Programmation C

Fonctions et passages d'adresses

ESIPE Info 1ère année

Université de Marne-la-Vallée

Les fonctions

- une fonction a un prototype :
  - un nom (identificateur)
  - des paramètres nommés et typés
  - un type de retour

- exemple de fonction :

```
1  int average(int t[], int n){
2    float sum=0;
3    int i;
4    for (i=0 ; i<n ; i++){
5      sum += t[i];
6    }
7    return sum/n;
8  }
```

Le type `void`

- type spécial :
  - en valeur de retour → pas de valeur de retour

```
1 void print_sum(int a, int b){
2     /* ... */
3 }
```

  - en paramètre → pas de paramètre (facultatif)

```
1 char get_upper_char(void){
2     /* ... */
3 }
```

ou bien

```
1 char get_upper_char(){
2     /* ... */
3 }
```

Définition VS déclaration

- une fonction doit être déclaré avec d'être appelée
- définition = code de la fonction
- déclaration = juste son prototype avec `;`
- prototype dans les header `.h`
- prototype dans un fichier de code `.c` pour les fonctions qu'on ne veut pas exporter

```
1  void get_oeuf(void);
2
3  void get_poule(void){
4     /* ... */
5     get_oeuf();
6     /* ... */
7  }
8
9  void get_oeuf(void){
10    /* ... */
11    get_poule();
12    /* ... */
13 }
```

return

- quitte la fonction
- renvoie une valeur si retour ≠ `void`
- non obligatoire seulement pour les fonctions dont le type de retour est `void`

```
1  int print_sum2(int a, int b){
2    printf("%d + %d = %d\n", a, b, a+b);
3  }
```

donne

```
attention : contrôle a atteint la fin non void de la
fonction [-Wreturn-type]
```

Valeurs de retour

- on peut ignorer une valeur de retour :
  - `printf`, `scanf`

- on ne peut pas utiliser une fonction `void` dans une expression :

```
1  void print_sum(int a, int b){
2    printf("%d + %d = %d\n", a, b, a+b);
3  }
4
5  int main(int argc, char* argv[]){
6    float f=print_sum(5,6);
7    printf("essai : %f\n", f);
8    return 0;
9  }
```

donne

```
erreur: valeur void n'a pas été ignorée comme elle aurait dû l'être
```

Le cas de `main`

- fonction particulière :

  - `return` quitte le programme
  - renvoie le code de retour du programme
  (par convention, on a 0 = OK, $\neq 0$ = erreur)

- si `main` est appelé explicitement par une fonction, son `return` fonctionne alors normalement.

- un programme se termine ”normalement” lorsque la pile d’appel de fonctions est vide. Il faut ainsi fermé le `return` de l’appel originel de `main` (celui qui est fait par la ligne de commande lors de l’appel de l’exécutable).

Paramètres de `main`

- `int main(int argc, char* argv[])`
- `argc` : nombre de paramètres, y compris l'exécutable
- `argv[]` : tableau de chaînes, contenant les paramètres (en `bash`, pas de type : tout est chaîne de caractère)

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

Écrire une fonction

- réfléchir à l'utilité de la fonction
- 1 fonction = 1 seule tâche
- Ne pas mélanger calcul et affichage

```
1  int minimun(int a, int b){
2     int min=(a<b)?a:b;
3     printf("min = %d\n", min);
4     return min;
5  }
```

On fait de l'affichage dans des fonctions de type de retour `void` ou pour du débogage. Mais ici, c'est une faute de goût.

Les paramètres

- ne pas mettre trop de paramètres !
- encapsuler les paramètres trop nombreux

```
1  typedef struct{
2     int nb_largeur_pixel;
3     int nb_hauteur_pixel;
4     int background_color;
5     int set_option;
6     int ...
7  }ParamGraph;
8
9  void create_image(ParamGraph pg);
10 void change_image(ParamGraph pg, int action);
11 ...
```

Définir le prototype

- de quoi la fonction a-t-elle besoin ?
- retourne-t-elle quelque chose ?
- y a-t-il des cas d'erreurs ?
- si oui, 3 solutions :

  - mettre un commentaire
  (bof, solution du feignant...)

