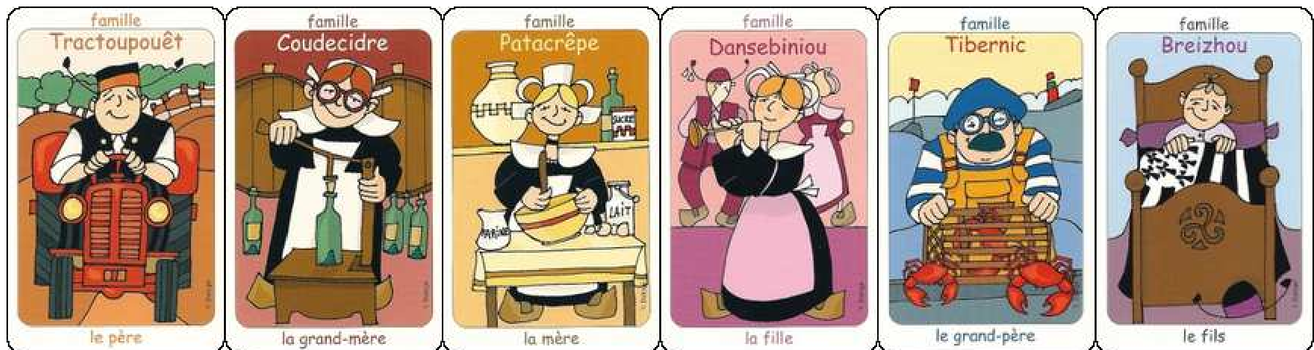
Aix-Marseille Université
Faculté des Sciences de Luminy

Licence Math-Info – Semestre 2
Mercredi 25 Juin 2014

Programmation — Examen

Durée : 2 heures, Calculatrices et Documents Interdits



L'examen est identique à celui de la session de Mai ;o)

On souhaite modéliser des éléments du Jeu des Sept Familles. Dans ce jeu de cartes, il y a généralement 42 cartes réparties en 7 familles, avec 6 membres par famille (le grand-père, la grand-mère, le père, la mère, le fils et la fille), mais on pourrait faire varier ces nombres. Dans les règles classiques, chaque joueur reçoit une main de 6 cartes au début et cherche à obtenir des familles complètes de 6 membres en demandant tour à tour une carte précise à un adversaire. Lorsque l'adversaire possède cette carte, il la donne au demandeur et celui-ci peut continuer à demander des cartes à ses adversaires. Sinon, il doit piocher et son tour de parole se termine.

Il n'y a cependant pas besoin de connaître les règles pour faire ce devoir, car on s'intéresse seulement à la modélisation objets (familles, cartes, main, paquet) et au transfert de cartes.

Dans tout le devoir, pour implémenter une fonction, vous <u>devez RÉUTILISER</u> les fonctions des questions précédentes lorsque cela est possible, même si vous ne les avez pas implémentées.

La section 3 est indépendante de la section 2.

1 Les membres de la famille : `enum Member`

On considère l'énumération suivante :

```
typedef enum Member {
  MEMBER_INVALID= -1,
  MEMBER_GRANDPA , MEMBER_GRANDMA , MEMBER_FATHER , MEMBER_MOTHER , MEMBER_SON , MEMBER_DAUGHTER ,
  NB_MEMBERS
} Member;
```

Question 1. Que valent `MEMBER_GRANDPA`, `MEMBER_GRANDMA`, `MEMBER_DAUGHTER` et `NB_MEMBERS` ?

