de fichiers ?
En C, on peut avoir des fichier **source** (.c) et **header** (.h [d'entête])

- Pour tout fichier source, un fichier header est associé (excepté main.c).
- Les headers contiennent les *signatures* de tous les fichiers, ainsi que les variables globales (à éviter). C'est un sommaire du code source.
- Le fichier source contient tout le code des fonctions.
- On peut inclure les fichiers headers pour utiliser ses fonctions.

Modularité

Header .h :

```h
#ifndef NOM_DE_MON_FICHIER_H
#define NOM_DE_MON_FICHIER_H

void auCarre(float* x);
int maFonctionA(int a, float b);
void maFonctionB();

#endif
```

Source .c :

```c
#include "nom_de_mon_fichier.h"

void auCarre(float* x) {
    *x *= *x;
}
int maFonctionA(int a, float b) {
    // ...
    return 0;
}
void maFonctionB() {
    // ...
}
```

main.c

```c
#include "nom_de_mon_fichier.h"
#include <stdio.h>
int main() {
    float x = 3.0;
    auCarre(&x);
    printf("x * x = %f", x);
    return 0;
}
```

Compiler plusieurs fichiers

On doit préciser à GCC tous les fichiers source utiles :

```
gcc -Wall main.c fichier1.c fichier2.c -o mon_programme
```

Pas besoin de préciser les fichier .h, ils sont implicitement demandés par les fichiers source.

Compiler plusieurs fichiers

Quelques erreurs typiques :

- `undefined reference to ‘auCarre’` : Vous avez probablement oublié de rajouter votre fichier source. Ou vous avez oublier d'écrire la fonction "auCarre" dans le fichier source.
- `unknown type name 'Etudiant'` : Vous avez probablement oublié d'inclure le fichier "fichier.h" dans main.c.
- `multiple definition of ‘auCarre’; [...] first defined here` ou `redefinition of 'Etudiant'` : Vous avez possiblement oublié de rajouter les `#ifndef`, `#define` et `#endif` dans le fichier header, qui protègent d'inclure plusieurs fois les mêmes bout de codes. Ou vous avez défini des fonctions ou variables portant le même nom.
- `undefined reference to ‘main‘` : Vous avez possiblement compilé vos fichiers dans un mauvais ordre. Commencez par compiler "main.c" en premier.
- N'hésitez pas à demander à Google pour toute autre erreur !

Outils

- Exploitez à fond les IDE ! (Eclipse, Code::Blocks, VSCode, Netbeans, ...)
  - Profitez de leur autocomplétion pour éviter les erreurs de frappe,
  - Observez les codes soulignés en rouge ou orange, qui symbolisent des erreurs ou des warnings,
  - Compiler avec eux permet d'éviter des erreurs dans la ligne de commande de GCC, d'oublier des fichiers, ou des bibliothèques externes,
  - Ils proposent systématiquement un débuggeur interactif, qui permet d'executer un programme ligne par ligne et de comprendre le comportement d'un code.
- Exploitez Google :
  - Recherchez "C + nom d'une fonction", le premier résultat vous explique cette fonction et vous montre des exemples.
  - Recherchez "C + votre objectif", dans les 3 premiers résultats MAX, vous trouvez le nom de la fonction qu'il vous faut.
- Limitez l'utilisation de LLM durant votre apprentissage : C'est en se plantant qu'on apprend le mieux et le plus vite.

Sujets non-abordés

Ce cours reste une courte introduction au langage C. Il manque donc beaucoup de matière ! Parmi les notions vues, votre travail à la maison est :

- Comprendre le principe de "formats" utilisé avec "printf()" et "scanf()", et ainsi comprendre ce que fait cette instruction :
  `scanf("%d:%d:%d", &heures, &minutes, &secondes);`.
- Comment et quand doit-on appliquer des cast (conversions de type).
- Trouver une documentation pour la manipulation de chaines de caractères (on a vu "strcmp", mais il existe aussi "strlen", "strcat", "strcpy", "strchr", ...)
- Pareil pour les tableaux (allez voir "memcpy", "memcmp", ...)

La suite en C ?

- Exploitation des arguments d'entrée du programme,
- Gestion de processus (programmation multi-coeurs, en parallèle, sur GPU),
- Utilisation et envoi de packets sur le réseau,
- Utilisation de GUI,
- Optimisation,
- Sécurité (gestion d'overflow, contrôle des entrées utilisateur, gestion d'erreurs, sécurité des pointeurs, ...)

Nous vous conseillons fortement de pratiquer les langages enseignés pour mieux les comprendre.

Quelques idées de projets

Quelques idées qui pourraient vous entrainer au C, et vous avancer sur les prochains projets :

- Implémenter une bibliothèque pour gérer des listes chaînées en C.
- Développer un petit shell en C qui peut exécuter des commandes simples du système d'exploitation.
- Créer une application en C qui simule une base de données très basique avec des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer)
- Écrire un programme en C qui implémente un algorithme de cryptage/décryptage simple, comme le chiffrement par substitution ou le chiffrement de César