Programmation Procédurale – Pointeurs

Polytech'Nice Sophia Antipolis

Erick Gallesio

2015 – 2016

Motivations

- Un pointeur contient l'adresse d'un objet
- l'objet lui même peut être accédé de façon *indirecte*
- possibilité de dsigner une zone de mémoire allouée dynamiquement

Les pointeurs sont beaucoup utilisés en C. Il permettent:

- de passer des objets par référence
- l'écriture de code plus compact
- l'écriture de code plus efficace
- mais aussi, l'écriture de code moins lisible :-(

Déclaration de pointeurs

Exemples:

```
int *p1, *p2;                /* pointeurs sur entiers */
int *p1, p2;                 /* Attention: un pointeur, un int */

struct {int x, y;} *ps;      /* pointeur sur structure */
void *r;                     /* pointeur sur void: adr. brute */
int *s[10];                  /* tableau de 10 pointeurs sur int */
int (*s)[10];                /* pointeur sur tableau de 10 int  */
```

Un pointeur peut désigner n'importe quelle variable (statique, dynamique, constante). Il peut aussi dénoter l'adresse d'une fonction

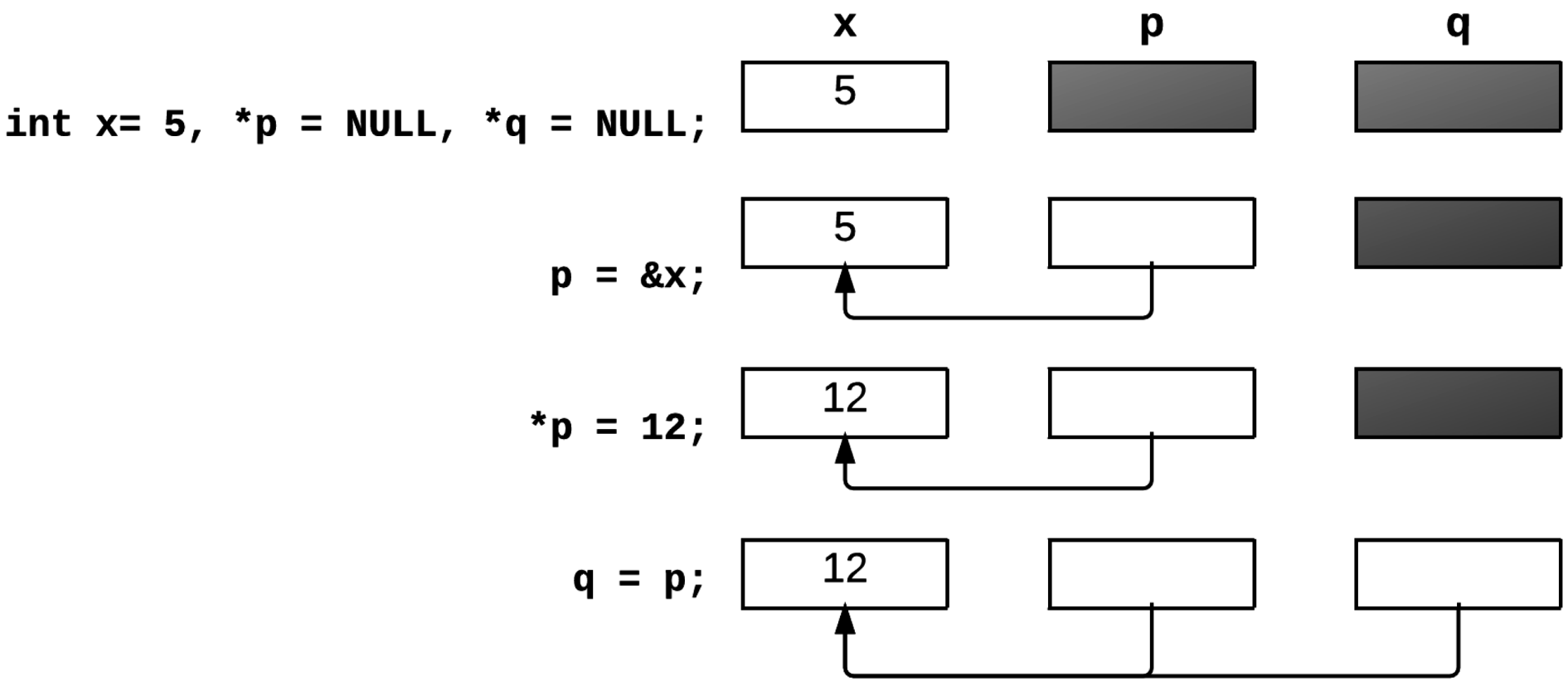
```
int (*pfunc)(void);          /* pointeur sur une fct sans paramètre
                                qui renvoie un entier */
int (*T[5])(void);           /* tableau de 5 pointeurs de ce type */
```

