

Langage C - Avancé

Ressources

<https://cheatsheets.zip/c>

Les Tableaux

Les tableaux sont une structure de données qui permet de stocker une collection d'éléments de même type.

Usage

Pour déclarer et utiliser les valeurs d'un tableau, on utilise la syntaxe suivante:

```
int t[5]; // Tableau de 5 entiers

t[0] = 1; // On affecte à la première valeur du tableau

t[1] = t[0];
```

c

On peut également initialiser un tableau lors de sa déclaration, et dans ce cas il n'est pas nécessaire de préciser la taille du tableau.

```
int t[] = {1, 2, 3, 4, 5}; // Tableau de 5 entiers
```

c

Les index des tableaux commencent à 0. Le dernier index est donc `taille - 1`.

INFO

La taille du tableau est déterminée au moment de la compilation, et ne peut pas être modifiée par la suite.

Dans les fonctions

Lorsque l'on passe un tableau en paramètre, on indique `int[]` comme type.

Attention: les fonctions ne peuvent pas connaître la taille d'un tableau, il faut donc passer en plus un entier indiquant la taille du tableau en paramètre.

```
void fonction(int t[], int taille) {}
```

Exemple avancé

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

void printTableau(int t[], int taille)
{
    for (int i = 0; i < taille; i++)
    {
        printf("t[%d] = %d\n", i, t[i]);
    }
    printf("\n");
}

int main(void)
{
    int t1[5] = {1, 2, 3, 4, 5}; // explicitement taille de 5
    int t2[] = {10, 11};        // implicitement taille de 2

    printf("t1[0]: %d\n", t1[0]); // 1: on compte à partir de 0
    printf("t1[4]: %d\n", t1[4]); // 5: pour une taille de 5, le dernier est do

    t1[3] = 6;
    printf("t1[3]: %d\n", t1[3]); // t1[3]: 6

    printTableau(t1, 5);
}
```

Multidimensionnels

Les tableaux multidimensionnels sont des tableaux de tableaux. Ils peuvent être de 2, 3, 4, ... dimensions.

Un tableau de deux dimensions est communément appelé une **matrice**.

```
int t1[2][3] = {{1, 2, 3}, {4, 5, 6}}; // 2 dimensions
t1[0][0] = 1;
t1[0][1] = 2;
t1[0][2] = 3;
t1[1][0] = 4;
t1[1][1] = 5;
t1[1][2] = 6;
```

```
int t2[2][3][4]; // 3 dimensions
```

t1	0	1	2
0	1	2	3
1	4	5	6

On peut parcourir un tableau multidimensionnel avec des boucles imbriquées.

```
for (int i = 0; i < 2; i++) {  
    for (int j = 0; j < 3; j++) {  
        printf("t1[%d][%d] = %d\n", i, j, t1[i][j]);  
    }  
}
```

Chaines de caractères

Les chaînes de caractère ne sont pas un type primitif en C, mais sont utilisables grâce à des tableaux de caractères (`char[]`).

```
char c1[] = "Bonjour"; // Taille implicite  
char c2[255] = "Au revoir"; // Taille explicite  
  
c1[0] = 'Z'; // On peut modifier les caractères d'une chaîne de caractères
```

On peut afficher une chaîne de caractères avec la fonction `printf` en utilisant le caractère spécial `%s` .

Pour récupérer une chaîne de caractère, il sera préférable d'utiliser la fonction `fgets` plutôt que `scanf` (`scanf` ne lit pas les espaces).

```
char c1[255] = "Inconnu";  
printf("Entrez votre nom: ");  
fgets(c1, 255, stdin);  
printf("Bonjour %s\n", c1);
```

WARNING

Attention à ne pas confondre les simples `'` et les doubles `"` .

- `'a'` est un caractère

- `"a"` est une chaîne de caractères

On ne peut pas réassigner entièrement une chaîne de caractères après l'avoir déclarée, on ne peut que modifier ses caractères un par un.

```
char c1[] = "Bonjour";  
c1 = "Au revoir"; // Pas possible, erreur de compilation  
c1[0] = 'Z'; // Ok
```

Fin de chaîne

Les chaînes de caractères sont terminées par un caractère nul `\0`, qui indique la fin de la chaîne même si le tableau contient d'autres caractères après.

- Avec l'exemple précédent, `c1` est de taille 7 (Bonjour + `\0`).
- `c2` quand à lui est de taille 255 mais `c[9] = '\0'`

Fonctions utilitaires

strlen

La fonction `strlen` permet de connaître la longueur d'une chaîne de caractères.

```
printf("La longueur de c1 est %d\n", strlen(c1));
```

strcmp

La fonction `strcmp` permet de comparer deux chaînes de caractères. Elle renvoie 0 si les deux chaînes sont égales, -1 si la première chaîne est plus petite que la deuxième et 1 si la première chaîne est plus grande que la deuxième.

```
printf("c1 et c2 sont %s\n", strcmp(c1, c2) == 0 ? "égales" : "différentes");
```

Exemple avancé

```
#include <stdio.h>  
#include <string.h>  
  
void inverse(char str[], int len)  
{  
    for (int i = 0; i < len / 2; i++)  
    {  
        char tmp = str[i];  
        str[i] = str[len - i - 1];  
        str[len - i - 1] = tmp;  
    }  
}
```

```

        str[len - i - 1] = tmp;
    }
}

int main(void)
{
    char c1[] = "Bonjour";      // Taille implicite
    char c2[255] = "Au revoir"; // Taille explicite

    printf("c1 = %s\n", c1);

    if (strcmp(c1, c2) == 0)
    {
        printf("Pareil\n");
    }
    else
    {
        printf("Pas pareil\n");
    }

    printf("c1[0] = %c\n", c1[0]);
    printf("c1[6] = %c\n", c1[6]);
    printf("c2[0] = %c\n", c2[0]);

    c1[0] = 'Z';
    printf("c1 = %s\n", c1);
    printf("c1[0] = %c\n", c1[0]);

    inverse(c2, strlen(c2));
    printf("c2 = %s\n", c2);

    return 0;
}

```

Les Structures

Il arrive souvent que l'on doive stocker des données pour lesquelles une seule variable ne suffit pas. Plutôt que de créer plusieurs variables, on peut préférer créer des structures, qui peuvent elles-mêmes contenir plusieurs variables.

Exemples:

- Une heure est composée d'heures, minutes et secondes
- Un point est composé d'une abscisse et d'une ordonnée
- Un vecteur est composé d'une direction et d'une longueur

Pour déclarer une structure, on utilise la syntaxe suivante:

```
struct NomStructure
{
    int variable1;
    float variable2;
};
```

On aura alors créé un nouveau type de variables, que l'on pourra utiliser comme un type primitif en indiquant `struct NomStructure` devant le nom de la variable.

On accèdera aux variables contenues dans la structure avec l'opérateur `.`

```
struct NomStructure variable;
variable.variable1 = 2;
variable.variable2 = 3.14;
```

On peut également initialiser une structure lors de sa déclaration. Les valeurs doivent être dans le même ordre que déclarés dans la structure.

