Le Langage C

class: lead

Support de cours complet — Réda BOUREBABA

📝

Changelog — V0.0.6

- 01/02/2026 20:05 — Fusion slides "Organisation de la mémoire" + "Segments mémoire" en une seule slide pour meilleure compéhension visuelle du layout mémoire (diagramme + description segmentée).

🎯

Objectifs du cours

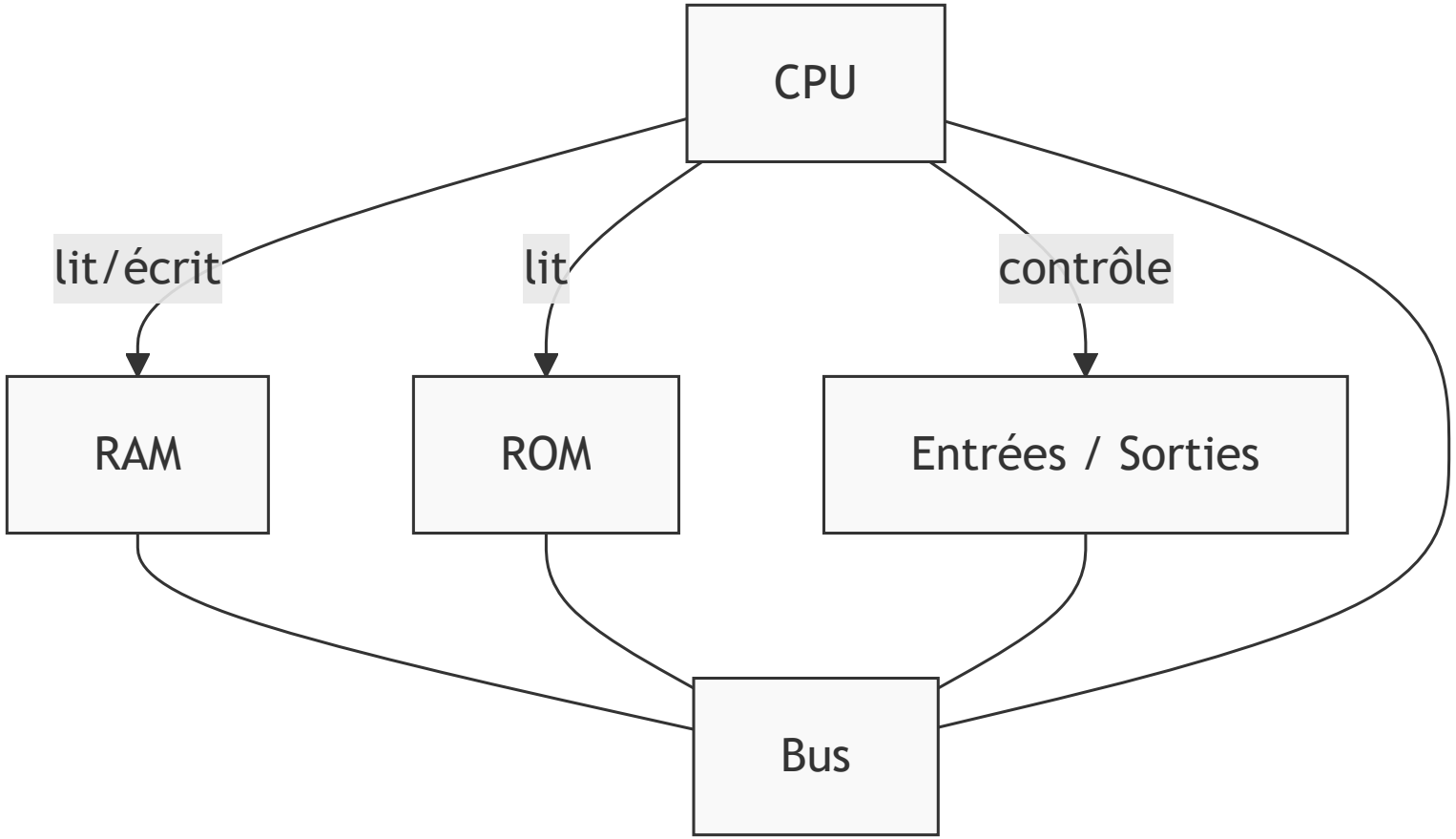
- Maîtriser les fondamentaux du langage C
- Comprendre la gestion mémoire et les pointeurs
- Savoir manipuler fichiers et structures de données
- Pratiquer via TPs et exercices guidés

Histoire du C

- Créé en 1972 par Dennis Ritchie & Ken Thompson
- Développé en même temps que UNIX
- Stabilisé en 1978 par Kernighan & Ritchie → C K&R

🏛️

Architecture CPU/RAM/ROM



Premier programme : Hello World

```c
#include <stdio.h>

int main(void){
  printf("Hello World!\n");
  return 0;
}
```

💾 Organisation de la mémoire

Segments mémoire

- `.text` : code exécutable
- `.data` : variables globales initialisées
- `.bss` : variables globales non initialisées
- `heap` : allocation dynamique (malloc)
- `stack` : variables locales et appels de fonctions

Les types — rappel

Type	Taille	Plage
`char`	1 octet	-128 à 127
`int`	4 octets	-2 147 483 648 à 2 147 483 647
`long`	8 octets	très grand
`float`	4 octets	3.4×10^{-38} à 3.4×10^{38}
`double`	8 octets	1.7×10^{-308} à 1.7×10^{308}

Types — usage moderne

À l'origine, les types servaient à économiser la mémoire.