Opérations sur les pointeurs (1 / 2)

Deux opérations seulement sur les pointeurs.

- `*p` permet l'indirection (déréférence)
- `&v` renvoie l'adresse de la variable `v` (réference)

Exemple



Opérations sur les pointeurs (2 / 2)

Cas particulier des pointeurs sur structure

```
struct {
  int a, b;
} x, *p = &x;
```

- `x` est une structure
- `*p` désigne cette structure
- l'accès au champ a de `x` : `x.a`
- l'accès au même champ avec `p` : `(*p).a`

Notation spéciale:

Pour simplifier, `(*p).a` peut aussi s'écrire `p->a`

Paramètres de type pointeur (1 / 2)

En C, le passage de paramètres se fait *par valeur*

```
    void swap(int a, int b) {
      int aux;

      aux = a;
      a = b;
      b = aux;
      printf("swap: a=%d b=%d\n", a, b);
    }

    int main(void) {
      int x = 1, y = 2;

      printf("main avant: x=%d y=%d\n", x, y);
      swap(x, y);
      printf("main après: x=%d y=%d\n", x, y);
      return 0;
    }
==>
main avant: x=1 y=2
swap: a=2 b=1
main après: x=1 y=2
```

Paramètres de type pointeur (2 / 2)

Les pointeurs permettent de simuler le passage *par référence*

```
    void swap(int *a, int *b) {
      int aux;

      aux = *a;
      *a = *b;
      *b = aux;
      printf("swap: *a=%d *b=%d\n", *a, *b);
    }

    int main(void) {
      int x = 1, y = 2;

      printf("main avant: x=%d y=%d\n", x, y);
      swap(&x, &y);
      printf("main après: x=%d y=%d\n", x, y);
      return 0;
    }
==>
main avant: x=1 y=2
swap: *a=2 *b=1
main après: x=2 y=1
```

Opérations arithmétiques sur les pointeurs

- Comparaison
  - == et !=
  - mais aussi <, >=, > et >=
- Addition / soustraction d'un entier
  - pointeur + entier $\longrightarrow$ pointeur
  - pointeur - entier $\longrightarrow$ pointeur
  - permet de calculer un décalage d'adresses
- Différence entre deux pointeurs de même type
  - pointeur - pointeur $\longrightarrow$ entier
  - permet de calculer le nombre d'éléments de ce type entre les deux adresses

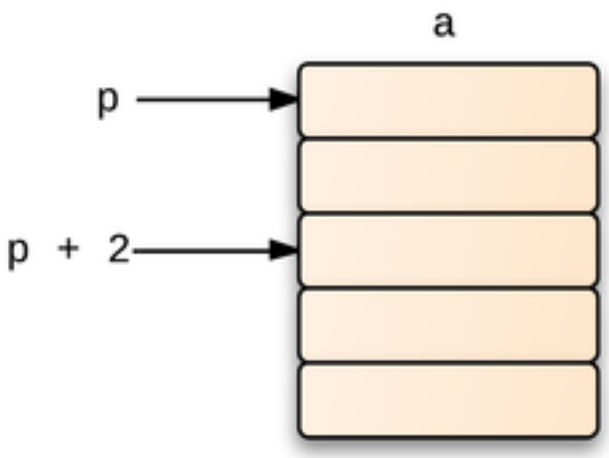
Conversions:

