Structures

Les objets

Regroupement de variables de types différents

Pour modéliser un objet, on utilise donc un ensemble de variables dans lesquelles seront stockées les propriétés ou attributs qu'on lui associe.

Le fait de modéliser ou de choisir les attributs pertinents pour représenter un objet consiste en ce qu'on appelle un travail d'analyse ou de spécifications : c'est un autre aspect de l'informatique que vous aurez l'occasion d'aborder plus tard.

Une structure est donc un type composé de plusieurs variables de type différents.

Nous avons déjà abordé cette notion de regroupement lors du traitement des tableaux. Mais à la différence des tableaux, puisque les types sont différents, on ne peut pas aussi simplement définir de variables de type composé !

Il faut définir ce type en listant les informations qu'il contient, puisqu'elles ne sont plus toutes du même type : un simple indice entier n'est pas suffisant pour retrouver une information particulière.

Le travail de modélisation

Créer un nouveau type procède d'un travail de modélisation, qui interviendra plus tard dans votre cursus, car on ne peut faire de la modélisation que si l'on maîtrise bien tous les aspects techniques qui interviendront lors de la création des algorithmes et des programmes. Il est à cet égard particulièrement important de connaître les capacités et les limites d'un ordinateur ou d'un langage.

Pour ce qui concerne, ce travail de modélisation sera très simple, et la plupart du temps, vous seront fournies les informations à regrouper pour créer ce nouveau type. Votre tâche principale est de bien savoir les manipuler, que ce soit en langage algorithmique ou en langage C, et, par la suite, dans d'autres langages.

En effet, les aspects les plus récents abordés lors des derniers cours (les fonctions et les pointeurs notamment) sont omniprésentes en informatique, et leur bonne maîtrise est indispensable.

Quelques exemples d'objets et de modélisation

Les énoncés des projets informatiques d'entreprise, ou plus prosaïquement les énoncés de TDs ou TPs de l'école, feront souvent référence à des objets à modéliser.

Un objet, en informatique, est une représentation ou un modèle (d'où le terme de modélisation) d'un objet, au sens large, de la vie réelle, par exemple une voiture, une personne, un CD, un film. Ces 'objets' réels sont très complexes, et on pourrait passer énormément de temps à lister leurs propriétés sans en faire complètement le tour. Pour se limiter à une liste 'raisonnable' (de 5 à 30 propriétés), il faudra bien entendu tenir compte des aspects que l'on veut manipuler grâce à un programme informatique : la liste des propriétés à stocker dépend de l'application envisagée et du domaine traité !

Définition d'un type composé

Pour définir un type composé, il faut en premier lieu lui choisir un nom, puis dresser la liste de toutes les propriétés, également appelées champs, que l'on souhaite stocker à l'intérieur de ce type. Chacun des champs est identifié par un nom, ce qui permet au programmeur de le choisir parmi ceux qui sont stockés dans le type composée, et d'un type, pour que l'ordinateur sache le manipuler.

Par convention de nommage, un nom de type composé doit systématiquement commencer par '`t_`'.

Pour définir un type composé, aussi appelé structure, on utilise simplement la syntaxe suivante :

```
structure nom_du_type
{
      type_champ1           nom_champ1;
      …
      type champ_n          nom_champ_n;
};
```

quelques exemples illustratifs :

Définition du type composé 'voiture' pour représenter l'objet voiture dans le cadre d'une concession de voitures d'occasion :

On choisit de représenter cet objet par les caractéristiques ou propriétés suivantes :

- Une marque
- Un modèle
- Une cylindrée
- Un millésime (année de mise en circulation)
- Un kilométrage

- Un état

Cela nous fournit donc la liste des champs, il reste maintenant à choisir les types de chacun de ces champs.

```
structure t_voiture
{
      caractere marque[10];
      caractere *modele;
      reel cyl;
      entier milles;
      entier kilom;
      entier etat;
};
```

remarques sur les choix des types pour les différents champs : lorsque la liste des différents champs est établie, on leur associe un type afin de pouvoir les utiliser en algorithmique.

Marque : donnée par un texte : une chaîne de caractères, on choisit un tableau statique car les noms des marques sont, dans l'ensemble, relativement courts : "Renault", "Peugeot", "Ford", "Toyota", etc…

Modèle : donné également par un texte, que l'on choisit de stocker plutôt sous la forme d'un tableau dynamique, car les longueurs des noms des modèles sont par contre très variables : "Clio Williams", "307 SW", "Fiesta TDI", "Polo Match série limitée", etc…

La cylindrée est une valeur à virgule

Le millésime, le kilométrage et l'état sont stockés par des entiers.

Définition du type 'complexe' pour représenter un nombre complexe:

```
structure t_complexe
{
      reel re, im;
};

ou

structure t_complexe
{
      reel module;
      reel argument;
};
```

Il existe deux possibilités, selon que l'on choisit de représenter un nombre complexe sous forme : partie réelle, partie imaginaire, ou sous forme polaire, en stockant les valeurs : module, argument.

