1102 – Éléments de syntaxe du C

```
 **/

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){
```

Les instructions

- L'élément constitutif d'un code C est l'**instruction**
  $\longrightarrow$ le C est un langage dit **impératif**
- Une **instruction simple** termine par un point virgule (';')
- Plusieurs instructions peuvent être réunies en une seule en les entourants d' accolades ('{' et '}').

Exemples

```
int x;
double moyenne = 3;
y = 3 * k + 2;
if(t > 0) resultat = valeur / 2;
return solution;
```

Les expressions

- sont des bouts de code qui ont une valeur
- peuvent avoir une valeur numérique ou autre

Examples

```
3 * (x + 1) + 2 * z
z > 0
ma_fonction(x,y,z)
4 * t % 7 == y + 3
'a'
```

Les variables

(En C, pas comme en maths)

- servent à associer un nom à une valeur
  $\longrightarrow$ ont toujours un nom, toujours une valeur
- peuvent être imaginée comme des cases mémoire (un tiroir dans lequel on stocke l'information)
- peuvent être modifiée dans le temps (changement du contenu du tiroir)

Les fonctions

(En C, toujours pas comme en maths)

- servent à associer une séquence d'instruction à un nom
- peuvent prendre des paramètres (des informations en entrée)
- peuvent avoir une valeur de retour
- peuvent être appelées autant de fois qu'on le souhaite
- ne font rien si on ne les appelle pas

⟶ Il existe une fonction très spéciale, appelée `main`, qui est la fonction qui sera lancée si on exécute le programme.

⟶ Les fonctions sans retours sont parfois appelées **procédures**

Et encore

mais pour plus tard

- des constantes
- des structures
- des pointeurs
- ...

Déclaration

Crée une variable avec son type

```
<type> nom_de_variable ;
<type> nom_1, nom_2, nom_3 ;
```

Les types :

- `int` : entier
- `double` : à virgule (double précision par rapport au `float`)
- `char` : caractère (ascii)
- ... et des types composés.

Le nom de variable :

- commence par une lettre
- contient des lettres, des chiffres, des '_'
- n'est pas un mot clé du langage (`if`, `for`, `case`...)

Affectation

Donne une valeur à la variable

nom_de_variable = <expression> ;

- Calcule la valeur de l'expression et l'assigne à la variable
- L'expression peut utiliser la valeur précédente de la variable (e.g. `x = x + 1`)
- peut faire un *cast* implicite (changement de type)

Les opérateurs des expressions

Binaires

Ces opérateurs se placent entre les deux éléments qu'ils lient.

- arithmétiques : `+`, `*`, `-`, `/`, `%` (reste de la div. euclidienne)
- logiques : `==` ($=$), `!=` ($\neq$), `<`, `>`, `<=`, `>=`, `&&` (et), `||` (ou)

Unaires

Ces opérateurs se placent avant l'élément qu'ils concernent.

- arithmétiques : `-`
- logiques : `!`

Les parenthèses

Un seul type de parenthèses, `( )`, s'emboitent à volonté.

Exemple : `(3 + 2 * ((- 4) * (32 % 7))) / 12`

Déclaration et initialisation

Déclare et donne une valeur initiale

```
<type> nom_de_variable = <expression> ;
```

- Combine les deux opérations précédentes.
- Si on n'initialise pas la valeur à la déclaration, on ne sait pas ce qu'elle vaudra.

Déclaration de fonction

Associe au nom de la fonction au instructions correspondantes

```
<type retour> nom_de_fonction(){
  <instructions>
}
<type retour> nom(<type1> param_1, <type2> param_2){
   <instructions>
}
```

- un type en plus, `void` (rien)
- le nommage de paramètre se comporte comme une définition de variable
- pas d'effet de bord possible (sauf pointeurs/tableaux)

Appel de fonction

```
nom_de_fonction(param1, param2);
<variable> = nom_de_fonction(param1, param2);
```

- n'importe quoi dans le paramètre, pourvu que le type soit bon.
- le nommage des variables est totalement indépendants

L'instruction `return`

`return <expression>;`

- permet de conclure une fonction en retournant la valeur de l'expression
- obligatoire dans les fonctions avec type de retour (cf warning)
- termine la fonction : ce qui se situe après n'est pas exécuté.
- dans le cas de la fonction `main`, `return 0` signifie que le programme s'est exécuté avec succès.

Pré-déclaration

```
<type retour> nom(<type1> param_1, <type2> param_2);
```

- permet d'annoncer l'existence de la fonction (un peu comme une déclaration sans initialisation)
- une deuxième déclaration n'importe où dans le fichier

Les conditionnelles

instruction `if`

```
if(<expression>){
  <instructions>
}
```

```
if(<expression>){
  <instructions>
} else {
  <instructions>
}
```

- le `else` est facultatif
- bien respecter les emboîtements.
- il est possible d'omettre les accolades autour des instructions quand elles sont uniques (pas recommandé !, ambiguïté du `else` (demo))
- il existe une forme *ternaire* (déconseillée (`a>0?3:5`))

Les conditionnelles

instruction `switch`

```
switch(<expression>){
   case <expression> :    |
   <instructions>         | répétable
   break;                 |
   default :
   <instructions>
   break;
}
```

- Équivalent à une série de `if - else if` successifs.
- des comportements variés possibles, notamment en jouant sur l'absence de `break`
- à utiliser notamment avec les `enum` (vues plus tard)

Les instructions itératives

La boucle `while`

```
while(<expression>){
   <instructions>
}
```

- l'`<expression>` conditionne la continuation de la boucle.
- risque de boucle infinie si la condition reste vraie
- peut-être interrompue avec un `return`, à utiliser plutôt pour des cas particuliers.

Les instructions itératives

La boucle `for`

```
for(<initialisation>; <condition>; <incrémentation>){
   <instructions>
}
```

---

exemple :

```
for(i = 0; i < 5; i = i + 1){
   x = x + i;
}
```

- Permet de répéter une instruction avec un compteur
- ne pas modifier le compteur dans la boucle
- peut-être interrompue avec un `return`
- préférer la boucle `while` à un usage exotique du `for`
- la condition est testé avant la première instruction.

Choisir entre `for` et `while`

Les deux permettent de faire presque la même chose.

- choisir en fonction du plus naturel dans la façon de penser
- préférer `for` pour les parcours de tableau, les itérations dont on sait la limite
- préférer `while` quand les cas de sortie sont multiples (for qui contiendrait plusieurs `return`) où quand l'itération n'est pas naturellement indicée.

Conclusion

Voilà pour les structures de base, permettent de faire presque tout...