Chapitre 4: Pointeurs

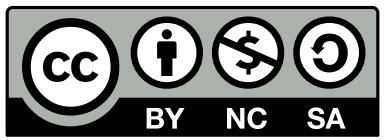
INF3135

Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v251



Plan

1. Adresse et pointeur
2. Tableaux et arithmétique des pointeurs
3. Fonctions
4. Entrées et sorties

Adresse et pointeur

Valeurs gauche et droite (*lvalue* et *rvalue*)

Syntaxe

Expression de la forme `L = R;`

Left-value ou *lvalue*

- Toute **expression** qui peut être placée à **gauche** de l'opérateur =
- Doit avoir un espace mémoire **réservé**
- **Exemples**: variable, champ d'une structure ou d'une union, pointeur

Right-value ou *rvalue*

- Toute autre **expression** valide
- Ayant une **valeur**

Adresse

- Les données stockées en mémoire ont une **adresse**
- L'opérateur `&` (esperluette) retourne l'adresse d'une *left-value*

```
#include <stdio.h>

int main() {
  int x = 210;
  printf("x = %d\n", x);
  printf("&x = %p\n", &x);
}
```

Résultat (le résultat peut varier):

```
x = 210
&x = 0x7ffe6b722f74
```

Adresses dans une structure (1/2)

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

struct Player {
  char fname[20];
  char lname[20];
  bool active;
  unsigned int rank;
  double winrate;
};

int main(void) {
  struct Player p = {"Novak", "Djokovic", true, 1, .95};
  printf("p.fname   = %-8s  &p.fname   = %p\n",  p.fname,   &p.fname);
  printf("p.lname   = %-8s  &p.lname   = %p\n",  p.lname,   &p.lname);
  printf("p.active  = %-8d  &p.active  = %p\n",  p.active,  &p.active);
  printf("p.rank    = %-8d  &p.rank    = %p\n",  p.rank,    &p.rank);
  printf("p.winrate = %-8lf  &p.winrate = %p\n", p.winrate, &p.winrate);
  return 0;
}
```

Adresses dans une structure (2/2)

Résultat (le résultat peut varier):

```
p.fname   = Novak     &p.fname   = 0x7ffead619e80
p.lname   = Djokovic  &p.lname   = 0x7ffead619e94
p.active  = 1         &p.active  = 0x7ffead619ea8
p.rank    = 1         &p.rank    = 0x7ffead619eac
p.winrate = 0.950000  &p.winrate = 0x7ffead619eb0
```

Pointeur

- **Pointeur**: *left-value* qui contient une adresse
- **Déclaré** à l'aide du symbole $*$

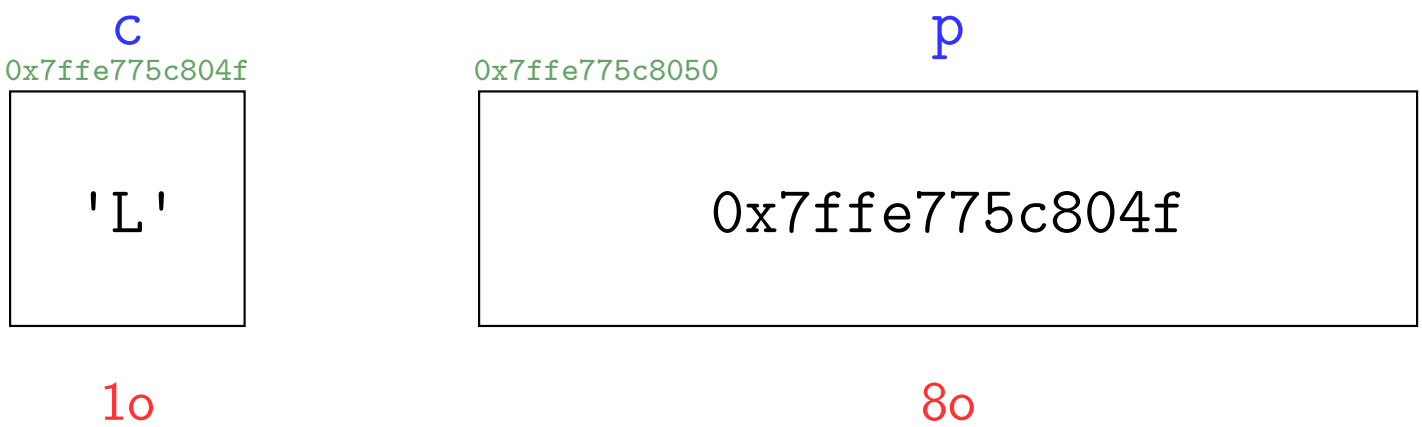
```
#include <stdio.h>

int main(void) {
  char c = 'L', *p = &c;
  printf("c          = %c\n", c);
  printf("&c         = %p\n", &c);
  printf("p          = %p\n", p);
  printf("&p         = %p\n", &p);
  printf("sizeof(c) = %ld\n", sizeof c);
  printf("sizeof(p) = %ld\n", sizeof p);
  return 0;
}
```

Résultat (le résultat peut varier):

