TP de programmation en langage C Cédric Bentz/Daniel Porumbel

Séance de TP no 1 du 12/09/2024

Exercice 1 : liste de pays (alias *"le test de positionnement"*)

Dans cet exercice, on vous demande d'écrire un programme (fonction `main` dans un unique fichier C, que nous appellerons ici `liste_pays.c`) qui récupère une liste de pays saisie par l'utilisateur au moment de l'appel du programme (liste des noms des pays qu'il souhaiterait visiter ou revisiter, par exemple, et qui doivent donc être séparés par des espaces), et qui affiche ensuite le nombre de ces pays, puis les liste un par un (un sur chaque ligne). Ainsi, après compilation du programme par le biais de la commande `gcc liste_pays.c -o liste_pays`, l'appel au programme

```
> liste_pays Canada Japon Maroc
```

entraînera l'affichage suivant :

```
Vous avez liste 3 pays, que voici :
Canada
Japon
Maroc
```

Indications :

1. On vous rappelle que les paramètres utilisés au moment de l'appel d'un programme C sont séparés par des espace et sont stockés, sous forme de valeurs de type `char *` (chaîne de caractères), dans un tableau passé en paramètre du `main` (et qu'on appellera `tab_args`). En d'autres termes, l'en-tête d'une fonction `main` en C s'écrit ainsi : `int main(int nb_args, char *tab_args[])`, où `nb_args` est le nombre de cases (c'est-à-dire d'éléments) du tableau `tab_args`.
2. À noter que `tab_args[0]`, l'élément en première case du tableau (celle à l'indice 0), est en fait le nom du programme lui-même, c'est-à-dire `liste_pays`. Il y aura donc en réalité `nb_args` -1 noms de pays.
3. L'affichage des noms de pays à l'écran nécessite l'utilisation de la procédure `printf`, qui se trouve dans le fichier d'en-tête `stdio.h` (qu'il faut donc inclure dans `liste_pays.c`, à l'aide de la directive `#include`). Les valeurs à afficher (noms de pays) étant de type `char *`, il faut utiliser l'indicateur de format `%s` lors de l'appel à `printf`.

Exercice 2 : le problème du boulanger

Dans cet exercice, on se propose d'implémenter en C un algorithme glouton qui résout le "problème du boulanger", c'est-à-dire le problème consistant à rendre une somme d'argent donnée en euros (et en centimes d'euros) en utilisant le moins de billets/pièces possible.

Informellement, cet algorithme glouton, qui considère l'ensemble des billets et pièces dans l'ordre décroissant de leur valeur, est très simple :

- *pour chaque valeur de billet/pièce, dans l'ordre précisé ci-dessus* :
  - *utiliser le nombre maximum de billets/pièces de la valeur courante, sans dépasser le montant restant à rendre, puis mettre à jour le montant restant à rendre.*

Le programme prendra 2 paramètres au moment de l'appel : la partie en euros du montant à rendre, et sa partie en cents. Après les avoir convertis en 2 entiers, il affichera ensuite le nombre total de billets et de pièces à utiliser, et le nombre de billets/pièces de chaque valeur nécessaires pour atteindre cette somme totale (en euros et cents) en en rendant le moins possible.

Pour rappel, on considèrera les 7 valeurs de billet suivantes (en euros) : 500, 200, 100, 50, 20, 10, et 5. De même, on considèrera les 8 valeurs de pièce suivantes (toutes inférieures à 5 euros, la plus petite valeur de billet) : 2 euros, 1 euro, 50 cents, 20 cents, 10 cents, 5 cents, 2 cents, et 1 cent.

Lors de l'affichage, on prendra soin de n'afficher que les billets et pièces effectivement utilisés. Par exemple, si le montant considéré est 43 euros, alors l'affichage correspondant sera le suivant :

```
> boulanger 43 0

Rendre 43.0 euros necessite d'utiliser 2 billets et 2 pieces :
2 billets de 20 euros
1 piece de 2 euros
1 piece de 1 euro
```

Indications :

1. On définira un type énuméré, dont chaque variable pourra prendre 3 valeurs : `BE` (pour "billet en euros"), `PE` (pour "pièce en euros") et `PC` (pour "pièce en cents"). On rappelle la syntaxe de la déclaration d'un tel type (ici nommé `type_enumere`) : `enum type_enumere {VALEUR1, VALEUR2, VALEUR3}`. Le mieux est ensuite de procéder à un renommage de type (pour ne pas avoir à écrire `enum type_enumere` en permanence), dont la syntaxe est la suivante : `typedef type1 alias_type1`. Ainsi, le nom `alias_type1` peut ensuite être utilisé partout dans le programme à la place de `type1`.

