Langage *C* : Structures / modularité

Sébastien Jean

IUT de Valence
Département Informatique

v1.0, 19 novembre 2025

IUT Valence
Université Grenoble Alpes

Interlude : Création/clonage d'un projet Gitlab

- **Créer un projet** `StructuresC` sur Gitlab (avec un `README`)
- **Cloner** le projet depuis *VsCode* et ouvrir le dépôt
- **Modifier** le fichier `README.md` pour indiquer à quoi sert ce projet
- N.B. : pour chacun des exemples/exercices `ExX` suivant :
  - **Ecrire le programme** dans `ExX/src/main.c` et le compiler dans `ExX/build/ExX`
  - `Rédiger (si nécessaire) un jeu d'essai` dans un fichier `ExerciceX/Essai`

IUT Valence Université Grenoble Alpes

Enregistrements (rappels)

- Un **enregistrement** est **composé de plusieurs valeurs** appelées champs ou membres
  - Nombre de champs **fixe**, champs **nommés** et de **type quelconque**
  - Les opérations se limitent à la **lecture et l'affectation des champs**
  - Accès aux champs via la **notation pointée** (`variable.champs`)

```
ENREGISTREMENT Point

  CHAMPS x  : réel
  CHAMPS y  : réel

FIN ENREGISTREMENT
```

```
VARIABLE p : Point

  p.x ← 0.0
  p.y ← p.x + 1
```

IUT Valence Université Grenoble Alpes

C : structures

- Déclaration d'une structure suivant la syntaxe `struct {...}`

Code C

```c
struct Point {
 double x;
 double y;
};
```

- Déclaration d'une variable suivant la syntaxe `struct type nom`

Code C

```c
struct Point p;
```

C : structures

- Initialisation d'une variable

Code C

```c
struct Point p1 = {1.0, 0.0}; // ordre des champs

struct Point p2 = {.y = 0.0, .x = 1.0}; // désordre

struct Point p3; // pas d'initialisation explicite
```

- Accès aux champs suivant la syntaxe `variable.champs`

Code C

```c
p3.x = 1.0; // ecriture
p3.y = 0.0;
p2.y = p1.y; // lecture
```

Exercice

Enoncé du problème

On veut afficher les coordonnées d'un point sur la sortie standard

Spécification du problème

- **Donnée d'entrée** : *p*, Point (le point à afficher)
- **Donnée de sortie** : (aucune)
- **Pré-condition** : (aucune)
- **Post-condition** : le point est affiché sous la forme `(x, y)`

Signature de la fonction

- **afficher_point** (p : Point) : (aucun)

IUT Valence
Université Grenoble Alpes

Exercice

- Donner l'implémentation en C (fonction + `main`)
  - N.B : définir la structure globalement

IUT Valence
Université Grenoble Alpes

Exercice : Point

Code C

```c
#include <stdio.h>

struct Point {
    double x;
    double y;
};

void afficher_point(struct Point p) {
    printf("(%.1f,%.1f)\n", p.x, p.y);
    // limite l'affichage à 1 chiffre après la virgule
}

int main() {
    struct Point point = {1.0, 0.0};
    afficher_point(point);
}
```

C : pointeurs sur structures

- L'accès au champs peut se faire directement avec `->`

Code C

```c
struct Point p = {1.0, 0.0};

struct Point *ptr = &p;

(*ptr).x = -1.0; // déréférencement + accès au champs

ptr->y = 2.0; //  accès direct au champs
```

IUT Valence
Université Grenoble Alpes

Exercice

Enoncé du problème

On veut décaler un point en x et en y.

