Programmation structurée

Les langages de programmation ont commencé très tôt à assembler les instructions sous la forme de groupes réutilisables, les fonctions. Les variables ont naturellement pris le même chemin, bien qu'un peu plus tardivement.

Le tableau permet de traiter certains algorithmes, à condition que la donnée à traiter soit d'un type uniforme (`char` , `int` ...). Lorsque la donnée à traiter contient des informations de natures différentes, il faut recourir à plusieurs tableaux, ou bien à un seul tableau en utilisant un type fourre-tout `void*` . Il faut bien le reconnaître, cette solution est à proscrire.

À la place, nous définissons des structures regroupant plusieurs variables appelées champs. Ces variables existent en autant d'exemplaires que souhaité, chaque exemplaire prenant le nom d'instance.

Le langage C++ connaît plusieurs formes composites :

- les structures et les unions, aménagées à partir du C ;
- les classes, qui seront traitées au chapitre suivant.

1. Structures

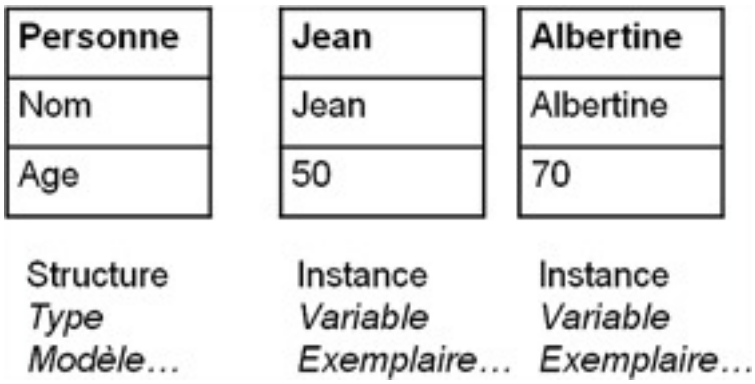
Les structures du C++ - comme celles du C - définissent de nouveaux types de données. Le nom donné à la structure engendre un type de données :

```
struct Personne
{
  char nom[50];
  int age;
} ;
```

À partir de cette structure `Personne` , nous allons maintenant créer des variables, en suivant la syntaxe habituelle de déclaration qui associe un type et un nom :

```
Personne jean, albertine;
```

`jean` et `albertine` sont deux variables du type `Personne`. Comme il s'agit d'un type non primitif (`char`, `int`...), on dit qu'il s'agit d'instances de la structure `Personne`. Le terme instance rappelle que le nom et l'âge sont des caractéristiques propres à chaque personne.



On utilise une notation particulière pour atteindre les champs d'une instance :

```
jean.age = 50;  // l'âge de jean
printf("%s",albertine.nom); // le nom d'albertine
```

Cette notation relie le champ à son instance.

a. Constitution d'une structure

Une structure peut contenir un nombre illimité de champ. Pour le lecteur qui découvre ce type de programmation et qui est habitué aux bases de données, il est utile de comparer une structure avec une table dans une base.

C++	SQL
structure	table
champ	champ / colonne
instance	enregistrement

Chaque champ de la structure est bien entendu d'un type particulier. Il peut être d'un type primitif (`char`, `int`...) ou bien d'un type structure. On peut aussi obtenir des constructions intéressantes, par composition :

```
struct Adresse
{
  char *adresse1, *adresse2;
  int code_postal;
  char* ville;
};

struct Client
{
  char*nom;
  Adresse adresse;
};
```

On utilise l'opérateur point comme aiguilleur pour atteindre le champ désiré :

```
Client cli;
cli.nom = "Les minoteries réunies";
cli.adresse.ville = "Pau";
```

Enfin, il est possible de définir des structures auto-référentes, c'est-à-dire des structures dont un des champs est un pointeur vers une instance de la même structure.

