Programmation avancée en C :

Préprocesseur, modificateurs et pointeurs de fonction

Licence informatique 3e année

Université Gustave Eiffel

Le préprocesseur

- Processus avant la compilation
- Opérations brutes sur les fichiers
- On peut voir ce que fait le préprocesseur avec : `gcc -E -P fichier.c`

```
1 #define OK (0)
2 #define MSG "Hello world!"
3
4 int main(int argc, char* argv[]){
5   printf("%s\n", MSG);
6   return OK;
7 }
```

```
nborie@perceval:~> gcc -E -P test.c
int main(int argc, char* argv[]){
  printf("%s\n", "Hello world!");
  return (0);
}
```


Inclusion de fichiers

- `#include foo` : inclusion du fichier désigné par `foo` dans le fichier courant
- `foo` peut être de la forme :
  - `"..."` : recherche dans le répertoire courant
  - `<...>` : recherche dans les répertoires d'inclusion (ceux par défaut et ceux définis par l'utilisateur)
  - possibilité de sous-répertoires : `<sys/types.h>`

Inclusion de fichiers

- On peut définir ses propres répertoires d'inclusion avec `gcc -Idir`

```
1 #include <my_stdio.h>
2
3 int main(int argc, char* argv[]){
4   printf("This is my printf\n");
5   return 0;
6 }
```

```
nborie@perceval:~> gcc test.c
test.c: erreur fatale: my_stdio.h Aucun fichier ou dossier de ce type
compilation terminée.
nborie@perceval:~> gcc test.c -I/home/nborie/perso_lib/include/
```

Inclusion de fichiers

- Chaque fichier inclus est d'abord traité par le préprocesseur.
- Il est ensuite inséré là où il a été inclus.

fichier **const.h** :

```
1 #define A 0
2 #define B 1
3
4 A,B,B,A,B
```

fichier **test.c** :

```
1 int t[] = {
2 #include "const.h"
3 };
4
5 int main(int arc, char* argv[]){
6   /* ... */
7 }
```

```
nborie@perceval:~> gcc -E -P test.c
int t[] = {
0,1,1,0,1
};
int main(int arc, char* argv[]){
  /* ... */
}
```


Définition de macros

- `#define NOM texte` (préférer les noms en majuscules)
- Remplace `NOM` brutalement par `texte`, sauf dans les chaînes et les identificateurs qui contiennent `NOM`, et seulement après le `#define`
- Utiliser des `\` si `texte` tient sur plusieurs lignes :

```
1 #define NOM debut \
2 fin
```

- Utile pour définir des constantes

Définition de macros

- On peut définir autres choses que des constantes :

```
#define SI if
#define SINON else
#define FOREVER for(;;)
```

- On peut même de rien mettre :

```
#define const_H
#define USE_REGEXP
```


Définition de macros

- On peut surcharger des choses existantes.

```
1  #include <stdio.h>
2  #define return printf("%s fini\n", __FUNCTION__); return
3
4  int fact(int n){
5    if (n <= 1) { return 1; }
6    return n * fact(n-1);
7  }
8
9  int main(int argc, char* argv[]){
10   printf("%d\n", fact(5));
11   return 0;
12 }
```

```
nborie@perceval:~> ./test
fact fini
fact fini
fact fini
fact fini
fact fini
120
main fini
```

Définition de macros

- On peut interdire certaines fonctions.

```c
#define fseek DONT_USE_FSEEK_BUT_FSETPOS
/* ... */
int main(int argc, char* argv[]){
  FILE* f=fopen("foo", "r");
  if (f == NULL) exit(1);
  fseek(f, SEEK_END, 0);
  /* ... */
  return 0;
}
```

```
fwjmath@fwjmath-hover:~/test/c$ gcc -ansi -Wall -o test test.c
test.c: In function ‘main’:
test.c:4:15: warning: implicit declaration of function
‘DONT_USE_FSEEK_BUT_FSETPOS’ [-Wimplicit-function-declaration]
collect2: error: ld returned 1 exit status
```


Définition de macros

- `#define NOM(a,b) texte`
- Définition d'une macro `NOM` prenant deux paramètres `a` et `b`
- Pas d'espace entre `NOM` et `(`
- Toujours parenthéser
- À utiliser avec soin !