```
struct NomStructure variable = {2, 3.14};
```

Exemple avancé

```
struct Coordonnees
{
    int x;
    int y;
};

struct Coordonnees additionner_coordonnees(struct Coordonnees c1, struct Coordonnees c2)
{
    struct Coordonnees c;

    c.x = c1.x + c2.x;
    c.y = c1.y + c2.y;

    return c;
}

int main(void)
{
    struct Coordonnees c1 = {1, 2};
    struct Coordonnees c2 = {3, 4};

    struct Coordonnees c3 = additionner_coordonnees(c1, c2);
}
```

```
printf("Coordonnees: %d %d\n", c3.x, c3.y);  
}
```

La mémoire

Les variables que nous utilisons dans un programme sont stockées dans la mémoire vive de l'ordinateur lorsque le programme s'exécute.

Lors de la compilation d'un programme, le compilateur détermine la taille des variables dont chaque fonction aura besoin, et lorsque celles-ci sont exécutées, la mémoire nécessaire est allouée dans la **stack**.

Cette mémoire est organisée en une grille de cases, chacune de ces cases pouvant stocker une valeur, et chaque type de variable occupera une place plus ou moins grande.

Voir [Tailles des types](#)

Stockage des variables

```
// Exemple de déclaration de variables  
int a = 5;  
double b = 10.12;  
long c = 15000000;  
int *d = &a;
```

c

Exemple de stockage des variables dans la mémoire:

Adresse	Valeur	Type	Taille
0xBBA000	5	int	4 octets
0xBBA004	10.12	double	8 octets
0xBBA00C	15000000	long	8 octets
0x888888	0xBBA000	int*	8 octets (64 bits)

Outil pour visualiser le fonctionnement de la mémoire d'un programme:

<https://pythontutor.com/c.html#mode=edit>

Les Pointeurs

Chaque variable possède donc une adresse, qui représente l'endroit où la variable est stockée.

- Il est possible de connaître l'adresse d'une variable en utilisant l'opérateur `&` devant la variable. Lorsqu'une variable stocke une adresse, on dit qu'elle est un **pointeur**.
- Pour déclarer une variable de type pointeur, on utilise l'opérateur `*` devant le nom de la variable. Le type de la variable pointeur doit être le même que le type de la variable que l'on veut pointer.
- Si on veut lire ou modifier une valeur à une adresse d'un pointeur, on utilise l'opérateur `*` devant la variable, on dit qu'on la **déréférence**.

```
int a = 10;    // variable      - a = 10
int *p = &a;   // pointeur     - p = 0xBBA000
int b = *p;    // déréférence  - b = 10

printf("a: %d\n", a);    // 10
printf("*p: %d\n", *p);  // 10

*p = 20;

printf("a: %d\n", a);    // 20
printf("*p: %d\n", *p);  // 20
```


	Adresse	Valeur
	0	145
	1	3
	2	82
pointeurSurAge	3	177450

age	177450	10

	3 448 765 900 126 (et des poussières)	940

INFO

Si `p` est un **pointeur** vers un **entier**, alors `*p` est un **entier**.

Interet des pointeurs

Les pointeurs sont utiles dans le cas des fonctions, car elles peuvent modifier les variables déclarées en dehors de leur scope.

```

void modify(int *a)
{
    *a = 20;
}

int main(void)
{
    int a = 10;

    modify(&a);

```

C

```
printf("a: %d\n", a); // a: 20
}
```

Sans les pointeurs, la fonction `modify` ne pourrait pas modifier la valeur de `a`, car une variable passée en paramètre est en fait une copie de la variable originale.

Démonstration

C (C17 + GNU extensions)

```

1 void modify(int *a)
2 {
3     *a = 20;
4 }
5
6 int main(void)
7 {
8     int a = 10;
9
10    modify(&a);
11
12    printf("a: %d\n", a); // a: 20
13 }
```

[Edit Code & Get AI Help](#)

→ line that just executed
→ next line to execute

0

< Prev Next >

Step 1 of 8

Visualized with pythontutor.com

Print output (drag lower right corner to resize)

Stack
Heap

main

a

int
?

Note: ? refers to an uninitialized value

C/C++ [details](#): none [default view] ▼

Exemple de mauvaise utilisation

Il est important de se rappeler que les variables sont toujours passées par **copie** dans les paramètres des fonctions. Sans les pointeurs, il est impossible de modifier une variable déclarée en dehors de la fonction.

```

void bad_modify(int a)
{
    // a est une copie de la variable a dans main, pas la meme variable
    a = 20;
}

int main(void)
{
    int a = 10;

    bad_modify(a);

    printf("a: %d\n", a); // a: 10
}
```

C (C17 + GNU extensions)

```

1 void bad_modify(int a)
2 {
3     // a est une copie de la vari
4     a = 20;
5 }
6
7 int main(void)
8 {
9     int a = 10;
10
11     bad_modify(a);
12
13     printf("a: %d\n", a); // a: 1
14 }

```

[Edit Code & Get AI Help](#)

→ line that just executed
→ next line to execute

< Prev
Next >

Step 1 of 8

Print output (drag lower right corner to resize)

Stack
Heap

main

a

int
?

Note: ? refers to an uninitialized value

C/C++ [details](#): none [default view] ▼

Attention à la syntaxe

La syntaxe `*p` ne signifie pas la même chose lors de la déclaration et lors de l'utilisation.

- Lors de la déclaration `int *p`, on dit que `p` est un pointeur vers un `int`.
- Lors de l'utilisation `*p` on dit que l'on **déréférence** le pointeur `p`
 - `*p = a` signifie que l'on va stocker la valeur de `a` dans la variable pointée par `p`
 - `c = *p` signifie que l'on attribue à `c` la valeur stockée à l'adresse pointée par `p`

```

int *b = &a;

