TP de programmation en langage C Cédric Bentz/Daniel Porumbel

Séance de TP no 2 du 19/09/2024

Exercice 1 : sous-ensembles d'un ensemble de n entiers (en version revisitée)

Dans cet exercice, on se propose d'écrire une version modifiée du code de l'exercice 4 du TP 1 (sous-ensembles d'un ensemble de n entiers), qui utilisera une *structure* (ou *enregistrement*, ou encore *type composé*), déclarée via le mot-clé `struct` en C. Chaque valeur de ce type sera composée de 2 champs : un tableau d'entiers (type `int *`), et un entier (dont la valeur sera égale à la taille du tableau qu'il accompagne). On vous conseille également de faire usage d'un renommage de type (mot-clé `typedef`), par souci de simplicité.

Ainsi, définir un type composé `personne` où chaque variable contient 2 champs `nom` (de type char *) et `age` (de type `int`) s'écrira :

```
struct personne {
    char *nom;
    int age;
};
```

On pourra ensuite, étant donnée une variable `p` de type `struct personne`, accéder au champ `age` de cette variable via la syntaxe `p.age`.

L'objectif ici est donc de modifier le code de l'exercice 4 du TP 1, pour l'adapter à l'utilisation de ce nouveau type composé. Par exemple, la fonction récursive à écrire n'aura plus que 2 paramètres : d'une part, le tableau `t` et sa taille (nombre d'éléments), regroupés au sein d'une variable du nouveau type composé défini, et, d'autre part, l'indice courant de l'élément considéré dans le tableau (qu'on appellera à nouveau `i`, par convention).

Exercice 2 : le jeu du pendu

Dans cet exercice, on se propose d'implémenter une version C du célèbre jeu du pendu, à l'aide d'une interface simple au clavier. Le programme prendra en paramètre le nom du fichier texte où il ira chercher un mot à deviner de façon aléatoire (sans chiffres ni lettres accentuées, mais potentiellement avec des tirets). Un fichier de mots vous sera fourni : la première ligne contient le nombre total n de mots contenus dans le fichier (un mot par ligne), et il faudra donc choisir aléatoirement un mot parmi ces n mots (autrement dit, choisir aléatoirement un numéro de ligne entre 1 et n).

Une fois le mot à deviner choisi, le programme invite le joueur à saisir une lettre. À chaque fois que le joueur saisit une lettre (via `scanf`) :

- Le programme vérifie si la lettre est bien une lettre (un caractère compris entre 'a' et 'z' ou entre 'A' et 'Z', ou bien '-') ; si ce n'est pas le cas, il demande au joueur de resaisir une lettre (boucle `do ... while`).
- Si c'est bien le cas, le programme vérifie si la lettre a déjà été saisie par le joueur ou non (on différenciera majuscules et minuscules, pour simplifier), et, si oui, il l'indique, et lui demande de resaisir une lettre.
- Sinon, le programme ajoute la lettre à celles proposées par le joueur, et vérifie si elle apparaît ou non dans le mot à deviner.
- Si c'est le cas, il fait apparaître cette lettre comme "devinée" dans le mot à deviner (dans lequel les lettres devinées apparaissent en clair, et les autres sont représentées par "_"), par exemple à l'aide d'un tableau de booléens (entiers). Si le joueur a trouvé toutes les lettres du mot à deviner, le programme s'arrête en lui signifiant qu'il a gagné.
- Sinon, le programme met à jour le nombre d'erreurs commises par le joueur. Si ce nombre d'erreurs est égal à un nombre prédéfini (par exemple, 6), le programme s'arrête, après avoir indiqué au joueur le mot à deviner, et signalé qu'il vient d'être pendu (et a donc perdu).

Indications :

1. On aura intérêt à définir le nombre maximum d'erreurs comme une constante globale (à l'aide du mot-clé `const`), et le nombre d'erreurs déjà commises par l'utilisateur sous la forme d'une variable globale.
2. Avant chaque saisie d'une lettre par le joueur, on affichera l'état actuel du mot à deviner (c'est-à-dire qu'on montrera dans ce mot les lettres déjà devinées), la liste des lettres déjà saisies par le joueur, et le nombre d'essais restants pour deviner le mot. Il faudra donc garder en mémoire un certain nombre d'informations, à l'aide de variables.
3. Pour générer un nombre entier pseudo-aléatoire compris entre 0 et $n-1$ pour un certain n, on peut utiliser la fonction prédéfinie `rand()` (qui renvoie une valeur entière pseudo-aléatoire comprise entre 0 et une certaine constante `RAND_MAX`, et se trouve dans le fichier d'en-tête `stdlib.h`), puis appliquer un modulo n au résultat renvoyé. Pour initialiser la séquence de nombres pseudo-aléatoires, on préconise en général d'utiliser une valeur plus ou moins aléatoire, comme par exemple l'heure exacte (en secondes), passée en paramètre à la procédure `srand()` (qui se trouve également dans le fichier d'en-tête `stdlib.h`), de la façon suivante : `srand(time(NULL))` (où `time` est une fonction prédéfinie qui se trouve dans le fichier d'en-tête `time.h`).