  - renvoyer un code d'erreur
  (qu'on peut ré-exploiter après)

  - afficher un message et quitter le programme
  (cas les plus critiques)

Le commentaire

- solution du programmeur pressé : utile quand la fonction n'est pas censée être appelée dans certains cas:

```
1 /* Copies the array 'src' to the 'dest' one.
2    'dest' is supposed to be large enought. */
3 void copy(int src[], int dest[]);
```

ou encore

```
1 /* returns 1 if 'w' is an English word,
2    returns 0 otherwise.
3    'w' is not supposed to be NULL. */
4 int is_english_word(char* w);
```

Le code d'erreur

- pas de problème si la fonction n'a pas vocation à retourner quelque chose.

```
1  int init(int t[], int size){
2    int i;
3    if (size <= 0)
4      return 0;
5    for (i=0 ; i<size ; i++)
6      t[i] = 0;
7    return 1;
8  }
```

- sinon, attention à la valeur choisie ! Il faut pouvoir distinguer un cas d'erreur d'une utilisation normale.

```
1  int main(int argc, char* argv[]){
2    printf("%s --> %d\n", argv[1], atoi(argv[1]));
3    return 0;
4  }
```

donne

```
nborie@perceval:> ./a.out 0
0 --> 0
nborie@perceval:> ./a.out foo
foo --> 0
```

Le code d'erreur

- Attention à la valeur de retour

```
1  /* Returns the minimum of the array 't'
2     or '-1' if 't' is empty */
3  int minimun(int t[], int size);
```

⇒ Impossible de savoir si −1 est une erreur ou bien réellement le maximum de `t`.

```
1  int str_length(char* s){
2    int i;
3    if (s == NULL)
4      return -1;
5    for (i=0 ; s[i]!='\0' ; i++) {}
6      return i;
7  }
```

⇒ Ok, −1 indique que l'argument n'était pas une vraie chaîne C.

Le code d'erreur

- lorsque toutes les valeurs possibles du type naturel de sortie correspondent potentiellement à un fonctionnement normal, on peut faire un passage par adresse.

```
1  int quotient(int a, int b, int* res){
2    if (b == 0)
3      return 0;
4    *res = a/b;
5    return 1;
6  }
```

Le retour de la fonction indique la viabilité de la valeur pointée par `res`.

L'interruption du programme

- À n'utiliser que dans des cas **TRÈS TRÈS GRAVES** !

  - Plus de mémoire disponible

  - erreurs d'entrées/sorties dangereuses pour la suite du programme

  - mauvais paramètres donnés au programme et risques pour la suite

- message d'erreur sur `stderr` + `exit(≠ 0)`

Dans la plupart des cas, on préférera ne pas quitter brutalement le programme mais retourner dans le `main` qui peut, par exemple, arrêter l'algorithmique et sauvegarder les résultats partiels calculés.

Tests d'erreur

- quitter dès que possible les fonctions quand il y a des erreurs, on évite ainsi les `else` inutiles

```
1  int res;
2  if (a == NULL)
3    res = -1;
4  else if ((*a).value < 0)
5    res = -1;
6  else{
7    res = (*a).value;
8    /* ... */
9    }
10 return res;
```

est plus lisible et plus efficace comme

```
1  int res;
2  if (a == NULL)
3    return -1;
4  if ((*a).value < 0)
5    return -1;
6  res = (*a).value;
7  /* ... */
8  return res;
```

Esthétique des fonctions

- fait : on lit 10 fois un code écrit 1 fois
- soignez la présentation:
  - commentaires
  - noms explicites des fonctions et variables
  - indentation
- regroupez les fonctions de même thème dans les `.c`