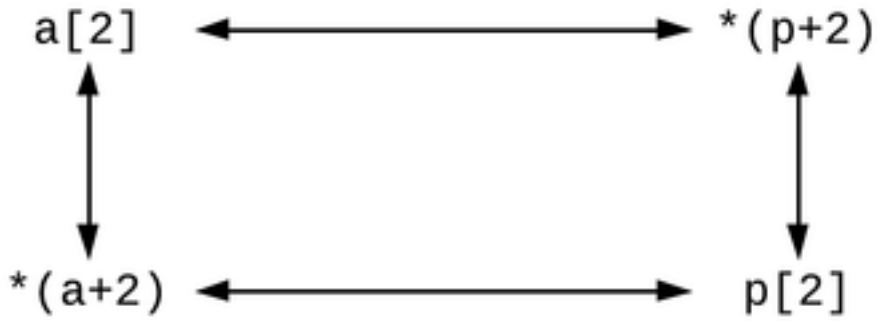
- conversion d'un `void *` vers (ou depuis) un pointeur quelconque: toujours OK
- les autres conversion nécessitent un *cast*

Pointeurs et tableaux

- En C, **pointeurs** et **tableaux** sont deux concepts proches.
- En fait, le nom d'un tableau correspond à l'adresse de la première case du tableau
- L'arithmétique de pointeurs permet d'indexer un tableau avec des pointeurs

```
int a[5];
int *p = &a[0];
```





Règle fondamentale

```
a[i]   <=>   *(a+i)
```

Pointeurs sur fonction (1 / 2)

```
int plus(int op1, int op2)        {return op1 + op2;}
int minus(int op1, int op2)       {return op1 - op2;}
...

int (*operation)(int, int);  /* pointeur sur fonction */
int a, b, res;

a = read_operand();
switch (getchar()) {
    case '+': operation = plus; break;
    case '-': operation = minus; break;
    ...
}
b = read_operand();
res = (*operation)(a, b);    /* En C ANSI: operation(a, b) */
```

Pointeurs sur fonction (2 / 2)

```
struct Func {
    char nom[10];
    double (*f)(double);
};

struct Func T[] = {
    {"sinus", sin},
    {"cosinus", cos},
    ...
};

double (*fct) (double) = chercher(T, "cosinus");
printf("cosinus(42) = %lf\n", fct(42));
```

Allocation mémoire dynamique

Fonctions standard:

- Allocation mémoire

```
void *malloc(size_t size);
void *calloc(size_t nelems, size_t size);
void *realloc(void *ptr, size_t new_size);
```

Ces fonctions renvoient un pointeur sur la zone allouée ou **NULL** si l'allocation est impossible

- Libération mémoire

```
void free(void *ptr);
```

Le pointeur `ptr` doit avoir été obtenu par une des 3 fonctions précédentes

Allocation dynamique: Liste chaînée (1 / 2)

```
typedef struct { /* Les infos que l'on met dans la liste */
   char name[30];
   ...
} Info;

 struct elem {  /* maillon de la liste chaînée */
   Info i;
   struct elem *next;
 };

 /* Allouer un nouvel élement */
 struct elem *NewInfo(Info inf) {
   struct elem *p = (struct elem *) malloc(sizeof(Elem));
   if (p == NULL) {
     fprintf(stderr, "allocation error\n");
     exit(1);
   }
   p->i = inf;
   p->next = NULL;
   return p;
 }
```

Allocation dynamique: Liste chaînée (2 / 2)

```c
/* Insertion d'un nouvel élément devant la liste L */
struct elem *new = NewInfo(les_infos);

new->next = L;
L = new;

/* Recherche dans la liste chaînée */
struct elem *SearchInfo(struct elem *list, char who[])
{
  struct elem *p;

  for (p = list; p != NULL; p = p->next)
    if (strcmp(p->i.name, who) == 0)
      return p;

  return NULL;  /* not found */
}
```

Allocation dynamique: un allocateur basique

```c
#define MAXBUF 1000
char allocbuf[MAXBUF];      /* le buffer */
char *allocptr = allocbuf;  /* première position libre */

char *alloc(int n)
{
    if (allocptr + n <= allocbuf + MAXBUF) {    /* ça rentre */
        char *prev = allocptr;

        allocptr += n;
        return prev;
    }
    else return NULL;
}

void release(char *p)
{
    if (allocbuf <= p && p < &allocbuf[MAXBUF])
        allocptr = p;
}
```

