Programmation en langage C

Plan

1. Rappels (Types de bases, variables, opérateurs, structures de contrôles,..)
2. Les fonctions
3. Les pointeurs
4. L'allocation dynamique
5. Les chaines de caractères
6. Les types composés (structures, énumérations,…)
7. Les fichiers
8. Les préprocesseurs

Le préprocesseur

- Le préprocesseur est un programme qui analyse un fichier texte et qui lui fait subir certaines transformations. Ces transformations peuvent être l'**inclusion d'un fichier**, **la suppression** d'une zone de texte ou **le remplacement** d'une zone de texte.

- Le préprocesseur effectue ces opérations en suivant des ordres qu'il lit dans le fichier en cours d'analyse.

- Il est appelé automatiquement par le compilateur, **avant la compilation**, pour traiter les fichiers à compiler.

Le préprocesseur

Le préprocesseur a trois grands rôles :

- **réaliser des inclusions** (la directive #include) ;
- **définir des macros** qui sont des substituts à des morceaux de code. Après le passage du préprocesseur, tous les appels à ces macros seront remplacés par le code associé ;
- **permettre la compilation conditionnelle**, c'est-à-dire de moduler le contenu d'un fichier source suivant certaines conditions.

Le préprocesseur

1. **Les inclusions**
2. **Les constantes de compilation**
3. **Les macros**
4. **La compilation conditionnelle**

Inclusion de fichier

- L'inclusion de fichier permet de factoriser du texte commun à plusieurs autres fichiers (par exemple des déclarations de type, de constante, de fonction, etc.).

- Le texte commun est mis en général dans un fichier portant l'extension .h (pour « header », fichier d'en-tête de programme).

Inclusion de fichier

- Pour inclure un fichier .h se trouvant dans le dossier où est installé votre IDE, vous devez utiliser les chevrons < > :

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

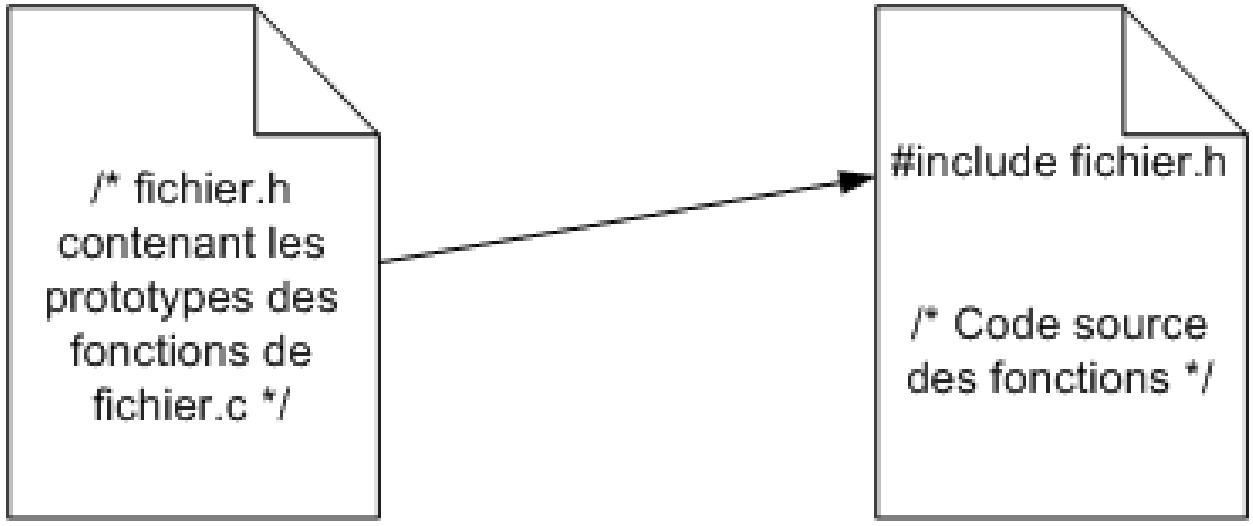
- Pour inclure un fichier .h se trouvant dans le dossier de votre projet, vous devez utiliser les guillemets :

```
#include "monfichier.h"
```

Inclusion de fichier

Le fichier inclus est traité lui aussi par le préprocesseur :

1. Le préprocesseur est démarré avant la compilation.
2. Il parcourt tous les fichiers à la recherche de directives de préprocesseur (les lignes qui commencent par un #).
3. Lorsqu'il rencontre la directive #include, il insère le contenu du fichier indiqué à l'endroit du #include.