// Equivalent à
int *b;
b = &a; // On stocke une adresse dans un pointeur

// Mais pas à
int *b;
*b = &a; // On stocke une adresse dans un entier (!pas ok!)

```

La règle est simple et identique à toutes les autres variables: **On ne peut stocker dans une variable qu'une valeur de son type.**

- Dans un pointeur, on ne peut stocker qu'une adresse.
- Dans un entier on ne peut stocker qu'un entier.

Allocation dynamique

L'allocation dynamique est une technique qui permet de réserver de la mémoire en cours d'exécution.

Il arrive souvent que l'on ne sache pas combien de mémoire sera nécessaire pour stocker les données d'un programme. Cela peut arriver notamment lorsqu'on reçoit des données d'un utilisateur, via une entrée en ligne de commande (`scanf`), lorsqu'on lit un fichier ou reçoit une requête d'un serveur.

On ne peut alors pas attribuer une taille fixe au moment de la compilation, et on doit donc utiliser l'allocation dynamique au moment de l'exécution du programme.

Allocation

La fonction `malloc` permet de réserver de la mémoire, et on doit lui indiquer la taille de la mémoire à réserver en octets.

La fonction `sizeof` permet de connaître la taille d'un type en octets. Par exemple `sizeof(int)` vaut 4 octets.

Libération

La fonction `free` permet de libérer la mémoire allouée dynamiquement. Elle prend en paramètre un pointeur vers la mémoire à libérer, et elle se rappellera de combien d'octets à été attribué à ce pointeur.

Exemple

```
int *a = malloc(sizeof(int));

*a = 10;

printf("a: %d\n", *a);

free(a);
```

WARNING

Ne pas oublier de libérer la mémoire allouée ! Tout oubli pourra entraîner une fuite de mémoire, et donc une perte de performance du programme et une saturation de la mémoire vive de l'ordinateur.



Dépassement de mémoire

Lors de l'utilisation de l'allocation dynamique, il est très important de faire attention à la taille de la mémoire allouée, et de ne pas écrire sur un pointeur qui n'a pas été alloué ou qui a été libéré.

```
int *a = malloc(sizeof(int));

*a = 10;

free(a);

*a = 20; // Erreur de segmentation (segmentation fault)

int b[2];

b[3] = 10; // Erreur de segmentation (segmentation fault)
```

Le compilateur ne pourra pas détecter ce genre d'erreur, et celles-ci peuvent entraîner de graves problèmes, comme une perte de données ou un plantage du programme.

Ces erreurs de programmation sont la base des exploits de vulnérabilité de type **Buffer Overflow**. D'autres langages de programmation comme le **Rust** sont dits "memory safe" car ils empêchent ce genre d'erreurs de manière statique.

Voir la section [Buffer Overflow](#)

Pointeurs & Types

Tableaux

Les tableaux sont en réalité des pointeurs vers le premier élément du tableau. Un type tableau et un type pointeurs sont similaires, et `int[]` est similaire à `int*`

```
int a[10];

int *b = a;

a == b
```

```

a[0] == b[0]
&a[0] == a

tab[0] == *tab;
tab[1] == *(tab + 1);

```

Lorsqu'on passe un tableau en paramètre, on peut le considérer comme un pointeur vers le premier élément du tableau.

Ne pas oublier qu'on aura souvent besoin de passer la taille du tableau en paramètre !

```

void fonction(int *a, int taille) {}
// identique à
void fonction(int a[], int taille) {}

```

Allocation dynamique

On allouera souvent de la mémoire pour des tableaux. Pour cela, on multipliera la taille d'un élément du tableau par le nombre d'éléments.

```

int *a = malloc(sizeof(int) * 10);

a[0] = 10;

free(a); // Libérera toute la mémoire allouée pour le tableau

```

Structures

Lorsque l'on manipule des pointeurs vers des structures, on doit utiliser l'opérateur `->` pour accéder aux membres de la structure au lieu de l'opérateur `.`

```

struct Coordonnees
{
    int x;
    int y;
};

int main(void)
{
    struct Coordonnees c1;
    c1.x = 1;
    c1.y = 2;

    struct Coordonnees *c2 = &c1;
    c2->x = 3;
    c2->y = 4;
}

```

```

    struct Coordonnees *c3 = malloc(sizeof(struct Coordonnees));
    c3->x = 5;
    c3->y = 6;
    free(c3);

    printf("Coordonnees: %d %d\n", c1.x, c1.y);
    // Coordonnees: 3 4
}

```

Pointeurs & Fonctions

Lorsqu'on passe un pointeur en paramètre, on passe alors la valeur par **reference**. La valeur passée ne sera donc pas copiée et pas dupliquée. La valeur sous-jacente sera donc la même dans la fonction appelée que dans la fonction appelante.

Pour les tableaux, on est obligé de les passer par référence. Pour les structures, on a le choix, mais les passer par référence augmentera les performances du programme car il n'y aura pas de copie de toutes les données dans la structure.

Structures

```

void additionner_coordonnees(struct Coordonnees *c1, struct Coordonnees c2)
{
    c1->x = c1->x + c2.x;
    c1->y = c1->y + c2.y;
}

int main(void)
{
    struct Coordonnees c1 = {1, 2};
    struct Coordonnees c2 = {3, 4};

    additionner_coordonnees(&c1, c2);

    printf("Coordonnees: %d %d\n", c1.x, c1.y);
    // Coordonnees: 4 6
}

```

Tableaux

```

void additionner_tableaux(int *a, int *b, int size)
{
    for (int i = 0; i < size; i++)
    {
        a[i] = a[i] + b[i];
    }
}

```

```

}

int main(void)
{
    int a[] = {1, 2, 3};
    int b[] = {4, 5, 6};

    additionner_tableaux(a, b, 3);

    printf("a: %d %d %d\n", a[0], a[1], a[2]);
    // a: 5 7 9
}

```

Structures de données avancées

Listes chaînées

Une liste chaînée en langage C est une structure de données composée de nœuds, où chaque nœud contient des données et une référence (ou un pointeur) vers le nœud suivant de la liste.

C'est une structure de données linéaire et dynamique, ce qui signifie qu'elle peut croître ou décroître en taille pendant l'exécution du programme.

Voici les composants principaux d'une liste chaînée en C :

1. **Nœud (Node)** : C'est l'unité de base d'une liste chaînée. Chaque nœud contient généralement deux éléments :
 - **Données (Data)** : La valeur ou l'ensemble de valeurs que le nœud est censé stocker.
 - **Pointeur vers le nœud suivant (Next)** : Un pointeur vers un autre nœud, permettant de lier les nœuds entre eux pour former la liste.
2. **Tête de liste (Head)** : Un pointeur vers le premier nœud de la liste. C'est le point d'entrée pour accéder à n'importe quel élément de la liste.
3. **Fin de liste (Tail)** : Dans certaines implémentations, un pointeur vers le dernier nœud de la liste peut être conservé pour faciliter l'ajout d'éléments à la fin de la liste. Ce n'est pas toujours nécessaire, surtout dans les listes simplement chaînées.
4. **Taille de la liste**: Renseigne sur le nombre d'éléments totaux.

