Durée 3 h Documents non autorisés

Sujet d'examen 1

# Pratique du C

Décembre 2011

## Introduction

Écrivez lisiblement et n'hésitez pas à commenter votre code en langage C. Vous ne pouvez utiliser que les fonctions C dont le prototype est donné dans l'énoncé et celles dont vous donnez la définition dans vos copies.

Les sections sont indépendantes ; lisez l'énoncé complet avant de commencer à le résoudre.

## 1 Quizz

1. Donner la valeur des variables ra, rb, rc après l'exécution du programme :

```
#include <stdio.h>
int p1 (int a)
{
          a = a * 2;
          return a + 5;
}
int p2 (int *b)
{
          *b = *b * 2;
          return *b + 5;
}
int p3 (int *c)
{
          return p1 (p2 (c));
}
int main(void)
{
  int a = 2, b = 3, c = 4;
  int ra, rb, rc;
  ra = p1 (a);
  rb = p2 (&b);
  rc = p3 (&c);
  printf ("%d, %d, %d\n", ra, rb, rc);
  return 0;
}
```

**Correction.** Les variables sont ra=9, rb=11 et rc=31.

2. Soient les définitions suivantes :

```
char *t1[4] = { "Il est beau", "le lavabo", "Il est laid", "le bidet"} ;
int t2[9] = { 0,2,4,6,7,1,456,24,4 } ;
int *p = &t2[4] ;
```

Donnez le type et précisez, quand c'est possible, la valeur retournée par l'évaluation des expressions suivantes :

```
t1[2]
t1[2][3]
*t1
t2[2]
*t2
*p
(*p)+2
*(p+2)
t1[t2[2]]
```

**Correction.**

| | type | value |
|---|---|---|
| t1[2] | char * | &"Il est laid" |
| t1[2][3] | char | e |
| *t1 | char * | &"Il est beau" |
| t2[2] | int | 4 |
| *t2 | int | 0 |
| *p | int | 7 |
| (*p)+2 | int | 9 |
| *(p+2) | int | 456 |
| t1[t2[2]] | char * | &???? |

3. À chaque compilation, le compilateur gcc défini des macros suivant l'architecture le supportant. Par exemple,
   – sur un ordinateur Sparc géré par Solaris, la macro `sparc` est définie ;
   – sur un Pentium géré par Linux, la macro `linux` est définie ;
   – sur un PowerPC géré par Mac OS X, la macro `darwin` est définie.
   Un utilisateur souhaite disposer d'un tableau `architecture` permettant de représenter une chaîne de caractères indiquant le type d'architecture (sparc, linux, darwin). Si aucune de ces macros n'est définie, la chaîne donnant le type de machine est `unknown`.

   Construisez un fichier d'entête permettant de définir cette variable à l'aide de directives au compilateur.

   **Correction.**

```
char architecture[] =
#ifdef sparc
      "sparc"
#else
#ifdef linux
      "linux"
#else
#ifdef darwin
      "darwin"
#else
     "unknown"
#endif /* darwin */
#endif /* sparc */
#endif /* linux */
;
```

## 2 Recherche binaire d'un objet dans un tableau

Donner la définition de la fonction `bsearch` de prototype :

```
void *bsearch
(
    const void *key, const void *base, size_t nel,
    size_t size, int (*compar)(const void*, const void*)
);
```

Cette fonction recherche de manière dichotomique la clef `key` dans le tableau base.

Elle suppose que les objets du tableau base sont rangés dans l'ordre croissant.

Les arguments de cette fonction sont :
- `key` : élément à rechercher (clef de recherche) ;
- `base` : adresse de l'élément 0 du tableau à scruter ;
- `nel` : nombre d'éléments dans le tableau ;
- `size` : taille de chaque élément dans le tableau ;
- `compar` : pointeur sur une fonction de comparaison définie par l'utilisateur.

La fonction `bsearch` fait appel à la fonction pointée par `compar` avec comme premier argument la clef de recherche et comme second argument l'élément du tableau à comparer avec la clef ; cette fonction retourne un entier donnant le résultat de la comparaison :
- valeur négative : la clef est plus petite que l'élément en cours ;
- valeur nulle : la clef et l'élément en cours sont égaux ;
- valeur positive : la clef est plus grande que l'élément en cours.

En cas de succès, `bsearch` retourne l'adresse de la première entrée identique à la clef, sinon `bsearch` retourne NULL.