4. Lire des informations dans un fichier se fait à l'aide d'un pointeur de type `FILE *` et de la fonction prédéfinie `fopen`, qui se trouve dans le fichier d'en-tête `stdio.h`, et qui prend en paramètre un nom de fichier (sous la forme d'une chaîne de caractères) et un mode d'ouverture du fichier (également sous la forme d'une chaîne de caractères ; par exemple, "r" pour ouvrir le fichier en lecture, et "w" pour l'ouvrir en écriture), de la façon suivante (ici, on veut lire le fichier "toto.txt") :

   ```
   FILE *f = fopen("toto.txt", "r");
   ```

   En appelant la procédure `fscanf` (qui fonctionne d'une façon similaire à `scanf`, excepté que son premier paramètre est le nom du fichier sur lequel on appelle la procédure), on lira le contenu de la ligne suivante du fichier de nom "toto.txt" (la lecture d'un fichier étant séquentielle).

5. Dans une fonction dédiée, on lira les mots du fichier dans une variable temporaire de type `char[]` et de longueur fixe arbitraire (sous la forme d'une "constante" assez grande, par exemple 100), mais en revanche on stockera le mot choisi aléatoirement dans le fichier, et que l'utilisateur devra deviner, dans un tableau de type `char *` qui aura exactement la taille requise. Pour rappel, les chaînes de caractères en C sont en réalité des tableaux d'éléments de type `char` qui se terminent par le caractère spécial `'\0'` : un mot de 5 lettres pourra donc être stocké dans un tableau de caractères de taille 6 ou plus. Une fois le mot aléatoire choisi, on récupèrera son nombre exact de lettres (par le biais de la fonction `strlen`, qui se trouve dans le fichier d'en-tête `string.h`), puis on allouera une zone mémoire de la bonne taille destinée à un tableau de type `char *`, par le biais de la fonction `malloc`. Enfin, on recopiera dans ce tableau nouvellement alloué le mot aléatoire choisi, à l'aide de la procédure `strcpy` (qui se trouve dans le fichier d'en-tête `string.h`, et prend en paramètres le tableau de caractères de destination et celui d'origine, dans cet ordre), puis on le renverra.

6. On peut concaténer 2 chaînes de caractères via la procédure `strcat`, qui est dans le fichier d'en-tête `string.h`, et qui prend en paramètres 2 chaînes de caractères (de type `char *`) : d'abord celle qui sera au début (et qui contiendra aussi le résultat), puis celle ajoutée à la fin.

7. On écrira une fonction dédiée qui sera chargée de gérer l'enchaînement des tours de jeu, et qui sera appelée à partir du `main`.

8. On n'oubliera pas de refermer l'accès au fichier contenant les mots une fois qu'il ne sera plus nécessaire d'y accéder, à l'aide de la fonction `fclose` (qui se trouve dans le fichier d'en-tête `stdio.h`, et qui prend en paramètre un pointeur de type `FILE *` vers le fichier à refermer).

9. On n'oubliera pas non plus de libérer (via la fonction `free`) tout espace mémoire inutilisé qui a été réservé par un appel à `malloc`.

10. Enfin, pour garantir le bon comportement de la procédure `scanf` tout au long du programme malgré des appels répétés, il est conseillé de vider le tampon de lecture clavier après tout appel à `scanf`. On peut le faire, par exemple, à l'aide du code suivant (qui utilise `stdio.h`) :

