Chapitre 3
Types composés

Plan du chapitre 3

Introduction

- But : définir des types de données structurées en C
  - à la manière des classes C++
  - appelés parfois enregistrements (*records*) dans d'autres langages
  - regroupent plusieurs informations au travers de différents champs
- De quoi dispose-t-on en C ?
  - seul type structuré vu jusqu'à présent : tableaux (*arrays*)
  - les tableaux (1D, 2D, ...) offrent de nombreuses possibilités
    - liens avec les pointeurs (chapitre 2) et les chaînes (chapitre 4)
- **Limitation des tableaux** : tous leurs éléments ont le même type
  - difficulté à modéliser des données structurées

1. Les structures : définition et exemples

Structure : définition

- C offre la possibilité de définir ce que l'on nomme des **structures**.

- On *définit* d'abord un modèle de structure (type), puis on *déclare* autant d'instances (variables) que nécessaire
  - **N.B.** Souvent les instances sont aussi appelées structures

- Une structure regroupe un ou plusieurs éléments appelés **membres** ou **champs**, qui *peuvent avoir des types différents*
  - chaque membre a un nom, qui permet d'y accéder
  - contrairement aux tableaux où on utilise les index [*n*]

N.B. Tous les membres d'une structure sont publics par définition (pas d'encapsulation en C !)

Structure : syntaxe

- Mot-clé `struct`
- Nom de la structure, i.e.
  nom du *modèle de structure* (*type*)

```
struct [Name] {
  /* au moins un membre */
  memberDecl;
  [memberDecl; ...]
} [variableId, ...]; /* premières variables */
```

  - ce nom peut être optionnel (« type anonyme »), si on ne compte pas réutiliser la structure pour déclarer d'autres instances (variables) que les toutes premières
  - les membres (champs) sont déclarés comme des variables habituelles ([qualifieur] + type + identificateur)
  - **Attention** : en C (contrairement à C++) :
    - le qualifieur `static` ne peut pas être utilisé
    - les champs ne peuvent pas être initialisés explicitement
  - on peut déclarer déjà les premières variables du type structure à la fin de la définition

Structure : exemples

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};
struct Date today, tomorrow;
```

```
struct {// struct anonyme (≈ Person)
   char    name[MAX_SIZE];
   uint8_t age;
} john, mary;
```

- **Structure `Date`** pour stocker des dates, avec trois champs
  - par analogie avec les classes C++, on peut utiliser une *majuscule* initiale pour le modèle de structure
  - les membres (en *minuscules*) sont stockés en mémoire dans l'ordre où ils ont été définis
- **Structure anonyme** s'il n'est plus nécessaire de faire d'autres déclarations
  - Ici, seuls « john » et « mary » seront déclarés (en *minuscules*)

Structure utilisant une autre structure

- Les membres d'une structure peuvent être de n'importe quel type, y compris d'un autre type structure

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};
```

```
struct Person {
   char name[MAX_SIZE];
   struct Date birthday;
} john, mary;
```

- Les structures existent en C et en C++, avec une différence
  - **en C++, `struct` est optionnel** (et peu utilisé) quand on déclare des variables, par analogie avec `class`
  - **en C, `struct` est obligatoire** : `struct Person kevin;`

Portée d'un type struct

- Si un modèle de structure (type `struct`) est défini dans une fonction (qui peut être le *main)*, il n'est accessible que depuis celle-ci
  - « accessible » = permet de créer des instances
- Si la définition n'est pas dans une fonction, elle est accessible dans toute la partie du fichier qui suit
- Comment partager une déclaration de type `struct` entre plusieurs fichiers ?
  - l'écrire dans un fichier header et inclure ce dernier dans tous les fichiers source où on en aura besoin
  - on ne peut pas utiliser `extern` pour réutiliser un modèle de structure défini dans un autre fichier source non inclus
  - utiliser une déclaration anticipée (voir plus loin)

Portée des noms de membres

- Les noms des membres définis dans une structure n'entrent pas en conflit avec d'autres variables, ou avec des noms de membres d'autres structures
- On dit que chaque structure a un espace de noms (*namespace*) séparé
- Exemple :
  Ici, pas de conflit entre « day » membre d'une date et la variable « day » séparée

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};

uint8_t  day;
uint8_t  month;
uint16_t year;
```

Déclaration anticipée

- La **déclaration anticipée** ou partielle (***forward declaration***) annonce une structure, et promet sa définition ultérieure
- Elle ne suffit pas pour utiliser la structure, mais elle suffit pour utiliser des pointeurs vers elle, p.ex. dans des déclarations de fonctions ou autres structures

