Pointeurs, tableaux et chaînes de caractères

Nicolas Belloir

24 octobre 2005

1 Structures, unions, énumérations et types

1.1 Définitions

Définition
- Un *enregistrement* est un mécanisme permettant de regrouper un certain nombre de variables de types différents au sein d'une même entité. Les éléments d'un *enregistrement* sont appelés les *champs*.
- En langage C/C++, un enregistrement est appelé une *structure* et un champ est appelé un *membre* de la structure.

Example 1. On utilise une structure pour représenter le concept d'adresse. Une adresse comprend généralement une rue, un code postal et une ville.

1.2 Déclarations

Première méthode
Déclaration d'une *étiquette de structure*.

```
/* Creation de l'etiquette de structure */
struct adresse{
    char sRue[100];
    int iCodePostal;
    char sVille[20];
};

/* Declaration de variables */
struct adresse ad1, ad2;
```

Deuxième méthode
Déclaration de variables de type structure sans utilisation d'*étiquette de structure*.

```
struct{
    char sRue[100];
    int iCodePostal;
    char sVille[20];
} ad1, ad2;
```

L'inconvénient : impossible par la suite de déclarer une autre variable du même type.

Troisième méthode
Combinaison d'une *étiquette de structure* avec une définition simple.

```
struct adresse {
    char sRue[100];
    int iCodePostal;
    char sVille[20];
} ad1, ad2;
```

Initialisation à la déclaration
Une structure peut être initialisée par une liste d'expressions constantes.

```
struct adresse ad1 = {"49, rue Emile Gare", 64000, "Pau"};
```

1.3 Opérations sur les structures

Accès aux membres de la structure
Pour accéder aux membres d'une structure, utiliser l'opérateur . (point).

Example 2. Pour accéder au membre *iCodePostal* de la variable *ad1* de type adresse, faire *ad1.iCodePostal*.

Affectation de structures
On peut affecter une structure à une variable structure de même type, grâce à l'opérateur d'affectation.

```
struct adresse ad1,ad2

...

ad2 = ad1;
```

Example 3.

Comparaison de structures
Aucune opération de comparaison n'est possible sur les structures, pas même avec les opérateurs == ou !=.

1.4 Cas particuliers

Enumérations
Les énumérations permettent de déclarer des constantes nommées. Les énumérations fonctionnent syntaxiquement comme des structures et peuvent être déclarées sous forme de variable.

```
/* enumeration simple */
enum {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
DIMANCHE};

/* Enumeration de type variable */
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
DIMANCHE};

enum jour j1, j2;
j1 = LUNDI;
j2 = MARDI;
```

Example 4.

Les unions
Les unions permettent de manipuler des variables auxquelles on désire affecter des valeurs de types différents.

```
union nombre{
    int i;
    float g;
}

union nombre n;

n.i = 10;    ou n.f = 3.14159
```

Example 5.

1.5 Définition de types particuliers

typedef
Le mot clé `typedef` permet de définir des types spécifiques. Cela permet d'alléger le code et de rendre les programmes plus lisibles.

```
/* declare tab comme le type tableau de 10 */
typedef int tab[10];

/* declare adresse comme etant le type structure à 2 champs */
typedef struct{
    char sRue[100];
    int iCodePostal;
    char sVille[20];
} adresse;

tab t1;
adresse ad1, ad2;
```

Example 6.

Exercice
Écrire le programme permettant de définir une structure représentant les informations d'un étudiant : nom, prénom, adresse, diplôme préparé, mention, année dans le cursus.

```
#include <iostream.h>
#define TLABEL 30
#define TLABEL_LONG 100

int main() {
    typedef struct{
        char sDiplome[TLABEL];
        char sMention[TLABEL];
        short iAnnee;
    } inscription;

    typedef struct{
        char sRue[TLABEL_LONG];
        int iCodePostal;
        char sVille[TLABEL];
    }adresse;

...
```

Example 7.

```
...
    typedef struct{
        char sNom[TLABEL];
        char sPrenom[TLABEL];
        adresse ad;
        inscription insc;
    }etudiant;

    etudiant ed1;
}
```

Example 8.

