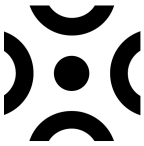
Perfectionnement à la programmation en C

Fiche de TP 4

L2 Informatique - 2024–2025

Chomp

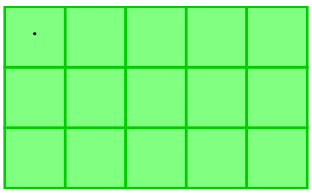


L'objectif de ce TP est d'implanter le *Chomp*, un jeu combinatoire abstrait à deux joueurs. Nous utilisons ici les notions de

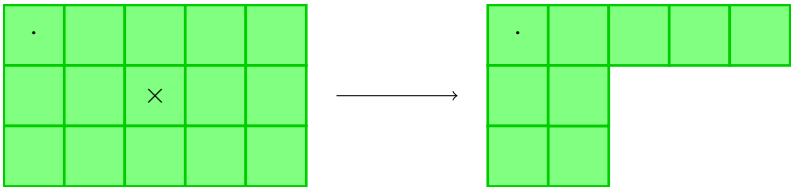
- tableau statique à deux dimensions ;
- type structuré ;
- type énuméré ;
- interface `ncurses` en gestion au tour par tour ;
- comparaison entre programmation par fichier unique et par modules ;
- compilation par un script bash et makefile

1 Spécification

Les règles du *Chomp* sont les suivantes. Deux joueurs se disputent une tablette de chocolat de dimension $n \times m$ où n et m sont des entiers supérieurs à un (voir figure suivante).



Chaque joueur choisit alternativement un carré de chocolat, le mange, et mange aussi tous les carrés qui se trouvent en bas et à sa droite (voir la figure suivante). La partie s'arrête lorsque l'un des deux joueurs mange le carré de chocolat en position $(0, 0)$. Ce carré est en effet empoisonné et ce joueur perd la partie.



L'un des objectifs de ce TP est de parfaire l'utilisation de la bibliothèque graphique **`ncurses`** qui permet d'afficher la tablette de chocolat et de faire en sorte que deux joueurs puissent jouer au *Chomp* à la souris. On suit dans la conception de ce projet une approche ascendante (dite *bottom-up*). Celle-ci consiste à réaliser le projet en constituant une à une ses pièces pour les assembler finalement.

Remarque importante : *le programme décrit dans les exercices 1, 2 et 3 est à écrire dans un unique fichier nommé* `Chomp.c`. *Il sera demandé de le modulariser dans l'exercice 4. Cette approche, consistant à concevoir un projet dans un unique fichier source puis à le modulariser ensuite, est* ***incorrecte*** *en pratique. On la considère ici pour apprécier les désavantages de la programmation dans un fichier unique au détriment de la programmation modulaire.*

2 Écriture du programme

Exercice 1 (Déclarations des types) Nous commençons par déclarer les types qui nous serviront à représenter les objets principaux apparaissant dans le *Chomp*.

1. Déclarer un type `Tablette` qui représente une tablette de chocolat de dimension $n \times m$. Utiliser un **tableau statique** de dimension $n \times m$ qui contient des entiers. Un contenu à `1` signifie que le carré de chocolat correspondant existe encore tandis qu'un contenu à `0` signifie qu'il a été mangé. Les dimensions n et m sont représentées par des alias `NB_LIGNES` et `NB_COLONNES` définis par des macro-instructions. Par défaut, `NB_LIGNES` sera un alias de `8` et `NB_COLONNES` un alias de `16`.

   Observer que nous allons écrire des fonctions qui renvoient des valeurs de type `Tablette` (voir par exemple la question 2 de l'exercice 2). Comme il est impossible de renvoyer un tableau statique, il faut recourir à l'astuce suivante. Sachant qu'il est possible de renvoyer des valeurs d'un type structuré, le type `Tablette` sera déclaré comme un type structuré qui contient un champ tableau statique. Le fait d'encapsuler un tableau statique ainsi garantit le bon fonctionnement (ce point sera détaillé dans l'un des futurs cours).