    ```
    int c; while ((c = getchar()) != '\n' && c != EOF);
    ```

Voici deux exemples de déroulement du programme attendu :

```
> pendu_fichier
Syntaxe : pendu_fichier [nom_du_fichier_de_donnees]

> pendu_fichier dictionnaire.txt
Vous avez 6 essais, a vous de jouer...

Mot : ______
Lettres saisies :
Nombre d'essais restants : 6
Entrer une lettre : "

Vous devez entrer une lettre de l'alphabet  !


Mot : ______
Lettres saisies :
Nombre d'essais restants : 6
Entrer une lettre : a

Desole, le mot recherche ne contient pas la lettre a.


Mot : ______
Lettres saisies : a
Nombre d'essais restants : 5
Entrer une lettre : e


Mot : e___e_
Lettres saisies : ae
Nombre d'essais restants : 5
Entrer une lettre : i
```

```
Mot : e_i_e_
Lettres saisies : aei
Nombre d'essais restants : 5
Entrer une lettre : o

Desole, le mot recherche ne contient pas la lettre o.


Mot : e_i_e_
Lettres saisies : aeio
Nombre d'essais restants : 4
Entrer une lettre : u

Desole, le mot recherche ne contient pas la lettre u.


Mot : e_i_e_
Lettres saisies : aeiou
Nombre d'essais restants : 3
Entrer une lettre : r


Mot : e_i_er
Lettres saisies : aeiour
Nombre d'essais restants : 3
Entrer une lettre : d

Desole, le mot recherche ne contient pas la lettre d.


Mot : e_i_er
Lettres saisies : aeiourd
Nombre d'essais restants : 2
Entrer une lettre : l


Mot : e_iler
Lettres saisies : aeiourdl
Nombre d'essais restants : 2
Entrer une lettre : x

Felicitations ! Vous venez de trouver le mot exiler !
```

Exercice 3 : un algorithme de programmation dynamique pour le problème du sac à dos

Le problème du sac à dos est un célèbre problème d'optimisation combinatoire, dans lequel on a un ensemble de n objets, chacun muni d'un poids entier $p_i \geq 0$ et d'une valeur entière $v_i \geq 0$. On a également un sac, dans lequel on doit emporter certains de ces objets. Néanmoins, le poids maximum d'objets qu'on peut transporter dans ce sac est limité par une quantité connue $P_{\max} > 0$, et on cherche donc à remplir le sac avec un ensemble d'objets de valeur totale maximum, mais sans dépasser un poids total de $P_{\max}$. Dans le cas général où les poids et les valeurs des objets sont des entiers positifs quelconques, ce problème est **NP**-difficile. Cependant, il existe un algorithme de programmation dynamique (c'est-à-dire basé sur des équations de récurrence) raisonnablement efficace pour le résoudre.

Dans cet exercice, on se propose donc d'implémenter cet algorithme en C pour résoudre de petites instances du problème, qui seront données sous la forme de fichiers texte que le programme devra lire. Le nom du fichier de données à considérer sera passé en paramètre lors de l'appel du programme.

Soit f la fonction définie de la façon suivante : $f(i,j) =$ valeur totale maximum d'un ensemble d'objets choisis parmi les i premiers, si le poids maximum supporté par le sac est égal à j, pour tout $i \in \{1,\ldots,n\}$ et tout $j \in \{0,\ldots,P_{\max}\}$. Clairement, la valeur qu'on cherche à calculer pour résoudre toute instance du problème du sac à dos est $f(n,P_{\max})$.

De façon évidente, on a, pour $i = 1$: $f(1,j) = v_1$ si $j \geq p_1$ (on emporte l'objet 1), et $f(1,j) = 0$ sinon (on ne peut pas emporter l'objet 1).

Pour $i > 1$, on peut montrer assez simplement que la valeur de $f(i,j)$ peut être calculée à l'aide d'équations de récurrence. Plus précisément, on a : $f(i,j) = f(i-1,j)$ si $j < p_i$ (on ne peut pas emporter l'objet i), et $f(i,j) = \max(f(i-1,j), v_i + f(i-1,j-p_i))$ sinon (on évalue, à l'aide de l'opérateur max, la meilleure solution entre ne pas emporter l'objet i dans le sac, dans le premier cas, et l'emporter dans le sac, dans le deuxième cas).

Le principe de l'algorithme est alors de calculer les valeurs de la fonction f, pour i allant de 1 à n (dans cet ordre). On pourrait considérer f comme une fonction récursive, les équations de récurrence devenant des appels récursifs. Néanmoins, pour implémenter cet algorithme en C de façon efficace, et éviter les appels récursifs en cascade qui augmenteraient considérablement son temps d'exécution, on va calculer et stocker les valeurs de la fonction f dans un tableau à 2 dimensions (qui aura donc n lignes et $P_{\max}+1$ colonnes), ligne par ligne. Appeler $f(i-1,j)$ lors du calcul de $f(i,j)$ reviendra alors simplement à lire une valeur dans ce tableau à 2 dimensions.

