12 Structures

Dans notre code, nous embarquons souvent plusieurs informations qui correspondent en réalité à une même entité et qu'il faut passer à nos fonctions. Une alternative peu propre et qui peut vite poser problème lorsque nous souhaitons réutiliser une fonction est d'avoir ces données en variables globales. Nous nous intéresserons ici à organiser nos informations dans notre code pour plus de maintenabilité.

12.1 Typedef

Une manière de créer un synonyme d'un type est l'opérateur `typedef`. Il peut être pratique lorsqu'on fera appel régulièrement à un type long ou compliqué à écrire. Il est par exemple possible d'abréger l'appel de `unsigned int` en `uint` :

```
typedef unsigned int uint;

int main() {

  uint entierPositif = 4000000000;
  printf("%u\n", entierPositif);

  exit(EXIT_SUCCESS);
}
```

Il est possible de complexifier le type sur lequel on veut faire un alias, comme un tableau ou un pointeur :

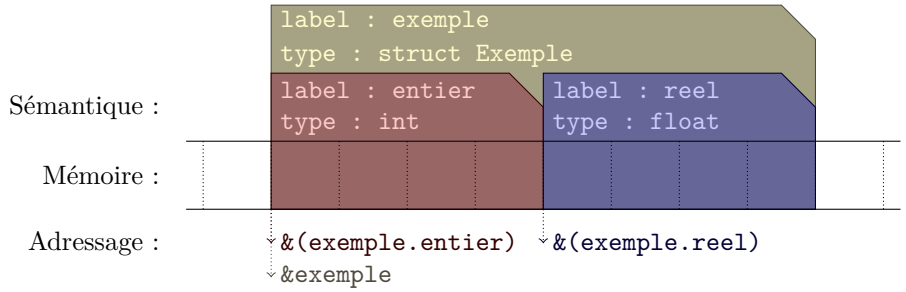
```
typedef int intListeStatique[];
/* intListeStatique sera équivalent au type d'un tableau de int
↪  */
typedef int * intListe;
/* intListe sera équivalent au type d'un pointeur sur un int */
```

```c
/* équivalent à avoir "int * liste" en argument */
void afficherIntListe(intListe liste) {
  int i;
  for(i = 0; liste[i] >= 0; ++i) {
    if(i) printf(", ");
    printf("%d", liste[i]);
  }
  printf("\n");
}

int main() {
  /* équivalent à "int liste[] = {1, 2, 3, 4, -1};" */
  intListeStatique liste = {1, 2, 3, 4, -1};
  afficherIntListe(liste);

  exit(EXIT_SUCCESS);
}
```

12.2 Structures

En langage C, un structure correspondra à une entité qui permettra d'en regrouper plusieurs. Par exemple, lorsque vous avez une liste, il faudra potentiellement avoir à disposition la liste, sa taille, une capacité maximale et autres. Il est possible de modéliser cette liste par un élément qu'est la structure et ne passer que cet élément aux fonctions qui l'auraient en paramètres.

```c
struct Exemple {
  int entier;
  float reel;
};

struct Exemple exemple;
```



12.3 Définition

Une structure se fabrique depuis le mot clé `struct`. On indique ensuite lors de sa définition les champs qui la composent. L'accès aux éléments de la structure se fera ensuite en séparant la variable fabriquée depuis la structure et le champs souhaité par un point `.`.

```c
/* Schéma de construction d'une Liste */
struct Liste {
  int * elements; /* valeurs de la liste */
  int taille; /* taille de la liste */
};

int main() {
  /* Construction d'une liste */
  struct Liste liste;
  /* accès aux champs de la liste */
  liste.elements;
  liste.taille;

  exit(EXIT_SUCCESS);
}
```