2. Déclarer un type énuméré `Joueur` qui permet de modéliser les deux joueurs. Cette notion de type énuméré sera approfondie par la suite. Pour le moment, il suffit de comprendre que les types énumérés servent à représenter des valeurs qui appartiennent à un ensemble fini. Ces valeurs sont codées par le système par des entiers positifs (`0` pour la 1^{re}, `1` pour la 2^{e}, *etc.*). Ici donc, le type énuméré `Joueur` doit contenir exactement deux valeurs : `JOUEUR_1` et `JOUEUR_2`.

   Voici un exemple de déclaration d'un type énuméré `Couleur` suivie d'une utilisation possible :

```
typedef enum {
    TREFLE,
    PIQUE,
    CARREAU,
    COEUR
} Couleur;
...
...
Couleur c;
c = PIQUE;
...
if (c == TREFLE) {
    ...
}
```

3. Déclarer un type `EtatJeu` qui permet de représenter l'état du jeu de *Chomp*. L'état du jeu est déterminé par une tablette de chocolat et le joueur dont c'est le tour de jouer. Il s'agit donc de toutes les informations nécessaires pour représenter une situation de jeu.

4. Déclarer un type `Coup` qui permet de modéliser un coup joué. Un coup est entièrement spécifié par les coordonnées `ligne` et `colonne` du carré de chocolat que le joueur souhaite manger.

Exercice 2 (Manipulation des objets)

Nous mettons au point ici les fonctions principales de manipulation des objets du jeu. Tout ceci se fait de manière totalement indépendante de l'aspect graphique qui sera ajouté plus tard.

Rappelons qu'une fonction manipulant une variable d'un type structuré ne peut la modifier que si cette dernière est passée par adresse. Ainsi, les fonctions suivantes sont paramétrées par des variables d'un type structuré ou par des pointeurs sur des variables d'un type structuré selon le comportement souhaité. Ces conventions adoptées ici sont raisonnables pour débuter mais nous allons en donner des plus précises plus tard dans le cours.

1. Écrire une fonction `Joueur adversaire(Joueur joueur)` qui renvoie l'adversaire du joueur `joueur`.

2. Écrire une fonction `Tablette creer_tablette(void)` qui crée et renvoie une variable de type `Tablette` de dimension `NB_LIGNES` × `NB_COLONNES`. La tablette renvoyée possède tous ses carrés de chocolat.

3. Écrire une fonction `void manger(Tablette *tab, int ligne, int colonne)` qui modifie la tablette `tab` de sorte à manger le carré de chocolat en position `(ligne, colonne)` ainsi que tous ceux qui sont en dessous de lui et à sa droite.

4. Un coup dans un état du jeu donné est légal s'il ordonne de manger un carré de chocolat qui existe encore. Écrire une fonction `int est_legal(EtatJeu etat, Coup coup)` qui renvoie `1` si le coup `coup` est légal dans l'état du jeu `etat` et `0` sinon.

5. La partie est terminée lorsque le carré de chocolat empoisonné vient d'être mangé ans ce cas, c'est le joueur qui vient de jouer qui a perdu et l'autre qui a gagné. Écrire une fonction `int est_jeu_termine(EtatJeu etat, Joueur *joueur_gagnant)` qui renvoie `1` si la partie représentée par l'état du jeu `etat` est terminée et `0` sinon. Dans le cas où la partie est terminée, la fonction affecte à la variable pointée par `joueur_gagnant` le joueur qui gagne la partie.

6. Écrire une fonction `void jouer_coup(EtatJeu *etat, Coup coup)` qui joue le coup `coup` dans l'état du jeu `etat`. Il ne faut pas oublier de modifier le champ qui contient le joueur dont c'est le tour de jouer.

Exercice 3 (Assemblage du jeu) Nous utilisons les types et fonctions de base mis au point précédemment pour mettre en place la boucle principale du jeu et une interface **`ncurses`** permettant d'y jouer.

1. Écrire une fonction `void afficher_jeu(EtatJeu etat)` qui affiche sur la fenêtre standard l'état du jeu `etat`. Les conventions pour l'affichage sont libres mais un affichage précis, lisible et minimaliste sera apprécié.

2. Écrire une fonction `Coup lire_coup(EtatJeu etat)` qui attend un clic de l'utilisateur sur la tablette de chocolat dans la fenêtre graphique et calcule et renvoie le coup spécifié par l'utilisateur. Si l'utilisateur ne clique pas sur la tablette, ou bien clique sur un carré de chocolat déjà mangé, la fonction ne fait rien et attend de traiter le prochain clic.

