Programmation C

Préprocesseur, modificateurs et fonctions avancées

ESIPE Info 1ère année

Université de Marne-la-Vallée

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` 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    int res;
6    if (n <= 1) { return 1; }
7    res = n*fact(n-1);
8    return res;
9  }
10
11 int main(int argc, char* argv[]){
12   printf("%d\n", fact(5));
13   return 0;
14 }
```

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

Définition de macros

- on peut interdire certaines fonctions

```
1  #define fseek DONT_USE_FSEEK_BUT_FSETPOS
2  /* ... */
3  int main(int argc, char* argv[]){
4    FILE* f=fopen("foo", "r");
5    if (f == NULL) exit(1);
6    fseek(f, SEEK_END, 0);
7    /* ... */
8    return 0;
9  }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
test.c:8:3: attention : implicit declaration of
function 'DONT_USE_FSSEK_BUT_FSETPOS'
collect2: erreur: ld a retourné 1 code d'état d'exécution
```

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 :

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

- Attention aux effets de bord :

```
1  #define SQUARE(a) a*a
2
3  int main(int argc, char* argv[]){
4     int i=2;
5     printf("%d\n", SQUARE(i++));
6     /* i++ * i++ : i incremente deux fois ? */
7     return 0;
8  }
```

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

```
1  #undef MAX
2
3  int MAX(int a, int b){
4    return (a>b)?a:b;
5  }
```

#expr

- avec un # devant un paramètre, 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

La date et l'heure sont celle de la compilation du fichier :

```
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!
```

Un profiler de code en C

- Suivant que l'on veut déboguer ou même profiler 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 nulle
- si oui, affiche un message sur `stderr` et quitte le programme très brutalement avec `abort`
- désactiver 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;
}
```

```
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

```
1  void swap(const char** const argv){
2    const char* tmp;
3    tmp = argv[1];
4    argv[1] = argv[2];
5    argv[2] = tmp;
6  }
7
8  int main(int argc, char* argv[]){
9    int i;
10   swap((const char** const)argv);
11   for (i=0 ; i<argc ; i++)
12     printf("%s\n", argv[i]);
13   return 0;
14 }
```

```
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

```
1 void cpy(char* dest, const char* src){
2   while (*dest++ = *src++);
3 }
```

- `const` permet d'éviter des bogues

- à utiliser autant que possible

Classes de stockage

- `extern`
- déclare quelque chose sans le définir, ni réserver d'espace
- utile pour partager une variable entre plusieurs `.c`

```
1  /* My religion forbids me to include
2     stdxxx headers. I must rewrite them...  */
3  extern int printf(const char *, ...);
4
5  int main(int argc, char* argv[]){
6    printf("Hello\n");
7    return 0;
8  }
```

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 :

```
1  #ifndef __CONST_H__
2  #define __CONST_H__
3
4  int MODE=12;
5
6  #endif
```

main.c :

```
1  #include <stdio.h>
2  #include "const.h"
3
4  extern void foo(void);
5
6  int main(int argc, char* argv[]){
7    foo();
8    return 0;
9  }
```

foo.c :

```
1  #include "const.h"
2
3  void foo(void){
4    MODE=MODE+5;
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 :

```
1  #ifndef __CONST_H__
2  #define __CONST_H__
3
4  extern int MODE;
5
6  #endif
```

main.c :

```
1  #include <stdio.h>
2  #include "const.h"
3
4  extern void foo(void);
5
6  int main(int argc, char* argv[]){
7    foo();
8    return 0;
9  }
```

foo.c :

```
1  #include "const.h"
2
3  void foo(void){
4    MODE=MODE+5;
5  }
```

const.c :

```
1  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;
6    if ((current=time(NULL))==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)

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  }
```

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)`

- C'EST AU PROGRAMMEUR DE SAVOIR QUEL EST LE TYPE DU PROCHAIN PARAMÈTRE !!!

  $\Rightarrow$ Ce qui vous explique le coût et la nécessité d'un parsing d'un format d'entrée/sortie pour la printf/scanf family.

- quand on a fini, on doit appeler une (et une seule fois) `va_arg(va_end)`

- La documentation précise que `va_arg(va_end)` peut être une fonction ou même une macro... Étant donné le danger des macros, pensez bien à fermer le parcours des arguments des fonctions variadiques

Formatage de date

```
 1  void print_date(const char* fmt, ...){
 2    if (fmt==NULL) return ;
 3    va_list args;
 4    va_start(args, fmt);
 5    int i=0, d;
 6    while (fmt[i]!='\0'){
 7      switch (fmt[i]){
 8        case 'D': case 'M': case 'Y':
 9        d = va_arg(args, int);
10        if (fmt[i]=='Y' && fmt[i+1]=='Y'){
11          printf("%04d", d);
12          i++;
13        } else printf("%02d", d%100);
14        break;
15        default: printf("%c", fmt[i]);
16      }
17      i++;
18    }
19    va_end(args);
20  }
21
22  int main(int argc, char* argv[]){
23    print_date("D/M/Y\n", 4, 11, 2006);
24    print_date("date: D-M-YY\n", 4, 11, 2006);
25    return 0; }
```