Spécification du problème

- **Donnée d'entrée** : *p*, Point (le point à décaler)
- **Donnée d'entrée** : *dx*, réel (le décalage en x)
- **Donnée d'entrée** : *dy*, réel (le décalage en y)
- **Donnée de sortie** : *r*, Point (le point décalé)
- **Pré-condition** : (aucune)
- **Post-condition** : si `p` est en `(x,y)` alors `r` est en `(x+dx, y+dy)`

Signature de la fonction

- **decaler_point** (p : Point, dx : réel, dy : réel) : Point

Exercice

- Donner l'implémentation en C (fonction + `main`)
  - N.B. le point en paramètre est modifié en place, la fonction retourne `void`

IUT Valence
Université Grenoble Alpes

Exercice : Point

Code C

```c
#include <stdio.h>

// ... définition de la structure Point
// ... fonction afficher_point

void decaler_point(struct Point *ptr, double dx, double dy) {
    ptr->x = ptr->x + dx;
    ptr->y = ptr->y + dy;
}

int main() {
    struct Point point = {1.0, 0.0};
    afficher_point(point);
    decaler_point(&point, -3.0, 4.5);
    afficher_point(point);
}
```

Alias

- Définition d'un alias de type : `typedef type synonyme`

Code C

```c
#include <stdio.h>

#define TAILLE 10

struct sPoint {
    double x;
    double y;
};

typedef struct sPoint Point; // alias

typedef Point TableauPoints[TAILLE]; // alias

(...A suivre)
```

Alias

- Réécrire la fonction `afficher_point` en utilisant le type `Point`
- Dans la fonction `main` :
  - Créer une variable de type `TableauPoints`
  - Itérer sur les cases, affecter les coordonnées et afficher le point

IUT Valence
Université Grenoble Alpes

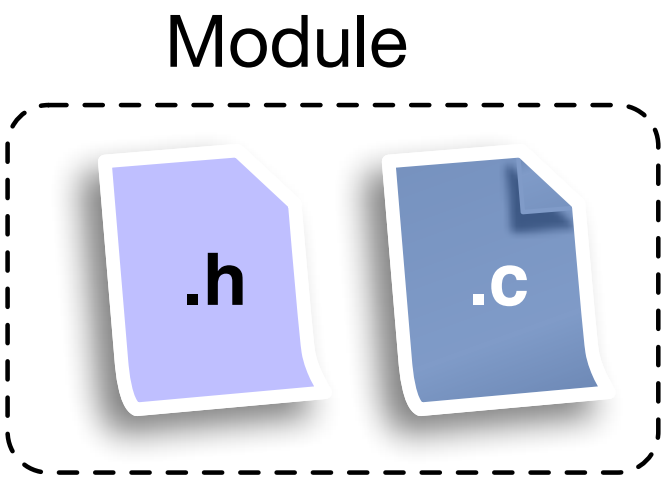
Alias

Code C

```
(...Suite)

void afficher_point(Point p) {
    printf("(%.1f,%.1f)\n", p.x, p.y);
}

int main() {
    TableauPoints t;
    for (int i=0; i<TAILLE; i++) {
        t[i].x = i;
        t[i].y = -i;
        afficher_point(t[i]);
    }
}
```

Modularité

- Plutôt que de redéfinir dans chaque application un même type et les fonctions associées, on les écrit une fois pour toutes dans un ***module***
- Le *module* est décomposé en 2 parties
  - Fichier `.h` (*header*) : **définitions** des constantes, de types, et de **signatures** de fonctions
  - Fichier `.c` : **code complet** des fonctions



IUT Valence Université Grenoble Alpes

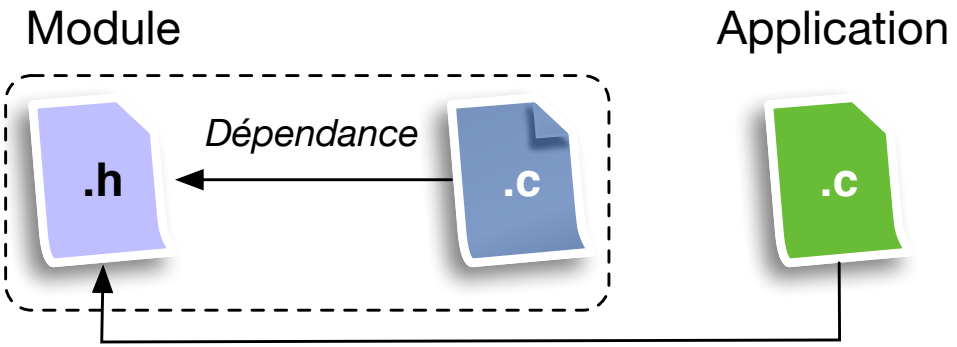
Modularité

Code C du fichier point.h :

```c
#define TAILLE 10    // définition de constante


struct sPoint {      // définition de type
    double x;
    double y;
};

// définition de type (alias)
typedef struct sPoint Point;

// signatures de fonctions
void afficher_point(Point p);
void decaler_point(Point *ptr, double dx, double  dy);
```