Aujourd'hui, la mémoire est abondante :

- Utilisez `int` pour les entiers
- Utilisez `float` pour les nombres à virgule

💻

Contexte historique

En janvier 1973 : premier micro-ordinateur **Micral** (François Gernelle, R2E)

- Basé sur Intel i8008
- 500 kHz, 8 ko RAM

Déclarer une variable

```c
int maVariable;  // déclaration
```

→ Réserve 4 octets en RAM référencés par `maVariable`

Initialisation :

```c
maVariable = 10;
```

Ou en une seule ligne :

```c
int maVariable = 10;
```

Les constantes

```c
const int NOMBRE_DE_VIES_MAXI = 5;
```

- Convention : écriture en MAJUSCULES
- Le compilateur refuse toute modification

Afficher une variable (printf & formats)

```c
int a = 10;
float b = 12.3;
printf("les valeurs de a:%d, b:%f\n", a, b);
```

Les variables sont remplacées dans l'ordre d'apparition.

Formats disponibles :

Format	Type attendu
`%d`	int
`%ld`	long
`%f`	float/double
`%c`	char
`%s`	chaîne

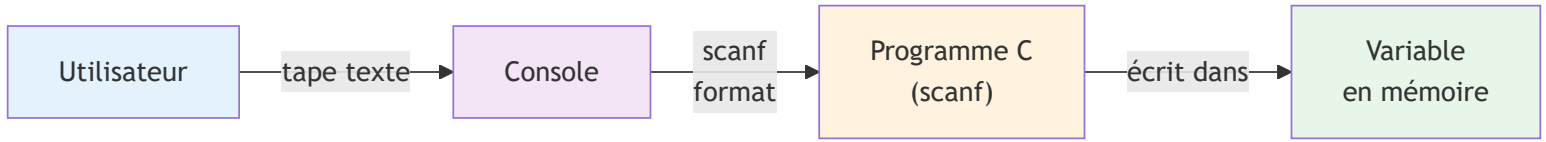
Récupérer une saisie utilisateur (scanf)

```c
int age = 0;
printf("Quel age avez vous ? ");
scanf("%d", &age);
printf("Vous avez %d ans\n", age);
```

⚠️
Attention : utiliser `&age` (adresse de la variable)

Comportement :

- Nombre décimal → tronqué (2.4 devient 2)
- Chaîne → valeur non modifiée
- Débordement possible avec `%s` → **danger**



🔒

Sécurité — lecture sécurisée

Alternative à scanf pour les chaînes : `fgets`

```c
char nom[50];
printf("Votre nom : ");
fgets(nom, sizeof(nom), stdin);
// Supprime le \n final
nom[strcspn(nom, "\n")] = '\0';
```

⚠️
Évitez absolument :

- `gets()` — retiré de C11, dangereux
- `scanf("%s", ...)` sans limite de taille

✅
Préférez : `fgets()` avec taille limite

🧮

Opérateurs arithmétiques

Opération	Symbole
Addition	`+`
Soustraction	`-`
Multiplication	`*`
Division	`/`
Modulo	`%`

Raccourcis d'affectation

Raccourci	Équivalent
`++` , `+=`	Incrémentation
`--` , `-=`	Décrémentation
`*=`	Multiplication
`/=`	Division
`%=`	Modulo

🔢 Bibliothèque mathématique

```c
#include <math.h>
```

Fonctions disponibles :

- `double fabs(double f)` : valeur absolue
- `double ceil(double f)` : arrondi supérieur
- `double floor(double f)` : arrondi inférieur
- `double pow(double nombre, int puissance)` : puissance
- `double sqrt(double nombre)` : racine carrée
- `double sin(double radians)`, `cos`, `tan` : trigonométrie
- `double exp(double nombre)`, `log(double nombre)` : exponentielle, logarithme

Conditions — if/else

```
if(/* condition */){
    /* condition vraie */
}else{
    /* condition fausse */
}
```

Chaînage :

```
if(/* condition */){
    /* condition vraie */
}else if(/* autre condition */){
    /* autre condition vraie */
}else{
    /* aucune des conditions vraie */
}
```

Opérateurs de comparaison

Symbole	Signification
`==`	égal
`>`	supérieur
`<`	inférieur
`>=`	supérieur ou égal
`<=`	inférieur ou égal
`!=`	différent

Switch... case

```c
int choix = 10;
switch(choix){
    case 1:
        /* choix == 1 */
        break;
    case 10:
        /* choix == 10 */
        break;
    default:
        /* aucun choix valide */
}
```

🔁 Boucle while

```c
while (/*condition*/){
    // répéter tant que la condition est vraie
}
```

Boucle do...while

```
do{
    // je serai exécuté au moins une fois
}while(/*condition*/)
```

Boucle for

Au début `i` vaut 0, tant que `i` est inférieur à 10 on exécute le code puis on ajoute 1 à `i`.

```
for(i=0; i<10; i++){
    // code
}
```

TP — Devine un nombre

L'ordinateur tire au sort un nombre entre 1 et 100.
Il vous demande de deviner. Vous entrez un nombre.
L'ordinateur compare et indique si le nombre mystère est supérieur ou inférieur.
Et ainsi de suite, jusqu'à ce que vous trouviez le nombre.