On remarque également, avec le premier choix (partie réelle, partie imaginaire), que plusieurs champs de même type peuvent être définis par une liste :

```
type_champ nom_champ1, nom_champ2,…,nom_champ_n;
```

Cette définition de champ est similaire à une liste de définitions de variables qui ont le même type.

A retenir : le mot structure permet de définir un nouveau type, et non pas une variable

*Char * : le type "fourre-tout"*

Définir des variables d'un type composé

Une fois ce type composé défini, il est utilisable dans un programme presque au même titre que les types de base de l'algorithmique, qui sont entier, reel et caractere. Il est notamment possible de définir des variables dont le type est un type composé ou structure, en utilisant la syntaxe classique de définition d'une variable : `type nom_de_variable ;`

Exemples avec les types définis précédemment :

`t_complexe var_comp;` est une définition de variable tout à fait correcte, et dont la signification est : la variable nommée `var_comp` est de type `t_complexe`.

`t_voiture my_car;` est également une définition de variable correcte.

Il reste maintenant à détailler la manière dont ces variables peuvent être manipulées. En effet, puisque les structures ne sont pas des types de base du langage, l'ordinateur ne sait pas les manipuler simplement. Réaliser un affichage des variables qui sont définies ci-dessus est déjà une tâche trop compliquée pour un ordinateur, car il ne sait afficher que des entiers, des réels ou des caractère, éventuellement du texte, mais il n'est pas capable de mieux.

Par contre, les champs situés dans la structure sont le plus souvent des champs dont les types sont connus et manipulables, même si cela n'est pas une obligation (nous aurons l'occasion de le voir plus tard). Un champ peut en effet être de n'importe quel type, y compris un type défini par une structure.

A partir du nom de la variable dont le type est une structure (comme les variables `my_car` ou `var_comp` définies ci-dessus), on peut accéder aux différents champs qui la constituent.

Accès aux champs

Continuons avec les exemples précédents : le but, dans un premier temps, est de manipuler les champs des structures définies.

Pour initialiser une variable de type t_complexe, on ne peut malheureusement pas écrire :

```
t_complexe z ;
z ← 2.25 + 3.4i ;     // initialisation avec un nombre complexe écrit
                      // sous forme standard
```

, car l'écriture 3.4i ne correspond pas à quelque chose de connu pour l'ordinateur : il va essayer de traduire cela en quelque chose de compréhensible pour lui, et ce n'est ni une valeur numérique (à cause du i), ni un nom de variable, car il commence par un chiffre. De plus, l'opérateur + sert uniquement à faire des additions de nombres ! On ne peut donc initialiser cette variable qu'en indiquant les valeurs à stocker dans ses champs, il existe donc un moyen d'accéder aux champs d'une variable dont le type est un type défini par l'utilisateur.

Le '.'

L'opérateur utilisé pour accéder à un champ à partir du nom d'une variable est le '.' : sa syntaxe d'utilisation est la suivante :

```
nom_de_variable.nom_du_champ
```

Cela s'interprète comme : le champ `nom_du_champ` de la variable `nom_de_variable`. Cette écriture est une expression dont le type est celui du champ référencé.

Quelques exemples :

```
structure toto
{
      entier une_valeur;
      caractere c;
      reel x;
      reel y;
};
```

Dans un programme, définissons une variable de ce type :

```
toto ma_var;

// liste d'instructions avec quelques commentaires

ma_var ← 5;     // FAUX, car ma_var n'est pas de type entier, elle
                // est de type toto
ma_var.une_valeur ← -6; // CORRECT, car ma_var.une_valeur est une
// expression qui signifie : le champ 'une_valeur' de la variable
// ma_var, or ce champ est de type entier

toto.y ← 3.17; // FAUX, car le . doit s'appliquer à un nom de
                // variable, et non au nom du type.
afficher(ma_var);     // FAUX? Car afficher() ne s'applique qu'aux
                      // types de base de l'algorithmique : entier,
                      // caractere, reel.
afficher(ma_var.c);   // CORRECT, car ma_var.c est un champ de type
                      // caractere
ma_var.x ← 3.1 * ma_var.y;// CORRECT, car ma_var.x et ma_var.y sont
                          // deux champs dont le type est reel
```

Une confusion fréquente consiste à faire précéder le '.' du nom du type défini à l'aide de la structure.

Autre exemple, avec des champs dont les types sont un peu plus sophistiqués :

```
structure tablo
{
     entier valeurs[20];
     entier util;
}
```

Cette structure est très intéressante, car elle permet d'associer au sein d'un même type deux informations qui doivent systématiquement être définies ensemble. En effet, lorsque l'on souhaite utiliser un tableau, qu'il soit statique ou dynamique, on doit toujours connaître sa taille utile. Jusqu'à présent, puisque les structures nous étaient inconnues, nous avons toujours du définir, en association avec un tableau, une variable entière pour stocker sa taille utile :cela peut mener à des oublis, surtout si le programme comporte un certain nombre de tableaux; et ce d'autant plus que l'usage systématique de la taille utile n'est pas encore passé au stade de réflexe…

