Document rédigé d'après : Méthodologie de la programmation en C, Jean-Pierre Braquelaire, Éditions Dunod, Paris, 2000. ISBN 2 10 004782 5 [1]

Macro constante : une macro-constante est un symbole, défini à l'aide de la directive **define** et dont la valeur est une chaîne de caractères :

1) définition dans les sources

Syntaxe : **#define IDENTIFICATEUR chaine de substitution**

Règles d'écriture : L'identificateur et la chaîne de substitution doivent être séparés par au moins un séparateur (espace ou tabulation)

La chaîne peut contenir n'importe quel caractère, en lettres majuscules uniquement. Le caractère _ est considéré comme une lettre.

note: les minuscules peuvent être utilisées, mais fortement déconseillées, car sources d'erreurs. Par ailleurs, la lisibilité du code diminue si on les utilise pour définir des macro-constantes.

Cette chaîne se termine sur le caractère '\n' de fin de ligne, **sauf** si le caractère précédant '\n' est le caractère ' \ ' (anti-slash, seul). Dans ce cas, la chaîne se poursuit sur la ligne suivante.

Fonctionnement : durant le pré-traitement d'un fichier, le préprocesseur recherche dans le fichier source, pour chaque macro-constante, les occurrences exactes de l'identificateur, et remplace celles qui lui sont **identiques** par la chaîne de substitution correspondante.

Exemple :

#define TAILLE_MOT_STANDARD 20

-> lors de la première étape de compilation, le préprocesseur va remplacer toutes les occurrences de TAILLE_MOT_STANDARD par la chaine de caractères "20".

Si une chaine de caractères (un autre identificateur) contient TAILLE_MOT_STANDARD, par exemple TAILLE_MOT_STANDARD_OS_1 , celle-ci n'est pas concernée par la substitution, car la longueur de cette occurrence est différente.

Attention : l'intérêt des macros-constantes est de permettre d'augmenter la modularité et la lisibilité d'un programme. Les utiliser quand ce n'est pas utile nuit à la qualité du code produit.

Erreurs classiques :

-> utiliser des minuscules dans le nom d'une macro-constante
- confusion possible si une variable porte le même nom
- la lisibilité diminue

-> le point virgule :-)

Exemple :

#define TMAX 100;
...
provoquera la réécriture de la ligne : « int tableau[TMAX]; » en : « int tableau[100;]; »

2) Définitions à la compilation

Avec gcc, on peut définir des macro-contantes pendant la phase de compilation, à l'aide de l'option -DXXX ou encore -DYYY=valeur (les deux possibilités sont utilisables).

Exemple : gcc -g -DNDEBUG main.c l.c -L/usr/lin -I/usr/include -lstdc++ ...etc
équivaut à ajouter la ligne :

#define DEBUG dans les 2 fichiers main.c **et** l.c

3) Macro-constantes pré-definies :
Cinq macro-constantes sont définies par la norme ISO :

__FILE__ : constante chaîne de caractères contenant le nom du fichier source courant

__LINE__ : constante entière valant le numéro de la ligne courante

__DATE__ : constante chaîne de caractère contenant la date de la compilation avec le format : Mmm jj aaaa

__TIME__ : constante chaîne de caractères contenant l'heure de la compilation
avec le format hh:mm:ss.

__STDC__ : constante entière qui vaut 1 lorsque le conpilateur se conforme à la norme

Exemple avec linefile.c :

```
#include <stdio.h>
#include <stdlib.h>

int
main (void)
{
   printf (« ligne %d du fichier \ "%s\ "\n", __LINE__ , __FILE__ );
   return EXIT_SUCCESS ;
}
```

Affichera les valeurs numéro de ligne et nom du fichier linefile.c.
Application : dans le cas d'un traitement d'erreur, ou d'une exception, dans un programme.

4) Macro-fonctions

Le mécanisme de substitution de la directive define est paramétrable :

#define identificateur(parametre) corps

Exemple avec la macro-fonction « valeur absolue » :

```
#define ABS(x) x>0 , x : - x;
```

Signifie que ABS(mot) , vaudra mot si mot est défini positif, et -mot sinon (en supposant que mot soit réel ou entier... un nombre quoi...).

Attention : les substitutions peuvent provoquer une multiplication du code et des calculs effectués, ce qui peut entraîner des erreurs dans le cas d'expressions à effet de bord.

La structure syntaxique des expressions peut être altérée.

Exemple :
ABS(x+y) sera interprété x+y > 0 ? x+y : -x+y

Ce qui n'est pas le résultat attendu..