Cela permet de construire des structures dynamiques, telles que les listes, les arbres et les graphes :

```
struct Liste
{
  char* element;
  Liste* suite;
} ;
```

Le compilateur n'a aucun mal à envisager cette construction : il connaît fort bien la taille d'un pointeur, généralement 4 octets, donc pour lui la taille de la structure est parfaitement calculable.

b. Instanciation de structures

Il existe plusieurs moyens d'instancier une structure. Nous l'avons vu, une structure engendre un nouveau type de données, donc la syntaxe classique pour déclarer des variables fonctionne très bien :

```
Personne mireille;
```

Instanciation à la définition

La syntaxe de déclaration d'une structure nous réserve une surprise : il est possible de définir des instances aussitôt après la déclaration du type :

```
struct Personne
{
  char*nom;
  int age;
} jean,albertine ;
```

Dans notre cas, `jean` et `albertine` sont deux instances de la structure `Personne`. Bien sûr, il est possible par la suite d'utiliser d'autres modes d'instanciation.

Instanciation par réservation de mémoire

Finalement, l'instanciation agit comme l'allocation d'un espace segmenté pour ranger les champs du nouvel exemplaire. La fonction `malloc()` qui alloue des octets accomplit justement cette tâche :

```
Personne* serge = (Personne*) malloc(sizeof(Personne))
```

La fonction `malloc()` retournant un pointeur sur `void`, on opère un transtypage (`cast`) vers le type `Personne*` dans le but d'accorder chaque côté de l'égalité. L'opérateur **`sizeof()`** détermine la taille de la structure, en octets.

On accède alors aux champs par l'opérateur `->` qui remplace le point :

```
serge->age = 37;
```

Pour réserver plusieurs instances consécutives - un tableau -, il faut multiplier la taille de la structure par le nombre d'éléments à réserver :

```
Personne* personnel = (Personne*) malloc(sizeof(Personne)*5)
```

Pour accéder à un champ d'une instance, on combine la notation précédente avec celle employée pour les tableaux :

```
personnel[0]->nom = "Georgette";
personnel[0]->age = 41;
personnel[1]->nom = "Amandine";
personnel[1]->age = 27;
```

La fonction `malloc()` est déclarée dans l'en-tête **`<memory.h>`** qu'il faut inclure si besoin. Ce type d'instanciation fonctionnait déjà en langage C, mais l'opérateur `new`, introduit en C++, va plus loin.

c. Instanciation avec l'opérateur new

En effet, l'opérateur `new` simplifie la syntaxe, car il renvoie un pointeur correspondant au

type alloué :

```
Personne*josette = new Personne;
```

Aucun transtypage n'est nécessaire, l'opérateur `new` appliqué à la structure `Personne` renvoyant un pointeur (une adresse) de type `Personne*` .

Pour réserver un tableau, il suffit d'ajouter le nombre d'éléments entre crochets :

```
Personne*employes = new Personne[10];
```

Là encore, la syntaxe est simplifiée, puisqu'il est inutile de préciser la taille de chaque instance. L'opérateur `new` la prend directement en compte, sachant qu'il est appliqué à un type particulier, en l'occurrence `Personne` .

La simplification de l'écriture n'est pas la seule avancée de l'opérateur `new`. Si la structure dispose d'un constructeur (voir à ce sujet le chapitre sur les classes), celui-ci est appelé lorsque la structure est instanciée par le biais de l'opérateur `new`, alors que la fonction `malloc()` se contente de réserver de la mémoire. Le rôle d'un constructeur, fonction "interne" à la structure, est d'initialiser les champs de la nouvelle instance. Nous reviendrons en détail sur son fonctionnement.

d. Pointeurs et structures

Quelle que soit la façon dont a été instanciée la structure, par `malloc()` ou par `new`, l'accès au champ se fait à l'aide de la notation flèche plutôt que point, cette dernière étant réservée pour l'accès à une instance par valeur.