Définition de macros

- Problème de paranthésage :

```c
#define SQUARE(a) a*a

int main(int argc, char* argv[]){
  printf("%d\n", SQUARE(1+2));
  /* 1+2*1+2 = 1+2+2 = 5 */
  return 0;
}
```

- Attention aux effets de bord :

```c
#define SQUARE(a) a*a

int main(int argc, char* argv[]){
  int i=2;
  printf("%d\n", SQUARE(i++));
  /* i++ * i++ : i incremente deux fois ? */
  return 0;
}
```


Undef

- `#undef nom`
- Permet d'annuler la définition de `nom`, si elle existe
- Pratique pour être sur qu'on appelle une fonction et pas une macro

```c
#undef MAX

int MAX(int a, int b){
  return (a > b) ? a : b;
}
```

#expr

- Avec un `#` devant un paramètre du macro, on peut en obtenir une chaîne.

```
1 #define EVAL(e) printf("%s=%d\n", #e, e)
2
3 int main(int argc, char* argv[]){
4   int i=4;
5   EVAL(2*i+7);
6   return 0;
7 }
```

```
nborie@perceval:~> ./test
2*i+7=15
```


Inclusion conditionnelle

- `#ifdef nom` : inclut tout jusqu'au `#endif`, `#else` ou `#elif` correspondant, si et seulement si `nom` a été défini
- `#ifndef nom` : si et seulement si `nom` n'est pas défini
- mécanisme pour éviter les inclusions multiples

```
1 #ifndef __FOO_MODULE__
2 #define __FOO_MODULE__
3
4 /* ... */
5
6 #endif
```


Inclusion conditionnelle

- `#if expr` : inclut tout jusqu'au `#endif`, `#else` ou `#elif` correspondant, si et seulement si `expr` est non nulle
- `expr` ne peut contenir que des opérations sur des constantes entières

```
1 #if NPLAYER==0
2 #define MODE DEMO
3 #elif NPLAYER==1
4 #define MODE SINGLE
5 #elif NPLAYER==2
6 #define MODE DUEL
7 #else
8 #define MODE MULTI
9 #endif
```


Inclusion conditionnelle

- `#if 0` : pratique pour ignorer un bout de code
- Pas de problème d'imbrication comme avec `/*` et `*/`

```
1 #if 0
2 /* I'm scared of this guru verion! */
3 void cpy(char* dest, char* src){
4   while(*dest++ = *src++) {}
5 }
6 #endif
7
8 void cpy(char* dest, char* src){
9   int i = -1;
10  do{
11    i++;
12    dest[i] = src[i];
13  }while(dest[i] != '\0');
14 }
```

Variables du préprocesseur

- `__LINE__` : numéro de la ligne courante
- `__FILE__` : nom du fichier courant
- `__DATE__` : mois, jour, année
- `__TIME__` : heures, minutes, secondes
- Pratique pour le débougage
- La date et l'heure sont celle de la compilation du fichier, plus précisément le passage par préprocesseur.

```
1 void info(void){
2   printf("Program compiled on %s at %s\n",
3          __DATE__, __TIME__);
4 }
```


Erreurs et avertissements

- `#warning texte` et `#error texte`

```
1  #ifndef NPLAYERS
2  #warning DEFAULT=1 PLAYER
3  #elif NPLAYERS<=0 || NPLAYERS>2
4  #error INVALID NUMBER OF PLAYERS!
5  #endif
6
7  int main(int argc, char* argv[]){
8    /* ... */
9  return 0;
10 }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
test.c:2:2: attention : #warning DEFAULT=1 PLAYER [-Wcpp]
nborie@perceval:~> gcc -o test test.c -Wall -ansi -DNPLAYERS=1
nborie@perceval:~> gcc -o test test.c -Wall -ansi -DNPLAYERS=4
test.c:4:2: erreur: #error INVALID NUMBER OF PLAYERS!
```


Changement du fonctionnement à la compilation

- Suivant le fonctionnement voulu du code, on peut faire passer un message à la compilation.

```
1  #ifndef PROFILE_MODE
2  #define PROFILE /* Empty macro!!! */
3  #else
4  #define PROFILE printf("foo foo foo\n");
5  #define return printf("bar bar bar\n"); return
6  #endif
7
8  int main(int argc, char* argv[]){
9    PROFILE
10     /* ... */
11   return 0;
12 }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
nborie@perceval:~> gcc -o test test.c -Wall -ansi -DPROFILE_MODE
nborie@perceval:~> ./test
foo foo foo
bar bar bar
```


Assert

- `assert(expr);` (`<assert.h>`)
- Macro qui test si `expr` est `0` (condition fausse).
- Si oui, affiche un message sur `stderr` et quitte le programme très brutalement avec `abort`
- Désactivé si `NDEBUG` est définie
- Peut être utilisée pour du débogage
- À désactiver à la production pour éviter les mauvais surpris !