```
c          = L
&c         = 0x7ffd6bf260ff
p          = 0x7ffd6bf260ff
&p         = 0x7ffd6bf26100
sizeof(c) = 1
sizeof(p) = 8
```

Représentation schématique

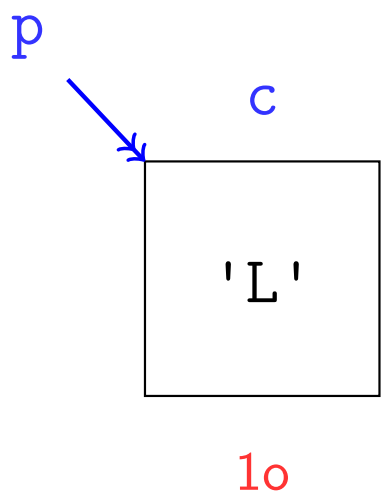
```
c           = L
&c          = 0x7ffd6bf260ff
p           = 0x7ffd6bf260ff
&p          = 0x7ffd6bf26100
sizeof(c) = 1
sizeof(p) = 8
```



Avec variables

Représentation schématique

```
c          = L
&c         = 0x7ffd6bf260ff
p          = 0x7ffd6bf260ff
&p         = 0x7ffd6bf26100
sizeof(c) = 1
sizeof(p) = 8
```



Avec flèches

Pointeurs typés

- **Exemples**: `char*`, `int*`, `double*`, `double**`, `enum answer*`, `struct player*`, `void*`, etc.
- **Intérêt?** détecter des erreurs à la compilation

```
int *p;         // Syntaxe préférée
int* p;         // Aussi acceptable
int *p1, *p2; // Deux pointeurs
int *p1, p2;  // p1 est un pointeur, p2 est un int
```

Plusieurs types de pointeurs

- **Pointeur nul**: identifié par la valeur `NULL`
- **Pointeur constant**: en lecture seule, écriture interdite
- **Pointeur générique**: `void*`
- **Pointeur de fonction**: permet de passer des fonctions en arguments

Espace mémoire

- Tous les pointeurs occupent le même **espace**
- Car contiennent des **adresses**

```
#include <stdio.h>

enum Answer { YES, NO, MAYBE };

int main(void) {
  char *c; int *i; double *d; enum Answer *a;
  printf("sizeof c = %ld\n", sizeof c);
  printf("sizeof i = %ld\n", sizeof i);
  printf("sizeof d = %ld\n", sizeof d);
  printf("sizeof a = %ld\n", sizeof a);
  return 0;
}
```

Résultat (le résultat peut varier selon l'architecture):

```
sizeof c = 8
sizeof i = 8
sizeof d = 8
sizeof a = 8
```

Le qualificatif `const`

- `const <type> *p`: pointeur en lecture seule (*read-only*)
- Le contenu pointé peut être **lu**
- Mais on ne peut pas **écrire**
- **Intérêt?** détecter une écriture non souhaitée à la compilation

```
const int *p;  // Pointeur constant vers un entier
const char *s; // Chaîne de caractère non modifiable
```

- `<type> *const p`: pointeur constant
- La **valeur** du pointeur ne peut pas être modifiée
- **Tableau**: souvent converti (*decay*) en pointeur constant

```
// La signature suivante
void initialize_array(double a[], unsigned int n);
// est équivalente à
void initialize_array(double *const a, unsigned int n);
```

Pointeur « constant »?

- `const <type> *p`: pointeur en lecture seule
- `<type> *const q`: pointeur constant

```c
#include <stdio.h>

int main(void) {
  int i;
  const int *p = &i; // Le pointeur p ne doit pas modifier
                     // le contenu qu'il référence
  *p = 13;           // On tente de modifier le contenu de i
  int *const q;      // Le pointeur q ne doit pas être modifié
  q = &i;            // On tente de le modifier
  return 0;
}
```

Résultat:

```
pointeur_const.c: In function ‘’main:
pointeur_const.c:7:8: error: assignment of read-only location ‘*’p
    7 |      *p = 13;           // On tente de modifier le contenu de i
      |        ^
pointeur_const.c:9:7: error: assignment of read-only variable ‘’q
    9 |      q = &i;            // On tente de le modifier
      |       ^
```

Déréférencement

- **Déréférencement**: accès à la donnée « pointée »
- À l'aide de l'**opérateur** $*$

```
#include <stdio.h>

int main(void) {
  double d = 1.5, *p = &d;
  printf("d      = %lf\n", d);
  printf("&d     = %p\n" , &d);
  printf("p      = %p\n" , p);
  printf("*p     = %lf\n", *p);
  printf("*(&d)  = %lf\n", *(&d));
  printf("&(*p)  = %p\n" , &(*p));
  return 0;
}
```

Résultat (le résultat peut varier):

```
d      = 1.500000
&d     = 0x7ffcae43c638
p      = 0x7ffcae43c638
*p     = 1.500000
*(&d)  = 1.500000
&(*p)  = 0x7ffcae43c638
```

Opérateur -> (sucre syntaxique) (1/2)

- Soit `p` un pointeur vers une structure ayant un champ `champ`
- Alors on peut écrire `p->champ` plutôt que `(*p).champ`
- Toujours préférer `p->champ` à `(*p).champ`

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

struct Point {
  double x;
  double y;
};

int main(void) {
  struct Point p = {-1.5, 2.3};
  struct Point *q = &p;
  printf("p.x = %-4lf  q->x = %-4lf  (*q).x = %-4lf\n",
         p.x, q->x, (*q).x);
  printf("p.y = %-4lf  q->y = %-4lf  (*q).y = %-4lf\n",
         p.y, q->y, (*q).y);
  return 0;
}
```

