Types Abstraits & Bases de la POO
II - Le langage C

- - - - - - - - - -
1 – Généralités
2 – types construits
3 – Modularité
- - - - - - - - - -

ENSMA A3-S5 - période A

2021-2022

M. Richard
richardm@ensma.fr

Sommaire

I - Généralités

1 – Bref historique
2 – Principes
3 – Syntaxe d'un programme C
4 – Types prédéfinis
5 – Les littéraux
6 – Les opérateurs
7 – Les variables
8 – Entrées/Sorties standard
9 – ↬ Cours/TD I-Exercice I
10 – Structures de contrôle
11 – ↬ Cours/TD I-Exercice II

Langage C :

- Apparu en 1972 et créé par Dennis Richie et Ken Thompson
- Première définition du C classique publiée par Brian Kernighan et Dennis Richie dans *The C Programming language*
- Le C devient de plus en plus populaire et l'ANSI décide de normaliser ce langage en 1983. la première norme verra le jour en 1989 : ANSI C
  - norme reprise en 1990 par l'ISO sans aucun changement
- la norme du langage C évoluera mais cette première norme fait souvent office de référence

Le langage C est un langage *compilé* ; la *compilation* d'un programme C comporte 4 phases :

1. *Préprocesseur :* phase d'analyse et de transformation uniquement textuelle d'un fichier `.c` ou `.h`
2. *Compilation :* phase de traduction du fichier généré à l'étape précédente en assembleur
3. *Assemblage :* phase de transformation du code assembleur précédent en fichier binaire, portant l'extension `.o`
   - on parle de fichier *objet*
4. *Édition des liens :* phase nécessaire lorsque le programme compilé fait appel (i.e. utilise) des bibliothèques externes et/ou est décomposé en différents modules

En plus des fichiers `.c`, `.h` ou `.o` vous rencontrerez également parfois :

- des fichiers `.s` correspondant au code assembleur
- des fichiers `.a` correspondant à des librairies pré-compilées

↪ La base

- Le langage C est sensible à la casse
- On distingue différents groupes de composants dans un programme C :
  - les identificateurs
  - les mots-clefs
  - les opérateurs
- les commentaires (mono ou multi-lignes) sont placés entre /* et */
  - possibilité d'utiliser // pour un commentaire mono-ligne (non ANSI)
- le point d'entrée d'un programme C est une fonction particulière : `main`

```
1  int main(int argc, char *argv []){
2    ...
3    return 0;
4  }
```

  - les paramètres de cette fonction permettent le passage d'information(s) lors du lancement du programme.
  - la fonction `main` retourne toujours une valeur :
    - la valeur 0 si tout se passe bien (i.e. pas d'erreur)
    - le code d'erreur sinon, à l'aide de l'instruction `exit` quand l'erreur survient

↪ Les identificateurs :

- permettent de donner un nom aux différentes entités d'un programme :
  - variable, fonction
  - type défini
  - étiquette

- c'est une suite de caractères telle que :
  - le premier caractère est obligatoirement une lettre
  - le caractère _ est considéré comme une lettre
    - par convention, on utilisera pas celui-ci comme premier caractère d'un identificateur car généralement employé pour définir les variables globales de l'environnement
  - pas de caractère accentué

- attention au choix de vos identificateurs pour améliorer la lisibilité du code
  - longueur
  - convention de nommage pour les différentes entités

↪ Les mots-clefs

Les mots-clefs (ou réservés) du langage peuvent être classés par catégorie :

- spécificateurs de stockage :
  `auto register static extern typedef`
- spécificateurs de type :
  `char double enum float int long short signed struct union unsigned void`
- qualificateurs de type :
  `const volatile`
- instruction de contrôle :
  `break case continue default do else for goto if switch while`
- divers :
  `return sizeof`

↪ Structure globale

En C, toute entité doit-être déclarée avant de pouvoir être utilisée ; de manière naturelle la structure globale d'un programme C devra respecter :

```
[ directives au préprocesseur ]

[ déclarations de variables globales ]

[ définition des fonctions secondaires ]

int main(int argc, char *argv[])
{
    déclarations de variables locales
    instructions
}
```

↪ Type entier :

- `int` désigne le type entier classique ...
  - définition dépendante de l'architecture !
  - représenté par un mot "naturel" de la machine (i.e. 16, 32 ou 64 bits)

- le type `char` modélise un entier représenté sur 8 bits

Type entier :	Modélise :
`char`	Entier sur 8 bits
`short`	Entier sur 16 bits
`int`	Représentation entière la plus efficace (mot machine)
`long` ou `long int`	Entier sur 32 ou 64 bits

- modificateurs de précision : `short` ou `long`
- modificateurs de représentation : `unsigned`
- exemple :

```
unsigned int entier non signé de la taille du mot machine: (0..2^n − 1)
unsigned seul est un raccourci pour unsigned int
unsigned char code caractère non signé (correspond au type character Ada): 0..255
unsigned short (0..65535) unsigned long (0..2^32 − 1)
```

↪ Type réel :

- float, `double` et `long double` permettent de modéliser des nombres en virgule flottante

Type réel :	Modélise :
`float`	Réel simple précision (32 bits)
`double`	Réel double précision (64 bits)
`long double`	Réel très grande précision (128 bits)

- utilise la représentation IEEE754

↪ Autre . . . :

- Type “tout”

Type :	Modélise :
void	Tout ou n’importe quoi . . .

  - on y reviendra . . .

- pas de type chaîne de caractères, ni de type booléen

↪ Littéraux caractères :

- caractère : cotes
  - 'A'
- n'importe quel caractère
- séquences d'échappement

\n	Saut de ligne
\t	Tabulation
\b	Epace arrière
\r	retour chariot
\f	Saut de page
\a	Sonnerie
\\	\
\'	'
\"	"

- représentation des caractères
  - entier sur un octet : le type `char`
  - nombre représentant le caractère de manière interne
    - ASCII
  - possibilité d'utiliser :
    - le code octal : `\code-octal`
    - le code hexadécimal : `\xcode-hexa`

↪ Littéraux chaînes de caractères :

Représentation des chaînes de caractères :

- chaine de caractères : guillemets
  - `"coucou"`
- suite finie de caractères
- longueur quelconque
- codage interne :
  - caractères rangés de manière contiguë
  - ajout d'un caractère nul après le dernier caractère utile
- un chaîne est donc référencée par l'adresse de la mémoire ou a été "stocké" le premier caractère de la chaîne
- en bref :
  - pas de type chaîne prédéfini
  - utilisation d'un tableau de caractères

