RAPPELS :

I. Les deux modes de passage de paramètres

En java, le seul mode de passage de paramètre est dit *"par valeur"* ou *"par recopie"*, car *on recopie la valeur* du **paramètre effectif** (de la fonction appelante) dans le **paramètre formel** (de la fonction appelée). Tous les paramètres sont donc des **paramètres d'entrée** car les valeurs entrent dans la fonction appelée (et n'en sortent pas !). Exemple :

```
void m1() { int vE = 12;  System.out.println( m2(vE) + " : " + vE ); }
int m2( int pE ) { pE = pE+1;  return pE*2; }
```

affichera `26 : 12` car le *paramètre effectif* `vE` (qui vaut 12) est d'abord recopié dans le *paramètre formel* `pE` (qui maintenant vaut aussi 12), puis `pE` est incrémenté (et vaut donc 13), puis `m2` retourne `pE*2` (donc 26), résultat qui est affiché par `m1` suivi d'un deux-points suivi de `vE` (qui vaut toujours 12).

En C, les paramètres que nous avons utilisé au TP C1 fonctionnent de la même manière, mais il en existe une deuxième sorte : les **paramètres de sortie** (ou d'entrée/sortie, car *Qui peut le plus peut le moins !*) qui utilisent un passage de paramètre dit *"par adresse"*.
Cela veut dire qu'on ne passe plus la valeur d'une variable, mais son adresse. Cela nous permettra donc non seulement d'accéder à la valeur, mais également de la modifier dans la fonction appelante ! Cette possibilité est très utilisée en C, notamment pour saisir des valeurs.

II. La saisie de valeurs

Pour saisir une valeur en C, il faut utiliser la fonction de bibliothèque `scanf` qui est déclarée dans `stdio.h` et qui s'utilise de la façon suivante : `scanf( format, lieu );`
où `format` est une chaîne de caractères spécifiant le type de donnée attendue (comme dans `printf`) et `lieu` est l'adresse de la variable dans laquelle on souhaite stocker la valeur saisie par l'utilisateur.
En C, pour calculer l'adresse d'une variable `var`, on écrit simplement **`&`** `var`.
Par exemple, `int vE; scanf( "%d", &vE );`
lit la valeur entière tapée au clavier et la stocke dans la variable `vE`.
Attention ! `scanf` n'affiche rien, il n'est donc pas possible de mettre un message dans le `format`. Comme `printf`, `scanf` accepte plusieurs `%` si chacun correspond à une variable qu'on lui fournit.

III. Mon premier pointeur

III.A On désire maintenant écrire une procédure saisie qui affiche un message avant la saisie. Cette procédure aura donc pour paramètre l'adresse de la variable dans laquelle on veut stocker la valeur. Mais comment indiquer que le paramètre est une adresse ?
- Lors de l'appel, nous avons vu qu'il suffit d'écrire `&var` pour passer l'adresse de la variable `var`
- Mais lors de la définition, il nous faut un moyen de noter le type "adresse d'entier" ; cela se note avec une `*` après le type : `void saisie( int * pVar )`

Une première façon de lire cette déclaration est de rapprocher l'`*` du `int` pour dire que le paramètre `pVar` est du type `int*`, c-à-d "adresse d'entier" appelé en C "**pointeur d'entier**".
Le corps de la procédure `saisie` va ressembler à :

```
{ printf( "Veuillez saisir un nombre entier : " );
  scanf( "%d", pVar );
} // saisie(.)
```

Remarquer qu'il n'y a pas `&` avant `pVar` dans le `scanf` car `pVar` est déjà une adresse d'entier !

III.B On aimerait pouvoir écrire, si la valeur était dans une variable entière `vX` :

```
if ( vX < 0 )  vX = -vX;
```

Mais dans notre cas, comment mettre la valeur saisie dans `vX` ? Écrire `int vX = pVar;` provoquerait une erreur de compilation, puisque `pVar` n'est pas un entier mais l'adresse d'un entier.
Si on regarde différemment la déclaration de la procédure saisie en rapprochant l'`*` de `pVar`, on obtient `int *pVar`, ce qui veut dire que `*pVar` est un entier ; nous tenons donc notre solution : `*pVar` se lit *"l'entier qui est à l'adresse `pVar`"*, d'où la possibilité de remplacer `vX` par `*pVar` partout dans l'instruction `if` ci-dessus.