2 Les pointeurs

Définition
Un pointeur est simplement une variable contenant un type particulier de donnée : **une adresse**.

```
int * pi ; /* pi est un pointeur vers un entier  (int)*/
short * psi ; /* psi est un pointeur vers un entier court */
char * pc ; /* pc est un pointeur vers un caractère */
```

Example 9.

2.1 L'opérateur de référence &

Définition
L'opérateur *&* fournit l'adresse de son opérande :

```
int a ;
/* &a est du type pointeur vers un entier */
/* &a est une adresse d'entier */
```

Example 10.

Remarque
Attention ! Les formes `&(x+1)` et `&3` sont interdites, mais `&x + 1` est permise. De même l'opérateur & n'est pas autorisé sur les variables ayant l'attribut `register`.

2.2 L'opérateur d'indirection *

Définition
L'opérateur * fournit l'objet pointé par son opérande :

```
/* si ptr est du type pointeur vers un entier */
/* *ptr est du type entier */
```

Example 11.

2.3 Equivalences

Equivalences

```
&(*pointeur)     <=>   pointeur
*(&var)          <=>   var
```

2.4 Le type pointeur

Définition
- Quel que soit un type T, on peut créer un type pointeur vers T,
- La partie réservée a la taille est nécessaire et suffisante pour contenir une adresse. La déclaration d'une variable pointeur fait toujours intervenir le type des objets pointés, on dit que le pointeur est typé.

```
T* ident ;
/* ident est du type pointeur vers T */
```

Example 12.

Remarque
- Quelle est la différence entre `int* ptr ;` et `int *ptr ;` ?
- Il n'y a aucune différence mais attention :

int* a, b ;	int *a, *b ;
/* a pointeur d'entiers */ /* b entier */	/* a et b pointeurs */

2.5 Arithmétique des pointeurs

La valeur *NULL*

Définition
`NULL` (toujours en majuscules) est une constante pointeur définie dans <`stdio.h`> et valant 0. Cette valeur signifie “ne repère aucun objet”.

```
int * ptr = NULL;
```

Example 13.

Affectation de pointeurs

Définition
Le signe `=` peut être employé.

```
int *p, *q;
p = NULL;
p = q;
```

Example 14.

Conversions automatiques

Définition
- Le nom d’une fonction est converti en pointeur sur cette fonction.
- Le nom d’un tableau est converti en pointeur sur son premier élément.

Relations entre pointeurs

Définition
Les opérateurs classiques de relation sont utilisables avec les pointeurs :

```
== != < > <= >=
```

Addition et soustraction avec un entier

```
T *ptr;
int i;
/* ptr + i <=> (ptr + i*sizeof(Type)) */
```

Example 15.

Remarque
Il en va de même pour la soustraction d’un entier à un pointeur.

Initialisation

Remarque
Comme toute variable en C, un pointeur doit être initialisé pour être utilisé correctement :
- Initialisation directe à une adresse : permet d'accéder à des zones spécifiques du système[1] ;
- Initialisation à une adresse calculée : `ptr = &i ; /*par exemple*/` ;
- Initialisation par allocation de mémoire. Exemple :

```
int *ptr;
/*  *ptr = 1 est incorrect */
ptr = malloc (100*sizeof(int));
*ptr = 1; /* maintenant autorisé */
```

Exercice
- Déclarer un entier `i` et un pointeur `pi` ;
- Initialiser l'entier à une valeur arbitraire et faire pointer `pi` vers `i` ;
- Imprimer la valeur de `i` ;
- Modifier l'entier pointé par `pi` (en utilisant `pi`, pas i) ;
- Imprimer la valeur de `i` ;

```
int main() {
    int i;
    int *pi;

    i = 1;
    pi = &i;

    cout << "Valeur de i avant modif : " << i;
    *p = 2;
    cout << "Valeur de i apres modif : " << i;
}
```

Example 16.

