

Licence informatique et vidéoludisme

---

# **Programmation avancée**

## **de C à C++**

---

Rebekka Kyriakoglou  
Cours et exercices  
26 octobre 2022

# Plan

Présentation

La mémoire

Allocation dynamique de mémoire

Listes chaînées

Arbres binaires

# Présentation générale

(les points cités ci-après ne sont pas nécessairement abordés dans l'ordre)

- ▶ Rapides révisions de notions abordées en *Programmation Impérative*.
- ▶ Maîtriser les concepts avancés de la programmation C : renforcement pointeurs, allocation mémoire et organisation multi-fichiers; structures de données efficaces; mesure de performances.
- ▶ Apprendre à utiliser une sélection d'outils : make, utilisation/création de bibliothèques, gnuplot, GraphViz, gprof, valgrind, ...
- ▶ La généricité en C (usage des void \*).
- ▶ Utilisation poussée des macros (*C preprocessor*).
- ▶ Introduction au C++ sans paradigme objet.
- ▶ Utilisation de la STL (C++).

# Compilation

Compilation et génération de l'exécutable :

1. `gcc -Wall -Wextra -Wfatal-errors exo1.c -o exo1`

- `Wall` : "tous" les warnings
- `Wextra` : parce que "`-Wall`" ne contiens pas tout! (voir man `gcc`)
- `-Wfatal-errors` : arrêt de la compilation dès la 1ère erreur

2. Exécution : `./exo1`

# Compilation

Compilation et génération de l'exécutable :

1. `gcc -Wall -Wextra -Wfatal-errors exo1.c -o exo1`

- `Wall` : "tous" les warnings
- `Wextra` : parce que "`-Wall`" ne contiens pas tout! (voir man `gcc`)
- `-Wfatal-errors` : arrêt de la compilation dès la 1ère erreur

2. Exécution : `./exo1`

?

Avez-vous créé un répertoire pour ce cours et cette séance?

# Compilation

Compilation et génération de l'exécutable :

1. `gcc -Wall -Wextra -Wfatal-errors exo1.c -o exo1`

- `Wall` : "tous" les warnings
- `Wextra` : parce que "`-Wall`" ne contiens pas tout! (voir man `gcc`)
- `-Wfatal-errors` : arrêt de la compilation dès la 1ère erreur

2. Exécution : `./exo1`



Avez-vous créé un répertoire pour ce cours et cette séance?

`~/work/cours/ProgAvancee/Seance01/print_argv/`



Au fait, le `~/` est la référence vers votre *home directory* (racine de votre compte utilisateur)

## stdio.h

?

Quelle bibliothèque faut-il appeler pour utiliser printf?

## stdio.h

?

Quelle bibliothèque faut-il appeler pour utiliser printf?

Il faut taper **#include <stdio.h>** en haut de votre code.

## stdio.h

?

Quelle bibliothèque faut-il appeler pour utiliser printf?

Il faut taper **#include <stdio.h>** en haut de votre code.

?

Ecrire un programme qui imprime votre nom.

## Variables



Une variable est le nom donné à une zone de stockage que nos programmes peuvent manipuler.

## Variables



Une variable est le nom donné à une zone de stockage que nos programmes peuvent manipuler.

Une variable est une entité "constituée" des cinq éléments :

# Variables



Une variable est le nom donné à une zone de stockage que nos programmes peuvent manipuler.

Une variable est une entité "constituée" des cinq éléments :

- ▶ un identificateur (nom),
- ▶ type (intervalle de valeurs possibles),
- ▶ une valeur (à un moment donné),
- ▶ une adresse (emplacement mémoire),
- ▶ scope (sa durée de vie dans le programme).

# Variables



Une variable est le nom donné à une zone de stockage que nos programmes peuvent manipuler.

Une variable est une entité "constituée" des cinq éléments :

- ▶ un identificateur (nom),
- ▶ type (intervalle de valeurs possibles),
- ▶ une valeur (à un moment donné),
- ▶ une adresse (emplacement mémoire),
- ▶ scope (sa durée de vie dans le programme).



Il faut déclarer toutes les variables avant de les utiliser.

Type	Size (bytes)	octect	bits	Format Specifier
char	1	1	8	%c
float	4	4	32	%f
double	8	8	64	%lf
int	4	4	32	%d, %i

- ▶ Le mot **bit (b)** vient de l'anglais « binary digit ». Un bit est un nombre binaire qui peut prendre les valeurs **0** ou **1**.
- ▶ Un **byte (B)** est composé de **8 bits**.  
Un bit peut prendre 2 valeurs (0 ou 1), donc un byte peut prendre  $2^8 = 256$  différents valeurs.
- ▶ Un **octet (o)** est 1 byte qui est 8 bits.



**void** signifie "rien" ou "aucun type". Vous pouvez considérer void comme une absence.

Si une fonction ne renvoie rien, son type de retour doit être void.

# I-value et r-value

Comment affecter une **valeur** à une **variable**?

- ▶ À l'aide de l'opérateur affectation « = », lors de la déclaration de la variable ou plus tard, **ou bien**, par passage de paramètres (*i.e.* valable pour les arguments d'une fonction). Il y a d'autres moyens tels que memset/memcpy ou autres manipulations passant par des pointeurs : à voir plus tard.
- ▶ Dans tous les cas il faut bien distinguer ce qu'est une **I-value** et une **r-value**; **I** et **r** faisant respectivement référence à *left* et *right*, et ceci est lié à l'emplacement du symbole par rapport à l'opérateur d'affectation « = ». Ainsi, une **I-value** est un contenant, comme par exemple une boîte, et une **r-value** fait référence à une donnée, ou une information, qui correspond *in fine* à une valeur.

# I-value et r-value

```
#include <stdio.h>

int main(void){
    // declare 'a', 'b' an object of type 'int'
    int a = 1, b;

    // declare pointer variable 'p', and 'q'
    int *p, *q; // *p, *q are lvalue
    *p = 1; // valid l-value assignment

    // Invalid since "p + 2" is not an l-value
    // p + 2 = 18;

    q = p + 5; // valid - "p + 5" is an r-value

    // Below is valid - dereferencing pointer
    // expression gives an l-value
    *(p + 2) = 18;

    p = &b;

    int arr[20]; // arr[12] is an lvalue; equivalent to *(arr+12)

    // Note: arr itself is also an lvalue
    return 0;
}
```

Code source 1.1 – Exemples **r-value** et **l-value** r-value\_l-value.c

# Operateurs



## Opérateur

Les opérateurs sont des symboles qui permettent de manipuler des variables.

Il y a plusieurs types d'opérateurs :

- ▶ Opérateurs de calcul.
- ▶ Opérateurs d'assignation.
- ▶ Opérateurs d'incrémantation.
- ▶ Opérateurs de comparaison.
- ▶ Opérateurs logiques.
- ▶ Opérateurs bit-à-bit.
- ▶ Opérateurs de décalage de bit.

## Opérateurs de calcul

Opérateur	Dénomination	Effet
+	Addition	Ajoute deux valeurs
-	Soustraction	Soustrait deux valeurs
*	Multiplication	Multiplie deux valeurs
/	Division	Divise deux valeurs
=	Affectation	Affecte une valeur à une variable

## Opérateurs d'assignation et opérateurs d'incrémentation

Opérateur	Effet
<code>+=</code>	Additionne deux valeurs et stocke le résultat dans la variable (à gauche)
<code>-=</code>	Soustrait deux valeurs et stocke le résultat
<code>*=</code>	Multiplie deux valeurs et stocke le résultat
<code>/=</code>	Divise deux valeurs et stocke le résultat

## Opérateurs d'incrémentation

Opérateur	Effet
<code>++</code>	Augmente d'une unité la variable
<code>-</code>	Diminue d'une unité la variable



`x++` permet de remplacer `x=x+1` avec `x+=1`.

## Opérateurs de comparaison

Opérateur	Dénomination	Effet
==	Egalité	Vérifie l'égalité entre deux valeurs
<	Infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur
<=	Infériorité	vérifie qu'une variable est Inférieure ou égale à une valeur
>	Supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur
>=	Supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur
!=	Différence	Vérifie qu'une variable est différente d'une valeur



La valeur 1 correspond à **Vrai** et la valeur 0 à **Faux**.

## Opérateurs logiques

Opérateur	Dénomination	Effet
	OU	Vérifie qu'une des conditions est réalisée
&&	ET	Vérifie que toutes les conditions sont réalisées
!	NON	Inverse l'état d'une variable booléenne

**Opérateurs bit**

Opérateur	Dénomination
&	AND
	OR
$\wedge$	XOR
$<<$	left shift
$>>$	right shift
$\sim$	Inverse

- ▶ & : Prend deux nbr et effectue le AND sur chaque bit des deux nbr.
- ▶ | : Prend deux nbr et effectue un OR sur chaque bit de deux nbr.
- ▶  $\wedge$  : (XOR au sens des bits) prend deux nbr et effectue un XOR sur chaque bit de deux nbr.  
Le résultat de XOR est 1 si les deux bits sont différents.
- ▶  $<<$  : Prend deux nbr, décale à gauche les bits du premier, le second décide du nbr de places à décaler.
- ▶  $>>$  : Prend deux nbr, décale à droite les bits du premier, le second décide du nbr de places à décaler.
- ▶  $\sim$  : Prend un nbr et inverse tous les bits de celui-ci.

# Operateur sizeof



## sizeof

**Opérateur** qui donne la **quantité de stockage**, en octets, obligatoire pour stocker un objet du type de l'opérande. Cet opérateur vous permet d'éviter de spécifier les tailles de données dépendantes de l'ordinateur dans vos programmes. Le résultat de sizeof est de type intégral non signé, généralement désigné par size\_t.