Pour manipuler les n objets, on définira un type composé, dont chaque valeur sera composée de 2 champs entiers (un pour p_i, et l'autre pour v_i). On pourra également stocker le nombre d'objets (n) et le poids maximum du sac ($P_{\max}$) dans une variable de ce type, par souci de simplicité et d'économie.

On écrira également une fonction chargée de lire les poids et valeurs des objets stockés dans le fichier de données considéré, et de les stocker dans un tableau. La place mémoire nécessaire à ce tableau sera allouée via un appel à `malloc` (et il ne faudra donc pas oublier d'appeler `free` ensuite). (Il en sera de même pour le tableau à 2 dimensions qui contiendra les valeurs de la fonction f, même si dans ce cas-là cela sera fait dans le `main`.)

Le format d'un fichier de données est le suivant : la première ligne contient seulement la valeur de n (le nombre d'objets), la deuxième ligne contient seulement la valeur de $P_{\max}$ (le poids maximum supporté par le sac), et chacune des n lignes suivantes contient le poids d'un objet puis sa valeur, séparés par une virgule. La lecture de ce fichier se fera par l'intermédiaire des fonctions `fopen`, `fscanf` et `fclose`, comme dans l'exercice précédent.

Enfin, on souhaite que la fonction qui lit les fichiers de données soit écrite dans un deuxième fichier `.c` (appelons-le `objet.c`), séparé de celui contenant le `main`. Elle sera au préalable déclarée dans un fichier d'en-tête (appelons-le `objet.h`), qui, en plus de contenir la déclaration de cette fonction, contiendra également la déclaration du type composé représentant un objet. Ce fichier d'en-tête devra donc lui-même être inclus à la fois dans `objet.c` et dans le fichier `.c` contenant le `main`. Pour compiler correctement l'ensemble de ces fichiers, on utilisera un fichier `Makefile`, et la commande `make`.

Voici deux exemples de déroulement du programme attendu :

```
> sad
Syntaxe : sad [nom_du_fichier_de_donnees]
Assertion failed!

Program: C:\agreg-info\sad.exe
File: C:\agreg-info\sad.c, Line 43

Expression: nb_args==2

> sad sad4.txt

La valeur optimale est 60 !
```

Indications :

1. Dans le cas d'un tableau d'entiers à 2 dimensions (donc de type `int **`), un appel à `malloc` permettra de réserver de l'espace mémoire pour un certain nombre d'éléments de type `int *`, mais pas pour l'intégralité du tableau à 2 dimensions. Pour chacun de ces éléments de type `int *`, il faudra ensuite refaire un appel à `malloc` pour réserver de l'espace mémoire pour un certain nombre d'éléments de type `int`.

2. Un autre point très important à noter est que l'allocation de l'espace mémoire nécessaire au tableau (appelons-le `tab_objets`) contenant les poids et les valeurs des objets sera effectuée à l'intérieur de la fonction qui sera définie dans `objet.c`. Cela nécessite donc de pouvoir modifier la *valeur* de la variable `tab_objets` (c'est-à-dire du pointeur `tab_objets`, dont la valeur sera passée en paramètre de cette fonction), et pas seulement la valeur référencée par ce pointeur.

3. Pour inclure dans un fichier `.c` un fichier d'en-tête (disons `toto.h`) qui ne fait pas partie de la librairie standard C, il ne faut pas écrire `#include <toto.h>`, mais `#include "toto.h"`.

4. On utilisera le mécanisme d'assertion pour vérifier que, lors de l'appel du programme, le nom du fichier de données à utiliser est bien fourni (et qu'il n'y a qu'un fichier). En C, cela nécessite d'utiliser l'instruction `assert`, qui se trouve dans le fichier d'en-tête `assert.h`. La syntaxe est alors la suivante : `assert(cond);`. Cela signifie simplement que, si la condition `cond` est vraie (c'est-à-dire de valeur non nulle), le programme continue son exécution normale. En revanche, si elle est fausse, le programme s'arrête, et indique quelle assertion fausse a entraîné cet arrêt (cf l'exemple en page 7, où il manque le nom du fichier).