↪ Littéraux entiers :

- définis de manière classique et sans signe
- base utilisée : décimale par défaut
- autre base possible :
  - octale (préfixé par `0`)
  - hexadécimale (préfixé par `0X` ou `0x`)
- type choisi par le compilateur :
  - plus petit type dans lequel la constante est représentable
  - en suffixant la constante il est possible de forcer le compilateur :
    - `U, u` : force le type unsigned
    - `L, l` : force le type long
    - possibilité de combiner les deux

- Exemples :

constante	type
`1234`	`int`
`02322`	`int /* octal */`
`0x4D2`	`int /* hexadécimal */`
`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

↪ Littéraux réels :

- constitués par :
  - suite de chiffres décimaux
  - un point
  - suite de chiffres décimaux
  - E ou e
  - signe + ou -
  - suite de chiffres décimaux

- règle d'écriture :
  - on peut ne pas renseigner :
    - la partie entière ou la partie décimale
    - le point ou l'exposant
  - exemple :
    - `.12E7 ; 53.e22 ; 48532. ; 10e6`

- type choisi par le compilateur :
  - `double` par défaut
  - possibilité de suffixer pour forcer le compilateur :
    - F ou f : `float`
    - L ou l : `long double`

↪ Opérateurs classiques :

- affectation : `=`
  - attention conversion implicite vers le type du membre gauche de l'expression
- arithmétique : `+ - * / %`
  - pas d'opérateur pour la division entière. Réalisée si les 2 opérandes sont de type entier
  - pas d'opérateur pour la puissance → utiliser la librairie `math.h`
- relationnels : `> >= < <= == !=`
  - attention, pas de type booléen !
    - ces opérateur retournent une valeur de type `int` →1 si relation vrai, 0 sinon
  - attention à ne pas confondre == avec =
- logiques : `&& || !`
  - évaluation de gauche à droite et s'interrompt dès que le résultat final est obtenu
- binaires : `& | ^ ~ << >>`
  - l'opérateur ~ change la valeur de chaque bit d'un entier (complément à 1)
  - les opérateurs << et >> réalise un décalage à droite ou à gauche soit une multiplication ou une division entière par une puissance de 2
    - attention au type choisi . . .

↪ Opérateurs avancés :

- affectation composée : `+= -= *= /= %= &= ^= |= <<= >>=`
  - fonctionnement : soit l'opérateur ope= alors :
    `expr_1 ope= expr_2` est équivalente à `expr_1 = expr_1 ope expr_2`
- incrémentation et décrémentation : `++ --`
  - s'utilise en opérateur suffixe ou en préfixe
    - la variable est toujours incrémentée ou décrémentée
    - en version suffixe, c'est la valeur avant opération qui est retournée par l'opérateur
- virgule : permet de construire une expression comme une suite d'expression séparée par l'opérateur `,`
  - évaluation de gauche à droite
  - opérateur délicat car peut prêter à confusion dans le cas de passage de paramètres d'une fonction
  - exemple :
    `res = ((val = 5),(val *2));` ici, `res` vaut 10
- condition ternaire : `cond ? exp_si_vrai : exp_si_faux`
- cast (i.e. changement de type) : `(type_dest) var`
  - modifie explicitement le type de la valeur de `var` en `type_dest`
- adresse : `&var`
  - permet d'obtenir l'adresse mémoire de la variable `var`

↪ Déclaration et portée :

- syntaxe de déclaration :
  `[Op_modif] type id [=valeur] ;`
- portée en fonction du lieu de déclaration :
  - si en dehors de toutes fonctions :
    - variable globale
  - si au début d'un bloc (i.e. délimité par { et })
    - variable locale ; visible dans le bloc et les sous blocs
  - en tant qu'argument de fonction
    - variable formelle, i.e. globale à la fonction
- exemple :

```
/* Déclaration de variables ... */
int numerique; /*un entier*/
float x,y,z = .5f; /*3 réels initialisés à 0.5*/
char c = 'd'; /*un caractère initialisé à d */
```

↪ Durée de vie et optimisation :

- Variables *statiques* :
  - `static`
  - permanente ; existe durant toute la vie du programme
  - les variables globales sont toujours statiques
- variables *registres*
  - `register`
  - à utiliser si accès très fréquent durant l'exécution
  - peuvent ainsi être placées dans des registres du CPU
- Variables *constantes* et volatiles
  - `const`
  - pas d'évolution de la variable au cours de l'exécution du programme
  - vérification à la compilation

↪ Initialisation :

- pas de restriction sur la valeur utilisée
- toujours initialiser une variable avant sa première utilisation

Les fonctions d'entrées/sorties standard sont définies dans la librairie standard `stdio.h`. Il est donc nécessaire d'inclure cette bibliothèque dans un programme utilisant ces fonctionnalités via une directive préprocesseur :
`#include <stdio.h>`

↪ Sortie standard : `printf`

- fonction d'impression écran formatée
- syntaxe : `printf("chaine_format",var_1, ..., var_n);`
  - `chaine_format` contient le texte à afficher et les *indicateurs de format* insérés aux différents endroits de la chaîne afin de permettre l'impression des valeurs de `var_1, ..., var_n` dans le bon format.
- indicateurs de format : préfixés par le caractère % suivi d'un caractère désignant le format voulu
- possibilité d'ajouter des options de formatage :
  - taille de l'espace d'impression : valeur placée entre le % et le caractère spécifiant le nombre de caractères à réserver pour l'impression
    - exemple : `%6d` : 6 caractère réservés pour afficher la valeur de l'entier
  - nombre de chiffres après la virgule dans le cas d'un réel : valeur, préfixée par un point, placée avant le caractère
    - exemple : `%6.2f` affiche la valeur du réel sur 6 caractère et avec 2 chiffres après la virgule

↪ Indicateurs de format d'affichage :

Indicateur	formatage (i.e. type correspondant)
%d	`int`
%ld	`long int`
%u	`unsigned int`
%lu	`unsigned long int`
%o	`unsigned int` (valeur en octal)
%x	`unsigned int` (valeur en Hexadécimal)
%f	`double`
%lf	`long double`
%e	`double` (notation exponentielle)
%g	`double` (affichage le plus court entre %f et %e)
%c	`unsigned char` (caractère)
%s	`char*` (chaîne de caractères)
...	

