Constructions de types

Plan

1. Constructions de types
2. Prototype de fonctions
3. Pointeur vers une fonction ou la confusion des genres
4. Autres exemples
5. Fonction en paramètre

1. Constructions de types

La déclaration de variables (ou de types) complexes à partir de types plus simples se fait à l'aide des “opérateurs”

`*` `[]` `()`

et également avec `struct` et `union`. Tout type est valable sauf

- les tableaux de fonctions;
- les fonctions retournant un tableau;
- les fonctions retournant une fonction.

En revanche, il existe des tableaux de pointeurs vers des fonctions.

La règle de base est que l'opérateur `[]` déclarant un tableau et l'opérateur `()` définissant une fonction ont précédence sur l'opérateur `*` de déclaration de pointeur.

Les formes les plus fréquentes de déclarations sont

Déclaration	Type de *id*
T *id*;	*T*
T **id*;	pointeur sur le type*T*
T *id*();	fonction retournant le type *T*
T *id*[n];	tableau de n éléments de type *T*
T **id*();	fonction retournant un pointeur sur le type *T*
T (**id*)();	pointeur sur fonction retournant le type *T*
T **id*[n];	tableau de n pointeurs sur le type *T*
T (**id*)[n];	pointeur sur un tableau de n éléments de type *T*
T *id*[n][m];	tableau à deux dimensions d'éléments de type *T*

Exemple:

```
char* a[23];
```

définit un tableau `a` de 23 pointeurs sur des caractères. En effet:

```
  a[23]
```

`a` est un tableau de 23 ....

```
  *a[23]
```

`a` est un tableau de 23 pointeurs vers ....

```
  char *a[23]
```

`a` est un tableau de 23 pointeurs vers des caractères.

Chaque déclaration de variable peut donner lieu à une déclaration de type. Il suffit pour cela de préfixer l'expression de `typedef` et de remplacer la variable par le nom de type choisi.

```
typedef char* vec[23];
```

définit `vec` comme type des tableaux de 23 pointeurs sur des caractères. On peut alors écrire:

```
vec a;
```

L'*expression* de type se déduit de la déclaration en omettant le nom de la variable (ou du type dans `typedef`). Ici

```
char*[23]
```

Les expressions de type servent dans les prototypes de fonctions ou pour la coërcition de type.
On serait tenté d'écrire la déclaration de variable

```
char*[23] a; /* erreur */
```

mais ceci est illégal.

Autre exemple:

`short (*b)[N];`

`*b`

`b` est un pointeur vers ....

`(*b)[N]`

`b` est un pointeur vers un tableau de `N` ....

`short (*b)[N]`

`b` est un pointeur vers un tableau de `N` entiers courts.

Encore un exemple:

`char f(int, float);`

déclare `f` comme une fonction d'un argument entier et d'un argument flottant à résultat caractère.

2. Prototype de fonctions

Une *fonction* a une *signature* qui est la liste composée du type de la valeur de retour, et des types des paramètres.

Une fonction peut être *déclarée* une ou plusieurs fois, et doit être *définie* une et une seule fois.

La déclaration d'une fonction sert à en préciser la signature. Elle permet son utilisation *avant* sa définition, car elle permet au compilateur de vérifier la concordance syntaxique.

La déclaration se fait à l'aide d'un *prototype* qui consiste à écrire la suite des expressions des types de variables.

```
int LireCar(void);
char* LireMot(char *);
char* RelireMot(char []);
```

Les noms des paramètres formels peuvent figurer dans le prototype; ainsi

```
char* LireMot(char* message);
```

3. Pointeur vers une fonction ou la confusion des genres

L'*identificateur* d'une fonction est une constante de type: pointeur vers une fonction; sa *valeur* est l'adresse de la première instruction de la fonction en mémoire.
On peut affecter l'identificateur d'une fonction à un pointeur du bon type. Si une fonction `f` a le prototype

```
int f(int, float);
```

un pointeur `p` vers une fonction de ce type se déclare par

```
int (*p)(int, float);
```

On peut alors logiquement écrire

```
p = &f;
```

et utiliser

```
(*p)(i,x);
```

comme équivalent de

```
f(i,x);
```

La réalité est plus souple, car dans ce cas (et dans le cas des fonctions seulement), il y a confusion entre `f` et `&f`.