5. Un fichier `Makefile` est constitué d'une ou plusieurs règles de la forme suivante (penser à bien faire attention à la syntaxe, y compris à la tabulation au début de la ligne des `commandes`) :

   ```
   cible: liste des dépendances
       commandes à exécuter
   ```

   Lors d'un appel au programme `make` (ou à une de ses déclinaisons, comme `mingw32-make`), le fichier dont le nom est passé en paramètre du `make` avec l'option `-f` (ou, à défaut, le fichier de nom `Makefile`) sera parcouru. Lors de ce parcours, le programme `make` évalue **la première règle rencontrée** (ou celle dont le nom est spécifié en paramètre du `make`, le cas échéant). L'évaluation d'une règle se fait récursivement :

   (a) Les dépendances sont d'abord analysées : si une dépendance est en réalité la cible (membre de gauche) d'une autre règle du fichier, cette règle est à son tour évaluée,

   (b) Quand toutes les dépendances ont été analysées (et les règles associées évaluées, si besoin), et si la cible (qui est un fichier) est plus ancienne qu'au moins une de ses dépendances (qui sont des fichiers), les commandes associées à la règle sont exécutées.

Il est à noter que, dans un fichier `Makefile`, on peut également utiliser des macros (l'équivalent de "constantes" C), à l'aide de la syntaxe suivante : `NOM_MACRO = VALEUR`. Dans le reste du fichier, il faudra alors écrire `$(NOM_MACRO)` pour pouvoir utiliser cette macro.

Voici un petit exemple de fichier `Makefile`, qui permet de compiler en deux temps le fichier `liste_pays.c` de l'exercice 1 du TP 1 (noter la macro CC pour définir le nom du **c**ompilateur **C** à utiliser, et les deux autres macros qui servent à définir l'ensemble des options à utiliser lors de la compilation et de l'édition de liens, respectivement) :

```
CC = gcc
OPTIONS_COMPILATION = -Wall
OPTIONS_EDITION_LIENS = -Wall

liste_pays: liste_pays.o
    $(CC) -o liste_pays liste_pays.o $(OPTIONS_EDITION_LIENS)

liste_pays.o: liste_pays.c
    $(CC) -o liste_pays.o -c liste_pays.c $(OPTIONS_COMPILATION)
```

Cet exemple illustre plusieurs points importants. D'abord, l'option `-Wall`, qui sert à demander l'affichage de l'ensemble des avertissements (*warnings*). Ensuite, le fait que la première règle précise comment créer le fichier exécutable à partir du fichier `.o` (fichier objet), ce qui correspond à l'étape de l'édition des liens, alors que la deuxième règle précise comment créer ce fichier `.o` à partir du fichier `.c` (fichier source), ce qui correspond à l'étape de compilation du fichier source proprement dite. Il faut noter à ce propos l'utilisation de l'option `-c` pour n'effectuer que la compilation (sans la phase d'édition des liens). Plus précisément, l'utilisation de cette option entraîne l'appel du préprocesseur C (qui se charge de remplacer automatiquement les directives qui lui sont destinées, c'est-à-dire celles qui commencent par le caractère `#`, par le code source correspondant), suivi de la compilation proprement dite.

Bien sûr, dans cet exemple, l'utilisation de deux règles distinctes n'est en réalité pas nécessaire, puisqu'on aurait pu écrire directement :

```
liste_pays: liste_pays.c
    gcc liste_pays.c -o liste_pays -Wall
```

Néanmoins, si plusieurs fichiers .c, et donc plusieurs fichiers .o, sont nécessaires pour produire le fichier exécutable (comme c'est le cas dans cet exercice), cet exemple a le mérite de détailler comment on peut compiler chacun des fichiers `.c` séparément pour produire un fichier `.o`, puis assembler tous les fichiers `.o` pour produire un fichier exécutable.

En effet, un fichier objet est le résultat de la compilation d'un fichier source, et contient les instructions machines associées à chaque fonction du fichier source, ainsi que la liste des fonctions appelées qui ne se trouvent pas dans le fichier source (par exemple, des fonctions des librairies standards comme printf, scanf, malloc, etc., ou des fonctions que vous avez vous-même écrites mais qui se trouvent dans d'autres fichiers sources). La phase d'édition des liens consiste justement à relier, à l'intérieur de chaque fichier objet, chaque appel de fonction non définie dans ce fichier avec le code correspondant (qui se trouve donc dans un autre fichier objet), pour produire un fichier exécutable.

6. On justifiera en détails dans les commentaires du code le fait que la complexité en temps de cet algorithme de programmation dynamique est $O(n \times P_{\max})$, et on expliquera pourquoi cet algorithme n'est donc pas un algorithme polynomial. (On rappelle à toutes fins utiles qu'un algorithme polynomial est un algorithme dont la complexité en temps est polynomiale en fonction de la taille du codage des données d'entrée.)