### Syntaxe :

sizeof( type )

sizeof expression

# Opérateurs



Saisir ce code, sans faire un copier/coller, dans un éditeur convenable ayant au minimum une coloration syntaxique et une capacité à l'indenter correctement!!!

```
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
    unsigned char a = 5, b = 9;
    // a = 5(00000101), b = 9(00001001)
    printf("a=_%d,_b=_%d\n", a, b);
    printf("a&b=_%d\n", a & b);
    printf("a|b=_%d\n", a | b);
    printf("a^b=_%d\n", a ^ b);
    printf("~a=_%d\n", a = ~a);
    printf("b<<1=_%d\n", b << 1);
    printf("b>>1=_%d\n", b >> 1);

    return 0;
}
```

Code source 1.2 – Exemples Operateurs bitwise\_operator\_ex1.c



Expliquer les opérations de ce programme.

# Exercices

## Exercise

■ Essayez d'expliquer ce qui se passe avec le code suivant.

```
#include <stdio.h>

int main(void){
    int x = 5;
    (x & 1) ? printf("Odd\n") : printf("Even\n");
    return 0;
}
```

Code source 1.3 – Pair ou impair bitwise\_operator\_ex2.c

## Notions de base – les fonctions (en C) 1/3

### ► Qu'est-ce qu'une fonction ?

 Dans un programme, une fonction peut être résumée à une séquence d'instructions regroupées pour réaliser une tâche (généralement actions ou calculs) particulière, dans un cadre délimité<sup>1</sup> ou restreint, lié à ses paramètres, s'ils existent. Une fonction peut donc être appelée (ou utilisée) plusieurs fois au sein du programme, ou, dans le cas spécifique de l'écriture d'une bibliothèque de fonctions, être réservée à un usage ultérieur par d'autres programmes utilisant la dite bibliothèque.

- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

---

1. Voir néanmoins le cas particulier de la fonction à effet de bord :  
[https://fr.wikipedia.org/wiki/Effet\\_de\\_bord\\_\(informatique\)](https://fr.wikipedia.org/wiki/Effet_de_bord_(informatique))

## Notions de base – les fonctions (en C) 2/3

- ▶ Qu'est-ce qu'une fonction?
- ▶ Déclarer ET définir une fonction?

**Déclarer** une fonction consiste à donner un prototype, le plus complet possible, permettant de la qualifier<sup>2</sup>, de donner son type de retour et d'informer sur ses paramètres. En donnant, à la suite, le type de chaque paramètre, nous déduisons leur nombre. Si le premier paramètre est void cela indique que la fonction n'en a pas. Il est important de déclarer une fonction avant de la définir; la déclaration se fait en haut du fichier C quand les fonctions sont statiques et dans un fichier d'en-têtes (fichier .h) quand les fonctions sont externes. Ci-après, une forme générale d'un prototype :

```
[qualifier1 [...] ] <type> [* [...] ] nom_de_la_fonction(<void> | <type1>[ , <type2>[...] ]);
```

**Définir** une fonction consiste à donner « *dans les grandes lignes*<sup>3</sup> » son prototype, directement (à la place du « ;») suivi du corps de la fonction entre accolades. Si le type de retour de la fonction est autre que void, alors le corps de la fonction doit avoir un retour (*i.e.* return ...;) correspondant à ce type dans toutes les situations rencontrées (tous les branchements du code). Ci-après, une forme générale d'une définition :

```
<type> [* [...] ] nom_de_la_fonction(<void> | <type1 nom1>[ , <type2 nom2>[...] ]) {  
    /* les instructions qui constituent le corps de la fonction */  
    /* ne pas oublier un return lié au type dans toute situation */  
}
```

- ▶ Quelques *qualifiers* et leurs impacts?

2. Définir ses particularités : sans être exhaustifs, citons extern contre static, exportable au sein d'une bibliothèque ou non, ou encore *inlinable* (cf. inline) ou non (les deux derniers points sont compilateur-spécifiques).
3. Si la fonction est correctement prototypée (déclarée) avant sa définition, remettre les *qualifiers* est optionnel, voire non-productif. Par contre, dans le cas de la définition de fonction, il est nécessaire de nommer les paramètres (en plus des types).

## Notions de base – les fonctions (en C) 3/3

- ▶ Qu'est-ce qu'une fonction?
- ▶ Déclarer ET définir une fonction?
- ▶ Quelques *qualifiers* et leurs impacts?

Le principal couple de *qualifiers* à connaître est : static ou extern. Le premier indique que la fonction a une portée limitée au fichier (à partir de sa déclaration) alors que le second lui permet d'avoir une portée au delà du fichier (*i.e.* le fichier .c), évidemment si son prototype est donné (généralement dans le fichier *header* .h inclus via la directive de préprocesseur #include).



La vidéo [https://youtu.be/Q\\_XliWuKsKw](https://youtu.be/Q_XliWuKsKw) peut aider à comprendre la structuration multi-fichiers d'un programme et les concepts de static et extern. Prenez le temps de la visualiser en entier.

# Fonctions

## ?

### Exercise

Écrire une fonction void bin(char c) pour imprimer la version binaire d'un intier. Vous devez utiliser les opérateurs bit à bit et décalage de bit.

## Exercices

1. Écrire un programme qui calcule les (deux) racines de l'équation quadratique :  $ax^2 + bx + c$  où  $a = 1.2$ ,  $b = 2.3$  et  $c = -3.4$ .
2. Exécutez le code ci-dessous et examinez la sortie.

```
#include <stdio.h>

//ERRORS
void function_call(int i, float f)
{
    printf("%d\n", i * f);
}

main()
{
    double d = 20;
    float f = 10;
    function_call(d, f);
}
```

**Code source 1.4 – correct\_errors\_1.c**

3. Ecrire un programme pour afficher le motif comme un triangle à angle droit en utilisant un astérisque.
4. Écrire un programme pour imprimer les 50 premiers nombres naturels en utilisant la récursion.
5. Ecrire un programme qui lit deux nombres entiers et  $a$  et  $b$  et donne le choix à l'utilisateur :  
1. de savoir si  $a + b$  est pair; 2. de savoir si  $ab$  est pair; 3. de connaître le signe de  $a + b$ ; 4. de connaître le signe de  $ab$ .  
Attention, vous pouvez utiliser switch().
6. Écrire un programme pour estimer la racine carrée d'un nombre en utilisant la méthode de Newton.  
<https://en.wikipedia.org/wiki/Newton>

## Rue la RAM

Imaginez la mémoire comme un tableau de cases numérotées ou adressées.

Par Exemple :

- ▶ Un octet peut représenter un char.
- ▶ Un pair d'octets peut représenter un entier short.
- ▶ Quatre octets peuvent représenter un long.

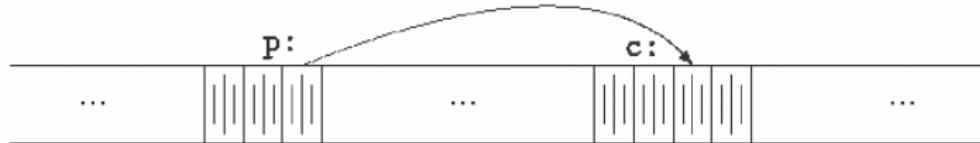
# Pointeurs

L'idée de base est qu'un **pointeur** est un type de données spécial (i.e. un groupe de cases), qui contient une adresse vers un emplacement en mémoire.



Pensez à un pointeur comme à une flèche qui pointe vers l'emplacement d'un bloc de mémoire qui contient à son tour des données intéressantes.

$$p = \&c$$



C'est un concept similaire à celui de l'adresse d'une maison.



L'opérateur unaire & donne l'**adress** d'un objet.



L'opérateur unaire **\*** représente l'opérateur d'**indirection** ou de **déréférence** (ie il donne accès à l'objet pointé par ce pointeur).

```
1 #include <stdio.h>
2
3 int main (void)
4 {
5     int age = 18;
6     int *p;
7     p = &age;
8     printf("age=%d\n", age);
9     printf("p=%p\n", p);
10    printf("*p=%d\n", *p);
11    printf("sizeof(p)=%ld\n", sizeof(p));
12    *p = 19;
13    printf("*p=%d\n", *p);
14    printf("age=%d\n", age);
15    return 0;
16 }
```

Code source : pointers\_print\_example.c

# Type d'un pointeur



## Type d'un pointeur

Même si la valeur d'un pointeur est toujours un entier, le type d'un pointeur dépend du type de l'objet vers lequel il pointe. En général, nous déclarons le type vers lequel pointe un pointeur, mais l'exception est un pointeur dit "**pointeur vide**" ("void pointer").

# Déclaration des pointeurs

Syntaxe générale pour déclarer les pointeurs :

```
data_type *pointer_name
```

# Déclaration des pointeurs

Syntaxe générale pour déclarer les pointeurs :

```
data_type *pointer_name
```



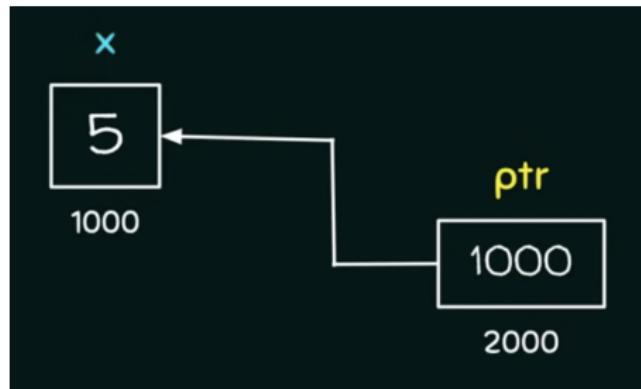
Déclarer un pointeur n'est pas suffisant, il est préférable de l'**initialiser**.

