TP n°5



I. Afficher quelques adresses sur la pile

1 Que fait ce programme ? On notera le spécificateur de format pour afficher un pointeur : `%p`.

2 Représenter l'évolution de la pile d'appels de ce programme au cours de son exécution.

3 Exécuter ce programme sur votre ordinateur. La pile grandit-elle dans le même sens que les adresses mémoire ou dans le sens inverse ? Que se passe-t-il si vous le relancez ?

```c
void affiche1() {
  int k = 108;
  printf("Haut de pile %p\n", &k);
}

void affiche2() {
  affiche1();
  int a = 42;
  printf("Milieu de pile %p\n", &a);
}

void affiche3() {
  affiche2();
  int b = 421;
  printf("Bas de pile %p\n", &b);
}

int main() {
  affiche3();
  return 0;
}
```

Avertissements de compilation

La manipulation de pointeurs peut s'avérer délicate et donner lieu à des erreurs subtiles. Pour la suite de ce TP et pour la suite de la prépa, nous utiliserons d'autres flags de compilation pour afficher davantage d'avertissements et ainsi être plus vigilant. La nouvelle ligne de compilation à connaître par cœur est la suivante :

```
gcc tp5.c -o tp5 -Wall -Wextra -pedantic
```


II. Premiers pas avec les pointeurs

Visualisation étape par étape de la mémoire

Pour cette partie exceptionnellement, on ira sur le site https://pythontutor.com/visualize.html#mode=display. Cliquez sur le menu déroulant pour sélectionner le langage C, écrivez votre programme puis suivez l'exécution.

4 On souhaite réaliser la division euclidienne d'un entier a par un autre entier b. Proposer une fonction permettant de récupérer à la fois la valeur du quotient et la valeur du reste à l'aide d'un pointeur.

5 De même, proposer et implémenter une fonction permettant de récupérer la valeur minimum et la valeur maximum d'un tableau d'entiers non vide donné, à l'aide d'un pointeur.

6 Implémenter une fonction `alloue_tab` permettant d'allouer un tableau de 108 entiers sur le tas, et renvoyant le pointeur obtenu. L'appeler depuis `main` et stocker le pointeur dans une variable `p`. Modifier quelques cases du tableau. Que se passe-t-il si on écrit ensuite `p = alloue_tab()` ?

7 Quels sont les avantages de l'allocation dynamique dans le tas ?

III. Entrée d'un programme

Jusqu'à présent la fonction `main` s'écrivait sans aucun paramètre. Si après compilation on produisait un exécutable **prog**, on le lançait avec la commande **./prog**.

```
./prog
```

Il est possible de donner des paramètres à la fonction `main`, ce qui permettra de lancer votre programme avec des valeurs au lancement de l'exécution. Comme ceci :

```
./prog 108 CIV
```

- `argc` désigne le nombre de valeurs reçues. Le programme (ici : **./prog**) compte comme une valeur.
- `argv` désigne les valeurs reçues (y compris le **./prog**) sous forme d'un tableau de chaînes de caractères. Attention, même 108 sera considéré comme une chaîne de caractères et non pas un entier !

Exemple

```c
int main(int argc, char **argv) {
  printf("J'ai reçu %d valeurs.\n", argc);
  printf("Je suis %s !\n", argv[0]);
  return 0;
}
```

8 Dans un nouveau fichier C, écrire cette fonction `main` et vérifier l'affichage produit. Modifier cette fonction pour que votre programme prenne deux entiers en entrée et affiche le plus grand des deux. On utilisera la fonction `atoi` pour convertir une chaîne de caractères en entier, et on affichera un message d'erreur si le mauvais nombre de valeurs est passé en entrée de votre programme.

IV. Type structuré en C

Les **types structurés** permettent d'implémenter des **enregistrements** en C, c'est-à-dire une collection d'éléments appelés **champs**, ayant chacun un nom et un type propre. L'exemple ci-contre stocke un nom d'élève, sa moyenne et son régime dans un seul type.

```c
struct eleve {
  char* nom;
  float moyenne;
  bool est_externe;
};
```

On écrit ces lignes en dehors de toute fonction pour définir un nouveau type nommé `struct eleve` qui pourra être utilisé par la suite. On peut remarquer qu'il n'y a pas de symbole `=` avant l'accolade, que les champs sont typés explicitement et se terminant par des **points-virgules** (et non des virgules). Et le point-virgule final.