Règles utiles à connaître :

Dans une définition de macro-fonction, toute occurrence d'un paramètre dans le corps de la définition doit être placé entre parenthèses. Lorsque le corps de la macro-fonction définit une expression C, il est lui même écrit entre parenthèses.

Il faut éviter d'utiliser une expression à effet de bord (comme x++) en paramètre d'une macro-fonction.

De façon générale, dès lors qu'il y a duplication de code, une erreur apparaît.

5) Génération de chaînes

Le préprocesseur permet de transformer un paramètre de macro-fonction en chaîne de caractères. Les caractères " et \ peuvent aussi être utilisés, mais doivent être échappés avec l'anti-slash ' \ '.

Syntaxe : l'identificateur de ce paramètre doit être précédé du caractère #

Exemple :

```
#define IPRINT(x) printf("Le résultat de %s vaut %d\n", #x, x);
```

Ainsi, IPRINT(2*3) est réécrit : printf("Le résultat de %s vaut %d\n", "2*3", 2*3);

Voir **[1]** pour des exemples plus complexes.

6) Concaténation de paramètres

À partir d'une macro-fonction, il est possible de former un mot.
L'opérateur de concaténation est ##

À gauche :

chaîne_de_début (espace) ##

Au milieu (chaîne vide comprise) :

..... chaîne .....

À droite :

(espace) chaîne_de_fin

Note : (espace) désigne le caractère espace utilisé comme séparateur ici.

Exemple :

#define COMPOSE(x,y) x ## _ ## y (x, y)

Génère l'écriture du motif x = COMPOSE(rouge, vert) en

x = rouge_vert (rouge, vert)

7) Critères de choix entre fonction ou macro-fonction

Historiquement, les macro-fonctions utilisées à la place de fonctions ont permis d'utiliser moins de ressources, et donc d'obtenir des programmes plus rapides à l'exécution.

Ceci est de moins en moins vrai. Les cas dans lesquels il est préférable d'utiliser une macro-fonction à une fonction sont énumérés ci-après :

- manipulation de constructions complexes
- le passage de paramètres par noms (ie cas de la construction va_start)
- utilisation de fonctionnalités spécifiques au préprocesseur (concaténation...)
- utilisation de costructions du préprocesseur (macro constantes prédéfinies)

8) Récursivité des définitions

Le mécanisme de réécriture du préprocesseur est linéaire et récursif. Après chaque réécriture, celui-ci examine à nouveau la chaîne obtenue pour y rechercher d'éventuelles nouvelles substitutions.

Exemple avec le fichier recdef.c :

```
#define NOMBRE_MAX 256
#define TAILLE_MAX (NOMBRE_MAX*sizeof(int))

#define RANG(n)      ((float)(n)/(float)TAILLE_MAX)

RANG(i+1);
```

À la compilation :

```
eric@tomate:~$ gcc -E -P recdef.c
((float)(i+1)/(float)(256*sizeof(int)));
eric@tomate:~$
```

C'est bien ce qui était attendu...

9) Inclusion conditionnelle

Les directives de compilation conditionnelles sont : if, ifdef, ifndef, elif, else et endif
Ces directives permettent de tester l'existence et la valeur de macro-constantes.

Syntaxe à connaître absolument :

```
#ifdef identificateur
        alors faire
#elif
        si autre cas faire
#else
        faire pour tous les autres cas
#endif
```

Qui peut se simplifier en

```
#ifdef identificateur
        alors faire
#else
        faire pour tous les autres cas
#endif
```

Voire même :

```
#ifdef identificateur
        faire
#endif
```

Note : ifndef teste si l'identificateur n'existe pas.

10) Évaluation de macro-expressions

Il est possible d'utiliser des expressions interprétables par le préprocesseur. Ces expressions sont des expressions constantes restreintes, contstruites au moyen :

- de constantes caractère ou entière
- de parenthèses
- d'opérateurs unaires comme - ! et ~ ,
- d'opérateurs binaires comme +, - , * / , %, | , << , >>, > , < , <= , >= , == , != , && et ||
- de l'opérateur conditionnel ?
- de conversions vers les types autorisés
- de l'opérateur unaire **defined**

Syntaxe : #if defined **macro-constante**

-> Cette expression est vraie (vaut1) si macro-constante est définie 0 sinon.

Exemple :

```
#if defined(MACOSX)
#define      isnan( x )        ( ( sizeof ( x ) == sizeof(double) ) ?    \
                          __isnand ( x ) :                          \
                            ( sizeof ( x ) == sizeof( float) ) ?      \
                          __isnanf ( x ) :                          \
                          __isnan  ( x ) )
#else
          // isNan is not needed

#endif
```

Remarque : noter les \ en fin de ligne, qui permettent d'aller à la ligne sans terminer l'instruction.