Inclusion de fichier

fichier.c

```
#include "fichier.h"
int maFonction(int a, double b)
{
    /* Code de la fonction */
}
void autreFonction(int valeur)
{
    /* Code de la fonction */
}
```

fichier.h

```
int maFonction(int a, double b);
void autreFonction(int valeur);
```

fichier.c
juste avant la compilation

```
int maFonction(int a, double b);
void autreFonction(int valeur);

int maFonction(int a, double b)
{
    /* Code de la fonction */
}
void autreFonction(int valeur)
{
    /* Code de la fonction */
}
```

Le préprocesseur

1. **Les inclusions**
2. **Les constantes de compilation**
3. **Les macros**
4. **La compilation conditionnelle**

Les constantes de compilation

- La directive **#define** : permet de définir une constante de préprocesseur. Cela permet d'associer une valeur à un mot.

```
#define identificateur valeur
```

- le #define remplace dans le code source tous les mots par leur valeur correspondante.

Exemple :

```
#define NOMBRE 4
```

Les constantes de compilation

Note 1 : Il faut distinguer entre les constantes de compilation définies avec *#define* et les constantes définies avec le mot clé *const* (**const int NOMBRE = 4**).

- Les constantes de préprocesseur ne réservent pas de mémoire. Ce sont des valeurs immédiates, définies par le compilateur.

- Les variables de la classe de stockage *const*, occupent de la place en mémoire, même si leur valeur ne change pas.

Les constantes de compilation

Note 2 : les #define sont généralement placés dans des .h, à côté des prototypes.

stdlib.h

```
23
24  /*
25   * RAND_MAX is the maximum value that may be returned by rand.
26   * The minimum is zero.
27   */
28  #define RAND_MAX    0x7FFF
29
30  /*
31   * These values may be used as exit status codes.
32   */
33  #define EXIT_SUCCESS    0
34  #define EXIT_FAILURE    1
35
```

Les constantes de compilation

Un define pour la taille du tableau

- On utilise souvent les define pour définir la taille des tableaux.

Exemple :

```
#define TAILLE_MAX 1000
int main()
{
    char chaine[TAILLE_MAX];
    Int tab[TAILLE_MAX];
```

Remarque : on ne peux pas mettre de variable ni de constante entre les crochets lors d'une définition de tableau, mais TAILLE_MAX n'est PAS une variable ni une constante

Les constantes de compilation

Calculs dans les define

- Il est possible de faire des petits calculs dans les define.

Exemple :

```
#define LARGEUR 80
#define HAUTEUR 60
#define SURFACE (LARGEUR * HAUTEUR)
```

Remarques :

- Par sécurité il faut mettre le calcul entre parenthèses pour bien isoler l'opération.
- Toutes les opérations de base sont possibles : (+), (-), (*), (/) et (%).

Les constantes de compilation

Les constantes prédéfinies

- Il existe quelques constantes prédéfinies par le préprocesseur. Elles commencent et se terminent par deux symboles underscore _

__LINE__ : donne le numéro de la ligne actuelle.
__FILE__ : donne le nom du fichier actuel.
__DATE__ : donne la date de la compilation.
__TIME__ : donne l'heure de la compilation.

Les constantes de compilation

Les constantes prédéfinies

- Les constantes prédéfinies peuvent être utiles pour gérer des erreurs.

Exemple :

```
printf("Erreur a la ligne %d du fichier %s\n", __LINE__, __FILE__);
printf("Ce fichier a ete compile le %s a %s\n", __DATE__, __TIME__);
```

Erreur a la ligne **9** du fichier **main.c**
Ce fichier a ete compile le **Apr 11 2017** a **16:21:10**

Les constantes de compilation

Les constantes simple

- Il est aussi possible de définir une constante sans préciser sa valeur.

Exemple : `#define CONSTANTE`

Le préprocesseur

1. Les inclusions
2. Les constantes de compilation
3. Les macros
4. La compilation conditionnelle

Les macros

- Le #define permet aussi de remplacer un mot par un code source tout entier. On dit alors qu'on crée **une macro**.

Une macro est constituée des éléments suivants.

1. La directive #define
2. Le nom de la macro
3. Une liste optionnelle de paramètres
4. La définition, c’est-à-dire le code par lequel la macro sera remplacée

Les macros

Les macros sans paramètres

Exemple :

```
#define MESSAGE() printf("Ceci est un message");
int main()
{
    MESSAGE()
    return 0;
}
```

Avant la compilation

```
int main()
{
    printf("Ceci est un message");
    return 0;
}
```

Au cours de la compilation

Les macros

Les macros sans paramètres

Pour mettre plusieurs lignes de code à la fois, il suffit de placer un **** avant chaque nouvelle ligne.

```
#define BONJOUR()   printf("BONJOUR\n"); \
                    printf("TOUS LE MONDE\n");
int main()
{
   BONJOUR()
   return 0;
}
```