Soit le programme suivant :

```
tablo mon_tab;  // définition de variable
entier cpt;     // compteur pour les boucles futures
```

```
// mon_tab.valeurs est un tableau d'entiers, on peut donc lui
// appliquer l'écriture [indice] pour accéder à un des éléments qui
// sont stockés dans ce tableau.

mon_tab.valeurs[0] ← 6; // affectation de la première case
mon_tab.util =1;      // mise à jour de la taille utile

// on suppose que l'on entre d'autres valeurs dans le tableau et que
// met à jour la taille utile en conséquence.

// affichage des valeurs :

pour cpt de 0 à mon_tab.util-1
{
     afficher(mon_tab.valeurs[cpt],"\n");
}
```

Rappel : l'écriture `nom_de_variable.nom_du_champ` est une expression dont le type est celui du champ référencé : on peut donc lui appliquer tous les opérateurs et toutes les fonctions permettant de manipuler ce type.

Autre exemple :

```
structure texte
{
     caractere mots[100];
     entier util;
}

programme utilise_fonctions_pour_le_texte

texte blabla;

lire(blabla.mots); // blabla.mots : le champ mots de la variable
// blabla : cette expression est de type : tableau de caractères, on
// peut donc lui appliquer les fonctions de gestion de texte.

blabla.util ← longueur_texte(blable.mots); // correct également
copier(blabla.mots,"coucou !"); // correct

// calcul de la taille du texte par un autre moyen

blabla.util ← 0;

tant que (blabla.mots[blabla.util] ≠ EOT)
{
     blabla.util ← blabla.util+1;
}
```

L'opérateur d'affectation

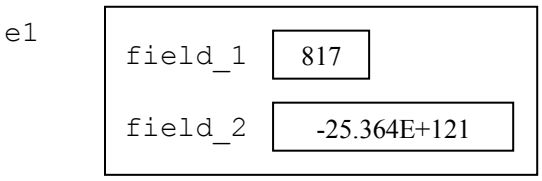
C'est le seul opérateur que l'ordinateur peut appliquer aux structures, car il s'agit dans ce cas de recopier les champs d'une variable de type composé dans une autre variable du même type. C'est d'ailleurs tout ce que fait cette opération.

Une instruction d'affectation entre deux variables d'un type composé copie les champs d'une variable dans une autre.

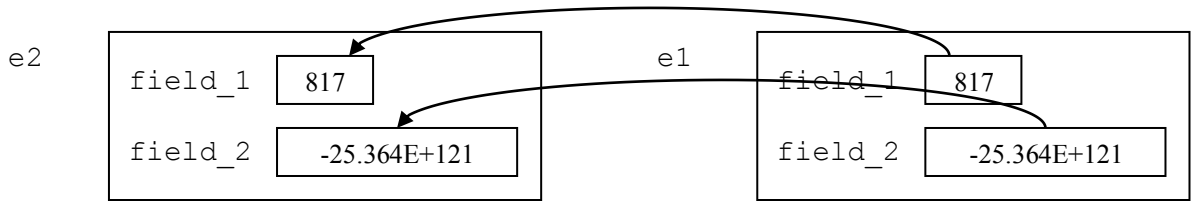
Il n'est d'ailleurs possible de l'utiliser qu'entre deux variables, il est notamment impossible d'affecter plusieurs constantes aux champs d'une variable par cette opération.

Exemples :

```
structure foo
{
      entier field_1;
      reel field_2;
};

programme affect

foo e1, e2;

e1 <- {1, 3.5}; // FAUX, pas d'initialisation avec une liste
e1.field_1 ← 817; // OK

e1.field_2 ← -25.364E+121; //OK
```




```
e2 ← e1; // OK , effets : recopie de la valeur du champ field_1 de
// e1 dans le champ field_1 de e2 et du champ field_2 de e1 dans le
// champ field_2 de e2.
```



Autres opérateurs : structures et programmation modulaire

A part l'opérateur d'affectation, on ne peut utiliser avec les structures les opérateurs de base de l'arithmétique, ni les fonctions standard telles que `afficher()` ou `saisir()`, tout simplement parce que les structures sont des types trop complexes pour le langage informatique.

Il faut donc prévoir dans ce cas de créer un ensemble de fonctions qui remplaceront ces opérateurs et fonctions standard pour manipuler de façon claire ces nouveaux types. Ce point particulier sera traité à deux occasions : dans la section 'Structures et fonctions' de ce document, ainsi que l'année prochaine, lors du cours de C++, lorsque seront abordés les concepts de **méthodes** et de **surcharge des opérateurs**.

Structures contenant des tableaux et pointeurs

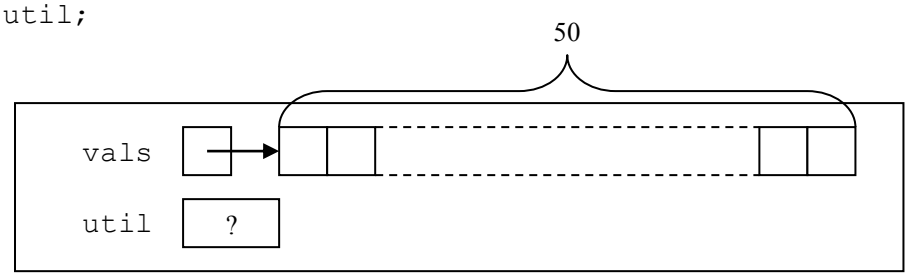
Tableaux statiques

Puisqu'un champ peut être de n'importe quel type, il peut s'agir entre autres d'un tableau, par exemple statique. On n'oubliera pas, dans ce cas, de stocker sa taille utile dans la structure.