Opérateur -> (sucre syntaxique) (2/2)

Résultat (le résultat peut varier):

```
p.x = -1.500000  q->x = -1.500000  (*q).x = -1.500000
p.y = 2.300000  q->y = 2.300000  (*q).y = 2.300000
```

Autres opérations

- `=`: affectation
- `(type*)`: conversion (implicite ou explicite)

Comparaison

- `==`: égalité d'adresses
- `!=`: différence d'adresses
- `<=`, `>=`, `<`, `>`: comparaison d'adresses

Arithmétique

- `+`: pointeur décalé vers la « droite »
- `-`: pointeur décalé vers la « gauche »
- `++`: incrémentation
- `--`: décrémentation

Affectation et conversion de pointeurs

- **Affectation**: `p = q`
- Si `p`, `q` sont de même type qualifié, fonctionne sans problème
- Si `p` est plus qualifié (`const`) que `q`, alors conversion **implicite**
- Sinon, une conversion **explicite** est requise

```
#include <stdio.h>

int main(void) {
  int x = 8, *p = &x;      // p pointe vers l'entier 8
  double *q = p;           // Avertissement: pointeurs incompatibles
  double *r = (double*)p; // Conversion explicite (à éviter)
  return 0;
}
```

Résultat:

```
cast_pointer.c: In function ‘’main:
cast_pointer.c:5:17: warning: initialization of ‘double ’* from incompatib
    5 |     double *q = p;          // Avertissement: pointeurs incompatib
      |                 ^
```

Prudence lors des conversions

Conversion de pointeur est à **éviter** sauf dans les cas suivants:

1. Vers un pointeur du **même type**, mais plus **qualifié** (`const`)

```
char *s = "linux"; // Pointeur vers une chaîne littérale
const char *t = s; // Conversion implicite (acceptable)
```

2. De/vers un pointeur `void*` (on va y revenir)
3. Vers la valeur `NULL`

```
int i = 8, *p = &i; // p pointe vers la valeur 8
p = NULL;           // p ne pointe vers aucune valeur
```

Tableaux et arithmétique des pointeurs

Adresse d'une valeur d'un tableau

On peut récupérer l'**adresse** d'une valeur dans un **tableau**

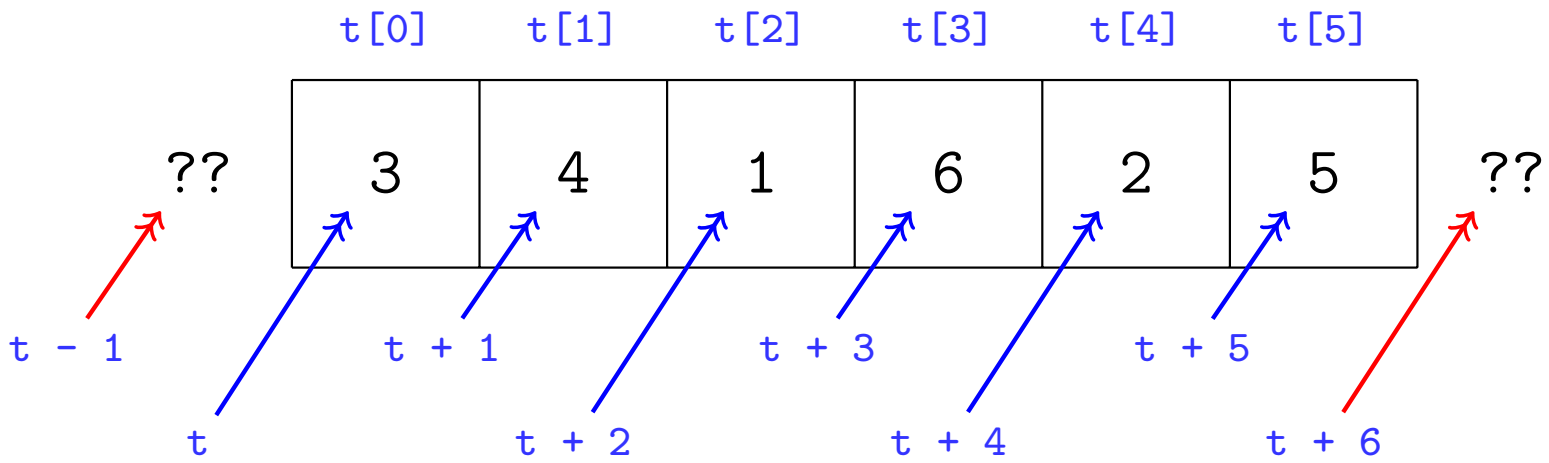
```
#include <stdio.h>

int main(void) {
  int t[4] = {1, 2, 3, 4};
  for (unsigned int i = 0; i < 4; ++i)
    printf("t[%d] = %d, &t[%d] = %p\n", i, t[i], i, &t[i]);
  return 0;
}
```