TP — Devine un nombre (squelette)

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

int main(){
    const int MAX = 100, MIN=1;
    srand(time(NULL));
    int randomNumber = (rand() % (MAX - MIN + 1)) + MIN;

    /* à vous !! */
}
```

🛠️

Les fonctions

Afin de mieux organiser le code, il faut le découper en fonctions.

Structure :

```
type nomFonction(type parametre1, type parametre2)
{
    // Instructions
}
```

- `type` : ce que la fonction renvoie (`int` , `double` , `void` si rien)
- `nomFonction` : même règle que variables (pas d'accents, pas d'espace)
- `paramètres` : les paramètres passés à la fonction

Fonctions — exemple

```c
// Déclarer la fonction triple
int triple(int nombre){
    return 3 * nombre;
}

int main(int argc, char *argv[]){
    int entree = 10, nb_triple = 0;
    nb_triple = triple(20);  // nb_triple = 60

    printf("triple est egal à %d", triple(10));
    printf(" le triple de %d est %d\n", entree, nb_triple);
    return 0;
}
```

Fonctions — TP

- L'utilisateur saisit les dimensions d'un rectangle → fonction calcule son aire
- L'utilisateur saisit les dimensions d'un cercle → fonction calcule son aire
- L'utilisateur saisit les dimensions d'un carré → fonction calcule son aire
- À partir d'un menu, l'utilisateur choisit :
  - La figure pour laquelle il veut effectuer un calcul
  - S'il veut calculer l'aire ou le périmètre

👉

Les pointeurs — le problème

Les pointeurs sont une spécificité du C/C++.

Problème : une fonction ne peut renvoyer qu'une seule valeur.

Exemple qui **ne fonctionne pas** :

```c
void makeHumanTime(int h, int m){
    h = m / 60;
    m = m % 60;
}

int main(int argc, char* argv[]){
    int heures = 0, minutes = 90;
    makeHumanTime(heures, minutes);
    printf("%d heures, %d minutes", heures, minutes);
    return 0;
}
```

Sortie : `0 heures, 90 minutes`

Pointeurs — adresse & valeur

```c
// affichage d'une valeur
int a = 10;
printf("La valeur de a vaut %d\n", a);

// affichage de l'adresse de a
printf("L'adresse de a vaut %p\n", &a);
```

Sortie :

```
La valeur de a vaut 10
L'adresse de a vaut 0061FEE4
```

Pointeurs — déclaration

```
int *ptr1, *ptr2, *ptr3;
```

Initialisation (recommandée) :

```
int *ptr1 = NULL, *ptr2 = NULL, *ptr3 = NULL;
```

Affectation :

```
int age = 10;
int *ptrAge = &age;
```

`ptrAge` est un pointeur sur `int` qui pointe sur `age`.

Pointeurs — accès à la valeur

```c
int age = 10;
int *ptrAge = &age;
printf("%d\n", *ptrAge);  // affiche 10
```

Pointeurs — passage à une fonction

```c
void triple(int *ptrN){
    *ptrN *= 3;
}

int main(){
    int nb = 5;
    int *ptrNb = &nb;
    triple(ptrNb);
    printf("%d\n", *ptrNb);  // affiche 15
}
```

Pointeurs — solution au problème initial

```c
void makeHumanTime(int *pH, int *pM){
    *pH = *pM / 60;
    *pM = *pM % 60;
}

int main(int argc, char* argv[]){
    int heures = 0, minutes = 90;
    makeHumanTime(&heures, &minutes);
    printf("%d heures, %d minutes", heures, minutes);
    return 0;
}
```

Sortie : `1 heures, 30 minutes`
✅

Opérateur sizeof

L'opérateur `sizeof` renvoie la taille en octets :

- `sizeof(int)` : 4
- `sizeof(double)` : 8
- `sizeof(float)` : 4
- `sizeof(char)` : 1
- `sizeof(int[10])` : 40
- `sizeof(int*)` : 4 (ou 8 sur système 64 bits)

Pointeurs — arithmétique

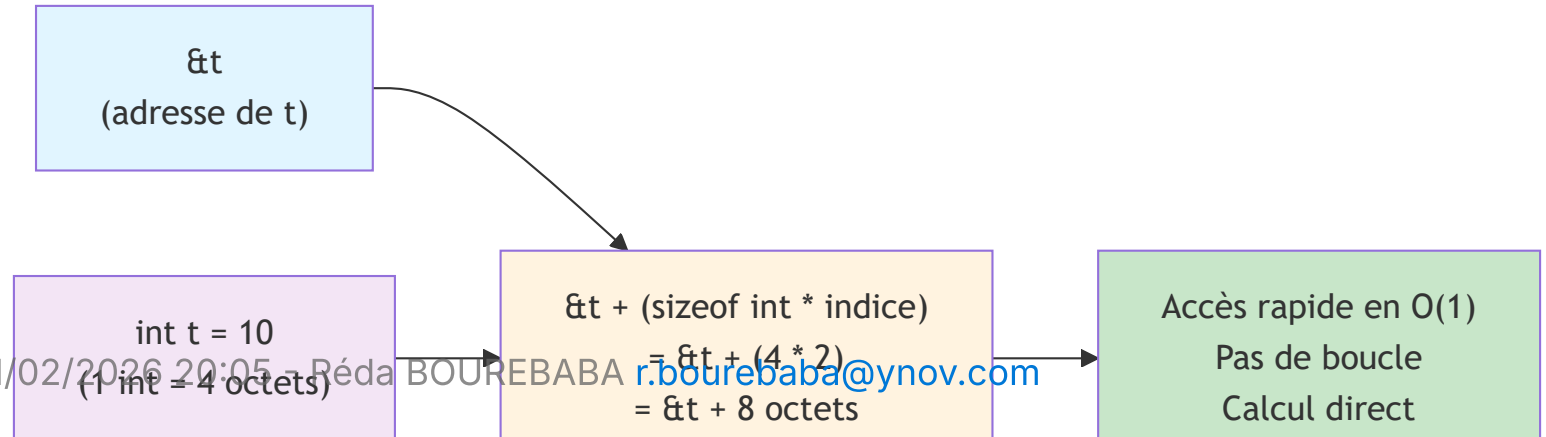
Affectation :

```
int age = 10;
int *pAge = &age;
int *pAge2 = pAge;  // pointent sur le même objet
```

Si `P` pointe sur `A[i]` d'un tableau :

- `P+n` pointe sur `A[i+n]`
- `P-n` pointe sur `A[i-n]`
- `P++` pointe sur `A[i+1]`
- `P--` pointe sur `A[i-1]`

⚠️
Ces opérations sont valides **seulement à l'intérieur d'un même tableau**.



Pointeurs — arithmétique (exemples)