```
nborie@perceval:~> gcc -o test test15.c -Wall -ansi
nborie@perceval:~> ./test
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 une `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

```
1  /* Parse the given format string supposed to be of the form
2     %[ 0][n]b. Return 0 if fmt is not correct, 1 otherwise */
3  int parse_format(const char* fmt, char* fill, int* min){
4    if (fmt==NULL || fmt[0]!='%'){
5      return 0;
6    }
7    int i=1;
8    *fill='\0';
9    if (fmt[i]==' ' || fmt[i]=='0'){
10     *fill=fmt[i++];
11   }
12   *min=0;
13   if (fmt[i]>='1' && fmt[i]<='9'){
14     do{
15       *min=(*min)*10+fmt[i++]-'0';
16     }while(fmt[i]>='0' && fmt[i]<='9');
17   }
18   if (fmt[i]!='b' || fmt[i+1]!='\0'){
19     return 0;
20   }
21   return 1;
22 }
```

Afficher en binaire

```
1  int print_bin(const char* fmt, unsigned int d){
2    char fill;
3    int n;
4    if (!parse_format(fmt, &fill, &n)){
5      return -1;
6    }
7    int n_digits=get_n_digits(d);
8    int i, ret=n_digits;
9    if (fill!='\0' && n_digits<n){
10     ret=n;
11     for (i=0 ; i<n-n_digits ; i++){
12       printf("%c", fill);
13     }
14   }
15   for (i=n_digits-1 ; i>=0 ; i--){
16     printf("%c", (d&(1<<i))?'1':'0');
17   }
18   return ret;
19 }
```

Personnaliser printf

- deuxième étape :

  - on voudrait que

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

  - puisse s'écrire de façon plus pratique :

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

Personnaliser printf

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

my_printf (1/2)

```
 1  int my_printf(const char* fmt, ...){
 2    if (fmt==NULL) return -1;
 3    va_list args; va_start(args, fmt);
 4    int i=0, n=0;
 5    while (fmt[i]!='\0'){
 6      if (fmt[i]!='%'){ /* normal char */
 7        if (EOF==fputc(fmt[i], stdout)) return -1;
 8        n++;
 9      } else {
10        switch(fmt[++i]) {
11          case '\0': return -1;
12          case '%': if (EOF==fputc(fmt[i], stdout)) return -1;
13                    n++; break;
14          default: { /* classic case of %xxx */
15            i--; /* going back to the char % */
16            int z=0;
17            char fmt2[64];
18            do{
19              fmt2[z++]=fmt[i++];
20            } while(z<64 && fmt2[z-1]!='\0'
21                  && !strchr("diouxXeEfgcspb", fmt2[z-1]));
22            if (z==64) return -1;
23            i--;
24            if (fmt2[z-1]=='\0') return -1;
25            fmt2[z]='\0';
```

my_printf (2/2)

```
1          int ret;
2          switch (fmt2[z-1]){
3            case 'd': case 'i': case 'o': case 'u': case 'x': case 'X':
4              {int k=va_arg(args, int); ret=printf(fmt2, k); break;}
5            case 'f': case 'g': case 'e': case 'E':
6              {double d=va_arg(args, double); ret=printf(fmt2, d); break;}
7            case 'c':
8              {char c=va_arg(args, int); ret=printf(fmt2, c); break;}
9            case 's':
10             {char* s=va_arg(args, char*); ret=printf(fmt2, s); break;}
11           case 'p':
12             {void* p=va_arg(args, void*); ret=printf(fmt2, p); break;}
13           case 'b':
14             {unsigned int value=va_arg(args, unsigned int);
15             ret=print_bin(fmt2, value); break;}
16           }
17           if (ret<0) return -1;
18           n=n+ret;
19         }}}i++;} va_end(args);
20       return n;
21     }
```

my_printf : ça marche !!!

```
1  int main(int argc, char* argv[]){
2    my_printf("%d en decimal = %b en binaire\n", 5, 5);
3    my_printf("%d en decimal = % 6b en binaire\n", 5, 5);
4    my_printf("%d en decimal = %06b en binaire\n", 5, 5);
5    return 0;
6  }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
nborie@perceval:~> ./test
5 en decimal = 101 en binaire
5 en decimal =    101 en binaire
5 en decimal = 000101 en binaire
```

Sans chaîne de formatage

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

```
1  /* Like strcat but optimized for multiple strings.
2     The last parameter must be NULL. 'dest' is
3     supposed to be large enough. */
4  void my_strcat(char* dest, ...){
5    va_list args;
6    va_start(args, dest);
7    dest=dest+strlen(dest);
8    char* s;
9    while ((s=va_args(args, char*)) != NULL){
10     while ((*dest++=*s++));
11       dest--;
12   }
13   va_end(args);
14 }
```

Quelques conseils

- Ce cours est à lire et à relire calmement pour être digéré...
- Connaître les règles des échecs ne suffit pas pour savoir y jouer.
- Pour maîtriser un langage, connaître sa syntaxe et ses fonctions de bases ne suffit pas.
- C'est plus l'appréhension de ses paradigmes et la compréhension de son esprit qui fait le bon programmeur.
- S'il y a 1000 façon de résoudre un problème dans un langage, il en reste 100 fois moins si on impose de respecter son esprit.