3 Les tableaux

3.1 Éléments de définition

Définition
Un tableau est un ensemble d'objets du même type. Chaque objet est appelé élément du tableau. La taille (i.e. le nombre d'éléments) est donné par une expression entière et positive[2], et précisée entre crochets [].

```
/* T objet[constante]([...]); */
int tab[10];
/* tab est un tableau de 10 entiers */
```

Example 17.

Les éléments

Définition
- L'implantation en mémoire d'un tableau est contiguë. Les éléments du tableau sont indicés de 0 à `nombre_d_éléments - 1`
- Attention ! La vérification de non débordement du tableau n'est pas assurée. L'origine ne peut être déplacée comme dans d'autres langages, elle est toujours à 0.

[1] Donc à éviter !
[2] Cette expression doit être statique, c'est à dire évaluable lors de la phase de compilation.

	Adresse	valeur
	...	...
tab[0][0]	52000	10
tab[0][1]	52002	14
tab[1][0]	52004	700
tab[1][1]	52006	54
	52008	...

Initialisation

```
int matrice [3][3] = {1,2,0,3};
/* <=> */
int matrice [3][3] = {
{1,2,0}, {3,0,0}, {0,0,0} };
```

Example 18.

[80] != []

Nombre d'éléments

Le nombre d'éléments d'un tableau peut être omis s'il n'est pas nécessaire au compilateur :

- le tableau est déclaré mais déjà défini : `extern int tab[] ;` par exemple,
- paramètre de fonction : `void f(int tab[]) ;` par exemple,
- initialisation à la définition : `char chaine[] = "coucou" ;` par exemple.

3.2 Tableaux et pointeurs

Éléments communs

- `sizeof(tab)` : taille du tableau,
- `&tab` : pointeur vers un tableau[3]

Dans toutes les autres utilisations, `tab` désigne le pointeur vers le premier élément du tableau :

```
tab <=> &tab[0]
```

Éléments communs

- L'opérateur crochet [ ], qui permet de désigner un élément d'un tableau, est automatiquement traduit (par le compilateur) en un chemin d'accès utilisant le nom du tableau[4]
- on peut donc écrire en C :
  `"coucou"[3] == 'c' == 3["coucou"]` !!

```
tab[i]  <=>  *(tab+i)
```

3.3 Opérations

Qu'utiliser ?

L'affectation et la comparaison de tableaux n'existent pas en C. Il faut utiliser des fonctions particulières pour pouvoir affecter et comparer des tableaux

[3] &tab == tab dans la plupart des compilateurs.

[4] <=> i[tab] !

Affectation

Formalisme

```
tab1 = tab2      /* INTERDIT */
```

A ne pas faire car cela équivaudrait à faire `&tab1[0] = ...` ce qui est interdit. Il faut utiliser la copie de blocs mémoire :

```
memcopy(tab1, tab2, sizeof(tab2))
```

Comparaison

Formalisme

```
tab1 == tab2
```

Attention ! Compare en fait les 2 adresses de début de tableau `&tab1[0]` et `&tab2[0]` ; ce n'est pas cela que l'on veut. Là aussi il faut utiliser une fonction de comparaison des blocs mémoire :

```
memcmp(tab1, tab2, max(sizeof(tab1),sizeof(tab2)))
```

4 Les chaînes de caractères

4.1 Elles n'existent pas en C !

Formalisme
Les chaînes de caractères ne sont utilisables qu'à travers un tableau de caractères dont le dernier élément est le caractère spécial '\0' (le caractère nul, dont le code ASCII est 0).

```
char chaine[4] = {'o', 'u', 'i', '\0'};
/*équivalent à : */
char chaine[4] = "oui";
```

Example 19.

Formalisme

- Les constantes chaînes de caractères (qui sont notées entre "double quotes") contiennent implicitement le caractère final '\0'.
- Pour afficher une chaîne avec `printf()`, ou en saisir une avec `scanf()`, utiliser la spécification de format %s et passer le nom de la chaîne en argument.

```
char chaine[4] = "oui";
printf("La chaîne vaut : %s \n", chaine);
```

Example 20.

4.2 Différentes déclaration de chaînes

Remarque
Les conséquences des deux définitions (char chaine[] et char *chaine) ne sont pas les mêmes.

Example 21.