2. On définira 3 constantes entières : une pour le nombre de valeurs différentes de billets en euros (7), une pour le nombre de valeurs différentes de pièces en euros (2), et une pour le nombre de valeurs différentes de pièces en cents (6). On rappelle que, en C, les constantes se déclarent à l'aide du mot clé `const`, en général de façon globale (c'est-à-dire en début de fichier, avant les fonctions/procédures), et la syntaxe associée est : `const type Nom_cte = val_cte;` (noter la majuscule).

3. On définira également 3 tableaux constants : le premier est le tableau des valeurs des billets en euros (soit 500, 200, 100, 50, 20, 10, et 5), le second est celui des valeurs des pièces en euros (soit 2 et 1), et le dernier est celui des valeurs des pièces en cents (soit 50, 20, 10, 5, 2, et 1). Si on passe un de ces tableaux en paramètres d'une fonction/procédure, ou bien si on définit une variable locale qui sera destinée à prendre une de ces valeurs, il faut alors préciser que c'est un(e) paramètre/variable constant(e), toujours à l'aide du mot-clé `const`.

4. On écrira une première fonction qui prendra en paramètres le montant à considérer (soit en euros, soit en cents), une variable de type énuméré (valant soit `BE`, soit `PE`, soit `PC`, selon qu'on considère les valeurs des billets en euros, des pièces en euros, ou des pièces en cents, respectivement), et un tableau dont la taille est le nombre de valeurs considérées (donc 7, 2 ou 6, selon que la variable précédente vaut `BE`, `PE`, ou `PC`, respectivement), et qui sera destiné à contenir le nombre de billets/pièces de chaque valeur nécessaires pour atteindre le montant considéré, selon l'algorithme glouton précédemment décrit.

   Cette fonction renverra une valeur entière (donc de type `int`), qui représente le nombre **total** de billets/pièces nécessaires pour atteindre le montant considéré (c'est-à-dire, tout simplement, la somme des éléments du deuxième tableau). On utilisera notamment l'instruction `switch` pour traiter les 3 valeurs possibles du paramètre qui est de type énuméré, en n'oubliant pas d'ajouter une instruction `break` après chaque instruction `case` pour sortir du `switch` après exécution du `case` (faute de quoi plusieurs `case` pourraient être exécutés à la suite).

5. On écrira également une ou plusieurs autres procédures qui seront chargées de l'affichage correct du résultat à l'écran (et qui devront donc gérer l'affichage des mots de liaison, des "s" quand cela est nécessaire, etc.). On pourra, là aussi, utiliser l'instruction `switch`.

6. Enfin, on écrira le `main`, qui sera en charge de faire les appels aux fonctions/procédures précédentes, et de convertir en entiers les 2 paramètres passés au moment de l'appel. Pour cela, on utilisera la procédure `atoi`, qui renvoie la valeur entière correspondant à la valeur de type `char *` passée en paramètre, et qui se trouve dans le fichier d'en-tête `stdlib.h`.

Exercice 3 : Puissance 4

Dans cet exercice, on se propose d'implémenter une version C du célèbre jeu de Puissance 4, à l'aide d'un affichage en mode texte et d'une interface simple au clavier. On rappelle que ce jeu se joue à 2 joueurs, sur une grille rectangulaire constituée de 6 lignes et 7 colonnes.

À tour de rôle, chacun des 2 joueurs va choisir une des 7 colonnes pour y jouer un de ses jetons. À chaque fois qu'un jeton est joué dans une colonne, il "descend" jusqu'à l'emplacement libre situé le plus bas sur cette colonne (pour simuler un jeton qui tomberait sous l'effet de la pesanteur), et cet emplacement devient alors occupé (par ce jeton). Une colonne est remplie lorsque plus aucun emplacement n'y est disponible, ce qui se produit lorsque 6 jetons ont été joués dans cette colonne. Lorsqu'une colonne est remplie, plus aucun joueur ne peut jouer de jeton dedans.