Résultat (le résultat peut varier):

```
t[0] = 1, &t[0] = 0x7ffd347db8d0
t[1] = 2, &t[1] = 0x7ffd347db8d4
t[2] = 3, &t[2] = 0x7ffd347db8d8
t[3] = 4, &t[3] = 0x7ffd347db8dc
```

Arithmétique des pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
```

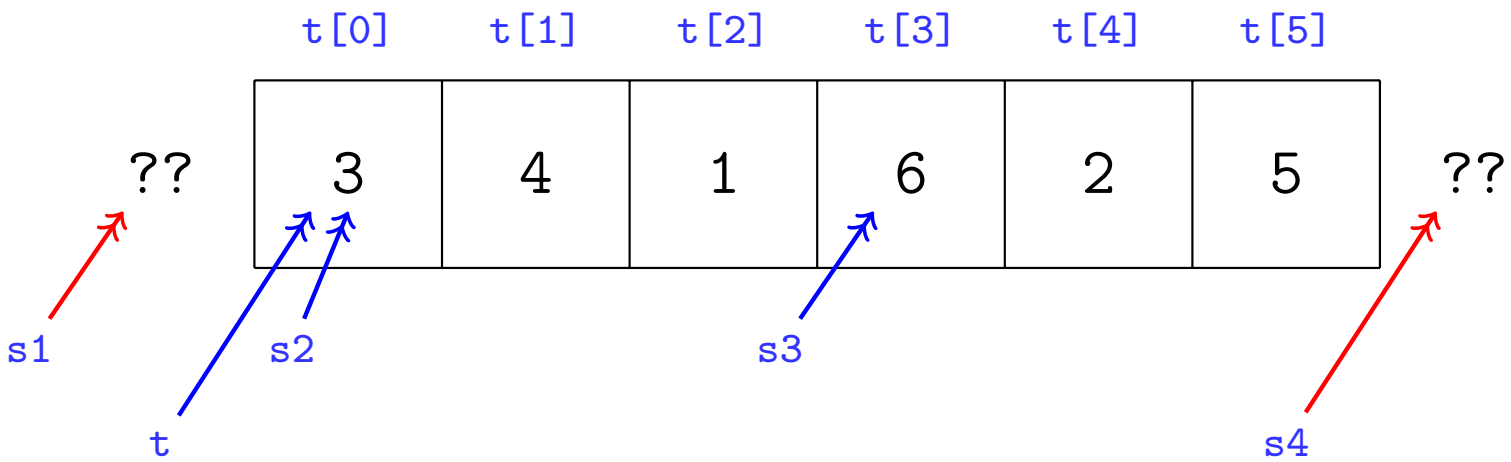


- `t`: pointe la première valeur du tableau
- `t + i`: pointe la `i`-ème valeur
- `t - 1`: pointe à « gauche » du tableau
- `t + 6`: pointe à « droite » du tableau
- Bref, `(t + i) == &t[i]`, ou `*(t + i) == t[i]`

Opérateurs de comparaison

- ==: indique si deux pointeurs contiennent la même adresse
- !=: négation de ==
- <=, >=, <, >: compare deux pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
int *s1 = t - 1, *s2 = t, *s3 = t + 3, *s4 = t + 6;
```



Les expressions `s2 == t`, `s1 != s2`, `s2 <= t`, `s3 >= s1`, `s3 < s4` et `s3 > t` sont **vraies**

Opérateurs + et - (1/2)

- Soit `p` un pointeur de type `t*`
- `p + i`: décalé de `p` de `+ sizeof(t)` octets
- `p - i`: décalé de `p` de `- sizeof(t)` octets

```
#include <stdio.h>

int main(void) {
  double values[] = {3.14, 2.71, 1.41, -0.5};
  double *p = &values[1];
  printf("values = [%lf, %lf, %lf, %lf]\n",
         values[0], values[1], values[2], values[3]);
  printf("p     = %p,  *p        = %lf\n", p, *p);
  printf("p + 1 = %p,  *(p + 1) = %lf\n", p + 1, *(p + 1));
  printf("p + 2 = %p,  *(p + 2) = %lf\n", p + 2, *(p + 2));
  printf("p - 1 = %p,  *(p - 1) = %lf\n", p - 1, *(p - 1));
  return 0;
}
```

Opérateurs + et - (2/2)

Résultat:

```
values = [3.140000, 2.710000, 1.410000, -0.500000]
p     = 0x7ffda8d29838,  *p        = 2.710000
p + 1 = 0x7ffda8d29840,  *(p + 1) = 1.410000
p + 2 = 0x7ffda8d29848,  *(p + 2) = -0.500000
p - 1 = 0x7ffda8d29830,  *(p - 1) = 3.140000
```

Opérateurs ++ et -- (1/2)