Pointeurs et chaînes de caractères (1 / 2)

```c
/* dernière de la fonction strcat */
void strcat(char s1[], char s2[])
{
  int i=0, j=0;

  while (s1[i]) i += 1;
  while (s1[i++] = s2[j++]) /* Nothing */;
}

/* version finale de la fonction strcat */
void strcat(char *s1, char *s2)
{
  while (*s1) s1++;
  while (*s1++ = *s2++) /* Nothing */;
}

/* Version "compacte" de la comparaison de chaînes */
int strcmp(char *s1, char *s2)
{
  for ( ; *s1 == *s2; s1++, s2++)
    if (*s1 == '\0') return 0;
  return *s1 - *s2;
}
```

Pointeurs et chaînes de caractères (2 / 2)

Pour mémoire:

```c
/* Version originale de la fonction strcat */
void strcat(char s1[], char s2[]) /* le strcat standard n'est pas void... */
{
  int i=0, j=0;

  while (s1[i] != '\0') i += 1;     /* parcours de s1 */
  while (s2[j] != '\0') {           /* parcours de s2 + copie */
    s1[i] = s2[j];
    i += 1; j += 1;
  }

  /* Ne pas oublier le caractère nul final */
  s1[i] = '\0';
}

/* version finale de la fonction strcat */
void strcat(char *s1, char *s2)
{
  while (*s1) s1++;
  while (*s1++ = *s2++) /* Nothing */;
}
```

Tableaux de chaînes de caractères (1 / 3)

Souvent utilisés car

- il permettent d'avoir des lignes de longueur variables
- sont plus efficaces que des tableaux bidimensionnels

Exemple:

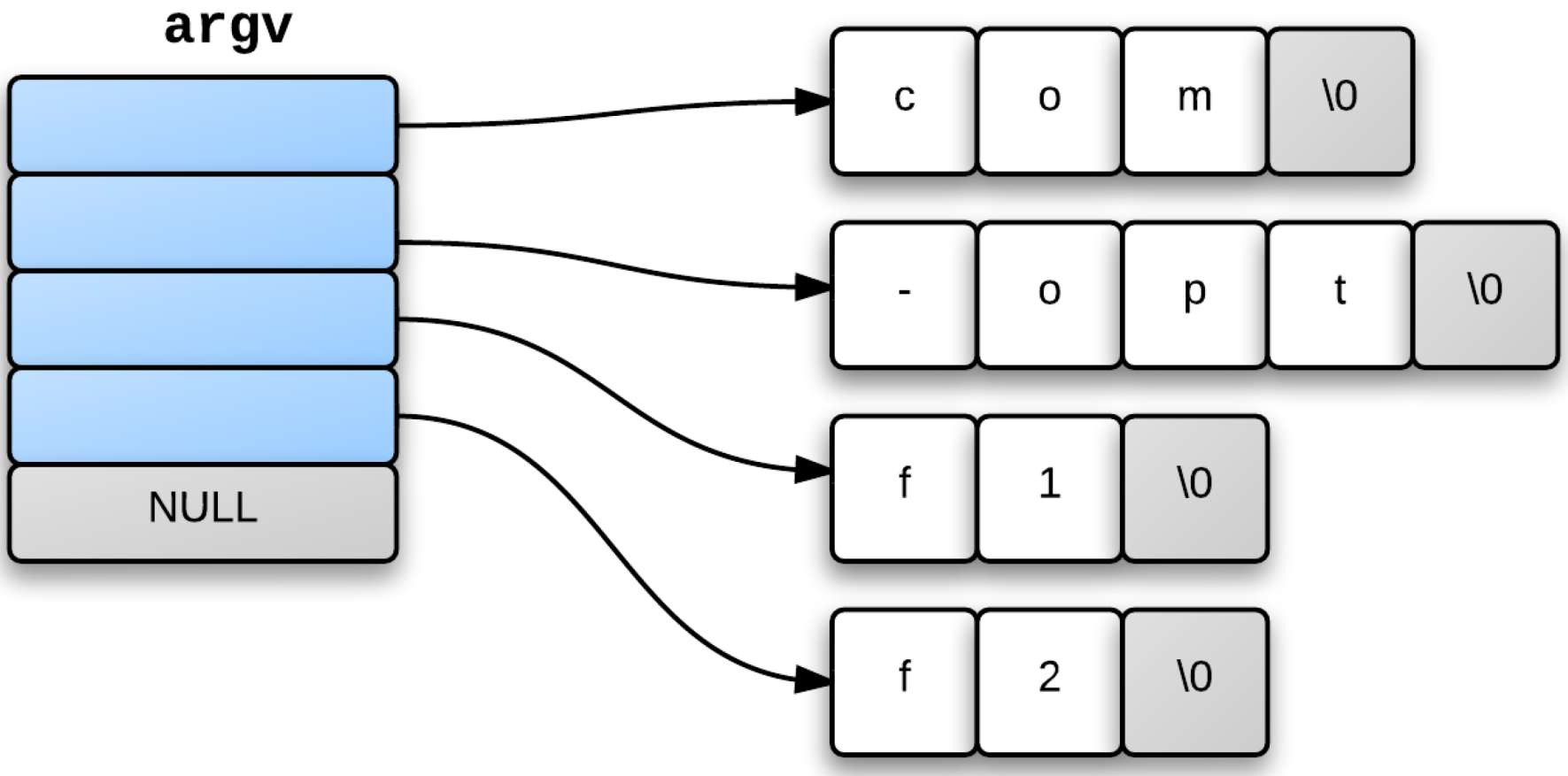
```
char *MonthNames[] = {
    "unknown",
    "January",
    "February",
    ...
    "December"
};
```