```

struct Node {
    int data;           // Les données stockées dans le nœud
    struct Node* next; // Pointeur vers le nœud suivant
};

```

C


```
struct LinkedList {
    unsigned int size;    // Taille de la liste
    struct Node* first;   // Pointeur vers le premier nœud
    struct Node* last;    // Pointeur vers le dernier nœud
};
```

Afin de manipuler ces listes, il est généralement nécessaire d'avoir des fonctions utilitaires pour les manipuler, avec des actions telles que:

- Parcourir la liste
- Ajouter un element à la liste
- Supprimer un element à la liste

Les arbres

Un arbre est une structure de données hiérarchique qui consiste en un ensemble de nœuds connectés de manière à simuler une hiérarchie arborescente. Chaque nœud de l'arbre contient des données et des pointeurs vers d'autres nœuds, appelés enfants.

La caractéristique principale d'un arbre est qu'il ne peut pas contenir de cycles, c'est-à-dire qu'un nœud ne peut pas être son propre ancêtre.

Voici les composants clés d'un arbre :

1. **Nœud racine (Root Node)** : Le nœud au sommet de l'arbre. Il ne possède pas de parent.
2. **Nœud interne (Internal Node)** : Un nœud qui a au moins un enfant.
3. **Feuille (Leaf)** : Un nœud sans enfants.
4. **Branche (Edge)** : La connexion entre deux nœuds, représentant la relation parent-enfant.
5. **Niveau (Level)** : La distance entre un nœud et la racine, où la racine est au niveau 0.
6. **Hauteur (Height)** : La distance entre le nœud le plus éloigné (la feuille la plus éloignée) et la racine.

```
struct TreeNode {
    int data;                // Les données stockées dans le nœud
    struct TreeNode* left;   // Pointeur vers le fils gauche
    struct TreeNode* right;  // Pointeur vers le fils droit
};
```

Les graphes

Un graphe est une structure de données abstraite utilisée pour modéliser un ensemble d'objets où certains paires d'objets sont connectés par des liens. Les objets sont

représentés par des entités appelées **sommets** (ou **nœuds/node**), et les liens qui les connectent sont appelés **arêtes** (ou **bords/edge**).

Les graphes sont utilisés pour représenter des réseaux de relations et de cheminements, comme des réseaux sociaux, des cartes routières, des réseaux de télécommunications, et bien d'autres systèmes dans divers domaines.

Les composants principaux d'un graphe sont :

1. **Sommet (Vertex)** : Un élément fondamental qui peut contenir des données et représente un point unique dans le graphe.
2. **Arête (Edge)** : Une connexion entre deux sommets qui peut être orientée (dans le cas d'un graphe orienté) ou non-orientée (dans le cas d'un graphe non orienté).
3. **Poids (Weight)** : Dans les graphes pondérés, chaque arête a une valeur appelée poids, qui représente généralement le coût ou la distance entre deux sommets.

```
// Structure pour représenter un noeud dans un graphe
struct Node {
    char* name;           // Nom du noeud
    struct Node **edges;  // Liste des noeuds connectés
};

// Structure pour représenter un graphe
struct Graph {
    unsigned int size;    // taille
    Node* nodes;          // Tableau de noeuds
};
```

Aller plus loin

Definition de type

On peut définir des types de données en utilisant le mot clef `typedef`.

```
typedef unsigned int ui;
unsigned int a = 10; // ==
ui a = 10;

typedef struct TreeNode {
    int data;           // Les données stockées dans le nœud
    struct TreeNode* left; // Pointeur vers le fils gauche
    struct TreeNode* right; // Pointeur vers le fils droit
} TreeNode;
```

```
struct TreeNode *root; // ==  
TreeNode *root;
```

Organisation du code

Lorsqu'on commence à avoir beaucoup de fonctions, tout mettre dans un seul fichier peut rendre difficile la lecture du code.

On va alors diviser notre code en plusieurs fichiers

Pour que des fonctions soient accessibles entre les fichiers, il faut définir des headers, qui définissent le prototype des fonctions et structures de données.

Ces headers devront être inclus dans chacun des fichiers qui utilisera ces fonctions.

```
// lib.h  
#ifndef LIB_H  
#define LIB_H  
  
int calculer(int);  
  
#endif
```

```
// lib.c  
#include "lib.h"  
  
int calculer(int a) {  
    return a * 2;  
}
```

```
// main.c  
#include <stdio.h>  
#include <stdlib.h>  
#include "lib.h"  
  
int main(void)  
{  
    int resultat = calculer(2);  
  
    return 0;  
}
```

Compilation

Pour compiler un programme qui est séparé en plusieurs fichiers, on indique à gcc tous les fichiers .c à compiler. Exemple:

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

Exemple

Exemple de template pour faire des exercices dans des fichiers séparés:

<https://replit.com/@pacoccino/Exemple-exercices-C>

Systeme de link

Quand on sépare le code en plusieurs fichiers, chaque fichier de code générera un objet (.o): c'est une représentation du code compilé.

Le compilateur, dans l'étape finale, regroupera tous ces objets pour en faire un exécutable.

Chaque objet embarquera une liste de fonctions utilisés ici et là dans les autres objets, et un seul embarquera la fonction `main()` de départ.

Préprocesseur

Les commandes préprocesseur sont des morceaux de code qui sont interprétés avant la compilation. Elles sont toujours précédées d'un `#`

Le contenu du fichier sera alors modifié en conséquence.

Instructions disponibles:

- `#include <library.h>` : inclut un fichier (header) dans le code
 - on écrit `<stdio.h>` pour une librairie externe
 - on écrit `"mylib.h"` pour un fichier local (chemin relatif au fichier)
- `#define PI 3.14` : déclare une constante. A Chaque utilisation de cette constante, sa valeur sera directement remplacée dans le code par le préprocesseur

