Chapitre 11 : Adresses mémoires et pointeurs – Passage de paramètres « par adresses »

I. Rappels : Stockage des données en mémoire vive (RAM).

- Le rôle d'un ordinateur et l'objectif des programmes informatiques est d'acquérir, mémoriser, traiter et transformer des données, pour en extraire et en restituer des informations.

- Tout programme manipule des données.
- **Toute donnée manipulée par un programme en cours d'exécution doit être stockée en mémoire vive (RAM).**
- Toutes les données y sont stockées et manipulées sous forme de signaux électriques.
  - La donnée brute la plus élémentaire en informatique est la présence ou non de courant électrique. Elle est représentée par un bit (binary digit) qui prend la valeur 0 (absence de courant) ou 1 (présence de courant).
  - Les données sont donc codées selon un alphabet binaire (0 ou 1).

- **La RAM peut être vue comme une suite de cases numérotées.** Toutes les cases font la même taille et contiennent exactement 8 bits, c'est-à-dire 1 **octet.**

- Chaque case mémoire possède un numéro unique qui l'identifie complètement et qu'on appelle son **adresse mémoire.** La numérotation commence à 0 pour la première case.

RAM

(Cases mémoires)	Adresses
01001110	N
11001011	N-1
…	
10110011	1
01100001	0

II. Notation hexadécimale.

- **Une adresse est un numéro de case mémoire.**

- Les adresses s'écrivent en hexadécimal (base 16).

- Voici les nombres de 0 à 31 écrits en base 16 :
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F …

  → 10 en base 16 s'écrit A
  → 16 en base 16 s'écrit 10
  → 17 en base 16 s'écrit 11 …

- En programmation quand un nombre est écrit en base 16 (hexadécimal), on le précède de ***0x***.

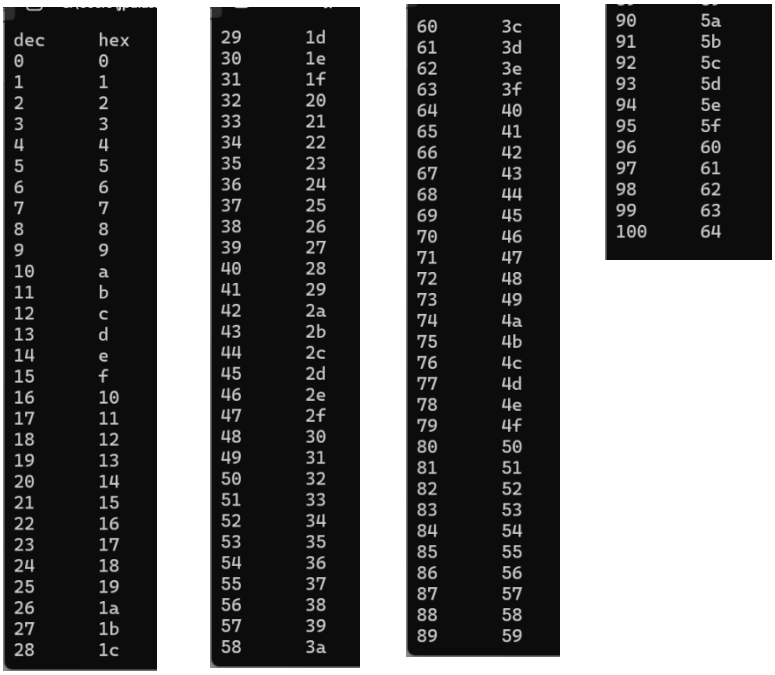
  → 0x2A5 est l'écriture en hexa de 677.

- En C, on peut afficher des adresses au format hexadécimal avec ***printf*** en utilisant le format ***%x***.

Exemple : programme qui écrit les entiers de 0 à 100 en notations décimales et hexadécimales :

```c
#include <stdio.h>
int main(){
    int i;
    printf("dec\thex\n");
    for(i=0;i<=100;i++){
        printf("%d\t%x\n",i,i);
    }
    return 0;
}
```

Code



dec	hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1a
27	1b
28	1c
29	1d
30	1e
31	1f
32	20
33	21
34	22
35	23
36	24
37	25
38	26
39	27
40	28
41	29
42	2a
43	2b
44	2c
45	2d
46	2e
47	2f
48	30
49	31
50	32
51	33
52	34
53	35
54	36
55	37
56	38
57	39
58	3a
60	3c
61	3d
62	3e
63	3f
64	40
65	41
66	42
67	43
68	44
69	45
70	46
71	47
72	48
73	49
74	4a
75	4b
76	4c
77	4d
78	4e
79	4f
80	50
81	51
82	52
83	53
84	54
85	55
86	56
87	57
88	58
89	59
90	5a
91	5b
92	5c
93	5d
94	5e
95	5f
96	60
97	61
98	62
99	63
100	64