Assert

```c
#include <stdio.h>
#include <assert.h>

int div(int a, int b){
  assert(b!=0);
  return a/b;
}

int main(int argc, char* argv[]){
  printf("%d\n", div(4,0));
  return 0;
}
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
test: test.c:5: div: Assertion 'b!=0' failed.
zsh: abort (core dumped)  ./test
```


Const

- `const type nom = valeur;`
- Impossible de modifier la variable `nom`
- Appliqué à un tableau en paramètre, empêche de modifier ses éléments
- Représailles dépendant du compilateur, comportement indéfini lors d'utilisation d'attaque par cast.
- Plusieurs niveaux pour figer les variables :
  `const char * *const argv`
  tableau de chaînes dont l'adresse `argv` est constante, les lettres `argv[i][j]` sont constantes mais on peut échanger deux chaînes `argv[1]` et `argv[2]` par exemple.

Const

```c
void swap(const char** const argv){
  const char* tmp;
  tmp = argv[1];
  argv[1] = argv[2];
  argv[2] = tmp;
}

int main(int argc, char* argv[]){
  int i;
  swap((const char** const)argv);
  for(i = 0; i < argc; i++) printf("%s\n", argv[i]);
  return 0;
}
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi -pedantic
nborie@perceval:~> ./test arg1 arg2
./test
arg2
arg1
```


Const

- Sur un tableau :
  - On ne peut pas modifier le contenu du tableau.
  - Mais on peut modifier son adresse.

```c
void cpy(char* dest, const char* src){
  while(*dest++ = *src++){}
}
```

- `const` permet d'éviter des bogues
- À utiliser autant que possible
- Quand en doute, utiliser `cdecl` !

```
fwjmath@fwjmath-hover:~$ cdecl explain "const char** const argv"
declare argv as const pointer to pointer to const char
```

Classes de stockage

- `extern`
- Déclare quelque chose sans le définir, ni réserver d'espace.
- Doit être fourni par l'extérieur lors de l'édition de liens.
- Utile pour partager une variable entre plusieurs `.c`

```c
/* My religion forbids me to include
   stdxxx headers. I must rewrite them... */
extern int printf(const char *, ...);

int main(int argc, char* argv[]){
  printf("Hello\n");
  return 0;
}
```

Pour une fonction, pas de différence... Par contre, c'est une indication pour le programmeur !

Classes de stockage

- Même avec la protection par macros, si on met une variable dans un `.h`, on a des problèmes

const.h :
```c
#ifndef __CONST_H__
#define __CONST_H__

int MODE=12;

#endif
```

main.c :
```c
#include <stdio.h>
#include "const.h"

extern void foo(void);

int main(int argc, char* argv[]){
  foo();
  return 0;
}
```

foo.c :
```c
#include "const.h"

void foo(void){
  MODE=MODE+5;
}
```

```
~> gcc -o test main.c foo.c -Wall -ansi
...définitions multiples de << MODE >>
...défini pour la première fois ici
erreur: ld -> 1 code d'état d'exécution
```


Classes de stockage

- Solution : mettre la variable dans un `.c` et sa déclaration en extern dans le `.h`

const.h :
```c
#ifndef __CONST_H__
#define __CONST_H__

extern int MODE;

#endif
```

main.c :
```c
#include <stdio.h>
#include "const.h"

extern void foo(void);

int main(int argc, char* argv[]){
  foo();
  return 0;
}
```

foo.c :
```c
#include "const.h"

void foo(void){
  MODE=MODE+5;
}
```

const.c :
```c
int MODE=12;
```

```
~> gcc -o test main.c foo.c const.c -Wall -ansi
```