```
#define PI 3.14
```

```
int i = PI * 2;
```

```
// Après le preprocesseur, le code deviendra
```

```
int i = 3.14 * 2;
```

- `#ifdef` : si une constante est déclarée, va avec `#ifndef` , `#endif` alors le code sera intégré au fichier, sinon il sera effacé
- Cela permet de ne pas déclarer plusieurs fois la même chose. Si on inclut un header à plusieurs endroits, sans les `#ifdef` , les déclarations qui se trouvent à l'intérieur se feront plusieurs fois et la compilation échouera

```
/* On utilise des directives pour compiler
différemment des codes sous Windows et sous Linux */
#ifdef _WIN32
    printf ("Code pour Windows\n");
#else
    printf ("Code pour Linux\n");
#endif
```

Makefile

Make est un outil aidant à la compilation de programmes avec de nombreux fichiers

Créer un fichier `Makefile` à la racine du projet

```
# Compiler
CC = gcc

# Compiler flags
CFLAGS = -Wall -Wextra -g

# Source files
# >> Inclure ici la liste de vos fichiers à compiler <<
SRCS = main.c linked_list.c graph.c

# Object files
OBS = $(SRCS:.c=.o)

# Executable
TARGET = main

# Default target
all: $(TARGET)

# Compile source files into object files
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Link object files into executable
$(TARGET): $(OBS)
    $(CC) $(CFLAGS) $^ -o $@
```

```
# Clean up object files and executable
clean:
    rm -f $(OBJS) $(TARGET)
```

Pour compiler notre programme, il faut lancer la commande `make`

Buffer Overflow

Les buffer overflows sont parmi les vulnérabilités les plus courantes et dangereuses dans les programmes C. Ils se produisent lorsqu'un programme écrit plus de données dans un tampon (buffer) qu'il ne peut en contenir, ce qui provoque l'écrasement des données adjacentes en mémoire.

Ils peuvent amener à

- Des plantages de programmes
- Des fuites d'informations sensibles
- L'exécution de code arbitraire par un attaquant

Rôle d'un expert en sécurité

- Trouver ces vulnérabilités dans des logiciels et les résoudre
- Empêcher ces vulnérabilités dans les nouveaux codes

Exemple de code vulnérable:

```
void vulnerable_function(char *str) {
    char buffer[10];
    int i = 0;
    strcpy(buffer, str);
}
```

Si la chaîne copiée depuis `*str` est plus longue que `buffer`, le contenu de `i` sera écrasé.

Protection

Il ne faut jamais supposer que les données entrantes sont sûres, et toujours imposer des limites sur la taille des données entrantes.

```
void safe_function(char *str) {
    char buffer[10];
    strncpy(buffer, str, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
}
```

Un attaquant pourra également modifier la mémoire pour changer le code du programme et modifier son comportement.

Ressources à lire:

- <https://retr0.blog/blog/llama-rpc-rce>
- https://www.theregister.com/2025/02/13/fbi_cisa_unforgivable_buffer_overflow/
- <https://overthewire.org/wargames/behemoth/>

Cheat Sheet - Langage C avancé

Tableaux

Les tableaux sont des variables qui contiennent plusieurs elements.

Le premier index est 0.

```
int tab[10];           // Declaration d'un tableau de taille 10
printf("%d", tab[0]); // Affichage du premier element du tableau
tab[9] = 5;           // Affectation du dernier element du tableau

int tab[] = { 2, 5, 7 }; // Affectation avec taille implicite lors d

int first_element(int tab[]) {
    return tab[0];
}

int tab[2][2] = { { 1, 2 }, { 3, 4 } }; // Tableau bidimensionnel (2
```

Chaines de caractères

- Les chaines de caractères (**string**) sont des suites de caractères.
- Représenté par des tableau de `char`
- Ecrit par des doubles apostrophes `"`
- Doivent finir par le symbole `\0`
- Le code pour `printf` est `%s`
- On ne met pas de `&` pour les `scanf` de chaines de caractère

C

```
char string[255] = "Bonjour";
string[0] = 'C';
printf("%s", string);
scanf("%s", string);
```

Structures

C

```
struct Coordonnees
{
    int x;
    int y;
};

struct Coordonnees point = { 10, 20 };
point.x = 15;

struct Coordonnees *p = &point;
p->x = 20;
```

Pointeurs

C

```
void increment(int *p) {
    *p += 1;
}

int a = 10;

int *b = &a;
printf("%d", *b); // 10

*b = 20;
```

```
printf("%d", a); // 20

int *p = malloc(sizeof(int)); // Allocation dynamique
*p = 10;

