Cours de C - IR1
2007-2008

Préprocesseur, modifieurs & fonctions avancées

Sébastien Paumier

MCours.com

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`**

```
#define OK (0)
#define MSG "Hello world!"

int main(int argc,char* argv[]) {
   printf("%s\n",MSG);
   return OK;
}
```

```
$>gcc -E -P preproc.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`**

```c
#include <my_stdio.h>

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

```
$>gcc preproc.c
preproc.c:1:22: my_stdio.h: No such file or directory
$>gcc preproc.c -I../foo-0.1.4/include
```

Inclusion de fichiers

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

const.h

```
#define A 0
#define B 1

A,B,B,A,B
```

pas forcément du code autonome

define.c

```
int t[]={
#include "const.h"
};

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

```
$>gcc -E -P define.c
int t[]={
0,1,1,0,1
};

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

Définition de macros

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

```
#define NOM debut\
fin
```

- utile pour définir des constantes

Définition de macros

- on peut définir autre chose que des constantes:

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

- on peut même ne rien mettre:

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

Définition de macros

- on peut surcharger des choses existantes

```c
#include <stdlib.h>

void* (*old_malloc)(size_t)=malloc;
#define malloc my_malloc

unsigned int n_malloc=0;

void* my_malloc(size_t n) {
   n_malloc++;
   return old_malloc(n);
}

int main(int argc,char* argv[]) {
   int* t=malloc(4*sizeof(int));
   /* ... */
   return 0;
}
```

```
$>gcc -E -P my_malloc.c

contenu de stdlib.h

void* (*old_malloc)(size_t)=malloc;

unsigned int n_malloc=0;

void* my_malloc(size_t n) {
   n_malloc++;
   return old_malloc(n);
}

int main(int argc,char* argv[]) {
   int* t=my_malloc(4*sizeof(int));

   return 0;
}
```

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;
}
```

```
$>gcc fseek.c -Wall -ansi
fseek.c:8: warning: implicit declaration of
function `DONT_USE_FSEEK_BUT_FSETPOS'
cci6baaa.o(.text+0x57):fseek.c: undefined
reference to `DONT_USE_FSEEK_BUT_FSETPOS'
```

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 parenthésage:

```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 incrémenté 2 fois */
   return 0;
}
```

undef

- **#undef nom**
- permet d'annuler la définition de **nom**, si elle existe
- pratique pour être sûr qu'on appelle une fonction et pas une macro

```
#undef MAX

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

#expr

- avec un # devant un paramètre, on peut obtenir une chaîne

```
#define EVAL(e) printf("%s=%d\n",#e,e)

int main(int argc,char* argv[]) {
   int i=4;
   EVAL(2*i+7);
   return 0;
}
```

```
$>gcc eval.c -P -E
int main(int argc,char* argv[]) {
   int i=4;
   printf("%s=%d\n","2*i+7",2*i+7);
   return 0;
}
```

Inclusion conditionnelle

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

```
#ifndef CONST_H_
#define CONST_H_

/* ... */

#endif
```

Inclusion conditionnelle

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

```
#if NPLAYER==0
#define MODE DEMO
#elif NPLAYER==1
#define MODE SINGLE
#elif NPLAYER==2
#define MODE DUEL
#else
#define MODE MULTI
#endif
```

Inclusion conditionnelle

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

```
#if 0
/* I'm scared of this guru version! */
void cpy(char* dest,char* src) {
   while (*dest++=*src++);
}
#endif

void cpy(char* dest,char* src) {
   int i=-1;
   do {
      i++;
      dest[i]=src[i];
   } while (dest[i]!='\0');
}
```

Variables du préprocesseur

- `__LINE__` : numéro de la ligne courante
- `__FILE__` : nom du fichier courant
- `__DATE__` : mois/jour/année
- `__TIME__` : heures/minutes/secondes
- la date et l'heure sont celles de la compilation du fichier

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

Erreurs et avertissements

- **#warning texte** et **#error texte**

```
#ifndef NPLAYERS
#warning DEFAULT=1 PLAYER
#elif NPLAYERS<=0 || NPLAYERS>2
#error INVALID NUMBER OF PLAYERS!
#endif

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