Lorsque l'on définit plusieurs variables d'un tel type, un tableau statique différent (d'adresse différente) est créé pour chacune des variables. En cas d'affectation, ce sont les contenus des cases du tableau qui sont recopiés.

Exemple :

```
structure t_tab
{
      entier vals[50];
      entier util;
}
```




```
programme str_tablo

t_tab t1,t2;
entier cpt;

t1.util ← 10;
```

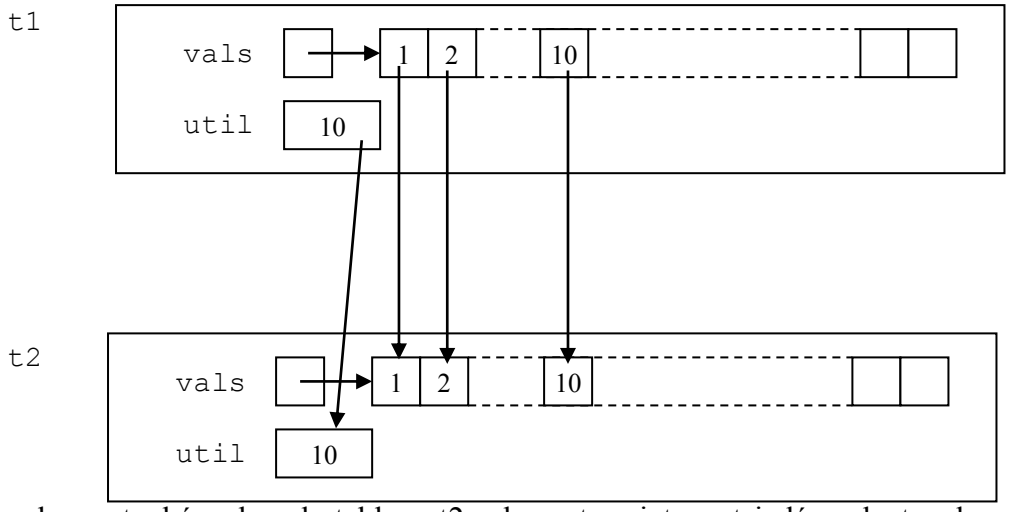
```
pour cpt de 0 à t1.util-1
{
      t1.vals[cpt] = cpt+1; // stocker les valeurs 1,2,3,4..,10
}

t2 ← t1;

// affichage des adresses pour vérification

afficher("adresse du tableau pour t1 : ",t1.vals,"\n");
afficher("adresse du tableau pour t2 : ",t2.vals,"\n");
```

```
adresse du tableau pour t1 : 2293380
adresse du tableau pour t2 : 2293172
```

Ceci est donc la seule exception à la règle d'affectation définie précédemment : lorsque l'on copie un champ qui est un tableau statique, ce sont les éléments du tableau qui sont recopiés, et non l'adresse stockée dans le tableau. En effet, un tableau statique est une adresse qui ne peut pas être modifiée.

```
pour cpt de 0 à t2.util-1
{
      afficher(t2.vals[",cpt,"] = ",t2.vals[cpt],"\n");
}
```

```
t2.vals[0] = 1
t2.vals[1] = 2
t2.vals[2] = 3
t2.vals[3] = 4
t2.vals[4] = 5
t2.vals[5] = 6
t2.vals[6] = 7
t2.vals[7] = 8
t2.vals[8] = 9
t2.vals[9] = 10
t2.vals[10] = 11
t2.vals[11] = 12
```



Les valeurs stockées dans le tableau t2.vals sont maintenant indépendantes de celles stockées dans le tableau t1.vals.

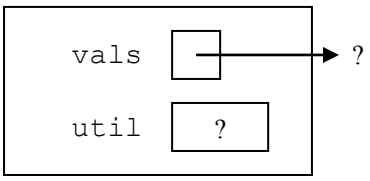
```
t1.vals[5] ← 0;
afficher(t2.vals[5]);
```

```
6
```