III. Les emplacements mémoire, les variables et leurs adresses

1. Réservation d'emplacement mémoire

Pour qu'un programme puisse mémoriser une donnée, il faut la stocker dans un **emplacement mémoire**. En C pour réserver de emplacements mémoire on peut :

- déclarer des **variables** (→ chapitre 3)
- faire de l'allocation dynamique (→ chapitre 16)

Quand on réserve un emplacement mémoire, il faut indiquer le type de la donnée qui y sera stockée. C'est ce qui détermine la taille de l'emplacement (nombre d'octets) et la manière dont la donnée sera codée/décodée en binaire.

2. Les variables et leurs adresses

Rappels :

- L'adresse d'une variable est l'adresse du début de l'emplacement qui lui a été réservé.
- Pour récupérer l'adresse d'une variable en C, on utilise l'**opérateur d'adressage &**.

Exercice 22. Compléments sur les types et les variables. Adresse d'une variable. Ecriture héxadécimale.

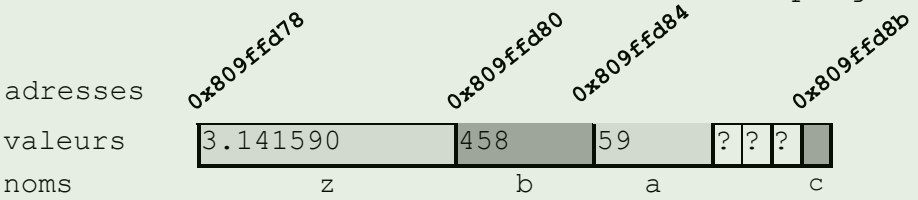
Editez, étudiez et exécutez le programme « ex22_adresses_hexa.c ». Documentez-vous pour répondre aux questions suivantes :

- Combien d'octets occupe la variable c ?
- Que font les lignes 8,9 et 10 ?
- A quoi correspond le mot clé ***sizeof*** ? Est-ce une fonction ou un opérateur ? A quoi sert-il ?
- De quel type est le résultat de l'instruction ***sizeof(int)*** ?
- A quoi sert le format ***%zu*** ?
- Que fait la suite d'instructions 17 à 19 ?

Lors d'une exécution, le programme a donné l'affichage suivant :

```
taille des types :
char : 1
int : 4
double : 8


variables :

adresses        valeurs
c : 809ffd8b    4

a : 809ffd84    59

b : 809ffd80    458

z : 809ffd78    3.141590


variables :

adresses        valeurs
a : 809ffd84    458

b : 809ffd80    59
```

Le schéma ci-dessous représente la mémoire à un instant donné au cours de l'exécution du programme :



- Ce schéma représente-t-il la mémoire avant ou après l'exécution des lignes 17 à 19 ?
- De combien d'octets sont séparées les variables a et c ?
- Qu'en déduisez-vous sur la manière dont sont réserver l'emplacement des variables ?

3. Les schémas mémoire

L'adresse des emplacements mémoires utilisés change généralement à chaque exécution. Tout dépend de la mémoire disponible lorsqu'on lance le programme. C'est le système d'exploitation qui gère l'attribution des zones mémoire aux différents programmes exécutés.

Les schémas mémoire permettent de comprendre la manière dont les données sont organisées en mémoire. Ne sachant pas à l'avance les adresses qui seront utilisées, on utilise des adresses fictives « imaginaires ».

4. Durée de vie et visibilité des variables locales – Passage d'adresses en paramètres.

- Rappels :

  Les variables locales déclarées dans un sous-programme :
  - N'existent que pendant la durée d'exécution de ce sous-programme
  - Ne sont pas connues des autres sous-programmes. Un sous-programme ne connait pas les noms des variables déclarées dans un autre sous-programme. Il ne peut pas y accéder de manière directe.

Pour qu'un sous-programme puisse modifier les données stockées dans les variables d'un autre sous-programme, il faut lui transmettre en paramètres les adresses mémoire auxquelles se trouvent ces données.

Si un sous-programme *f1* appelle un sous-programme *f2*, *f2* ne connait pas les variables déclarées dans *f1*.