Passage d'adresse

```
1  void add_3(int a){
2    a = a+3;
3  }
4
5  int main(int argc, char* argv[]){
6    int foo = 14;
7    add_3(foo);
8    printf("foo = %d\n", foo);
9    return 0;
10 }
```

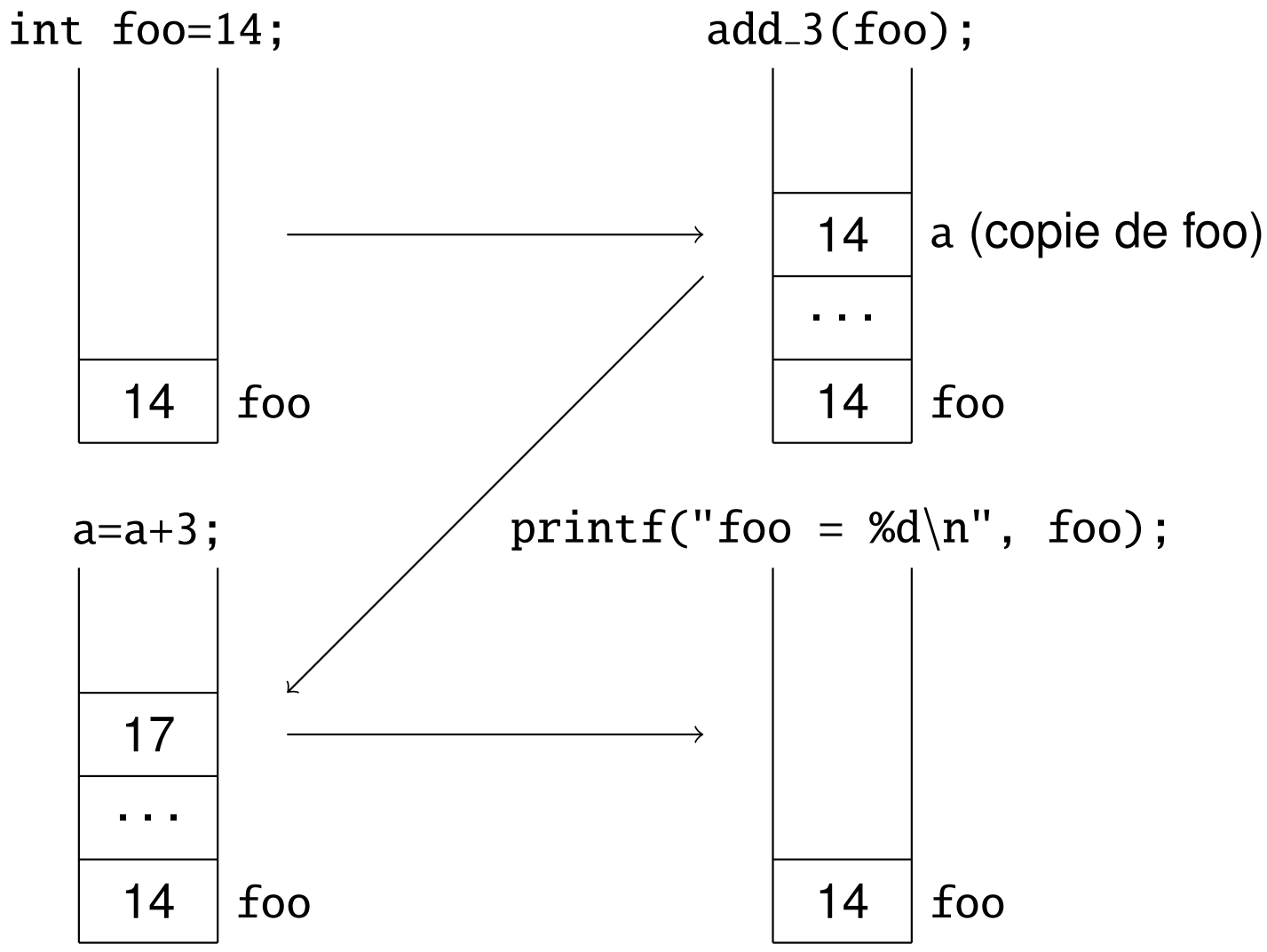
donne

```
$> ./a.out
foo = 14
```

- Pourquoi ?