Comme vu précédemment, il est possible de fabriquer un alias du même nom pour s'éviter l'écriture de `struct` à la construction d'une liste juste avant sa définition.

```c
/* Alias */
typedef struct Liste Liste;
/* Schéma de construction d'une Liste */
struct Liste {
  int * elements; /* valeurs de la liste */
  int taille; /* taille de la liste */
};

int main() {
  /* Construction d'une liste */
  Liste liste;
  /* accès aux champs de la liste */
  liste.elements;
  liste.taille;

  exit(EXIT_SUCCESS);
}
```

12.4 Avec des pointeurs

À noter que souvent en langage C, les structures seront souvent gérées par pointeurs. L'accès aux champs d'un pointeur sur une structure peut se faire principalement de deux manière équivalentes :

```c
Liste * liste;
/* déréférencement puis accès */
(*liste).elements;
/* version raccourcie */
liste->elements;
```

Il sera souvent plus pratique d'avoir des fonctions qui permettent l'allocation et la libération d'une structure. Ceci en particulier si elle est allouée dynamiquement ou que ses champs peuvent l'être :

```c
Liste * Liste_alloc(int taille) {
  Liste * res = NULL;
  if((res = (Liste *)malloc(sizeof(Liste))) == NULL) {
    fprintf(stderr, "Liste_alloc : Erreur alloc liste\n");
    return NULL;
  }
  res->taille = taille;
  if((res->elements = (int *)calloc(taille, sizeof(int))) == NULL)
  ↪  {
    free(res);
    fprintf(stderr, "Liste_alloc : Erreur alloc éléments\n");
    return NULL;
  }
  return res;
}

void Liste_free(Liste ** liste) {
  free((*liste)->elements);
  free(*liste);
  *liste = NULL;
}

int main() {
  /* Construction d'un pointeur de liste */
  Liste * liste = Liste_alloc(4);
  /* accès aux champs de la liste référencée */
  liste->elements;
  liste->taille;

  Liste_free(&liste);

  exit(EXIT_SUCCESS);
}
```

Il est aussi possible de masquer l'aspect pointeur à l'utilisateur en le précisant

dans le typedef. À noter que si vous souhaitez aller plus loin et respecter les principes d'encapsulation que l'on peut voir dans d'autres langages, vous pouvez déclarer l'existence de la structure à l'avance. Notez que vous ne pourrez l'utiliser dans le code qu'après sa définition ou présenter un pointeur avant (nécessité pour le compilateur de connaître la taille de la structure lorsqu'il opère réellement dessus là où la taille d'une adresse est fixe).

```c
/* Alias version pointeur */
typedef struct Liste * Liste;
/* Déclaration en amont */
struct Liste;

Liste Liste_alloc(int taille);

void Liste_free(Liste * liste);

int main() {

  /* Construction d'un pointeur de liste */
  Liste liste = Liste_alloc(4);
  /* La liste ne peut plus être manipulée directement */
  /* Il faut maintenant passer par des fonctions */
  Liste_free(&liste);

  exit(EXIT_SUCCESS);
}

/* Votre partie du programme : */

/* pourra être cachée à l'utilisateur de votre code */
/* Schéma de construction d'une Liste */
struct Liste {
  int * elements; /* valeurs de la liste */
  int taille; /* taille de la liste */
};
```

```
Liste Liste_alloc(int taille) {
  Liste res = NULL;
  if((res = (Liste)malloc(sizeof(struct Liste))) == NULL) {
    fprintf(stderr, "Liste_alloc : Erreur alloc liste\n");
    return NULL;
  }
  res->taille = taille;
  if((res->elements = (int *)calloc(taille, sizeof(int))) == NULL)
  ↪  {
    free(res);
    fprintf(stderr, "Liste_alloc : Erreur alloc éléments\n");
    return NULL;
  }
  return res;
}

void Liste_free(Liste * liste) {
  free((*liste)->elements);
  free(*liste);
  *liste = NULL;
}
```