# Initialisation des pointeurs



Une manière d'initialiser un pointeur est d'**assigner l'adresse d'une variable**.

```
int x = 5;  
int *p; //declaration of the pointer p  
p = &x; //initialisation of the pointer p
```

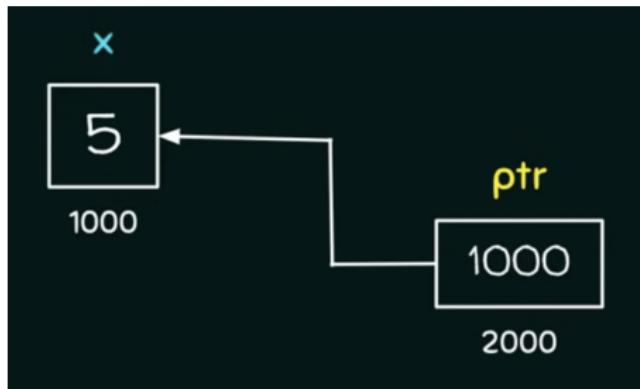


# Initialisation des pointeurs



Une manière d'initialiser un pointeur est d'**assigner l'adresse d'une variable**.

```
int x = 5;  
int *p; //declaration of the pointer p  
p = &x; //initialisation of the pointer p
```



Equivalent à : int x = 5; \*p = &x;

## Quiz



Figure – Votez

Soit  $n$  est une variable et  $p$  un pointeur sur  $n$ , quelles expressions parmi les suivantes sont des alliances avec celle-ci ?

1.  $*p$
2.  $*\&p$
3.  $\&p$
4.  $*n$
5.  $*\&n$

# Questions



## Question

Dans le code suivant, la première ligne est une **déclaration** et la deuxième est une instruction d'**affectation**.

Pourquoi, dans la deuxième ligne, p n'est pas précédé du symbole \*?

```
int *p = &n;  
p = &n;
```

# Questions



## Question

Dans le code suivant, la première ligne est une **déclaration** et la deuxième est une instruction d'**affectation**.

Pourquoi, dans la deuxième ligne, p n'est pas précédé du symbole \*?

```
int *p = &n;  
p = &n;
```

### SOLUTION :

Le symbole \* dans la déclaration indique au compilateur que p est un **pointeur vers un entier**. Il ne s'agit pas d'un opérateur d'indirection.

Si nous écrivons \*p=&i alors c'est FAUX, car \* symbol indique l'opération d'indirection et nous ne pouvons pas affecter l'adresse à une variable int.

# Adresse la mémoire



Saisir ce code, sans faire un copier/coller :

```
#include <stdio.h>

int main(void){
    int i = 1, j = 2, k = 3;
    printf("adresse_i=%p\n", &i);
    printf("adresse_j=%p\n", &j);
    printf("adresse_k=%p\n", &k);
    /* Que remarquez-vous concernant leur adresse ?*/

    int l;
    int *p = &i;
    /* Je prends un risque :) */
    for (l = 0; l < 3; l++){
        printf("%d\n", p[l]);
    }
    return 0;
}
```

Code source 1.5 – printf les adresse de la mémoire printf\_ex1.c



Expliquer ce programme.

# Adresse la mémoire



Saisir ce code, sans faire un copier/coller :

```
#include<stdio.h>

int main(void){
    int t[] = {4, 5, 6};
    int *p = t;

    //this code is secure in comparison to the printf_ex1.c
    int i;
    for (i = 0; i < 3; i++){
        printf("%d\n", p[i]);
    }
    return 0;
}
```

Code source 1.6 – printf les adresse de la mémoire printf\_ex2.c



Expliquer ce programme.

# Arithmétique des pointeurs

Les opérations arithmétiques valides sur les pointeurs :

## Arithmétique des pointeurs

Les opérations arithmétiques valides sur les pointeurs :

- ▶ L'addition d'un entier à un pointeur.

Le résultat est un pointeur de même type que le pointeur de départ.

# Arithmétique des pointeurs

Les opérations arithmétiques valides sur les pointeurs :

- ▶ L'addition d'un entier à un pointeur.

Le résultat est un pointeur de même type que le pointeur de départ.

- ▶ La soustraction d'un entier à un pointeur.

Le résultat est un pointeur de même type que le pointeur de départ.

## Arithmétique des pointeurs

Les opérations arithmétiques valides sur les pointeurs :

- ▶ L'addition d'un entier à un pointeur.  
*Le résultat est un pointeur de même type que le pointeur de départ.*
- ▶ La soustraction d'un entier à un pointeur.  
*Le résultat est un pointeur de même type que le pointeur de départ.*
- ▶ La différence de deux pointeurs pointant tous deux vers des objets de même type.  
*Le résultat est un entier.*

Soit **i** un entier et **p** est un pointeur sur un objet de type type



L'expression **p + i** désigne un **pointeur** sur un objet de type type dont la valeur est :

la valeur de **p** incrémentée de **i \* sizeof(type)**.

Soit **i** un entier et **p** est un pointeur sur un objet de type type



L'expression **p + i** désigne un **pointeur** sur un objet de type type dont la valeur est :

la valeur de p incrémentée de **i \* sizeof(type)**.

```
main(void){  
    int i = 3;  
    int *p1, *p2;  
    p1 = &i;  
    p2 = p1 + 1;  
    printf("p1=%d\t p2=%d\n", p1, p2);  
}
```



## Exercice 1

Essayez le même programme avec des pointeurs sur des objets de type double. Qu'est-ce que vous attendez ?

```
#define N 5

int tab[4] = {1, 2, 3, 4};

int main(void){
    int *p;
    printf("\n_ordre_croissant:\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf(" %d \n", *p);
    printf("\n_ordre_decroissant:\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf(" %d \n", *p);

    return 0;
}
```



## Exercise 2

Pouvez-vous expliquer le programme ci-dessus? Quel est le rôle des opérations arithmétiques?

```
#define N 5

int tab[4] = {1, 2, 3, 4};

int main(void){
    int *p;
    printf("\n_ordre_croissant:\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf("%d\n", *p);
    printf("\n_ordre_decroissant:\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf("%d\n", *p);

    return 0;
}
```



## Exercise 2

Pouvez-vous expliquer le programme ci-dessus? Quel est le rôle des opérations arithmétiques?

Réponse : Ils ont utilisé pour parcourir des tableaux.

## Tableaux et pointeurs (1/2)



Lorsque vous déclarez le tableau `int tab[4]`, ce qui se passe c'est qu'un bloc de mémoire est alloué (sur la pile dans ce cas) suffisamment grand pour contenir 4 entiers, et la variable `tab` est un pointeur qui pointe vers le premier élément du tableau.

## Tableaux et pointeurs(2/2)

Notation des tableaux	Équivalent
tab[0]	$*a$
tab[1]	$*(a+1)$
tab[2]	$*(a+2)$
tab[3]	$*(a+3)$
tab[n]	$*(a+n)$

## Révision : Opérateur ++ et - comme préfixe et suffixe

- ▶ **++var** : la valeur de var est incrémentée de 1, puis la valeur est retournée.
- ▶ **var++** : la valeur originale de var est retournée en premier, puis, var est incrémentée de 1.

# Révision : Opérateur ++ et - comme préfixe et suffixe

- ▶ **++var** : la valeur de var est incrémentée de 1, puis la valeur est retournée.
- ▶ **var++** : la valeur originale de var est retournée en premier, puis, var est incrémentée de 1.

```
#include <stdio.h>
int main(void){
    int var1 = 5, var2 = 5;

    // 5 is displayed. Then, var1 is increased to 6.
    printf("%d\n", var1++);
    printf("var1=%d\n", var1);

    // var2 is increased to 6. Then, it is displayed.
    printf("%d\n", ++var2);
    printf("var2=%d\n", var2);

    return 0;
}
```

## Pointeurs : Opérateur ++ et – comme préfixe et suffixe

- ▶ `*(++p)` : l'incrémentation se produit d'abord, puis l'affectation
- ▶ `*(p++)` : l'affectation se produit d'abord, puis l'incrémentation

# Pointeurs : Opérateur ++ et – comme préfixe et suffixe

- ▶ **\*(++p)** : l'incrémentation se produit d'abord, puis l'affectation
- ▶ **\*(p++)** : l'affectation se produit d'abord, puis l'incrémentation

## Programme 1 :

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[0];

    printf("%d\n", *p);
    printf("%d\n", *(p++));
    printf("%d\n", *p);

    return 0;
}
```

# Pointeurs : Opérateur ++ et - comme préfixe et suffixe

- ▶ **\*(++p)** : l'incrémentation se produit d'abord, puis l'affectation
- ▶ **\*(p++)** : l'affectation se produit d'abord, puis l'incrémentation

## Programme 1 :

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[0];

    printf("%d\n", *p);
    printf("%d\n", *(p++));
    printf("%d\n", *p);

    return 0;
}
```

## Programme 2 :

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[0];

    printf("%d\n", *p);
    printf("%d\n", *(++p));
    printf("%d\n", *p);

    return 0;
}
```

# Décrémentation

## Exemple

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[2];

    printf("%d\n", *(--p));
    printf("%d\n", *(p--));
    printf("%d\n", *p);

    return 0;
}
```

# Décrémentation

## Exemple

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[2];

    printf("%d\n", *(--p));
    printf("%d\n", *(p--));
    printf("%d\n", *p);

    return 0;
}
```

OUTPUT :

16  
16  
5

# Opérateur d'incrémentation et pointeurs

```
//Difference between    +++p, *p++ and ***p in C

#include <stdio.h>

int main(void) {
    int arr[3] = {20, 30, 40};
    int *p = arr;
    int q;

    //value pointed by p (20) incremented by 1 and returned
    q = ++(*p);
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    //value pointed by p is returned pointer incremented by 1
    q = (*p)++;
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    //pointer incremented by 1 value returned
    q = *(++p);
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    return 0;
}
```

Code source 1.7 – pointers\_and\_increment\_operator.c

# Opérateur d'incrémentation et pointeurs

```
//Difference between    +++p, *p++ and ***p in C

#include <stdio.h>

int main(void) {
    int arr[3] = {20, 30, 40};
    int *p = arr;
    int q;

    //value pointed by p (20) incremented by 1 and returned
    q = ++(*p);
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    //value pointed by p is returned pointer incremented by 1
    q = (*p)++;
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    //pointer incremented by 1 value returned
    q = *(++p);
    printf("arr[0]=%d, arr[1]=%d, *p=%d, q=%d\n", arr[0], arr[1], *p, q);

    return 0;
}
```

Code source 1.8 – pointers\_and\_increment\_operator.c

```
arr[0]=21, arr[1]=30, *p=21, q=21
arr[0]=22, arr[1]=30, *p=22, q=21
arr[0]=22, arr[1]=30, *p=30, q=30
```

## Quiz



Figure – Votez

Quel est le résultat du programme suivant?