Dès qu'un des 2 joueurs arrive à aligner 4 de ses jetons (horizontalement, verticalement, ou en diagonale), il gagne aussitôt la partie. Si toutes les colonnes sont remplies mais qu'aucun des 2 joueurs n'a réussi à aligner 4 jetons, alors la partie se conclut sur un match nul.

Pour que le programme puisse écrire l'identité du vainqueur dans le cas où la partie ne se solderait pas par un match nul, on fournira en paramètres, *au moment de l'appel du programme*, les prénoms des 2 joueurs.

On se propose de modéliser la grille de jeu à l'aide d'une variable de type matrice (tableau à 2 dimensions) : ainsi, chaque emplacement de la grille correspondra à une case de cette matice. Une case pourra donc soit être libre, soit être occupée par un jeton du joueur 1, soit être occupée par un jeton du joueur 2. Voici des indications générales sur la marche à suivre :

1. On écrira, puis on appellera au début de chaque tour de jeu, une procédure permettant d'afficher la grille : on affichera, en haut, les numéros des 7 colonnes, puis les éléments de chaque colonne les uns en-dessous des autres. Un jeton du joueur 1 sera représenté par un caractère donné (par exemple, 'X'), un jeton du joueur 2 sera représenté par un autre caractère (par exemple, 'O'), et un emplacement libre sera représenté par un troisième caractère (par exemple, '_'). Ces symboles seront également rappelés au début de chaque tour de jeu, entre parenthèses, à côté du prénom du joueur courant.

2. On écrira une procédure pour tester si un des joueurs (le joueur courant) a réussi à aligner 4 jetons sur la grille (horizontalement, verticalement ou en diagonale). Il y a pour cela plusieurs façons de faire. L'une d'entre elles consiste à tester, pour chaque case de la matrice contenant un jeton du joueur, la présence consécutive de 4 jetons lui appartenant (dans les trois directions possibles) à partir de cette case.

3. On aura également besoin d'écrire une procédure gérant un tour de jeu, c'est-à-dire la saisie du numéro de colonne choisie pour jouer le jeton, puis l'ajout d'un jeton sur la case libre la plus basse de cette colonne (après avoir vérifié que le numéro de colonne est valide et que la colonne n'est pas pleine, sinon le joueur doit en choisir une autre !).

4. Enfin, on aura besoin de gérer l'enchaînement des tours de jeu (c'est-à-dire l'alternance entre les deux joueurs), ainsi que la fin de partie : après avoir joué son jeton, le joueur courant a-t-il remporté la partie, ou provoqué un match nul ? Lorsque la partie s'achève, on affichera le résultat : match nul, victoire du joueur 1 (en rappelant son prénom et le symbole représentant ses jetons), ou victoire du joueur 2 (en rappelant là aussi son prénom et le symbole représentant ses jetons).

Voici à présent des indications plus techniques sur le code à produire :

1. On définira le nombre de lignes et de colonnes de la grille de jeu à l'aide de 2 constantes entières (à nouveau à l'aide du mot-clé `const`). De la même façon, on définira à l'aide de 3 `char` constants les caractères représentant respectivement un jeton du joueur 1 (disons 'X'), un jeton du joueur 2 (disons 'O') et une case inoccupée (disons '_').

2. On définira 3 types énumérés (cf exercice 2) : le premier pour représenter l'état du jeu (avec 4 valeurs, pour signifier respectivement que la partie n'est pas encore finie, ou bien qu'elle s'est achevée sur un match nul, une victoire du joueur 1, ou une victoire du joueur 2), le second pour garder trace du joueur courant (joueur 1 ou joueur 2), et le troisième pour symboliser le contenu d'une case (libre ou occupée par un jeton du joueur 1 ou du joueur 2). La grille de jeu sera alors représentée par une matrice d'éléments de ce dernier type.

3. On définira une *variable globale* (c'est-à-dire déclarée en-dehors des fonctions et procédures, en général tout en haut du fichier) dont la valeur (entière) représentera le nombre total de jetons joués depuis le début de la partie (en considérant les jetons joués par les 2 joueurs).

