Plan

Structuration de données

Les données manipulées jusqu'à présent ont un type scalaire, ce sont des données élémentaires, indécomposables :

- une constante entière : `23`
- une variable entière : `int a;`
- un pointeur d'entier : `int *p;`

Différents types composés sont proposés en C :

- les **structures** : une suite finie d'éléments de types quelconques. Ils ont une variante binaire, les **champs de bits** ;
- les **tableaux** : une suite finie d'éléments de même type ;
- les **énumérations** (le type somme simple du C) : un type défini par un ensemble fini de valeurs possibles.
- les **unions** (le type somme complexe du C) : un élément d'un type choisi parmi un ensemble fini de types possibles ;

Plan

Intérêt des structures

Une structure permet de regrouper des éléments de types quelconques pour les manipuler comme un seul élément cohérent qui souvent correspond à un concept du problème informatisé.

Exemples :

- un complexe regroupant une partie réelle et une partie imaginaire
- un compte bancaire regroupant un identifiant de banque, un numéro de compte, et un solde.
- une carte à jouer regroupant une couleur et une valeur
- une date regroupant un jour, un mois et une année.

Cela allège la manipulation des données et fonctions en évitant la multiplication des variables et paramètres.

- On transmet une date au lieu d'un jour, un mois et une année.

Spécifications

Une structure est composée d'un nombre fixé de membres (127 au maximum) :

- nommés (`banque, numero, solde`)
- typés (`int` pour `banque, numero`, `float` pour `solde`)

Une variable structurée peut être manipulée :

- membre par membre (lecture, modification)
- globalement (initialisation, copie, paramètre fonction)

Définition d'une structure en C

Syntaxe

```
struct nomStruct {
    type1 membre1;
    type2 membre2;
    ...
};
```

Cela définit un nouveau type de données `struct nomStruct`

- de nom `nomStruct` qui est un identificateur avec les mêmes règles de nommage et la même portée que les variables.
- composé des membres de nom `membre1, membre2, ...` (des identificateurs).
- ayant respectivement pour type `type1, type2, ...` qui doivent être des types déjà définis (scalaires, structures précédemment définies, ...)
- le `;` est obligatoire à la fin de la définition.

Déclaration de variables structurées

La déclaration d'une variable (ou d'un paramètre) structurée doit intervenir après la définition de la structure (règles de portée) :

Syntaxe

```
struct nomStruct var;
```

La déclaration peut être accompagnée d'une initialisation des différents membres :

Syntaxe

struct nomStruct var = {exp1, exp2, ...} ;

- Le $membre_i$ reçoit comme valeur initiale, la valeur de exp_i :
  - exp_i doit être du même type que $membre_i$ (sinon conversion) ;
  - s'il y a moins d'expression que de membres, les derniers membres sont initialisés à 0 (quelque soit leur type) ;
  - s'il y en a plus, les dernières sont ignorées.

Manipulation de variables structurées

Pour accéder aux membres, on utilise la **notation pointée** :

Syntaxe

```
var.membre
```

- var est une variable structurée ;
- membre est l'un de ces membres.

Cela permet de manipuler le membre `membre` comme une variable :

- possédant une adresse : `&(var.membre)`
- accessible en lecture/écriture : `var.membre = var.membre + ...`

Exemple

```c
#include <stdio.h>

struct compte {
   int banque;
   int numero;
   float solde;
};

int main() {
   struct compte cpt = {76,1234};
   cpt.solde = cpt.solde + 100;
   printf("compte %d-%d : solde=%f\n",
            cpt.banque,cpt.numero,cpt.solde);
   return 0;
}
```

Exécution :

```
compte 76-1234 : solde=100.000000
```

Manipulation globale des structures : affectation

Si les différents membres d'une structure peuvent être lus et modifiés comme des variables, la variable structurée peut être **globalement copiée**, **transmise** lors d'un appel de fonction, ou **résultat** d'une fonction.

- la copie est faite champ par champ.

Exemple

Soit `cpt1`, `cpt2` deux variables de type `struct compte` :

```
cpt1 = cpt2;
```

est équivalent à

```
cpt1.banque = cpt2.banque;
cpt1.numero = cpt2.numero;
cpt1.solde = cpt2.solde;
```

Structures comme paramètre/résultat de fonction

```c
#include <stdio.h>

struct compte { int banque;  int numero;  float solde; };

struct compte creerComptePromo(int banque, int num) {
    struct compte cpt = {banque, num};
    cpt.solde = 800;
    return cpt;
}

void afficheRIB(struct compte cpt) {
    printf("RIB : |%d|%d|\n",cpt.banque, cpt.numero);
}

void verseInterets(struct compte *cpt, float taux) {
    (*cpt).solde = (*cpt).solde*(1.0+taux);
}

int main() {
  struct compte monCpt;
  monCpt = creerComptePromo(12, 3456);
  afficheRIB(monCpt);
  printf("solde avant interets : %f\n", monCpt.solde);
  verseInterets(&monCpt, 0.01);
  printf("solde apres interets : %f \n", monCpt.solde);
  return 0;
}
```

Simplification d'accès aux membres via pointeur

Attention

Le . est prioritaire sur l'* :
*var.membre correspond à *(var.membre) et non à (*var).membre

Pourtant il est courant de vouloir accéder aux membres d'une structure dont on connaît l'adresse :

```
void verseInterets(struct compte *cpt, float taux) {
    (*cpt).solde = (*cpt).solde * (1.0+taux);
}
```

Le langage C propose un opérateur spécial : ->

- var->membre est strictement équivalent à (*var).membre

```
void verseInterets(struct compte *cpt, float taux) {
    cpt->solde = cpt->solde * (1.0+taux);
}
```

Définition de type

On peut associer un **alias** à n'importe quel type. Cela permet :

- de manipuler les types avec des noms plus proches du type d'information qu'ils représentent ;
- d'éviter de trimbaler le mot clé `struct` lorqu'on manipule des structures.