- Si *f2* doit utiliser les valeurs de certaines variables de *f1*, il faut les lui transmettre en paramètres.
- Si *f2* doit **modifier** les valeurs de certaines variables de *f1*, il faut lui **transmettre leurs adresses en paramètres.**

IV. Les pointeurs

1. Définitions

Un pointeur est une variable qui permet de stocker l'adresse d'un emplacement mémoire. On dit qu'il « pointe » sur un emplacement mémoire.

Pointeur : variable dont la valeur est une adresse.

Emplacement pointé : emplacement dont l'adresse est stockée dans un pointeur.

Donnée pointée : donnée contenue dans l'emplacement dont l'adresse est stockée dans un pointeur.

Variable pointée : variable dont l'adresse est contenue dans un pointeur.

2. Déclaration d'un pointeur

a. Taille d'un pointeur

Un pointeur contient une adresse. Une adresse est un numéro de case mémoire donc un entier. La plage de valeurs dépend du nombre maximal de cases mémoire « utilisables » (qui est fixée par l'architecture système).
Actuellement les adresses et donc les pointeurs occupent souvent 8 octets.

Activité 13. Vérifier la taille occupée par les pointeurs sur son ordinateurs.

Pour afficher la taille des pointeurs sur votre ordinateur, ajoutez l'instruction suivante à la fin du programme précédent :

```
printf("\n\ntaille des adresses (pointeurs) : %d %d %d\n",sizeof(&c),sizeof(&a),sizeof(&z));
```

b. Pointeurs génériques

Tous les pointeurs contiennent le même type de données : une adresse.
Tous les pointeurs font la même taille.

Si on ne se préoccupe pas du type de la donnée pointée, c'est-à-dire si on se contente de manipuler des adresses sans consulter ce que contiennent les emplacements pointés, on peut déclarer et utiliser des pointeurs génériques, c'est-à-dire des pointeurs sur « n'importe quoi ». Ils se déclarent avec le type ***void.*** Le nom du pointeur doit être précédé d'une ***.**

Exemples :

Déclaration d'un pointeur générique de nom p :
```
void *p;
```
Déclaration de deux pointeurs génériques de noms p1 et p2 :
```
void *p1,*p2;
```

Nous verrons que les pointeurs génériques sont indispensables pour la gestion dynamique de la mémoire (→ chapitre 16).

c. Pointeurs typés

Très souvent, comme dans le passage d'adresses aux sous-programmes, un pointeur sert à accéder de manière indirecte à l'emplacement mémoire d'une variable pour y lire ou en modifier le contenu. Or pour lire ou écrire une donnée dans un emplacement mémoire, il faut connaitre son type (pour savoir combien d'octets lire ou écrire, pour savoir comment décoder/coder en binaire).
Dans ce cas, on utilisera des pointeurs typés : le type du pointeur est celui de la donnée pointée.

Exemples :

Déclaration d'un pointeur sur *int* de nom p :
```
int *p;
```
Déclaration de deux pointeurs sur *float* de noms p1 et p2 :
```
float *p1,*p2;
```

3. La valeur NULL.

Un pointeur qui ne pointe sur rien doit être initialisé avec la valeur NULL :

```
int *x=NULL;
```

4. Affecter l'adresse d'une variable à un pointeur : operateur d'adressage &

- ✓ Opérateur unaire
- ✓ S'applique à un nom de variable
- ✓ Signifie « adresse de »
- ✓ Renvoie l'adresse de la variable auquel il s'applique
- ✓ Permet d'affecter l'adresse d'une variable à un pointeur

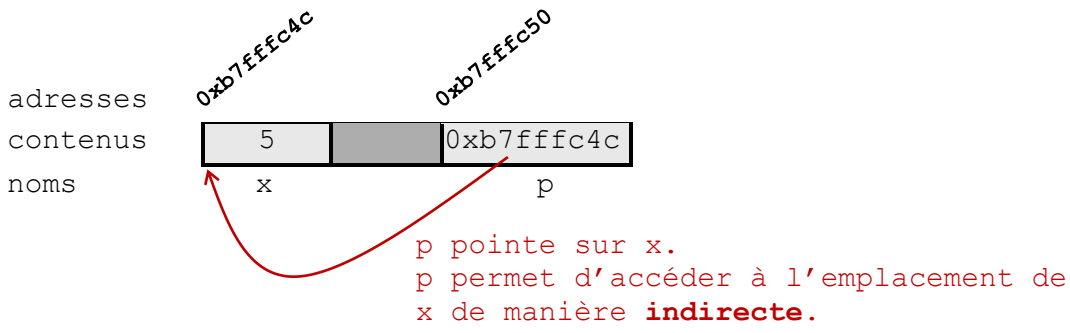
```c
#include <stdio.h>
int main(){
    //déclaration d'un pointeur sur int
    int *p;
    //déclaration et initialisation d'une variable de type int
    int x=5;
    /*"accrochage" du pointeur sur la variable
    p contiendra l'adresse de la variable x
    p pointera sur x */
    p=&x;
    printf("\nvariable p (pointeur) :");
    printf("\nadresse :%x\ttaille en octets: %zu\tvaleur :%x",&p,sizeof(p),p);
    printf("\nvariable x (int) :");
    printf("\nadresse :%x\ttaille en octets: %zu\tvaleur :%d",&x,sizeof(x),x);
    return 0;
}
```

