CCI - Principes des langages de programmation
Examen - Correction

Christine Paulin, Julie Bernauer, Xavier Ripoche

4 Novembre 2004

Seuls les documents suivants sont autorisés :
- Polycopiés de cours, énoncés de TD distribués cette année
- Notes de cours et de TD manuscripts

L'examen dure 3 heures, il est composé de 6 pages.
Les programmes demandés devront être écrits en C.

1 Précédences (2 points)

Le langage C propose des opérateurs qui travaillent sur les entiers bit à bit. Il s'agit des opérateurs de décalage à droite `>>` et à gauche `<<`, de l'opération de conjonction (`et`) bit-à-bit notée `&`, de la disjonction (`ou`) exclusive bit-à-bit notée `^` et inclusive notée `|`. Ces opérateurs associent à gauche. Les opérateurs de décalage ont la plus forte précédence et l'opérateur `|` a la plus faible précédence. L'ordre des précédences de la plus forte à la moins forte est (en regroupant les opérations de décalage qui ont même précédence) :

$$\{>>, <<\}, \&, \hat{}, |$$

Donner le parenthésage et la valeur des expressions suivantes :

```
int x = 18;
int y = 3;
printf("%d\n", y | y << y ^ 5);
printf("%d\n", x ^ y & x | y ^ 1);
printf("%d\n", y >> 1 << 2 | x >> 1);
```

On rappelle que la représentation binaire de 18 (resp. 3) est 10010 (resp. 11). Le ou-exclusif vaut 1 si et seulement exactement l'une de ses opérandes est 1, alors que le ou-inclusif vaut 1 lorsqu'au moins une de ses opérandes est 1. Le décalage x<<y (resp. x>>y) lorsque x est un entier positif correspond à la multiplication (resp. la division) de x par 2^y.

Correction :

```
printf("%d\n", y | ((y << y) ^ 5));
printf("%d\n", (x ^ (y & x)) | (y ^ 1));
printf("%d\n", ((y >> 1) << 2) | (x >> 2));
```

Résultats : 31, 18, 13

2 Expressions régulières (1 point)

1. Donner une expression régulière qui permette de reconnaître des adresses électroniques valides. L'expression régulière devra reconnaître en particulier les adresses suivantes :

   `Christine.Paulin@lri.fr`
   `julie.bernauer@ibbmc.u-psud.fr`
   `ripoche@limsi.fr`
   `paulin@ie2.u-psud.fr`
   `master-m1@lri.fr`
   `postmaster@www.org`

On remarquera la présence de caractères alpha-numériques et des caractères spéciaux tels que `-`. La partie à la droite du caractère `@` est composée d'au moins un nom de domaine (`lri`, `u-psud`, `www` ...) suivi d'une extension de 2 ou 3 caractères alphabétiques (`fr`, `org` ...).

Correction : *On introduit une expression régulière pour les caractères alphabétiques en majuscule ou minuscule, une autre pour les noms qui peuvent comprendre des caractères alphabétiques , des nombres ou des symboles spéciaux. Finalement une adresse e-mail est composée de deux noms séparés par le symbole '@' et suivis d'un point et d'un suffixe à 2 ou 3 caractères.*

alpha = ['A'-'Z'] | ['a'-'z']
ident = alpha | ['0'-'9'] | '.' | '-' | '_'
mail = ident '@' ident '.' alpha alpha (alpha)?

3 Structures et Tableaux (4 points)

On souhaite manipuler des vecteurs à trois dimensions. Pour cela on choisit de définir le type des vecteurs par un tableau de dimension 3.

```
typedef int vect[3];
```

1. Ecrire une fonction `add_vect` qui prend en argument deux vecteurs v et w, et ajoute à v le vecteur w.
   Correction :

   ```
   add_vect (vect v, vect w)
     {int i; for (i=0; i<3; i++) v[i]+=w[i];}
   ```

2. On décide maintenant de changer la représentation des vecteurs en utilisant une structure à trois composantes `x`, `y` et `z`.
   Définir le nouveau type `vect` et proposer un nouveau codage de la fonction `add_vect` utilisant cette nouvelle représentation.
   Correction :

   ```
   typedef struct {int x; int y; int z;} vect2;
   add_vect2 (vect2 * v, vect2 w)
    { v->x += w.x;  v->y +=w.y; v->z +=w.z; }
   ```

