Utilisation du langage C en programmation de systèmes

Jacques Mossière

21 septembre 2007

1 Introduction

Cette note présente quelques particularités de l'utilisation du langage C très utiles pour la programmation de systèmes. Nous ne présentons que les principes généraux et les éventuelles difficultés d'emploi seront signalées sur des exemples. Pour un traitement complet, il faut se reporter à un ouvrage sur le langage C. Celui que nous utilisons est « Introduction au langage C » par Bernard Cassagne. Dans chaque paragraphe, nous indiquons les pages ou paragraphes où la notion présentée est traitée. Une lecture attentive de l'annexe F (le bêtisier) permet d'éviter pas mal d'erreurs classiques .

2 Compilations séparées

Tout logiciel un tant soit peu complexe doit être divisé en un certain nombre de composants que nous appellerons ici des modules. En plus de la complexité, un système d'exploitation est presque toujours écrit dans plusieurs langages de programmation, par exemple en C et en assembleur.

Quand on écrit un module, il est classique de distinguer son interface, c'est à dire les éléments définis dans le module et utilisables de l'extérieur et le corps du module. Par convention, l'interface du module `toto` sera placée dans le fichier de nom `toto.h` et le corps du module dans le fichier de nom `toto.c`.

L'interface d'un module peut contenir des déclarations de constantes, de types, de variables ou de référence de fonctions. Les règles de correspondance entre les éléments définis dans un module et utilisés dans d'autres sont explicitées dans Cassagne, chapitre 9.

En simplifiant, tout se passe bien si :

- pour une fonction, on fait figurer dans l'interface l'en-tête de la fonction,
- pour une constante ou un type, l'interface contient la déclaration complète,
- pour une variable, on place la déclaration, précédée du mot clé `extern` sans initialisation.

Pour inclure dans un module l'interface d'un autre module (ou du module lui-même), on utilise l'instruction C `#include` (Cassagne p. 22). Cette instruction recopie dans le fichier en cours de compilation le texte contenu dans le fichier dont le nom figure en argument du `#include`.

L'interface d'un module doit être introduite une fois et une seule dans chaque unité de compilation qui l'utilise. Il est commode pour cela d'utiliser la commande de compilation conditionnelle `#ifndef` (Cassagne p. 140). Le texte d'un fichier d'interface `toto.h` aura ainsi la structure ci-dessous.

```
#ifndef  __TOTO_H    // ou n'importe quel nom d'utilisation peu probable
#define __TOTO_H

// les définitions de l'interface, par exemple
#define NB 25
extern int t[NB] ;
int max (int a, int b) ;

#endif
```

À la première inclusion du fichier `toto.h`, le nom `__TOTO_H`, qui n'est pas encore défini, sera défini et le contenu du fichier effectivement recopié. L'utilisation de `#ifndef` évite de nouvelles inclusions puisque le nom `__TOTO_H` a été défini au premier appel.

Toutes les variables et fonctions qui apparaissent dans l'interface d'un module doivent bien sûr être définies dans le corps de ce module.

3 Définition de constantes

Les définitions des paramètres de votre système (nombre max de processus, taille d'une pile système, nombre max de files de messages ou de sémaphores, etc.) doivent être regroupées dans un même module. De même, les définitions des constantes utilisées dans un module doivent figurer au début du module et non pas réparties dans les instructions.

Le respect de ces règles permet d'effectuer sans risque d'erreur des modifications des valeurs des paramètres.

On utilise pour définir une constante l'instruction C `#define` (Cassagne p. 15). Lorsque le compilateur a rencontré

`#define identificateur reste-de-la-ligne`

il remplacera toute occurrence d'`identificateur` par `reste-de-la-ligne`

Exemple 1

`#define NBPROC 256`

Exemple 2 Une définition de constante peut faire intervenir des expressions, par exemple :

`#define NB NBPROC + 1`

Telle quelle, cette définition présente un piège : que calcule l'expression `NB * 2` ? Le remplacement introduit par la définition conduit à l'expression `NBPROC + 1 * 2` que la priorité des opérateurs ramène à `NBPROC + 2`, ce qui n'était probablement pas la valeur attendue. Une façon d'éviter le piège est d'écrire

`#define NB (NBPROC + 1)`

4 Définition de macros

L'instruction `#define` que nous venons d'utiliser pour définir des constantes est en fait de portée beaucoup plus générale : le remplacement d'un identificateur par un texte peut être paramétré. Les macros (diminutif de macro-instructions) et leurs avantages et pièges sont présentées en détail dans Cassagne paragraphe 8.1. Nous donnons ci-dessous quelques éléments, puis nous montrerons l'utilisation que nous en avons faite pour le paquetage de gestion de files.

4.1 Principe

Une macro sans paramètre correspond à l'utilisation faite ci-dessus. La ligne

`#define identificateur reste-de-la-ligne`

conduit au remplacement de l'identificateur par le reste de la ligne. Si une ligne est insuffisante, on peut passer à la ligne suivante en plaçant un caractère backslash (\).

Une macro avec paramètre est définie par

`#define identificateur(liste-de-parametres-formels) reste-de-la-ligne`.

Elle peut être appelée à chaque utilisation d'`identificateur` suivi d'une liste de paramètres effectifs ayant la même longueur que la liste des paramètres formels. L'`identificateur` est alors remplacé par le corps de la macro dans lequel chaque occurrence d'un paramètre formel est remplacé par le paramètre effectif correspondant.