Modularité

- Un fichier `.c` qui contient l'implémentation d'une fonction ou utilise une fonction, une constante ou un type défini dans un `.h` doit **inclure** le fichier `.h` via la directive `#include`

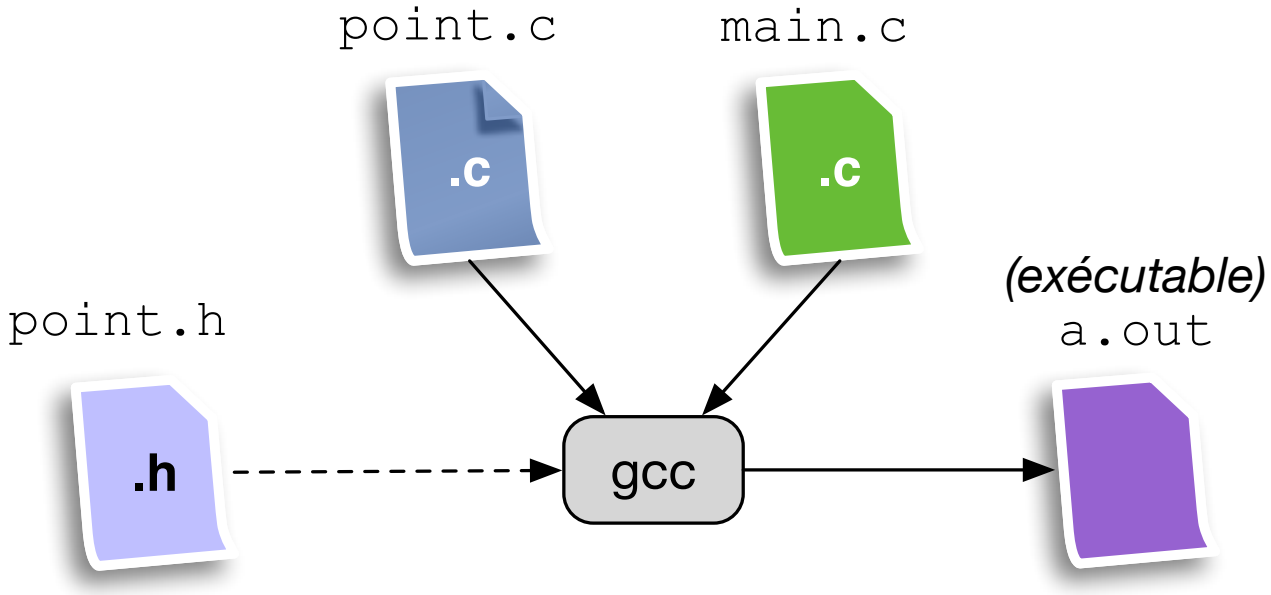


- `#include <stdio.h>`
  - `<...>` utilisé pour les modules standards (*libC*)
- `#include "point.h"`
  - `"..."` utilisé pour les autres modules
  - **chemin relatif** au fichier `.h` , **recherche** à partir du répertoire courant ou de chemins de recherche (cf. option `-I` de *gcc*)