Il est lourd de manipuler un type nommé `struct eleve`, c'est pourquoi il est très courant de **renommer** les types structurés à l'aide du mot-clef `typedef`. Ici par exemple, on le renomme en `eleve`, tout court.

```c
typedef struct eleve eleve;
```

9 Implémenter un nouveau type structuré pour manipuler des fractions en stockant deux entiers, le premier nommé `numerateur` et le second nommé `denominateur`. Renommer ce type `frac`.

Il est possible d'initialiser la valeur d'un type structuré directement lors de sa définition, comme pour un tableau :

```
<type> <nom_variable> = (<type>) {.<champ1> = <valeur1>, ..., .<champk> = <valeurk>};
```

Par exemple avec notre type structuré `eleve`, on peut initialiser tous les champs dès la création ainsi :

```
eleve sebastien = (eleve) {.nom = "Sebastien", .moyenne = 16.4, .est_externe = false};
```

Il est également possible de déclarer un type structuré sans valeur initiale (`eleve sebastien;`), puis d'initialiser ses champs un par un plus tard. Les types structurés profitent du passage par valeur, une fonction peut renvoyer un tel type ou prendre un tel type en paramètre et une copie sera effectuée.

10 Implémenter les fonctions de prototype `frac zero()` et `frac un()` qui renvoient une fraction correspondant aux entiers 0 et 1 respectivement. Implémenter également une troisième fonction `frac nouvelle_frac(int num, int denom)` pour générer une fraction à partir de deux entiers.

On accède à un champ en indiquant le nom de l'enregistrement, puis le nom du champ après un point. On peut par exemple récupérer la moyenne de Sebastien avec `sebastien.moyenne`. On peut également modifier cette valeur.

11 Implémenter une fonction de prototype `void affiche_frac(frac f)` permettant d'afficher une fraction de manière claire, par exemple de la forme `2 / 7`.

12 Implémenter une fonction de prototype `frac produit(frac f1, frac f2)` qui renvoie une fraction représentant le produit de deux fractions. Faire de même pour l'inverse, la division et la somme.

13 Implémenter une fonction de prototype `bool frac_egales(frac f1, frac f2)` qui vérifie si deux fractions ont la même valeur. Deux fractions peuvent être égales avec des champs différents.

V. Piles et files à l'aide d'un tableau

Implémentation d'une structure de pile

14 Implémenter un type structuré que l'on renommera `pile` contenant un entier nommé `taille`, un entier `remplissage` et un tableau d'entiers `tab` sous forme d'un pointeur vers un entier.

On propose les deux fonctions suivantes pour allouer un tel enregistrement dynamiquement sur le tas, et pour pouvoir le libérer. On remarque qu'il est possible d'écrire `p->taille` (plus usuel) plutôt que `(*p).taille` quand on manipule un pointeur vers un enregistrement. Cette syntaxe alternative est à connaître.

```c
pile *nouvelle_pile(int taille) {
  pile *p = malloc(sizeof(pile));
  p->taille = taille;
  p->remplissage = 0;
  p->tab = malloc(sizeof(int) * taille);

  return p;
}
```

```c
void libere_pile(pile *p) {
  free(p->tab);
  free(p);
}
```

15 Implémenter ces fonctions. Faire un dessin illustrant pas à pas ce qu'il se passe en mémoire. Était-on obligé d'allouer `tab` sur le tas ? Et la structure de pile ? Dans la libération, pourquoi ne peut-on pas écrire juste `free(p)` pour libérer une telle pile ?

16 Implémenter une fonction de prototype `void empile(pile *p, int k)` qui ajoute l'élément `k` au sommet de la pile. Il faudra pour cela l'ajouter dans le tableau et mettre à jour le champ `remplissage`. Vous pouvez vérifier que la pile n'est pas pleine à l'aide d'un `assert`.

17 Implémenter une fonction `int depile(pile *p)` qui renvoie l'élément du sommet de pile et l'enlève de la pile. Il n'y aura pas besoin de modifier la valeur du tableau, uniquement le champ `remplissage`. Vous pouvez vérifier que la pile n'est pas vide à l'aide d'un `assert`.

18 Implémenter une fonction de tests vérifiant que cette structure de pile fonctionne comme prévu. Empiler plusieurs valeurs et vérifier que les valeurs dépilées sont celles attendues.

Implémentation d'une structure de file

19 En se basant sur la partie précédente et l'implémentation d'une structure de pile, proposer une structure de file (comme une file d'attente) pour des entiers. Tout comme pour une pile, il faut écrire une fonction d'allocation, de libération, ainsi que deux fonctions permettant d'enfiler et défiler un élément. On pourra utiliser un entier `debut` et un entier `fin` désignant les deux bords.