Tableaux de chaînes de caractères (2 / 3)

Paramètres de la ligne de commande : `argc` et `argv`

```
$ com -opt f1 f2
```

```
argc = 4
```



.

Tableaux de chaînes de caractères (3 / 3)

Exemple d'utilisation de argc et argv: commande echo

```
void main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], i < argc-1 ? ' ' : '\n');
}
```

Autre écriture

```
void main(int argc, char **argv)
{
    while (--argc)
        printf("%s%c", *++argv, argc > 1 ? ' ' : '\n');
}
```

Quelques erreurs classiques en C : Top 12 (1 / 5)

```c
void f1(void)
{
  char *x = malloc(50);     // pas une erreur, mais préferer un tableau local
                            // si x ne sert que dans f1
  ....;
}

void f2(void)
{
  char *s = malloc(10);      // malloc inutile => fuite de mémoire

  s = "abcd";
}


int *f3(void)
{
  int s[10] = { 1, 2, 3};   // return d'un tableau local

  ...;
  return s;
}
```

Quelques erreurs classiques en C : Top 12 (2 / 5)

```c
void f4(void)
{
  char *s;            // non allocation d'un buffer

  gets(s);            // et pourtant man gets(3):  char *gets(char *s);
  ...;
}


void f5(int a, int b)
{
  char *p = malloc(10);

  if (a < b) {                      // fuite mémoire ==> préférer l'allocation statique
    fprintf(stderr, "Erreur!\n");
    return;
  }
  ...;
}
```

Quelques erreurs classiques en C : Top 12 (3 / 5)

```c
char *f6(char *s)
{
  char *copy=malloc(sizeof(s));   // sizeof au lieu de strlen
  ......;
}


char *f7(char *s)
{
  char *copy=malloc(strlen(s));   //  manque +1
  ......;
}


void f8(void)
{
  char *p = "Hello";              // Modification d'une constante (-fwritable-strings)

  p[1] = 'Z';
  ...
}
```

Quelques erreurs classiques en C : Top 12 (4 / 5)

```c
void f9(void)
{
  char s1[]   = "abc";
  char s2[30] = "abc";
  char s3[]   = "abracadabra";

  strcpy(s1, s3);       // Erreur
  strcpy(s2, s3);       // OK
}


void f10(void)
{
  char p[] = "Hello";
  char *q  = "World";

  q = p;                // OK
  p = q;                // Erreur
  q++;                  // OK
  p++;                  // Erreur
}
```

Quelques erreurs classiques en C : Top 12 (5 / 5)

```c
void f11(int n)
{
  char *p = malloc(n * sizeof(int));

  p[0] = 100;
  if (a < b)
      free(p);
  else
      p[1] = 300;
  ...;
  p[2] = 500;      // Ouch si a < b: utilisation d'une zone déallouée
}

void f12(int n)
{
    char *p = malloc(n * sizeof(int));

    if (a < b)
        free(p);
    else
        g(p)
    free(p);        // Double libération de la même zone
}
```