3. Utiliser les fonctions précédentes pour construire le programme `Chomp.c` qui permet de jouer au *Chomp*. Utiliser pour cela l'algorithme suivant :

    1. initialiser l'état du jeu `etat` ;

2. tant que l'état du jeu `etat` ne représente pas une partie terminée :
   1. afficher l'état du jeu `etat` sur la fenêtre graphique ;
   2. lire un coup `c` sur la fenêtre graphique ;
   3. jouer le coup `c` dans l'état du jeu `etat` ;
3. afficher le numéro du joueur gagnant.

Exercice 4 (Modularisation)

1. Imaginer un bon découpage en modules du programme réalisé. Justifier le choix de découpage adopté. Nous conseillons ici de procéder au découpage qui consiste à créer un module pour chaque type et d'y répartir raisonnablement les fonctions précédemment écrites. Cette approche est en général un très bon point de départ pour modulariser un projet de manière satisfaisante.

2. Reprendre le fichier `Chomp.c` et le découper en plusieurs modules conformément à la question précédente. Il n'est en rien problématique d'avoir des modules déséquilibrés en terme de taille (certains peuvent même être quasi vides).

   Note : Le fichier `Chomp.c` peut être conservé à titre de référence pour le découpage.

3. Donner la liste de commandes nécessaires à la compiliation du projet. Les placer dans un script bash nommé `Compilation.sh`.

   Note : les fichiers `Makefile` offrent un moyen plus évolué de répondre à cette question. Il est cependant très important de savoir comment compiler manuellement un projet modularisé.

4. Comparer les deux versions du projet, la première en un unique fichier et la seconde modularisée. Citer les avantages et les inconvénients des deux méthodes.

   Note : dans les futurs projets, on adoptera uniquement l'approche de programmation par modules.

5. Écrire un fichier `Makefile` permettant de compiler la version modularisée du projet.

---

3 Améliorations

Exercice 5 (Améliorations et questions)

Cet exercice n'est a envisager que lorsque tout le reste fonctionne parfaitement. Les modifications et ajouts proposés sont à apporter à la version modularisée du projet.

1. *(Facile.)* Proposer une option au programme pour gérer des *matchs*. Un match est une succession de parties dans laquelle le nombre de victoires de chaque joueur est pris en compte. Le déroulement est le suivant : chacun des deux joueurs entre un nombre souhaité de parties ; ensuite, un nombre de parties égal à la moyenne (arrondie à l'entier supérieur) entre ces deux entiers sont jouées. À l'issue de ces parties, le résultat du match est affiché (victoire pour l'un ou l'autre des joueurs ou bien match nul).

2. *(Facile.)* On souhaite concevoir un système de sauvegarde de partie. Une partie est encodée par la taille du plateau et la suite des coups joués depuis le début de la partie. Proposer un format de fichier pour coder une partie de cette façon. Ajouter au programme la possibilité de sauvegarder une partie (et donc écrire son encodage dans un fichier) et la possibilité de charger une partie (et donc de lire son encodage depuis un fichier).

3. *(Moyen.)* On peut remarquer que, si l'on souhaite changer la taille de la tablette, il est nécessaire de modifier les alias `NB_LIGNES` et `NB_COLONNES` et de recompiler le programme. Pour faire en sorte d'avoir une tablette dont la taille peut être choisie une fois le programme exécuté (et donc sans avoir à le recompiler), il est possible d'utiliser des tableaux dynamiques. Modifier en ce sens le type `Tablette` de sorte à représenter le contenu des carrés par un tableau dynamique. Modifier le reste du programme afin qu'il supporte les dimensions de la tablette comme arguments.

4. *(Difficile.)* Proposer un mode de jeu contre l'ordinateur. Suggérer plusieurs niveaux de difficulté : dans le plus facile, l'ordinateur joue ses coup au hasard ; dans l'autre, imaginer un moyen de faire jouer l'ordinateur avec une certaine force.

5. *(Mathématique.)* Démontrer que le joueur qui commence la partie gagne contre toute défense de son adversaire (on dit que le premier joueur dispose d'une stratégie gagnante).