↪ Entrée standard : `scanf`

- fonction permettant de récupérer les données saisies au clavier et de les stocker (avec le bon type) dans les variables passées en paramètre de celle-ci
- syntaxe : `scanf("indicateurs_type",adr_var_1,...,adr_var_n);`
  - `indicateurs_type` correspond aux différents types de valeurs attendues, séparés par des virgules. Même construction que les indicateurs de format (i.e. % suivi d'une lettre)
  - les paramètres `adr_var_1,...,adr_var_n` passées à la fonction doivent correspondre à des *adresses* !
    - pour une variable non pointeur, on utilise l'opérateur &
  - il doit y avoir autant de paramètres que d'indicateurs de type
- exemple :

```
1    #include <stdio.h>
2    int main (int argc, char *argv[]) {
3      int val;
4      printf ("Donner un entier :");
5      scanf("%d",&val);
6      printf("la valeur saisie est : %d", val);
7      return 0;
8    }
```

↪ Indicateurs de format de saisie :

Indicateur	type correspondant
%d	int
%hd	short int
%ld	long int
%u	unsigned int
%hu	unsigned short int
%lu	unsigned long int
%o	int (octal)
%ho	short int (otcal)
%lo	long int (octal)
%x	int (hexadécimal)
%hx	short int (hexadécimal)
%lx	long int (hexadécimal)
%f	float
%lf	double
%Lf	long double
%e	float (écriture exponentielle)
%le	double (écriture exponentielle)
%Le	long double (écriture exponentielle)
%g	float (écriture exponentielle ou classique)
%lg	double (écriture exponentielle ou classique)
%Lg	long double (écriture exponentielle ou classique)
%c	char
%s	char* (chaîne)

↪ Affichage et saisie d'un caractère :

- saisie : utiliser la fonction `int getchar()`
  - retourne un entier correspondant au code du caractère
    - choisir un type `int` pour récupérer la sortie
  - la constante `EOF` est définie dans la bibliothèque et vaut généralement -1
    - correspond à la fin de la lecture ou à une erreur

- affichage : utiliser la fonction `putchar(char c)`
  - affiche le caractère passé en paramètre

Cours/TD I - Exercice 1 : pour se mettre en jambes . . .

1. Ouvrir le document cours-td1-EnvDev.pdf et réaliser la mise en place et le test de l'environnement de développement
2. Réaliser les programmes suivants :
   - 2.1 Écrire un programme carre.c qui demande un entier et affiche son carré.
   - 2.2 Écrire un programme diagonale.c qui demande le côté d'un carré et affiche sa diagonale [a].

a. Il faut utiliser la fonction "racine carrée" qui provient de la bibliothèque math.h

↪ Branchement conditionnel :

- Syntaxe :

```
...
if (condition) {
    /* bloc d'instructions du then */
  } else if (condition2){
    /* bloc d'instruction du then */
  } else
    /* bloc d'instruction du else */
}
```

↪ Branchement multiple :

- Syntaxe :

```
Switch (variable){
  case valeur :
    /* ensemble d'instructions */
  break;
  case valeur2 :
    /* ensemble d'instructions */
  break;
  ...
  ...
  default :
    /* ensemble d'instructions par défaut */
  break;
}
```

↪ itération : `while`

- Syntaxe :

```
while (condition) {
  /* bloc d'instructions si condition vérifiée */
}
```

↪ itération : `do while`

- Syntaxe :

```
do {
  /* bloc d'instructions si condition vérifiée */
  /* bloc toujours exécuté au moins  une fois ! */
} while (condition)
```

↪ itération : `for`

- Syntaxe :

```
for (expr1;expr2;expr3){
  /* bloc d'instructions à itérer */
}
```

  - `expr1` exécuté 1 seule fois avant le début de l'itération
  - `expr2` condition d'arrêt de l'itération
  - `expr3` exécuté à chaque itération, avant l'évaluation de `expr2` pour l'itération suivante

↪ **Branchement non conditionnel `break` :**

- cette instruction peut-être utilisée au sein de toute structure itérative (à éviter tout de même ...)
- interrompt l'itération et "saute" directement à l'instruction suivant immédiatement la boucle

↪ **Branchement non conditionnel `continue` :**

- cette instruction peut-être utilisée au sein de toute structure itérative (à éviter tout de même ...)
- stop l'itération en cours (i.e. les instructions suivant celle-ci ne sont pas exécutées)
- "saute" directement à l'évaluation de la condition pour l'itération suivante

↪ **Branchement non conditionnel `goto` :**

- permet d'effectuer un "saut" jusqu'à l'étiquette demandée à la suite de l'instruction
- cette instruction est À PROSCRIRE de tout programme ! ! !

Cours/TD I - Exercice 2 : conditions et itérations . . .

1. Écrire un programme poly_2.c qui demande trois réels a, b, c et indique le nombre et les valeurs des racines de l'équation $ax^2 + bx + c = 0$
2. Écrire un programme fibo.c qui demande un entier et affiche le n$^{\text{ième}}$ terme de la suite de Fibonacci. ($f0 = 0, f_1 = 1, f_n = f_n - 1 + f_n - 2$)
3. Écrire un programme reverse.c qui demande un entier n, et l'affiche en inversant l'ordre des chiffres. On utilisera la division entière et le modulo [a].

a. Dans la bibliothèque stdlib

II - Types construits

1 – Les tableaux
2 – Articles ou Structures
3 – Unions
4 – Énumérations
5 – Identificateur de type

↪ Cas général

- Définition : ensemble *fini* d'éléments de même type stockés en mémoire de manière contiguë
- Syntaxe de déclaration : `type_elem nom_tab[nb_elem]`
  - il est recommandé de déclarer `nb_elem` comme constante (directive préprocesseur)
- l'accès aux éléments (i.e. indexation) se fait via l'opérateur `[]`
  - attention, l'indexation débute à 0
- initialisation possible par un agrégat par valeur : ensemble de valeurs séparées par une virgule et entre accolades
  - si nombre de valeurs de l'agrégat inférieur au nombre d'éléments du tableau, seuls les premiers seront initialisés
- *un tableau est un pointeur* sur le premier élément de celui-ci (Cf.plus loin).
  - il n'est donc pas possible d'appliquer une opération sur la globalité du tableau (copie par exemple)
    - on effectuera l'opération sur chacun des membres
- exemple :

```
#define TAILLE_TAB 5
...
int mon_tab[TAILLE_TAB] = {12,7,35,14,58};
mon_tab[2] = 44;
```

