Langage C avancé
Utilisation du préprocesseur cpp

Samuel KOKH

samuel.kokh@cea.fr

MACS 1 – Institut Galilée

Le préprocesseur cpp

Qu'est-ce que le préprocesseur (preprocessor)

Le préprocesseur est un programme qui effectue des modifications syntaxiques dans les fichiers sources de manière automatique.

shell

```
$ which cpp
/usr/bin/cpp
```

Appel du préprocesseur

Appels encapsulés dans les appels à gcc (usage courant)

shell

```
$ gcc -Wall -c A.c
```

Lance le préprocesseur sur le fichier `A.c` et compile le résultat.

Appel « à la main » (éventuellement utile pour du débuggage)

Affichage du résulat sur la sortie standard

```
$ cpp A.c
```

Envoie du résultat dans un fichier via un pipe

```
$ cpp A.c > newA.c
```

Comment expliquer au préprocesseur ce qu'il doit faire ?

Utilisation de «directives» préprocesseurs. Les directives commencent toujours par le caractère #.

Remplacement de texte (macros)

Code source

```
1 #define LINESIZE 1024
2
3 char buffer[LINESIZE];
4 snprintf(buffer, LINESIZE, "hello \n");
```

Code source apres traitement par cpp

```
1
2
3 char buffer[1024];
4 snprintf(buffer, 1024, "hello \n");
```

Exemple d'utilisation des macros

```
#define TRUE   (1==1)
#define FALSE  (0==1)

if (test==TRUE){
   DoSomething
}
```

Code source apres traitement par cpp

```
/* extrait du fichier math.h */
# define M_PI          3.14159265358979323846   /* pi */

/* extrait du fichier stddef.h */
#define NULL ((void *)0)
```

Définition de macros avec paramètres

```
#define SQUARE(a)   ((a)*(a))
#define MULTIPLY(a,b)   ((a)*(b))

double d = SQUARE(3.24);
double dd = MULTIPLY(3.2,4.5);
```

Code source apres traitement par cpp

```
double d = ((3.24)*(3.24));
double dd = ((3.2)*(4.5));
```

Définition de macros avec paramètres : écueils

Les macros avec paramètres peuvent être la source de très gros problèmes. Il convient par exemple de faire très attention à l'usage des parenthèses !

```
1 #define SQUARE(a)   a*a
2 #define MULTIPLY(a,b)   a*b
3
4 double d = SQUARE(3 + 2);
5 double dd = MULTIPLY(3 + 1,2);
```

Code source apres traitement par cpp

```
1 double d = 3 + 2*3 + 2;
2 double dd = 3 + 1*2;
```

Définition de macros avec paramètres : écueils

```c
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
#define SQUARE  (a)*(a)

int i=3;
int j=0;
double x = 3;
double m = min(3.145,x);
double M = min(i,j++);
double d = 1./SQUARE(2);
```

Code source apres traitement par cpp

```c
double x = 3;
double m = ((3.145) < (x) ? (3.145) : (x));
double M = ((i) < (j++) ? (i) : (j++));
/* j est incremente deux fois ! */

double d = 1./(2)*(2);
/* d = 1 ! */
```

Inclusion de fichiers

La directive `#include` permet d'inclure dans le fichier à compiler le contenu d'un autre fichier (à la manière d'un copier-coller).

```
1 #include<stdio.h>
2 #include "myHeader.h"
```

- `#include<>` cherche les fichiers parmi une liste de directories « système » comme `/usr/local/include` ou `/usr/include`.
- `#include""` cherche les fichiers dans le directory local ou une liste de directories passés en argument à `gcc` via l'option de compilation `-I`, par ex. :

```
$ gcc -I dir1 -I dir2 -I ../dir3 -Wall -c foo.c
```

Compilation conditionnelle : #ifdef/#else/#endif

```
#ifdef XXX
..... // option 1
#else
..... // option 2
#endif
```

Si XXX est défini par une directive #define alors on tient compte des lignes option 1 sinon c'est l'option 2.

```
#ifdef ENGLISH
char msg[] = "hello";
#else
char msg[] = "bonjour";
#endif
```

Compilation conditionnelle : #ifndef/#else/#endif

```
#ifndef XXX
..... // option 1
#else
..... // option 2
#endif
```

Si XXX est n'est **pas** défini par une directive #define alors on tient compte des lignes option 1 sinon c'est l'option 2.

```
#ifndef _WIN32
// _WIN32 is defined by all Windows 32 compilers, but not by others.
#include <unistd.h>
#else
#include <windows.h>
#endif
```