```c
int A[10];
int *P;

P = A+9;   /* dernier élément -> légal */
P = A+11;  /* dernier élément + 2 -> illégal */
P = A-1;   /* premier élément - 1 -> illégal */
```

Les tableaux

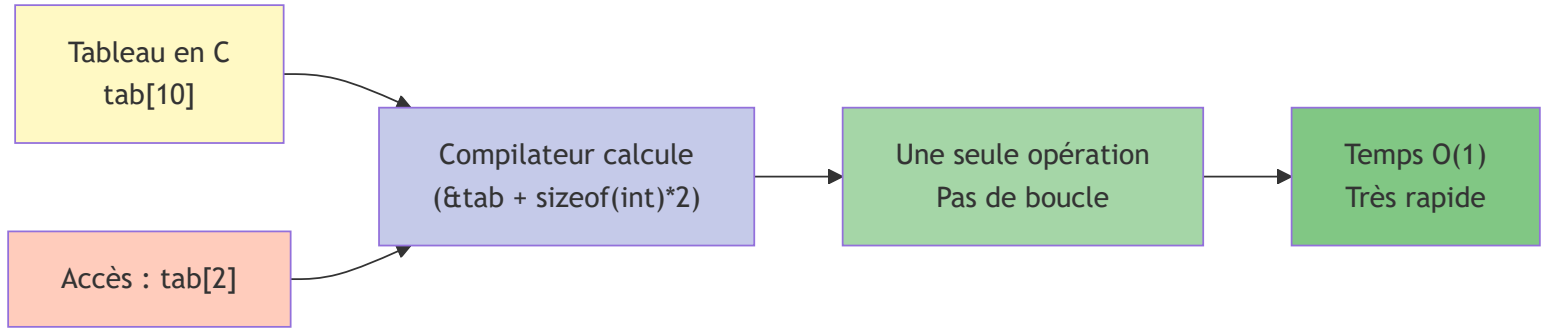
```c
// déclaration
int tableau[4];

// initialisation
int autre_tableau[4] = {10, 33, 12, 5};

// écriture
tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;

// lecture
printf("%d", tableau[1]);
```

Accès O(1) : le compilateur calcule directement l'adresse `&tableau + sizeof(int)*indice`



TP Pointeurs

Soit un tableau d'entiers : `1 2 3 4 1 2 3 4 5`

1. Codez une fonction qui renvoie le nombre d'occurrences d'un chiffre donné :

```
int compte(const int* tab, int longueur_tab, int chiffre_a_compter)
```

2. Codez une fonction qui remplit un tableau avec les valeurs d'un autre tableau multiplié par une valeur :

```
void multi(const int* tab_in, int longueur_tab, int* tab_out, int multiplicateur)
```

TP Pointeurs (suite)

3. Codez une fonction qui applique une opération (addition, multiplication, soustraction, division) :

```
void calcul(const int* tab_in, int longueur_tab, int* tab_out, int operation, int valeur)
```

Astuce : `int size = sizeof(tab) / sizeof(int)`

abc

Chaînes de caractères

En C, les chaînes de caractères sont techniquement un **tableau de** `char`.

Un ordinateur ne stocke que des nombres → table de conversion **ASCII**.
http://www.asciitable.com/

```c
int main(int argc, char *argv[])
{
    char lettre = 'A';
    printf("%d\n", lettre);  // affiche 65
    return 0;
}
```

Chaînes — affichage avec printf

```c
int main(int argc, char *argv[])
{
    char lettre = 'A';
    printf("%c\n", lettre);  // affiche A
    return 0;
}
```

Le format `%c` permet d'afficher un caractère.

Chaînes — lecture simple avec scanf

```c
int main(int argc, char *argv[])
{
    char lettre = 0;
    scanf("%c", &lettre);
    printf("%c\n", lettre);
    return 0;
}
```

Chaînes — stockage

Une chaîne de caractères est stockée en mémoire comme un **tableau**.

⚠️
Cette chaîne doit être terminée par un `\0` (caractère nul).

Donc une chaîne de 5 lettres est un tableau de **6 éléments**.

Chaînes — exemple de stockage

```c
int main(int argc, char *argv[])
{
    char chaine[6]; // S-a-l-u-t + \0

    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0';

    printf("%s", chaine);  // affiche Salut
    return 0;
}
```

Chaînes — scanf

Pour lire une chaîne, surdimensionnez le tableau :

```c
int main(int argc, char *argv[])
{
    char prenom[100];

    printf("Comment t'appelles-tu ? ");
    scanf("%s", prenom);
    printf("Salut %s", prenom);
    return 0;
}
```

Question : Pourquoi manque-t-il le `&` devant `prenom` ?

Chaînes — strlen

```c
int main(int argc, char *argv[])
{
    char chaine[] = "Salut";
    int longueurChaine = 0;