```
$>gcc macros.c -Wall -ansi
macros.c:4:2: warning: #warning DEFAULT=1 PLAYER
$>gcc macros.c -Wall -ansi -DNPLAYERS=1
$>gcc macros.c -Wall -ansi -DNPLAYERS=4
macros.c:6:2: #error INVALID NUMBER OF PLAYERS!
```

assert

- `assert(expr);` (`assert.h`)
- macro qui teste si `expr` est nulle
- si oui, affiche un message sur `stderr` et quitte le programme très brutalement avec `abort`
- désactivée si `NDEBUG` est définie
- peut être utilisée pour du débogage

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;
}
```

```
$>gcc macros.c -Wall -ansi
$>./a.out
Assertion failed: b!=0, file macros.c, line 5
$>gcc macros.c -Wall -ansi -DNDEBUG
$>./a.out
Floating point exception
```

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:

```
void foo(const int t[]) {
   t[0]=0;
}
```

```
$>gcc const.c -Wall -ansi
const.c: In function `foo':
const.c:17: warning: assignment of read-only location
```

const

- sur un tableau:
  - on ne peut pas modifier le contenu du tableau
  - mais on peut modifier son adresse

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

- **const** permet d'éviter des bugs
- à utiliser autant que possible!

Classes de stockage

- **extern**
- déclare quelque chose sans le définir, ni réserver d'espace

pour une fonction, ça ne fait pas de différence, mais c'est une façon de montrer qu'elle n'est pas définie dans le même fichier

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

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

- utile pour partager une variable entre plusieurs **.c**

Classes de stockage

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

const.h

```
#ifndef CONST_H_
#define CONST_H_

int MODE=12;

#endif
```

foo.c

```
#include "const.h"

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

main.c

```
#include <stdio.h>
#include "const.h"

extern void foo(void);

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

```
$>gcc main.c foo.c -Wall -ansi
ccs3baaa.o(.data+0x0):foo.c: multiple definition of `MODE'
ccmqbaaa.o(.data+0x0):main.c: first defined here
```

Classes de stockage

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

const.c

```
int MODE=12;
```

const.h

```
#ifndef CONST_H_
#define CONST_H_

extern int MODE;

#endif
```

foo.c

```
#include "const.h"

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

main.c

```
#include <stdio.h>
#include "const.h"

extern void foo(void);

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

```
$>gcc 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

foo.c

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

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

test.c

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

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

```
$>gcc test.c foo.c -Wall -ansi
foo.c:1: warning: `var1' defined but not used
foo.c:4: warning: `foo1' defined but not used
cc8qbaaa.o(.text+0x1f):test.c: undefined reference to `foo1'
```

Classes de stockage

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

```
/**
 * This function performs a given task, but no more
 * than once per second if called too often.
 */
void temporize() {
   static time_t previous;
   time_t current;
   if ((current=time(NULL))==previous) return;
   previous=current;
   /* do the job here... */
}
```

Classes de stockage

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

```
#define LIM 2000

int main(int argc,char* argv[]) {
  OPT unsigned int i,j,k,res;
  for (i=0;i<LIM;i++)
    for (j=0;j<LIM;j++)
      for (k=0;k<LIM;k++)
         res=res+i+j+k;
  return 0;
}
```

```
$>gcc z.c -Wall -ansi -DOPT=
$>time -p a.out
real 23.81
user 23.66
sys 0.02
$>gcc z.c -Wall -ansi -DOPT=register
$>time -p a.out
real 12.95
user 12.88
sys 0.01
```

Pointeurs de fonctions

- nom d'une fonction=adresse de cette fonction
- déclarer un pointeur de fonction:

```
type_retour (*nom)(paramètres);
```

```c
#include <stdio.h>

int main(int argc,char* argv[]) {
    int (*f)(const char*,...)=printf;
    f("%p\n",f);
    return 0;
}
```

Pointeurs de fonctions

- on peut utiliser les pointeurs de fonctions comme n'importe quelles variables
- exemple: `void qsort(void* base,size_t n,size_t size, int (*compare)(const void*,const void*));`