12.4.1 Champs de bits

Une optimisation mémoire existe pour les structures. En effet, il est possible de préciser le nombre de bits sur lequel un champ doit être stocké. Ceci se fait en précisant après le champs : puis le nombre de bits nécessaires au maximum pour le champ. Ceci peut être intéressant si la structure contient beaucoup de booléens par exemple et qu'elle est gardée un nombre important de fois en mémoire.

```
struct Personnage {
  unsigned int pointsDeVie;
  unsigned int pointsDeVieMax;
  unsigned int niveau;
  unsigned long experience;
```

```
  unsigned char aStatutPoison;
  unsigned char aStatutParalyse;
  unsigned char aStatutEndormi;
  unsigned int attaque;
  unsigned int defense;
  unsigned int attaqueSpe;
  unsigned int defenseSpe;
  unsigned int vitesse;
};

struct PersonnageCompress {
  unsigned int pointsDeVie : 10;
  unsigned int pointsDeVieMax : 10;
  unsigned int niveau : 7;
  unsigned long experience : 40;
  unsigned char aStatutPoison : 1;
  unsigned char aStatutParalyse : 1;
  unsigned char aStatutEndormi : 1;
  unsigned int attaque : 10;
  unsigned int defense : 10;
  unsigned int attaqueSpe : 10;
  unsigned int defenseSpe : 10;
  unsigned int vitesse : 10;
};

int main() {
  printf("%lu\n", sizeof(struct Personnage));
  printf("%lu\n", sizeof(struct PersonnageCompress));
  exit(EXIT_SUCCESS);
}
```

```
48
24
```

12.5 Unions

Dans une construction similaire à celle des structures, il existe les `union`. À noter qu'une `union` liste des champs qui sont des alias vers une même donnée. C'est-à-dire que si une `union` regroupe un `float` et un `int`, ces deux champs pointent vers le même emplacement mémoire, mais c'est l'accès au champ qui permet de déterminer comment retranscrire l'information en le type demandé.

```
union Scalaire {
  int entier;
  float flottant;
  double flottantPrecis;
};

int main() {
  union Scalaire nombre;
  nombre.entier = 42;
  printf("sizeof(Scalaire) : %lu\n", sizeof(union Scalaire));
  printf("entier :           %d\n", nombre.entier);
  printf("flottant :         %g\n", nombre.flottant);
  printf("flottantPrecis :   %g\n", nombre.flottantPrecis);
  nombre.flottant = 42;
  printf("entier :           %d\n", nombre.entier);
  printf("flottant :         %g\n", nombre.flottant);
  printf("flottantPrecis :   %g\n", nombre.flottantPrecis);
  nombre.flottantPrecis = 42;
  printf("entier :           %d\n", nombre.entier);
  printf("flottant :         %g\n", nombre.flottant);
  printf("flottantPrecis :   %g\n", nombre.flottantPrecis);
  exit(EXIT_SUCCESS);
}
```

Il est possible de créer des structures à l'intérieur de l'union pour que l'information codée dans l'union correspondent à plusieurs champs :

```
union Scalaire {
  int entier;
```

```
  float flottant;
  double flottantPrecis;
  struct {
    float x;
    float y;
  };
};

int main() {
  union Scalaire nombre;
  nombre.x = 42;
  nombre.y = 1337;
  printf("sizeof(Scalaire) : %lu\n", sizeof(union Scalaire));
  printf("entier :           %d\n", nombre.entier);
  printf("flottant :         %g\n", nombre.flottant);
  printf("flottantPrecis :   %g\n", nombre.flottantPrecis);
  printf("(x, y) :           (%g, %g)\n", nombre.x, nombre.y);
  exit(EXIT_SUCCESS);
}
```

12.6 Énumérations

Souvent pour ajouter de la sémantique et de la lisibilité au code, il peut être utile de nommer des constantes. De même, on peut souhaiter considérer qu'une variable ne devrait prendre que certaines valeurs constantes prédéfinies. Il est possible de construire un type dont les valeurs attendues seront celles de noms prédéfinis à l'aide du mot clé `enum` :

```
enum MapItem {
  MAP_VIDE,
  MAP_JOUEUR,
  MAP_ADVERSAIRE,
  MAP_MUR,
  MAP_SORTIE
};
```

```
enum MapItem item;
```