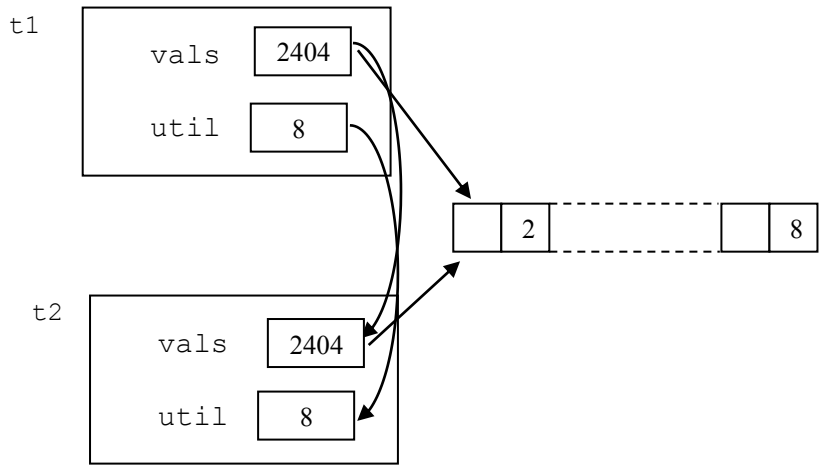
Tableaux dynamiques

Dans le cas d'un champ défini à l'aide d'un pointeur (tableau dynamique), si une affectation est faite entre deux structures, c'est la valeur du pointeur qui sera recopiée, comme pour les passages de paramètre à une fonction : le contenu sera donc commun, et cela peut poser certains problèmes pour la gestion des textes par exemple.

```
structure t_tab
{
      entier *vals;
      entier util;
}
```




```
entier longueur;
t_tab t1, t2;

// initialisation du champ vals de t1 qui est un pointeur

longueur ← 8;
t1.util ← longueur;
t1.vals ← reservation(longueur entier); // hypothèse : @ 2404

//    initialisation des valeurs du tableau dynamique avec
//    1,2,3,4,5,6,7,8

t2 ← t1;
```




```
t2.vals[4] ← 17;
```

Structure contenant des structures

Un champ d'une structure peut être de n'importe quel type, donc il peut être d'un type composé, c'est-à-dire une structure. Cela aide à bien hiérarchiser les différents niveaux pour éviter que les structures ne comportent trop de champs.

Il est possible dans ce cas d'avoir un accès à un champ en utilisant plusieurs fois de suite la notation '.'

Pour qu'un champ d'une structure `s1` soit lui-même une structure `s2`, il faut cependant que cette structure `s2` soit définie avant `s1`.

Prenons un exemple, avec la structure `personne`, pour laquelle on souhaite stocker le nom, le prénom et la date de naissance.

Pour le nom et le prénom, il est naturel de choisir le type tableau de caractères (tableau statique ou dynamique, peu importe). Mais qu'en est-il pour la date de naissance. A priori, aucun des types de base ne sont satisfaisants pour stocker cette information. Fort heureusement, le rôle des structures est de pallier au manque d'adéquation entre les types de base d'un langage informatique et les informations que l'on souhaite stocker. Créons donc une structure pour stocker cette information sur la date de naissance dans un premier temps :

exemple

```
structure date_naissance
{
      entier jj, mm, aa;
};
```

puis la structure `personne`

```
structure personne
{
      caractere        *nom;
      caractere        *prenom;
      date_naissance   dat_n;
};
```

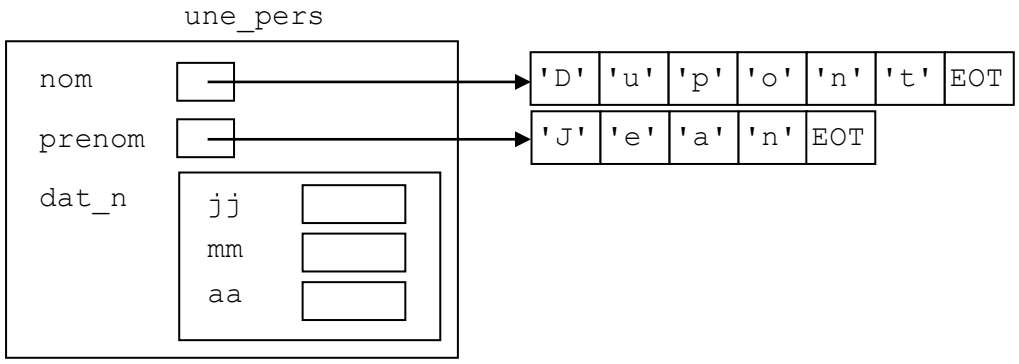
Puisqu'une structure définit un nouveau type, le type `date_naissance` est disponible pour être utilisé seul ou à l'intérieur d'une autre structure.

Accès aux sous-structures :

Un programme souhaite utiliser la structure personne définie ci-dessus pour stocker les informations suivantes : M. Jean DUPONT, né le 03 Septembre 1964.

```
fonction principale()
{
      personne une_pers;

      une_pers.nom ← reservation(6+1 caractere); // une place pour
EOT
      une_pers.prenom ← reservation(4+1 caractere); // idem

      copier(une_pers.nom, "Dupont");
      copier(une_pers.prenom, "Jean");

      // initialiser la date de naissance
```

Dressons le schéma de cette variable à ce moment du programme :



Afin d'initialiser la date de naissance, l'information à exploiter est toujours la même : le type de la variable (ou champ) à initialiser.