IV. Les chaînes de caractères

IV.A Le type `String` n'existe pas en C : une chaîne de caractères est un simple tableau de caractères dont chaque case contient un caractère de la chaîne. On utilise alors le type `char` qui occupe un seul octet pour stocker le code ASCII du caractère.
Pour utiliser les chaînes plus agréablement, de nombreuses fonctions ont été définies dans `string.h`, comme par exemple `strcpy` (qui signifie **str**ing **copy**) qui recopie la chaîne passée en second paramètre dans la chaîne passée en premier paramètre :

```
char vChaine[10];  strcpy( vChaine, "Bonjour" );
```

où le tableau `vChaine` est déclaré comme un tableau de 10 `char` dont on ne connaît a priori pas les valeurs (qui peuvent être vues comme des caractères aléatoires).
Après le `strcpy`, `vChaine[0]` vaut `'B'` jusqu'à `vChaine[6]` qui vaut `'r'`. Les *simple quotes* (ou apostrophes) sont utilisées pour désigner un seul caractère (c'est-à-dire une constante du type `char`) à la différence des *double quotes* (ou guillemets) qui désigneraient une chaîne de caractères (c'est-à-dire un tableau de `char`).

IV.B On peut donc écrire : `void saisie( char pMessage[], double * pVar )`
mais comme le nom d'un tableau en C (par exemple `vTab`) représente l'adresse de son premier élément (donc `vTab == &vTab[0]`), on peut aussi écrire :
`void saisie( char * pMessage, double * pVar )`
C'est ce qu'on appelle **l'équivalence tableau <--> pointeur**. Il faut donc comprendre ici que passer un tableau en paramètre, c'est passer son adresse (ou plus exactement l'adresse de son premier élément), et donc c'est permettre la modification de ses éléments.

Attention ! Cette équivalence tableau <--> pointeur ne doit pas masquer une différence essentielle lors de la déclaration :
`char vChaine[10];` déclare un tableau de 10 octets, alors que `char * vPtr;` déclare seulement un pointeur de caract. (de 4 octets sur une machine 32 bits).
Ensuite, si on stocke dans `vPtr` l'adresse du premier élément de `vChaine` par l'instruction `vPtr = &vChaine[0];` ou plus simplement `vPtr = vChaine;` l'équivalence tableau <--> pointeur jouera à plein et on pourra indifféremment utiliser `vPtr` ou `vChaine` pour accéder au tableau en écrivant `vPtr[6]` (qui vaudra `'r'` dans l'exemple du III.A) ou `*vChaine` (qui vaudra `'B'`).

Comme nous voulons afficher le message passé en paramètre, il nous faut connaître le format qui permet à `printf` d'afficher une chaîne de caractères : il s'agit de **`"%s"`**.
Mais au fait, si on ne passe à `printf` que l'adresse du premier élément du tableau de caractères, comment fait-elle pour savoir quand s'arrêter d'afficher des caractères du tableau ? Le tableau a été déclaré avec 10 cases, mais s'il ne contient que le mot `"Bonjour"`, il ne faut afficher que 7 caractères !
En fait, toutes les fonctions de la bibliothèque standard du C comme `printf` ou celles de `string.h` utilisent la même convention : elles s'arrêtent dès qu'elles trouvent le **caractère nul**, c'est-à-dire le caractère de code ASCII 0. Même le compilateur ajoute automatiquement ce caractère à la fin des chaînes de caractères littérales comme `"Bonjour"` : il faut donc un tableau d'au moins 8 caractères pour contenir une chaîne d'apparemment 7 caractères !
C'est une des premières causes de plantage des programmes C (lorsqu'on recopie une chaîne de N caractères dans un tableau de N `char` ...)
Le caractère nul se note **`'\0'`** qui veut dire *caractère de code ASCII 0*.
A noter également que le contenu des cases 8 et 9 du tableau est aléatoire (comme toute variable non initialisée).

V. Attention ! `char[]` est un pointeur constant alors que `char*` est un pointeur variable.

VI. Équivalence fonction / procédure

En C, une fonction à N paramètres peut s'écrire sous forme d'une procédure à N+1 paramètres, le dernier étant un paramètre de sortie fournissant le résultat habituellement retourné par la fonction.
Reste une différence fondamentale entre fonction et procédure : l'appel d'une fonction est une expression, alors que l'appel d'une procédure est une instruction.