Compilation conditionnelle : les « include guards »

Tous les fichiers `*.h` doivent impérativement être écrits comme suit.

fichier `myheader.h`

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

... contenu du fichier myheader.h

#endif
```

Ceci permet d'éviter les inclusions récursives de fichier `*.h`.

Compilation conditionnelle : #if//#elif#else/#endif

```
#if test1
..... // option 1
#elif test2
..... // option 2
#else
..... // option 3
#endif
```

test1 et test2 sont des expressions entières évaluées par le préproceseur. Les valeurs non-nulles sont équivalentes à « vrai » et déclenchent la compilation.

```
#if DEBUG_LEVEL >= 2
printf("a lot of debug info\n");
#elif DEBUG_LEVEL == 1
printf("a bit of debug info\n");
#endif
```

Compilation conditionnelle : commenter un bloc de code

fichier `myheader.h`

```
1 #if 0
2 printf("ceci est un bloc de lignes");
3 printf("de code ");
4 printf("que je souhaite commenter");
5 #endif
```

Macros qui génèrent des instructions

Il convient d'être prudent avec les macros qui génèrent des instructions.

```
#define ERROR(msg) fprintf(stderr,msg); exit(EXIT_FAILURE)


if(NULL == ptr)
    ERROR("problem with the pointer");
```

est transformé en

```
if(NULL == ptr)
    fprintf(stderr,msg);
exit(EXIT_FAILURE);
```

Macros qui génèrent des instructions

Cette version ne convient pas non plus.

```
#define ERROR(msg) {fprintf(stderr,msg); exit(EXIT_FAILURE)};

if(NULL == ptr)
    ERROR("problem with the pointer");
```

est transformé en

```
if(NULL == ptr){
    fprintf(stderr,msg);
    exit(EXIT_FAILURE);
};
```

et on ne peut pas ajouter de `else`

```
if(NULL == ptr)
    ERROR("problem with the pointer");
else {
    /* something else */
}
```

Macros qui génèrent des instructions

La solution consiste à utiliser un do {} while(0)

```
#define ERROR(msg) do{fprintf(stderr,msg);exit(EXIT_FAILURE)}while(0)

if(NULL == ptr)
    ERROR("problem with the pointer");
```

est transformé en

```
if(NULL == ptr)
    do {
        fprintf(stderr,msg);
        exit(EXIT_FAILURE);
    } while(0);
```

Scope de définition d'une macro

Le préprocesseur remplace une expression par la valeur de sa macro à partir de l'endroit où elle est définit dans le fichier.

La directive `#undef` permet d'annuler la définition d'une macro.

```
1 bob = X;
2 #define X 125
3 alice = X;
4 peter = X;
5 #undef X
6 mary = X;
```

est remplacé par

```
1 bob = X;
2 alice = 125;
3 peter = 125;
4 mary = X;
```

On peut écrire les macros sur plusieurs lignes grâce au backslash.

```
#define MSG "hello \
world\
"

#define ERROR(msg) do{\
fprintf(stderr,msg);\
exit(EXIT_FAILURE)}\
while(0)


printf(MSG);
if(NULL == ptr)
    ERROR("wrong pointer");
```

Quelques Macros prédéfinies

- `__DATE__` : est remplacé par la date de compilation
- `__FILE__` : est remplacé par le nom du fichier source en train d'être compilé
- `__LINE__` : est remplacé par le numéro de ligne dans le fichier source en train d'être compilé
- `__TIME__` : est remplacé par l'heure de compilation

Quelques Macros utiles pour le débuggage

```
#define DISPLAYLINE printf("DB %s : %d\n",__FILE__,__LINE__)

#define ShowDouble(var) printf("DB %s = %g\n",#var,var)
#define ShowInt(var) printf("DB %s = %d\n",#var,var)
#define ShowChar(var) printf("DB %s = %c\n",#var,var)
#define ShowString(var) printf("DB %s = %s\n",#var,var)
```

Définition de macro à l'appel de gcc

Il est possible de définir ou de spécifier la valeur d'une macro à la volée, à l'appel de gcc, sans modifier les fichiers sources.

Définition de la macro `myMacro`

```
$ gcc -DmyMacro -Wall -c foo.c
```

Définition de la valeur de la macro `myMacro`

```
$ gcc -DmyMacro=2 -Wall -c foo.c
```