Portée des variables

1102 – Portée des variables et structures

```
 **/

int required_nb_monsters(cgame game, content monster);



/**
 * @brief Test if the game is over (that is the grid is filled according to the require
ments).
 * @return true if all the constraints are satisfied
 **/

bool is_game_over (cgame g);



/**
 * @brief Restart a game by cleaning monsters from the board.
 **/

void restart_game(game g);



/**
 * @brief adds a monster on the game board.
 * Can also be used to remove any monster by adding EMPTY.
 * This function does not have effect on mirrors so it can be used safely in the course
 of the game.
 * @param game the game board where to add the monster
 * @param monster the type of monster to add
 * @param col the column where to insert the monster
 * @param line and the line where to insert the monster
 * @return false if the monster was not placed since the square was occupied by a mirro
r.
 **/

bool add_monster(game game, content monster, int col, int line);

/**
 * @brief says how many monsters can be seen on the current game board
 * @param game the game board to look at
 * @param side the side of the board we consider (N, S, E, or W)
 * @param pos the coordinate on that side (from S to N or from W to E)
 * @return the number of monsters that can be seen through all the mirrors from a given
 side at position x
 **/

int current_nb_seen(cgame game, direction side, int pos);
```

```
/**
 * @brief check whether all squares are occupied
 * @param game the game board
 **/

bool is_filled(cgame game)
{
  return (game->cur_nb_item[EMPTY] == 0);
}


bool is_game_over(cgame g){
  bool res = is_filled(g);
  for(int i=0; i < g->width; i++){
     res = res && (g->labels[N][i] == current_nb_seen(g,N,i));
     res = res && (g->labels[S][i] == current_nb_seen(g,S,i));
  }
  for(int i=0; i<g->height; i++){
     res = res && (g->labels[E][i] == current_nb_seen(g,E,i));
     res = res && (g->labels[W][i] == current_nb_seen(g,W,i));
  }
  res = res && (g->nb_ghosts == g->cur_nb_item[GHOST]);
  res = res && (g->nb_vampires == g->cur_nb_item[VAMPIRE]);
  res = res && (g->nb_zombies == g->cur_nb_item[ZOMBIE]);
  res = res && (g->nb_spirits ==g->cur_nb_item[SPIRIT]);

  return res;
}

void restart_game(game g){
  restart_board(g->width, g->height, g->board);
  g->cur_nb_item[EMPTY] += g->cur_nb_item[GHOST];
  g->cur_nb_item[GHOST] = 0;
  g->cur_nb_item[EMPTY] += g->cur_nb_item[VAMPIRE];
  g->cur_nb_item[VAMPIRE] = 0;
  g->cur_nb_item[EMPTY] += g->cur_nb_item[ZOMBIE];
  g->cur_nb_item[ZOMBIE] = 0;
  g->cur_nb_item[EMPTY] += g->cur_nb_item[SPIRIT];
  g->cur_nb_item[SPIRIT] = 0;
  update_current_nb_seen(g);
}

bool add_monster(game g, content monster, int col, int line){
  if(
     (line>=0 && line<g->height && col>=0 && col<g->width)
     &&
     ((monster>=VAMPIRE && monster<= SPIRIT) || monster == EMPTY)
     )
  {
    if(g->board[line][col]>=MIRROR && g->board[line][col]<=HMIRROR){
```

Organisation d'un programme

```
void affiche_HMS(int secondes){
  int h, m, s;
  h = secondes / 3600;
  m = (secondes % 3600) / 60;
  s = secondes % 60;
  printf("%d:%2d:%2d\n",h,m,s);
}
```

- déclaration de la fonction
- déclaration des variables
- séquence d'instructions
- fermeture de la fonction

```
int main(){
  int sec;
  scanf("%d",&sec)
  affiche(sec)
  return 0;
}
```

- déclaration de la fonction
- déclaration des variables
- séquence d'instructions
- fermeture de la fonction

Portée des variables

Durée de vie de la variable

Une variable existe :

- à partir de sa déclaration :
  - soit dans une instruction (déclaration explicite)
  - soit comme paramètre d'une fonction (déclaration implicite)
- jusqu'à la fermeture du bloc contenant la déclaration.

  $\longrightarrow$ fermeture d'accolade correspondante

Variables masquées

Une variable n'est plus visible :

- dans les fonctions auxiliaires appelées
- si on *déclare* une autre variable du même nom

Exemple

```
{
        int x = 32;
        double y = 1.2;
        double z;
        {
                double z = y + 3;
                int x = 52;
                x ++;
                y = 3 * y;
        }
}
```

Derrière, il y a la machine

- À l'ouverture du bloc, la machine réserve la mémoire pour mettre toutes ses variables
- à la fermeture, elle libère la mémoire, et oublie les variables
- chaque appel de fonction vit dans un espace différent

Structures

Les structures en C

`struct nom { contenu };`

- Les structures permettent de définir des types construits.
- Le contenu d'une structure est simplement un ensemble de déclarations de variables.
- Les définitions de structures sont souvent globales (parfois même dans le ".h")

Exemple

Une structure horaire composée de 3 entiers :

```
struct horaire {
  int heure;
  int minute;
  int seconde;
};
```

Initialisation de structure

On peut initialiser une structure de deux façons :

- directement, comme on initialise un entier :

```
struct horaire courant = {12, 45, 32};
```

- champ par champ, avec un `.` pour désigner le champ.

```
struct horaire courant;
courant.heure = 12;
courant.minute = 45;
courant.seconde = 37;
```

- Cet accès `nom.membre` permet d'accèder aux membres d'une structure comme une variable normale.

```
if(courant.minute >= 60){
  courant.heure ++;
  courant.minute = courant.minute - 60;
  }
```

Utilisation

Une fois définie, une structure est un type comme un autre :

- on peut définir des fonctions qui renvoient des `struct` :

```
struct horaire seconde_vers_heure( int secondes);
```

- et des fonctions qui prennent des `struct` en paramètre :

```
int horaire_vers_secondes( struct horaire heure);
struct horaire somme_horaires(struct horaire heure1,
                              struct horaire heure2);
```

Elles obéissent aux mêmes règles de portée que les variables.

Bilan sur les structures

- Les structures permettent de créer des types construits.
- Elles se comportent comme des types usuels (en deux mots)
- Elles ont les mêmes règles de portée que les variables.
- Affecter une structure en fait une copie.

Constantes et enumérations

Les constantes

`#define NOM <valeur>`

Une constante est un mot-clé associé à une valeur, immuable après compilation.

- par convention, tout en majuscule
- utile pour définir un paramètre global du programme (e.g. `RAND_MAX`)
- remplacé par le pré-processeur, avant la compilation.

Exemples

```
#define PI 3.141591654
#define RAND_MAX 2147483647
#define TAILLE_PLATEAU 3
```

Les énumérations

`#define NOM <valeur>`

Une constante est un mot-clé associé à une valeur, immuable après compilation.

- par convention, tout en majuscule
- utile pour définir un paramètre global du programme (e.g. `RAND_MAX`)
- remplacé par le pré-processeur, avant la compilation.

Exemples

```
#define PI 3.141591654
#define RAND_MAX 2147483647
#define TAILLE_PLATEAU 3
```