- ++: incrémenter un pointeur vers le bloc suivant
- --: décrémenter le pointeur vers le bloc précédent
- Pratique pour **itérer** sur des tableaux

```
#include <stdio.h>

int main(void) {
  char *s = "linux";
  double values[] = {3.14, 2.71, 1.41};
  for (char *p = s; // On initialise p au début de s
       *p != '\0';  // On répète tant qu'on ne rencontre pas '\0'
       ++p)         // On incrémente de sizeof(char) octet(s)
    printf("%c ", *p);
  printf("\n");
  for (// On commence à la fin du tableau
       double *p = &values[2];
       // On répète tant qu'on est dans le tableau
       p >= values;
       // On décrémente de sizeof(double) octets
       --p)
    printf("%lf ", *p);
  return 0;
}
```

Opérateurs ++ et -- (2/2)

Résultat:

```
l i n u x
1.410000 2.710000 3.140000
```

Fonctions

Fonction et variables statiques

Variables statiques

- On peut attacher des variables **statiques** à une fonction
- À l'aide du mot réservé `static`
- Mémoire réservée au **début** du programme
- Et **libéré** à la fin du programme
- **Initialisée** à `0` par défaut

Mémoïsation

- On considère n'importe quelle **fonction pure**
- On **« mémorise »** le résultat de la fonction
- Pour certains **arguments** donnés
- Lors d'un appel de fonction avec les **mêmes** arguments
- On n'a pas à **refaire** le calcul

Exemple: `fibo.c`

```c
#include <stdio.h>

unsigned long long fibonacci(unsigned int n) {
  if (n == 0 || n == 1)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
  // %lld: long long decimal
  for (unsigned int n = 0; n < 40; ++n)
    printf("%lld ", fibonacci(n));
  printf("\n");
  return 0;
}
```

Problème?

On recalcule plusieurs fois les **mêmes** valeurs

Chronomètre

```
$ gcc fibo.c -o fibo
$ time ./fibo
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m1.452s
user    0m1.451s
sys 0m0.000s
```

- **Solution 1**: « dérécursifier » le programme
- Mais pas **facile** en général
- **Solution 2**: « mémoriser » les valeurs déjà calculées
- À l'aide d'une variable **statique**

Exemple: fibofast.c

```c
#include <stdio.h>

#define MAX_N 1000

unsigned long long fibonacci(unsigned int n) {
  // Le tableau f contient les MAX_N premières valeurs
  // On l'initialise avec {1,1,0,0,0,0,...}
  static unsigned long long f[MAX_N] = {1, 1};
  if (n >= MAX_N)
    // Tableau pas assez grand, on y va naïvement
    return fibonacci(n - 1) + fibonacci(n - 2);
  if (f[n] == 0)
    // Tableau assez grand, on mémoïse
    f[n] = fibonacci(n - 1) + fibonacci(n - 2);
  return f[n];
}

int main(void) {
  for (unsigned int n = 0; n < 40; ++n)
    printf("%lld ", fibonacci(n));
  printf("\n");
  return 0;
}
```

Chronomètre

```
$ gcc fibofast.c -o fibofast
$ time ./fibofast
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m0.002s
user    0m0.002s
sys 0m0.000s
```

Fonctions comme arguments de fonctions

- Passage de **fonctions** comme **arguments** d'autres fonctions
- Supporté dans plusieurs **langages**: Java, C++, Python, Haskell, ...
- **Exemples:** les fonctions `map` et `filter`

Python:

```
>>> map(len, ["alpha", "beta", "gamma"])
[5, 4, 5]
>>> is_palindrome = lambda s: s == s[::-1]
>>> filter(is_palindrome, ["radar", "allo", "ici", "ressasser"])
['radar', 'ici', 'ressasser']
```

Haskell:

```
Prelude> map (*3) [5,4,1,2,3]
[15,12,3,6,9]
Prelude> filter (<=3) [8,1,4,3,5,6,2,7]
[1,3,2]
```

Pointeur vers une fonction

- Les pointeurs de fonctions sont **typés**
- Ils peuvent être contenus dans des **structures**, des **tableaux**, etc.
- Attention à la **syntaxe**

```
// Pointeur de fonction de type int -> int
int (*f)(int x);
// Pointeur de fonction de type (int, int) -> int
int (*g)(int x, int y);
// Fonction de int -> int*
int *f(int x);
// Fonction de (int,int) -> int*
int *g(int x, int y);
```

La fonction `map` en C (1/2)

```c
#include <stdio.h>

// Retourne le carré de `x`
int carre(int x) { return x * x; }

// Applique la fonction `f` sur le tableau `domaine` de taille `n`
// et stocke le résultat dans le tableau `image`
void map(const int *domaine,
         int *image,
         int (*f)(int),
         unsigned int n) {
  for (unsigned int i = 0; i < n; ++i)
    image[i] = f(domaine[i]);
}

int main(void) {
  int domaine[5] = {2,3,5,7,11}, image[5];
  map(domaine, image, carre, 5);
  for (unsigned int i = 0; i < 5; ++i)
    printf("carre(%d) = %d\n", domaine[i], image[i]);
  return 0;
}
```

La fonction `map` en C (2/2)

Résultat:

```
carre(2) = 4
carre(3) = 9
carre(5) = 25
carre(7) = 49
carre(11) = 121
```

Tableau de fonctions

```c
#include <stdio.h>

int carre(int x) {
  return x * x;
}

int cube(int x) {
  return x * x * x;
}

int main(void) {
  // f est un tableau contenant 2 pointeurs de fonctions
  // On accède aux fonctions avec f[ ]
  int (*f[2])(int) = {carre, cube};
  printf("%d %d %d\n", 4, f[0](4), f[1](4));
  return 0;
}
```