Code illustratif

```
variable p (pointeur) :
adresse :b7fffc50       taille en octets: 8     valeur :b7fffc4c
variable x (int) :
adresse :b7fffc4c       taille en octets: 4     valeur :5
Process returned 0 (0x0)   execution time : 0.203 s
```

Résultat lors d'une exécution

Schéma mémoire :



5. Aller à l'adresse pointée : operateur d'indirection *

- ✓ Opérateur unaire
- ✓ S'applique à une adresse, par exemple à un pointeur
- ✓ Permet d'accéder à l'emplacement pointé

```c
#include <stdio.h>
int main(){
    int *p;
    int x=5,y;
    //"accrochage" de p sur x
    p=&x;
    //accès indirect à la valeur de x grâce à p
    y=*p;
    //modification indirecte de la valeur de x grâce à p
    *p=13;
    printf("\nx=%d",x);
    printf("\ny=%d",y);
    return 0;
}
```

Code illustratif

```
x=13
y=5
```

Résultat à l'exécution

V. Comprendre l'importance des pointeurs pour les sous-programmes

1. Pour qu'un sous-programme puisse modifier des données.

Exercice 23. Passage d'adresses en paramètres – Permuter les valeurs de deux variables.

La permutation est un traitement de base qui est utilisé dans de nombreux algorithmes, en particulier dans les algorithmes de tri.

On souhaite écrire un sous-programme qui permet de permuter les valeurs de deux variables.

Editez et exécutez le programme fourni « permutation.c ». Répondez aux questions puis corrigez-le.

```c
#include <stdio.h>
void permut(int x,int y){
    printf("\navant permutation :");
    printf("\n\tvariable x (parametre) du sous-programme permut : adresse=%x\tvaleur=%d",&x,x);
    printf("\n\tvariable y (parametre) du sous-programme permut : adresse=%x\tvaleur=%d",&y,y);
    int temp;
    temp=x;
    x=y;
    y=temp;
    printf("\napres permutation :");
    printf("\n\tvariable x (parametre) du sous-programme permut : adresse=%x\tvaleur=%d",&x,x);
    printf("\n\tvariable y (parametre) du sous-programme permut : adresse=%x\tvaleur=%d",&y,y);

}
int main(){
    int x,y;
    x=3;
    y=16;
    printf("variable x du sous-programme principal : adresse=%x\tvaleur=%d",&x,x);
    printf("\nvariable y du sous-programme principal : adresse=%x\tvaleur=%d",&y,y);
    //appel du sous-programme permut
    printf("\n\nAppel de permut :");
    permut(x,y);
    printf("\n\nFin de l'appel de permut, retour au sous-programme principal :\n");
    printf("variable x du sous-programme principal : adresse=%x\tvaleur=%d",&x,x);
    printf("\nvariable y du sous-programme principal : adresse=%x\tvaleur=%d\n\n",&y,y);

    return 0;
}
```

Code

```
variable x du sous-programme principal : adresse=fe5ff8ac       valeur=3
variable y du sous-programme principal : adresse=fe5ff8a8       valeur=16

Appel de permut :
avant permutation :
        variable x (parametre) du sous-programme permut : adresse=fe5ff880      valeur=3
        variable y (parametre) du sous-programme permut : adresse=fe5ff888      valeur=16
apres permutation :
        variable x (parametre) du sous-programme permut : adresse=fe5ff880      valeur=16
        variable y (parametre) du sous-programme permut : adresse=fe5ff888      valeur=3

Fin de l'appel de permut, retour au sous-programme principal :
variable x du sous-programme principal : adresse=fe5ff8ac       valeur=3
variable y du sous-programme principal : adresse=fe5ff8a8       valeur=16
```