4. À chaque tour de jeu, on enregistrera le numéro de la colonne dans laquelle le joueur courant souhaite jouer son jeton à l'aide de la procédure `scanf` (avec l'indicateur de format `%d`, puisqu'on veut faire saisir un entier), qui se trouve dans le fichier d'en-tête `stdio.h`. Après avoir vérifié que ce numéro est valide (sinon, le joueur doit resaisir un coup à jouer), on cherchera la case libre la plus "basse" sur cette colonne, pour y placer le jeton. Si une telle case n'existe pas, c'est que la colonne est pleine, et il faut donc resaisir un autre numéro de colonne. Sinon, on place le jeton, et on vérifie alors si la partie s'achève ou non.

Voici un exemple de déroulement d'une partie (on ne montre que les premiers tours de jeu) :

```
> puissance4 tata toto

*** Bienvenue pour une nouvelle partie de PUISSANCE 4 : tata vs toto ! ***


   1234567

   -------
   -------
   -------
   -------
   -------
   -------


tata (X) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ? 0

   1234567

   -------
   -------
   -------
   -------
   -------
   -------


tata (X) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ? 1

   1234567

   -------
   -------
   -------
   -------
   -------
   X------
```

```
toto (O) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ? 2

   1234567

   -------
   -------
   -------
   -------
   -------
   XO_____


tata (X) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ? 5

   1234567

   -------
   -------
   -------
   -------
   -------
   XO__X__


toto (O) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ? 2

   1234567

   -------
   -------
   -------
   -------
   _O_____
   XO__X__


tata (X) : quel est le numero de la colonne
dans laquelle vous souhaitez jouer un jeton ?
```

Exercice 4 : sous-ensembles d'un ensemble de n entiers

Dans cet exercice, on vous demande d'écrire en C une procédure récursive (c'est-à-dire qu'à l'intérieur de la procédure on trouve un ou plusieurs appels à la procédure elle-même, mais avec des valeurs de paramètres différentes) d'énumération, qui sera appelée à partir du `main` (appel initial) et qui, étant donné un entier $n \geq 1$, énumère et affiche les $2^n - 1$ sous-ensembles non vides de l'ensemble $\{1, \ldots, n\}$. Pour cela, on utilisera un tableau d'entiers `t` de taille n, dont chaque élément sera destiné à prendre la valeur 0 ou 1.

Ainsi, il existe une bijection entre les valeurs possibles de l'ensemble des éléments de ce tableau `t` (sauf celle où tous les éléments valent 0) et les sous-ensembles non vides de l'ensemble $\{1, \ldots, n\}$.

La procédure récursive à écrire prendra en paramètre le tableau `t`, sa taille (nombre d'éléments), et l'indice courant de l'élément considéré dans le tableau (appelons-le `i`). Informellement, sauf dans le cas *terminal* (c'est-à-dire sans appel récursif, puisque dans ce cas il faudra simplement procéder à l'affichage du sous-ensemble obtenu) où `i` est égal à la taille de `t`, cette procédure fixera la valeur du `i`ème élément de `t` à 0 ou 1 (les 2 seules possibilités), et pour chaque valeur fera un appel récursif dans lequel la valeur de `i` sera incrémentée de 1 (pour considérer ensuite l'élément suivant).

L'appel initial à cette procédure récursive se fera à l'intérieur du `main`, après avoir demandé à l'utilisateur de saisir la valeur de n (toujours à l'aide de la procédure `scanf` qui se trouve dans le fichier d'en-tête `stdio.h`). On vous demande de gérer "à la main" les saisies de valeurs erronées pour n : ainsi, si l'entier saisi n'est pas strictement positif, on demandera à l'utilisateur de resaisir un entier, et ce jusqu'à ce que la saisie soit correcte. Pour cela, on utilisera obligatoirement une boucle `do {...} while(...)`. Une fois la valeur de n saisie, on affichera également explicitement l'ensemble $\{1, \ldots, n\}$.