Résultat:

```
4 16 64
```

Tri rapide

- Dans `stdlib.h`: on trouve la fonction `qsort`:

```
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base`: pointeur vers le premier élément du tableau
- `nmemb`: nombre d'éléments dans le tableau
- `size`: taille individuelle d'un élément (utiliser `sizeof`)
- `compar` : pointeur de fonction qui compare deux éléments

Remarque

- Le type `void*` est utilisé pour une plus grande **généricité**
- En pratique, on doit faire des **conversions** (*cast*)

Utilisation de qsort

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

/* La fonction de comparaison doit avoir
 * la signature (const void *, const void *)
 */
int compare_ints(const void *a, const void *b) {
  // On doit donc faire des conversions à l'intérieur
  return *(int*)a - *(int*)b;
}

int main(void) {
  int a[] = {8,3,4,2,0,5};
  qsort(a, 6, sizeof(int), compare_ints);
  for (unsigned int i = 0; i < 6; ++i)
    printf ("%d ", a[i]);
  return 0;
}
```

Résultat:

```
0 2 3 4 5 8
```

Fouille binaire

- Toujours dans `stdlib.h`, on trouve la fonction `bsearch`:

```
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

- `key`: pointeur vers la valeur recherchée
- `base`: pointeur vers le premier élément du tableau
- `nmemb`: nombre d'éléments dans le tableau
- `size` : taille individuelle d'un élément
- `compar`: pointeur de fonction qui compare deux éléments

Utilisation de `bsearch` (1/2)

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

int compare_ints(const void *a, const void *b) {
  return *(int*)a - *(int*)b;
}

int main(void) {
  int a[] = {2,5,7,8,13,15};
  int *p;
  int key;

  for (key = 1; key <= 7; ++key) {
    p = (int*)bsearch(&key, a, 6, sizeof(int), compare_ints);
    printf("%d %sest %sdans le tableau\n",
        key, p == NULL ? "n'" : "", p == NULL ? "pas " : "");
  }
  return 0;
}
```

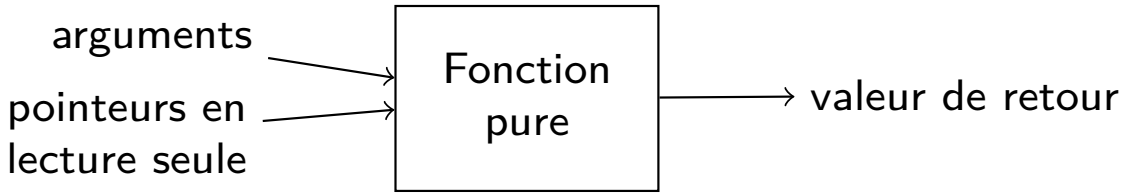
Utilisation de `bsearch` (2/2)

Résultat:

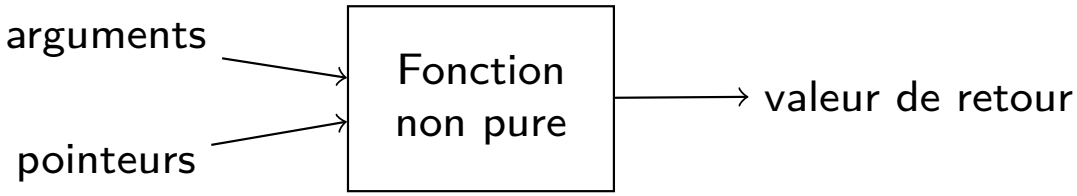
```
1 n'est pas dans le tableau
2 est dans le tableau
3 n'est pas dans le tableau
4 n'est pas dans le tableau
5 est dans le tableau
6 n'est pas dans le tableau
7 est dans le tableau
```

Entrées et sorties

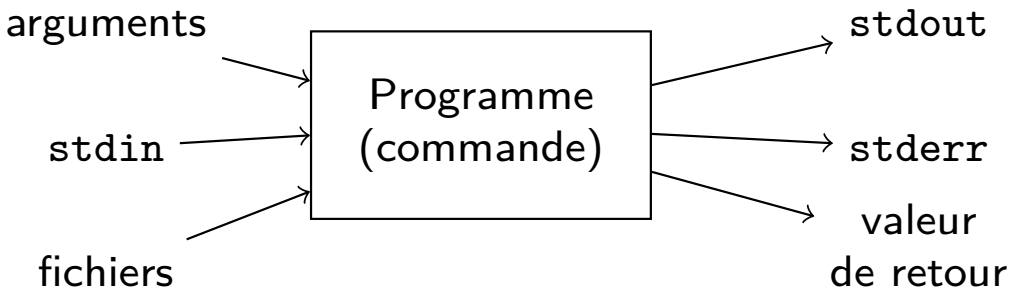
Entrées et sorties



Entrées



Sorties



La bibliothèque `stdio.h`

- *stdio* = *standard input output*
- Inclusion avec `#include <stdio.h>`

Macros:

- `EOF`: caractère de fin de fichier
- `stdin`, `stdout`, `stderr`: canaux standards
- `NULL`: pointeur nul, ...

