Université Grenoble Alpes
L3 Informatique

Année 2024-2025

Architectures logicielles et matérielles
Semaine 2 TP (2)

L'objectif de ce TP est d'approfondir les révisions sur les accès via pointeurs, les opérations bit à bit et masquages.

Exercice 1.

En question préliminaire, vous avez prévu le comportement (initialisations, modifications) de la portion de code C ci-dessous :

```
int *q;
int x=5,y;
int *p = &x;
*p = 3;
p = &y;
*p += 1;
```

Ecrire un programme qui vous permet de vérifier vos prévisions, compiler et exécuter ce programme : vérifier que toutes les valeurs correspondent bien à ce qui a été prévu (sous gdb de préférence).
En particulier assurez-vous d'avoir bien compris à tout moment si les valeurs suivantes sont **définies ou imprévisibles** et pourquoi : la valeur de `q`, la valeur de `*q`, la valeur de `p`, la valeur de `*p`.

Exercice 2.

Revenons sur une variante d'un exemple simple vu en séances de mise à niveau.

Q1. Selon vous, avec les déclarations suivantes :

```
uint16_t tab[8] = {0x1234, 0x5678, 0x9ABC, 0xDEF1, 0x1FDE, 0xCBA9,
                   0x8756, 0x4321};
uint16_t *p;
uint8_t *q;
```

quels sont les **types**, et les **tailles** de `tab`, `p`, `q`, `*p`, `*q` ?

Q2. Vérifier vos prévisions en utilisant ces déclarations suivies de l'instruction suivante (ne pas oublier d'inclure stdint.h) :

```
 printf("Tailles utilisees : \
     uint16_t -> %lu, tab -> %lu, uint8_t -> %lu, uint16_t * -> %lu, \
     uint8_t * -> %lu\n",
     sizeof(uint16_t), sizeof(tab), sizeof(uint8_t), sizeof(uint16_t *),
     sizeof(uint8_t *));
```

Q3. Selon vous, **qu'obtiendrez-vous** à l'exécution du code suivant, et pourquoi ?

```
p = tab;
q = (uint8_t *)p;
printf("Affichage tab, parcours avec p :\n");
for (i = 0; i < 8; i++) {
  printf("tab[%d] = %x ; *(p+%d) = %x\n", i, tab[i], i, *(p+i));
}
printf("Affichage tab, parcours avec q :\n");
for (i = 0; i < 8; i++) {
  printf("*(q+%i) = %x\n", i, *(q+i));
}
```

Q4. Vérifier vos prévisions en compilant et exécutant ce code, assurez-vous d'avoir bien

compris quelles valeurs sont affichées et pourquoi.

Exercice 3.

Q1. Ecrire une **fonction**

`int consulte_bit(uint16_t x, unsigned int k)` : renvoie la valeur (0 ou 1) du bit numéro `k` dans l'entier 16 bits `x`

Tester : Quelle est la valeur du bit numéro 9 dans l'entier 37797 (c'est-à-dire 0x93A5) ? Quelle est la valeur du bit numéro 10 dans ce même entier ?

Q2. Ecrire une **fonction**

`void modifie_bit(uint16_t *x, unsigned int k, int v)` : place la valeur `v` à l'emplacement du bit numéro `k` dans l'entier 16 bits `x`

Tester : Que devient l'entier 37797 (c'est-à-dire 0x93A5) si on insère un 0 à l'emplacement du bit numéro 9 ? Et que devient ce même entier si on insère un 1 à l'emplacement du bit numéro 10 ?

Exercice 4. *(à **finir lors de l'APNEE** si besoin)*

Dans cet exercice, on considère deux réalisations pour le *calcul de nombres premiers selon le principe du crible d'Ératosthène*, avec des codages différents.
On rappelle que le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné N. L'algorithme procède par élimination : au début tous les entiers naturels entre 2 et N sont dans le crible, et on supprime successivement les entiers non premiers :

```
x ← 2
Tant que x n'a pas atteint N
  Si x est encore présent dans le crible alors en supprimer tous ses multiples
  x ← x+1
Fin
```

Q1. Dans un premier temps, le **crible** sera simplement **modélisé par un tableau de 0 et de 1** (chacun codé sur un entier) : crible[k] est à **1** si k est encore dans le crible, et il est à **0** si k n'est plus dans le crible.

Ecrire les **fonctions** suivantes :

`int present(int k, int *c)` : renvoie 1 si l'entier k est dans le crible c

`void suppression(int k, int *c)` : supprime l'entier k du crible c

`void construit_crible(int *c, int n)` : construit le crible `c` de façon à caractériser les nombres premiers inférieurs à `n` (en utilisant les fonctions précédentes bien sûr).

Tester en utilisant le programme suivant :

```
int main(int argc, char **argv) {
  int *crible, x, k;
  if (argc == 2) {
    x = atoi(argv[1]);
    crible = (int *)malloc(x*sizeof(int));
    construit_crible(crible, x);
    printf("Nombres premiers jusqu'a %d : \n", x);
    for(k=0; k<x; k++)
       if (present(k,crible)) printf("%d, ", k);
    printf("\n");
  }
  else {
    fprintf(stderr, "./crible <nombre>\n");
    return 1;
  }
  return 0;
}
```

Q2. Nous allons maintenant optimiser le codage en écrivant une autre version dans laquelle **la valeur 1 ou 0** permettant de savoir si un entier est présent ou pas dans le crible sera **codée sur un bit**.
Pour représenter le **crible**, on utilise alors un tableau d'éléments de type *uint32_t*, tableau de taille suffisante pour disposer d'autant de bits que d'entiers à caractériser dans le crible.

Ecrire les nouvelles **fonctions** suivantes :

```
int present(int k, uint32_t *c){   // l'entier k "est dans" le crible c
  unsigned int indice, place;
  indice =          // completer : indice est l'indice de l'element a
                    // consulter dans le tableau c
  place =           // completer : place est le numéro du bit dans cet element
  return     // completer
}
```

`void suppression(int k, uint32_t *c)` : sur le même principe que ci-dessus, supprime l'entier k du crible c

`void construit_crible(uint32_t *c, int borne, int n)` : construit le crible `c` de façon à caractériser les nombres premiers inférieurs à `n` (en utilisant les fonctions précédentes bien sûr). Le paramètre `borne` est la taille du tableau c.

Tester en utilisant le programme suivant :

```
int main(int argc, char **argv) {
  int x, borne;
  uint32_t *crible;
  if (argc == 2) {
    x = atoi(argv[1]);
    borne = x/32 + 1;
    crible = (uint32_t *)malloc(borne*sizeof(uint32_t));
    construit_crible(crible, borne, x);
    printf("Nombres premiers jusqu'a %d : \n", x);
    // mettre instructions pour affichage...
  }
  else {
    fprintf(stderr, "./crible <nombre>\n");
    return 1;
  }
  return 0;
}
```