Résultat lors d'une exécution

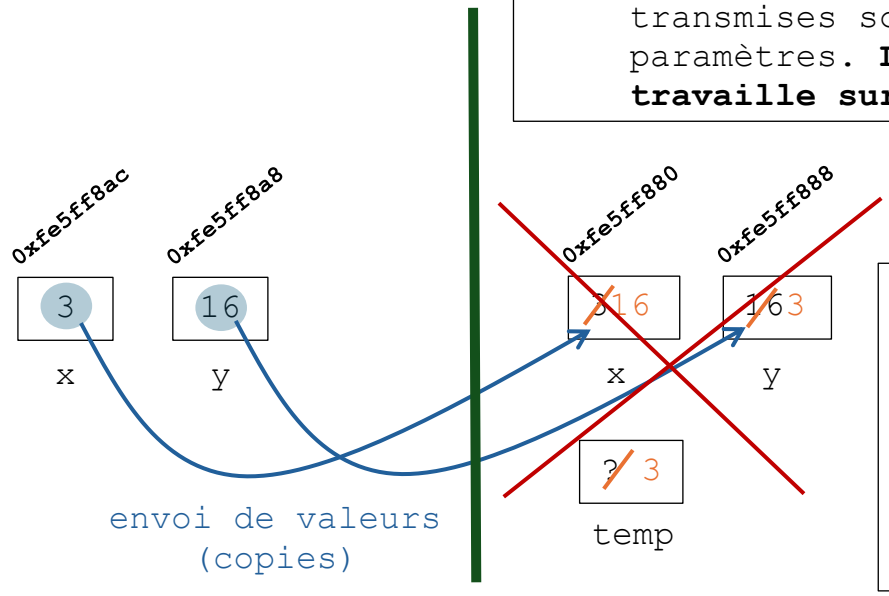
- Le sous-programme remplit-il son rôle : les valeurs des variables sont-elles bien permutées après l'appel du sous-programme ?
- Observez les adresses affichées : combien y-a-t-il d'emplacements mémoire réservés lorsque la première instruction du programme *permut* (ligne 3) est exécutée ?
- Le sous-programme travaille-t-il sur les variables du *main()* ?

- Qu'en déduisez-vous sur les affirmations suivantes :
  - On ne transmet pas en paramètres des variables mais juste leurs valeurs. ☐ vrai ☐ faux
  - On aurait pu coder le sous-programme en utilisant d'autres noms de paramètres par exemple a et b, cela n'aurait rien changé. ☐ vrai ☐ faux
  - Lors d'un projet en équipe, un programmeur n'a pas à se préoccuper des noms de paramètres et de variables utilisés par les autres programmeurs dans leurs sous-programmes. ☐ vrai ☐ faux
  - Pour utiliser un sous-programme, il faut connaitre les noms donnés aux paramètres. ☐ vrai ☐ faux

Schéma explicatif :
Voici un schéma illustrant ce qui se passe :

1) Le *main* s'exécute.
   - Des emplacements mémoire sont réservés pour ses variables locales x et y.

2) Le *main* appelle *permut* : le *main* s'interrompt et *permut* s'exécute.
   - Des emplacements mémoire sont réservés pour ses variables locales (paramètres) x et y. Ce ne sont pas les mêmes que celles du *main*.
   - x et y sont initialisées avec les valeurs transmises par l'appelant : les valeurs transmises sont copiées dans les paramètres. **Le sous-programme travaille sur des copies.**



- Un emplacement mémoire est réservé pour sa variable locale *temp*.
- Les variables locales sont modifiées

3) *permut* termine son exécution. Ses variables locales sont libérées. Retour au *main* : les variables du *main* n'ont pas changées

Conclusion : **Les pointeurs sont indispensables pour qu'un sous-programme puisse modifier les données stockées dans les variables d'un autre sous-programme.**

Explication imagée :
Vous souhaitez montrer à un ami le contenu d'un document pour qu'il en prenne connaissance et puisse l'utiliser : vous pouvez lui en envoyer une **copie**.
Vous souhaitez que votre ami modifie le document (le complète ou le corrige) : vous devez lui envoyer le **lien** vers l'original ou lui donner les **clés** de votre bureau.

- **Modifiez le sous-programme pour qu'il remplisse son rôle à l'aide de pointeurs. De retour dans le *main*, les valeurs des variables doivent être permutées.**
- **Faire un schéma mémoire illustrant le fonctionnement de cette version corrigée.**

2. Pour qu'un sous-programme puisse retransmettre plusieurs résultats.

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

Comment faire si un sous-programme doit retransmettre plusieurs résultats ?
Il faut lui donner des adresses auxquelles il ira stocker les résultats en utilisant l'indirection.

