TD 1 de langage C

Les instructions du pré-processeur en C

Une particularité propre au compilateur C est qu'il ne procède pas directement sur le code source fournit par le programmeur. Une phase spéciale de réécriture du programme précède toute compilation. L'utilitaire se chargeant de cette phase de pré-traitement se nomme le pré-processeur.

Macrogénération de code

Le langage C possède un moyen de réaliser de la macrogénération de code. Il est clair que cette possibilité est un atout indéniable pour un langage. Cependant, en C, le moyen d'y parvenir peut troubler certains puristes de l'Informatique. En effet, Cette macrogénération ne peut s'effectuer que durant la phase du pré-processeur. Il apparaît donc que ce mécanisme ne fait pas partie intégrante du langage.

L'instruction #define

L'instruction du préprocesseur (et donc pas réellement C) qui permet cette possibilité se nomme **#define** (attention en C il y a une différence entre minuscules et majuscules). On remarquera que cette instruction commence par un dièse, comme toutes les autres instructions du préprocesseur.

Cette instruction (ou cette directive, ce terme étant aussi employé) permet de définir soit une macro constante, soit une macro paramétrée. L'exemple qui suit vous expliquera mieux les choses. Sachez juste que l'instruction doit être suivie du nom de la macro (avec éventuellement des paramètres) puis de sa valeur de remplacement, les trois parties étant juste séparées par des espaces.

Définition	Exemple d'utilisation

`#define False 0`
`#define True 1`
`#define max(a,b) (a>=b?a:b)`
`#define min(a,b) (a<=b?a:b)`
	`if (var==False) return max(var,autreVar)`
	`else return min(var,encoreAutreVar);`
	sera remplacé par

Attention : le programme n'est pas compilé durant le traitement du pré-processeur. Si des erreurs existent, elles seront toujours présentes après la phase de pré-processing. Le pré-processeur ne fait que réécrire le programme selon des règles que vous précisez.

Remarques :

- la définition de macros est récursive `max(min(a,b),c)` produira `((a<=b?a:b)>=c?(a<=b?a:b):c)` et non pas `(min(a,b)>=c?min(a,b):c)`
- la norme ANSI dit que si il y a redéfinition de la macro, alors il y a remplacement de la définition. Certains pré-processeurs signalent tout de même la redéfinition par un petit message. Certains autres pré-processeurs admettent même une pile de définition, permettant ainsi de restituer les anciennes définitions lors de l'utilisation de l'instruction #undef (mais l'existence d'une telle pile de définitions ne fait pas partie de la norme ANSI).

L'instruction #undef

Cette instruction permet d'annuler une définition de macro. Il suffit juste de spécifier le nom de la macro à invalider et ce directement après l'instruction.

Avant le préprocesseur	Après le préprocesseur

`#define a 10`
`a + b;`
`#undef a`
`a + c;`	

Les macros prédéfinies

Un certain nombre de macros sont prédéfinies. Ces macros ne peuvent en aucun cas être supprimées ou altérées. Elles sont donc immuables. Le tableau suivant vous donne le nom de quelques macros ainsi que leur développement.

Nom des macros	Développement de la macro associée.
__LINE__	Se développe en une valeur numérique associée au numéro de la ligne courante dans le code du programme.
__FILE__	Cette macro sera remplacée par le nom du fichier en cours de traitement.
__DATE__	Celle-ci se transformera en une chaîne de caractères contenant la date de traitement du fichier sous un format "Mmm jj aaaa".
__TIME__	De même, se transformera en un chaîne représentant l'heure de traitement du fichier sous le format "hh:mm:ss".
__STDC__	Cette macro n'est censée être définie uniquement que dans les implémentations respectant la norme ANSI. Si c'est le cas, elle doit de plus valoir 1.
Attention : les deux caractères _ (*underscore*) font partie du nom de la macro	

Inclusion de fichiers

L'instruction **#include** permet d'inclure un fichier dans celui en cours de traitement. Cette instruction prend en unique paramètre le nom du fichier à inclure. Cette inclusion est ponctuelle : la totalité du fichier à inclure se retrouve donc entre la ligne précédent l'instruction d'inclusion et la suivante. Le compilateur, lui, n'aura plus qu'à traiter un gros fichier résultant de la fusion des deux fichiers initiaux.

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

Il vous est possible d'entourer le nom du fichier par les signes < >, plutôt que par des guillemets. Le fichier sera alors recherché uniquement dans les répertoires contenants les fichiers standard fournis avec le compilateur. Dans le cas de l'exemple le fichier est cherché d'abord dans le répertoire courant dans lequel vous vous trouvez, et si le fichier n'existe pas, alors le pré-processeur cherchera dans les répertoires standard.

Compilation conditionnelle

Afin d'introduire une portion de code dans laquelle les données peuvent varier selon un critère particulier, il vous faut utiliser une des trois instructions commençant par *if* (**#if, #ifdef, #ifndef**). Il vous faut aussi marquer la fin de cette zone en utilisant l'instruction **#endif**. Dès lors la section de code intermédiaire sera laissée telle quelle uniquement si la condition est remplie, sinon la section ne sera pas prise en compte dans le fichier résultant.

L'instruction **#if** demande en paramètre une expression constante. Notez que l'opérateur **defined** *identifier* retourne la constante 1 si la variable est définie en temps que macro, 0 dans tout autre cas. Pour les deux autres instructions, elles peuvent se définir à partir de la première. Le tableau suivant vous donne les correspondances.

