Pratique du C
Bétisier

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2013-2014

Règles du jeu

Les bouts de codes se prêtent aux commentaires.
On cherche à savoir si ces codes

- provoquent une erreur à
  - la compilation,
  - l'exécution ;
- font ce qu'il devrait.

```
#define MAX = 10 ;
int t[MAX], x = MAX ;
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#define MAX = 10 ;
int t[MAX], x = MAX ;
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

Une macro n'est ni une déclaration ni une initialisation mais provoque une substitution textuelle.

```
#define add (a,b) (a + b)

int main(void){
   return add(1,2) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#define add (a,b) (a + b)

int main(void){
   return add(1,2) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

L'espace est le séparateur entre l'identificateur de macro et la chaîne à substituer.

```
#define CARRE(a) ((a) * (a))
int main(void){
   int x = 2 ;
   return CARRE(x++) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#define CARRE(a) ((a) * (a))
int main(void){
   int x = 2 ;
   return CARRE(x++) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Même bien constituée une macro n'immunise pas contre l'effet latéral.

```
/* On veut retourner 1 */
int main(void){
   int a=0, b=0, res ;

   if (a)
      if (b)  res = !b ;
   else  res = !a ;

   return res ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
/* On veut retourner 1 */
int main(void){
   int a=0, b=0, res ;

   if (a)
      if (b)  res = !b ;
   else  res = !a ;

   return res ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : oui
- fait ce qu'il devrait :

Il est difficile de savoir à quel if se vouer, alors autant utiliser des blocs.

```
int main(void){
  char c ;
  while (c = getchar() != EOF) putchar(c) ;
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void){
  char c ;
  while (c = getchar() != EOF) putchar(c) ;
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

Il faut rendre lisible le code plutôt que de compter sur la priorité entre les opérateurs.

```
/* On veut parcourir une cha\^ine de caract\‘eres */
int
main
(void)
{ int i ;
  char * bibi="La vie est belle" ;
  for(i=0;i<strlen(bibi);i++);
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
/* On veut parcourir une cha\^ine de caract\‘eres */
int
main
(void)
{ int i ;
  char * bibi="La vie est belle" ;
  for(i=0;i<strlen(bibi);i++);
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

C'est un parcours en n^2 ! ! ! ! ! ! ! À éviter impérativement.

Utilisez les fonctions de la libraire standard à loisir après les avoir comprises (i.e. après votre cours de pratique des systèmes).

```
int main(void){
  int a=1,b=1 ;
  if(a=0)
     printf("Attention z\\'ero") ;
  else b /= a ;
  return b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void){
  int a=1,b=1 ;
  if(a=0)
     printf("Attention z\\'ero") ;
  else b /= a ;
  return b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

L'affectation est un opérateur et non une instruction.

```
/* on veut retourner 0 */
int main(void){
   char a=1,b=0 ;
   switch(a){
      case 1 : a = b;
      defau1t : return 1 ;
   }
   return a  ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
/* on veut retourner 0 */
int main(void){
   char a=1,b=0 ;
   switch(a){
      case 1 : a = b;
      defau1t : return 1 ;
   }
   return a  ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

L'existence du `goto` et l'usage du `break` induisent cette erreur difficilement détectable (sans l'aide des avertissements du compilateur).

```
int tab[12] = {
   1, 2, 3, 4,
   5, 6, 7, 8,
   9,10,11,12
 } ;

int main(void){ /* on veut retourner 3 */
  int i=2,j=2 ;
  return tab[i++,j++] ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int tab[12] = {
   1, 2, 3, 4,
   5, 6, 7, 8,
   9,10,11,12
 } ;

int main(void){ /* on veut retourner 3 */
  int i=2,j=2 ;
  return tab[i++,j++] ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

La virgule est l'opérateur qui délivre comme résultat l'opérande droit après avoir évalué l'opérande gauche.

```
/* On veut retourner 2 au shell */
int main(void){
    return fct(2) ;
}

int fct(int i){
    return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
/* On veut retourner 2 au shell */
int main(void){
    return fct(2) ;
}

int fct(int i){
    return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

La fonction `fct` devrait être déclarée par un prototype avant la fonction principale.

```
int pair(int i){
   if (i)
      return impair(i-1) ;
   return 1 ;
}

int impair (int i){
   if (i)
      return pair(i-1) ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int pair(int i){
   if (i)
      return impair(i-1) ;
   return 1 ;
}

int impair (int i){
   if (i)
      return pair(i-1) ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

Encore une fois, un prototype permet de fournir au compilateur la déclaration qui lui manque.

```
typedef int fct_t (int) ;

fct_t fct ;

int main(void){
     return fct(2) ;
}

int fct(int i){
    return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
typedef int fct_t (int) ;

fct_t fct ;

int main(void){
     return fct(2) ;
}

int fct(int i){
    return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

Un prototype de fonction est une déclaration.

```
int fct(int i){
    return i ;
}

int main(void){
     fct(2) ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int fct(int i){
    return i ;
}

int main(void){
     fct(2) ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

Même si la fonction `fct` retourne une valeur, rien ne nous force à la récupérer.