Le champ `dat_n` de la variable `une_pers` est de type `date_naissance`, qui n'est pas un type de base, il faut donc accéder aux champs de `dat_n` !

```
    une_pers.dat_n.jj ← 3;
    une_pers.dat_n.mm ← 9;
    une_pers.dat_n.aa ← 1964;

    retourner;
}
```

Il est donc nécessaire d'utiliser autant de fois la notation '.' qu'il y a de niveaux de structures pour traiter les champs qui ont des types de base.

Pointeurs vers les structures

Est-il possible d'utiliser des pointeurs pour stocker l'adresse d'une variable de type composé ? La réponse à cette question a des conséquences assez importantes, dans la mesure où, comme pour les exemples déjà traités dans les chapitres précédents, des applications (programmes professionnels utilisés dans l'entreprise) auront à utiliser beaucoup de structures.

Or, la possibilité de stocker en mémoire un certain nombre de structures implique que l'on puisse utiliser des tableaux, et le plus souvent, des tableaux dynamiques. En réalité, les tableaux statiques ne sont quasiment pas utilisés, seule la notation [] est vraiment confortable d'un point de vue de la programmation. Cela est tellement vrai que les langages objets comme C++, Java, C# et consorts l'utilisent beaucoup et étendent son utilisation pour d'autres cas que les simples tableaux !

L'utilisation de tableaux dynamiques nécessite bien entendu l'emploi de pointeurs.

D'autre part, le rôle des pointeurs est également de transmettre des adresses à des fonctions. Nous en avons vu l'utilité pour conserver un accès à un contenu entre fonction appelante et fonction appelée, mais nous aurons l'occasion de rencontrer un autre emploi des pointeurs pour passer des paramètres à une fonction dans le cas des structures.

La dernière phrase du paragraphe précédent lève en fait la question posée. Oui, on peut pointer vers une variable, et ceci quel que soit son type. Regardons en détail pourquoi cela est possible d'un point de vue informatique, d'autant plus que cette explication ne fait référence qu'à des notions connues.

Manipuler un contenu

Un petit retour en arrière vers les pointeurs est ici nécessaire.

Afin de pouvoir manipuler un contenu, l'ordinateur a besoin de deux informations :

- Une adresse lui permettant de déterminer l'endroit de la mémoire à consulter;
- Un type.

Cette information de type, indispensable à la définition d'un pointeur, est exploitée pour savoir combien d'octets doit manipuler l'ordinateur lors de l'accès à la mémoire. A titre

de rappel, une variable de type `caractère` occupe 1 octet, une variable de type `entier` ou pointeur occupe 4 octets, une variable de type `réel` occupe 8 octets.

Le point principal pour déterminer si une variable de type structure peut être pointée est de connaître la manière dont l'ordinateur stocke les champs de cette variable en mémoire. Il serait possible de le déterminer en écrivant un programme qui affiche les adresses des champs et le résultat en serait le suivant :

Les champs d'une telle variable sont stockés les uns à la suite des autres de manière contiguë dans la mémoire, cette variable forme un bloc en mémoire. Elle a donc une adresse (comme pour les tableaux, celle du début du bloc), ou plus précisément, une seule adresse permet de la repérer.

En ce qui concerne la taille de cette variable, le raisonnement est encore plus simple : la taille d'une variable de type structure est la somme de la taille de tous ses champs ! Dans le cas d'une structure contenant une autre structure, il suffit de re-détailler les champs. Il suffit donc de faire la somme des tailles de tous les champs qui sont des types de base.

Ayant une taille fixe (et calculable par la méthode exposée ci-dessus) et une adresse unique, une variable de type structure peut être manipulée comme un contenu et donc :

Il est possible de définir un pointeur vers une variable de type structure

La conclusion général de cette section est donc :

Il est possible de définir un pointeur vers une variable quel que soit son type.

Aspects syntaxiques :

Les notions d'adresse, de valeur, de contenu restent dans ce cas tout à fait valides, et les notations usuelles s'appliquent. Soit `t_com` un type composé (une structure). Définir un pointeur `ptr` vers un contenue de type `t_com` s'écrit donc naturellement :

```
t_com *p;
```

Initialiser ptr :

Les règles ne changent pas : pour initialiser un pointeur, il faut lui donner une adresse valide : celle d'une variable existante ou l'adresse d'une zone de mémoire obtenue par réservation.

Exemple :

```
structure toto
{
      entier ch1;
      reel  ch2;
};

fonction principale()
{
      toto *ptr_t;
      toto x;

      entier nb;

      // pour initialiser ptr_t, 2 solutions

      ptr_t ← &x; // ok car types compatibles

      afficher("taille du tableau contenant des toto:");
      saisir(nb);

      ptr_t ← reservation(nb toto); // ok, comme pour tout pointeur
}
```

Accès aux champs :

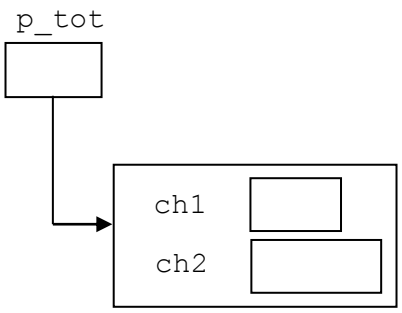
Comment accéder aux champs d'une variable de type structure lorsque l'on ne dispose que d'un pointeur vers cette variable et non de sa valeur ?