Classes de stockage

- `static`
- Pour une variable globale ou une fonction, indique qu'elle ne peut pas être visible de l'extérieur

```c
static int var1; /* =0 */
int var2;

static void foo1(void) {}
void foo2(void) {}
```

```c
extern int var1, var2;
extern void foo1(void);
extern void foo2(void);

int main(int argc, char* argv[]){
  foo1();
  foo2();
  return 0;
}
```

```
nborieperceval:~> gcc -o test test.c foo.c -Wall -ansi
foo.c:1:12: attention : 'var1' defined but not used [-Wunused-variable]
foo.c:4:13: attention : 'foo1' defined but not used [-Wunused-function]
/tmp/ccdSWaGJ.o: dans la fonction << main >>:
test.c:(.text+0x10): référence indéfinie vers << foo1 >>
collect2: erreur: ld -> 1 code d'état d'exécution
```

Classes de stockage

- Avec `static`, initialisation par défaut à zéro
- Variable locale `static` = globale non visible à l'extérieur de la fonction

```
1  /* This function do an hard job but no more
2     than once per second if called too often... */
3  void temporize(){
4    static time_t previous;
5    time_t current = time(NULL);
6    if (current == previous) return;
7    previous = current;
8    /* Do the hard job now!!! */
9  }
```


Classes de stockage

- `register type nom;`
- Demande à utiliser un registre pour la variable `nom`, si possible

```
1  #define LIM 2000
2
3  int main(int argc, char* argv[]){
4    OPT unsigned long int i,j,k,res;
5    for (i=0 ; i<LIM ; i++)
6      for (j=0 ; j<LIM ; j++)
7        for (k=0 ; k<LIM ; k++)
8          res=res+i+j+k;
9    return 0;
10 }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi -DOPT=
nborie@perceval:~> time ./test
./test  19,71s user 0,00s system 99% cpu 19,722 total
nborie@perceval:~> gcc -o test test.c -Wall -ansi -DOPT=register
nborie@perceval:~> time ./test
./test  6,56s user 0,00s system 99% cpu 6,565 total
```

Quel est la taille de mes registres ?

Pointeurs de fonctions

- Nom d'une fonction = adresse de cette fonction
- Déclarer un pointeur de fonction : `type_retour (*nom)(paramètres);`

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]){
4    int (*f)(const char*, ...) = printf;
5    f("%p\n", f);
6    return 0;
7  }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
0x400420
```


Pointeurs de fonctions

- On peut utiliser les pointeurs de fonctions comme n'importe quelles variables.
- Exemple : `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`

```
1  int cmp(const void* a, const void* b){
2    int* x=(int*)a;
3    int* y=(int*)b;
4    return *y-*x;
5  }
6
7  int main(int argc, char* argv[]){
8    int i, t[]={4,51,57,42,2,18};
9    qsort(t, 6, sizeof(int), cmp);
10   for (i=0 ; i<6 ; i++) printf("%d ", t[i]);
11   printf("\n");
12   return 0;
13 }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
57 51 42 18 4 2
```

Pointeurs de fonctions

- On peut définir des types pointeurs de fonctions.
- Exemple 1 : type de `printf`

```
1  typedef int(*PRINT)(const char*, ...);
2
3  int main(int agrc, char* argv[]){
4    PRINT p = printf;
5    int i;
6    for(i = 0; i < 5; i++){
7      p("%d ", i);
8    }
9    return 0;
10 }
```


Pointeurs de fonctions

- Exemple 2 : encodage de caractères

```
1  typedef int(*Encoding)(int, FILE*);
2
3  int utf8(int, FILE*);
4
5  Encoding ASCII = fputc;
6  Encoding UTF8 = utf8;
7
8  int utf8(int c, FILE* f){
9    /* ... */
10   return c;
11 }
12
13 void print_string(Encoding e, char* s){
14   while(*s!='\0' && EOF != e(*s, stdout)){
15     s++;
16   }
17 }
```


Pointeurs de fonctions

- Attention aux parenthèses !!!!!!!!
- - `int *f(char)` : fonction prenant un `char` et retournant un pointeur sur un `int`
- - `int (*f)(char)` : pointeur sur une fonction prenant un `char` et retournant un `int`
- Moralité : toujours ajouter un couple de parenthèses pour l'étoile et le nom de la fonction (ou du nouveau type défini lorsque l'on définit un type de pointeur de fonctions)
- En cas de doute, utiliser `cdecl` !

Pointeurs de fonctions

- Exemple 3 : tableau de pointeurs de fonctions

```
1  typedef double(*Trigo)(double);
2
3  Trigo f[] = {cos, sin, tan, acos, asin, atan};
4
5  int main(int argc, char* argv[]){
6    double PI=3.14159;
7    printf("sin(%f)=%f", PI, f[1](PI));
8    return 0;
9  }
```