    longueurChaine = strlen(chaine);
    printf("La chaine %s fait %d caracteres de long", chaine, longueurChaine);
    return 0;
}
```

🛠️

TP — Recodez strlen

Recodez la fonction `strlen` qui renvoie la taille d'une chaîne de caractères.

https://gitlab.bzctoons.net/bzctoons/tp-c

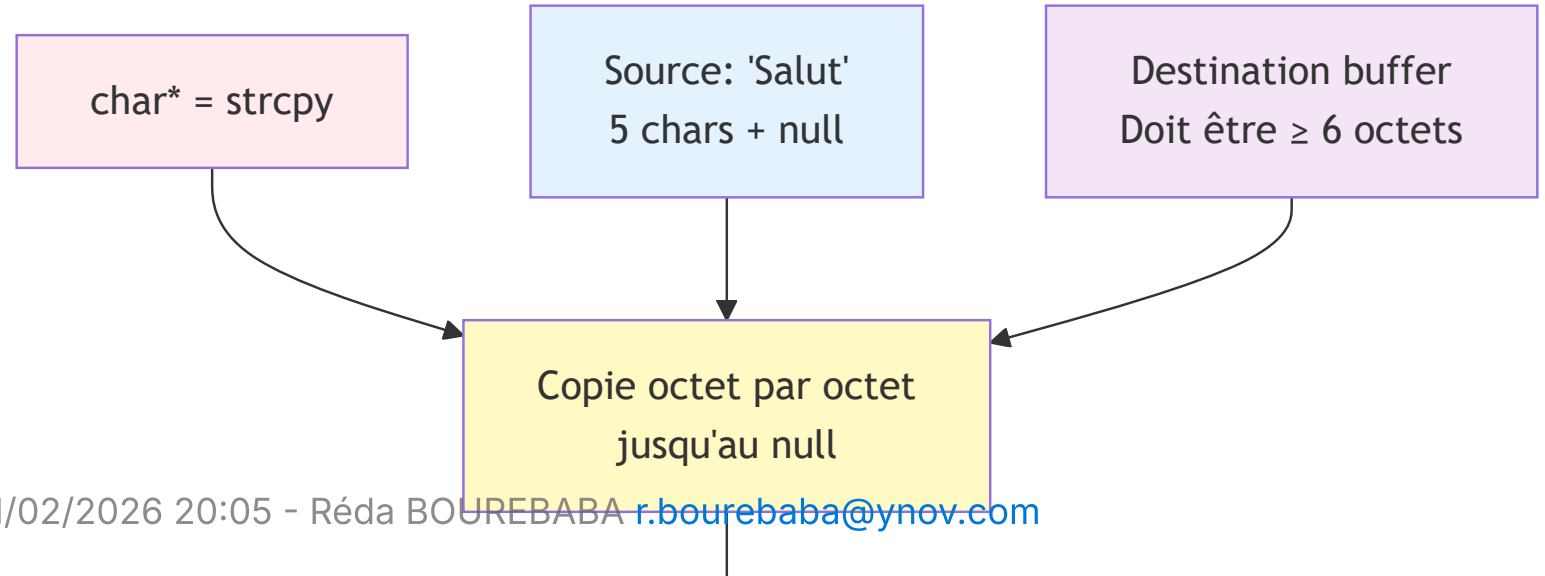
Chaînes — strcpy

```c
char* strcpy(char* copieDeLaChaine, const char* chaineACopier);
```

```c
int main(int argc, char *argv[])
{
    char chaine[] = "Texte", copie[100] = {0};

    strcpy(copie, chaine);

    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```



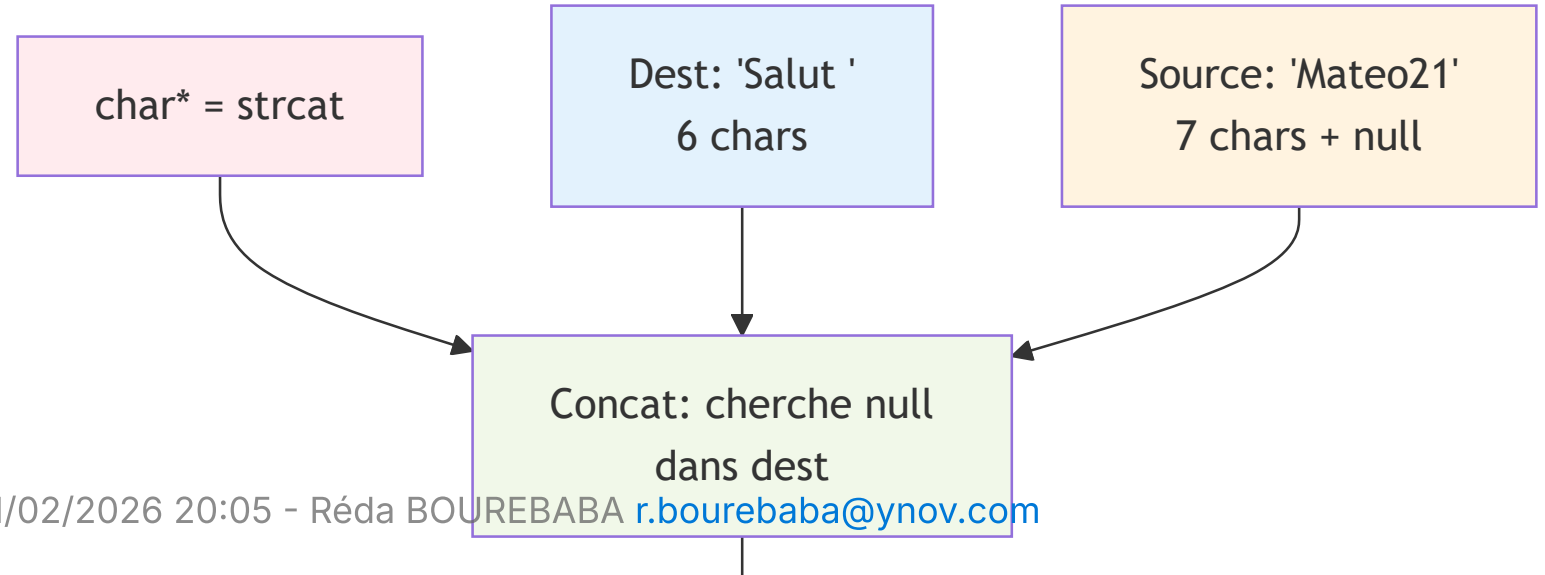
Chaînes — strcat

```
char* strcat(char* chaine1, const char* chaine2)
```

```
int main(int argc, char *argv[])
{
    char chaine1[100] = "Salut ", chaine2[] = "Mateo21";

    strcat(chaine1, chaine2);

    printf("chaine1 vaut : %s\n", chaine1);  // Salut Mateo21
    printf("chaine2 vaut toujours : %s\n", chaine2);
    return 0;
}
```



🔒

Chaînes — versions sécurisées

Problème : `strcpy` et `strcat` ne vérifient pas les débordements.