```
struct Date;
/* déclaration anticipée */

struct Person {
   char name[MAX_SIZE];
   struct Date* birthday;
}; /* marche pas sans * */

struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};
```

Déclaration anticipée

- En fait, le simple fait d'écrire « `struct NOM` » réalise une déclaration partielle implicite de NOM, avec une portée locale, mais qui souvent suffit

```
struct Element;
/* déclaration anticipée,
   en fait optionnelle ici
*/

struct Element {
   int   num;
   float x;
   float y;
   struct Element* next;
};
```

- **La déclaration anticipée reste une bonne pratique !**
- Clarifie le statut des types
- Nécessaire avec des fonctions
- Peut éviter des circularités

2. Accès aux membres et initialisation

Opérateurs pour les structures

- Trois opérateurs importants
  - **Point** (.) pour accéder aux membres : `today.day`
  - **Egal** (=) pour l'affectation, y compris l'initialisation
  - Pointeurs sur des structures : * et &
    - **Flèche** : `(*pToday).day` s'écrit : `pToday->day`
- Pas d'autres opérateurs applicables aux structures (à moins de les surcharger explicitement (en C++))
  - en particulier, **on ne peut pas utiliser == ou !=**

Déclaration et initialisation

- Lorsqu'on déclare une variable de type structure (une instance), on peut l'initialiser dans la même instruction
- Même syntaxe que pour les tableaux
  - on fournit les valeurs des membres *dans l'ordre*, entre accolades
  - s'il y en a moins, les membres restants reçoivent la valeur 0 (mais warning : *missing initializer for field…)*

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};

struct Date
   myBirthday   = {1, 1, 1970},
   yourBirthday = {3, 4, 2000};
```

- Depuis C99, les expressions dans la liste d'initialisation n'ont plus à être statiques (évaluables à la compilation); elles peuvent être dynamiques.

Exemple particulier

- Déclaration et initialisation d'un tableau (constant) de `Date`

```
const struct Date JOURS_FERIES_VAUD_2019[] = { { 1,  1, 2019},
                                               { 2,  1, 2019},
                                               {19,  4, 2019},
                                               {22,  4, 2019},
                                               {30,  5, 2019},
                                               {10,  6, 2019},
                                               { 1,  8, 2019},
                                               {16,  9, 2019},
                                               {25, 12, 2019}
                                             };
```

Initialisation par désignation

- L'initialisation dans l'ordre oblige le programmeur à se souvenir de l'ordre exact des membres, et rend difficile un changement ultérieur de la structure (exemple : l'ajout d'un membre)
- **Solution** : indiquer spécifiquement quel membre prend quelle valeur (*designated initialization*)
  - on indique le nom du membre précédé d'un point (***designator***)

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};

struct Date myBirthday = {.day = 1, .year = 2000, .month = 1};
```

Affectation

- On peut affecter (avec copie) le contenu d'une instance d'une structure à une autre instance du même modèle de structure
  - exemple : `myBirthday = yourBirthday;`
  - copie les valeurs de tous les membres de `yourBirthday` vers ceux de `myBirthday`

- **N.B.** Cela offre une façon de copier un tableau vers un autre, en l'incluant comme membre d'une structure

```
struct {
  int tab[10];
} tab1, tab2;

/* .. plus tard .. */
tab1 = tab2;
```

L'opérateur point (.)

- On peut accéder aux valeurs des membres d'une instance d'une structure (plus court : « aux membres d'une structure ») en combinant le nom de l'instance et le nom du membre avec un point
  - p.ex., `myBirthday.year`
  - ce sont des *l-values*, donc on peut aussi les modifier

- Le point (.) est un opérateur C/C++ comme les autres
  - seulement, il est l'un des plus prioritaires
    - au niveau 1, à égalité avec `()`, `[]`, `++`, et `--`

Exemple : accéder à un membre

```c
struct Date {
    uint8_t  day;
    uint8_t  month;
    uint16_t year;
};

int main(void) {
    struct Date myBirthday = {.day = 1, .year = 2000, .month = 1};
    printf("My birthday day is the %d of %d", myBirthday.day, myBirthday.month);
    return EXIT_SUCCESS;
}
```

Exemple : modifier un membre

