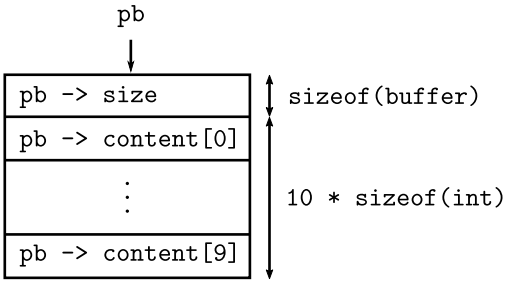
Programmation C
TP n° 10

Dans ce TP, il est attendu de séparer le programme en plusieurs fichiers : un pour le code des fonctions décrites, un fichier `.h` correspondant et dernier fichier qui contiendra uniquement la fonction `main`.
Il est possible de définir en C des types de structures dont le dernier champ est ce qu'on appelle un *tableau flexible*, un tableau sans spécification de taille. La taille de la structure est dans ce cas la taille nécessaire pour stocker tous les champs qui précèdent ce champ final.

```
typedef struct{
    size_t size;
    int content[];
} buffer;
```

Il est cependant possible de demander l'allocation d'une zone mémoire de taille *plus grande* que la taille de la structure elle-même. Les champs de la structure qui précèdent le tableau flexible seront stockables au début de cette zone. Le nom de champ du tableau flexible permettra d'accéder en lecture et en écriture à la zone allouée excédentaire :




```
  buffer *pb = malloc (sizeof (buffer) + 10 * sizeof(int));
  pb -> size = 10;
  for (int i = 0; i < 10; i++) {
    pb -> content[i] = i;
  }
```

Noter l'avantage de ce choix d'implémentation : les données d'une structure de type `buffer` sont en un seul bloc en mémoire, ce qui permet par exemple de les écrire dans un fichier en un seul appel de la fonction `fwrite`. La libération d'une structure allouée ne nécessite qu'un seul `free`.
Dans les fonctions qui suivent, le paramètre `pb` sera toujours supposé pointer vers une structure de type `buffer` correctement initialisée au sens suivant : la taille d'allocation de cette structure est toujours exactement égale à `sizeof (buffer) + pb -> size * sizeof(int)`.

Exercice 1 : Affichage

Écrire une foncion `void print_buffer(buffer *pb)` affichant les valeurs du tableau flexible d'un buffer en les séparant par des espaces, suivi d'un retour à la ligne final. Cette fonction vous sera utile pour tester les autres.

Exercice 2 : Allocation

Écrire `buffer *alloc_buffer(size_t size)` allouant un buffer correctement initialisé dont le tableau flexible pourra contenir exactement `size` entiers, en renvoyant l'adresse d'allocation.

Tester votre fonction en ajoutant dans un `main` les instructions nécessaires à la lecture d'un entier `n` depuis le terminal, l'allocation d'un buffer pouvant contenir `n` données, puis le remplissage de ce dernier à partir de `n` entiers saisis dans le terminal.

Exercice 3 : Sauvegarde

Écrire `void write_buffer (buffer *pb, const char *file_name)`. Cette fonction doit ouvrir en écriture un fichier binaire de nom `file_name`, et y sauvegarder en un seul bloc tout le contenu de l'espace mémoire alloué pour la structure d'adresse `pb`. Utilisez un seul appel de `fwrite`.

Exercice 4 : Chargement

Écrire `buffer *read_buffer (const char *file_name)`. Cette fonction suppose que `file_name` est le nom d'un fichier contenant la sauvegarde d'un buffer effectuée à l'aide de la fonction précédente. Elle doit allouer et reconstruire le contenu de ce buffer, puis renvoyer l'adresse d'allocation.

Indications. Commencez par allouer une structure de taille `sizeof(buffer)` (via `malloc`, ou via une variable locale de type `buffer`), et contentez-vous de transférer dans cette structure les `sizeof(buffer)` premiers octets du fichier par un appel de `fread`. La valeur du champ `size` de la structure vous indiquera quelle est la taille totale d'allocation nécessaire pour recomposer le buffer tout entier. Si vous vous êtes servi de `malloc`, veillez à éviter toute fuite de mémoire.

Exercice 5 : Ajout de données à une sauvegarde existante

Écrire `void append_buffer (buffer *pb, const char *file_name)`. Cette fonction suppose à nouveau que `file_name` est le nom d'un fichier contenant la sauvegarde d'un buffer. Elle doit modifier cette sauvegarde de manière à ajouter toutes les données de `pb -> content` *après* celle du tableau flexible du buffer sauvegardé, tout en mettant bien sûr à jour la valeur du champ `size` de la sauvegarde.

Indications. Le fichier doit être ouvert en mode mise à jour (`"r+"`). Comme dans la fonction `read_buffer`, le nombre de données du buffer sauvegardé peut être lu en début de fichier et mémorisé dans une structure de taille `sizeof(buffer)`. Le champ `size` de la structure en mémoire peut être mis à jour, et la structure réécrite en début de fichier sans altérer les données du champ `content` de la sauvegarde (`rewind`, `fwrite`). On peut ensuite se déplacer dans le fichier après les données du buffer sauvegardé (`fseek`), puis écrire les données à ajouter (`fwrite`).

Les `main` ci-dessous vous permettra de tester les fonctions des trois derniers exercices.

```c
int main () {
  buffer *pb = alloc_buffer (10);
  for (int i = 0; i < 10; i++) {
      pb -> content[i] = i;
  }
  print_buffer (pb);
  // 0 1 2 3 4 5 6 7 8 9

  // tests exercices 3 et 4
  write_buffer (pb, "save.dat");
  free (pb);
  pb = read_buffer ("save.dat");
  print_buffer (pb);
  // 0 1 2 3 4 5 6 7 8 9

  // test exercice 5
  append_buffer (pb, "save.dat");
  free (pb);
  pb = read_buffer ("save.dat");
  print_buffer (pb);
  // 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

  free (pb);
  return 0;
}
```