```
#include <stdio.h>
int main(void){
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};s
    int *p = &a[1];
    int *q = &a[5];

    printf("%d\n", *(p+3));
    printf("%d\n", *(q-3));
    printf("%d\n", q-p);
    printf("%d\n", q<p);
    printf("%d\n", q>p);

    return 0;
}
```

1. 45, 7, 4, 1, 1
2. 45, 4, 7, 1, 1
3. 44, 7, 4, 1, 0
4. 45, 7, 4, 0, 1

★ Une application des pointeurs est de contourner la limite du C qui ne permet d'avoir qu'**une seule valeur de retour** d'une fonction.



Une application des pointeurs est de contourner la limite du C qui ne permet d'avoir qu'**une seule valeur de retour** d'une fonction.

## Exercise

Pourquoi la fonction doubler n'a pas d'effet sur y?

Comment devons-nous modifier le programme pour surmonter ce problème?

```
void doubler(int x){  
    x *= 2;}  
  
int main(void){  
    int y;  
    y = 1;  
    printf("%d\n", y); /* prints 1 */  
    doubler(y); /* no effect on y */  
    printf("%d\n", y); /* prints 1 */  
    return 0;  
}
```



Une application des pointeurs est de contourner la limite du C qui ne permet d'avoir qu'**une seule valeur de retour** d'une fonction.

## Exercice

Pourquoi la fonction doubler n'a pas d'effet sur y?

Comment devons-nous modifier le programme pour surmonter ce problème?

```
void doubler(int x){  
    x *= 2;  
  
int main(void){  
    int y;  
    y = 1;  
    printf("%d\n", y); /* prints 1 */  
    doubler(y); /* no effect on y */  
    printf("%d\n", y); /* prints 1 */  
    return 0;  
}
```

### Réponse :

void doubler(int \*x){ \*x \*= 2; }

ET

doubler(&y);

## Exemple - Tableau comme argument

```
#include<stdio.h>

int add(int *arr, int len){
    int sum = 0;
    int i;
    for (i = 0; i < len; i++){
        sum += arr[i];
    }
    return sum;
}

int main(void){
    int tab[] = {1, 2, 3, 4, 5};
    int taille = sizeof(tab)/sizeof(tab[0]);
    printf("sum_= %d\n", add(tab, taille));
    return 0;
}
```

Code source 1.9 – Exemple simple\_exemple\_array\_as\_argument.c



Il faut faire attention à bien distinguer les notions de paramètre et d'argument qui sont deux choses différentes.

## Exemple - Tableau comme argument

```
#include<stdio.h>

int add(int *arr, int len){
    int sum = 0;
    int i;
    for (i = 0; i < len; i++){
        sum += arr[i];
    }
    return sum;
}

int main(void){
    int tab[] = {1, 2, 3, 4, 5};
    int taille = sizeof(tab)/sizeof(tab[0]);
    printf("sum_= %d\n", add(tab, taille));
    return 0;
}
```

Code source 1.10 – Exemple simple\_exemple\_array\_as\_argument.c



Il faut faire attention à bien distinguer les notions de paramètre et d'argument qui sont deux choses différentes.

- ▶ Les symboles arr et len de son prototype sont ses **paramètres**.
- ▶ Lors de l'appel, les expressions tab et taille sont les **arguments** de l'appel.

- ▶ Si vous passez la valeur d'une variable dans une fonction (sans &), vous pouvez être assuré que la fonction ne peut pas modifier votre variable d'origine.
- ▶ Lorsque vous passez un pointeur, vous devez supposer que la fonction peut et va modifier la valeur de la variable.
- ▶ Si vous voulez écrire une fonction qui prend un pointeur en argument mais ne pas modifie la cible du pointeur, utilisez const.

## Portée lexicale des paramètres



Les variables locales d'une fonction ont pour portée lexicale la fonction elle-même.



Les paramètres d'une fonction ont pour portée lexicale la fonction elle-même.

# Portée lexicale des paramètres



Les variables locales d'une fonction ont pour portée lexicale la fonction elle-même.



Les paramètres d'une fonction ont pour portée lexicale la fonction elle-même.

```
int foo(int a){  
    return 2 * a;}  
  
int main(void){  
    int a;  
    a = 10;  
    a = foo(a + 1);  
    {  
        int a = 0; \\shadow  
        for (; a < 10; a++)  
            printf("%d\n",a);  
    }  
    printf("%d\n",a);  
    return 0;  
}
```

Il y plusieurs occurrences de a MAIS ce sont des variables différentes!!!

# Exemple



## Exercise

Essayez d'expliquer ce qui se passe avec le code suivant.

```
#include<stdio.h>

int main(void){
    int i = -10;
    for(int i = 0; i < 10; ++i)
        printf("%d\n",i);
    printf("%d\n",i);

    return 0;
}
```

## Exemple

La rue de la RAM 4/XX (slides Farés Belhadj)

# Double et triple pointeur

## ★ Double Pointeur

Le pointeur est une variable (elle existe physiquement en RAM) qui stocke l'adresse mémoire d'une autre variable. Ainsi, lorsque nous définissons un **pointeur vers un pointeur**, le premier pointeur est utilisé pour stocker l'adresse de la variable, et le second pointeur est utilisé pour stocker l'adresse du premier pointeur.

Syntaxe : `data_type **variable_Name ;`

# Double et triple pointeur

## ★ Double Pointeur

Le pointeur est une variable (elle existe physiquement en RAM) qui stocke l'adresse mémoire d'une autre variable. Ainsi, lorsque nous définissons un **pointeur vers un pointeur**, le premier pointeur est utilisé pour stocker l'adresse de la variable, et le second pointeur est utilisé pour stocker l'adresse du premier pointeur.

Syntaxe : `data_type **variable_Name ;`

## ★ Triple Pointeur

Un **pointeur triple** est un pointeur qui pointe vers un emplacement mémoire où est stocké un pointeur double. Le pointeur triple lui-même n'est qu'un pointeur.

Par exemple, `int ***` est un pointeur qui renvoie à la valeur d'un pointeur double, qui renvoie à son tour à la valeur d'un pointeur simple, qui renvoie à la valeur d'un int.

Syntaxe : `data_type ***variable_Name ;`

## Exercice (Facile)

Dessinez cet exemple puis faites l'exercice :

En n'utilisant que printf et d, affichez le contenu de chaque variable d, c, b, a.

```
#include <stdio.h>

int main(void){
    short a = 42, * b = &a, ** c = &b, *** d = &c;
    printf("a_=_%d,_son_adresse_est_%p\n", a, &a);
    printf("b_=_%p,_son_adresse_est_%p\n", b, &b);
    printf("c_=_%p,_son_adresse_est_%p\n", c, &c);
    printf("d_=_%p,_son_adresse_est_%p\n", d, &d);
    /* EXERCICE ! (FACILE) */

    //printf("%p, %p, %p, %d\n", .....);
    return 0; }
```

Code source 1.11 – Exemple triple\_pointer.c

# La mémoire

Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la **mémoire**.



On peut voir la mémoire comme un **très grand tableau d'octets**.

La mémoire est segmentée en plusieurs parties :

- ▶ la zone statique qui contient le code et les données statiques;
- ▶ le tas, de taille variable au fil de l'exécution;
- ▶ la pile, de taille variable au fil de l'exécution;
- ▶ d'autres zones (non repr. ici).

# La mémoire

Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la **mémoire**.



On peut voir la mémoire comme un **très grand tableau d'octets**.

La mémoire est segmentée en plusieurs parties :

- ▶ la zone statique qui contient le code et les données statiques;
- ▶ le tas, de taille variable au fil de l'exécution;
- ▶ la pile, de taille variable au fil de l'exécution;
- ▶ d'autres zones (non repr. ici).



Le **tas** contient les variables allouées dynamiquement.



La **pile** contient les variables locales lors des appels aux fonctions.

# La mémoire



# Exercice

Faire un schéma du contenu connu de mémoire liée à ce programme. Sans tester sur une machine, essayez de deviner quel affichage nous obtenons suite à son exécution.

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

int main(void) {
    char str01[] = "Hello", str02[] = "world";
    char ** ptr = NULL;
    ptr = malloc(2 * sizeof *ptr);
    assert(ptr);
    ptr[0] = str01;
    ptr[1] = str02;
    printf("%c\n", *ptr[1]);
    printf("%c\n", (*ptr)[1]);
    free(ptr);
    return 0;
}
```

Code source 2.1 – Exemple memory\_allocation\_ex1.c

## Stack (en français pile)

Lors de l'appel d'une fonction, les valeurs de ses arguments sont recopiées dans une zone de la mémoire appelée **pile**.