Solutions sécurisées :

```
// strncpy : copie avec limite
strncpy(dest, src, sizeof(dest) - 1);
dest[sizeof(dest) - 1] = '\0';  // garantir terminaison

// strncat : concaténation avec limite
strncat(dest, src, sizeof(dest) - strlen(dest) - 1);
```

⚠️ `strncpy` ne termine pas toujours par `\0` → toujours le faire manuellement.

Chaînes — fonctions utiles

- `int strlen(const char*)` : longueur
- `void strcpy(char* dest, const char* src)` : copie
- `void strcat(char* dest, const char* src)` : concaténation
- `int strcmp(const char* s1, const char* s2)` : comparaison (renvoie 0 si égales)
- `int strchr(const char* s, char c)` : rechercher un caractère
- `int strstr(const char* s1, const char* s2)` : rechercher une chaîne

TP — Recodez les fonctions strings

Recodez :

- `strlen` : longueur d'une chaîne
- `strcpy` : copie d'une chaîne
- `strcat(const char* A, const char* B, char* dest)` : concaténation
- `strcmp` : comparaison
- `strchr` : rechercher un caractère
- `strstr` : recherche une chaîne dans une autre

Bonus : recodez les fonctions strings de PHP
https://www.php.net/manual/fr/ref.strings.php

TP — TDD (Test Driven Development)

Codez un programme qui met en œuvre les fonctions ci-dessus.

Vous devez le coder en TDD et il doit afficher votre score à la fin : **1 point par fonction**.

https://gitlab.bzctoons.net/bzctoons/tp-c

🔧

Préprocesseur

Le préprocesseur est un programme qui s'exécute **juste avant la compilation**.

Les lignes commençant par `#` sont des **directives préprocesseur**.

Nous en avons déjà vu une : `#include`

Préprocesseur — #include

Pour inclure un fichier fourni par le compilateur :

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

Pour inclure des fichiers de votre projet :

```
#include "monfichier.h"
```

Préprocesseur — #include (exemple)

functions.c :

```c
void test1(){
    printf("test1\n");
}
void test2(const char * text){
    printf("%s\n", text);
}
```

functions.h :

```c
void test1();
void test2(const char * text);
```

TP — Création de librairie

- `main.c` : votre main
- `functions.h` : les prototypes des fonctions
- `functions.c` : le code des fonctions

```c
int main(){
    test1();
    test2("hello");
}
```

Préprocesseur — #define

`#define` est une constante préprocesseur.

```
#define NB_ITEMS 3
int main(){
    int nb_items = NB_ITEMS;
}
```

⚠️
À ne pas confondre avec les constantes (`const`).

Il est aussi possible de définir une constante sans valeur :

```
#define CONSTANTE
```

Préprocesseur — #define (calculs)

```
#define LARGEUR 500
#define HAUTEUR 100
#define SURFACE LARGEUR * HAUTEUR
```

Constantes prédéfinies :

- `__LINE__` : numéro de ligne
- `__FILE__` : nom du fichier
- `__DATE__` : date de compilation
- `__TIME__` : heure de compilation

```
printf("Erreur a la ligne %d du fichier %s\n", __LINE__, __FILE__);
```

Préprocesseur — macros

```c
#define BONJOUR() printf("Bonjour");

int main(){
    BONJOUR();
    return 0;
}
```

Devient après préprocesseur :

```c
int main(){
    printf("Bonjour");;
    return 0;
}
```

Préprocesseur — macros sur plusieurs lignes

```c
#define BONJOUR()  printf("Bonjour\n"); \
                   printf("Tu vas bien ?\n");

int main(){
    BONJOUR();
    return 0;
}
```

Préprocesseur — macros avec paramètres

```c
#define MAJEUR(age)  if(age > 18) \
                         printf("Tu es majeur\n");

int main(){
    MAJEUR(21);
    return 0;
}
```

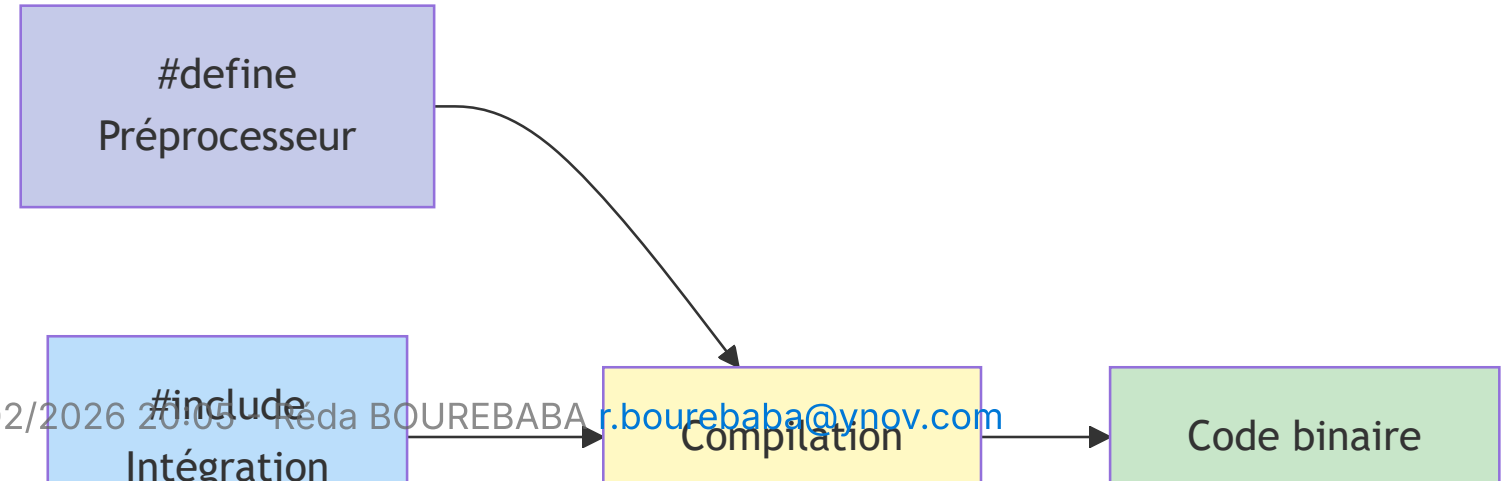
Devient :

```c
int main(){
    if(21 > 18)
        printf("Tu es majeur\n");
    return 0;
}
```