Par exemple, le code suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int (*fptr)(const void *, const void *);

int
compare1
(const int *n1, const int *n2)
{
    return(*n1 - *n2);
}

int
compare2
(const char *str1, const char *str2)
{
   return strlen(str1) - strlen(str2);
}

int
main
(void)
{
    int tab1[9] = {123, 234, 345, 456, 567, 678, 789, 890, 901};
```

```
    int key1 = 890;
    char *tab2[5] = {"ABCDEF", "PHM", "EB", "DA", "DF"};
    char *key2 = "DA";

    if ( bsearch(&key1, tab1, 9, sizeof(int), (fptr) compare1) != NULL )
       printf("%d est dans tab1\n", key1);
    else
       printf("%d n'est pas dans tab1\n", key1);

    if ( bsearch(&key2, tab2, 5, sizeof(char *), (fptr) compare2) != NULL )
       printf("%s est dans tab2\n", key2);
    else
      printf("%s n'est pas dans tab2\n", key2);
    return 0;
}
```

produit l'exécution :

```
% ./a.out
890 est dans tab1
DA est dans tab2
```

**Correction.**

```
void           *
                bsearch
                (
                        const void *key, const void *base, int nel, int size,
 int (*compar) (const void *, const void *)
)
{
int             ordre;
void           *res;

nel--;

while (1) {
/* comparaison de la clef avec l'\'el\'ement courant */
res = (void *) (((char *) base) + size * nel / 2);
ordre = compar(key, res);

if (!ordre)
return res;

/* la condition d'arret */
if (!nel)
break;

nel /= 2;

if (ordre > 0)
base = res;
}
```

```
return NULL;
}
```

# 3 Implantation de matrices de dimensions variables

La taille d'une matrice pleine de dimensions variables n'est pas connue à la compilation mais seulement lors de sa création lors de l'exécution. Les coefficients de cette matrice sont des entiers machines signés.

On se donne la description des types suivante :

- le type `matrix_t` est un pointeur sur des objets définis suivant le modèle d'identificateur `matrix_m` ;
- un tel objet est :
  - soit un objet d'identificateur `zero` étant vrai si la matrice est nulle et faux dans le cas contraire,
  - soit un objet suivant le modèle d'identificateur `truematrix_m`.
- un objet de la famille d'identificateur `truematrix_m` est composé :
  - d'un entier signé d'identificateur `nblig` codant le nombre de lignes ;
  - d'un entier signé d'identificateur `nbcol` codant le nombre de colonnes ;
  - d'un pointeur sur des entiers signés d'identificateur `body`.

On souhaite disposer de l'opération d'addition de telles matrices : l'addition de deux matrices n'est possible que si ces matrices ont le même nombre de colonnes et le même nombre de lignes.

Une variable du type `matrix_t` vaut NULL si elle ne représente pas une matrice valide (par exemple, si on tente de construire une matrice avec une dimension négative ou si on tente d'additionner 2 matrices de tailles différentes).

**Question.**

1. Donnez les déclarations des types décrits ci-dessus.

   **Correction.**

   ```
   struct truematrix_m
   {
   unsigned int nblig ;
   unsigned int nbcol ;
   int *body ;
   } ;

   union matrix_m
   {
     int zero ;
     struct truematrix_m matrix ;
   } ;

   typedef struct truematrix_m truematrix_t;
   typedef union matrix_m *matrix_t ;
   ```

2. Donnez la définition d'une fonction de prototype

   ```
   matrix_t makenullmatrix ();
   ```

   qui construit une matrice zéro ;

   **Correction.**

   ```
   matrix_t
   makenullmatrix
   ```

```
(void)
{
 matrix_t res ;
 res =  (matrix_t) malloc(sizeof(union matrix_m));
 res->zero = TRUE ;
 return res ;
}
```

3. Donnez la définition d'une fonction de prototype

   `matrix_t makematrix (int, int);`

   qui construit une matrice dont le nombre de lignes est passé en premier paramètre et le nombre de colonnes en second (si un de ces entiers est négatif ou nul, on retourne `NULL`). Cette fonction réserve de l'espace pour les coefficients mais n'affecte pas cet espace.

   **Correction.**

```
#define TRUE (1==1)
#define FALSE !TRUE

matrix_t
  makematrix
(int nblig, int nbcol)
{
  matrix_t res ;

  if(nblig<=0 || nbcol <=0)
    return NULL ;

  res =  (matrix_t) malloc(sizeof(union matrix_m));
  res->matrix.nblig = nblig ;
  res->matrix.nbcol = nbcol ;
  res->matrix.body = (int*) malloc(nblig*nbcol*sizeof(int)) ;
  res->zero = FALSE ;

  return res ;
}
```

4. Donnez la définition d'une fonction de prototype

   `void killmatrix (matrix_t);`

   qui désalloue une matrice.

   **Correction.**

```
void
killmatrix
(matrix_t mat)
{
  if(!mat)
    return ;
  if((mat->zero==FALSE) && mat->matrix.body)
    free(mat->matrix.body) ;
  free(mat) ;
}
```

5. Donnez la définition d'une fonction de prototype

   `matrix_t addmatrices (matrix_t,matrix_t);`

qui retourne la matrice résultant de l'addition des matrices passées en paramètres. Si ces matrices sont de dimensions différentes, cette fonction retourne `NULL`. Si chaque coefficient de la somme de ces matrices est zéro, on retourne une matrice zéro.

**Correction.**