- ▶ Les **variables locales** d'une fonction se situent dans la pile.
- ▶ La valeur renvoyée (si son type de retour n'est pas void) se situe dans la pile.
- ▶ Après avoir appelé une fonction, la pile se trouve dans le même état qu'avant l'appel.

```
#include <stdio.h>
double multiply (double input) {
    double twice = input * 2.0;
    return twice;
}

int main(void){
    int age = 30;
    double salary = 12345.67;
    double myList[3] = {1.2, 2.3, 3.4};
    printf("Votre salaire est %.3f\n", multiply(salary));

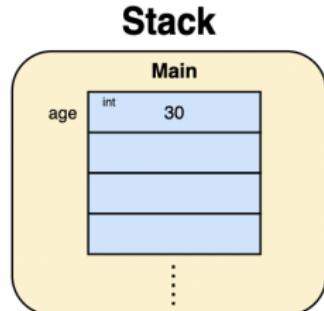
    return 0;
}
```



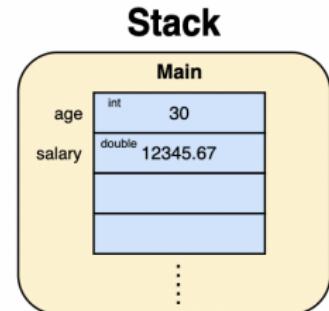
## Exercise 1<sup>4</sup>

I Le programme ci-dessus crée ses variables sur la pile. Pouvez-vous expliquer comment cela fonctionne.

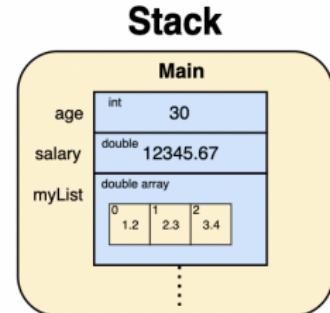
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```



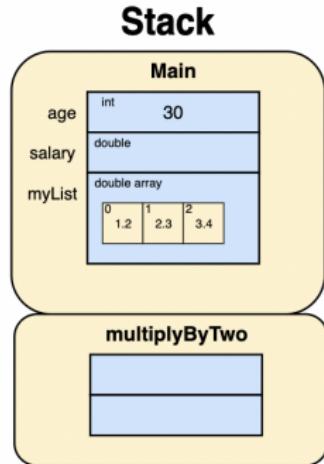
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```



```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

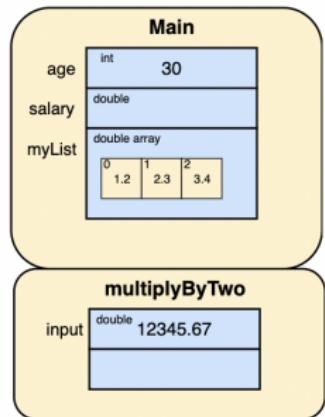


```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```



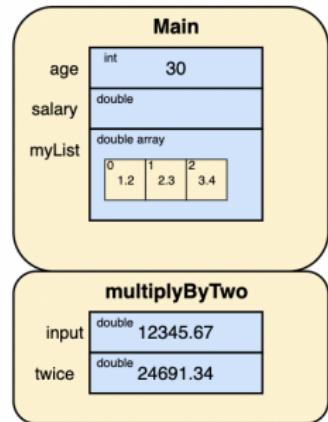
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

## Stack



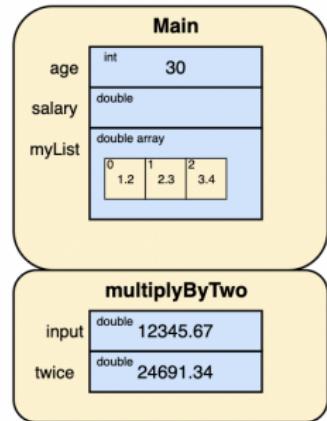
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

## Stack



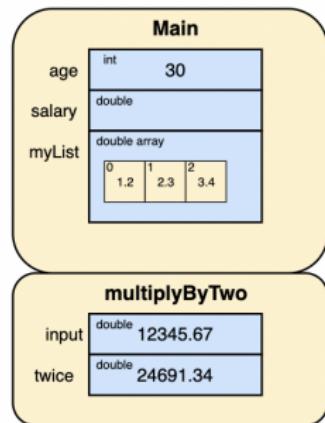
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

## Stack



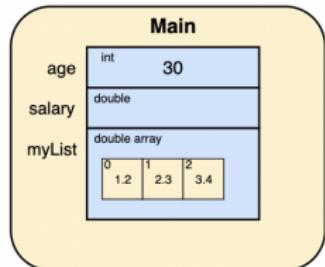
```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

## Stack



```
1 #include <stdio.h>
2
3 double multiplyByTwo (double input) {
4     double twice = input * 2.0;
5     return twice;
6 }
7
8 int main (int argc, char *argv[])
9 {
10    int age = 30;
11    double salary = 12345.67;
12    double myList[3] = {1.2, 2.3, 3.4};
13
14    printf("double your salary is %.3f\n", multiplyByTwo(salary));
15
16    return 0;
17 }
```

## Stack



# Pile et fonctions récursives

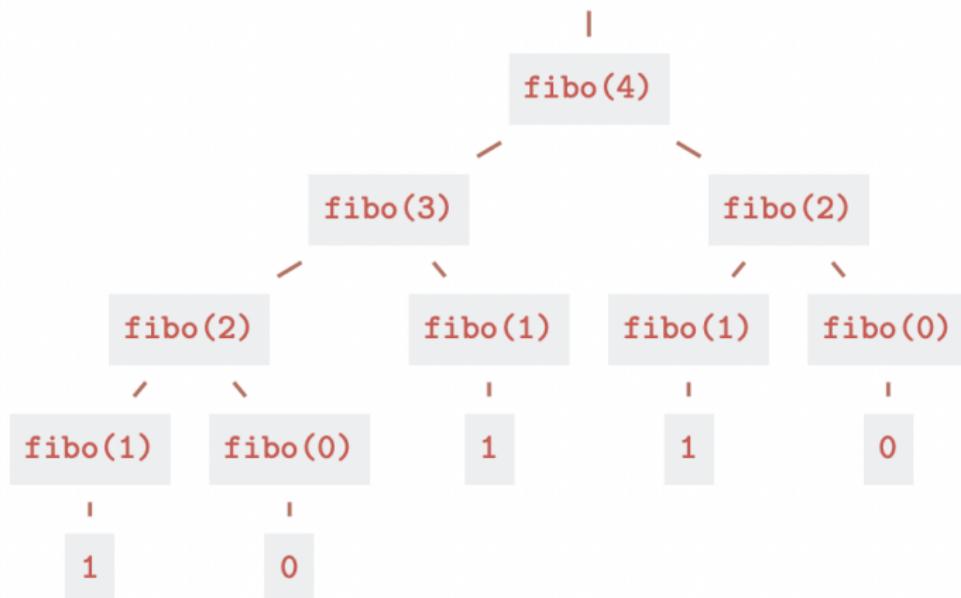
```
int fibo(int a){  
    if (a <= 1)  
        return a;  
    return fibo(a - 1) + fibo(a - 2);  
}  
  
int main(void){  
    int x;  
    x = fibo(4);  
    return 0;  
}
```



## Exercise 2

■ Pouvez-vous représenter son exécution par un arbre des appels?

# Solution



## Fonctions à effet de bord

Une fonction est dite **à effet de bord** si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

```
float double_val(float *x){  
    *x = 2 * (*x);  
    return *x;  
}
```

## Fonctions à effet de bord

Une fonction est dite **à effet de bord** si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

```
float double_val(float *x){  
    *x = 2 * (*x);  
    return *x;  
}
```



La fonction `double_val` est à effet de bord puisqu'elle modifie une zone de la mémoire.

# Exemples

```
char *alloquer(int n) {
    char *res;
    res = (char *)malloc(sizeof(char) * n);
    /* good way
       res = malloc(n * sizeof *res);
       assert(res);
    */
    return res;
}
```

# Exemples

```
char *allouer(int n) {
    char *res;
    res = (char *)malloc(sizeof(char) * n);
    /* good way
       res = malloc(n * sizeof *res);
       assert(res);
    */
    return res;
}
```



Cette fonction est à effet de bord. Elle réserve en effet, par l'appel interne à la fonction `malloc`, une zone de la mémoire, ce qui modifie son état.

```
int h(int a, int b) {
    if (a * b == 0)
        printf("z\n");
    return a - b;
}
```

# Exemples

```
char *alloquer(int n) {
    char *res;
    res = (char *)malloc(sizeof(char) * n);
    /* good way
       res = malloc(n * sizeof *res);
       assert(res);
    */
    return res;
}
```



Cette fonction est à effet de bord. Elle réserve en effet, par l'appel interne à la fonction `malloc`, une zone de la mémoire, ce qui modifie son état.