↪ Chaîne de caractères

- le tableau de caractères correspond à la modélisation d'une chaîne de caractères
- fonctionnement général identique aux tableaux classiques, sauf :
  - lors de l'affectation d'une chaine à un tableau de caractères, le compilateur rajoute le caractère nul \0 à la fin
    - nécessite donc de prévoir un élément supplémentaire dans la taille de la chaîne
- exemple :

```
#define TAILLE_TAB 5
#define TAILLE_CHAINE 8
...
int mon_tab[TAILLE_TAB] = {12,7,35,14,58};
char chaine[TAILLE_CHAINE] = "exemple";
mon_tab[2] = 44;
```

↪ Taille implicite

- lors de la déclaration il est possible de ne pas spécifier le nombre d'éléments max du tableau
  - la taille sera alors obtenue implicitement par la taille de l'agrégat d'initialisation
- exemple : `char chaine[] = "exemple";`

↪ Tableaux multi-dimensions

- possibilité de définir des tableaux multi-dimension
  - attention, n'est en fait qu'un tableau unidimensionnel dont chaque élément est lui même un tableau mono ou multi dimension
- syntaxe de déclaration :
  `type_elem nom_tab[nb_elem_dim1][nb_elem_dim2]...`
- accès à une dimension ou à un élément : opérateur `[ ][ ]...`
- initialisation : on utilise une composition récursive d'agrégats
- exemple :

```
#define TAILLE_DIM1 3
#define TAILLE_DIM2 5
...
int mon_tab[TAILLE_DIM1][TAILLE_DIM2] = {{12,7,35,14,58},{24,59,44,7,1},{2,27,5,10,37}};
mon_tab[2][0] = 44;
```

↪ Définition

- un article, ou structure, est une composition (produit cartésien) de différentes variables (appelées *champs* ou *membres*) pouvant être de type différent
- les différents champs ne sont pas obligatoirement stockés de manière contiguë en mémoire

↪ Utilisation

- on distingue la déclaration d'un type article de la déclaration d'une variable de ce nouveau type
  - la grammaire du langage permet néanmoins de le faire en une seule fois
- syntaxe de déclaration d'un article :

```
struct mon_article {
     type champ1;
     type champ2;
     ...
 };
```

↪ Utilisation

- syntaxe de déclaration d'une variable de ce type :
  `struct mon_article var;`
- accès aux membres de l'article : on utilisera la notation pointée (i.e. l'opérateur .)
- initialisation et affectation de littéraux :
  - utilisation d'un agrégat par valeurs (comme pour les tableaux)
  - champ par champ
- affectation d'une variable de même type : contrairement au cas du tableau, l'opérateur d'affectation est autorisé entre 2 variables
- exemple :

```
struct mon_article {
  int ch1;
  float ch2;
};
struct mon_article art1, art2;
art1 = {44 , .55};
art1.ch1 = 444;
art2 = art1;
```

↪ Définition

- une union se définit comme un ensemble de variables de types différents mais stocker *dans la même zone mémoire*
  - correspond à un *OU* des différents membres de l'union
  - permet de modéliser une notion à l'aide de types différents
  - si les types des champs d'une union sont différents, l'espace mémoire réservé correspond à la plus grande taille

↪ Utilisation

- les déclaration et opérations pour les unions fonctionnent de la même manière que les articles

↪ Définition & utilisation

- Définition d'un nouveau type par extension, i.e. par la liste des valeurs possibles
- syntaxe de déclaration : `enum mon_enum {val1, val2, ..., valn}`
- techniquement, les énumérations sont implémentées comme des `int`
  - les valeurs constituants l'énumération sont codées par des entiers de 0 à `n-1`
- possibilité de modifier la valeur de l'entier associée à la valeur de l'énumération au moment de la déclaration
- déclaration d'une variable : `enum mon_enum var_enum;`
- affectation d'une valeur : les valeurs de l'énumération peuvent être utilisées comme des littéraux classiques
- exemple :

```
enum jour {lundi, mardi, mercredi, jeudi , vendredi, samedi, dimanche};
enum mois {janvier=1, fevrier=2, mars=3, avril=4, mai=5, juin=6, juillet=7, aout=8,
      septembre=9, octobre=10, novembre=11, decembre=12};
enum jour lejour;
lejour = lundi;
```

- `typedef` : permet d'associer un identifiant à un type construit
- syntaxe : `typedef type identifiant;`
- ne pas hésiter à utiliser cet outils pour alléger grandement l'écriture des programmes et donc à en améliorer la lisibilité
- exemple :

```
struct mon_article {
  int ch1;
  float ch2;
};
typedef struct mon_article Tarticle;
...
Tarticle var ;
```

- ou encore, en une seule fois :

```
typedef enum jour {lundi, mardi, mercredi, jeudi , vendredi, samedi, dimanche} T_jour;
...
T_jour var;
```

III - Modularité

1 – Les principes
2 – Les sous-programmes
3 – ↬ Cours/TD I-Exercice III
4 – Les modules
5 – ↬ Cours/TD I-Exercice IV

↪ Trois grands principes . . .

La réalisation d'un code, quelque soit le langage choisie, doit, au maximum, respecter les 3 principes de base ci-dessous :

- *Abstraction des constantes littérales :*
  - éviter, autant que possible, l'utilisation de constantes littérales dans le corps d'un programme et d'un sous-programme
  - utiliser les *constantes symboliques* définies à l'aide de la directive préprocesseur `#define`
- *Factorisation du code :*
  - éviter *absolument* la redondance (ou duplication) de code
  - éviter, autant que possible, les codes longs
  - utiliser, sans modération, les *sous-programmes*
- *Découpage du code :*
  - répartir le code en plusieurs fichiers et en faire des *modules*
    - améliore grandement la lisibilité
    - permet et augmente grandement la réutilisation de code

↪ Généralités

- Il ne doit y avoir qu'une seule fonction `main` dans un programme C : c'est le programme principal.
- Il par contre possible de définir des sous-programme qui, en C, sont *exclusivement des fonctions*.