Enfin, toujours dans le `main`, et avant d'appeler la procédure récursive, on allouera l'espace mémoire nécessaire au tableau `t`. Cette allocation sera une allocation dynamique de mémoire, puisque la taille de `t` n'est pas connue au moment de la compilation du programme (elle ne le sera qu'une fois la saisie de l'utilisateur effectuée). Cela nécessite donc l'utilisation d'une fonction prédéfinie, appelée `malloc`, qui se trouve dans le fichier d'en-tête `stdlib.h` (qu'il faut donc inclure, en plus de `stdio.h`).

Cette fonction prend une seule valeur en paramètre : le nombre d'octets occupés par le tableau à allouer. Pour connaître cette valeur, on utilise en général l'opérateur `sizeof(un_type)`, qui permet de connaître (en nombre d'octets) l'espace mémoire occupé par *une* valeur de type `un_type`. Ainsi, un tableau d'entiers de taille 100 occupera une place mémoire égale à $100 \times$ `sizeof(int)` octets. L'autre difficulté liée à l'utilisation de la fonction `malloc` est que sa valeur de retour est de type `void *`, c'est-à-dire *pointeur sur un élément dont le type n'est pas défini.*

On rappelle qu'un *pointeur* est une variable dont la valeur est l'adresse d'une zone mémoire où se trouve la valeur de la variable pointée (on dit aussi que ce pointeur *référence* la valeur pointée). Ainsi, la déclaration `int *x` signifie que la valeur de la variable `x` est l'adresse d'une zone mémoire où est stockée la valeur entière (donc de type `int`) d'une variable qu'on écrit assez logiquement `*x` en C. On rappelle également que *les valeurs passées en paramètres au moment de l'appel d'une procédure/fonction en C (et appelées paramètres effectifs) sont,* ***et ce quel que soit leur type****, systématiquement recopiées (localement) dans les paramètres (dits paramètres formels) déclarés au moment de la déclaration/définition de la procédure/fonction.* Il n'y a donc aucun lien entre les paramètres formels et effectifs, si ce n'est que les *valeurs* des premiers, au moment où débute l'appel de la procédure/fonction considérée, sont initialisées à l'aide des seconds. Cela a donné lieu à ce qu'on appelle le *passage par valeur* et le *passage par référence* d'une variable `x`. Cela signifie en fait simplement que, dans le premier cas, on passe en paramètre *la valeur de* `x` (qui sera donc recopiée localement dans un paramètre formel), et dans le second on passe en paramètre *l'adresse de l'emplacement mémoire où se trouve la valeur de* `x` (cette adresse est notée `&x` en C, et sera dans ce cas-là elle aussi recopiée localement dans un paramètre formel). On peut remarquer à ce propos que c'est exactement ce qui se passe lors de l'appel à la procédure `scanf`, puisque si on souhaite stocker dans une variable `x` un entier saisi par l'utilisateur, on passera en paramètre de `scanf` la valeur `&x`.

La manipulation de tableaux en C est en fait basée implicitement sur la notion de pointeur : plus précisément, si on déclare un tableau `tab` d'entiers à l'aide de l'instruction `int tab[]`, alors la *valeur de la variable* `tab` *est en fait l'adresse du premier élément de ce tableau d'entiers*, c'est-à-dire que c'est une valeur de type `int *`. En d'autres termes, si on cherche à allouer de l'espace pour un tableau d'entiers `t`, alors le type de `t` devra être `int *`. Comme la fonction `malloc` renvoie une valeur de type `void *` (et non `int *`), il faudra procéder à une conversion de type explicite (ou *transtypage*). Pour rappel, si on souhaite explicitement convertir une valeur `val` pour que son type devienne `type` (et à supposer que cette valeur puisse effectivement être convertie en une valeur de type `type`), il suffit d'écrire `(type) val`.

Enfin, *à chaque fois* que la fonction `malloc` est appelée pour obtenir l'allocation d'une zone mémoire de taille donnée, il faut ensuite (c'est-à-dire à partir du moment où le programme n'a plus besoin d'utiliser la zone allouée) appeler la fonction `free` (qui se trouve dans `stdlib.h`, et prend en paramètre l'adresse de la zone mémoire allouée) pour "libérer" cette zone (c'est-à-dire la désallouer), faute de quoi le programme risque de générer des *fuites de mémoire* (espaces mémoires alloués mais jamais libérés, et donc non réutilisables tant que le programme n'a pas fini de s'exécuter), et d'occuper inutilement trop de place dans la RAM, au détriment d'autres programmes.