Il est possible de se passer du mot clé `enum` lors de la déclaration d'une variable de ce type énuméré à l'aide d'un typedef :

```
typedef enum {
  MAP_VIDE,
  MAP_JOUEUR,
  MAP_ADVERSAIRE,
  MAP_MUR,
  MAP_SORTIE
} MapItem;

MapItem item;
```

Ces types énumérés conviennent parfaitement à une utilisation dans un `switch`. À noter qu'il est recommandé d'ajouter l'option `-Wall` lors de la compilation. Si un élément de type énuméré n'est pas géré par le switch ceci sera précisé à la compilation.

```
typedef enum {
  ITEM_MOB_DYNAMIC,
  ITEM_MOB_STATIC,
  ITEM_MOB_UNKNOWN
} ItemMobility;

ItemMobility getMapItemMobility(MapItem item) {
  switch(item) {
    case MAP_JOUEUR :
    case MAP_ADVERSAIRE :
    return ITEM_MOB_DYNAMIC;

    case MAP_MUR :
    case MAP_SORTIE :
    return ITEM_MOB_STATIC;
```

```
    default :
    return ITEM_MOB_UNKNOWN;
  }
}
```

À noter qu'en réalité un type énuméré est un entier dont certaines valeurs sont associées à un nom pour s'accorder avec l'utilisation qui doit en être faite. Il est possible dans l'énumération d'encoder certaines valeurs. Les valeurs successives suivront de un en un.

```
typedef enum {
  MAP_VIDE = 0,
  MAP_JOUEUR = 10,
  MAP_ADVERSAIRE,
  MAP_MUR = 20,
  MAP_SORTIE
} MapItem;
```

Puisque un type énuméré correspond à un entier et qu'un caractère aussi, il est possible de faire correspondre les valeurs du type énuméré à des caractères. Ceci peut apporter de la lisibilité dans le code et aussi avoir une conversion directe si cela devait être éditable dans un fichier par exemple.

```
typedef enum {
  MAP_VIDE =       ' ',
  MAP_JOUEUR =     '@',
  MAP_MUR =        '#',
  MAP_ADVERSAIRE = '£',
  MAP_SORTIE =     'x'
} MapItem;
```

12.7 Résumé

L'opérateur `typedef` permet de créer un synonyme d'un type :

```
typedef type synonyme;
```

Il est possible de créer des type structurés regroupant des champs sous un même nom :

```
struct Nom {
  typeChamp nomChamp;
};
struct Nom variable;
struct Nom * pointeur = &variable;
variable.nomChamp; /* accès au champ */
pointeur->nomChamp; /* accès au champ via pointeur */
```

De la même manière, on peut regrouper des champs sous un même nom mais que ceux-cis correspondent à un même emplacement mémoire pour réduire la taille de ce groupement en mémoire.

```
union Nom {
  int entier;
  float reel;
};
union Nom nombre;
nombre.entier = 42; /* nombre.reel est aussi modifié */
nombre.reel = 13.37; /* nombre.entier est aussi modifié */
```

Lorsqu'un entier prend des valeurs que l'on veut nommer, on peut construire un type énuméré :

```
typedef enum {
  VALEUR1 = 0x1,
  VALEUR2 = 0x2
} Liste;
Liste liste = VALEUR1;
```

12.8 Entraînement

Exercice noté 46 (⋆⋆⋆ Définir une structure Vecteur2d).

Définir une structure `Vecteur2d` avec les champs suivants :

- `double` x.
- `double` y.