Types:

- `FILE`: flux (*stream*)
- `size_t`: taille en octets
- `fpos_t`: position dans un flux, ...

Variables externes: `optarg`, `opterr`, `optind`, `optopt` (gestion des arguments)

Plusieurs dizaines de **fonctions** (`man stdio`)

Opérations générales (1/2)

Ouverture et fermeture:

```
// Ferme un flux
int       fclose(FILE *);
// Ouvre un flux avec un descripteur de fichier
FILE     *fdopen(int, const char *);
// Ouvre un flux
FILE     *fopen(const char *, const char *);
// Réouvre un flux
FILE     *freopen(const char *, const char *, FILE *);
```

Suppression et renommage:

```
// Supprime un fichier ou un répertoire
int       remove(const char *);
// Renomme un fichier ou un répertoire
int       rename(const char *, const char *);
```

Manipulation de fichier temporaires:

```
// Retourne un nom de fichier temporaire
char     *tmpnam(char *);
// Crée un fichier temporaire
FILE     *tmpfile(void);
```

Opérations générales (2/2)

Informations générales

```
// Vérifie si la fin du flux est atteinte
int       feof(FILE *);
// Récupère le descripteur d'un flux
int       fileno(FILE *);
// Récupère le nom du terminal courant
char     *ctermid(char *);
// Récupère l'utilisateur courant
char     *cuserid(char *);(LEGACY)
```

Gestion des erreurs:

```
// Réinitialise les indicateurs de fin de fichier et d'erreur
void      clearerr(FILE *);
// Vérifie si une erreur est survenue
int       ferror(FILE *);
// Écrit un message d'erreur sur stderr
void      perror(const char *);
```

Manipulation de caractères

```
// Lit le prochain caractère d'un flux
int       fgetc(FILE *);
// Équivalent à fgetc, avec passes multiples du flux
int       getc(FILE *);
// Écrit un caractère sur un flux
int       fputc(int, FILE *);
// Équivalent à fputc, avec passes multiples du flux
int       putc(int, FILE *);
// Remet un caractère sur un flux
int       ungetc(int, FILE *);
```

Canaux standards:

```
// Équivalent à getc(stdin)
int       getchar(void);
// Écrit un caractère sur stdout
int       putchar(int);
```

Manipulation d'octets (*binary stream*)

```
// Lit des octets provenant d'un flux
size_t   fread(void *, size_t, size_t, FILE *);
// Écrit des octets sur un flux
size_t   fwrite(const void *, size_t, size_t, FILE *);
// Lit un mot (int) provenant d'un flux
int      getw(FILE *);
// Écrit un mot (int) sur un flux
int      putw(int, FILE *);
```

Manipulation de chaînes de caractères

```
// Lit une ligne provenant d'un flux
char    *fgets(char *, int, FILE *);
// Écrit une chaîne sur un flux
int      fputs(const char *, FILE *);
```

Canaux standards:

```
// Lit une chaîne sur stdin (obsolète, deprecated)
// Préférer fgets(..., ..., stdin)
char    *gets(char *);
// Écrit une chaîne sur stdout
int      puts(const char *);
```

Lecture et écriture formatées

```
// Écrit des données formatées sur un flux
int      fprintf(FILE *, const char *, ...);
// Lit des données formatées provenant d'un flux
int      fscanf(FILE *, const char *, ...);
// Équivalent à fprintf avec va_list
int      vfprintf(FILE *, const char *, va_list);
// Équivalent fprintf, mais sur stdout
int      printf(const char *, ...);
// Équivalent à fscanf, mais provenant de stdin
int      scanf(const char *, ...);
// Écrit des données formatées dans une chaîne
int      sprintf(char *, const char *, ...);
// Équivalent à sprintf, mais avec écriture tronquée
int      snprintf(char *, size_t, const char *, ...);
// Lit des données formatées depuis une chaîne
int      sscanf(const char *, const char *, int ...);
// Équivalent à printf, mais avec a va_list
int      vprintf(const char *, va_list);
// Équivalent à snprintf, mais avec va_list
int      vsnprintf(char *, size_t, const char *, va_list);
// Équivalent à sprintf, mais avec va_list
int      vsprintf(char *, const char *, va_list);
```

Navigation

```
// Modifie la position courante dans un flux
int       fseek(FILE *, long int, int);
// Modifie la position courante dans un flux (avec off_t)
int       fseeko(FILE *, off_t, int);
// Récupère la position courante dans un flux
long int ftell(FILE *);
// Récupère la position courante dans un flux (avec off_t)
off_t     ftello(FILE *);
// Modifie la position courante dans un flux
int       fsetpos(FILE *, const fpos_t *);
// Récupère la position courante dans un flux
int       fgetpos(FILE *, fpos_t *);
// Remet la position courante d'un flux au début
void      rewind(FILE *);
```

Autres fonctions (1/2)

Traitement des arguments:

```
// Traite les options d'une commande
int       getopt(int, char * const[], const char); (LEGACY)
```