Voici un exemple d'exécution du programme attendu :

```
Ce programme va afficher tous les sous-ensembles non vides de
l'ensemble {1, ..., n} pour un certain entier n > 0.

Merci de saisir la valeur de n : -2
Vous devez saisir un entier strictement positif !

Merci de saisir la valeur de n : 3

Voici tous les sous-ensembles non vides de l'ensemble {1, 2, 3} :

{3}
{2}
{2,3}
{1}
{1,3}
{1,2}
{1,2,3}
```

Exercice 5 : recherche dichotomique

Dans cet exercice, on vous demande d'implémenter la recherche dichotomique (en version récursive) d'un élément (entier) dans un tableau (d'entiers) trié par ordre croissant. Le programme demandera d'abord à l'utilisateur de saisir le nombre d'éléments que contiendra le tableau trié, en lui demandant de resaisir ce nombre si l'utilisateur saisit un entier qui n'est pas strictement positif. Cela se fera à l'aide d'une boucle `do {...} while(...)` similaire à celle de l'exercice 4. Ensuite, le programme demandera à l'utilisateur de saisir, un par un et dans l'ordre, les éléments de ce tableau d'entiers trié. Il vérifiera alors que ces entiers sont bien triés par ordre croissant, le confirmera à l'utilisateur le cas échéant, et s'arrêtera si ce n'est pas le cas. Enfin, dans le cas où les entiers saisis sont bien triés par ordre croissant, il demandera à l'utilisateur de saisir l'élément (entier) recherché dans ce tableau trié. S'il ne trouve pas cet élément, il en informera l'utilisateur. Dans le cas contraire, il affichera l'indice d'une case du tableau où est stocké cet élément.

Voici un premier exemple de déroulement du programme :

```
> dichotomie

Ce programme va verifier si, dans un tableau d'entiers tries par ordre croissant,
que vous allez saisir, se trouve un element (entier) donne,
que vous allez egalement saisir.
```

```
Saisir d'abord le nombre d'elements du tableau trie : 0
Vous devez saisir un entier strictement positif !

Saisir d'abord le nombre d'elements du tableau trie : 2
Saisir l'element no 1 (sur 2) de ce tableau trie d'entiers : 3
Saisir l'element no 2 (sur 2) de ce tableau trie d'entiers : 1

Le tableau de 2 entiers saisi est le suivant :
[3 1]
Les elements de ce tableau ne sont pas tries par ordre croissant !
```

Voici un deuxième exemple de déroulement du programme :

```
> dichotomie

Ce programme va verifier si, dans un tableau d'entiers tries par ordre croissant,
que vous allez saisir, se trouve un element (entier) donne,
que vous allez egalement saisir.

Saisir d'abord le nombre d'elements du tableau trie : 4
Saisir l'element no 1 (sur 4) de ce tableau trie d'entiers : 1
Saisir l'element no 2 (sur 4) de ce tableau trie d'entiers : 10
Saisir l'element no 3 (sur 4) de ce tableau trie d'entiers : 15
Saisir l'element no 4 (sur 4) de ce tableau trie d'entiers : 100

Le tableau de 4 entiers saisi est le suivant :
[1 10 15 100]
Les elements de ce tableau sont bien tries par ordre croissant !

Saisir maintenant l'element recherche dans ce tableau : 15

L'entier 15 se trouve en case 2 du tableau !
```

Enfin, voici un troisième exemple de déroulement du programme :

```
> dichotomie

Ce programme va verifier si, dans un tableau d'entiers tries par ordre croissant,
que vous allez saisir, se trouve un element (entier) donne,
que vous allez egalement saisir.

Saisir d'abord le nombre d'elements du tableau trie : 0
Vous devez saisir un entier strictement positif !
```

```
Saisir d'abord le nombre d'elements du tableau trie : 4
Saisir l'element no 1 (sur 4) de ce tableau trie d'entiers : 1
Saisir l'element no 2 (sur 4) de ce tableau trie d'entiers : 10
Saisir l'element no 3 (sur 4) de ce tableau trie d'entiers : 15
Saisir l'element no 4 (sur 4) de ce tableau trie d'entiers : 100

Le tableau de 4 entiers saisi est le suivant :
[1 10 15 100]
Les elements de ce tableau sont bien tries par ordre croissant !

Saisir maintenant l'element recherche dans ce tableau : 90

L'entier 90 n'a pas ete trouve !
```