```c
struct Date {
    uint8_t  day;
    uint8_t  month;
    uint16_t year;
};

int main(void) {
    struct Date myBirthday = {.day = 1, .year = 2000, .month = 1};
    myBirthday.day = 3;
    myBirthday.year++;
    return EXIT_SUCCESS;
}
```

Cas des structures emboîtées

- Les membres (champs) d'une structure peuvent être eux-mêmes des structures
- On accède alors aux membres de façon hiérarchique, en utilisant plusieurs fois le point
- Exemple : une personne a une date de naissance

```
mary.birthday.day = 2;

persons[2] = john;

persons[3].birthday.day = 4;
```

```
struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};

struct Person {
   char name[MAX_SIZE];
   struct Date birthday;
} john, mary;

struct Person persons[10];
```

Comparer deux structures

- A part l'affectation, le langage C n'offre pas d'autre opération globale sur les structures
  - comme mentionné, on ne peut pas utiliser les opérateurs `==` et `!=` pour comparer deux structures
  - on ne peut pas non plus utiliser `memcmp`, car les champs ne sont pas toujours strictement contigus en mémoire (*voir plus loin*)
- Comment comparer deux instances d'une structure ?

```
if (myBirthday.day   == yourBirthday.day   &&
    myBirthday.month == yourBirthday.month &&
    myBirthday.year  == yourBirthday.year) ...
```

Les littéraux composés (*compound literal*)

- Un littéral composé permet de créer un « objet » sans nom, temporaire, à utiliser tout de suite
  - forme : nom du type (composé) + ensemble de valeurs entre accolades
    - disponible à partir de C99
- On peut par exemple affecter le littéral à une variable

```
struct Date day = (struct Date) {1, 1, 2010};
```

- Un littéral composé peut comporter des *designators*

```
(struct Date) {.month = 1, .day = 1, .year = 2010};
```

Usage des littéraux composés

- Les littéraux composés sont nécessaires pour les affectations à des `struct` (≠ *initialisation !*) et pour des objets sans nom

```
#include <stdio.h>        /* Début du programme exemple */
#include <stdlib.h>
#include <stdint.h>

struct Date {
   uint8_t  day;
   uint8_t  month;
   uint16_t year;
};

void display(const struct Date d) {
   printf("%04d.%02d.%02d\n", d.year, d.month, d.day);
}
```

Usage des littéraux composés

```
/* Suite et fin du programme exemple */

int main(void) {

   struct Date d1 = {25, 12, 2021};
   display(d1);

   /* display({25, 12, 2021}); est incorrect */
   display((struct Date){25, 12, 2021});

   struct Date d2;
   /* (struct Date) est obligatoire pour une affectation */
   d2 = (struct Date){25, 12, 2021};
   /* ou (struct Date){.day = 25, .month = 12, .year = 2021} */
   display(d2);

   return EXIT_SUCCESS;
}
```

```
2021.12.25
2021.12.25
2021.12.25
```

3. Structures, fonctions et pointeurs

Structures et fonctions

- Les structures s'utilisent comme les autres variables en ce qui concerne les fonctions. Elles peuvent apparaître comme arguments (paramètres), comme variables locales, ou comme valeurs de retour

```
struct Date mostRecentDate(struct Date d1,
                           struct Date d2) {
   if (d1.year != d2.year) {
      return d1.year > d2.year ? d1 : d2;
   }
   if (d1.month != d2.month) {
      return d1.month > d2.month ? d1 : d2;
   }
   return d1.day > d2.day ? d1 : d2;
}
```

Passage d'une structure par adresse

- On peut utiliser des pointeurs pour éviter de copier une structure lorsqu'on la passe en argument à une fonction

- **Exemple** : pour afficher une date, on peut passer seulement l'adresse de la structure (ci-dessous le pointeur `d`)

```
void printDate(const struct Date* d) {
   printf("%d/%d/%d\n", (*d).day, (*d).month, (*d).year);
}
```

```
printDate(&uneDate);
```

- **Important** : le point (.) est plus prioritaire que le déréférencement (*), d'où la nécessité des parenthèses

https://en.cppreference.com/w/c/language/operator_precedence

Exemple : remplissage des champs

- Sur le modèle de `scanf`, qui requiert une adresse, on peut demander à l'utilisateur une date
  - **solution** : passer à `scanf` l'adresse de chaque membre de l'instance `d` de la structure `Date` ('d' étant aussi passé à getBirthday par adresse)