increment(p);
printf("%d", *p); // 11

free(p); // Liberation de la memoire allouée
```

Exercices - Langage C Avancé

Si vous souhaitez revoir rapidement les bases, vous pouvez faire les [exercices préparatoires](#).

Tableaux

Exercice 1.1

- Ecrire une fonction qui calcule la somme de tous les éléments d'un tableau de nombres
- Ecrire une fonction qui calcule la moyenne d'un tableau de nombres

Exercice 1.2

- Ecrire une fonction qui prend en paramètre un tableau et qui affiche tous les nombres pairs dans ce tableau
- Ecrire une fonction qui prend en paramètre un tableau et qui retourne le nombre le plus petit de ce tableau

Exercice 1.3

Tableaux multidimensionnels

Ecrire une fonction qui affiche un tableau bi-dimensionnel à l'écran

Bonus: dessiner quelque chose de joli !

Exercice 1.4

Chaines de caractères

Ecrire une fonction qui demande un mot de passe et retourne 1 ou 0 si le mot de passe est correct ou non

L'utiliser au debut de votre programme et le stopper si le mot de passe est incorrect

Exercice 1.5

Chaines de caractères et tableaux

Ecrire une fonction qui recoit une chaine de caractères et qui retourne 1 ou 0 si la chaine est un palindrome ou non

Exercice 1.6

- Créer une fonction qui compte combien d'espaces sont présents dans une chaine de caractère
 - Recréer les fonctions `strlen` , `strcmp`
-

Structures

Exercice 2.1

Ecrire un programme qui définit :

- une structure de qui représente une couleur en RGB
- une fonction qui retourne la luminosité d'une couleur (moyenne des valeurs RGB)
- Declarer une variable de couleur et calculer sa luminosité

Exercice 2.2

Ecrire un programme qui définit :

- Une structure Horaire qui contient des heures et des minutes
- Une fonction qui additionne deux horaires

Attention à ce que les minutes ne dépassent pas 60

- Une fonction qui affiche un Horaire
 - Utiliser ces fonctions
 - **Bonus:** Une fonction qui convertit un timestamp (secondes) en Horaire (ajouter les secondes)
-

Pointeurs

Exercice 3.1

Ecrire une fonction qui multiplie une variable par référence

Exercice 3.2

Ecrire une fonction qui inverse les données de deux variables avec l'utilisation des pointeurs

Si $a = 1$ et $b = 2$ alors apres l'appel de la fonction $a = 2$ et $b = 1$

Exercice 3.3

Recreer la fonction `strcpy`

Allocation dynamique

Exercice 4.1

Allocations mémoire

- Ecrire un programme qui alloue de la mémoire pour un tableau de 10 flottants
- Ecrire une fonction qui remplit les valeurs d'un tableau avec les premiers multiples de 3
- Utiliser cette fonction et afficher le tableau

Exercice 4.2

Allocations mémoire et tableaux

Ecrire un programme qui:

- Demande à l'utilisateur une taille
- Alloue de la mémoire pour un tableau d'entiers de la taille demandée
- Pour chaque élément du tableau, demander sa valeur à l'utilisateur et la stocker dans le tableau
- Afficher tous les éléments du tableau

Exercice 4.3

Structures et pointeurs

Ecrire un programme qui gère le niveau d'un joueur

- Créer une structure Player qui contient le niveau et le nombre de points d'expérience d'un joueur
- Créer une fonction qui:
 - Ajoute de l'expérience à un joueur
 - Si l'expérience atteint un seuil, augmenter le niveau, et mettre à jour l'exp

Buffer Overflow

Exercice 5.1

Ecrire un programme qui est protégé par un mot de passe. Si le mauvais mot de passe est utilisé le programme doit s'arrêter, sinon le reste du programme peut s'exécuter (afficher un message de succès)

Rendre le logiciel vulnérable à un buffer overflow, et exploiter le programme pour qu'il s'exécute même quand le mot de passe est incorrect.

Corriger le programme en ajoutant des protections.

Rendre les deux versions du programme dans des fichiers différents (vulnerable et protégé) et indiquer comment exploiter le programme vulnérable.

Exercice 5.2

Ecrire un programme qui appelle des fonctions par pointeurs. Le rendre vulnérable à un buffer overflow et l'exploiter pour changer son comportement.

Structures avancées

Exercice 6.1 - Liste chaînées

Créer un programme qui implémente une liste chaînées d'entiers

Le programme doit contenir les fonctions suivantes:

- `struct chained_list creerListe(int* item, unsigned int size)`
- `void parcourirListe(struct chained_list *liste)`
- `void ajouterItem(struct chained_list *liste, int item)`
- `void supprimerItem(struct chained_list *liste, unsigned int index)`

Aide: voir [Listes chaînées](#)

Exercice 6.2 - Graphes

Ecrire un programme qui implémente un système de graphes

- Le graph doit pouvoir avoir un nombre arbitraire d'éléments
- On doit pouvoir ajouter des éléments au graph
- On doit pouvoir parcourir le graph en entier en suivant les connexions

Aide: voir [Les graphes](#)

Divers

Separation du code

Ecrire un programme séparé en plusieurs fichiers

Dans un nouveau dossier:

- Créer un fichier `main.c` qui contiendra une fonction `main`
- Créer des fichier `lib.c` et `lib.h` qui contiendront une fonction utilitaire de votre choix (estPair, estPremier, palindrome, ...)
- Appeler cette fonction utilitaire depuis le main
- Compiler et executer le programme
 - Inclure la liste des fichier à compiler à gcc (`gcc main.c lib.c ...`)

Aide: voir [Organisation du code](#)

Tri

- Réaliser un algorithme de tri
 - Trier un tableau d'entiers
 - Comparer différentes strategies de tri et leurs performances

Mini projets

Carte de crédit

Les numéro de crédits sont assez long pour qu'il y en aie assez pour tout le monde, mais permettent facilement de savoir si ils sont correct ou pas. Si on fait une erreur en tapant un numero de carte bleue en ligne, le site nous dira rapidement si il y a une faute de frappe. Cela permet de se tromper et d'utiliser la carte de quelqu'un d'autre.

Alors, quelle est la formule secrète ? Eh bien, la plupart des cartes utilisent un algorithme inventé par Hans Peter Luhn d'IBM. Selon l'algorithme de Luhn, vous pouvez déterminer si un numéro de carte de crédit est (syntaxiquement) valide comme suit :

1. Multipliez chaque autre chiffre par 2, en commençant par l'avant-dernier chiffre du numéro, puis additionnez les chiffres des produits obtenus.
2. Ajoutez cette somme à la somme des chiffres qui n'ont pas été multipliés par 2.
3. Si le dernier chiffre du total est 0 (ou, plus formellement, si le total modulo 10 est congruent à 0), le numéro est valide !

Essayons un exemple avec la Visa de David : 4003600000000014.

Pour la discussion, soulignons d'abord chaque autre chiffre, en commençant par l'avant-dernier chiffre du numéro :

4003600000000014

D'accord, multiplions chacun des chiffres soulignés par 2 :

$$1 \cdot 2 + 0 \cdot 2 + 0 \cdot 2 + 0 \cdot 2 + 0 \cdot 2 + 6 \cdot 2 + 0 \cdot 2 + 4 \cdot 2$$

Cela nous donne :

$$2 + 0 + 0 + 0 + 0 + 12 + 0 + 8$$

Maintenant, additionnons les chiffres des produits (c'est-à-dire, pas les produits eux-mêmes) ensemble :

$$2 + 0 + 0 + 0 + 0 + 1 + 2 + 0 + 8 = 13$$

Ajoutons maintenant cette somme (13) à la somme des chiffres qui n'ont pas été multipliés par 2 (en commençant par la fin) :

