Chapitre 3
Structuration des données

1. 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 La programmation orientée objet.

1.1 Structures

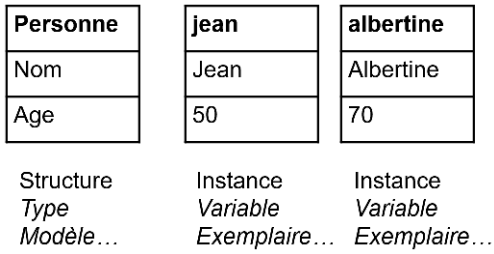
Les structures du C++, comme celles du C, définissent de nouveaux types de données. Le nom donné à la structure engendre un type de données :

```
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.

1.2 Constitution d'une structure

Une structure peut contenir un nombre illimité de champs. 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 autoréférentes, 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 la taille d'un pointeur, généralement 4 octets, donc, pour lui, la taille de la structure est parfaitement calculable.

1.3 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;
```

1.4 Instanciation au moment de 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.

1.5 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.

1.6 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 La programmation orientée objet 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 par la suite.

1.7 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.

1.8 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`.

1.8.1 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;
}
```

1.8.2 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.

1.9 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.