```
bool getBirthday(struct Date* d) {
   printf("Enter your birthday day: ");
   bool ok = scanf("%d", &(*d).day);
   /* error checking here */
}
```

- **Important** : vu la priorité très élevée de (.), l'expression `&((*d).day)` est équivalente à `&(*d).day`

https://en.cppreference.com/w/c/language/operator_precedence

Opérateur « flèche » : `->`

- La forme `(*ptr).membre` est très fréquente
  - on passe souvent des pointeurs sur les structures, on doit les déréférencer, puis indiquer le membre
  - il y a un potentiel d'erreur si on oublie les parenthèses, car le point (.) est plus prioritaire que l'étoile (*)
- La flèche permet d'écrire *la même chose* plus simplement
- Au lieu de

```
printf("%d/%d/%d\n", (*d).day, (*d).month, (*d).year);
```

- On écrit plus simplement et sans risques

```
printf("%d/%d/%d\n", d->day, d->month, d->year);
```

Place en mémoire et alignement

- Les membres d'une structure (instance) occupent des places consécutives dans la mémoire … ou presque !
  - **toujours vrai** :
    (1) les champs sont placés dans l'ordre de leur déclaration
    (2) l'adresse de l'instance est celle du 1er champ
- Les compilateurs sont autorisés à aligner les places mémoire sur la taille des « mots » du processeur (typiquement 32 ou 64 bits, donc 4 ou 8 octets) pour lire les données en une fois
- Si un membre est d'une taille inférieure au mot, alors des octets de remplissage (***padding***) pourront être ajoutés, pour aligner le champ suivant sur le début d'un mot, si ce champ est plus long qu'un mot
- **Conséquences**
  - les calculs de pointeurs ne sont pas exacts
  - utiliser `sizeof(nomStruct)` dans le doute
  - on ne peut pas comparer deux structures avec `memcmp`

Exemple

```
struct S1 {
  double d;
  int    i;
  char tab[3];
  char   c;
};
```

```
struct S2 {
  char   c;
  double d;
  int    i;
  char tab[3];
};
```

```
struct S3 {
  char   c;
  char __pad1[7];
  double d;
  int    i
  char tab[3];
  char __pad2;
};
```

- Les structures **S1 et S2 ont le même contenu**, mais dans un ordre différent : auront-elles la même taille ?

```
printf("%zu %zu\n", sizeof(struct S1), sizeof(struct S2));
```

16 24

- Le compilateur (ici pour 64 bits) aligne 'd' sur deux mots de 4 octets (et aussi 'i' sur un mot)
  - **S2 prendra la même place que S3**
  - option `-Wpadded` de `gcc` : *warning* si le *padding* est nécessaire

Source : https://fr.wikipedia.org/wiki/Alignement_en_mémoire

Exemple (suite)

double d

int i

char tab[3]

char c

padding

struct S1

16 octets

struct S2, struct S3

24 octets