```c
int cmp(const void* a,const void* b) {
   int* x=(int*)a;
   int* y=(int*)b;
   return *y-*x;
}

int main(int argc,char* argv[]) {
   int i,t[]={4,51,57,82,2};
   qsort(t,5,sizeof(int),cmp);
   for (i=0;i<5;i++) printf("%d\n",t[i]);
   return 0;
}
```



```
$>./a.out
82
57
51
4
2
```

Pointeurs de fonctions

- on peut définir des types pointeurs de fonctions
- exemple 1: type de **printf**

```
typedef int(*PRINT)(const char*,...);

int main(int argc,char* argv[]) {
    PRINT p=printf;
    int i;
    for (i=0;i<5;i++) {
        p("%d ",i);
    }
    return 0;
}
```

Pointeurs de fonctions

- exemple 2: encodage de caractères

```
typedef int(*Encoding)(int,FILE*);

int utf8(int,FILE*);

Encoding ASCII=fputc;
Encoding UTF8=utf8;

int utf8(int c,FILE* f) {
   /* ... */
   return c;
}

void print_string(Encoding e,char* s) {
   while (*s!='\0' && EOF!=e(*s,stdout)) {
      s++;
   }
}
```

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`

Pointeurs de fonctions

- exemple: tableau de pointeurs de fonctions

```
typedef double(*Trigo)(double);

Trigo f[]={cos,sin,tan,acos,asin,atan};

int main(int argc,char* argv[]) {
   double PI=3.14159;
   printf("sin(%f)=%f\n",PI,f[1](PI));
   return 0;
}
```

Nombre d'arguments variable

- comme `int printf(const char*,...);`
- type et macros dans `stdarg.h`
- au moins un paramètre
- on crée une variable de type `va_list` qui va parcourir les paramètres
- on l'initialise à partir du dernier paramètre connu avec `va_start`

Nombre d'arguments variable

- on accède au prochain paramètre avec **`va_arg(va_list,type)`**
- au programmeur de savoir quel est le type du prochain paramètre
  - ex: chaîne de formatage pour **`printf`**
- quand on a fini, on doit appeler une fois (et une seule) **`va_end`**

Formatage de date

```
void print_date(const char* fmt,...) {
   if (fmt==NULL) return;
   va_list args;
   va_start(args,fmt);
   int i=0,d;
   while (fmt[i]!='\0') {
      switch (fmt[i]) {
         case 'D': case 'M': case 'Y':
            d=va_arg(args,int);
            if (fmt[i]=='Y' && fmt[i+1]=='Y') {
               printf("%04d",d);
               i++;
            } else printf("%02d",d%100);
            break;
         default: printf("%c",fmt[i]);
      }
      i++;
   }
   va_end(args);
}

int main(int argc,char* argv[]) {
   print_date("D/M/Y\n",4,11,2006);
   print_date("date: D-M-YY\n",4,11,2006);
   return 0;
}
```

on ne peut pas afficher **D**, **M** ou **Y**!

```
$>./a.out
04/11/06
date: 04-11-2006
```

Nombre d'arguments variable

- on ne peut pas passer les `...` à une autre fonction
- mais on peut passer un `va_list`
- fonctions acceptant des `va_list`:
  - `vprintf,vfprintf,vsprintf`
  - `vscanf,vfscanf,vsscanf`

Personnaliser printf

- on voudrait afficher en binaire avec des belles options comme `"%02b"`
- première étape: afficher un nombre en binaire
  - parser le format
  - afficher, en respectant les règles de `printf`:
    - retourner le nombre de caractères affichés
    - une valeur négative en cas d'erreur

Parser un format

```
/**
 * Parses the given format string that is supposed to be of the
 * form %[ 0][n]b, where n is an integer representing the minimum number
 * of digits to print. Returns 0 if 'fmt' is not correct; 1 otherwise.
 */
int parse_format(const char* fmt,char *fill,int *min) {
   if (fmt==NULL || fmt[0]!='%') {
      return 0;
   }
   int i=1;
   *fill='\0';
   if (fmt[i]==' ' || fmt[i]=='0') {
      *fill=fmt[i++];
   }
   *min=0;
   if (fmt[i]>='1' && fmt[i]<='9') {
      do {
         *min=(*min)*10+fmt[i++]-'0';
      } while (fmt[i]>='0' && fmt[i]<='9');
   }
   if (fmt[i]!='b' || fmt[i+1]!='\0') {
      return 0;
   }
   return 1;
}
```

Afficher en binaire