L'opérateur **&** appliqué à une instance a le même sens que pour n'importe quelle variable, à savoir son adresse.

Si cet opérateur est combiné à l'accès à un champ, on peut obtenir l'adresse de ce champ pour une instance en particulier. Le tableau ci-après résume ces modalités d'accès. Pour le lire, nous considérons les lignes suivantes :

```
Personne jean;
Personne* daniel = new Personne;
Personne* personnel = new Personne[10];
```

`jean.age`	Le champ `age` se rapportant à `jean`.
`daniel->age`	Le champ `age` se rapportant à `daniel`, ce dernier étant un pointeur.
`&jean`	L'adresse de jean. Permet d'écrire `Personne*jean_prime=&jean`.
`&jean.age`	L'adresse du champ `age` pour l'instance `jean`.
`&daniel`	L'adresse du pointeur `daniel`, qui n'est pas celle de l'instance.
`&daniel->age`	L'adresse du champ `age` pour l'instance `daniel`.
`personnel[2]->age`	L'âge de la personne à l'index 2 du tableau (soit la troisième case en partant de 0).
`personne[2]`	L'adresse de la personne à l'index 2 du tableau (soit la troisième case en partant de 0).
`&personne[2]->age`	L'adresse du champ `age` pour la personne à l'index 2 du tableau (soit la troisième case en partant de 0).

Nous constatons qu'aucune nouveauté n'a fait son apparition. Les notations demeurent cohérentes.

e. Organisation de la programmation

Lorsqu'une structure est créée, il n'est pas rare de la voir définie dans un fichier d'en-tête `.h` portant son nom. Par la suite, tous les modules `cpp` contenant des fonctions qui vont utiliser cette structure devront inclure le fichier par l'intermédiaire de la directive `#include`.

Inclusion à l'aide de #include

Prenons le cas de notre structure `Personne`, elle sera définie dans le fichier `personne.h` :

```
// Fichier : personne.h
struct Personne
{
  char nom[50];
  int age;
} ;
```

Chaque module d'extension `.cpp` l'utilisant doit lui-même inclure ce fichier, sachant qu'il est compilé séparément des autres :

```
#include "personne.h"

int main()
{
Personne jean;
  Jean.age=30;
}
```

Protection contre les inclusions multiples

Le système des en-têtes donne de bons résultats mais parfois certains fichiers `.h` sont inclus plusieurs fois, ce qui conduit à des déclarations multiples du type `Personne`, fait très peu apprécié par le compilateur.

Nous disposons de deux moyens pour régler cette difficulté. Tout d'abord, il est possible d'employer une directive de compilation `#ifndef` suivie d'un `#define` :

```
#ifndef _Personne
#define _Personne

// Fichier : personne.h
struct Personne
{
  char nom[50];
  int age;
} ;
#endif
```

La seconde technique, plus simple, consiste à utiliser une directive propre à un compilateur, `#pragma once`. Cette directive, placée en début de fichier d'en-tête, assure que le contenu ne sera pas accidentellement inclus deux fois. Si le cas se produit, le compilateur recevra une version ne contenant pas deux fois la même définition, donc, nous n'aurons pas d'erreur.

La première technique semble peut-être moins directe, pourtant elle est davantage portable, puisque les directives `#pragma` (pour pragmatique) dépendent de chaque compilateur. Vous êtes bien entendu susceptible de rencontrer les deux dans un programme C++ tiers.

2. Unions

Une union est une structure spéciale à l'intérieur de laquelle les champs se recouvrent. Cette construction particulière autorise un tassement des données, une économie substantielle. Lorsqu'un champ est écrit pour une instance, il écrase les autres puisque tous les champs ont la même adresse. La taille de l'union correspond donc à la taille du champ le plus large.

