Le préprocesseur (1)

- traite toutes les lignes de code commençant par # avant la compilation
  - les macro-définitions (#define)
  - les inclusions de fichiers (#include)
  - les compilations conditionnelles (#if, #else, ...)

la macro define

- syntaxe

  #define *<identificateur>* *<expression>*

- substitution de toutes les occurrences de *<identificateur>* par la chaîne *<expression>*

- exemples : #define N 10

  # define M (5 + 1)

  ⇒ A = N + M est remplacé par A = 10 + (5 + 1)

- <u>remarque</u> :

  #undef *<identificateur>*

  supprime la macro définition de *<identificateur>*

macro-définitions paramétrées (1)

- permet d'exprimer des fonctions simples sous forme de macro
- remarque : définition de constante = macro-définition sans paramètres
- style : par convention, nom d'une macro paramétrée précédé du caractère '_'
- ex : _AuCarre(x) ⇒ macro-définition

  AuCarre(x) ⇒ fonction
- exemple :
  - macro-définition :

    #define _AuCarre(x) ((x)*(x))
  - fonction :

    int AuCarre (int x)
    { return (x*x); }

macro-définitions paramétrées (2)

```
main( )
   {
      int y, z, x = 2;

      y = _AuCarre(x + 1);          /* remplacé par y = ( (x+1) * (x+1)) */
      z = AuCarre(x + 1);
   }
```

macro-définitions - utilisation (1)

- macro-définition ⇒ substitution par le préprocesseur
- fonction ⇒ pendant l'exécution :
  - création du contexte d'exécution
  - transfert des paramètres
  - destruction du contexte

  ⇒ substitution plus rapide ....

  mais fonctions parfois plus efficaces (redondance de calcul)
- macro-définition
  - plusieurs substitutions ⇒ plusieurs compilations du même code
  - augmente taille exécutable

macro-définitions - utilisation (2)

- attention aux effets de bord !

  ex : y = _AuCarre(++x);

  remplacé par y = ((++x) * (++x))

- attention au parenthésage

  ex : #define _AuCarre(x) x*x

  y = _AuCarre(x+1) ⇒ y = x + 1 * x + 1

  y = 2 / _AuCarre(x) ⇒ y = 2 / x * x

  ⇒ conseil : parenthéser complètement l'expression

macro-définitions - utilisation (3)

- macro-définition : pas d'information de type ; simple mécanisme de substitution

  ⇒ une macro peut être utilisée avec des arguments de types différents

  ⇒ comportement de procédures génériques

- fonction : vérification de la concordance de type entre paramètres formels et paramètres effectifs par le compilateur

Macro définitions : exemples d'utilisation (1)

- exemple 1 :

```
main()
{   int x = 2, y;
    float z = 3.4, t;
    y = _AuCarre(x);
    t = _AuCarre(z);
}
```

Macro définitions : exemples d'utilisation (3)

- exemple 2 :

```
#define _Allouer(type) ( (type *) malloc(sizeof(type)))
typedef struct cell
    {
        int val;
        struct cell *suiv;
    }cellule, *pcellule;
main ()
{
    pcellule p;
    p = _Allouer(cellule);
}
```

Macro-définitions - Conclusions

- Faire une utilisation efficace et intelligente des macros
- attention à
  - lisibilité des programmes
  - efficacité des programmes (calculs redondants)
  - taille mémoire

la macro include

- #include "référence"

  ⇒ recherche dans le catalogue de travail puis dans /usr/include

- #include <référence>

  ⇒ inclusion de /usr/include/référence

la compilation conditionnelle (1)

- directives permettant de créer des fichiers exécutables différents à partir du même source
- 3 conditions possibles
  - #if *<expression constante>*
  - #ifdef *<identificateur>* : teste si *<identificateur>* a été préalablement défini
  - #ifndef *<identificateur>* : vrai si *<identificateur>* n'a pas été préalablement macro-défini

la compilation conditionnelle (2)

- 2 formes possibles
  - #if ...
    suite1
    #else
    suite2
    #endif

    #if ...
    suite
    #endif
- condition vérifiée ⇒
  - forme1 : suite1 compilée, suite2 non compilée
  - forme2 : suite compilée

la compilation conditionnelle - exemples (1)

- un seul programme source pour obtenir des binaires différents : le préprocesseur peut éliminer certaines lignes de code
  - exemple : retirer ou insérer certains énoncés d'impression de valeurs qui ne servent que lors de la mise au point du programme

```
/* #define DEBUG */
#ifdef DEBUG
   printf("fonction max : a = %d, b = %d\n", a, b);
#end
```

- offre facilité de création de différents exécutables à partir du même source
- assure une cohérence : modification du fichier source ⇒ modification de tous les exécutables