↪ Déclaration & définition

- déclaration en utilisant la syntaxe de signature d'une fonction :
  `type_de_retour nom_fonction (paramètres);`
  - si aucun paramètre, mettre des parenthèses vides
  - si aucun type de retour (i.e. une procédure), utiliser le type `void` pour le type de retour (tout sous-programme est une fonction)
- définition du corps du sous-programme :

```
type_de_retour nom_fonction (paramètres) {
    définitions (avec ou sans initialisation) des variables locales
    corps
}
```

↪ Passage de paramètres

- passage *par valeur* par défaut
- précisément, les paramètres sont en mode IN

```
void echange(int a, int b) {
  int c=a;
  a=b;
  b=c;
}
...
echange(x,y);
/* les valeurs n'ont pas été échangées ...*/
```

- Pour obtenir un mode de passage en OUT ou IN OUT :
  - il faut passer les adresses des paramètres (i.e. un pointeur) en utilisant l'opérateur *
  - ainsi, ce sont les adresses qui sont recopiée ; par indirection, les valeurs des variables "pointées" deviennent modifiables

```
void echange(int *a, int *b) {
  int c=*a;
  *a=*b;
  *b=*c;
}
...
echange(&x,&y);
/* les valeurs ont été échangées*/
```

↪ Passage de paramètres : cas des tableaux

- Attention, les tableaux sont déjà des pointeurs ! !
- inutile donc de passer un pointeur sur un paramètre de type tableau
  - sauf si l'on souhaite modifier l'adresse du tableau ... mais ça, c'est pour plus tard ...

```
void inverser(int T[], int tailleT) {
  int i,tmp;
  for (i=0;i<tailleT/2;i++) {
    tmp=T[i];
    T[i]=T[tailleT-1-i];
    T[tailleT-1-i]=tmp;
}
...
inverser(T,10);
/* le tableau est inversé */
```

↪ Passage de paramètres : aide du compilateur

- Le mot clé `const` peut être utilisé sur les paramètres d'une fonction
  - améliore la lisibilité
  - permet au compilateur de vérifier s'il n'y a pas de modification du paramètre dans la fonction
  - permet d'imposer le passage d'un tableau en mode `IN`

```
void f(const int T[], const int tailleT) {
  /* la modification de T et/ou de tailleT sera détectée par le compilateur */
}
```

  - certains compilateurs ne donnent qu'un Warning en cas de modification d'un paramètre constant dans le corps du sous-programme

↪ Récursivité

- un sous-programme, ou algorithme, est dit récursif lorsque celui-ci :
  - est défini en faisant référence à lui-même
  - contient une condition de terminaison
- ce type d'algorithme peut paraitre plus facile à écrire et à comprendre
- mais attention à la condition d'arrêt.
  - la condition d'arrêt marque la fin de "l'empilement des appels" et le début du "dépilement"
- fréquemment utilisée pour mettre en place une approche "diviser pour régner" :
  - divise le problème en sous-problèmes
  - l'algorithme est appelé sur chaque sous-problème
  - le résultat est obtenu en combinant les résultats des appels.

↪ Sous-programmes & variables

On peut distinguer 2 catégories de variables :

- *variables permanentes (ou statiques) :*
  - occupent le même emplacement mémoire durant toute la vie du programme
    - allouées dans la zone mémoire appelée *segment de données*
  - initialisées par le compilateur
  - les variables permanentes sont soit :
    - des variables globales, i.e. déclarée en dehors de toute fonction
    - des variables locales qualifiées à l'aide du mot-clef `static` ; dans ce cas, la valeur de la variable est conservée d'un appel de la fonction à l'autre
- *variables temporaires :*
  - espace mémoire allouée dynamiquement dans la zone mémoire appelée *segment de pile*
  - les variables déclarées classiquement dans une fonction sont des variables temporaires
  - la durée de vie d'une variable temporaire est liée à celle du bloc dans lequel elle a été déclarée
    - ainsi une variable locale à une fonction, déclarée classiquement, n'existe plus dans la mémoire à la terminaison de la fonction

Cours/TD I - Exercice 3 : sous-programmes et tableaux ...

1. Écrire un programme `fact.c` contenant une fonction `fact` qui prend un entier n, et calcule sa factorielle de manière itérative. On souhaite ici récupérer le résultat du calcul dans un paramètre du sous-programme et non par retour de valeur de celui-ci. De plus la valeur de n devra être passée via la ligne de commande.
2. Écrire un programme `fact_r.c` contenant une fonction `fact_r` qui prend un entier n, et calcule sa factorielle de manière récursive. La fonction s'appelle elle-même, mais avec un paramètre de plus en plus petit. N'oubliez pas, 0 !=1. A l'aide du debugger, analyser l'état de la pile d'appels à chaque étape du programme.
3. Écrire un programme `pgcd.c` contenant une fonction `pgcd` qui calcule le pgcd de deux entiers de manière récursive sachant que si k divise a et b, il divise $|a - b|$. Ce moyen doit faire diminuer la différence entre a et b à chaque appel jusqu'à ...
4. Écrire un programme `inter.c` qui interclasse deux tableaux triés de taille m et n. Ce dernier permettra la transmission des valeurs par la ligne de commande. Par exemple :
   > `inter 3 5 10 15 4 1 2 10 16` représente deux tableaux (5,10,15) et (1,2,10,16)

↪ Structure d'un module

Un module en langage C se décompose en 2 fichiers :

- un fichier `MonModule.h`
  - contenant la spécification du module (comme le `.ads` Ada), c'est-à-dire les déclarations de types, déclarations de fonctions, ...
- un fichier `MonModule.c`
  - contenant, au minimum, l'implémentation des fonctions déclarées dans le fichier `.h` (comme le `.adb` en Ada).
  - d'autres fonctions peuvent être ajoutées, mais elles ne seront alors pas visibles depuis l'extérieur du module.
  - le fichier `MonModule.c` doit faire l'inclusion du fichier `MonModule.h` à l'aide de la directive préprocesseur `#include ....`

Mais attention au problème d'inclusion multiple ...