```
int MaVar = 1 ;

int main(void){
   return ++Mavar ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int MaVar = 1 ;

int main(void){
   return ++Mavar ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

Les identificateurs sont sensibles à la casse.

```
int var(void){ return 2;}
int main(void){
    int a = 10, var = 1, cinq = a/var ;
    return cinq ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int var(void){ return 2;}
int main(void){
    int a = 10, var = 1, cinq = a/var ;
    return cinq ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

La porté des identificateurs n'est pas un vain mot.

```
int deux(void){ return 2 ; }

int main(void){
    int (*deux) = deux ;
    return deux ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int deux(void){ return 2 ; }

int main(void){
    int (*deux) = deux ;
    return deux ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Ce code retourne une adresse.

On souhaite retourner la somme de deux entiers saisis au clavier.

```
int
main(void){
    int a=0, b=0 ;
    scanf("%d%d",a,b) ;
    return a+b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

On souhaite retourner la somme de deux entiers saisis au clavier.

```
int
main(void){
    int a=0, b=0 ;
    scanf("%d%d",a,b) ;
    return a+b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Le passage de paramètre par référence permet d'exporter de l'information d'une fonction.

```
int SommeDes10Premiers(int tab[10]) {
int i, res =0 ;
for(i=1;i<11;i++)
   res += tab[i] ;
   return res ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int SommeDes10Premiers(int tab[10]) {
int i, res =0 ;
for(i=1;i<11;i++)
   res += tab[i] ;
   return res ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Les indices de tableaux commencent à 0 et rien ne vous empêche de passer outre.

```
int main(void){
     int zero = 0 ;
     int tab[5] = {1,2,3,4,10} ;
     zero = 0[tab] + *(tab+1) + *(2+tab) + tab[3] - 4[tab];
     return zero ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void){
     int zero = 0 ;
     int tab[5] = {1,2,3,4,10} ;
     zero = 0[tab] + *(tab+1) + *(2+tab) + tab[3] - 4[tab];
     return zero ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

La notation `tab[1]` est équivalente à `*(tab+1)` ce qui permet ce code surprenant.

```
#include<stdio.h>
int main(void){
     int i = 7, a[5]; char c = 0 ;
/* qu'est qui est affich\'e */
     printf( "%d", i++ * i++ );
/* comment est modifi\'e ce tableau */
     i = 1; a[i] = i++;
/* ce code peut il s'arr\^eter */
     do { c = getchar(); } while(c != EOF) ;
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#include<stdio.h>
int main(void){
     int i = 7, a[5]; char c = 0 ;
/* qu'est qui est affich\'e */
     printf( "%d", i++ * i++ );
/* comment est modifi\'e ce tableau */
     i = 1; a[i] = i++;
/* ce code peut il s'arr\^eter */
     do { c = getchar(); } while(c != EOF) ;
  return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : ? ? ?

Si on ne lit pas la documentation du compilateur ou l'assembleur, il existe des questions sans réponses i.e. la norme ne spécifie pas tout.

```
/* ce code retourne 100 */
int main(void){
   i=0 ; j=1 ;
   while(i<100) ;
      i=i+j ;
   return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
/* ce code retourne 100 */
int main(void){
   i=0 ; j=1 ;
   while(i<100) ;
      i=i+j ;
   return i ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Attention aux trivialités.

```
char foo[10] = '\0' ;
char bar[10] = "\0" ;

int * ptr1, * ptr2  ;
int * pptr1,  pptr2 ;
pptr1 = ptr1 ;
pptr2 = ptr2 ;
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
char foo[10] = '\0' ;
char bar[10] = "\0" ;

int * ptr1, * ptr2  ;
int * pptr1,  pptr2 ;
pptr1 = ptr1 ;
pptr2 = ptr2 ;
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

De petites différences produisent de grands effets.

```
int *a[10] ;
int (*b)[10] ;
b[9]=a[0] ;
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int *a[10] ;
int (*b)[10] ;
b[9]=a[0] ;
```

Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

La compilation signale error : incompatible types in assignment. En effet, le pointeur d'entier a[0] pourrait "avoir" plus de 10 cellules.

```
int main(void){
/* Emboitememement de /* commentaire */ */
int a,b,c = 3;
int *pointer = &c ;
   /*divise c par lui m\^eme */
   a=c/*pointer ;
   b=c /*met b \‘a 3*/ ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void){
/* Emboitememement de /* commentaire */ */
int a,b,c = 3;
int *pointer = &c ;
   /*divise c par lui m\^eme */
   a=c/*pointer ;
   b=c /*met b \‘a 3*/ ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Attention aux commentaires.

```
#include<stdio.h>
char *foo(void){
    char str[31]="Pourquoi vais-je disparaitre~?" ;
    return str ;
}

int main(void){
   printf("%d %d %s\n",1,2,foo()) ;
   return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#include<stdio.h>
char *foo(void){
    char str[31]="Pourquoi vais-je disparaitre~?" ;
    return str ;
}

int main(void){
   printf("%d %d %s\n",1,2,foo()) ;
   return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Le compilateur indique que `warning: function returns address of local variable`.