Question 2. Implémenter la fonction `Member_ToString()` qui retourne la chaîne décrivant un membre de famille `m` en français, soit `le grand-pere`, `la grand-mere`, `le pere`, `la mere`, `le fils`, ou `la fille`.

```
char const * Member_ToString (Member m);
```

Voici un exemple d'utilisation de la fonction et la sortie produite :

```
int main (void)
{
  char name []= "Patacrepe";
  Member member= MEMBER_MOTHER;
  fprintf (stdout , "Dans␣la␣famille␣%s,␣je␣demande␣%s.\n",
           name , Member_ToString (member));
  return 0;
}
```

```
Dans la famille Patacrepe, je demande la mere.
```

2 Famille : struct Family

Le type structuré Family représente une famille sous-ensemble de la famille complète à NB_MEMBERS. Si f est une variable de ce type, alors le champ f.count indique son nombre de membres, et le champ f.members est un tableau de NB_MEMBERS booléens tel que f.members[m]=1 si et seulement si le membre m est présent dans la famille. Par exemple, si f veut signifier qu'un joueur possède la mère et le fils de la famille Cidredou, alors f.members[MEMBER_MOTHER]=1, f.members[MEMBER_SON]=1, les autres indicateurs de présence valent 0, et f.count=2.

```
typedef struct Family {
  int count;
  int members [NB_MEMBERS];
} Family;
```

Question 3. Implémenter la fonction Family_InitEmpty() qui initialise une famille (pointée par) f comme une famille vide, c'est à dire sans membre.

```
void Family_InitEmpty (Family * f);
```

Question 4. Implémenter le prédicat Family_ContainsMember() qui indique si une famille (pointée par) f contient le membre m. De plus, la fonction contrôle avec assert() que m est dans la plage de valeurs valides.

```
int Family_ContainsMember (Family const * f, Member m);
```

Question 5. Implémenter la fonction Family_AddMember() qui rajoute le membre m à une famille (pointée par) f. De plus, la fonction contrôle avec assert() que m n'y était pas déjà.

```
void Family_AddMember (Family * f, Member m);
```

Question 6. Implémenter la fonction Family_RemoveMember() qui retire le membre m d'une famille (pointée par) f. De plus, la fonction contrôle avec assert() que m y était bien déjà.

```
void Family_RemoveMember (Family * f, Member m);
```

Question 7. Un joueur ne peut pas demander à un adversaire le membre d'une famille f s'il ne possède pas déjà un membre de f dans sa main. De plus, il peut étaler une famille dès qu'elle devient complète. Implémenter le prédicat Family_IsEmpty() qui teste si une famille (pointée par) f est vide, ainsi que le prédicat Family_IsComplete() qui teste si f est complète.

```
int Family_IsEmpty    (Family const * f);
int Family_IsComplete (Family const * f);
```

Question 8. On considère le programme suivant qui utilise une fonction d'affichage Family_Print() :

```
int main (void)
{
  Family f;
  char name []= "Coudecidre";

  Family_InitEmpty (& f);
  Family_Print (& f, name, stdout);

  Family_AddMember (& f, MEMBER_SON);
  Family_AddMember (& f, MEMBER_MOTHER);
  Family_AddMember (& f, MEMBER_FATHER);
  Family_Print (& f, name, stdout);

  Family_RemoveMember (& f, MEMBER_SON);
  Family_RemoveMember (& f, MEMBER_FATHER);
  Family_Print (& f, name, stdout);
  return 0;
}
```

Voici les affichages produits par les trois appels à Family_Print() sur la sortie standard stdout :

```
0 membre  de la famille Coudecidre: [ ]
3 membres de la famille Coudecidre: [ <le pere> <la mere> <le fils> ]
1 membre  de la famille Coudecidre: [ <la mere> ]
```

Implémenter la fonction `Family_Print()` qui affiche sur la sortie `file` la liste des membres d'une famille nommée par `name` et (pointée par) `f`. Le format d'affichage doit être identique à l'exemple.

```
void Family_Print (Family const * f, char const name [], FILE * file);
```

Remarque : N'essayez de gérer la distinction singulier/pluriel sur `membre/membres` que si vous savez utiliser l'opérateur ternaire `test?expr1:expr2`. Notez que l'ordre d'affichage dans la liste ne dépend pas de l'ordre des insertions dans la famille, mais de l'ordre des déclarations de constantes dans `enum Member`.