Les unions ont deux types d'applications. Pour commencer, cela permet de segmenter une structure de différentes manières. Nous pouvons citer comme exemple la structure `address`, qui accueille une union destinée à représenter différents formats d'adresses réseau. En fonction de la nature du réseau - IP, Apple Talk, SPX -, les adresses sont représentées de manières différentes.

Comme deuxième type d'application, nous pouvons penser à la représentation des nombres. Suivant que nous souhaitons privilégier la vitesse ou la précision des calculs, nous pouvons utiliser une union pour calculer en `int`, en `float` ou en `double`. Utiliser une structure ne serait pas une bonne idée car chaque "nombre" occuperait 4+4+8 soit 16 octets. L'union donne de meilleurs résultats :

```
union Valeur
{
   int nb_i;
   float nb_f;
   double nb_d;
} ;
```

La taille de cette union égale 8 octets, soit l'espace occupé par un double.

Il n'est pas rare d'inclure une union dans une structure, un champ supplémentaire indiquant lequel des champs de l'union est "actif" :

```
enum TNB { t_aucun,t_int, t_float, t_double };

struct Nombre
{
  char t_nb;
  union Valeur
  {
    int nb_i;
    float nb_f;
    double nb_d;
  } val ;
} ;
```

Nous pouvons à présent imaginer quelques fonctions pour travailler avec cette construction :

```
void afficher(Nombre& n)
{
  switch(n.t_nb)
```

```
 {
  case t_int:
   printf("%d\t",n.val.nb_i);
   break;
  case t_float:
   printf("%f\t",n.val.nb_f);
   break;
  case t_double:
   printf("%f\t",n.val.nb_d);
   break;
  }
}

Nombre* lire_int()
{
  Nombre* c=new Nombre;
  c->t_nb=t_int;
  printf("entier: ");
  scanf("%d",&c->val.nb_i);

  return c;
}

Nombre* lire_float()
{
  Nombre* c=new Nombre;
  c->t_nb=t_float;
  printf("décimal: ");
  scanf("%f",&c->val.nb_f);
  //cin >> c->val.nb_f; // cin >> float&

  return c;
}
Nombre lire_double()
{
  Nombre c;
  c.t_nb=t_double;
  printf("double: ");
  scanf("%lg",&c.val.nb_d);
  return c;
```

```
}
```

Dans cet exemple, nous avons mélangé différents modes de passage ainsi que différents modes de retour (valeur, adresse...).

Pour ce qui est de l'instanciation et de l'accès aux champs, l'union adopte les mêmes usages (notations) que la structure.

3. Copie de structures

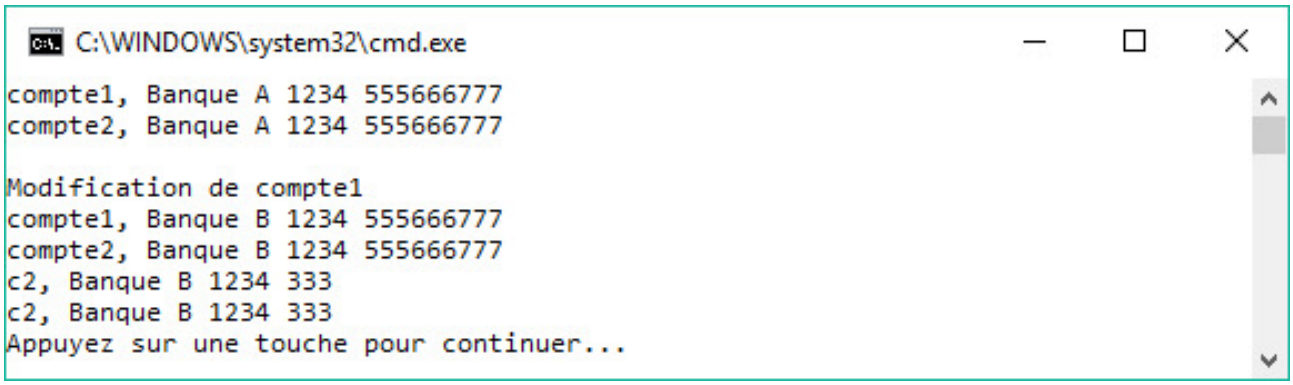
Que se passe-t-il lorsque nous copions une structure par valeur dans une autre structure ? Par exemple, que donne l'exécution du programme suivant ?

```
struct Rib
{
  char*banque;
  int guichet;
  int compte;
} ;

int main(int argc, char* argv[])
{
  Rib compte1,compte2;
  compte1.banque = "Banca";
  compte1.guichet = 1234;
  compte1.compte = 555666777;

 compte2 = compte1;

  printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
  printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);
  return 0;
}
```