IUT Valence Université Grenoble Alpes

Modularité

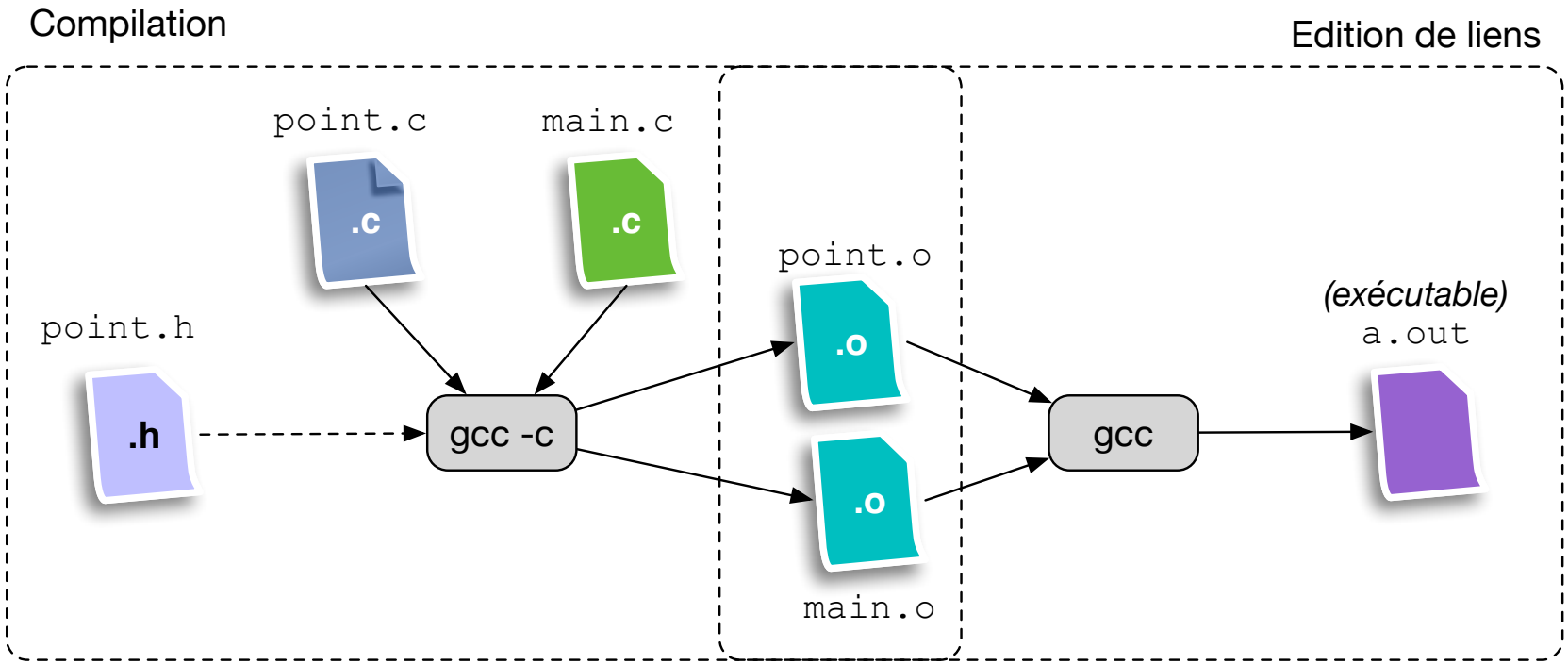
- La production de l'exécutable nécessite de fournir **tous les fichiers `.c` à la chaîne de compilation**



- En ligne de commande : `gcc main.c point.c`

Modularité

- **Deux étapes successives : compilation et édition de liens**



- En ligne de commande : `gcc -c main.c point.c`
  - Production des fichiers `.o`
- En ligne de commande : `gcc main.o point.o`
  - Production de l'exécutable

IUT Valence Université Grenoble Alpes

Exercice

- Ecrire `point.c`, réécrire `main.c`
- Compiler (avec ou sans production des fichiers `.o` intermédiaires)
- Exécuter l'application
- N.B. sources dans `src`, executable et `.o` dans `build`

IUT Valence
Université Grenoble Alpes

Exercice : point.c

Code C

```c
#include <stdio.h>
#include "point.h"

void afficher_point(Point p) {
    printf("(%.1f,%.1f)\n", p.x, p.y);
}

void decaler_point(Point *ptr, double dx, double dy) {
    ptr->x = ptr->x + dx;
    ptr->y = ptr->y + dy;
}
```

IUT Valence Université Grenoble Alpes

Exercice : main.c

Code C

```c
#include <stdio.h>
#include "point.h"

typedef Point TableauPoints[TAILLE];

int main() {
    TableauPoints t;
    for (int i=0; i<TAILLE; i++) {
        t[i].x = i;
        t[i].y = -i;
        afficher_point(t[i]);
    }
}
```

Fin !

IUT Valence
Université Grenoble Alpes