`char ch1[] ="oui";`

ch1[0]	52000	'o'
ch1[1]	52002	'u'
ch1[2]	52004	'i'
ch1[3]	52006	'\0'
	52008	...
	...	...

`char *ch2 = "oui";`

ch2	72000	**80000**
	...	...
	80000	'o'
	80002	'u'
	80004	'i'
	80006	'\0'
	80008	...

Donc : `ch1 = ...;` est interdit et `ch2 = ...;` autorisé.

4.3 Les fonctions de manipulation

Librairie
On les trouve dans la librairie <`string.h`>

```
#include <string.h>
```

Entrées/sorties
- La fonction `gets()` lit les caractères en entrée jusqu'au moment où elle rencontre '\n',
- La fonction `puts()` imprime la chaîne (jusqu'à '\0').

```
gets(chaine);
puts(chaine);
```

Example 22.

La fonction de copie et de comparaison
- La fonction `strcpy()` permet de recopier les chaînes de caractères.
- Pour comparer le contenu de deux chaînes de caractères, on utilise la fonction `strcmp()`.

```
strcpy(chaine1, "coucou");
strcmp(chaine1, chaine2);
```

Example 23.

4.4 Différences de déclaration

La différence entre char **tab, char *tab[], char tab[][] et char (*tab)[]
Les tableaux de chaînes de caractères ne sont donc que des tableaux de tableaux de caractères. Lorsqu'on les manipule, on se pose très souvent la question de la différence entre les formes d'écriture :

```
char **tab
char *tab[]
char tab[][]
char (*tab)[]
```

La déclaration char tab[5][10]

Réserve 5*10=50 cases pour les 50 caractères du texte composé de 5 lignes et de 10 colonnes :

tab[0][0]	52000	char
tab[0][1]	52002	char
...	...	...
tab[4][9]	52098	char
	52100	...
	...	...

La déclaration char *tab[5]

Signifie "**tableau de 5 pointeurs de caractères**"

tab[0]	52000	ptr
tab[1]	52002	ptr
...	...	...
tab[4]	52006	ptr
	52008	...

Remarque
Cette déclaration permet de gagner de la place mémoire. De plus, comme nous l'avons déjà vu, cette définition est utile lorsque le tableau est déjà défini et dans le cas de la déclaration d'un paramètre formel. En effet si l'on considère une fonction `void fct(int tab[40])`, ce n'est donc pas les 40 valeurs qui sont communiquées mais seulement un pointeur vers ce tableau. La déclaration de la taille dans le paramètre formel est donc inutile.

La déclaration char **tab

Signifie "**pointeur de pointeurs de caractères**"

tab	520	**720**
	...	...
	720	ptr
	722	ptr
	...	...

Remarque
- Ici nous avons un pointeur qui pointe vers zéro ou plusieurs pointeurs consécutifs (chacun d'eux pointant vers zéro ou plusieurs caractères consécutifs).
- Pour utiliser ce pointeur comme un tableau dynamique de chaînes de caractères, il faudra dans un premier temps connaître le nombre de chaînes à manipuler, et pour chaque chaîne, faire une allocation du nombre de caractères à stocker.

La déclaration char (*tab)[]

Signifie "**pointeur de tableaux de caractères**"

tab	520	**720**
	...	...
	720	ptr
	722	a
	724	b
	...	c

Remarque
- Ici le problème vient du fait que ce pointeur pointe sur zéro ou plusieurs tableaux consécutifs et que chacun contient un nombre inconnu de caractères. Dans notre exemple, avons-nous 2 tableaux de 2 caractères ou 4 tableaux de 1 caractère ?
- Pourquoi donc utiliser cette notation au lieu de char *tab ? En fait cette notation n'est effectivement utile que lorsque l'on connaît la taille des tableaux en question (ainsi dans notre exemple, une déclaration `char (*tab)[2]` nous permet de référencer c par `tab[1][0]`).