Tampons (*buffer*):

```
// Vide un tampon
int       fflush(FILE *);
// Modifie la stratégie de tampon
// Possibilités: line buffered, unbuffered, fully buffered
int       setvbuf(FILE *, char *, int, size_t);
// Modifie la stratégie de tampon à unbuffered ou fully buffered
void      setbuf(FILE *, char *);
```

Tubes (communication inter-processus):

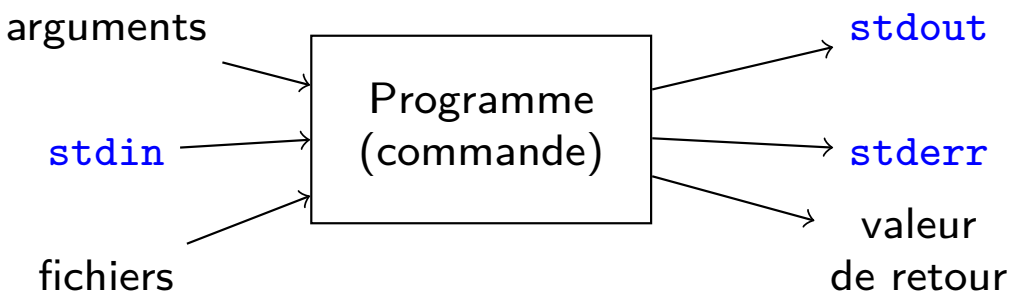
```
// Ferme un tube
int       pclose(FILE *);
// Ouvre un tube
FILE     *popen(const char *, const char *);
```

Autres fonctions (2/2)

Verrouillage:

```
// Verrouille un flux (thread-safety)
void      flockfile(FILE *);
// Version non bloquante de verouillage
int       ftrylockfile(FILE *);
// Déverrouille un flux
void      funlockfile(FILE *);
// Lit un caractère provenant d'un flux (non sécuritaire)
int       getc_unlocked(FILE *);
// Écrit un caractère (non sécuritaire)
int       putc_unlocked(int, FILE *);
// Équivalent à getc(stdin) (non sécuritaire)
int       getchar_unlocked(void);
// Écrit un caractère sur stdout (non sécuritaire)
int       putchar_unlocked(int);
```

Canaux



3 canaux standards

- `stdin`: entrée standard (canal 0)
- `stdout`: sortie standard (canal 1)
- `stderr`: sortie d'erreur standard (canal 2)

Comportement par défaut (peut être redéfini)

- `stdin`: saisie clavier (*line buffered*)
- `stdout`: affichage sur le terminal (*line buffered*)
- `stderr`: affichage sur le terminal (*unbuffered*)

L'entrée standard (stdin)

- Un programme peut lire des données sur l'**entrée standard**
- Par défaut, lit la **saisie clavier**

```c
#include <stdio.h>

#define BUFFER_SIZE 20

int main(void) {
  char c1 = getchar(), c2 = getchar(), c3 = getchar();
  char line[BUFFER_SIZE];
  fgets(line, BUFFER_SIZE, stdin);
  printf("3 premiers caractères: %c %c %c\n", c1, c2, c3);
  printf("Reste de ligne: %s\n", line);
  return 0;
}
```

La sortie standard (stdout)

- Un programme peut écrire des données sur la **sortie standard**
- Par défaut, l'affichage se fait sur le **terminal**

```c
#include <stdio.h>

int main(void) {
  char c = 'A';
  putchar(c);
  putchar('\n');
  puts("C est un langage particulier");
  fwrite("Incroyable!", sizeof(char), 4, stdout);
  printf("\n%s %c %p %lf\n",
      "Fantastique", '!', &c, 1.23456789);
  return 0;
}
```

Résultat:

```
A
C est un langage particulier
Incr
Fantastique ! 0x7ffc9bcf5b47 1.234568
```

La sortie d'erreur standard (stderr)

- On peut aussi écrire sur la **sortie d'erreur standard**
- Par défaut, l'affichage se fait aussi sur le **terminal**

```c
#include <stdio.h>

int main(void) {
  char c = 'A';
  fputc(c, stderr);
  fputc('\n', stderr);
  fputs("C est un langage particulier", stderr);
  fwrite("Incroyable!", sizeof(char), 4, stderr);
  fprintf(stderr, "\n%s %c %p %lf\n",
      "Fantastique", '!', &c, 1.23456789);
  return 0;
}
```

Résultat:

```
A
C est un langage particulierIncr
Fantastique ! 0x7ffdcb2fa557 1.234568
```

Valeur de retour

- En Unix, tout programme retourne une **valeur entière**
- Lorsque son exécution est **terminée**

Sémantique

- `0`: le programme s'est terminé **« normalement »**
- $\neq$ `0`: le programme s'est terminé **« anormalement »**

Récupérer la valeur de retour

- Contenue dans la **variable spéciale** `$?`
- Valeur de retour de la **dernière commande**

Valeur de retour en C

Fonction `main`

- Valeur retournée à l'aide de `return`
- Doit être **entière**
- Peut être **négative**
- Par défaut, retourne `0`
- **Bonne pratique**: toujours spécifier la valeur de retour

La fonction `exit`

```
void exit(int status);
```

- Permet de **terminer** l'exécution du programme proprement
- Vide et ferme les **flux** encore ouverts
- Supprime les **fichiers temporaires**