Syntaxe

typedef type alias ;

- alias est un identificateur
- type est un type

Dès lors alias peut-être utilisé à la place de type dans la suite (règles de portée).

Définition de type : Exemples

```c
typedef int entier;
typedef entier * pEntier;

entier inc(pEntier pe) {
    entier res = *pe;
    *pe = *pe + 1;
    return res;
}

int main() {
    int a=2;
    printf("%d\n",inc(&a));
    printf("%d\n",a);
    return 0;
}
```

Définition de type : Exemples

```c
typedef int entier;
typedef entier * pEntier;

entier inc(pEntier pe) {
    entier res = *pe;
    *pe = *pe + 1;
    return res;
}

int main() {
    int a=2;
    printf("%d\n",inc(&a));
    printf("%d\n",a);
    return 0;
}
```

```c
struct compte { int banque;  int numero;  float solde; };

typedef struct compte tCompte;

tcompte creerComptePromo(int banque, int num) {
    tcompte cpt = {banque, num};
    cpt.solde = 800;
    return cpt;
}
```

Structures imbriquées

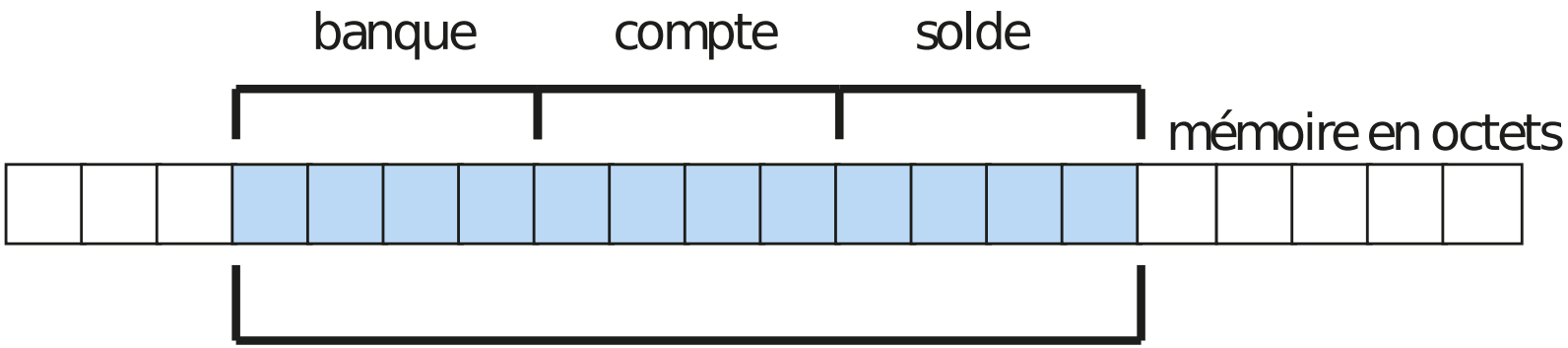
Les membres d'une structure peuvent être des structures

Syntaxe

```
struct nomStruct{
    struct autreStruct membre;
    ...
};
```

- Le type `struct autreStruct` doit être précédemment défini ;
- L'affectation, l'initialisation, le passage en argument et le retour de fonction d'une structure **copie récursivement** chacun des membres.

Stockage des structures en mémoire



Remarque

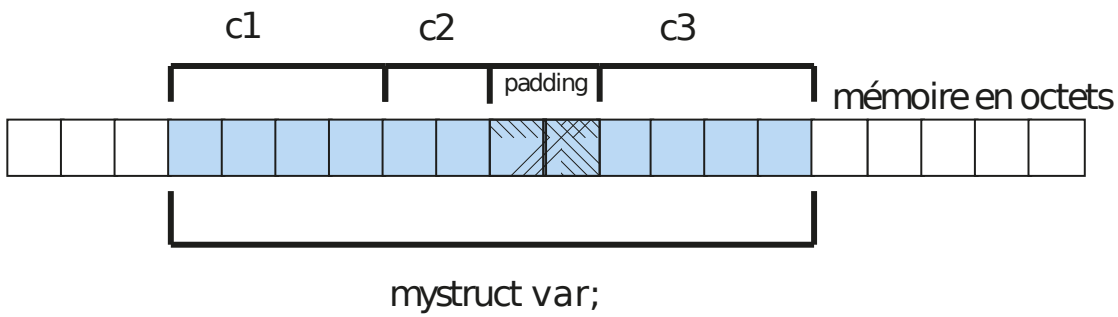
Les membres sont ordonnés en mémoire dans l'ordre de leur déclaration dans la structure.

Stockage des structures en mémoire

Attention

Les membres des structures sont alignés suivant l'alignement des types de base en mémoire (32 ou 64 bits).

```
struct mystruct {
   int c1;
   short int c2;
   float c3;
};
```



- on peut obtenir l'occupation mémoire en utilisant `sizeof`
  exemple : `sizeof(struct mystruct)`
- les règles de padding sont complexes (au delà de ce cours)