$$13 + 4 + 0 + 0 + 0 + 0 + 0 + 3 + 0 = 20$$

Oui, le dernier chiffre de cette somme (20) est un 0, donc la carte de David est légitime !

Ainsi, valider des numéros de carte de crédit n'est pas difficile, mais cela devient un peu fastidieux à faire à la main.

Ecrire un programme en C qui permet de valider un numéro de carte de crédit.

Scrabble

Dans le jeu de Scrabble, les joueurs créent des mots pour marquer des points, et le nombre de points est la somme des valeurs en points de chaque lettre du mot.

A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	3	3	2	1	4	2	4	1	8	5	1	3	1

Par exemple, si nous voulons calculer le score du mot "CODE", nous noterons que le 'C' vaut 3 points, le 'O' vaut 1 point, le 'D' vaut 2 points et le 'E' vaut 1 point. En additionnant ces valeurs, nous obtenons que "CODE" vaut 7 points.

Implémentez un programme en C qui détermine le gagnant d'un court jeu de type Scrabble. Votre programme doit demander une entrée deux fois : une fois pour que le "Joueur 1" saisisse son mot et une fois pour que le "Joueur 2" saisisse son mot. Ensuite, en fonction du joueur qui marque le plus de points, votre programme doit afficher soit quel joueur gagne ou si il y a égalité.

Autres idées

- Jeu du pendu
- TODO list
- TOP1:
 - Poser des questions "tu preferes ... ?" (plat préféré, ville, film, ...)
 - Parmi les preferences demander "tu preferes ... ou ... ?"
 - Dire ce qui est préféré

Shaders

Qu'est-ce qu'un shader ?

Un pixel shader, également connu sous le nom de fragment shader, est un type de programme utilisé dans les systèmes de rendu graphique pour déterminer les propriétés visuelles de chaque pixel à l'écran. Il s'agit d'un composant clé du pipeline de rendu 3D moderne, qui fonctionne au sein des unités de traitement graphique (GPU).

Voici quelques points clés sur les pixel shaders :

1. **Personnalisation de l'image:** Les pixel shaders permettent aux développeurs de définir comment les pixels doivent être colorés et traités sur la base de divers facteurs, tels que l'éclairage, la couleur de la texture, et les données de profondeur.
2. **Programmabilité:** Contrairement aux anciennes méthodes de rendu, les pixel shaders sont programmables, ce qui signifie que les développeurs peuvent écrire des shaders dans des langages de haut niveau comme HLSL (High-Level Shader Language) pour DirectX ou GLSL (OpenGL Shading Language) pour OpenGL.
3. **Effets visuels:** Ils sont utilisés pour créer des effets visuels complexes tels que les reflets, les ombres douces, l'anticrénelage, le brouillard, les effets de lumière volumétrique, la transparence, et bien plus encore.
4. **Exécution parallèle:** Les shaders sont exécutés en parallèle sur le GPU, ce qui permet de traiter efficacement de nombreux pixels simultanément, offrant ainsi des performances élevées pour les graphiques en temps réel.
5. **Interaction avec d'autres shaders:** Les pixel shaders travaillent souvent en tandem avec d'autres types de shaders, comme les vertex shaders, qui déterminent la position et d'autres attributs des sommets dans une scène 3D, et les geometry shaders, qui peuvent générer ou modifier des géométries.

6. **Version et compatibilité:** Les capacités des pixel shaders ont évolué au fil des générations de matériel graphique, avec des versions plus récentes supportant des fonctionnalités plus avancées. Les développeurs doivent souvent tenir compte des capacités du matériel cible lors de l'écriture de shaders.

7. **Rendu non graphique:** Bien que leur fonction principale soit le rendu graphique, les pixel shaders peuvent également être détournés pour effectuer des calculs généraux (GPGPU), tels que ceux utilisés dans le traitement d'images ou la simulation physique.

En résumé, les pixel shaders sont essentiels pour la création d'images de synthèse réalistes et attrayantes dans les jeux vidéo, les simulations, et d'autres applications graphiques interactives. Ils offrent aux développeurs la flexibilité nécessaire pour créer une grande variété d'effets visuels et améliorer l'immersion visuelle dans les environnements numériques.

Les moteurs d'inférence en **intelligence artificielle** n'utilisent pas les shaders pour leur calculs, mais utilisent la même infrastructure matérielle via des bibliothèques comme **CUDA** afin de paralléliser les calculs de matrices et de vecteurs des réseaux neuronaux.

Librairies et compatibilité

Les programmes de calcul graphique existent sous différentes formes, différentes syntaxes et différentes méthodes d'exécution.

Ils diffèrent par leur syntaxe et par les bibliothèques et matériel supporté. Les programmes codés avec DirectX pour NVIDIA ne seront pas compatibles avec les cartes AMD ou Linux par exemple. Avec OpenGL, on sera compatible sur les cartes AMD et NVIDIA, sous windows et linux, mais pas sous MacOS...

Il en est de même pour les shaders.

WebGL

Les moteurs web des navigateurs intègrent maintenant **WebGL**, qui permet, avec une syntaxe unifiée, de créer des programmes graphiques compatibles sous toutes les plateformes qui possèdent les capacités nécessaires.

Cela veut dire qu'avec un seul code, on pourra afficher des graphismes sous Windows, Mac, Linux, iOS, Android tant qu'il y a suffisamment de puissance de calcul.

WebGL supporte l'utilisation des shaders.

ShaderToy

ShaderToy est un site web qui permet de créer, tester, et exposer des shaders avec WebGL

<https://www.shadertoy.com/>

On peut voir les shaders réalisés par les membres du site, le code qui y est associé, et on peut les modifier pour en créer de nouveaux.

Quand on modifie le code, on peut cliquer sur le bouton "play" en bas de l'éditeur de code pour compiler et exécuter notre code.

Exemples

Voici quelques exemples:

<https://www.shadertoy.com/view/XcXXzS>

<https://www.shadertoy.com/view/McsSRB>

<https://www.shadertoy.com/view/mtyGWy>

<https://www.shadertoy.com/view/ddcGW8>

<https://www.shadertoy.com/view/XsX3z8>

Comment fonctionne un fragment shader de base

Un fragment shader est donc un petit bout de programme qui va être exécuté, en parallèle, par la carte graphique, pour chacun des pixels de l'espace de rendu.

Son role va être de déterminer la couleur du pixel auquel il est attribué.

Données en entrée

- La coordonnée du pixel attribué
- La résolution de l'espace de rendu
- Différentes variables de contexte:
 - Textures
 - Temps écoulé
 - Données personnalisées injecté par le programme...

Données en sortie

- Couleur du pixel

Une fois que tous les fragment shader sont exécutés, le programme connaît la couleur de tous les pixels, et ils sont alors affichés à l'écran.

Informations utiles

Couleurs

Une couleur est représentée par 3 valeurs: **Rouge, Vert, Bleu**

Vecteurs

Un vecteur est une liste de valeurs du même type.

Ils sont utilisés pour stocker des données multidimensionnelles, telles que:

- des **coordonnées** (x, y → 2 dimensions)
- des **couleurs** (r, g, b → 3 dimensions)

En C, c'est représenté par un tableau:

```
int vecteur2Dimension[2] = { 1, 2 }
```

C

```
int vecteur3Dimension[3] = { 1, 2, 3 }
```

Spécificités de langage



Veillez noter qu'à partir d'ici, nous aborderons la syntaxe et les outils disponibles dans ShaderToy. Celle ci sera différente dans d'autres environnements.

CheatSheet de la syntaxe utilisée:

<https://gist.github.com/markknol/d06c0167c75ab5c6720fe9083e4319e1>

ShaderToy est basé sur la technologie **OpenGL GLSL ES**

Le langage de shader que nous allons voir est un *subset* du langage C.

Si vous connaissez le C, vous connaîtrez déjà ce langage. Seulement certaines choses ont été enlevées pour des raisons d'optimisations. D'autres choses ont été ajoutées pour aider au developpement.

Vous devrez en apprendre les spécificités pour s'en servir.

Pointeurs

On ne peut pas utiliser les pointeurs et references dans les shaders ! C'est complètement désactivé.

Valeurs et types

Couleurs

Les couleurs seront représentées par des variables de type `float`, pour des valeurs allant de 0 à 1.


```

float r = 0.5;
float g = 0.; // attention, ne pas oublier le '.' pour signifier que
float b = 0.5;
// -> Magenta

{ 0., 0., 0. } // -> noir
{ 1., 1., 1. } // -> blanc

```

Coordonnées

Les coordonnées représentent la position d'un pixel à l'écran, et sont également représentées par des `float`. Valeurs entre 0 et la largeur/longueur de l'espace de travail.

Structures de données

Afin de faciliter le développement, différentes structure de données pré-existantes ont été définie, avec des fonctions utilitaires qui vont avec.

Vecteurs

Différentes structures ont été définies pour différentes tailles de vecteurs:

`vec2`, `vec3`, `vec4`, ... de type `float`

On accede aux valeurs contenues par `.x .y .z` ou `.r .g .b .a`, ces deux ensembles contiennent les mêmes valeurs et sont juste des raccourcis.

Des fonctions et opérations sont disponible et similaires pour chacun d'eux.

```

struct vec3 {
    float x;
    float y;
    float z;
    float r; // = x
    float g; // = y

```

```

    float b;    // = z
}

vec3 vec3(float x, float y, float z); // Créer un vecteur
vec3 rgb = vec3(0.5, 0.5, 0.);

vec3 vec4(vec3, float w);    // Créer un vecteur de dimensions plus
vec4 rgba = vec4(rgb, 1.);

vec3 vec4(float xyz);    // Créer un vecteur avec toutes les dimensi
vec3 r = vec3(1.);

// Opérations

rgb = rgb * 2.; // Multiplication de toutes les valeurs
// Equivalent à
rgb.r = rgb.x * 2.;
rgb.g = rgb.y * 2.;
rgb.b = rgb.z * 2.;

```

Remarque: la 4^e composante sur vec4 est nommée .w ou .a

Variables disponibles

- `vec2 fragCoord`
 - Coordonnées du pixel en cours de traitement (x, y)
- `vec3 iResolution`
 - Résolution de l'espace de travail (x, y)
 - z représente l'aspect ratio
- `float iTime`
 - Temps actuel en millisecondes
- `vec4 iMouse`
 - Coordonnées de la souris en x et y
 - z,w → clicked

- `sampler2D iChannel0..3`
 - Textures fournies par l'utilisateur

Variables à retourner

Le seul objectif indispensable est de définir la couleur de notre pixel, qui est stockée dans la variable `vec4 fragColor`

Celle ci est un vecteur de 4 dimensions, pour R,G,B,A

A représente l'alpha et n'est pas utilisé, il est conseillé de le définir à 1.

```
fragColor = vec4(0.5, 0., 0.5, 1.);
```

C

Fonctions

- `float length(vecX)` : Retourne la norme d'un vecteur (pythagore)
- `float cos(float)` : retourne le cosinus d'un angle
- `float mod(float, float)` : modulo avec nombre decimaux
- `float min/max(float, float)` : minimum ou maximum entre deux nombres
- `float clamp(float x, float min, float max)` : retourne x bornée entre min et max
- `float smoothStep(float min, float max, float x)` : "normalise" x entre 0 et 1 selon sa position entre min et max

Exemple

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )  
{  
    vec3 color = vec3(1., 0., 0.); // rouge
```

C

```
// retourne la valeur
fragColor = vec4(color,1.0);
}
// Toute l'image sera rouge
```

Exercices d'entraînement

Gradient de rouge

Ecrire un shader qui affiche un gradient entre le noir et le rouge de gauche à droite

- Correction

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Coordonnées normalisées (entre 0 et 1)
    vec2 uv = fragCoord/iResolution.xy;

    vec3 color = vec3(uv.x, 0., 0.);

    // Definition de la couleur de sortie
    fragColor = vec4(color,1.0);
}
```

Transition rouge/vert

Ecrire un shader qui affiche une couleur unie qui alterne lentement entre le rouge et le vert

- Correction

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Coordonnées normalisées (entre 0 et 1)
```

```
vec2 uv = fragCoord/iResolution.xy;

vec3 color = vec3(uv.x, 0., 0.);

// Definition de la couleur de sortie
fragColor = vec4(color,1.0);
}
```

Gradient circulaire

Ecrire un shader qui affiche un gradient circulaire en partant du milieu

Indice: `length`

- Correction

Avec la webcam:

Inverser les couleurs

Inverser l'image

Déplacer l'image avec repetition sur les bords (indice: `cos(iTime)`)

Liens utiles

<https://www.shadertoy.com/>

<https://graphtoy.com/>

[https://www.khronos.org/opengl/wiki/Data_Type_\(GLSL\)#Basic_types](https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)#Basic_types)

<https://thebookofshaders.com/>

<https://www.youtube.com/watch?v=f4s1h2YETNY>

<https://shadeup.dev/>

Exercices préparatoires - Langage C Avancé

Exercice 1

Ecrire un programme qui:

- Demande un nombre à l'utilisateur
 - Via une fonction, affiche la table de multiplication d'un nombre donné en paramètre
 - Affiche la table de multiplication de tous les nombres de 1 à 10
 - Affiche si le nombre est pair ou impair
-

Conversion python

Convertir les programmes python suivants en C:

Condition

```
x = 10
if x > 0:
    print("x est positif")
else:
    print("x est négatif")
```

python

Boucle

```
x = 10
for i in range(x):
    print(i)
```

python

Fonctions

```
def sum(a, b):  
    return a + b  
  
print(sum(1, 2))
```

python

Factorielle

```
def factorielle(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorielle(n - 1)  
  
print(factorielle(5))
```

python