```
#include <stdlib.h>

matrix_t
addmatrix
(matrix_t a, matrix_t b)
{
  int pos,i,j, nblig, nbcol, null ;
  int *body ;
  matrix_t res ;
  if(!a || !b)
  /* a ou b n'est pas valide, la somme n'est pas valide */
   return NULL ;

  if(a->zero)
    return b ;
  if(b->zero)
    return a ;
  nblig = a->matrix.nblig ;
  nbcol = a->matrix.nbcol ;
  if( ! ((nblig==b->matrix.nblig) &&
         (nbcol==b->matrix.nbcol) ))
    return NULL ;

  body = (int*) malloc(nblig*nbcol*sizeof(int)) ;

  for(i=0 ; i<nblig ; i++)
    for(j=0 ; j<nbcol ; j++)
      {
 pos = i*nbcol+j ;
 body[pos] = a->matrix.body[pos]+b->matrix.body[pos] ;
      }

  null = TRUE ;
  for(i=0 ; i<nblig ; i++)
    for(j=0 ; j<nbcol ; j++)
      {
 pos = i*nbcol+j ;
 null = null && !body[pos] ;
      }

  if(null)
    {
      free(body) ;
      return makenullmatrix();
   }

  res = (matrix_t) malloc(sizeof(union matrix_m));
  res->matrix.nblig = nblig ;
  res->matrix.nbcol = nbcol ;
```

```
  res->matrix.body = body ;
  res->zero = FALSE ;
  return res ;
}

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

**Rappels :** L'allocation mémoire se fait par le biais de la fonction `malloc` et la désallocation par le biais de la fonction `free`.

# 4 Une implantation sommaire de la fonction printf

Dans cette section, on se propose d'implanter la fonction printf dans une architecture dans laquelle le passage des paramètres se fait par une pile (voir les indications en fin d'exercice).

Par ailleurs, on suppose ne disposer que d'une seule fonction externe d'affichage dont le prototype est `int putchar(int);` et qui écrit dans la sortie standard un caractère dont le code ASCII est passé en paramètre. Par exemple, pour afficher le caractère ~ on peut utiliser le code suivant :

```
#include<stdio.h>

int main(void){
  char c = '~' ;
  putchar( c ) ;
  return 0 ;
}
```

L'objectif est d'implanter la fonction de prototype : `void mprintf(const char *format, ...)` permettant d'afficher sur la sortie standard :

- des caractères ASCII ;
- des chaînes de caractères ;
- des entiers machines dans les bases décimale et binaire ;
- des entiers machines de Gauss nommés i.e. des nombres complexes dont les parties réelles et imaginaires sont des entiers machines et auxquels on associe une chaîne de caractères.

Cette fonction a un paramètre obligatoire et un nombre variable de paramètres.

Le paramètre obligatoire est constitué par une chaîne de caractères qui est composée de caractères ordinaires et de $0, 1$ ou plusieurs directives. Un caractère ordinaire est un caractère ASCII à l'exception du caractère %. Une directive commence par le caractère % et peut être de plusieurs formes :

- la directive %c indique que l'on souhaite afficher un caractère ;
- la directive %s indique que l'on souhaite afficher une chaîne de caractères ;
- la directive %d indique que l'on souhaite afficher un entier en base décimale ;
- la directive %b indique que l'on souhaite afficher un entier en base binaire (cette base est indiquée par la lettre minuscule b : l'entier décimal 2 est affiché comme $10b$) ;
- la directive %Gd indique que l'on souhaite afficher un entier de Gauss en base décimale (par exemple $2 + 2I$) ;
- la directive %Gb indique que l'on souhaite afficher un entier de Gauss en base binaire (par exemple $10b + 10bI$) ;
- toute autre lettre suivant un % est affichée (ce qui permet d'afficher le caractère ASCII %).

**Questions**

1. Donnez la définition d'une fonction `void PrintInt(int i,int b)` qui écrit sur la sortie standard l'entier $i$ dans la base $b \in \{2, 10\}$.
2. Donnez la définition d'une fonction `void PrintString(char *s)` qui écrit sur la sortie standard la chaîne de caractères associée à s.
3. Donnez la définition de la fonction `mprintf`.

**Indications.** Rappelons que les paramètres d'une fonction sont passés par la pile. Cette pile est composée de cellules dont la taille en octet correspond au type `int`, elle croit vers les adresses décroissantes et elle a la structure suivante :

| | |
|---|---|
| 0000 | ··· |
| | seconde variable locale |
| | première variable locale |
| | ancien pointeur de contexte |
| | adresse de retour |
| | premier paramètre |
| | second paramètre |
| FFFF | ··· |

Ainsi si `foo` est la première variable locale automatique définie dans la fonction appelée et que `ptr` est un pointeur sur cette variable, `ptr+1` pointe sur l'ancien pointeur de contexte.

De plus, lors de l'appel d'une fonction, ses paramètres sont empilés du dernier au premier et on note qu'un paramètre de type

- entier occupe une cellule ;
- pointeur occupe une cellule ;
- caractère occupe une cellule (sizeof(char) octet pour le caractère et sizeof(int)-sizeof(char) octets inutilisés ;
- entier de Gauss occupe 3 cellules, les champs sont empilés du dernier au premier.