↪ Espace de nommage & inclusion

- définition : Un espace de nommage est un contexte dans lequel les entités (types, variables, fonctions, ...) sont définies :
- dans de nombreux langages, un espace de nommage est associé à un module/package, contrairement au langage C où cette notion n'existe pas.
- un *conflit de nom* peut se produire dans 2 cas :
  - si le code comporte des inclusions multiples (i.e. plusieurs `#include` du même fichier)
  - si 2 entités définies dans 2 modules différents sont identiques (identificateur pour les types et variables, signature pour les fonctions)
- 2 règles d'écriture doivent être appliquées pour palier ce problème :
  - utiliser les directives préprocesseurs pour éviter les inclusions multiples
  - préfixer les entités définies dans un module par le nom de module
  - exemple d'écriture du fichier `monmodule.h`

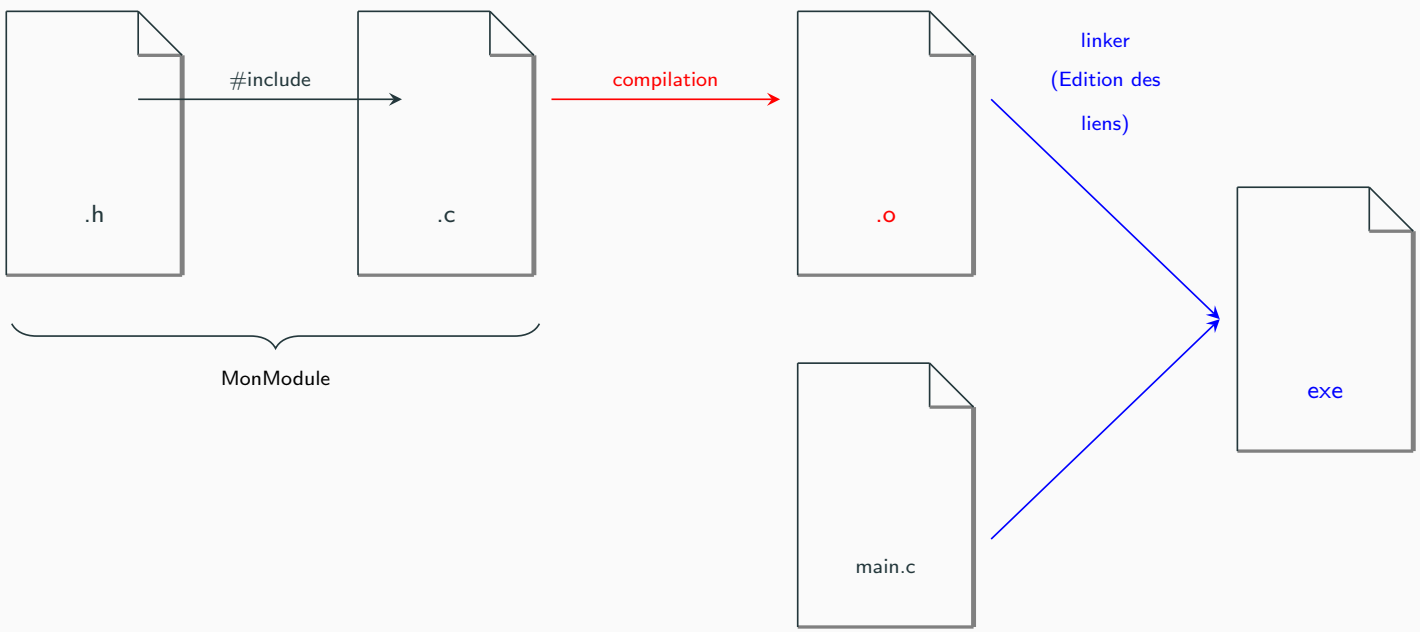
```
#ifndef MONMODULE_H_
#define MONMODULE_H_

int monmodule_lafonction();

#endif //MONMODULE_H
```

↪ Compilation & utilisation

- une fois créé le module (i.e. fichier .h et .c) est compilé pour produire un fichier .o
  - `gcc -c momodule.c` produit le fichier `momodule.o`
- le fichier `momodule.o` est utilisé lors de la construction du programme utilisant le module :
  - `gcc -o main.exe main.c monmodule.o`
- la figure ci-dessous décrit ce processus.



Cours/TD I - Exercice 4 : module nombre complexe ...

1. Réaliser un module `libcomplexe` permettant :
   - de définir un complexe sous sa forme cartésienne et polaire,
   - de réaliser les opérations d'addition, soustraction, multiplication et division entre complexes,
   - de réaliser les opérations d'addition, soustraction, multiplication et division entre complexe et réel,
   - de permettre un affichage console d'un nombre complexe.
2. Dans un fichier `test-complexe.c`, tester votre nouvelle API `libcomplexe`

IV - Flux d'entrées/sorties et gestion des erreurs

↪ Définition

Un programme C communique avec son environnement à l'aide de la notion de flux (déjà rencontrée lors de l'utilisation des fonctions `printf` et `scanf`).
Au lancement, un programme C dispose de trois flux déjà ouvert :

- le flux d'entrée standard : `stdin`
  - par défaut, le clavier
- le flux de sortie standard : `stdout`
  - par défaut, l'écran
- le flux d'erreur standard : `stderr`
  - par défaut, l'écran

Ces trois variables sont en fait de type "fichier", i.e. `FILE *` et s'utilise donc avec les fonctions `fscanf` et `fprintf`. Ainsi :

- `fscanf(stdin,...)` : lecture sur le flux d'entrée standard (comme `scanf(...)`)
- `fprintf(stdout,...)` : permet d'écrire sur le flux de sortie standard (comme `printf(...)`)
- `fprintf(stderr,...)` permet d'écrire sur le flux de d'erreur standard

↪ Exemple

Ci-dessous un exemple d'utilisation du flux d'erreur standard lors d'une erreur de lecture du flux d'entrée standard. On utilise la valeur de retour de la fonction scanf qui correspond au nombre de lectures effectuées pour détecter l'erreur.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int val;
  int ret;