Préprocesseur — compilation conditionnelle

```c
#if condition
    /* si la condition est vraie */
#elif condition2
    /* si la condition2 est vraie */
#endif

#define WINDOWS
#ifdef WINDOWS
    /* Code source pour Windows */
#endif
#ifdef LINUX
    /* Code source pour Linux */
#endif
#ifndef CONSTANTE
    /* code exécuté si CONSTANTE n'est pas définie */
#endif
```



Préprocesseur — exemple conditionnel

```
#define IS_FRENCH 0

#if IS_FRENCH
    #define TEXT_TEST1 "bonjour\n"
#else
    #define TEXT_TEST1 "hello\n"
#endif

void test1(){
    printf(TEXT_TEST1);
}
```

🏷️

Typedef

`typedef` permet de définir des **synonymes de types** (alias).

```
typedef un_type synonyme_du_type;
```

Exemples :

```
#define TRUE 1
#define FALSE 0
typedef int BOOL;
BOOL bValue = TRUE;

typedef char* STRING;
STRING sText = "hello";

typedef char* STRLIST[];
STRLIST liste = {"hello", "world"};
```

Structures

Une structure est un **assemblage de variables** de différents types.

Elles sont généralement définies dans un `.h`.

```
struct MaStruct{
    int var1;
    int var2;
    double monDouble;
};
```

Le point-virgule à la fin est **obligatoire**.

Structures — utilisation

```c
typedef struct MaStruct MaStruct;

struct MaStruct{
    int a;
    int b;
    int c;
};

int main(){
    struct MaStruct ma_struct;
    MaStruct ma_struct2;  // grâce au typedef
    MaStruct ma_struct3 = {0,0,0};

    ma_struct.a = 10;
    ma_struct.b = 20;
    ma_struct.c = 30;
    printf("a:%d, b:%d, c:%d\n", ma_struct.a, ma_struct.b, ma_struct.c);
}
```

Structures — pointeurs

```
MaStruct ma_struct3 = {0,0,0};
MaStruct *pMa_struct3 = &ma_struct3;

pMa_struct3->a = 10;
pMa_struct3->b = 20;
pMa_struct3->c = 30;
printf("a:%d, b:%d, c:%d\n", pMa_struct3->a, pMa_struct3->b, pMa_struct3->c);
```

🔢 Énumérations

Une énumération est aussi un **type personnalisé**. Elle contient une liste de valeurs possibles.

```
typedef enum Volume Volume;
enum Volume{
    FAIBLE, MOYEN, FORT
};
Volume musique = MOYEN;

if(musique == FORT){
    // ...
}
```

Le compilateur associe un entier : `FAIBLE=0`, `MOYEN=1`, `FORT=2`.

Énumérations — valeurs personnalisées

```
enum Volume{
    FAIBLE=0, MOYEN=50, FORT=100
};
```

TP — Périmètre d'un polygone convexe

Calculez le périmètre du polygone suivant.

Résultat attendu : 28.21

- Vous devez utiliser des **structures**
- Votre code doit comporter au moins une fonction de calcul
- Cette fonction doit pouvoir calculer le périmètre de **n'importe quel polygone convexe**
- Les points sont en dur dans le code

Tableaux de structures — initialisation

```c
struct MaStruct{
    int a, b, c;
};

struct MaStruct mon_tableau[] = {
    {1,2,3},
    {4,5,6},
    {7,8,9},
    {10,11,12}
};
```

💾 Allocation dynamique

```c
// allocation d'un bloc
void* malloc(size_t taille_a_allouer);

// allocation + remise à zéro
void* calloc(size_t nombre, size_t taille_d_un_element);

// réallocation d'un bloc existant
void* realloc(void* bloc_existant, size_t taille_a_allouer);

// libération d'un bloc
free(void* bloc_a_liberer);
```

Allocation dynamique — exemple

```c
int * tableau_dynamic = (int*)malloc(100 * sizeof(int));
if (tableau_dynamic == NULL) {
    printf("Erreur allocation\n");
    return 1;
}

// Utilisation du tableau

free(tableau_dynamic);
tableau_dynamic = NULL;  // bonne pratique
```

⚠️
Règles :

- Toujours vérifier si `malloc` renvoie `NULL`
- Toujours `free` après usage → évite fuites mémoire
- Mettre à `NULL` après `free` → évite double-free

Les fichiers — ouverture

```
FILE* fopen(const char* nomDuFichier, const char* modeOuverture);
```

Modes d'ouverture :

- `r` : lecture seule, le fichier doit exister
- `w` : écriture seule, crée ou vide le fichier
- `a` : ajout à la fin, crée le fichier si besoin
- `r+` : lecture/écriture, le fichier doit exister
- `w+` : lecture/écriture, crée ou vide le fichier
- `a+` : lecture/écriture à la fin

En ajoutant un `b` : mode binaire.