3. On introduit maintenant un tableau t de vecteurs de taille `N`.
   Définir une fonction `add_tab_vect` qui prend en argument un vecteur w et ajoute le vecteur w aux eléments de t. Donner les deux codages de cette fonction suivant la représentation choisie pour les vecteurs.
   Correction :

   ```
   vect t[N];
   add_tab_vect (vect w)
   {int i; for (i=0; i<N; i++) add_vect(t[i],w);}
   add_tab_vect2 (vect2 w)
   {int i; for (i=0; i<N; i++) add_vect2(&(t2[i]),w);}
   ```

4 Fonctions récursives (3 points)

On souhaite définir une fonction d'élévation à la puissance d'un réel x par un entier positif ou nul n (`exp`(x, n)) en n'effectuant que des multiplications.

1. Définir une fonction `exp` qui effectue n multiplications successives dans une boucle.
   Correction :

```
float exp1 (float x, int n) {
  int i;
  float res=1.;
  for (i=1;i<=n;i++) { res *= x; }
  return res;
}
```

2. Afin de diminuer le nombre de multiplications, on utilise la propriété suivante :

$$x^n = x^{\frac{n}{2}} \times x^{\frac{n}{2}} \times x^{n \bmod 2}$$

Donner un codage récursif de la fonction `exp` utilisant cette égalité.
Correction :

```
float exp2 (float x, int n) {
  float res;
  if (n==0) return 1;
  else {res = exp2(x,n/2); res *= res;}
  if (n%2 == 1) res *= x;
  return res;}
```

3. Détailler le calcul de `exp`$(x, 10)$ suivant la méthode récursive, combien de multiplications sont nécessaires ?
Correction : *`exp`$(x, 10)$ fait un appel à `exp`$(x, 5)$ + 1 multiplication*
`exp`$(x, 5)$ fait un appel à `exp`$(x, 2)$ + 2 multiplications
`exp`$(x, 2)$ fait un appel à `exp`$(x, 1)$ + 1 multiplication
`exp`$(x, 1)$ fait un appel à `exp`$(x, 0)$ + 2 multiplications
On a en tout 6 multiplications (4 en traitant directement le cas $n = 1$)

5 Modélisation (10 points)

On considère une version modifiée du problème de planning vu en cours. Une réservation est composée d'un numéro unique, d'une salle et d'un horaire, une salle est composée d'un numéro unique, d'une capacité et d'une liste chaînée de réservations comportant toutes les réservations qui lui sont associées. Les questions sont largement indépendantes.

Horaires On suppose défini un type `Horaire` pour les créneaux horaires avec trois opérations :

```
int cmphor (Horaire h1, Horaire h2);
int avant (Horaire h1, Horaire h2);
int apres (Horaire h1, Horaire h2);
```

`cmphor`$(h1, h2)$ renvoie un entier négatif (resp. positif, resp. nul) si l'horaire $h1$ débute avant (resp. après, resp. au même moment que) $h2$.
`avant`$(h1, h2)$ renvoie 1 si l'horaire $h1$ se termine avant le début de $h2$ et 0 sinon.
`apres`$(h1, h2)$ renvoie 1 si l'horaire $h1$ débute après la fin de $h2$ et 0 sinon.

Réservations On définit une structure pour les réservations (la structure des salles sera précisée ensuite).

```
struct salle;
struct reservation {int numresa; struct salle *salle; Horaire hor;};
```

1. On s'intéresse à la modélisation de listes chaînées formées de réservations triées par odre croissant de début d'horaires.
   (a) Définir une structure `resaseq` pour les nœuds d'une liste chainée de pointeurs sur des réservations. Introduire une fonction `cree_resa` qui étant donné un pointeur sur une réservation, renvoie une liste ne contenant que cette réservation.
   Correction :

```
typedef struct reservation * Reservation;
struct resaseq {Reservation val; struct resaseq * next;};
typedef struct resaseq * ResaSeq;
ResaSeq cree_resa (Reservation val) {
  ResaSeq tmp = (ResaSeq) malloc (sizeof(struct resaseq));
  tmp -> val = val; tmp -> next = NULL; return tmp;}
```

(b) Définir une fonction d'insertion `add_resa_seq` d'une nouvelle réservation dans une liste chainée de réservations dans l'ordre croissant des horaires.