5 Les fonctions

Quelques points
- En programmation structurée, le programmeur décompose le travail à faire en modules logiques (ou "sous-programmes"). Le concept de sous-programme est directement lié à l'approche de décomposition fonctionnelle.
- Un sous-programme dans le langage C est appelé une fonction (contrairement au PASCAL où on différencie procédures et fonctions).
- Le programme principal de l'application n'est autre qu'une fonction qui doit porter le nom "main" (comme "principal" en anglais).
- Toutes les fonctions d'un programme sont définies au 1er niveau : elles ne peuvent pas être emboîtées comme en PASCAL par exemple.

5.1 La déclaration

Quelques points
- Tous les éléments, en C, doivent être déclarés avant leur utilisation. C'est également le cas des fonctions.
- La déclaration, ou prototype, permet au compilateur de vérifier que la valeur retournée est bien du même type à l'utilisation et à la définition.
- Dans le cas où la fonction n'est pas déclarée, le compilateur considère que la fonction retourne une valeur de type int.

```
/* [type retourné] nom_de_la_fonction(); */
int fct1();
float fct2();
```

Example 24.

Différence entre C et C ANSI.
Lorsque K&R ont inventé le C ils ont définis la déclaration de fonction comme précédemment expliqué. La normalisation ANSI du C a modifié cette déclaration (pour permettre notamment de vérifier, en plus du retour de la fonction, ses paramètres) :

C "K&R"	C ANSI
`int fct1();`	`int fct1(int, float);`

5.2 La définition

Définition
Une fonction doit :
- être déclarée pour être référencée,
- être définie pour être compilée,
- définir les instructions qu'elle exécutera.

```
/* [type retourné] nom_de_la_fonction([paramètres])
[déclaration et définition des paramètres]; */
{
    [liste de déclarations]
    [liste d'instructions]
}
```

Définition
- Les actions réalisées par une fonction sont regroupées dans un bloc.

- Le retour est effectué par :
  - la rencontre de la fin du corps de la fonction
  - ou par la rencontre d'une instruction `return`.

```
int plus(a,b) int a,b; {
    int c;
    c = a+b;
    return(c);
} main() {
    int i=1,j=2,sum;
    sum = plus(i,j);
}
```

Example 25.

Remarques

- Différence entre C et C ANSI. Comme pour la déclaration, la définition ANSI est différente :

C "K&R"	C ANSI
`int plus(a,b)` `int a,b;`	`int plus(int a, int b);`

- Les types pour les paramètres ou le retour des fonctions ne peuvent être que scalaires ou pointeurs.

5.3 Le passage des paramètres

Définition

L'appel d'une fonction est réalisé par l'occurrence de l'identificateur de la fonction, suivi de la liste des paramètres effectifs.

Fonction appelante

```
main(){
...
/* Appel de fonction */
sum = plus(1,2);
...
}
```

Fonction appelée

```
plus()
```

Remarque importante

- En C, il n'existe **que** le **passage par valeur**. C'est à dire que la valeur des paramètres effectifs (lors de l'appel) sont recopiés dans un emplacement local à la fonction, qui travaille avec cette copie. Il n'y a pas recopie inverse à la sortie de la fonction (en dehors du retour de la fonction).
- ne pas retourner l'adresse d'une variable créée dans une fonction, car le résultat sera indéterminé.

```
/*Quelles sont les valeurs de i, j, a et b
au fur et à mesure de l'exécution?*/

int plus(a,b) int a,b;
    int c;
    c = a+b;
    return(c);
} main(){
    int i=1,j=2,sum;
    sum = plus(i,j);
}
```

Example 26.

5.4 Le passage des paramètres par adresse

Remarque importante

- Lorsque l'on souhaite qu'une fonction modifie des paramètres effectifs, on ne va pas transmettre leur valeur mais leur adresse. Ainsi la fonction pourra accéder à la zone mémoire à modifier.
- Cette approche est également utile lorsque les paramètres à transmettre sont volumineux (paramètres structurés par exemple).

5.5 Passage d'un tableau en paramètre

Procédure

C'est ainsi que pour transmettre un tableau en paramètre, il suffit de transmettre l'adresse de son premier élément (et parfois la taille du tableau si nécessaire) pour que la fonction y accède.

```
void trier(int *tab, int taille) ; {...}
```

Example 27.