Chapitre 5

Les directives au préprocesseur

Le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur, introduites par le caractère `#`, ont pour but :

- l'incorporation de fichiers source (`#include`),
- la définition de constantes symboliques et de macros (`#define`),
- la compilation conditionnelle (`#if`, `#ifdef`,...).

5.1 La directive #include

Elle permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h`, `math.h`,...) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :

```
#include <nom-de-fichier>
```

recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, `/usr/include/`) ;

```
#include "nom-de-fichier"
```

recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option `-I` du compilateur (*cf.* page 10).

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

5.2 La directive #define

La directive `#define` permet de définir :

- des constantes symboliques,
- des macros avec paramètres.

5.2.1 Définition de constantes symboliques

La directive

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

demande au préprocesseur de substituer toute occurence de *nom* par la chaîne de caractères *reste-de-la-ligne* dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée. Par exemple :

```
#define NB_LIGNES 10
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES * NB_COLONNES
```

Il n'y a toutefois aucune contrainte sur la chaîne de caractères *reste-de-la-ligne*. On peut écrire

```
#define BEGIN {
#define END }
```

5.2.2 Définition de macros

Une macro avec paramètres se définit de la manière suivante :

```
#define nom(liste-de-paramètres) corps-de-la-macro
```

où *liste-de-paramètres* est une liste d'identificateurs séparés par des virgules. Par exemple, avec la directive

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

le processeur remplacera dans la suite du code toutes les occurences du type

```
MAX(x,y)
```

où `x` et `y` sont des symboles quelconques par

```
(x > y ? x : y)
```

Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution.

La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante. Ainsi, si l'on écrit par erreur

```
#define CARRE (a) a * a
```

la chaîne de caractères `CARRE(2)` sera remplacée par

```
(a) a * a (2)
```

Il faut toujours garder à l'esprit que le préprocesseur n'effectue que des remplacements de chaînes de caractères. En particulier, il est conseillé de toujours mettre entre parenthèses le corps de la macro et les paramètres formels qui y sont utilisés. Par exemple, si l'on écrit sans parenthèses :

```
#define CARRE(a) a * a
```

le préprocesseur remplacera `CARRE(a + b)` par `a + b * a + b` et non par `(a + b) * (a + b)`. De même, `!CARRE(x)` sera remplacé par `! x * x` et non par `!(x * x)`.

Enfin, il faut être attentif aux éventuels effets de bord que peut entraîner l'usage de macros. Par exemple, `CARRE(x++)` aura pour expansion `(x++) * (x++)`. L'opérateur d'incrémentation sera donc appliqué deux fois au lieu d'une.

5.3 La compilation conditionnelle

La *compilation conditionnelle* a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- la valeur d'une expression
- l'existence ou l'inexistence de symboles.

5.3.1 Condition liée à la valeur d'une expression

Sa syntaxe la plus générale est :

```
#if  condition-1
   partie-du-programme-1
#elif  condition-2
   partie-du-programme-2
     ...
#elif  condition-n
   partie-du-programme-n
#else
   partie-du-programme-∞
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque ***condition-i*** doit être une expression constante.

Une seule ***partie-du-programme*** sera compilée : celle qui correspond à la première ***condition-i*** non nulle, ou bien la ***partie-du-programme***-∞ si toutes les conditions sont nulles.

Par exemple, on peut écrire

```
#define PROCESSEUR ALPHA
```

```
#if PROCESSEUR == ALPHA
  taille_long = 64;
#elif PROCESSEUR == PC
  taille_long = 32;
#endif
```

5.3.2 Condition liée à l'existence d'un symbole

Sa syntaxe est

```
#ifdef  symbole
   partie-du-programme-1
#else  condition-2
   partie-du-programme-2
#endif
```

Si *symbole* est défini au moment où l'on rencontre la directive `#ifdef`, alors *partie-du-programme-1* sera compilée et *partie-du-programme-2* sera ignorée. Dans le cas contraire, c'est *partie-du-programme-2* qui sera compilée. La directive `#else` est évidemment facultative.

Da façon similaire, on peut tester la non-existence d'un symbole par :

```
#ifndef  symbole
   partie-du-programme-1
#else  condition-2
   partie-du-programme-2
#endif
```

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

```
#define DEBUG
  ....
#ifdef DEBUG
  for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif /* DEBUG */
```

Il suffit alors de supprimer la directive `#define DEBUG` pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation `-D`*symbole*, qui permet de définir un symbole. On peut remplacer

```
#define DEBUG
```

en compilant le programme par

```
gcc -DDEBUG fichier.c
```