Passage d'adresse

- Regardons la pile !



Passage d'adresse

- comment faire ?

  $\Rightarrow$ donner l'adresse de `foo` pour pouvoir modifier le contenu de la zone mémoire correspondante

- opérateur d'adressage `&`:

  - `&abc` = adresse mémoire de la variable `abc`

  - comme dans `scanf`, qui a besoin de savoir où stocker les résultats de la saisie

Passage d'adresse

- pour indiquer qu'une fonction reçoit une adresse, on met `*` entre le type et le nom du paramètre :

  ```
  void copy(int a, int*b);
  ```

- `b` représente l'adresse d'un `int`
- `*b` est le contenu de la zone mémoire d'adresse `b`
- la taille de la zone dépend du type :

  `*a` différent dans `char *a` et `float *a`

Passage d'adresse

```
1  void add_3(int *a){
2    *a = *a+3;
3  }
4
5  int main(int argc, char* argv[]){
6    int foo = 14;
7    add_3(&foo);
8    printf("foo = %d\n", foo);
9    return 0;
10 }
```

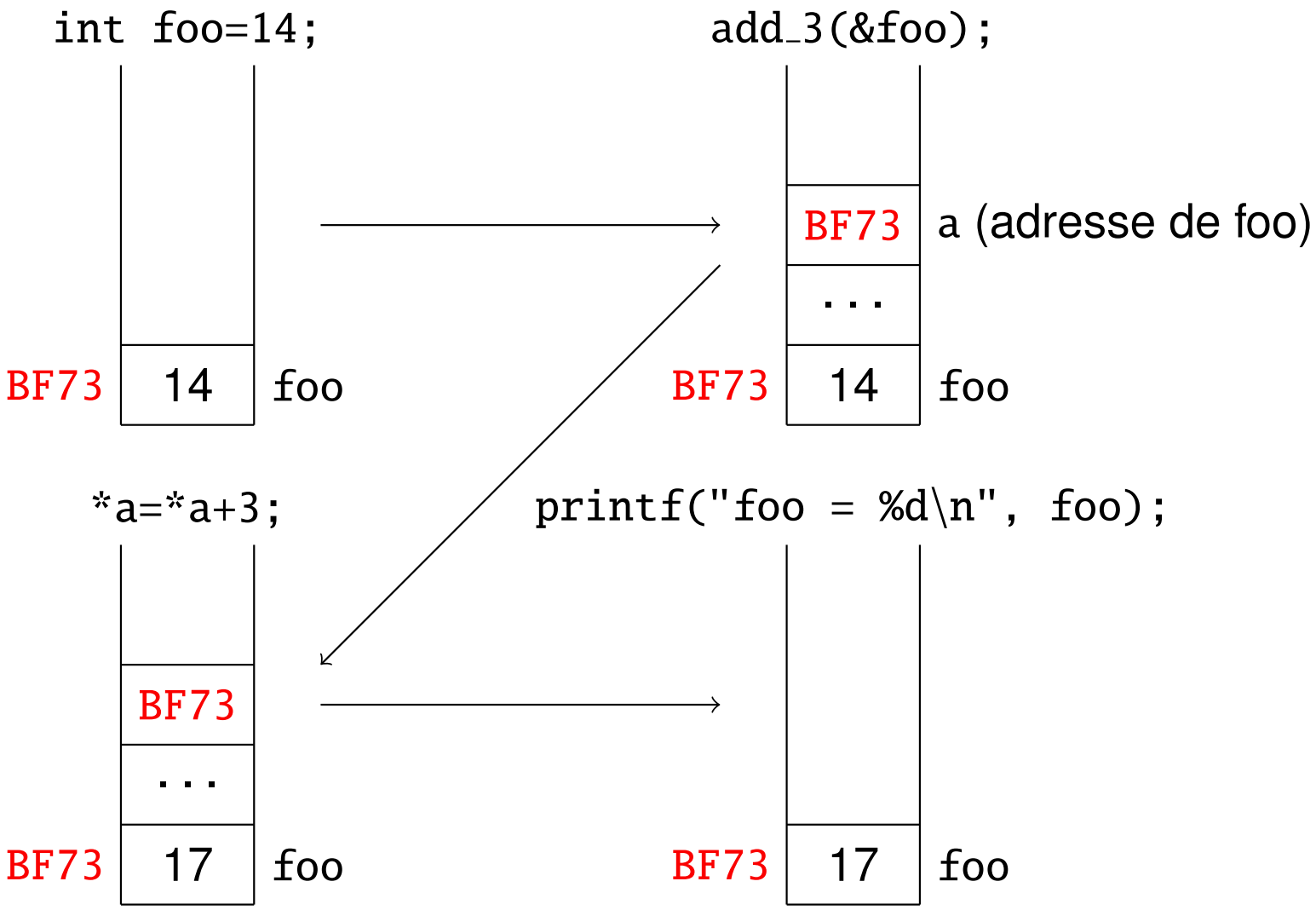
donne

```
$> ./a.out
foo = 17
```

- ça marche !

Passage d'adresse

- Regardons la pile !



Passage d'adresse

- Utile

  - quand on doit modifier un paramètre

  - quand on doit retourner plusieurs valeurs

  - quand on doit retourner une ou plusieurs valeurs + un code d'erreur

Modifier une variable

- exemple classique : les compteurs
- attention : `*n++` n'est pas `(*n)++`

```
1  /* Reads from the given file the first character */
2  /* that is not a newline. Returns it or -1 if the */