```
int h(int a, int b) {
    if (a * b == 0)
        printf("z\n");
    return a - b;
}
```

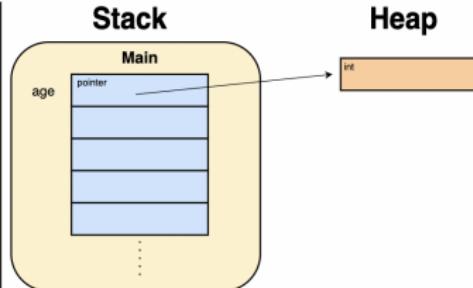


Cette fonction est à effet de bord. En effet, l'appel à `h` provoque un affichage sur la sortie standard, modifiant l'état de la mémoire.

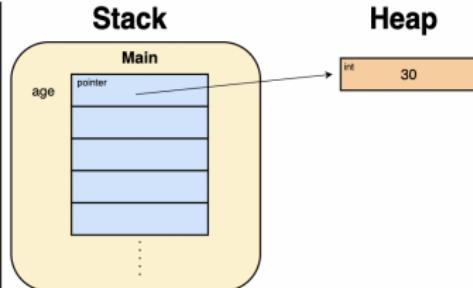
## Heap (en français tas)

En informatique, un **tas** (heap en anglais) est une structure de données de type arbre qui permet de retrouver directement l'élément que l'on veut traiter en priorité.

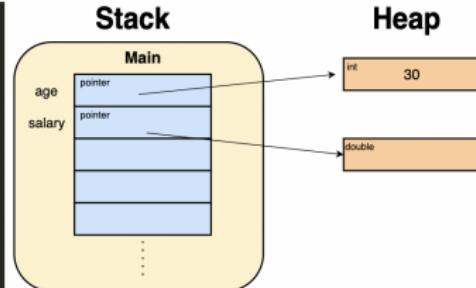
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



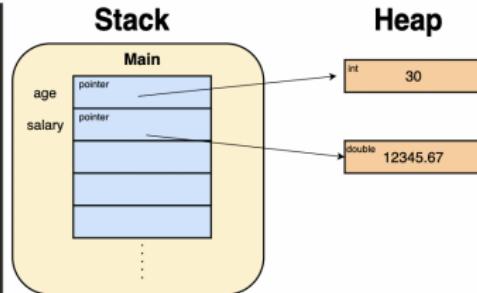
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



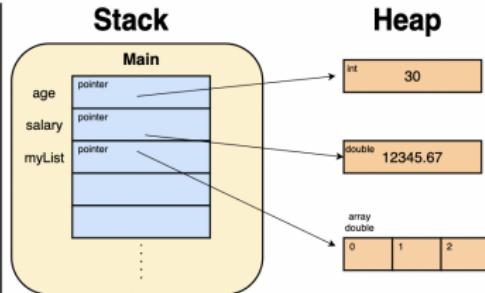
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



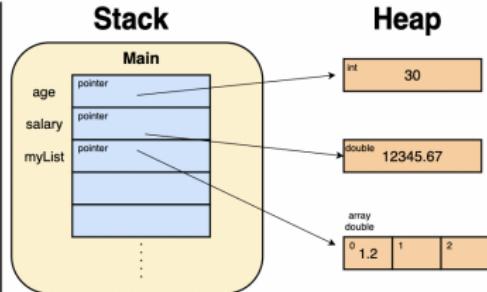
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



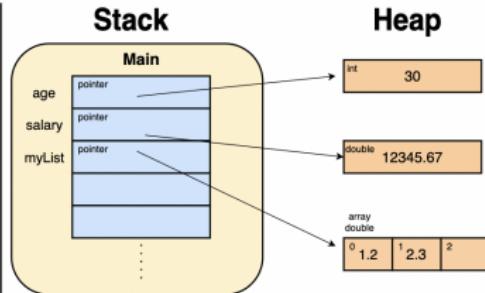
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



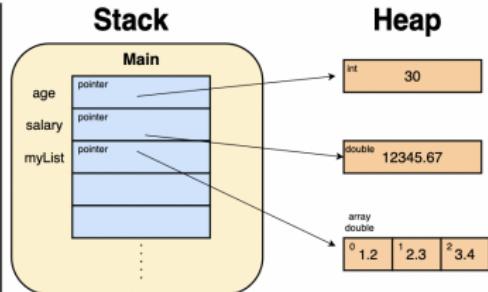
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



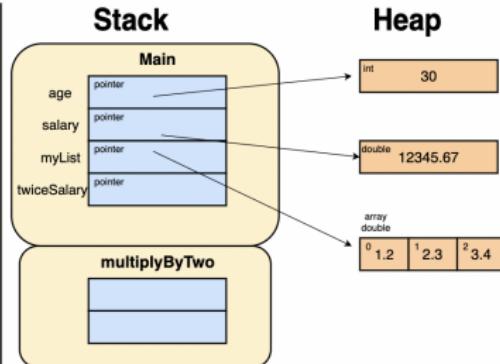
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



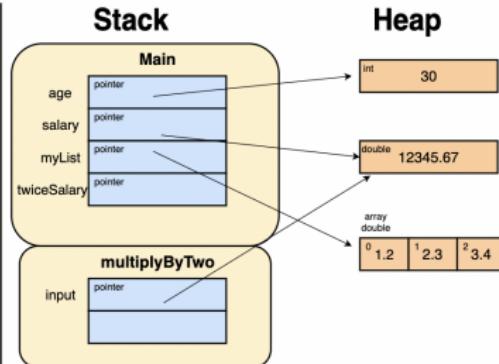
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



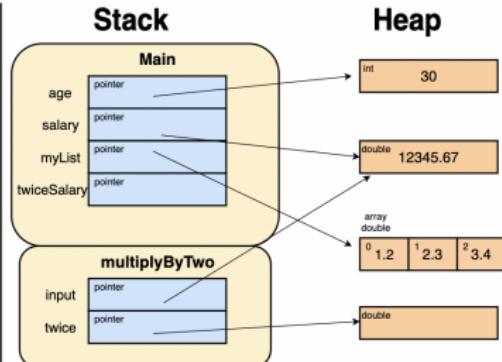
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



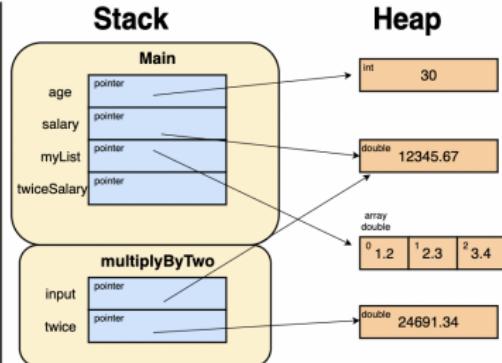
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



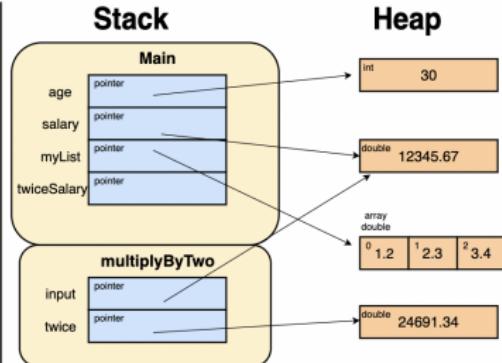
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



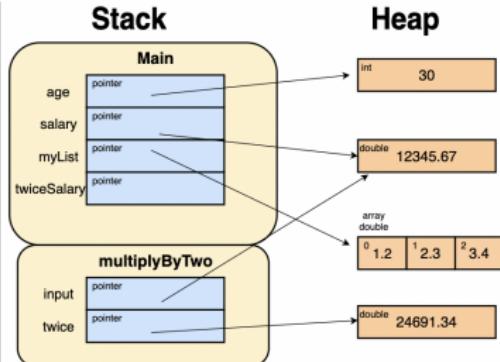
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



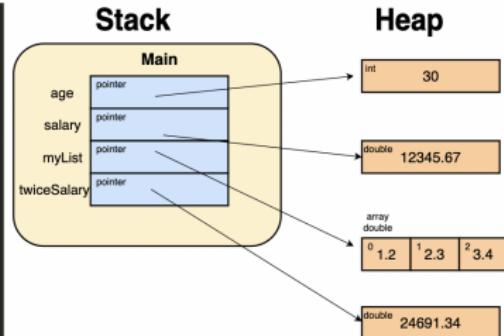
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



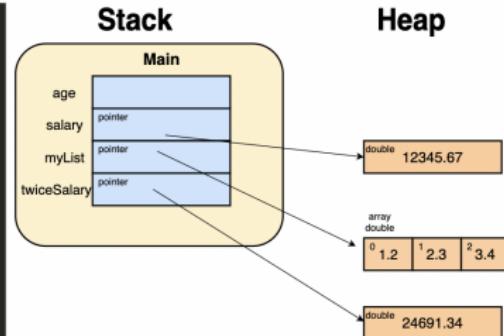
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



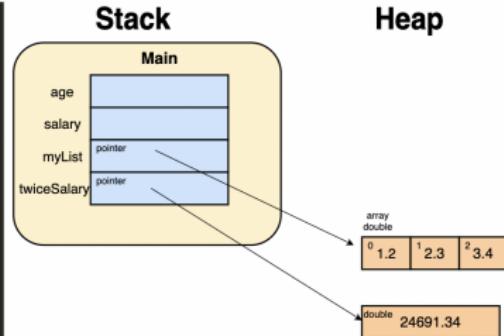
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



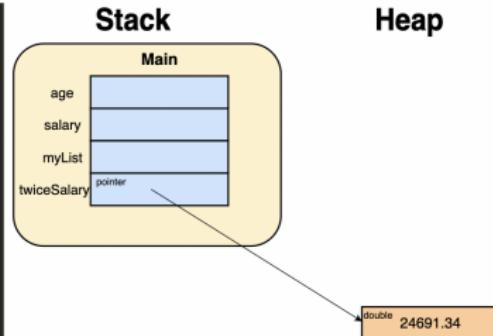
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



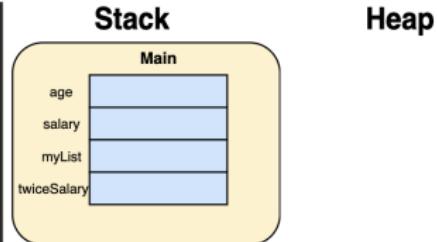
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double *multiplyByTwo (double *input) {
5     double *twice = (double*)malloc(sizeof(double));
6     *twice = *input * 2.0;
7     return twice;
8 }
9
10 int main (int argc, char *argv[])
11 {
12     int *age = (int*)malloc(sizeof(int));
13     *age = 30;
14     double *salary = (double*)malloc(sizeof(double));
15     *salary = 12345.67;
16     double *myList = (double*)malloc(3 * sizeof(double));
17     myList[0] = 1.2;
18     myList[1] = 2.3;
19     myList[2] = 3.4;
20
21     double *twiceSalary = multiplyByTwo(salary);
22
23     printf("double your salary is %.3f\n", *twiceSalary);
24
25     free(age);
26     free(salary);
27     free(myList);
28     free(twiceSalary);
29
30     return 0;
31 }
```



## Références

- ▶ <https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/>
- ▶ <https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap>
- ▶ <https://www.learncpp.com/cpp-tutorial/the-stack-and-the-heap/>

# Allocation dynamique de mémoire

Pour allouer une zone de la mémoire pouvant accueillir N valeurs d'un type T, on utilise l'instruction :

```
ptr = malloc(N * sizeof *T);
```

Équivalent à (MAUVAISE FAÇON) :

```
ptr = (T *) malloc(N * sizeof(T));
```

où ptr est un pointeur pointant sur une zone de la mémoire de type T.