Les macros

Les macros avec paramètres

Exemple 1:

```
#define MAJEUR(age) if (age >= 18) \
                        printf("Vous etes majeur\n");\
                        else\
                        printf("Vous etes mineur\n");

int main()
{
   MAJEUR(22)
   return 0;
}
```

Les macros

Les macros avec paramètres

Exemple 2:

```
#define EUR(x) ((x) * 11)
#define DH(x) ((x) / 11)
int main(void)
{
    printf("200 euros valent %f Dhs.\n", EUR(200));
    printf("200 Dhs valent %f euros.\n", DH(200));
    return 0;
}
```

Les macros

Priorité des opérations

Exemple

```
#define MUL(a, b) (a * b)
int main(void)
{
    printf("%d" , MUL(2+3, 4+5));
    return 0;
}
```

2 + 3 * 4 + 5 = 19

```
#define MUL(a, b) ((a) * (b))
int main(void)
{
    printf("%d" , MUL(2+3, 4+5));
    return 0;
}
```

(2 + 3) * (4 + 5) = 45

Les macros

Manipulation des chaines de caractères dans les macros

Tout argument de macro peut être transformé en chaîne de caractères dans la définition de la macro s'il est précédé du signe #.

```
#define CHAINE(s) #s
```

CHAINE(2+3)
devient : "2+3"

Les macros

Manipulation des chaines de caractères dans les macros

Le préprocesseur permet également la concaténation de texte grâce à l'opérateur **##**. Les arguments de la macro qui sont séparés par cet opérateur sont concaténés (sans être transformés en chaînes de caractères).

```
#define NOMBRE(chiffre1,chiffre2) chiffre1##chiffre2
```

NOMBRE(2,3)
est remplacé par le nombre décimal *23*

Le préprocesseur

1. Les inclusions
2. Les constantes de compilation
3. Les macros
4. La compilation conditionnelle

La compilation conditionnelle

- Il est possible de réaliser des conditions en langage préprocesseur.
- Le préprocesseur dispose de cinq directives conditionnelles : **#if, #ifdef, #ifndef, #elif et #else.** Ils permettent de conserver ou non une partie du code en fonction de la validité d'une condition.
- Chacune de ces directives (ou suite de directives) doit être terminée par une directive **#endif**.

La compilation conditionnelle

Les directives #if, #elif et #else

```
#if condition1
/* Code source à compiler si la condition 1 est vraie */
#elif condition2
/* Sinon si la condition 2 est vraie compiler ce code source */
...
#elif conditionN
/* Code source à compiler si la condition N est vraie */
#else
/* Code source à compiler */
#endif
```

La compilation conditionnelle

Les directives #if, #elif et #else

- les conditions ne peuvent comporter **que des expressions *entières*** (pas les nombres flottants et les pointeurs) **et *constantes***. Aussi, les mots-clés et autres identificateurs (sauf les macros) présent dans la définition *sont ignorés* (plus précisément, ils sont remplacés par 0).

```
#if 1.89 > 1.88
/* Incorrect, ce ne sont pas des entiers */

#if sizeof(int) == 4
 /* Équivalent à 0(0) == 4, `sizeof' et `int' étant des mots-clés */

#if (a = 20) == 20
/* Équivalent à (0 = 20) == 20, `a' étant un identificateur */
```

La compilation conditionnelle

#ifdef, #ifndef

- il est possible d'utiliser **#ifdef** (c'est-à-dire « if defined ») pour dire **« Si la constante est définie »** et **#ifndef**, **« Si la constante n'est pas définie »**

```
#define WINDOWS

#ifdef WINDOWS
   /* Code source pour Windows */
#endif

#ifdef LINUX
   /* Code source pour Linux */
#endif
```

La compilation conditionnelle

L'opérateur defined

- Le préprocesseur fourni un opérateur supplémentaire utilisable dans les condtions : **defined**. Il ne peut être utilisé que dans le contexte d'une commande #if ou #elif.
- Il peut être utilisé sous l'une des deux formes suivantes : defined *nom* ou bien : defined (*nom*).
- Il délivre la valeur 1 si *nom* est une macro définie, et la valeur 0 sinon.
- L'intérêt de cet opérateur est de permettre d'écrire des tests portant sur la définition de plusieurs macros, alors que #ifdef ne peut en tester qu'une.

Exemple :

```
#if defined TAILLE
#if defined(SYS) || defined(SYSV)
```

Le préprocesseur

Autres commandes

Le préprocesseur est capable d'effectuer d'autres actions à l'aide de d'autres directives tel que :

- **# :** ne fait rien (directive nulle) ;
- **#error message :** permet de stopper la compilation en affichant le message d'erreur donné en paramètre ;
- **#line numéro [fichier] :** permet de changer le numéro de ligne courant et le nom du fichier courant lors de la compilation ;