```
struct S1 {
  double d;
  int i;
  char tab[3];
  char c;
};
```

```
struct S2 {
  char c;
  double d;
  int i;
  char tab[3];
};
```

```
struct S3 {
  char c;
  char __pad1[7];
  double d;
  int i
  char tab[3];
  char __pad2;
};
```

4. Les unions

Définition de l'union

- L'union ressemble à la structure :
  il s'agit d'un type défini par l'utilisateur,
  qui regroupe plusieurs champs (ou membres)

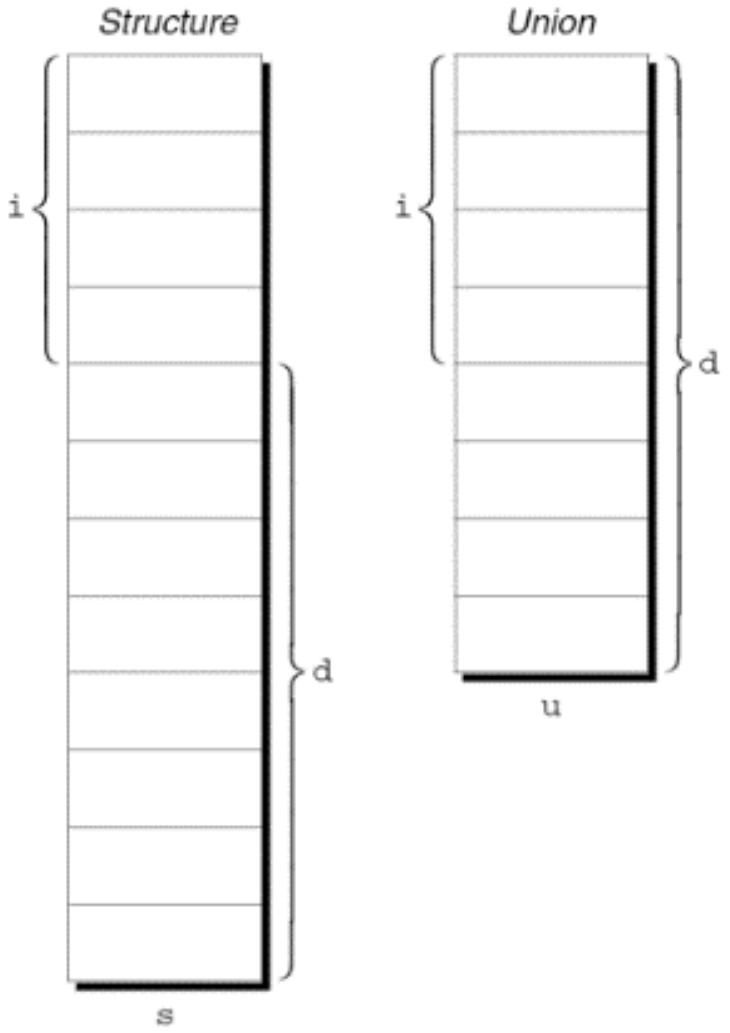
```
union [Name] {
   memberDecl;
   [memberDecl; ... ]
} [variableId, ... ];
```

- **Différence principale** : pour l'union, ces champs sont superposés en mémoire (ils commencent tous à la même adresse), donc un seul est disponible à la fois
  - on doit faire attention quel champ on veut utiliser
  - la taille mémoire est celle du champ le plus long

- Note : « union » ≈ « ou » (champ1 *ou* champ2 *ou* …),
  par opposition à l'intersection (champ1 *et* champ2 *et* …)

Représentation en mémoire

- **La structure s et l'union u diffèrent fondamentalement** : les champs de `s` sont stockés à des adresses *différentes*, alors que ceux de u sont tous stockés à la *même* adresse
- Changer un champ d'une union change la valeur dans l'espace mémoire commun utilisé !!
  - ici, stocker une valeur dans `u.d` fait perdre la valeur stockée dans `u.i`
- L'accès aux membres est semblable, p.ex. `s.i = 82;` ou `u.d = 74.8;`



Affichage des tailles (avec *padding*)

```c
union {       /* union anonyme */
  int    i;
  double d;
} u;

struct {      /* structure anonyme */
  int    i;
  double d;
} s;

int main(void){
  printf("sizeof(u) = %d\n", (int) sizeof(u));
  printf("sizeof(s) = %d\n", (int) sizeof(s));
  return EXIT_SUCCESS;
}
```

```
sizeof(u) = 8
sizeof(s) = 16
```

N.B. On ne peut pas définir une `struct` et une `union` de même nom dans la même portée

Initialisation d'une union

- Sans désignation d'un membre : on initialise le premier
  - avant C99, la valeur devait être une constante

```
union {
  int    i;
  double d;
} u = {0};
```

- Avec désignation d'un membre : on peut initialiser n'importe quel membre (champ)

```
union {
  int    i;
  double d;
} u = {.d = 10.0};
```

Quand utiliser les unions

1. Pour gagner de la place lorsqu'on veut stocker des variables dont on n'aura pas besoin en même temps
2. Pour créer des variables pouvant stocker plusieurs types simples à la fois (sorte de polymorphisme)
   - **souhait** : un tableau pouvant stocker aussi bien des entiers (`int`) que des réels (`double`)
   - **solution** : définir une union avec un champ par type, puis définir un tableau contenant de telles unions

```
union Number {
   int    i;
   double d;
};

union Number numbers[N];
numbers[0].i = 0;
numbers[1].d = 1.0;
/* etc. */
```

Dépasser la limitation des unions

- Comment savoir quel est le champ valide d'une union, celui qui a changé en dernier ?
- **Solution** : utiliser un **champ discriminant** (*tag*), en général au sein d'une structure englobante
- **Attention** : il faudra gérer correctement l'indication !