3 Paquet de Cartes : `struct Card` et `struct Deck`

S'il y a `NB_FAMILIES` familles, elles sont numérotées de `0` à `NB_FAMILIES-1`, et on appelle `fid` (*family identifier*) leur numéros. Une carte est un membre donné d'une famille de numéro `fid`, et est modélisé par la structure `Card` ci-dessous. Par ailleurs, on donne la fonction `Card_Make()` qui permet de fabriquer une carte.

```
#define NB_FAMILIES 7

typedef struct Card {
  Member member;
  int fid;
} Card;

Card Card_Make (int fid, Member member)
{
  return (Card) { .fid= fid, .member= member };
}
```

Le paquet de cartes de la pioche est représenté par la structure `Deck`. Si `d` est un paquet, alors il peut contenir jusqu'à `DECK_CAPACITY` cartes dans un tableau `d.cards`, et `d.length` est le nombre cartes qu'il contient réellement aux indices `0 ... d.length-1`. Enfin, La fonction `Deck_InitEmpty()` initialise un paquet à vide.

```
#define DECK_CAPACITY (NB_FAMILIES * NB_MEMBERS)

typedef struct Deck {
  int length;
  Card cards [DECK_CAPACITY];
} Deck;

void Deck_InitEmpty (Deck * d) { d->length= 0; }
```

Question 9. Implémenter la fonction `Deck_AddCardAtEnd()` qui ajoute une carte `card` à la fin du paquet (pointé par) `d`. La fonction vérifie préalablement avec `assert()` que le paquet n'était pas déjà plein.

```
void Deck_AddCardAtEnd (Deck * d, Card card);
```

Question 10. Implémenter la fonction `Deck_RemoveCardAtEnd()` qui retire et retourne la dernière carte du paquet (pointé par) `d`. La fonction vérifie préalablement avec `assert()` que le paquet n'était pas déjà vide.

```
Card Deck_RemoveCardAtEnd (Deck * d);
```

Question 11. Implémenter la fonction `Deck_InitFullSorted()` qui initialise un paquet (pointé par) `d` afin que toutes les familles y apparaissent complètes, triées par membres et `fid` croissants. La fonction doit initialiser le paquet à vide et ajouter les cartes une à une.

```
void Deck_InitFullSorted (Deck * d);
```

Question 12. Implémenter la fonction `Deck_Shuffle()` qui mélange les cartes d'un paquet (pointé par) `d`. Pour ce faire, la fonction balaye les indices en ordre croissant et permute la carte courante avec une carte choisie au hasard entre elle (inclue) et la fin du paquet (la carte courante peut parfois être permutée avec elle-même).

```
void Deck_Shuffle (Deck * d);
```

4 Main de joueur : struct Hand

On modélise la main d'un joueur par la structure `Hand` qui encapsule un tableau de familles. Si `h` est une main et `fid` un numéro de famille, alors on trouve cette famille dans `h.families[fid]`, qui permet de connaître les membres que `h` possède pour cette famille.

```
typedef struct Hand {
  Family families [NB_FAMILIES];
} Hand;
```

Question 13. Implémenter la fonction `Hand_InitEmpty()` qui initialise une main (pointée par) `h` à vide (la main n'a aucune carte et donc toutes ses familles sont vides).

```
void Hand_InitEmpty (Hand * h);
```

Question 14. Implémenter `Hand_AddCard()` et `Hand_RemoveCard()` qui, respectivement, ajoute et retire une carte `card` à une main (pointé par) `h`.

```
void Hand_AddCard    (Hand * h, Card card);
void Hand_RemoveCard (Hand * h, Card card);
```

Question 15. Pour distribuer les cartes ou pour piocher, il faut pouvoir transférer la dernière carte de la pioche vers une main. Implémenter la fonction `Hand_DrawFromDeck()` qui transfère la dernière carte du paquet (pointé par) `d` vers la main (pointée par) `h`.

```
void Hand_DrawFromDeck (Hand * h, Deck * d);
```