Correction :

```
int infresa (Reservation val1, Reservation val2) {
  return (val1 -> horaire.debut <= val2 -> horaire.debut);}
add_resa (Reservation val, ResaSeq * lp) {
  ResaSeq tmp,tmp1,tmp2;
  tmp  = cree_resa (val);
  if (* lp==NULL || infresa(val,(* lp)->val))
    { tmp -> next = * lp; * lp = tmp;}
  else {
    tmp1 = *lp; tmp2 = (*lp) -> next;
    while (tmp2 != NULL && infresa(tmp2 -> val, val))
      { tmp1 = tmp2; tmp2 = tmp2 -> next; }
    tmp -> next = tmp2; tmp1 -> next = tmp; }}
```

(c) Modifier la fonction précédente pour vérifier que l'horaire de la réservation que l'on cherche à introduire ne chevauche pas l'horaire des autres réservations de la séquence. La valeur de retour de la fonction devra indiquer si la réservation a pu être ajoutée ou non.

Correction :

```
#define DONE 0          #define ERROR 1

int avant_resa (Reservation val1, Reservation val2) {
  return avant(val1->horaire,val2->horaire);}

int ok_resa (Reservation val, ResaSeq l) {
  return (l==NULL || avant_resa(val,l->val));}

int ajout_resa (Reservation val, ResaSeq * lp) {
  ResaSeq tmp,tmp1,tmp2;
  tmp  = cree_resa (val);
  if (* lp==NULL || infresa(val,(* lp)->val))
    { if (ok_resa(val,*lp))
        {tmp->next = *lp; *lp = tmp; return DONE;}
      else return ERROR;}
  else { tmp1 = *lp; tmp2 = tmp1->next;
    while (tmp2 != NULL && infresa(tmp2->val, val))
      { if (avant_resa(tmp1->val, val))
        { tmp1 = tmp2; tmp2 = tmp2->next;}
      else return ERROR; }
    if (avant_resa(tmp1->val, val))
      {tmp->next = tmp2; tmp1->next = tmp; return DONE;}
    else return ERROR;  }}
```

2. Définir la structure des salles qui comporteront un numéro, une capacité et un pointeur vers une liste de réservations. Introduire un type `Salle` pour cette structure.

Correction :

```
struct salle {int numsalle; int capacite; ResaSeq resa;};
typedef struct salle Salle;
```

3. On suppose que l'on a (comme dans le cours) introduit un tableau comportant l'ensemble des salles triées par numéro de salle et un autre tableau comportant l'ensemble des réservations triées par numéro.
On dispose de fonctions de base pour ajouter une réservation dans la table des réservations et pour chercher une salle dans la table des salles. Les prototypes de ces fonctions sont :

```
typedef enum {OK,PLEIN,EXISTE} Erreur
Erreur ajout_resa_table(int num,Salle *salle,Horaire hor);
Salle * cherche_salle(int num);
```

`cherche_salle` renvoie `NULL` s'il n'y a pas de salle avec ce numéro dans la table et renvoie un pointeur sur la salle sinon.
`ajout_resa_table` renvoie `OK` si la réservation a pu être ajoutée, elle renvoie `PLEIN` si la table était pleine et `EXISTE` s'il existait déjà une réservation de même numéro.

(a) Définir une fonction `ajout_resa_systeme` de même prototype que `ajout_resa_table` mais qui effectue à la fois l'ajout de la réservation dans la table des réservations et dans la liste chaînée des réservations associée à la salle. Cette fonction doit vérifier qu'il n'y a pas de chevauchement d'horaire. Modifier le type `Erreur` afin de prendre en compte tous les cas possibles d'erreur.

Correction :

```
typedef enum {OK,PLEIN,EXISTE,CHEVAUCHE} Erreur;
Reservation cherche_resa(int n);
void retire_resa(int n);
Erreur ajout_resa_systeme(int n, Salle * s, Horaire h) {
  Erreur ajout;
  Reservation val;
  ResaSeq rs = s -> resa;
  ajout = ajout_resa_table(n,s,h);
   if (ajout==OK)
     { val = cherche_resa(n);
     if (ajout_resa(val,&rs)==DONE) {s->resa=rs; return OK;}
     else {retire_resa_table(n); return CHEVAUCHE;}}
   else return ajout;}
```

(b) Définir une fonction `ajoute_resa` qui prend en argument un numéro de salle et un horaire et qui crée un nouveau numéro de réservation et effectue la réservation correspondante.

