Macros C & hacks

Lionel Lacassagne

Sorbonne University / LIP6 / ALSOC

https://www.lip6.fr/

lionel.lacassagne@lip6.fr

Pourquoi utiliser des macros C (de nos jours) ?

- Pour améliorer la sémantique du programme
  - `#define add3(a,b,c) a+b+c` ⇒ `s=add3(x0,x1,x2);`
  - `#define load1(X,i) X[i]` ⇒ `x0=load1(X,i);`
  - `#define store1(Y,i,y) Y[i]=s` ⇒ `store1(Y,i,s);`
  - pas de ; à la fin de la définition de la macro, mais à son appel
- Parce qu'un niveau d'indirection est suffisant pour faire de la magie
  - pour ajouter une fonctionnalité supplémentaire sans modifier le code:
  - pour du debug caché `#define load1(X,i) load1_check(X,i)`
  - pour faire des logs `#define load1(X,i) load1_histogram(X,i)`
  - pas aussi souple que du C++ (variables cachées dans des objets), mais plus simple
  - peut nécessiter des variables globales non passées à la macro ou à la fonction (`gi0`, `gi1`)

```
float load1_check(float* X, int i)} {
  if((i<gi0) || (i>gi1)) {
    printf("i = \%d not in [\%d, \%d]\n", i, gi0, gi1); exit(-1);
  }
  return X[i];  // comportement du load1


float load1_histogram(float* X, int i)} {
  H[i]++;
  return X[i];  // comportement du load1
```

Composition de Macros

à l'infini et au delà ...

- Une macro peut appeler :
  - un opérateur `#define load1(X,i) X[i]`
  - plusieurs opérateurs `#define add3(a,b,c) a+b+c`
  - plusieurs macros `#define add6(a,b,c,d,e,f) add2(add3(a,b,c),add3(d,e,f))`
- Faire attention à la priorité des opérateurs = mettre des parenthèses
  - si on a (sur deux lignes) `#define add(a,b) a+b` et `#define mul(a,b) a*b`
  - alors `mul(add(a,b),add(c,d))` donne `a+b*c+d`
  - mais si on `#define add(a,b) (a+b)` et `#define mul(a,b) (a*b)`
  - alors `mul(add(a,b),add(c,d))` donne `((a+b)*(c+d))`
- Macros récursives: pour limiter la taille des définitions
  - les parenthèses peuvent créer des arbres d'évaluation déséquilibrés...

```
#define add2(x0,x1) (x0+x1)
#define add3(x0,x1,x2) (x0+x1+x2)
#define add4(x0,x1,x2,x3) (x0+x1+x2+x3)
#define add5(x0,x1,x2,x3,x4) (x0+x1+x2+x3+x4)

#define add2(x0,x1) (x0+x1)
#define add3(x0,x1,x2) (add2(x0,x1)+x2)
#define add4(x0,x1,x2,x3) (add3(x0,x1,x2)+x3)
#define add5(x0,x1,x2,x3,x4) (add4(x0,x1,x2,x3)+x4)
```

Fonctions d'affichage #1

- Pour simplifier et activer/désactiver automatiquement du code de debug
  - sans modifier le code source car modifier = revalider le code via tests unitaires
  - `#x` dans une macro s'appelle une *stringification*: conversion d'une variable en chaine de caractères

```
#define ENABLE
#ifdef ENABLE
  #define VERBOSE(X) X
  #define PUTS(str) puts(str)
  #define idisp(x) printf("%s = %d", #x, x)
  #define fdisp(x) printf("%s = %.0f", #x, x)
  #define CR putchar('\n')
#else
  #define VERBOSE(X)
  #define PUTS(str)
  #define idisp(x)
  #define CR
#endif
```

- Utilisation (en commentant `#define ENABLE` on aura juste `x=X[i];`)

```
PUTS("chargement d'un point");
idisp(i);CR;
x = load1(X,i);
fdisp(x);CR;
VERBOSE(if(i==0) printf("premier load de la boucle"););
```

- Remarque
  - les macros d'accès mémoire et de calcul peuvent en plus faire des affichages
  - pas de limite ...

Macros et variables locales

Notion de scope

- Un scope est défini par une paire d'accolades `{ /* scope */ }`
  - il y en a pour chaque fonction et pour chaque boucle (de plus d'une instruction)
  - cela permet de définir des variables locales au *scope* (de la fonction, de la boucle)
- Il est possible de faire pareil pour une macro via une boucle `do{ ...}while(0)`
  - la macro possède alors ses propres variables locales qui n'auront pas d'effet de bord dans la fonction dans laquelle la macro sera appellée
  - ce qui permet d'éviter de déclarer les variables de la macro dans la fonction appellante
- Il est aussi possible de définir une macro sur plusieurs lignes avec un \
  - même si les *one-liners* sont des hackers cools
  - dans ce cas, on prendra soin de les aligner (1 par ligne sauf pour la dernière)
  - on prendra aussi soin d'indenter la macro, pour avoir le *look n' feel* d'une fonction

```
#define FIND_MAX(X,N, MAX) do{ \
    float m = X[0];             \
    for(int i=1; i<N; i++) {    \
        if(X[i]>m) m = X[i];    \
    }                           \
    MAX = m;                    \
}while(0)
```

Conclusion

- Un code unique ...
  - pour la mise au point d'un code (tests d'accès mémoire cachés dans `load` et `store`)
  - pour sa production (code rapide sans `if-then-else` pour faire des tests)
  - sans modification, juste quelques macros paramétrables par d'autres macros ...
- ... permettant
  - de trouver facilement des bugs,
  - de diminuer le temps de debug (80% du temps de dev total)
  - pour accélérer sa vitesse de dev et donc sa productivité...
- Plus d'informations et d'exemples (liens clickables)
  - *C PreProcessor* (`#ifdef #else #endif`) `https://gcc.gnu.org/onlinedocs/cpp/`
  - macros `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`
  - stringification = `#`: `https://gcc.gnu.org/onlinedocs/cpp/Stringizing.html`
  - concaténation = `##`: `https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html`