4.2 Exemple

```
#define max(a,b) ((b > a)? b : a)
```

Bien que cet exemple donne souvent le résultat attendu, il permet d'illustrer un piège des appels de macros. Soit

```
x = 2;
```

Quel est l'effet de l'instruction `y = max(1,x++);`

Cette instruction produit l'instruction C

```
y = ( (x++ > 1)? x++ : 1 )
```

et après son exécution,

```
x == 4; y ==3;
```

5 Les pointeurs

Les pointeurs sont décrits dans Cassagne, chap. 3 et 4.

5.1 Généralités

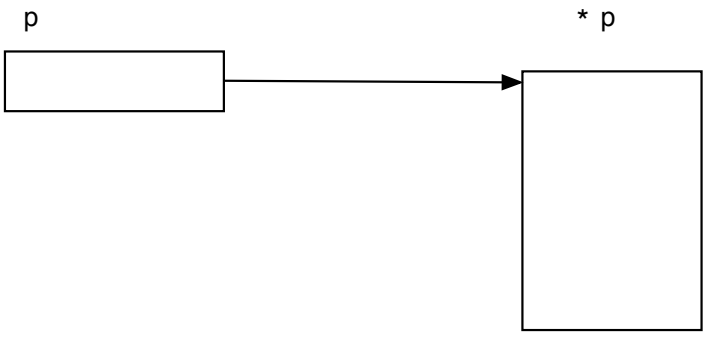

FIG. 1 – Pointeur et structure pointée

Comme dans beaucoup de langages de programmation, un pointeur C est une variable qui permet de « désigner » une autre variable ; la valeur d'un pointeur est l'adresse de la variable pointée. Lorsqu'un pointeur `p` pointe vers un élément `e`, l'élément `e` est désigné par `*p`. Si l'élément `e` est une structure comportant un champ `c`, on peut accéder à ce champ soit par `(*p).c` soit par `p->c`.

En C, un pointeur ne peut désigner qu'une variable d'un type donné, celui précisé lors de la déclaration du pointeur. Si l'élément qu'on veut désigner ne possède pas le bon type, il faut recourir à une conversion (cf. **??**).

5.2 Affectation de valeur à un pointeur

On peut affecter une valeur à un pointeur de trois façons différentes :

1. si `p` et `q` sont deux pointeurs vers un même type et si `q` a déjà reçu une valeur, alors cette valeur peut être recopiée dans `p` par une simple affectation `p = q;`
2. si une variable `v` de type `t` existe, et si `p` est un pointeur vers le type `t`, on peut placer dans `p` une valeur de pointeur vers v par l'instruction `p = &v;`
3. on peut aussi créer dynamiquement une structure (appel à la fonction `malloc`) et récupérer un pointeur vers l'élément créé. Le pointeur récupéré est du type `void *` qu'il faut transformer pour lui donner le type voulu.

5.3 Pointeurs et tableaux

Les relations entre tableaux et pointeurs sont étudiées dans Cassagne chapitre 4. Schématiquement, la déclaration `toto t[10]` est équivalente, en plus de la réservation de 10 éléments, à la déclaration de `t` comme pointeur non modifiable vers `toto` et de donner comme valeur à `t` l'adresse du premier élémént.

Il y a de même équivalence entre `t[i]` et `*(t+i)` ; cette règle permet de comprendre l'arithmétique sur les pointeurs : lorsqu'on ajoute un entier `n` à un pointeur, la valeur résultante doit pointer vers le `n`ème élément d'un tableau fictif dont l'élément d'indice 0 est pointé par `t`.

En bref, si `q` pointe vers un caractère, `q+1` augmente la valeur de `q` de 1, de 4 si `q` pointe sur un entier long, etc.

5.4 Conversion de type : cast

Dans certains langages de programmation, le type associé à un pointeur est fixe et ne peut être converti (conversion de type ou coertion) que dans un petit nombre de cas bien définis. Au contraire, on peut en C transformer pratiquement n'importe quelle expression `exp` vers une valeur de pointeur de n'importe quel type `toto` grâce à l'opérateur de conversion (appelé `cast` en C)

`(toto) exp`

Cet opérateur transforme l'expression `expr` en une expression « équivalente » de type `toto`.

5.5 Pointeur et adresse, transformation d'un entier en pointeur

Pour transformer un entier en un pointeur, il suffit d'utiliser l'opérateur de conversion que l'on vient de présenter.

En programmation de systèmes, on est souvent amené à accéder à des structures par l'intermédiaire de leur adresse (pensez au descripteur d'une zone de mémoire libre comme dans la programmation de l'allocateur du TP1).

Le type dans lequel on doit convertir l'adresse dépend de la valeur qu'on veut y ranger : si c'est un caractère ou un tableau de caractères, on utilisera le type `char *`, si c'est un entier ou un tableau d'entiers, le type `int *`, etc.

Remarque En toute rigueur, la transformation d'un entier en pointeur et inversement dépend de l'architecture de la machine dont on dispose. Ce qui est énoncé dans ce paragraphe est correct sur la plupart des machines disponibles, et en particulier sur les machines sur lesquelles nous réalisons le projet (Intel IA 32).

5.6 Exemple

Le programme ci dessous contient la définition d'un élément de liste composé d'un doublet (valeur, pointeur vers l'élément suivant) et une procédure permettant de créer un doublet et de l'insérer en tête d'une liste. A noter le pointeur de pointeur `**l` qui permet de passer à la procédure l'adresse du pointeur de tête de liste.

```
#define NULL (0)
struct elem {
        int valeur ;
        struct elem *suivant ;
}  ;
struct elem *Maliste ;
int a ;

void inserer (int x, struct elem **l)
{
struct elem *prec, *cour, *aux ;
aux = malloc (sizeof (struct elem)) ;
if (aux == NULL) erreur(...) ;  /* on admet qu'erreur stoppe l'exécution */
aux->valeur = x ;
aux->suivant = *l ;
*l = aux ;
}
```