```
int print_bin(const char* fmt,unsigned int d) {
   char fill;
   int n;
   if (!parse_format(fmt,&fill,&n)) {
      return -1;
   }
   int n_digits=get_n_digits(d);
   int i,ret=n_digits;
   if (fill!='\0' && n_digits<n) {
      ret=n;
      for (i=0;i<n-n_digits;i++) {
         printf("%c",fill);
      }
   }
   for (i=n_digits-1;i>=0;i--) {
      printf("%c",(d&(1<<i))?'1':'0');
   }
   return ret;
}
```

Personnaliser printf

- deuxième étape:
  - on voudrait que:

```
printf("%d en decimal = ",5);
print_bin("%b",5);
printf(" en binaire\n");
```

  - puisse s'écrire de façon plus pratique:

```
my_printf("%d en decimal = %b en binaire\n",5,5);
```

Personnaliser printf

- solution:
  - parser soi-même le format
  - utiliser `print_bin` pour les formats `%...b`
  - laisser `printf` s'occuper des autres formats
  - toujours respecter les règles de retour de `printf`

my_printf, 1/2

```
int my_printf(const char* fmt,...) {
   if (fmt==NULL) return -1;
   va_list args; va_start(args,fmt);
   int i=0,n=0; /* n=number of chars printed */
   while (fmt[i]!='\0') {
      if (fmt[i]!='%') { /* case of a normal char */
         if (EOF==fputc(fmt[i],stdout)) return -1;
         n++;
      } else { /* case of a format indication %... */
         switch (fmt[++i]) {
            case '\0': return -1;
            case '%': if (EOF==fputc(fmt[i],stdout)) return -1;
                      n++; break;
            default: {/* If we have '%???', we let printf do the job */
               i--; /* We get back on the '%' */
               int z=0;
               char fmt2[64];
               do {
                  fmt2[z++]=fmt[i++];
               } while (z<64 && fmt2[z-1]!='\0'
                        && !strchr("diouxXeEfgcspb",fmt2[z-1]));
               if (z==64) return -1;
               i--; /* We get back one character */
               if (fmt2[z-1]=='\0') return -1;
               fmt2[z]='\0';
```

my_printf, 2/2

```
                int ret;
                switch (fmt2[z-1]) {
                   case 'd': case 'i': case 'o': case 'u': case 'x': case 'X':
                      {int k=va_arg(args,int); ret=printf(fmt2,k);
                      break;}
                   case 'e': case 'E': case 'f':  case 'g':
                      {double d=va_arg(args,double); ret=printf(fmt2,d);
                      break;}
                   case 'c':
                      {char c=va_arg(args,int); ret=printf(fmt2,c);
                      break;}
                   case 's':
                      {char* s=va_arg(args,char*); ret=printf(fmt2,s);
                      break;}
                   case 'p':
                      {void* p=va_arg(args,void*); ret=printf(fmt2,p);
                      break;}
                   case 'b':
                      {unsigned int value=va_arg(args,unsigned int);
                      ret=print_bin(fmt2,value); break;}
                }
                if (ret<0) return -1;
                n=n+ret;
 }}}i++;} va_end(args); /* A éviter, sauf quand ça ne tient pas dans la page :) */
   return n;
}
```

Sans chaîne de formatage

- si on suppose le type connu, on peut utiliser une valeur spéciale pour tester la fin des arguments

```
/**
 * Like strcat, but optimized for multiple strings. The last
 * parameter must be NULL. 'dest' is supposed to be large enough.
 */
void my_strcat(char* dest,...) {
   va_list args;
   va_start(args,dest);
   dest=dest+strlen(dest);
   char* s;
   while ((s=va_arg(args,char*))!=NULL) {
      while ((*dest++=*s++));
      dest--;
   }
   va_end(args);
}
```

- connaître les règles des échecs ne suffit pas pour savoir y jouer
- pour savoir vraiment programmer en C, cf. cours du slot 2

To be continued...