Indications :

1. Après saisie de la taille du tableau d'entiers trié à l'aide de la procédure `scanf` utilisée au sein d'une boucle `do {...} while(...)`, on allouera l'espace mémoire nécessaire à ce tableau à l'aide de la fonction `malloc` (cf exercice 4 ; ne pas oublier non plus d'utiliser la fonction `free`).

2. On écrira une procédure à part (donc à appeler dans le `main`) pour tester si les entiers saisis sont bien triés par ordre croissant.

3. L'appel initial à la fonction récursive à écrire sera bien entendu à son tour effectué dans le `main`. Dans cette fonction récursive, on définira un espace de recherche lié au tableau trié à l'aide de 2 indices (disons `gauche` et `droite`) ; toutes les cases dont les indices sont situés entre `gauche` et `droite` feront par définition partie de l'espace de recherche. Initialement, cet espace de recherche contiendra toutes les cases du tableau. À chaque appel, on différenciera le cas terminal où l'espace de recherche est de taille 1 (c'est-à-dire ne contient qu'une seule case) du cas avec récursion, dans lequel on testera si l'élément recherché est plus petit ou plus grand que l'élément qui se trouve au milieu de l'espace de recherche (c'est-à-dire à l'indice $\left\lfloor \frac{gauche+droite}{2} \right\rfloor$). Selon le résultat de ce test, on réduira la taille de l'espace de recherche en se concentrant sur sa moitié située à gauche du milieu ou sur sa moitié située à droite du milieu, via un appel récursif adéquat.

4. On peut montrer, à l'aide de l'équivalent de 2 invariants de boucle (qui seraient ici davantage des invariants liés à un nouvel appel récursif, plutôt qu'à un nouveau passage dans une boucle), que cette procédure s'exécute en temps $O(\log n)$ sur un tableau de taille n. On écrira dans les commentaires du code C ces 2 "invariants de boucle", en expliquant pour quelle(s) raison(s) ils garantissent une telle complexité.

Petits rappels de C en vrac (instructions et syntaxe)

- En C, toute variable utilisée doit au préalable être déclarée, via la syntaxe : `type_variable nom_variable`, où `type_variable` peut être n'importe quel type valide en C, et où `nom_variable` ne doit pas commencer par un chiffre. Par exemple, `int x` permet de déclarer une variable qui s'appelle `x` et qui est de type `int` (entier).
- Une variable ainsi déclarée est visible (i.e., utilisable) au sein du *bloc* (accolades ouvrante et fermante qui se correspondent) où elle a été déclarée. Une variable déclarée en-dehors de tout bloc (*variable globale*) est visible partout dans le fichier.
- Les 4 types de base en C sont : `int` (pour représenter des entiers), `char` (pour représenter des caractères, chacun étant codé sur 1 octet), `float` et `double` (les 2 derniers pour représenter des nombres à virgule flottante, une variable de type `double` ayant une précision double par rapport à une de type `float`, donc étant codée sur 8 octets).
- Le type `int` peut être `unsigned` (c'est-à-dire représenter uniquement des valeurs entières positives, ce qui double la valeur du plus grand entier codé) ou non (par défaut). En réalité, on peut, si besoin, utiliser des types entiers (`unsigned` ou non) spécifiant le nombre exact de bits sur lequel ils sont codés (sinon, ce nombre de bits peut varier en fonction de la machine où le programme est exécuté) : ces types sont `int8_t` (sur 8 bits), `int32_t` (sur 32 bits), et `int64_t` (sur 64 bits), et leurs équivalents `unsigned` sont `uint8_t` (sur 8 bits), `uint32_t` (sur 32 bits), et `uint64_t` (sur 64 bits). Tous ces types sont utilisables en incluant le fichier d'en-tête `stdint.h`.
- On peut également, à l'aide de ces 4 types de base, définir des types composés, des pointeurs, ou des tableaux d'éléments (qui regroupent un nombre donné d'éléments de même type). La déclaration d'un tableau qui s'appelle `tab` et dont tous les éléments sont de type `double` s'écrira `double tab[]`. La déclaration d'un pointeur qui s'appelle `p` et qui référence une valeur entière s'écrira `int *p`.
- Il n'y a pas de type *booléen* natif en C, mais un type `bool` est défini dans le fichier d'en-tête `stdbool.h`, tout comme les valeurs `true` et `false` (qui sont en fait les valeurs entières 1 et 0 renommées, respectivement).
- Une affectation (d'une valeur à une variable) en C s'écrira simplement `nom_variable = valeur`. À noter que les opérandes gauche et droite ne jouent pas du tout le même rôle ici : le membre de gauche correspond à la variable dont la valeur sera modifiée, alors que, même si le membre de droite est passé sous la forme d'une variable, seule la *valeur* de cette variable sera utilisée (pour être recopiée dans le membre de gauche).