Il suffit dans ce cas de bien déterminer les types des expressions à écrire pour réaliser cet accès sans aucune ambiguïté :

Plaçons-nous dans le cas suivant, dans lequel la structure toto définie dans la section précédente va être réutilisée :

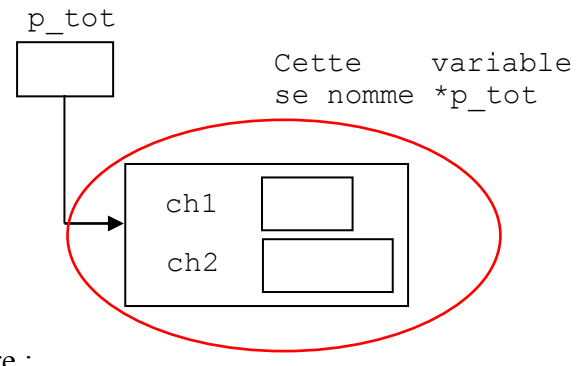
```
toto *p_tot;
p_tot ←reservation(1 toto);
```

Le schéma des variables devient le suivant :



Le but est de pouvoir initialiser les champs `ch1` et `ch2`, connaissant uniquement `p_tot`. Or, ces champs ne sont accessibles, par la notation '.', qu'à partir d'une variable de type `toto`, et non de type `toto *` (ce qu'est `p_tot`).

Fort heureusement, la simple définition de `p_tot` nous indique ce qui doit être fait afin d'initialiser les champs `ch1` et `ch2`. Cette définition est : `toto *p_tot`. Donc, ce qui est de type `toto` et auquel on peut appliquer la notation'.', c'est le contenu de `p_tot`, qui s'écrit `*p_tot`. Cela est d'ailleurs matérialisé sur le schéma ci-dessous.



Ainsi, il suffit d'écrire :

```
(*p_tot).ch1 ← une valeur; // ch1 est de type entier
(*p_tot).ch2 ← une valeur; // ch2 est de type reel
```

La notation -> (flèche)

Dernier opérateur abordé pour ce qui concerne l'algorithmique, l'opérateur -> n'est pas à proprement parler indispensable, son rôle est de procurer une syntaxe plus intuitive que le couple `*` et `.` évoqué dans le paragraphe précédent. Cet opérateur, bien que se lisant 'flèche' est bien symbolisé par un tiret '-' suivi d'un chevron supérieur '>' et non pas →, ceci pour éviter la confusion avec l'opérateur d'affectation ←, et également parce que cette syntaxe est la même que celle qu'utilise le langage C.

Son utilisation est des plus simples. Soit `ptr` un pointeur vers une variable de type structure, et soit `ch` un des champs de cette variable. Dans ce cas, deux écritures sont équivalentes :

```
(*ptr).ch  ⇔  ptr->ch
```

-> ou . ???

Structures et fonctions

Utilité des fonctions dans le cas particulier des structures

L'un des principaux défauts des structures est la limite du langage à devoir travailler uniquement avec les types de base pour les opérations les plus simples (opérateurs arithmétiques, afficher(), saisir()). Ainsi, regrouper des champs dans des structures ne semble servir qu'à définir rapidement les variables de type composé.

Afficher une variable de type composé revient à afficher tous ses champs (pourvu que ceux-ci aient bien un type de base), ce qui multiplie le nombre d'instructions à écrire pour une opération somme toute simple. Simple, certes, mais pas forcément intuitive, car tous les champs n'ont pas le même type. Cela explique pourquoi le compilateur ne peut prendre l'initiative de traiter ces opérations.

Cependant, le fait d'écrire plusieurs instructions n'est pas si gênant que cela si ces instructions doivent être exécutées plusieurs fois : nous disposons en effet des fonctions, qui sont un outil parfait pour ce genre de choses !

Un pas de plus vers la programmation orientée objet

Lorsqu'un travail de modélisation est effectué pour analyser un problème à informatiser, l'ensemble des structures obtenues n'est qu'une partie du résultat. L'autre partie consiste à donner la liste de toutes les fonctions utilisables avec ces structures. Cette analyse répond don (entre autres) à deux questions :

- Que sont mes objets ?
- Que font mes objets ?

Pour toute structure créée, la question à se poser est : quelles sont les opérations à réaliser avec des variables qui ont ce type ?

Ces opérations tombent dans deux catégories :

- Les opérations de base, c'est à dire celles que le langage offre pour les types de base, telles que afficher(), saisir(), les opérateurs arithmétiques

- Les opérations spécifiques à la structure, dans le cas où elle regroupe des informations que l'on souhaite traiter de manière sophistiquée.

Exemple :

La structure pour représenter (modéliser) les nombres complexes sous la forme partie réelle/partie imaginaire

```
structure complexe
{
      reel re, im;
};
```

Il est dans ce cas possible de dresser la liste des opérations à réaliser

- Opérations de base :

  Afficher, saisir, additionner, soustraire, multiplier, diviser

  Comparaisons : égalité, inégalité
- Opérations spécifiques :

  Calcul du module, de l'argument, du complexe conjugué.