L'exécution de ce programme indique que les valeurs de chaque champ sont effectivement copiées de la structure `compte1` à la structure `compte2`. Il faut cependant faire attention au champ `banque`. S'agissant d'un pointeur, la modification de la zone pointée affectera les deux instances :

```
Rib compte1,compte2;

 // alloue une seule chaîne de caractères
 char* banque=new char[50];
 strcpy(banque,"Banque A");

 // renseigne la première instance
 compte1.banque = banque;
 compte1.guichet = 1234;
 compte1.compte = 555666777;

 // copie les valeurs dans la seconde
 compte2 = compte1;

 // affichage
 printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
 printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);

 // apparemment, on modifie uniquement compte1
printf("\nModification de compte1\n");
 strcpy(compte1.banque,"Banque B");

 // vérification faite, les deux instances sont sur "Banque B"
 printf("compte1, %s %d %d\n",compte1.banque,compte1.guichet,
compte1.compte);
 printf("compte2, %s %d %d\n",compte2.banque,compte2.guichet,
compte2.compte);
```

Vérification faite, l'exécution indique bien que les deux pointeurs désignent la même adresse :



La copie des structures contenant des champs de type pointeur n'est pas la seule situation à poser des problèmes. Considérons à présent l'extrait suivant :

```
Rib *c1,*c2;
 c1 = &compte1;
 c2 = c1;
 c1->compte = 333;
 printf("c2, %s %d %d\n",c2->banque,c2->guichet,c2->compte);
```

L'exécution indique 333 comme numéro de compte pour `c2`. Autrement dit, `c1` et `c2` désignent le même objet, et l'affectation `c2=c1` n'a rien copié du tout, sauf l'adresse de l'instance.

Il aurait été plus judicieux d'utiliser la fonction `memcpy()` :

```
c2 = new Rib;
 memcpy(c2,c1,sizeof(Rib));
 c1->compte = 222;
 printf("c2, %s %d %d\n",c2->banque,c2->guichet,c2->compte);
```

Cette fois, l'exécution indique bien que `c2` et `c1` sont indépendants. C'est l'allocation par `new` (ou `malloc()`) qui aura fait la différence.

4. Création d'alias de types de structures

L'instruction `typedef` étudiée au chapitre précédent sert également à définir des alias (de types) de structures :

```
// une structure décrivant un nombre complexe
struct NombreComplexe
{
   double reel,imaginaire;
};

typedef NombreComplexe Complexe;    // alias de type
typedef NombreComplexe* Pcomplexe;    // alias de type pointeur
typedef NombreComplexe& Rcomplexe;    // alias de type référence

int main()
{
   Complexe c1;
   PComplexe pc1 = new Complexe;
   RComplexe rc1 = c1;

   return 0;
}
```

Cet usage est généralement dévolu aux API système qui définissent des types puis des appellations pour différents environnements. Par exemple, le `char*` devient `LPCTSTR` (*Long Pointer To Constant STRing*), le pointeur de structure `Rect` devient `LPRECT` ...