Voici un exemple:

```
#include <stdio.h>

  int f(int);
  int (*q)(int);
  int (*p)(int);

  int f(int i) {return 2*i;}

main() {
  int i;

  p = &f;
  q = f;
  for (i=0; i<5; i++)
    printf("%d %d, %d %d %d\n", f(i), q(i), (*q)(i),
      (*p)(i), p(i));
}
```

Ce programme compile et s'exécute sans erreur !

Pour simplifier l'écriture, on choisit en général:

- l'affectation de l'identificateur au pointeur: `q = f;`
- l'appel via le pointeur : `q(i)`.

4. Autres exemples

```
int *(*t[N])();
```

Ici

```
t[N];
```

`t` est un vecteur de N ...

```
*t[N];
```

`t` est un vecteur de N pointeurs vers...

```
(*t[N])();
```

`t` est un vecteur de N pointeurs vers des fonctions...

```
*(*t[N])();
```

`t` est un vecteur de N pointeurs vers des fonctions retournant un pointeur...

```
int *(*t[N])();
```

`t` est un vecteur de N pointeurs vers des fonctions retournant un pointeur vers un entier.

Un tableau de pointeurs vers des fonctions prenant deux entiers en argument et retournant un entier:

```
t[N];
```

`t` est un vecteur de N ...

```
*t[N];
```

`t` est un vecteur de N de pointeurs vers ...

```
(*t[N])(int, int);
```

`t` est un vecteur de N de pointeurs vers des fonctions de deux entiers ...

```
int (*t[N])(int, int);
```

`t` est un vecteur de N de pointeurs vers des fonctions de deux entiers et retournant un entier.

Quelques *expressions* de type: On les obtient en "oubliant" la variable. Pour une fonction de prototype

```
int f(int, float);
```

l'expression est

```
int (int, float)
```

Pour un pointeur déclaré

```
int (*q)(int, float);
```

l'expression est:

```
int (*)(int, float)
```

D'où vient

```
int (*[2])(int)     ?
```

Fréquemment, on utilise des `typedef` pour simplifier. Ainsi, en posant

```
typedef int (*CQF)(int, float);
```

on définit un pointeur `d`

```
int (*d)(int, float);
```

plus simplement par

```
CQF d;
```

5. Fonction en paramètre

On peut mettre en paramètre d'une fonction une autre fonction ou un pointeur vers une autre fonction.

Considérons une fonction de tri comme le tri par sélection

```
void TriSelection() {
  int i, j, min;

  for (i = 0; i < N-1; ++i) {
    min =i;
    for (j=i+1; j < N; ++j)
      if (a[j]<a[min]) min =j;
    Echange(a+min,a+i);
  }
}
```

Cette fonction trie en ordre croissant. Pour trier en ordre décroissant, on fait dépendre la fonction d'une fonction de comparaison. En voici deux;

```
int croissant (int x, int y) { return (x < y) ; }
int decroissant (int x, int y) { return (x > y) ; }
```

On remplace la fonction `TriSelection()` par

```
void TriSelection( int comp(int,int)) {
  int i, j, min;

  for (i = 0; i < N-1; ++i) {
    min =i;
    for (j=i+1; j < N; ++j)
      if (comp(a[j],a[min])) min =j;
    Echange(a+min,a+i);
  }
}
```

et on fait les deux appels

```
TriSelection(croissant);
TriSelection(decroissant);
```

ce qui donne, à partir de:

```
 70 126   1 107  75 123  98 123  84 118
```

les suites

```
  1  70  75  84  98 107 118 123 123 126
126 123 123 118 107  98  84  75  70   1
```

On aurait pu aussi bien définir

```
void TriSel(int (*comp)(int,int)) {...}
```

et appeler

```
TriSel(croissant);
TriSel(&decroissant);
```

etc.