Exercice 24. Petit jeu style pacman/chasse au trésor. Programmation modulaire. Introduction à la programmation évènementielle.

En plus de vous faire **travailler sur les pointeurs**, cet exercice a pour objectifs de :

- rappeler la **démarche à suivre pour la bonne conception d'un programme.**
- initier à la **programmation évènementielle** (→ Annexe 1 sur boostcamp)
- montrer une **gestion un peu plus évoluée du mode console** (déplacement dans la console, affichage de couleurs …)
- guider et entrainer en vue de la **réalisation du projet.**

- **Cahier des charges**

  - **Objectif du jeu**

Le joueur incarne un "chasseur" qui doit attraper dans un temps limité, un maximum de trésors en se déplaçant dans une zone de jeu en console. À la fin du temps imparti, le programme affiche le score total.

  - **Règles du jeu**

- Le joueur commence avec un score de 0.
- Un seul trésor est visible à la fois, placé aléatoirement dans la zone de jeu.
- Lorsqu'il atteint la position du trésor, le joueur gagne 1 point et un nouveau trésor apparaît ailleurs.
- Le jeu dure un temps limité de 30 secondes.
- Quand le temps est écoulé, la partie se termine automatiquement et le score final est affiché.

- **IHM**

  - Le programme affiche un menu proposant de :
    1. Jouer
    2. Voir les règles du jeu
    3. Quitter

    Le programme tourne en boucle tant que l'utilisateur n'a pas choisi l'option Quitter
  - Le joueur peut quitter une partie à tout moment en appuyant sur Espace.
  - Zone de jeu rectangulaire de 20 colonnes × 10 lignes.
  - Caractères et couleurs utilisés :
    Joueur = @ en vert
    Trésor = * en rouge

  - Touches de déplacement :
    Z = haut, w = bas, q = gauche, s = droite

  - Informations affichées en permanence : score actuel, temps restant

- **Contraintes techniques**

- Langage : C.
- Mode graphique : console Windows.
- Bibliothèques autorisées :
  - <stdio.h> et <stdlib.h> (affichage, aléatoire)
  - <time.h> (aléatoire, gestion du chrono)
  - <conio.h> (lecture non bloquante des saisies clavier)
    → Annexe 1 : Programmation évènementielle
  - "affichage_console.h" (module supplémentaire fourni pour gérer l'affichage console : déplacement du curseur, couleurs …)

- **Programmation modulaire**
  Le programme doit être décomposé en plusieurs modules :

  - Un **module affichage** qui regroupe tous les sous-programmes d'affichage : affichage du menu, affichage des règles du jeu, affichage du cadre de la zone de jeu, affichage des objets à leurs positions ...

  - Un **module chasseur_tresor** regroupant tous les sous-programmes de traitement des données du jeu : déplacer le chasseur, générer un trésor, vérifier si le chasseur à attraper un trésor.