5. Structures et fonctions

Ce sont les fonctions qui opèrent sur les structures. Il est fréquent de déclarer les structures comme variables locales d'une fonction dans le but de les utiliser comme paramètres d'autres fonctions. Quel est alors le meilleur moyen pour les transmettre ? Nous avons à notre disposition les trois modes habituels, par valeur, par adresse (pointeur) ou par référence.

a. Passer une structure par valeur comme paramètre

Le mode par valeur est indiqué si la structure est de petite taille et si ses valeurs doivent être protégées contre toute modification intempestive de la part de la fonction appelée. Ce mode implique la recopie de tous les champs de l'instance dans la pile, ce qui peut prendre un certain temps et consommer des ressources mémoire forcément limitées. Dans le cas des fonctions récursives, la taille de la pile a déjà tendance à grandir rapidement, il n'est donc pas judicieux de la surcharger inutilement.

Toutefois, cette copie empêche des effets de bord puisque c'est une copie de la structure qui est passée.

```
void afficher(Nombre n)
{
 switch(n.t_nb)
 {
 case t_int:
  printf("%d\t",n.val.nb_i);
  break;
 case t_float:
  printf("%f\t",n.val.nb_f);
  break;
 case t_double:
  printf("%f\t",n.val.nb_d);
  break;
 }
}
```

b. Passer une structure par référence comme paramètre

Ce mode constitue une avancée considérable, puisque c'est la référence (adresse inaltérable) de la structure qui est transmise. Cette information occupe 4 octets (en compilation 32 bits) et autorise les effets de bord sous certaines conditions. De plus, le passage par référence est transparent pour le programmeur, ce dernier n'ayant aucune chance de passer une valeur littérale comme instance de structure.

```
void afficher(Nombre& n)
```

Nous verrons comment la structure (classe) peut être aménagée de manière à ce que

l'accès en modification des champs puisse être contrôlé de manière fine.

Il est possible de protéger la structure en déclarant le paramètre à l'aide du mot-clé `const` :

```
void afficher(const Nombre& n)
{
  n.val.nb_i=10;  // erreur, n invariable
}
```

c. Passer une structure par adresse comme paramètre

Ce mode continue à être le plus utilisé, sans doute car il est la seule alternative au passage par valeur autorisée en langage C. Il nécessite parfois d'employer l'opérateur `&` à l'appel de la fonction, et le pointeur reçu par la fonction se conforme à l'arithmétique des pointeurs. Enfin, ce mode autorise les effets de bord.

```
void afficher(Nombre* n)
```

d. De la programmation fonctionnelle à la programmation objet

En admettant que certaines fonctions puissent migrer à l'intérieur de la structure, dans le but évident de s'appliquer aux champs d'une instance en particulier, nous découvrons cette notion de la programmation orientée objet que l'on appelle l'encapsulation, c'est-à-dire la réunion d'une structure et de fonctions. Cette construction est tout à fait légale en C++ :

```
struct Nombre
{
  char t_nb;
  union Valeur
  {
    int nb_i;
    float nb_f;
    double nb_d;
```

```
 } val ;

 void afficher()
 {
  switch(t_nb)
  {
   case t_int:
      printf("%d\t",val.nb_i);
      break;
   case t_float:
     printf("%f\t",val.nb_f);
      break;
   case t_double:
      printf("%f\t",val.nb_d);
      break;
  }
 }
} ;
```

Connaissant un `nombre`, il est très facile de modifier les notations pour utiliser cette méthode `afficher()` en remplacement de la fonction `afficher(Nombre&)` :

```
Nombre a;
  a =lire_float();  // utilise la fonction lire_float()
  a.afficher();  // utilise la méthode afficher()
  afficher(a);  // utilise la fonction afficher()
```

Nous verrons au chapitre suivant ce qui distingue la programmation fonctionnelle de la programmation orientée objet. Pour le lecteur qui connaît la notion de visibilité dans une classe, il est utile de préciser que les membres d'une structure (champs ou méthodes) sont publics par défaut.