Explications :

- ▶ (T \*) sert à préciser que la zone de la mémoire à allouer est de type T . Cette précision, qui est une coercition, est nécessaire car par défaut malloc renvoie un pointeur sur une zone non typée ( void \* );
- ▶ l'argument sizeof(T) \* N permet de demander à allouer une zone mémoire de taille sizeof(T) \* N octets. Celle-ci pourra donc accueillir N valeurs de type T .



ptr pointe sur le début d'une zone de la mémoire de sizeof(T) \* N octets.

# Allocation dynamique de mémoire



Un test vérifiant que l'allocation est effective (résultat différent de NULL) est nécessaire. Pour faire simple, nous recommandons l'usage de assert (cf. le man et inclure assert.h) :

```
assert(ptr);
```



Une autre façon de le tester est : if (NULL == ptr){ACTION}  
ACTION peut être un exit(EXIT\_FAILURE); si l'on se trouve dans le main ou un  
return NULL; voire un return 0; si l'on se trouve dans une fonction.

# Allocation dynamique de mémoire



Un test vérifiant que l'allocation est effective (résultat différent de NULL) est nécessaire. Pour faire simple, nous recommandons l'usage de assert (cf. le man et inclure assert.h) :

```
assert(ptr);
```



Une autre façon de le tester est : if (NULL == ptr){ACTION}  
ACTION peut être un exit(EXIT\_FAILURE); si l'on se trouve dans le main ou un  
return NULL; voire un return 0; si l'on se trouve dans une fonction.



Libérez la mémoire allouée, à l'aide de free, quand vous n'en avez plus  
l'usage :

```
free(ptr);
```

# Allocation dynamique de mémoire



Un test vérifiant que l'allocation est effective (résultat différent de NULL) est nécessaire. Pour faire simple, nous recommandons l'usage de assert (cf. le man et inclure assert.h) :

```
assert(ptr);
```



Une autre façon de le tester est : if (NULL == ptr){ACTION}  
ACTION peut être un exit(EXIT\_FAILURE); si l'on se trouve dans le main ou un  
return NULL; voire un return 0; si l'on se trouve dans une fonction.



Libérez la mémoire allouée, à l'aide de free, quand vous n'en avez plus  
l'usage :

```
free(ptr);
```



Remettez le pointeur à NULL afin d'éviter de faire des « bêtises » avec :

```
ptr = NULL;
```

# La fonction `calloc`

La fonction :

```
void *calloc(int length, int size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `length * size` octets, tous initialisés avec des 0.

Exemple :

```
long *ptr;  
ptr = (long *) calloc(13, sizeof(long));
```

alloue une zone de la mémoire pouvant accueillir 13 valeurs de type `long`, initialisées à 0 .

# Exercice

## La rue de la RAM 9/XX (slides Farés Belhadj)

~LV/ @ Université Paris 8      3 - Du côté de la mémoire      F. Belhadj — 38/41

## La rue de la RAM 9/XX

### Entraînement (1)

À faire ou à lire (solution dans le slide suivant) : créer une structure pour un point 3D<sup>12</sup>, contenant trois champs flottants x, y et z. Proposer deux différentes fonctions pour remplir un tableau d'éléments de cette structure (n, le nombre d'éléments sera donné) avec des valeurs pseudo-aléatoires<sup>13</sup> comprises dans l'intervalle [0, 1]. L'une des fonctions de remplissage, `void initpoint3dv(point3d_t * p3darray, int n);`, remplit les n éléments de type `point3d_t`, l'autre, `void init3fv(float * triplefloatsarray, int n);`, remplit les  $3 \times n$  flottants du tableau. Pour deux tableaux statiques de `point3d_t`, de même taille et non initialisés, appeler pour chacun une fonction de remplissage différente. Comparer les deux tableaux à l'aide de la fonction `memcmp` (voir le *man*) et afficher le résultat de la comparaison.

---

12. Par exemple, cette structure pourra être nommée : `point3d_t`

13. Utilisez la fonction `rand()` de la `stdlib` (cf. [man 3 rand](#)). Nous utiliserons `srand` avec la même graine, avant chaque appel de la fonction de remplissage, afin d'avoir la même chaîne de nombres pseudo-aléatoires dans les deux cas.

# Listes chaînées

La liste est une structure de donnée permettant de stocker des collections ordonnées d'objets (en général de même type).



## Liste chaînée

Une liste chaînée est un ensemble de nœuds alloués dynamiquement, disposés de telle sorte que chaque nœud contient une valeur et un pointeur. Le pointeur pointe toujours vers le nœuds suivant de la liste. Si le pointeur est NULL, alors il s'agit du dernier nœud de la liste.



Par rapport à un tableau classique :

- ▶ Faciliter les insertions/suppressions.
- ▶ Taille dynamique.

Une liste chaînée l est représentée par un pointeur :

- ▶ pour la liste vide : on utilise la valeur NULL
- ▶ sinon le pointeur désigne une structure ("node") comprenant un élément (la tête de liste) et un nouveau pointeur (la queue de liste).

Une liste chaînée l est représentée par un pointeur :

- ▶ pour la liste vide : on utilise la valeur NULL
- ▶ sinon le pointeur désigne une structure ("node") comprenant un élément (la tête de liste) et un nouveau pointeur (la queue de liste).

Déclaration :

```
1  typedef int element_t;
2  typedef struct node node_t;
3
4  struct node{
5      element_t elem; //it could be called data
6      struct node* next;
7  };
8
9  node_t* add(node_t* queue, element_t elem){
10     node_t* l = malloc(sizeof *l);
11     l->elem = elem; // (*l).elem = l->elem
12     l->next = queue; // (*l).next = l->next
13     return l;
14  };
15
```

## Construction d'une liste chaînée :

```
1  node_t* add(node_t* queue, element_t elem){
2      node_t* l = malloc(sizeof(*l));
3      // equivalent to:
4      // node_t* l = (node_t*) malloc(sizeof(cellule_t));
5      l->elem = elem; // (*l).elem = l->elem is of type
6      element_t
7      l->next = queue; // (*l).next = l->next is of type
8      node_t*
9      return l;
};
```

## Exemple