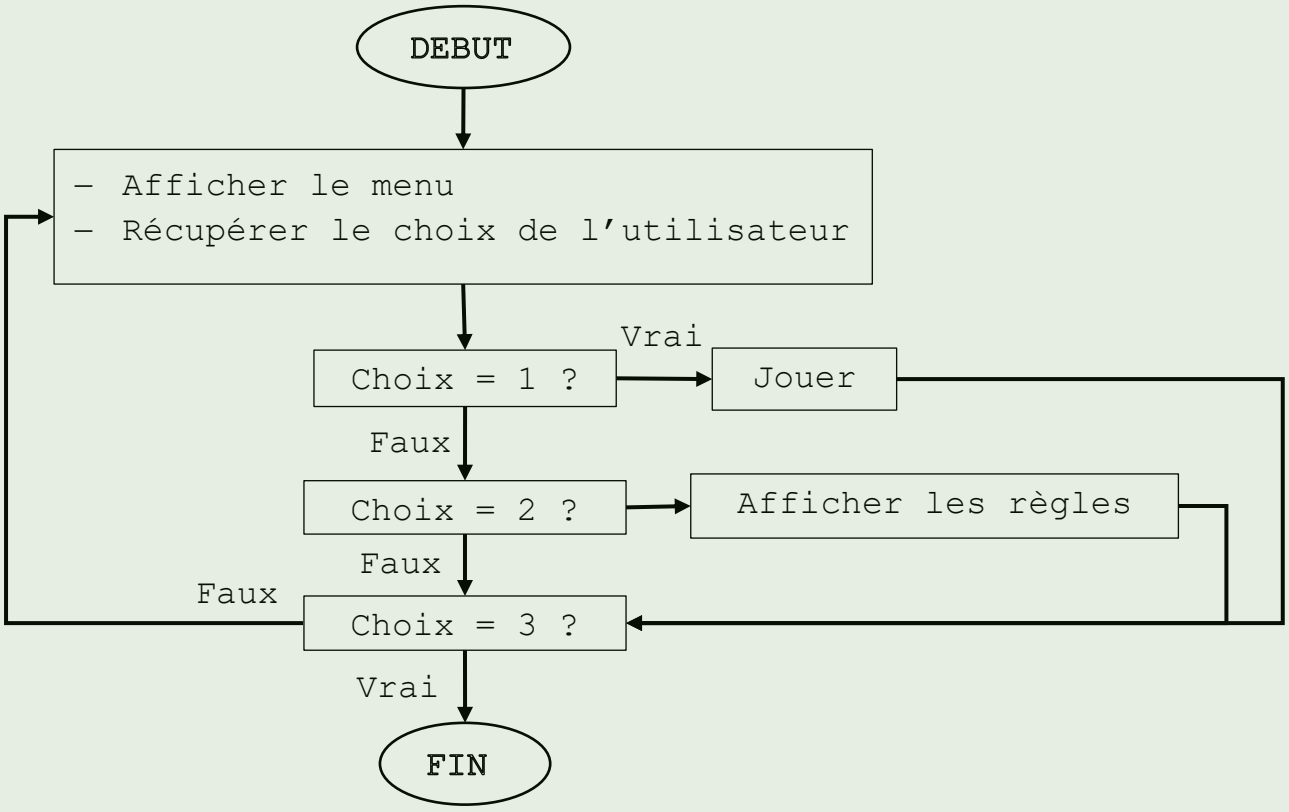
- Le fichier main.c avec le sous-programme principal. Pour éviter de surcharger le *main()*, on pourra déléguer la gestion d'une partie à un autre sous-programme.

• Conception

- **Les fonctionnalités du programme :**

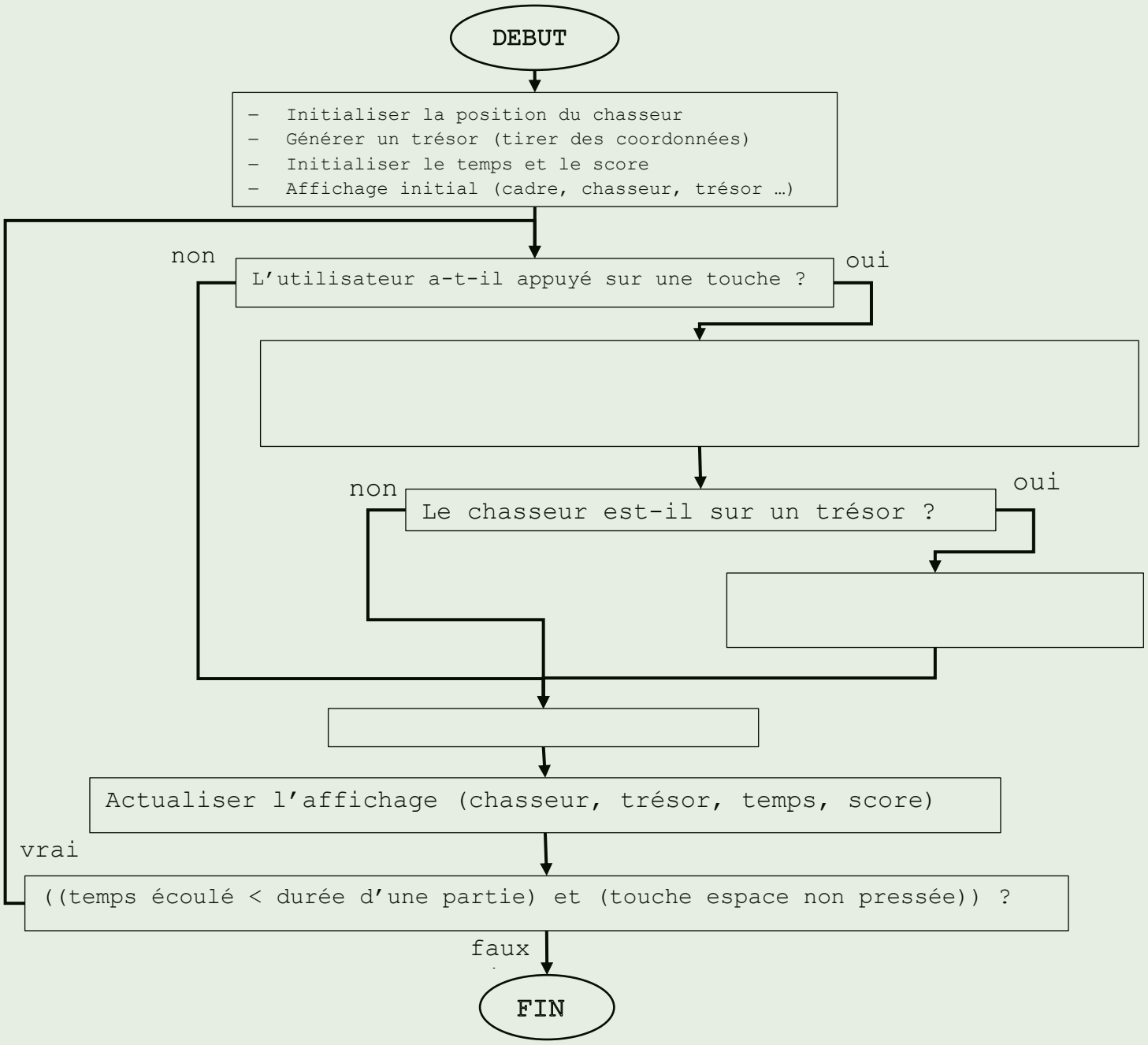
Fonctionnalité	**Données en entrée**	**Données en sortie**
Afficher le menu et récupérer le choix de l'utilisateur		le choix (entier)
Afficher les règles du jeu		
Jouer		