`#if defined` *identifier*	`#ifdef` *identifier*
`#if ! defined` *identifier*	`#ifndef` *identifier*

Correspondances entre les directives **#if, #ifdef et #ifndef**

Notez que dans l'exemple précédent, le **!** est un opérateur du pré-processeur qui renvoie 1 si son opérande est 0 et renvoie 0 dans tous les autres cas.

Avant le préprocesseur	Après le préprocesseur

`printf("Ligne 1\n");`
`#ifdef Toto`
`printf("Ligne 2\n");`
`#endif`
`printf("Ligne 3\n");`	

Les instructions #else et #elif

Les instructions **#else** et de **#elif** permettent de spécifier du code quand le test n'est pas validé. La différence, entre les deux instructions, tient dans le fait que **#elif** permet de spécifier un nouveau test qui, s'il est vérifié, sélectionnera une nouvelle section de code.

Avant le préprocesseur	Après le préprocesseur

`#define tata`
`printf("Ligne 1\n");`
`#ifdef Toto`
`printf("Ligne 2\n");`
`#elif defined tata`
`printf("Ligne 3\n");`
`#endif`
`printf("Ligne 4\n");`	

Il peut y avoir autant de **#elif** que vous le souhaitez, le **#else** ne pouvant apparaître qu'une seule fois. Par contre, seule une condition vérifiée sera acceptée, même si par la suite d'autres conditions pourraient l'être.

Exercices

Exercice 1 :

On dispose des fichiers sources suivants qui définissent une collection de modules constituant un programme appelé `analyseur`.

Expliquer le fonctionnement des directives `#include` et `#ifdef ... #endif`, dans les fichiers `".h"`.
Donner le résultat du traitement du pré-processeur sur chacun des fichiers `".c"`.

analyseur.h

```
#ifdef MAIN
 char gLigne [1024];
 int SuiteSymbol [1024];
 int ErrLexicale = 0;
#else
 externe char gLigne [];
 extern int SuiteSymbol [];
 extern int ErrLexicale;
#endif
```

lex.h

```
#ifdef LEX
#else
 extern void AnalyseLexicale (void);
#endif
```

syntax.h

```
#ifdef SYNTAX
#else
 extern int AnalyseSyntaxique (int *);
#endif
```

syntax.c

```
#define SYNTAX 0
#include "analyseur.h"
#include "syntax.h"

int AnalyseSyntaxique (int *i)
{
...
}

static int Instruction (int *i)
{
...
}

static int Expression (int *i)
{
...
}

static int Facteur (int *i)
{
...
}
...
```

lex.c

```
#define LEX 0
#include "analyseur.h"
#include "lex.h"

void AnalyseLexicale ()
{
...
}
```

analyseur.c

```
#define MAIN 0
#include "analyseur.h"
#include "lex.h"
#include "syntax.h"

main ()
{
 for (;;) {
  ...
  AnalyseLexicale ();
  if (!ErrLexicale) {
   ...
   if (AnalyseSyntaxique (&i))
   ...
  }
 }
}
```

Exercice 2 :

Définir les macros suivantes :

- valeur absolue d'une expression
- minimum de deux expressions
- maximum de deux expressions
- minimum de trois expressions
- maximum de trois expressions
- minimum en valeur absolue de deux expressions (en utilisant valeur absolue d'une expression et en ne l'utilisant pas)
- maximum en valeur absolue de trois expressions (en utilisant valeur absolue d'une expression et en ne l'utilisant pas)

Exercice 3 :

On dispose des deux fichiers suivant qui permettent de calculer les racines d'une équation du second degré. Ajouter à ces fichiers les instructions du pré-processeur nécessaires sachant que l'on veut, pendant la phase de déboguage, afficher les résultats de tous les calculs intermédiaires.

Fichier racinesEqn.h

```
#include <stdio.h>

/* fonction de calcul du discriminant */
float dis (float a, float b, float c)
 {
   return (b * b - 4 * a * c);
 }

/* fonction calculant les racines reelles */
void rac2 (float r, float r1)
 {
   printf ("2 racines reelles : %f et %f\n", r + r1, r - r1);
 }

/* fonction calculant les racines complexes */
void  complex (float r, float r1)
 {
   printf ("2 racines complexes : %f + %f i et %f - %f i\n", r, r1, r, r1);
}

/* fonction calculant la racine double */
void  racd (float r)
 {
   printf (" racine double : %f\n", r);
 }
```

Fichier racinesEqn.c

```
#include <stdio.h>
#include <math.h>
#include "racinesEqn.h"
int  main () {
   float a,b,c,r,r1;
   double rdis;
   float res;
   printf("calcul des racines de ax2 + bx + c\n\n");
   /* saisie des coefficients */
   printf("saisissez les coefficients a b et c\n");
   scanf("%f %f %f",&a,&b,&c);
   if( a == 0 ){
     printf(" Equation du premier degre \n");
     printf(" La solution est x = %f \n", -c / b);
     exit(0);
   }
   r = -b/(2 * a);

switch ( res < 0 ? -1 : (res >0  ? 1 : 0 )) {
   case 1:
     rdis = sqrt(res);
     r1 = rdis / ( 2 * a);
     rac2(r,r1);
     break;
   case -1:
     rdis = sqrt(-res);
     r1 = rdis / ( 2 * a);
     complex(r,r1);
     break;
   case 0: racd(r);
     break;
   }
   return 0;
}
```