Puis définir des fonctions qui permettent la manipulation d'un élément $V = (V_x, V_y)$:

- **Translation** de vecteur $T = (T_x, T_y)$:

$$V = V + T$$

- **Agrandissement** de rapport α et de centre $C = (C_x, C_y)$:

$$V = \alpha\,(V - C) + C$$

- **Rotation** d'angle δ et de centre $C = (C_x, C_y)$:

$$V = \begin{pmatrix} \cos(\delta) & -\sin(\delta) \\ \sin(\delta) & \cos(\delta) \end{pmatrix} (V - C) + C$$

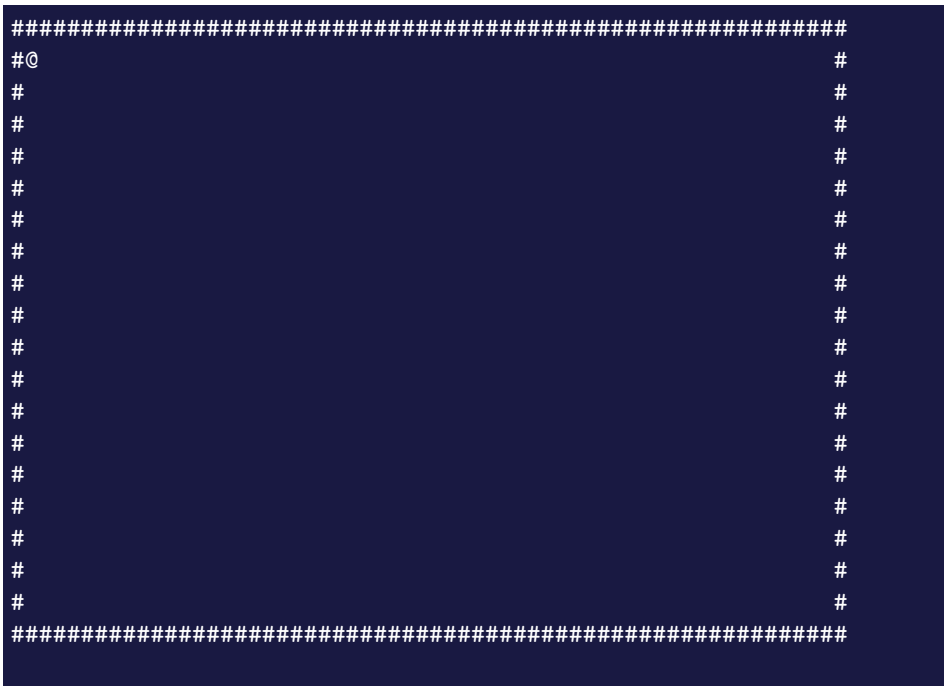
```
Vecteur2d : (0, 0)
Translation par un Vecteur2d : (1, 2)
(1, 2)
Agrandissement de rapport 0.5 et de centre un Vecteur2d : (1, 0)
(1, 1)
Rotation d'angle 135 deg et de centre un Vecteur2d : (0, 2)
(1.11022e-16, 3.41421)
```

Exercice noté 47 (⋆⋆⋆ Structure pour gérer une grille).

Compléter le code suivant pour qu'il affiche une grille à l'écran :

```c
#include <stdio.h>
#include <stdlib.h>
#include <ncurses.h>

typedef struct Grille Grille;
struct Grille {
  char * grille;
  int largeur;
  int hauteur;
};

/* donne un pointeur sur une case de la grille */
char * Grille_case(const Grille * grille, int x, int y);

/* crée une grille de taille donnée */
Grille Grille_creer(int largeur, int hauteur);

/* affiche une grille à l'écran */
void Grille_afficher(const Grille * grille);

/* libère une grille */
void Grille_free(Grille * grille);

int main() {
  int largeur = 60, hauteur = 20;
  Grille grille = Grille_creer(largeur, hauteur);
  int x = 1, y = 1;
  initscr();
  noecho();
  cbreak();
  do {
    clear();
    Grille_afficher(&grille);
    mvprintw(y, x, "@");
    mvprintw(y, x, "");
    refresh();
    getch();
    /* gestion des événements */
```

```
  } while(1);
  refresh();
  clrtoeol();
  refresh();
  endwin();
  Grille_free(&grille);
  exit(EXIT_SUCCESS);
}
```



Exercice noté 48 (⋆⋆⋆⋆ Gérer un combat de personnages).