3  /* end of file is reached. If there are newlines, */
4  /* '*n' is updated. */
5  int read_char(FILE* f, int *n){
6    int c;
7    while ((c=fgetc(f)) != EOF){
8      if (c=='\n') (*n)++;
9      else return c;
10   }
11   return -1;
12 }
```

Retourner plusieurs valeurs

- On utilise des passages par adresse sauf potentiellement pour une seule donnée.

```
1  /* compute the irreducible fraction x/y such that */
2  /* x/y = a1/b1 + a2/b2 */
3  void add_fractions(int a1, int b1, int a2, int b2,
4                                          int *x, int *y){
5    int p;
6    *x = a1*b2 + a2*b1;
7    *y = b1*b2;
8    p = gcd(*x, *y);
9    *x = *x / p;
10   *y = *y / p;
11 }
```

- pourquoi favoriser le numérateur ou le dénominateur ?
  $\Rightarrow$ ce serait une faute de goût de ne pas mettre les deux variables au même niveau.

avec un code d'erreur

- on choisit plutôt de retourner le code d'erreur et de passer par adresse les variables pour récupérer les résultats.

- l'écriture suivante s'inscrit complètement dans l'esprit du langage :

```
1    ...
2    if (!fonction_code_erreur(&arg1, &arg2, &arg3)){
3      fprintf(stderr, "Il aurait fallu des choses, "
4                      "des choses et des choses....\n");
5      return -1;
6    }
7    var = arg1 + arg2 +arg3;
8    ...
```

$\Rightarrow$ c'est un `if` avec effet de bord qui poursuit l'erreur ou laisse continuer l'algorithmique.