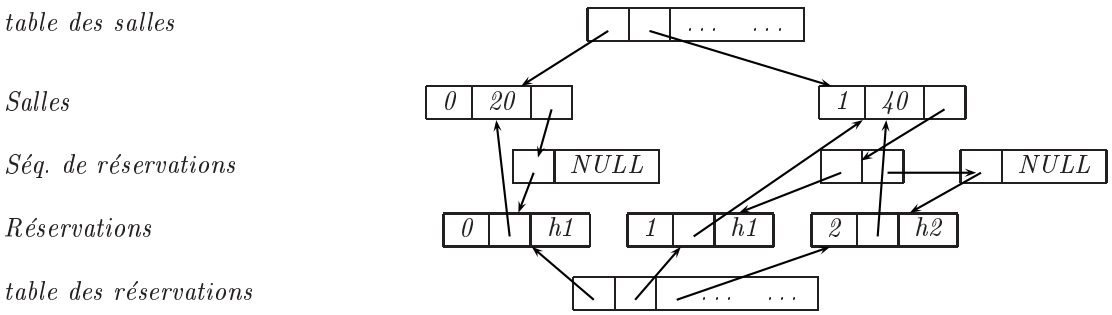
Correction :

```
Erreur ajoute_resa (int nums, Horaire h) {
  static int numresa=-1;
  Salle * s;
  s=cherche_salle(nums);
  numresa++;
  return ajout_resa_systeme(numresa,s,h);}
```

(c) On suppose qu'initialement il n'y a aucune réservation de faite et qu'il y a deux salles référencées dans la table des salles : la salle $n^o 0$ de capacité 20 et la salle $n^o 1$ de capacité 40. On suppose que h_1 et h_2 sont des horaires tels que h_2 est avant h_1. Décrire l'état du système après l'évaluation du programme suivant :

```
ajoute_resa(0,h1); ajoute_resa(1,h1); ajoute_resa(1,h2);
```

Correction :



(d) Quel est l'intérêt sur le plan algorithmique d'avoir associé à la salle la liste chaînée des réservations ? Le choix de trier cette liste par ordre croissant d'horaires vous parait-il pertinent ? quel autre choix aurait-on pu faire ?

Correction : *L'association de la séquence de réservations à une salle permet de vérifier les conflits horaires en ne parcourant que les réservations associées à la salle. C'est un choix intéressant s'il y a un grand nombre de salles.*
L'insertion d'une nouvelle réservation a un coup proportionnel au nombre de réservations déjà faites pour des dates antérieures. Il sera important dans ce cas de figure ne pas garder trace dans la séquence des réservations passées. On pourrait aussi choisir l'ordre inverse.

4. En reprenant les hypothèses de travail de la question précédente, définir une fonction `planning` qui étant donnés un numéro de salle et deux horaires h_1 et h_2 renvoie la liste chainée des réservations de cette salle comprises entre les deux horaires (ie qui débutent après le début de h_1 mais avant le début de h_2).

Correction :

```
/* fin est un pointeur qui est un suivant de debut,
   renvoie une copie de la sequence debut, écoupe à partir de fin */
ResaSeq copy (ResaSeq debut, ResaSeq fin) {
  ResaSeq tmp;
  if (debut == fin) return NULL;
  tmp = cree_resa (debut -> val);
  tmp -> next = copy(debut -> next,fin);
  return tmp;}
ResaSeq planning(int nums, Horaire h1, Horaire h2)
{ Salle * s;
  ResaSeq tmp1, tmp2;
  s=cherche_salle(nums);
  tmp1 = s->resa;
  while (tmp1 != NULL && cmphor(tmp1->val->horaire,h1)<0)
    {tmp1 = tmp1 -> next;}
  tmp2=tmp1;
  while (tmp2 != NULL && cmphor(tmp2->val->horaire,h2)<=0)
    {tmp2 = tmp2 -> next;}
  if (tmp2 == NULL) return tmp1;
  else return copy(tmp1,tmp2);}
```