Permettre à deux personnages de s'affronter. On vous demande de mettre en place :

- Des `Stats` contenant la vie, l'attaque, la défense et la vitesse.
- Énumération des stats.
- Des `Personnages` symbolisés par un nom, des statistiques de base et des statistiques actuelles.
- Énumération des sorts (qui influent sur les stats du joueur et ceux de l'adversaire).

Voici l'interface de combat à proposer :

```
              Joueur VS Adversaire
>>>>>>>>>>>>>>>>>>> | <<<<<<<<<<<<<<<<<<<
Vie :            100 | Vie :            100
Attaque :         50 | Attaque :         50
Defense :         20 | Defense :         20
Vitesse :        100 | Vitesse :        100
--------------------+--------------------
1 . coup              5 . soin
2 . reduire attaque   6 . augmenter attaque
3 . reduire defense   7 . augmenter defense
4 . reduire vitesse   8 . augmenter vitesse
--------------------+--------------------
Votre sort :
```

Exercice noté 49 (⋆⋆⋆⋆ Implémenter une liste chaînée).

Un informaticien vous voit utiliser uniquement des tableaux pour vos liste. Il vous informe qu'il préfère utiliser des liste chaînées dans certains cas. Il vous propose d'implémenter les fonctionnalités d'une liste chaînée et d'une liste depuis un tableau pour vous en faire une idée vous-même. Vous préciserez pour chaque fonction sa complexité algorithmique en fonction de la taille de la liste ($\mathcal{O}(1)$ pour constante lorsque ça ne dépend pas de la taille de la liste et $\mathcal{O}(n)$ pour linéaire lorsqu'on pourrait au pire des cas passer sur chaque élément de la liste).

```c
typedef struct LinkedList * LinkedList;
struct LinkedList {
        int value;
        LinkedList next;
};

/* Renvoie une liste vide */
LinkedList LL_empty();

/* Libère la liste */
void LL_free(LinkedList * liste);

/* Ajoute un élément en fin */
int LL_add_tail(LinkedList * liste, int value);

/* Ajoute un élément en tête */
int LL_add_head(LinkedList * liste, int value);

/* Ajoute un élément à une position donnée */
int LL_insert(LinkedList * liste, int id, int value);

/* Supprime l'élément en fin */
int LL_pop_tail(LinkedList * liste, int * value);

/* Supprime l'élément en tête */
int LL_pop_head(LinkedList * liste, int * value);

/* Supprime l'élément à une position donnée */
int LL_delete(LinkedList * liste, int id, int * value);

/* Affiche la liste */
void LL_print(FILE * flow, const LinkedList * liste);
```

```c
typedef struct ArrayList ArrayList;
struct ArrayList {
        int * values;
        int size;
        int capacite;
};

/* Renvoie une liste vide */
ArrayList AL_empty();

/* Libère la liste */
void AL_free(ArrayList * liste);

/* Ajoute un élément en fin */
int AL_add_tail(ArrayList * liste, int value);

/* Ajoute un élément en tête */
int AL_add_head(ArrayList * liste, int value);

/* Ajoute un élément à une position donnée */
int AL_insert(ArrayList * liste, int id, int value);

/* Supprime l'élément en fin */
int AL_pop_tail(ArrayList * liste, int * value);

/* Supprime l'élément en tête */
int AL_pop_head(ArrayList * liste, int * value);

/* Supprime l'élément à une position donnée */
int AL_delete(ArrayList * liste, int id, int * value);

/* Affiche la liste */
void AL_print(FILE * flow, const ArrayList * liste);
```