  /* Comme un printf */
  fprintf(stdout, "Saisir un entier :");
  /* Comme un scanf */
  ret = fscanf(stdin, "%d", &val);
  if (ret == 0)
  {
    fprintf(stderr, "Erreur de saisie ...\n");
    return 1;
  }
  printf("valeur lue : %d\n", val);
  return 0;
}
```

↪ Définition

La méthode précédente, basée sur la valeur de retour d'une fonction, n'est pas applicable pour toutes les fonctions et, de plus, ne fournit pas d'information sur le type de l'erreur.

Le langage C fournit une API dédiée à la gestion des erreurs, `errno.h`, qui contient :

- des codes d'erreur prédéfinis, utilisés par les fonctions de la bibliothèque standard
- une variable globale `errno` utilisée pour la transmission et la détection de l'erreur
- des méthodes d'affichage du message d'erreur associée au numéro de l'erreur, avec en particulier :
  - `char * strerror(int code);` : permet d'obtenir le message associé au numéro d'erreur au format chaîne.
  - `void perror(const char *chaine);` : permet d'envoyer sur le flux `stderr` le message d'erreur. La chaine en paramètre, si différente de `NULL` sera affichée avant le message d'erreur.

↪ Utilisation pour les fonctions de la bibliothèque standard

Ainsi, pour détecter une erreur se produisant lors de l'appel d'une fonction de la bibliothèque standard, on procédera de la manière suivante :

- initialiser `errno` à 0
- Appel de la fonction
- Vérifier si elle a échoué
  - Si échec : la valeur de `errno` est disponible pour traiter l'erreur ;
  - Sinon : suite du programme.

Ci-conte, un exemple de détection d'erreurs avec la fonction `atoi`

- exécution avec la valeur douze :
  - provoque `Erreur detectée ... : Invalid argument`
- exécution avec la valeur 455555555555555555545112 :
  - provoque `Erreur detectée ... : Result too large`

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <errno.h>
4
5   int main(int argc, char *argv[])
6   {
7     char val[250];
8     int ret;
9     fprintf(stdout, "Saisir un entier :");
10    fscanf(stdin, "%s", &val[0]);
11    errno = 0;
12    ret = atoi(val);
13    if (errno)
14    {
15      perror("Erreur detectée ... ");
16      return errno;
17    }
18    printf("valeur lue : %d\n", ret);
19    return 0;
20  }
```

↪ Mise en œuvre dans nos propres fonctions

- Principe : s'appuyer sur `errno.h` pour mettre en place une gestion d'erreurs plus robuste
- une fonction pouvant modifier la valeur de la variable `errno`
  - trouver le code erreur prédéfini correspondant le mieux à l'erreur que l'on souhaite signaler
  - modifier la valeur de `errno`

- exemple : fonction dont le seul but est de signaler une erreur `EBUSY`
- À l'exécution :

```
Je genere une erreur EBUSY ...
Erreur detectée ... : Resource busy
```

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

void fonction_erreur(){
  printf("Je genere une erreur EBUSY ...\n");
  errno = EBUSY;
}

int main(int argc, char * argv[]){
  errno = 0;
  fonction_erreur();
  if (errno)
  {
    perror("Erreur detectée ... ");
    return errno;
  }
}
```

V - Pointeurs & Allocation mémoire

↪ Définition & utilisation

Les pointeurs sont fondamentaux en C :

- Un pointeur contient une adresse.
- Syntaxe de déclaration : préfixer le nom de la variable par *

```
int * pt_entier; char c,* pt_char;
```

- L'accès à l'adresse d'une variable pointeur se fait par l'opérateur &

```
int i=3;
pt_entier = &i;
```

  - remarque : `&i` est considérée une constante

- L'Accès à la valeur d'une variable pointeur se fait par l'opérateur *

```
int j= *pt_entier; // j vaut dorénavant 3
*pt_entier = 4; /* la valeur pointée par pt_entier vaut 4
```

- La maladie du programmeur C :
  - Utiliser les `void *` sans aucune notion du type de la valeur pointée ...

↪ Arithmétique des pointeurs

La valeur d'un pointeur est un entier. On peut donc lui appliquer certains opérateurs :

- addition d'un entier à un pointeur ; retourne un pointeur de même type
- soustraction d'un entier à un pointeur ; retourne un pointeur de même type
- différence de eux pointeurs ; retourne un entier

Mais attention, le résultat est dépendant de la taille du type de la variable pointée. Précisément :

- si `p` est un pointeur sur une varibale de type `Typ`, et `ent` un entier, l'expression `p+ent` correspond en réalité à :
  `p + ent * sizeof(Typ)`
  - il en va de même pour la soustraction et pour les opérateurs `++` et `--`

↪ Pointeurs & tableaux

- rappelez vous, un *tableau est un pointeur* . . . et ses éléments sont stockés de manière *contiguë* . . .
- le pointeur est donc un outils utile pour la manipulation des tableaux
- le code suivant permet le parcours d'un tableau en utilisant un index pointeur se déplaçant sur les éléments :

```
1   #define TAILLE 5
2   int tab[TAILLE] = {12, 2, 56, 44   , 7};
3
4   int main(int argc, char *argv[])
5   {
6     int *p;
7     for (p = tab; p <= &tab[TAILLE-1]; p++) {
8       printf(" %d \n",*p);
9     }
10    return 0;
11  }
```

↪ Pointeurs & structures/unions

- Il est bien évidement possible de déclarer un pointeur sur une structure ou sur une union
  - plusieurs écritures possibles
- l'accès aux champs de la structure depuis la variable pointeur se fait par l'une des deux syntaxes suivantes (totalement équivalentes) :
  - `(*p).champ`
  - `p->champ`

```
1   struct article {
2     int champ1;
3     float champ2;
4   };
5
6   typedef struct article *T_particle;
7   T_particle varptr;
8   ...
9   (*varptr).champ1 = 4;
10  varptr->champ2 = 2.5f;
```

↪ Pointeur de fonction

- permet le passage d'une fonction en paramètre d'un sous programme
- correspond à l'adresse du début du code de la fonction
- un pointeur sur une fonction f de signature :
  `type fonc(type_1, ... , type_n)`
  sera de type :
  `type (*) fonc(type_1, ... , type_n)`
- ainsi la signature de la fonction nécessitant un paramètre de type pointeur sur fonction s'écrira :
  `type foncgenerale(type (*f) (type_1, ... , type_n))`
- dans la cette fonction, l'utilisation de la fonction en paramètre se fera par :
  `(*f)(val_1,...,val_n)`
- l'appel de la fonction générale se fera par :
  `foncgenerale(fonc)`
  - toute fonction respectant la signature de `fonc` peut-être utilisée lors de l'appel

↪ Principe

Avant de pouvoir stocker une valeur dans une variable "pointeur", il faut *allouer de la mémoire*.
Cette opération consiste à réserver un espace mémoire dans le tas (ou heap) de taille suffisante pour stocker une valeur du type pointé (4 octet pour un `int` par exemple)

- dans le bloc mémoire occupé par le programme, le tas est un emplacement mémoire de taille variable, contrairement à la pile (ou stack), dédié aux allocations dynamiques.

↪ Les fonctions d'allocation et libération mémoire

L'allocation mémoire est réalisée en utilisant des fonctions définies dans la bibliothèque `stdlib.h`, et en particulier :

- la fonction `sizeof(type)`
  - permettant d'obtenir la taille du type de la valeur à stocker
- la fonction `void *malloc(size_t elsize)`
  - permettant d'allouer (i.e. de réserver) une zone mémoire de taille `elsize`
  - retourne l'adresse de départ du bloc alloué sous la forme d'un type `void *` devant être "casté" pour correspondre au type de la variable pointée
  - si l'allocation échoue, par manque d'espace par exemple, la valeur retournée est égale à `NULL` et `errno` vaut `ENOMEM`
- ou la fonction `void *calloc( size_t nbElem, size_t elsize ))`
  - utilisée pour les tableaux
  - même fonctionnement que la fonction `malloc`
  - assure également l'initialisation à 0 des valeurs pointées.
- la fonction `void free(vod * p)`
  - permettant de libérer la mémoire allouée pointée par le pointeur `p`
  - attention lors de la manipulation de cette fonction

↪ Exemple

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

int main(int argc, char *argv[])
{
  int *ptr = (int *)malloc(sizeof(int));

  if (ptr == NULL) {
    /* problème lors de l'allocation */
    perror("malloc failed");
  } else {
    /* la mémoire est allouée */
    ...
    /* libération de la mémoire une fois la varibale pointeur inutile */
    free(ptr);
  }


  return 0;
}
```