Fichiers — ouverture/fermeture

```c
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");

    if (fichier != NULL)
    {
        // On peut lire et écrire dans le fichier
        fclose(fichier);
    }
    else
    {
        printf("Impossible d'ouvrir le fichier test.txt");
    }
    return 0;
}
```

Fermeture : `int fclose(FILE* pointeurSurFichier);`

Fichiers — écriture

- Écriture d'un caractère :

```
int fputc(int caractere, FILE* pointeurSurFichier);
```

- Écriture d'une chaîne :

```
char* fputs(const char* chaine, FILE* pointeurSurFichier);
```

- Écriture formatée (similaire à printf) :

```
int fprintf(FILE* pointeurSurFichier, const char* format, ...);
```

Fichiers — lecture

- Lecture d'un caractère (renvoie EOF à la fin) :

```
int getc(FILE* pointeurSurFichier);
```

- Lecture d'une chaîne (s'arrête au `\n` ou longueur max) :

```
char* fgets(char* chaine, int longueur, FILE* pointeurSurFichier);
```

- Lecture formatée (similaire à scanf) :

```
int fscanf(FILE* pointeurSurFichier, const char* format, ...);
```

Fichiers — lecture exemple 1

```c
#define TAILLE_MAX 1000

int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    char chaine[TAILLE_MAX] = "";

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        while (fgets(chaine, TAILLE_MAX, fichier) != NULL)
        {
            printf("%s", chaine);
        }
        fclose(fichier);
    }
    return 0;
}
```

Fichiers — lecture exemple 2

```c
int main(int argc, char *argv[])
{
    FILE* fichier = NULL;
    int score[3] = {0};

    fichier = fopen("test.txt", "r");

    if (fichier != NULL)
    {
        while (fscanf(fichier, "%d,%d,%d\n", &score[0], &score[1], &score[2]) != -1){
            printf("Les meilleurs scores sont : %d, %d et %d\n", score[0], score[1], score[2]);
        }
        fclose(fichier);
    }
    return 0;
}
```

Fichiers — se déplacer

- Renvoie la position du pointeur :

```
long ftell(FILE* pointeurSurFichier);
```

- Se positionner (fichiers binaires) :

```
int fseek(FILE* pointeurSurFichier, long deplacement, int origine);
```

Origine :

- `SEEK_SET` : début de fichier
- `SEEK_CUR` : position actuelle
- `SEEK_END` : fin de fichier

Fichiers — renommer/supprimer

Renommer :

```c
int rename(const char* ancienNom, const char* nouveauNom);
```

Supprimer :

```c
int remove(const char* fichierASupprimer);
```

TP — Fichiers (carnet de contacts)

Structure :

```
struct contact{
    char firstname[50];
    char lastname[50];
    int age;
};
```

Format fichier : CSV `contacts.txt`

```
Bob,Marley,75
Alice,Wonder,28
```

- **10 points** : `createContacts.exe` — saisir et enregistrer des contacts
- **10 points** : `contactsList.exe` — afficher nombre et liste des contacts
- **12 points** : `searchContacts.exe` — recherche par nom/prénom/age

Exemples :

```
searchContacts.exe "firstname:bob,lastname:marley,age:20"
searchContacts.exe "any:bob"
```

⚠️ La recherche est **insensible à la casse**.

Arguments en ligne de commande

```c
int main(int argc, char *argv[])
```

- `argc` : nombre d'arguments
- `argv` : liste des arguments (tableau de chaînes)

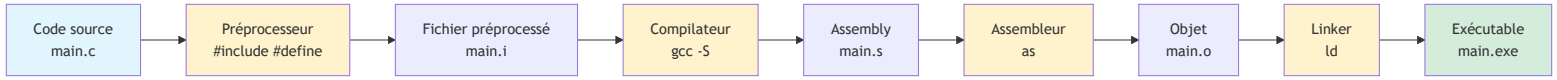
`argv[0]` contient le nom de l'exécutable.

Exemples :

- `monprog.exe 1 arg1 arg2` → 3 arguments
- `monprog.exe "1 arg1 arg2"` → 1 argument

🔨

Processus de compilation



Compiler avec GCC

```
gcc <liste de fichier source *.c> -o <executable de sortie>
```

Exemple :

```
gcc main.c list.c -o list.exe
```

Flags recommandés :

```
gcc -Wall -Wextra -Werror -g -O2 main.c -o main
```

- `-Wall -Wextra` : active tous les warnings
- `-Werror` : warnings = erreurs
- `-g` : symboles de debug (pour gdb/valgrind)
- `-O2` : optimisation niveau 2

Debugging avec GDB

Compiler avec `-g` pour activer les symboles :

```
gcc -g -Wall main.c -o main
gdb ./main
```

Commandes GDB essentielles :

```
break main          # point d'arrêt sur main
run                 # exécuter le programme
next                # ligne suivante (sans entrer)
step                # ligne suivante (entre dans fonctions)
print variable      # afficher valeur
continue            # continuer jusqu'au prochain break
quit                # quitter gdb
```

🔍 Valgrind — détection fuites mémoire

```
valgrind --leak-check=full ./main
```

Détecte :

- Fuites mémoire (`malloc` sans `free`)
- Accès mémoire invalides
- Utilisation de mémoire non initialisée

Exemple de sortie :

```
HEAP SUMMARY:
  in use at exit: 400 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 400 bytes allocated

LEAK SUMMARY:
  definitely lost: 400 bytes in 1 blocks
```

Références & ressources

- Table ASCII : http://www.asciitable.com/
- man pages : `man scanf`, `man fopen`, `man malloc`
- Exercices et référentiels fournis dans le cours

class: small
© C Avancé M2 — Réda BOUREBABA