```
#define INT_KIND    0
#define DOUBLE_KIND 1

struct Number {
  int kind; /* tag */
  union {
    int    i;
    double d;
  } u;
};

void print(struct Number n) {
   if (n.kind == INT_KIND)
      printf("%d", n.u.i);
   else
      printf("%g", n.u.d);
}
```

5. Les énumérations

`enum`, `struct` et `union`

- Les énumérations (*enum*) ont été étudiées dans le cours PRG1, chap. 3. Nous y renvoyons le lecteur.
- **Attention** : Les énumérations fortement typées (*enum class*) ne sont pas utilisables en C !
- ***Pourquoi en parler ici ?***
  - ressemblance syntaxique avec `struct` et `union`
  - utiles pour les tags des unions
  - apparaissent souvent dans `typedef` (voir plus loin)

Type énuméré pour le tag d'une union

- Il s'avère souvent judicieux de définir le champ discriminant (*tag*) d'un type `struct` comme étant de type `enum`

```
struct Number {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int    i;
    double d;
  } u;
};
```

- **Avantages** (par rapport à la version du slide 41)
  - on évite les `#define`
  - le type `Number` est autonome
  - on garde la même fonction d'affichage

Portée d'un type enum

- Un type `enum` déclaré dans un type `struct` est accessible (peut être utilisé) en dehors du type `struct`.
- Ceci n'est pas le cas d'un type `enum` déclaré dans une fonction

```
struct S {
    enum AB {A = 1, B = 2} champ1; // enum visible à l'extérieur
    double champ2;
};

void f() {
    enum CD {C = 3, D = 4}; // enum pas visible à l'extérieur
    ...
}

int main(void) {
    enum AB ab = A; // Correct
    enum CD cd = C; // Ne compile pas !
    return EXIT_SUCCESS;
}
```

6. typedef

Syntaxe et utilisation de `typedef`

- Mot-clé permettant de **définir des synonymes d'un type**

```
typedef existing_type_name new_type_name;
```

- Pratique pour donner un nom à des `struct`, `union` ou `enum`
  - mais pas seulement : aussi à des types simples,
    pour clarifier ou simplifier l'écriture d'un programme

- Un synonyme peut servir à spécifier de nouveaux types

```
typedef int vecteur[3];  puis  vecteur* ptr;
```

- Des synonymes d'un même type désignent un seul type
  - on peut p.ex. affecter des valeurs entre des structures

Syntaxe et utilisation de `typedef`

- On peut utiliser un synonyme dans `sizeof`, `cast` et les déclarations de fonctions
- On peut définir des synonymes pour des types fonctions
  (mais il vaut mieux ne pas nommer les arguments)

```
typedef void f_type(char*, double);
```

- Une marque de signe (par ex `unsigned`)ne peut pas être ajoutée lors de la déclaration d'objets

```
typedef int ENTIER;
unsigned ENTIER n; // Interdit !
```

Syntaxe et utilisation de typedef

- Les attributs const (et volatile) peuvent s'appliquer, mais pas les classes de mémorisation :

```
typedef const  int ENTIER; // OK
typedef static int ENTIER; // Interdit !
```

- Le cas de `const` est complexe. Attention à son utilisation avec les pointeurs.

```
typedef int* PTR;
typedef const int* CPTR;

const PTR ptr1 = …;  // /!\ ptr1 de type int* const
CPTR ptr2 = …;       // ptr2 de type const int*
const CPTR ptr3 = …; // ptr3 de type const int* const
```

typedef avec struct et union

- Au lieu de :

```
struct Number {
   int    i;
   double d;
};

struct Number n;
```

on peut écrire :

```
typedef struct {
   int    i;
   double d;
} Number;

Number n;
```

- **Avantages**
  - améliore la lisibilité : « `Number` » s'utilise comme un type simple
  - même syntaxe en C et C++ (« `struct` » optionnel en C++)
- **Inconvénients**
  - cache la vraie nature du type d'origine
  - monopolise un nom global de plus
- *Tout ça pour éviter d'écrire '`struct`' ? Le débat reste ouvert…*

Définition et `typedef` simultanés

- On peut écrire en répétant le nom

```
struct Date {...};

typedef struct Date Date;
```

- Ou de manière équivalente

```
typedef struct Date {...} Date;
```

- Possible car C gère des *namespaces* différents pour les `struct` et `union` par rapport à ceux des autres identificateurs comme ceux dans `typedef`
  - **on utilise en général une `struct` anonyme avec `typedef`**

7. Résumé

Résumé

- Notions valables en C et en C++, mais les classes C++ offrent beaucoup plus de possibilités
- `struct` : utile pour rassembler des informations hétérogènes sous un même identificateur
- `union` : utile pour économiser de la mémoire et pour modéliser des structures avec parties variantes
- `enum` : notion de bas niveau; bonne façon de contraindre les valeurs d'une variable (si peu nombreuses); permet de regrouper dans un même ensemble des constantes logiquement reliées
- `typedef` : notion très générale; peut être utile pour la lisibilité et la portabilité d'un programme; en C++, on utilise de préférence using.