INFO-F105 – Langages de programmation 1
PODs

1 Introduction

Une *structure de données* est un moyen d'organiser plusieurs données dans une seule variable.

Quelques exemples en Python :

- Listes
- Tuples
- Dictionnaires

En C++, nous avons déjà vu les tableaux (*arrays*) dans le chapitre Pointeurs & Tableaux, mais il en existe d'autres que nous allons détailler dans ces deux prochaines sessions.

Classification

On peut distinguer deux types de structures de données :

- Les structures *passives*, dont l'unique vocation est de stocker des données
- Les structures *actives*, qui peuvent également manipuler ces données

Ce chapitre porte uniquement sur les structures de données passives en C++, qu'on appelle aussi *Plain Old Data* (PODs). Les stuctures actives seront abordées dans le chapitre ADTs.

2 Énumérations

Une *énumération* est un type dont les valeurs font partie d'un ensemble de constantes.

```
enum class Course {
  INFO_F101,
  INFO_F102,
  INFO_F103,
  INFO_F105,
  INFO_F106,
};

Course course = Course::INFO_F105;
```

Les énumérations sont souvent utilisées dans des `switch`.

```
std::string name(Course course) {
  switch (course) {
    case Course::INFO_F103:
      return "Algorithmique 1";
    case Course::INFO_F105:
      return "Langages de programmation 1";
    // ...
  }
}
```

2.1 Typage & conversion

Il est possible de définir des valeurs aux constantes.

```
enum class Direction: unsigned char {
  UP = 'z',
  LEFT = 'q',
  RIGHT = 's',
  DOWN = 'd',
};

Direction direction = Direction('z');  // UP
std::cout << char(direction) << std::endl;
```

Il devient alors possible de faire des conversions entre l'`enum` et ces valeurs.

2.2 C enum

Il existe aussi un autre type de `enum`, hérité du langage C, avec deux différences majeures :

1. Les constantes ne sont pas à l'intérieur du namespace de l'`enum`.
2. Elles permettent des conversions implicites vers des entiers.

Ces particularités viennent chacune avec un inconvénient :

1. Pollution de l'espace de nommage global (cf. Namespace)
2. Les conversions implicites peuvent nuire à la clarté du code

```
enum Color {
  RED,
  GREEN,
  BLUE,
  YELLOW,
};

Color color = RED;
assert(color == 0);
```

3 Structures

Une *structure* est un agrégat de données contiguës en mémoire (les *champs* = *fields*).

```
struct Point {
  float x;
  float y;
};

Point p1 = { .x = 5, .y = 10 };
Point p2 = { 2, 4 };

Point center = {
  .x = p1.x + (p2.x - p1.x) / 2,
  .y = p1.y + (p2.y - p1.y) / 2,
};
```

Les syntaxes d'initialisation de `p1` et `p2` sont équivalentes. Si les noms des champs ne sont pas précisés, le compilateur considérera qu'ils sont dans le même ordre que dans la définition.

L'opérateur '`.`' dans `p1.x` permet d'accéder au champ `x` de `p1`.

3.1 Pointeurs

```
Point* ptr1 = new Point { .x = 5, .y = 10 };

Point center_with_origin = {
  .x = p1->x / 2,
  .y = p1->y / 2,
};
```

L'opérateur '`->`' dans `ptr1->x` permet d'accéder au champ `x` de `*ptr1`.

Rappel : n'oubliez pas de `delete` les données créées à l'aide d'un `new`.

3.2 Imbrications

Il est tout à fait possible d'imbriquer des `struct`.

```
struct Line {
  Point points[2];
  enum {
    RED, GREEN, BLUE, YELLOW
  } color;
};

Line line = { .points = { p1, p2 }, .color = Line::RED };
```

Dans cet exemple, une `Line` contient deux `Point` organisés dans un tableau.

Le type de `color` est une `enum` anonyme avec 4 valeurs possibles : `RED`, `BLUE`, `GREEN`, `YELLOW`.

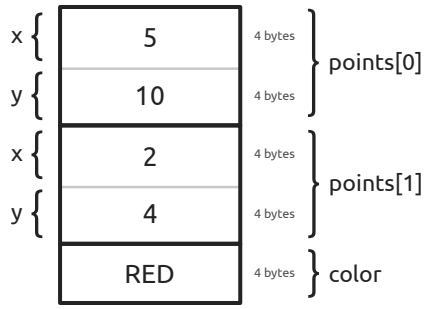

Figure 1: `struct Line` en mémoire

4 Unions

Une *union* est une structure où les membres partagent le même espace mémoire.

Cela permet de définir un type qui peut contenir soit l'un soit l'autre des membres.

```
union Node {
  struct {
    Node* left;
    Node* right;
  } children;
  int data;
};

Node leaf = { .data = 5 };
Node node { .children = { nullptr, &leaf } };

int data = node.children.right->data;
```

Dans cet exemple, un `Node` est le nœud d'un arbre binaire [1] et peut contenir soit une donnée, soit des pointeurs vers ses nœuds enfants (cf. figure 2).

Le type de `children` est une `struct` anonyme avec deux membres : `left` et `right`. Il est nécessaire d'utiliser des pointeurs `Node*` plutôt que des `Node` car il n'est pas possible de faire des imbrications récursives en C++ sans passer par des pointeurs.

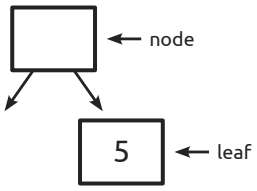

Figure 2: Arbre utilisant l'`union Node`

[1] Voir cours d'algorithmique 1