- Le type `size_t` (celui de la valeur renvoyée par l'opérateur `sizeof`) et la valeur `NULL` (celle d'un pointeur qui ne contient aucune adresse) sont tous 2 définis dans le fichier d'en-tête `stddef.h`.
- Il existe en C un opérateur ternaire, dont la syntaxe est `cond ? v : f`, ce qui signifie qu'il renvoie la valeur `v` si la condition `cond` est vraie (c'est-à-dire si sa valeur est un entier non nul), et la valeur `f` sinon.
- Il existe également en C des opérateurs booléens : `!` (pour $\neg$, i.e., on prend la valeur de vérité opposée), `&&` (ET), et `||` (OU).
- Étant données 2 valeurs `x` et `y`, on peut tester si `x` et `y` sont égales à l'aide de `x == y`, et si elles sont différentes à l'aide de `x! = y`.
- Les opérateurs arithmétiques classiques existent évidemment en C : `/` (division), `+` (addition), `-` (soustraction), et `*` (multiplication). À noter que, si les 2 opérandes de `/` sont des entiers (i.e., `x/y` avec `x` et `y` entiers), alors c'est une division entière (euclidienne) qui sera réalisée, et le résultat sera donc le quotient entier de cette division. Par ailleurs, utiliser l'opérateur `%`, par exemple en écrivant `x%y` avec `x` et `y` 2 entiers ≥ 0, permet d'obtenir le reste de la division euclidienne de `x` par `y`.
- Un bloc `if(cond) ... else` permet de tester la condition `cond`, et d'exécuter une partie de code si elle est vraie (partie `if`), c'est-à-dire non nulle, et une autre partie de code si elle est fausse (partie `else`). À noter qu'une instruction `if` (ou `else`, ou `for`, ou `while`) n'exécute qu'une seule instruction (ou un seul bloc d'instructions) : si on souhaite en exécuter plusieurs dans ce cas, il faut les inclure dans un bloc (entre `{ }`), pour qu'elles soient considérées comme un seul bloc d'instructions.
- Un bloc `while(cond)` exécute 1 instruction (ou 1 bloc d'instructions) tant que la condition `cond` est vraie (c'est-à-dire de valeur non nulle). Un bloc `do ... while(cond)` fonctionne de façon similaire, si ce n'est que l'instruction (ou le bloc d'instructions) est toujours exécuté(e) une fois *avant* d'évaluer la valeur de `cond`.
- Un bloc `for(init;cond;next)` exécute une instruction (ou un bloc d'instructions) tant que la condition `cond` (par exemple `i<10`) est vraie (c'est-à-dire de valeur non nulle). Ici, `init` est l'ensemble des instructions exécutées à l'initialisation de la boucle (par exemple `i=0`), et `next` est l'ensemble des instructions exécutées à chaque nouveau passage dans la boucle (par exemple `i++`, qui incrémente la valeur de `i`).
- L'instruction `break` (en général déconseillée, à part dans un `switch`) permet de sortir immédiatement d'un bloc (de type `for`, `while`, etc.).
- L'instruction `return` permet de renvoyer une valeur à la fin d'une fonction, et stoppe immédiatement l'exécution de la fonction concernée.