Fonction récursive

- fonction s'appelant elle-même.

```
 1  void hanoi(int n, char src,
 2             char tmp, char dest){
 3    if (n == 1){
 4      printf("%c --> %c\n", src, dest);
 5      return ;
 6    }
 7    hanoi(n-1, src, dest, tmp);
 8    printf("%c --> %c\n", src, dest);
 9    hanoi(n-1, tmp, src, dest);
10  }
11
12  int main(int argc, char* argv[]){
13    hanoi(3, 'A', 'B', 'C');
14    return 0;
15  }
```

```
$> ./a.out
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
```

Fonction récursive

- attention à la condition d'arrêt
- fait gonfler la pile
- à éviter s'il on peut faire facilement autrement

  `Python`, par default : 1000 appels récursifs $\Rightarrow$ `max depth of recursion`

  `Caml` gère la récursion terminale, pas le C.
- utilisation judicieuse : voir cours d'algo!

Fonction récursive

- Ne tester qu'une seule fois les cas d'erreurs

```
1  int sum(Tree* t, int *s){
2    if (t==NULL) return ERROR;
3      *s=*s+t->value;
4    if (t->left!=NULL)
5      sum(t->left,s);
6    if (t->rigth!=NULL)
7      sum(t->rigth,s);
8    return OK;
9  }
```

est mieux écrit comme

```
1  int sum_slave(Tree* t){
2    if (t==NULL) return 0;
3    return t->value+sum(t->left)+sum(t->rigth);
4  }
5
6  int sum(Tree* t, int *s){
7    if (t==NULL) return ERROR;
8    *s = sum_slave(t);
9    return OK;
10 }
```

Fonctions sur les chaînes

- Fonctions définies dans `string.h`

```
char* strcpy(char* dest, const char* src);
char* strcat(char* dest, const char* src);
size_t strlen(const char* s);
char* strdup(const char* s);
```

- `strcpy` copie, `strcat` concatène, `strlen` mesure, `strdup` duplique

- attention, aucun test sur la validité des chaînes
  - les chaînes ne doivent pas être `NULL`
  - `dest` doit être assez large
  - le caractère de fin de chaîne doit bien être placé

Fonctions sur les chaînes

Toujours bien consulter le manuel pour l'utilisation des fonctions !

```
1 void get_email(char* first, char* last, char* email){
2    strcpy(email, first);
3    strcat(email, ".");
4    strcat(email, last);
5    strcat(email, "@u-pem.fr");
6 }
```

Sans le caractère de fin de chaîne, l'utilisation des fonctions provoquent une erreur de segmentation dans le meilleur des cas ou donne des résultats aléatoires (suivant ce qui se trouve dans la mémoire non initialisée)

Comparer des chaînes

- `int strcmp(const char* a, const char* b);`

- retourne:
  - 0 si $a = b$
  - < 0 si $a < b$
  - > 0 si $a > b$

```
1  int main(int argc, char* argv[]){
2     printf("%d %d %d %d\n",
3     strcmp("ball", "ball"),
4     strcmp("ace", "ball"),
5     strcmp("basic", "ball"),
6     strcmp("ball", "balls"));
7  }
```

donne

```
0 -1 1 -1
```

Fonctions mathématiques

- définies dans `math.h`:
  - `double cos(double x);`
  - `double sin(double x);`
  - `double tan(double x);`
  - `double sqrt(double x);`
  - `double exp(double x);`
  - `double log(double x);`
  - `double log10(double x);`
  - `double pow(double x, double y);`
- compiler avec l'option `-lm`

Nombres aléatoires

- fonctions définies dans `stdlib.h`
- `int rand(void);`
  - retourne un entier entre `0` et `RAND_MAX`
- `void srand(unsigned int seed);`
  - initialise le générateur avec une graine
- pour une graine qui change:
  - `time_t time(time_t*);` dans `time.h`

Nombres aléatoires

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main(int argc, char* argv[]){
6     int i;
7     srand(time(NULL));
8     for (i=0 ; i<10 ; i++){
9        printf("%d ", rand()%20);
10    }
11    printf("\n");
12    return 0;
13 }
```

peut donner

```
$> ./test
6 4 4 10 15 10 13 0 6 6
$> ./test
16 7 9 8 5 3 14 2 11 17
```