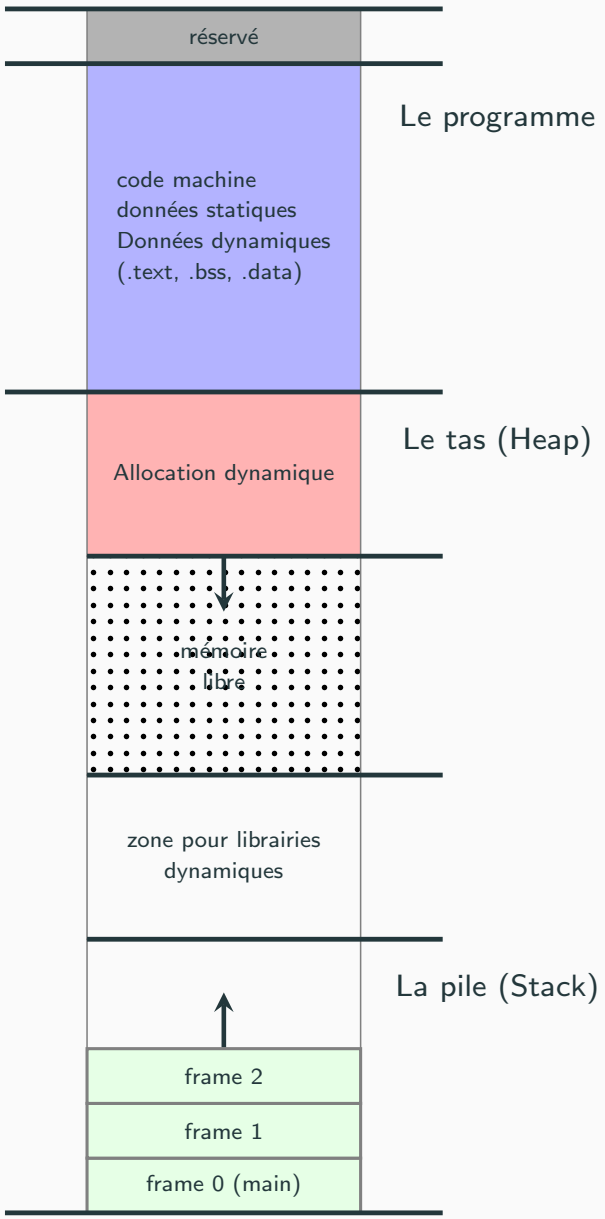
- notez la vérification du bon fonctionnement de l'allocation
  - opération indispensable à effectuer dès que l'on alloue de la mémoire
  - utiliser le mécanisme de gestion des erreurs

↪ Les fonctions de manipulation mémoire

- ré-allocation : il est possible de redimensionner la zone de mémoire allouée avec `realloc`
  `void * realloc ( void * base , size_t t )`
  - nécessite l'adresse de départ de la zone précédemment allouée et la nouvelle taille.
  - Le système de gestion d'erreur reste le même
  - retourne un pointeur vers l'adresse du nouveau bloc mémoire et le contenu de la zone de départ est conservée
- comparaison : comparer les valeurs de deux zones de même taille : `memcmp`
  `int memcmp(void *T,void *T2,size_t n)`
  - compare les `n` premier octet de `T` et `T2`
  - retourne une valeur négative si inférieur, positive si supérieur et nulle si identique
- copie : la copie d'une zone mémoire dans une autre se fait à l'aide de `memcpy`
  `void *memcpy(void *dest, const void * src, size_t n)`
  - copie `n` octets de `src` dans `dest` (qui doit être alloué ...)

↪ Voyage au centre de la mémoire . . .

- la zone programme : découpée en plusieurs segments
  - segment *.text* : le code machine
  - segment *.bss* : données non initialisées
  - segment *.data* : données initialisées
- le tas :
  - pour l'allocation dynamique (malloc, ...)
  - croissance vers les adresses hautes
  - consomme la mémoire libre
- la pile :
  - commence à l'adresse la plus haute et croissance vers les adresses basses
  - composée de *frames* correspondant à des sous-programmes
    - lorsqu'un sous programme est appelé une frame est créée sur la pile
    - à la fin de celui-ci la frame est supprimée



Cours/TD I - Exercice 5 : module calcul matriciel ...

1. Créer un module calcul matriciel [a]. On rappel ici qu'un module est composé de deux fichiers : un fichier `monModule.h` contenant les parties déclaratives et un fichier `monModule.c` contenant les réalisations des sous-programmes du module.
   - il devra contenir la définition d'un type Matrice contenant
     - un tableau à deux dimensions,
     - deux entiers donnant la dimension de la matrice
   - ce module fournira les opérations suivantes :
     - Création d'une matrice (allocation mémoire à partir des dimensions)
     - Destruction d'une matrice (désallocation)
     - Saisie d'une matrice (contenu)
     - Affichage d'une matrice
     - Addition de deux matrices
     - Multiplication de deux matrices
   - ce module devra être le plus robuste possible aux erreurs
   - le programme principal permettra de saisir deux matrices, de les additionner et d'afficher la matrice résultante

a. On nommera ce module `Matrice`