- **Logigramme montrant le fonctionnement général du programme**



- **Compléter le logigramme ci-dessous montrant le fonctionnement d'une partie (fonctionnalité jouer) en remettant les traitements suivants dans le bon cadre et dans le bon ordre :**

1. Générer un nouveau trésor (tirer de nouvelles coordonnées)
2. Augmenter le score de 1
3. Effacer le chasseur à son ancienne position
4. Déplacer le chasseur : modifier la position du chasseur selon la direction correspond à la touche pressée.
5. Récupérer la valeur de la touche pressée
6. Calculer le temps écoulé



- **Principaux traitements à coder dans des sous-programmes**

Pour vous aider dans la réalisation du jeu, on vous propose ci-dessous un découpage en sous-programmes :

Compléter les tableaux ci-dessous. Pour chaque donnée en entrée, précisez si elle est modifiée ou pas

- **Module affichage**

Nom du sous-programme	rôle	Données en entrée	Données en sortie	Traitements à effectuer
menu	Afficher le menu et récupérer le choix de l'utilisateur			Affichages Saisie
afficherRegles	Afficher les règles du jeu			
afficherCadre	Afficher le cadre de la zone de jeu			
afficherObjet	Afficher un caractère à une position dans la console			

- **Module chasseur_tresor**

Nom du sous-programme	rôle	Données en entrée	Données en sortie	Traitements à effectuer
deplacerChasseur	Déplacer le chasseur selon la valeur d'une touche appuyée	-la valeur de la touche - La position horizontale **(modifiée)** -		Selon la valeur de la touche pressée, augmenter ou diminuer la coordonnée x ou y de 1 en vérifiant que le chasseur ne sort pas de la zone de jeu.
genererTresor	Tirer aléatoirement les coordonnées du trésor dans les limites de la zone de jeu	- _		
attraperTresor	Vérifier si le chasseur a attrapé le trésor			Si le chasseur est sur un trésor : – Augmenter le score de 1 – Générer de nouvelles coordonnées pour le trésor

- **Implémentation**

o Ouvrez le projet « chasse_tresor.cbp » créé au chapitre précédent.

o Téléchargez le fichier « fonctions_chasse_tresor.c » fourni sur Boostcamp. Il contient les sous-programmes ci-dessus codés ou **à compléter**.

o Répartissez les sous-programmes dans les différents modules : les prototypes dans les headers, les implémentations dans les .c

- Complétez les sous-programmes.
- N'oubliez-pas d'inclure les headers des différents modules dans les fichiers qui en ont besoin.
- Vous pouvez utiliser la directive ***#define*** pour définir les dimensions de la zone de jeu et la durée d'une partie comme des constantes globales.
- Compilez, testez et jouez !