```
int
main
(void)
{
   char *str = "stringX" ;
   str[6]='Y' ;
   return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int
main
(void)
{
   char *str = "stringX" ;
   str[6]='Y' ;
   return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

La chaîne "stringX" est stockée dans un segment en lecture seule (.rodata) et la tentative d'accès se solde par un `segmentation fault`.

```
#include<stdio.h>
struct tab_s{  int tab[2] ; } ;

struct tab_s foo(void){
    struct tab_s res ;
    res.tab[0] = 1 ; res.tab[1] = 2 ;
    return res ;
}

int main(void){
   struct tab_s tmp = foo() ;
   int zero = 2*tmp.tab[0] - tmp.tab[1] ;
   return  zero ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
#include<stdio.h>
struct tab_s{  int tab[2] ; } ;

struct tab_s foo(void){
    struct tab_s res ;
    res.tab[0] = 1 ; res.tab[1] = 2 ;
    return res ;
}

int main(void){
   struct tab_s tmp = foo() ;
   int zero = 2*tmp.tab[0] - tmp.tab[1] ;
   return  zero ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : oui

On ne peut affecter un tableau à un autre, c'est possible pour les structures.

Ce code retourne la taille de la chaîne de caractères

```
int main(void){
   char *ch1 = "Hello world";
   return sizeof(ch1) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

Ce code retourne la taille de la chaîne de caractères

```
int main(void){
   char *ch1 = "Hello world";
   return sizeof(ch1) ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

N'utilisez `sizeof` que sur des types.

Une implantation de `memset`

```
void *memset(void *b, int c, int len)
{
  int i ;
  for(i=0;i<len;i++)
     b[i]=c ;
  return b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

Une implantation de `memset`

```
void *memset(void *b, int c, int len)
{
  int i ;
  for(i=0;i<len;i++)
     b[i]=c ;
  return b ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution :
- fait ce qu'il devrait :

Ni arithmétique ni déréférencement de pointeur void .

```
int main(void){
   int *p ;
   p = malloc(10*sizeof (int));
   p[10] = 666 ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void){
   int *p ;
   p = malloc(10*sizeof (int));
   p[10] = 666 ;
   return 0 ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Affectation hors limite.

```
#include<stdio.h>
#include <stdlib.h>

/* strtod, strtof, strtold - convert
ASCII string to floating point number
float strtof(const char *, char **); */

int main(int argc, char **argv){
  printf("%f\n", strtof(argv[1], NULL)) ;
  return 0 ;
}
```

Si on compile avec l'option ansi, le prototype de la fonction strtof — n'étant pas dans cette norme et étant à l'intérieur d'une directive conditionnelle — n'est pas pris en compte. Sans prototype, la valeur de retour de la fonction est supposée être un entier machine et le résultat n'est pas codé comme un flottant et donc faux.

```
int main(void) {
int i ;
int tampon[5] ;

for(i=0;i<666;i++)
tampon[i]=2 ;
return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int main(void) {
int i ;
int tampon[5] ;

for(i=0;i<666;i++)
tampon[i]=2 ;
return 0;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation : non
  - l'exécution : non
- fait ce qu'il devrait : non

Cela provoque une boucle infinie ! ! !

Le code

```
int main(void){
    int tab[10] ;
    int i = 0 ;
    for(;i<300;i++)
        tab[i] = i ;
    return 0 ;
}
```

donne l'exécution suivante :

```
espoir % gcc code.c
espoir % a.out
Segmentation fault
```

```
int tab[10]; /* dans le fichier foo */
.....
extern int * tab ; /* dans le fichier bar */
```

Il n'y a pas d'allocation de mémoire lors de la déclaration d'un pointeur (hormis la mémoire utilisée pour contenir l'adresse).

```
extern foo; /* ceci est valide (c'est un entier) */
```

Il existe beaucoup de règles implicites en C.

```
int bar(int n) { return n+2 ;}
int
main
(void)
{
   int (*foo) (int) = bar ;
   foo++ ;
   return foo(1)
}
```

Quid de l'arithmétique des pointeurs de fonctions ?

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int bar(int n) { return n+2 ;}
int
main
(void)
{
   int (*foo) (int) = bar ;
   foo++ ;
   return foo(1)
}
```

Quid de l'arithmétique des pointeurs de fonctions ?
Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

Comme on ne connait pas la taille d'une fonction mais seulement son adresse et que cette taille est variable pour de fonctions de même signature, impossible de faire de l'arithmétique des pointeurs de fonctions.

```
int
bar
(int n)
{
   int tab[n] ;

   /* comparer avec malloc */
   return ;
}
```

Ce bout de code

- provoque une erreur à
  - la compilation :
  - l'exécution :
- fait ce qu'il devrait :

```
int
bar
(int n)
{
   int tab[n] ;

   /* comparer avec malloc */
   return ;
}
```

Il n'y a pas que l'ANSI dans la vie.Ce bout de code

- provoque une erreur à
  - la compilation : oui
  - l'exécution :
- fait ce qu'il devrait :

Ce code est valide en C99.