```
1     node_t* l = NULL;  
2     l = add(l, -3);  
3     l = add(l, 42);  
4     l = add(l, 12);  
5
```



### Exercise

■ Essayer de dessiner étape par étape la création de `node_t* l`.

## Exemple

*l* : *NULL*

## Exemple

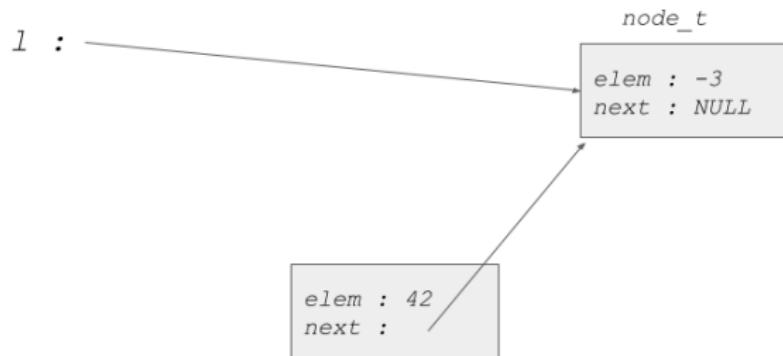
*l* : *NULL*

*node\_t*  
elem : -3  
next : *NULL*

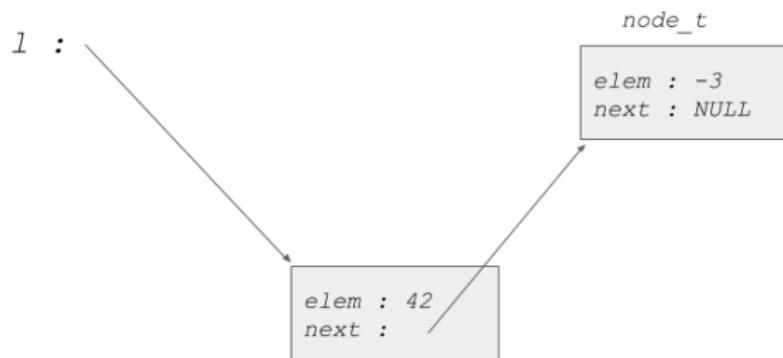
## Exemple



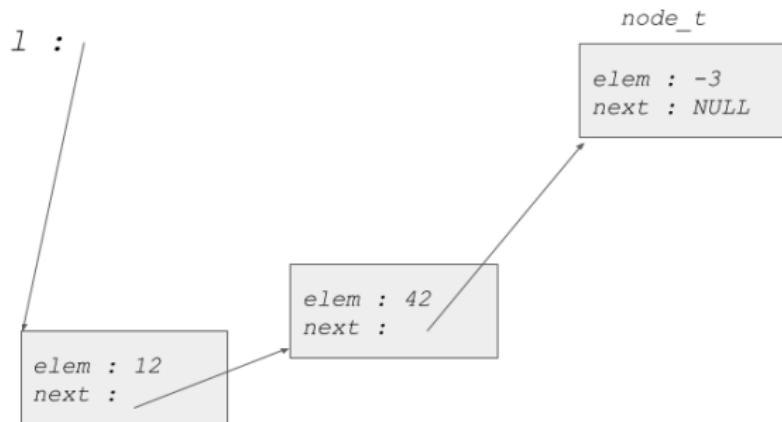
## Exemple



## Exemple



## Exemple



## Liste doublement chaînée (LDC)

**Avantages de la liste doublement chaînée (LDC) par rapport à la liste singulièrement chaînée :**

- ▶ Une LDC peut être parcourue dans les deux sens, en avant et en arrière.
- ▶ L'opération de suppression dans LDC est plus efficace si un pointeur vers le nœud à supprimer est donné.
- ▶ On peut rapidement insérer un nouveau nœud avant un nœud donné.
- ▶ Dans une liste singulièrement chaînée, pour supprimer un nœud, un pointeur vers le nœud précédent est nécessaire. Pour obtenir ce nœud précédent, il faut parfois parcourir la liste. Dans LDC, nous pouvons obtenir le noeud précédent en utilisant le pointeur précédent.

**Désavantages de LDC par rapport à la LC :**

- ▶ Chaque nœud de LDC nécessite un espace supplémentaire pour un pointeur précédent.
- ▶ Toutes les opérations nécessitent le maintien d'un pointeur précédent supplémentaire.

## Liste doublement chaînée (LDC)

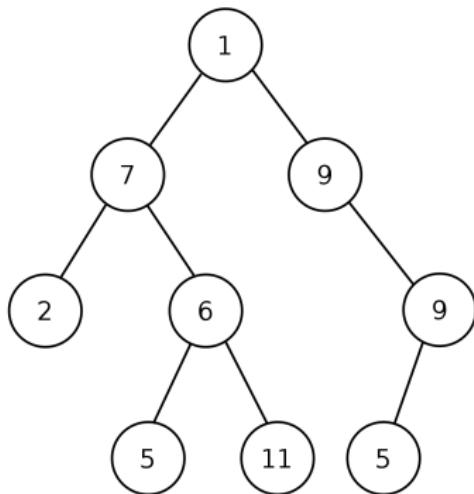
```
1  typedef struct nodeDL nodeDL_t;
2
3  struct nodeDL{
4      element_t elem;
5      struct nodeDL* prev;
6      struct nodeDL* next;
7  };
8
9  typedef struct {
10     nodeDL_t* first;
11     nodeDL_t* last;
12 } headDL_t;
```

# Arbres Binaires



## Arbres binaires

Arbres dont chaque noeud a au plus deux fils, un fils gauche et/ou un fils droite.



## Arbres Binaires - Terminologie

- ▶ **Noeud interne** : Un noeud qui possède au moins un fils.
- ▶ **Feuille** : Un noeud sans aucun fils.
- ▶ **Ascendants** : Le père d'un noeud et ses ascendants.
- ▶ **Descendants** : Les fils d'un noeud et leurs descendants (c-à-d noeuds du sous-arbre)/
- ▶ **Chemin** : d'un noeud à un descendant.
- ▶ **Profondeur** : Nombre d'ascendants d'un noeud.
- ▶ **Hauteur** : D'un arbre vide = 0.  
D'un arbre avec une feuille = 1.  
Sinon, 1+hauteur maximal de sous-arbres.

## Questions

1. Quelle est la hauteur maximal d'un arbre binaire à  $n$  noeuds?
2. Quel est le nombre maximal de noeuds au niveau  $k$  dans un arbre binaire?
3. Quelle est la hauteur minimal d'un arbre binaire à  $n$  noeuds?
4. Quel est le nombre de feuilles d'un arbre binaire complet à  $n$  noeuds?

# Arbres binaires

```
1  typedef int element_t;
2  typedef struct bt_node bt_node_t;
3
4  struct bt_node{
5      element_t data;
6      struct bt_node* childR;
7      struct bt_node* childL;
8  };
9
```

Un arbre binaire est représenté par un pointeur.

- ▶ Si l'arbre est vide : on utilise la valeur NULL.
- ▶ Sinon le pointeur désigne une structure `bt_node_t` comprenant une étiquette (la donnée de la racine) et des pointeurs vers ses deux fils (`childR` et `childL`) qui peuvent être vides.

## Arbres binaires



Si nous avons un arbre non-binaire ( $n > 2$ ), il serait possible de stocker un nombre plus large de fils :

- ▶ soit avec un tableau (si on connaît le nombre maximum de fils),
- ▶ soit avec une liste chaînée.

## Arbres binaires - Fonction de construction

```
1 bt_node_t* newTree(element_t elem, bt_node_t* childR,
2                     bt_node_t* childL){
3     bt_node_t* tree = malloc(sizeof *tree);
4     if (tree == NULL)
5         return NULL;
6     tree->data = elem;
7     tree->childR = childR;
8     tree->childL = childL;
9     return tree;
10 }
11 // Exemple
12 int main(void){
13     bt_node_t* a = newTree(5, newTree(2, NULL, NULL),
14                           newTree(3, NULL, NULL));
15     bt_node_t* b = newTree(8, NULL, a );
16     return 0;
}
```

## Parcours sur les arbres



Pour effectuer une opération (ex. print, rechercher une étiquette), nous devons "visiter" tous les nœud.

L'ordre est important ⇒ trois parcours classiques :

- ▶ Préfixe : un nœud est visité avant ses descendants
- ▶ Suffixe : un nœud est visité après ses descendants
- ▶ Infixe (arbres binaires) : un nœud est visité entre son sous-arbre gauche et son sous-arbre droit.

Dans les trois cas : les nœuds de n'importe quel sous-arbre sont visités consécutivement.

## Parcours Préfixe

```
1 void print_prefix(bt_node_t* tree) {  
2     if (tree == NULL) return;  
3     printf("%d ", tree->data);  
4     printPrefix(tree->childL);  
5     printPrefix(tree->childR);  
6 }  
7
```

?

Qu'imprimera-t-il pour l'exemple précédent?

## Exercices

1. Ecrire la fonction `bt_suffix_print(bt_node_t* a)` qui affiche les nœuds de l'arbre en ordre suffixe.
2. Ecrire la fonction `bt_infix_print(bt_node_t* a)` qui affiche les nœuds de l'arbre en ordre infixé. Quelle est la complexité de ces fonctions (en fonction du nombre  $n$  de nœuds de l'arbres)?
3. `int bt_nb_leaves(bt_node_t* a)` qui compte le nombre de feuilles de l'arbre binaire.
4. `int bt_height(bt_node_t* a)` qui calcule la hauteur de l'arbre binaire.
5. `bt_node_t* bt_copy_tree(bt_node_t* a)` qui renvoie une copie de l'arbre binaire.
6. `void bt_free_tree(bt_node_t** ptree)` qui libère l'ensemble de l'arbre binaire.

# Arbre binaire de recherche 1/2

```
bt_node_t * bt_insert_node(bt_node_t ** pos, bt_node_t * new_node) {
    bt_node_t * previous = *pos;
    *pos = new_node;
    if(previous != NULL) {
        if(previous->data < new_node->data)
            new_node->l_child = previous;
        else
            new_node->r_child = previous;
    }
    return new_node;
}

bt_node_t ** bt_find_ordered_position(bt_node_t ** ptree, element_t elem) {
    if(*ptree == NULL)
        return ptree;
    if(elem < (*ptree)->data)
        return bt_find_ordered_position(&((*ptree)->l_child), elem);
    return bt_find_ordered_position(&((*ptree)->r_child), elem); /* else */
}
```

?

Utilisez ces deux fonctions, ainsi que le fonction `bt_infix_print`, pour remplir un arbre vide de valeurs pseudo-aléatoires et l'afficher de manière à ce que les valeurs soient données dans un ordre croissant.

## Arbre binaire de recherche 2/2

Sur le modèle du comparatif entre *liste chaînée* et *vecteur* fait précédemment<sup>5</sup> :

- ▶ Compléter le comparatif pour pouvoir réaliser une insertion ordonnée dans un vecteur<sup>6</sup>;
- ▶ Ajouter l'arbre binaire sous la forme d'une bibliothèque (fichier .h et .c) et proposer, dans le fichier test\_them.c, un test de performance équivalent à celui du vecteur, mais sans vérification de l'ordre des éléments;
- ▶ Essayer d'ajouter une vérification de l'ordre des éléments insérés dans l'arbre binaire (*i.e.* avoir un ordre croissant selon un parcours infixé);
- ▶ Essayer d'améliorer l'insertion dans le vecteur en modifiant la recherche de position : modifier find\_ordered\_pos\_in\_vec pour qu'elle utilise une recherche dichotomique à la place d'une recherche naïve.

---

5. Archive du TP noté :[https://expreg.org/amsi/C/PA2223S1/dl/pa\\_linkedList\\_n\\_vector-0.10.tgz](https://expreg.org/amsi/C/PA2223S1/dl/pa_linkedList_n_vector-0.10.tgz)

6. Si vous n'avez pas réussi à compléter la partie concernant la liste chaînée, mettez-la de côté en supprimant son utilisation dans le fichier test\_them.c.

~LIV/

**L**icence **I**nformatique et **V**idéoludisme  
Université Paris 8