Chaque opération donnera lieu à la définition d'une fonction

Exemple:

La structure modélisant un compte en banque

```
structure contenbank
{
      entier      num_compte;
      reel        solde;
};
```

- Opérations de base :

  Afficher, saisir;
- Opérations spécifiques :

  Dépôt, retrait (modification du solde dans les deux cas).

Chaque nouveau type vient accompagné d'un ensemble de fonctions auquel il faut réfléchir avant de coder. Deux types ne sont jamais identiques (sinon, pourquoi se donner la peine d'en écrire deux ?), et donc la liste des fonctions associées n'est jamais la même.

Passage de paramètres

Les structures des comportent comme les autres types en ce qui concerne les fonctions : il est possible de spécifier des entrées dont les types sont des structures, auquel cas le mécanisme de passage de paramètre reste valable. La différence réside dans le fait que plusieurs valeurs numériques doivent être transmises entre l'argument et le paramètre. Pour réaliser cette transmission, le compilateur utilise l'opérateur d'affectation à deux reprises : en réalité, une variable temporaire (à usage interne de l'ordinateur) est créée pour recevoir le valeur de l'expression utilisée comme argument. Cette variable temporaire est ensuite recopiée dans le paramètre de la fonction, toujours à l'aide de l'opérateur d'affectation. Or, cet opérateur, comme nous l'avons déjà expliqué, effectue une recopie champ par champ de toutes les valeurs stockées dans la structure ! Ainsi, pour une structure comportant 20 champs, ce sont 40 copies de valeurs qui seront faites pour le passage d'un paramètre. Cela est très coûteux en temps de calcul pour une opération somme toute assez simple : recopier des valeurs. A ce titre, et pour éviter de surcharger inutilement l'ordinateur, il est de loin préférable de transmettre une adresse lorsque l'on travaille avec des structures.

Tout paramètre dont le type est une structure doit être transformé en pointeur vers une structure pour accélérer le traitement.

Cela n'est pas une obligation syntaxique, le compilateur acceptera une structure en entrée, il vous appartient donc de bien veiller à l'emploi de pointeurs.

Sorties de type structure

De manière symétrique à celui des entrées (paramètres), il est tout à fait possible d'avoir en sortie de fonction une structure. Le mécanisme de transmission de la valeur de retour étant quasiment identique à celui du passage de paramètres, la technique à employer est également la même : il faut utiliser un pointeur vers une structure comme type de retour plutôt que la structure elle-même. Cette technique possède également des avantages liés à la réservation de mémoire, avantage qui sera exploité avec l'utilisation des listes chaînées traitées dans le chapitre suivant.

Des exemples de fonctions avec des structures : les nombres complexes.

Rappelons une fois de plus la structure utilisée en exemple :

```
structure complexe
{
      reel re,im;
};
```

Les trois fonction traitées à titre d'exemple sont : affichage, saisie et calcul du module d'un nombre complexe.

Pour l'affichage, l'entête de fonction qui vient naturellement à l'esprit est la suivante :

`fonction affich_comp(entrée : complexe c)`, qui est tout à fait correcte, mais le paramètre n'est pas un pointeur.

La bonne version de la fonction est la suivante :

```
fonction affich_comp(entrée : complexe *p_c)
{
      afficher(pc->re);
      afficher(p_c->im);

      retourner;
}
```

Pour la fonction de saisie, la question de l'emploi ou non d'un pointeur ne se pose même pas : puisqu'une saisie modifie les valeurs de la structure, il est nécessaire d'employer un pointeur. La fonction est donc, dans ce cas :

```
fonction saisie_comp(entrée : complexe *p_c)
{
      afficher("entrez la partie reelle puis la partie imaginaire:");

      saisir(pc->re);
      saisir(p_c->im);

      retourner;
}
```

Enfin, pour le calcul du module d'un nombre complexe (qui est une valeur réelle), l'entête suivante est correcte : `fonction module(entree : complexe c → sortie : reel)`, mais on lui préfèrera la fonction suivante :

```
fonction module(entree : complexe *p_c → sortie : reel)
```

```
{
      reel a,b,m;

      a ← p_c->re;
      b ← pc->im;

      m ← racine(a*a+b*b);

      retourner m;
}
```

une dernière fonction à titre d'exemple : l'addition de deux complexes. Cette fonction doit avoir deux entrées et une sortie, toutes de type complexe.

Version sans pointeur (correcte mais plus longue à l'usage)

```
fonction addition(entree : complexe a, complexe b → sortie : complexe)
{
      complexe res;

      res.re ← a.re + b.re;
      res.im ← a.im + b.im;

      retourner res;
}
```

Version avec pointeurs :

```
fonction addition(entree : complexe *a, complexe *b → sortie : complexe *)
{
      complexe *res;

      res ← reservation(1 complexe); // et oui, res est un pointeur

      res->re ← a->re + b->re;
      res->im ← a->im + b->im;

      retourner res;
}
```