Ch. 4 - Structures

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

11 octobre 2021

Table des matières

Introduction

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
  - Dans une `union`, la mémoire est partagée entre les attributs
- Accès aux attributs
  - via `->` si derrière un pointeur
  - via `.` sinon
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`

Exemple

- Fichier `basic.c`

```c
struct point
{
    int x, y;
};

int main()
{
    struct point p;
    p.x = 1;
    p.y = 2;
    printf("(%d,%d)\n", p.x, p.y);

    struct point * pt = &p;
    pt->x = 3; //same as (*pt).x = 3;
    pt->y = 4;
    printf("(%d,%d)\n", pt->x, pt->y);
}
```

Alias de structure

- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

```
1  struct A {};
2  typedef struct A A; //alias
3
4  //void print_addr(struct A* a) //if no alias
5  void print_addr(A * a)
6  {
7      printf("%p\n", a);
8  }
```

- Fichier `typedef.c`

Structures

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Exemple

- Fichier `point.c`

```c
struct point
{
  double x, y;
};

struct point2
{
  double x, y;
};

typedef struct point point;
typedef struct point2 point2;

double dist(point p1, point p2)
{
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}
```

Exemple

- Fichier `point.c`

```
1  int main()
2  {
3    point p;
4
5    printf("%f %f\n", p.x, p.y);//undefined
6
7    p.x = p.y = 0;
8
9    point p2 = {1, 1}; //try with = {1}
10   printf("%f %f\n", p2.x, p2.y);
11
12   printf("%f\n", dist(p, p2));
13
14   point2 brol;
15   //point p3 = (point)brol;
16   //dist(p1, brol); //no conversion
17 }
```

Unions

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Exemple (1/2)

- Fichier `union.c`

```c
union Data
{
   int i;
   float f;
   char str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   data.f = 220.5;
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
}
```

Champs de bits

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Types autorisés

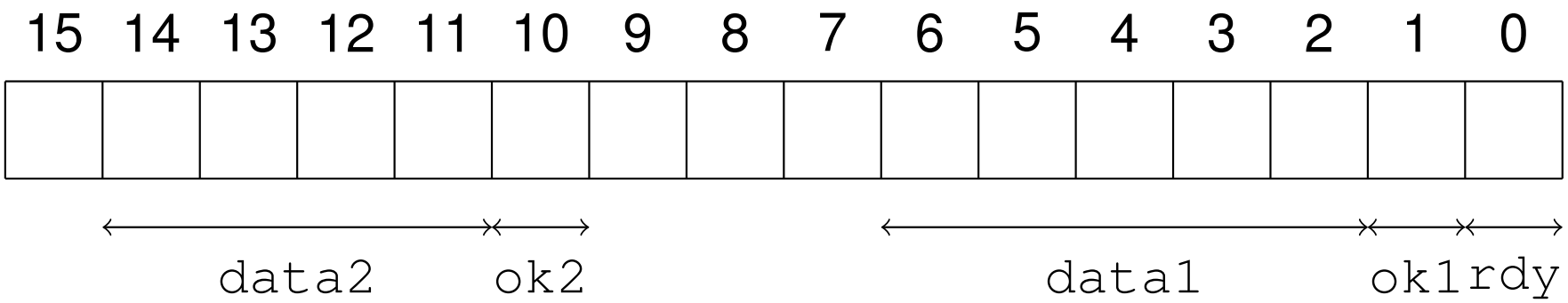
- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Exemple

```
1  struct state
2  {
3    unsigned rdy : 1;
4    unsigned ok1 : 1;
5    int data1    : 5;
6    int          : 3;
7    unsigned ok2 : 1;
8    int data2    : 4;
9  };
```



Énumérations

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Exemple

- Fichier `enum.c`

```
1  enum couleur {  rouge, vert, bleu   };
2  enum boolean {vrai = 1, faux = 0};
3
4  main()
5  {
6    enum couleur c1 = rouge;
7    enum couleur c2 = c1;
8
9    printf("%i\n", c1);
10   printf("%i\n", c2);
11
12   int n = bleu;
13   int p = vert * n + 2;
14
15   printf("%i\n",n);
16   printf("%i\n",p);
17   //vert = n;
18
19   enum couleur c3 = faux;
20   enum boolean brol = 3 * c2 + 4;
21
22   printf("%i\n",c3);
23   printf("%i\n",brol);
24 }
```

Exemple

- Fichier `enum2.c`

```
1  typedef enum color { RED, GREEN, BLUE} color;
2
3  enum Foo { A, B, C=10, D, E=1, F, G=F+C}; //don't try this at home
4  //A=0, B=1, C=10, D=11, E=1, F=2, G=12
5
6  int main()
7  {
8      color c = RED;
9
10     switch(c)
11     {
12         case RED   : printf("red\n"); break;
13         case GREEN : printf("green\n"); break;
14         case BLUE  : printf("blue\n"); break;
15     }
16 }
```

Déclaration, définition et inclusion

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

Exemple avec fonctions indépendantes

Fichiers `fct-decl.h`

```
1  void print_str(const char*);
```

Fichier `fct-decl.c`

```
1  #include "stdio.h"
2  #include "fct-decl.h"
3
4  void print_str(const char* s)
5  {
6    printf("%s\n", s);
7  }
```

Fichier `fct-decl-main.c`

```
1  #include "fct-decl.h"
2
3  int main()
4  {
5    print_str("Hello World!");
6  }
```

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Exemple

- On veut modéliser le répertoire `home` d'un utilisateur
  - Un utilisateur a un `home`
  - Chaque répertoire `home` appartient à un utilisateur
- Fichiers `*-wrong.*`

```c
#include "user-wrong.h"

typedef struct HomeDir
{
    char home[20];
    User user;
} HomeDir; //typedef in one go
```

```c
#include "homedir-wrong.h"

typedef struct User
{
    char name[20];
    Homedir home;
} User; //typedef in one go
```

Problèmes

Inclusion multiple de headers

- le compilateur tourne en boucle dans les `#include`

- Il manque les `#ifndef` / `#define`

Problèmes de taille

- Il est impossible de calculer `sizeof` pour `User` et `HomeDir`

- Il faut que l'un des attributs soit un pointeur
  - Ainsi, on sait calculer la taille

Solution

- Fichiers `*-fixed.*`

```
1  #ifndef USER_H
2  #define USER_H
3  //no include here
4
5  struct HomeDir; //anticipated declaration
6
7  typedef struct User
8  {
9      char name[20];
10     struct HomeDir* home; //here, no alias is defined
11 } User;
12
13 #endif
```

```
1  #ifndef HOMEDIR_H
2  #define HOMEDIR_H
3  #include "user-fixed.h"
4
5  typedef struct HomeDir
6  {
7      char home[20];
8      User user; //here, the alias exists
9  } HomeDir;
10
11 #endif
```