

INF3135

Construction et maintenance de logiciels

Chapitre 3: Pointeurs

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

- 1 Adresse et pointeur
- 2 Opérations sur les pointeurs
- 3 Tableaux et arithmétique des pointeurs
- 4 Chaînes de caractères
- 5 Pointeurs de fonctions

Adresse et pointeur

Adresse

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

```
#include <stdio.h>

int main(void) {
    int x = 210;
    printf("x           = %d\n", x);
    printf("&x         = %p\n", &x);
    printf("sizeof(x)    = %ld (= sizeof(int))\n", sizeof(x));
    printf("sizeof(&x) = %ld (4 ou 8, selon l'architecture)\n",
           sizeof(&x));
    return 0;
}
```

Résultat (le résultat peut varier):

```
x           = 210
&x          = 0x7ffef8a2c3e4
sizeof(x)    = 4 (= sizeof(int))
sizeof(&x)   = 8 (4 ou 8, selon l'architecture)
```

Adresses dans une structure

```
#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;
}
```

Résultat (le résultat peut varier):

p.fname	= Novak	&p.fname	= 0x7fff0eb44280
p.lname	= Djokovic	&p.lname	= 0x7fff0eb44294
p.active	= 1	&p.active	= 0x7fff0eb442a8
p.rank	= 1	&p.rank	= 0x7fff0eb442ac
p.winrate	= 0.950000	&p.winrate	= 0x7fff0eb442b0

Pointeur

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

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

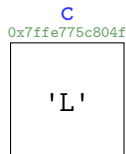
```
int main(void) {  
    char c = 'L', *p = &c;  
    printf("c          = %c\n", c);  
    printf("&c        = %p\n", &c);  
    printf("p          = %p\n", p);  
    printf("&p        = %p\n", &p);  
    printf("sizeof(c) = %ld\n", sizeof c);  
    printf("sizeof(p) = %ld\n", sizeof p);  
    return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

Résultat (le résultat peut varier):

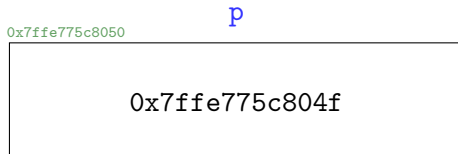
```
c          = L  
&c        = 0x7ffe775c804f  
p          = 0x7ffe775c804f  
&p        = 0x7ffe775c8050  
sizeof(c) = 1  
sizeof(p) = 8
```

Représentation schématique

```
c          = L
&c         = 0x7ffe775c804f
p          = 0x7ffe775c804f
&p         = 0x7ffe775c8050
sizeof(c)  = 1
sizeof(p)  = 8
```



1o

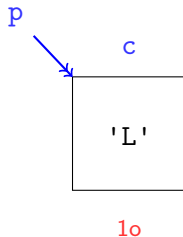


8o

Avec variables

Représentation schématique

```
c          = L
&c         = 0x7ffe775c804f
p          = 0x7ffe775c804f
&p         = 0x7ffe775c8050
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
- **Syntaxe:** coller `*` sur la variable et non le type:

```
int *p;           // Syntaxe préférée
int* p;           // Non recommandé
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

```
1  #include <stdio.h>
2
3  int main(void) {
4      int i;
5      const int *p = &i; // Le pointeur p ne doit pas modifier
6                          // le contenu qu'il référence
7      *p = 13;           // On tente de modifier le contenu de i
8      int *const q;      // Le pointeur q ne doit pas être modifié
9      q = &i;            // On tente de le modifier
10     return 0;
11 }
```

Résultat:

pointeur_const.c: In function 'main':

pointeur_const.c:7:8: error: assignment of read-only location '*p'

```
    *p = 13;           // On tente de modifier le contenu de i
    ^
```

pointeur_const.c:9:7: error: assignment of read-only variable 'q'

```
    q = &i;            // On tente de le modifier
    ^
```

Opérations sur les pointeurs

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     = 0x7ffcbfa1e798  
p      = 0x7ffcbfa1e798  
*p     = 1.500000  
*(&d)  = 1.500000  
&(*p) = 0x7ffcbfa1e798
```

Opérateur -> (sucre syntaxique)

- 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;
}
```

Résultat (le résultat peut varier):

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

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 from incompatible
    pointer type [-Wincompatible-pointer-types]
        double *q = p;          // Avertissement: pointeurs incompatibles
```

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>

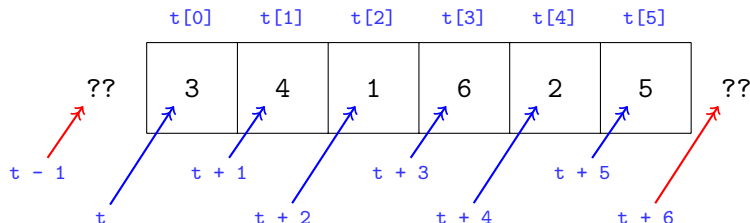
int main(void) {
    int t[4] = {1, 2, 3, 4};
    for (unsigned int i = 0; i < 4; ++i)
        printf("t[%d] = %d, &t[%d] = %p\n",
               i, t[i], i, &t[i]);
    return 0;
}
```

Résultat (le résultat peut varier):

```
t[0] = 1, &t[0] = 0x7ffebfb89390
t[1] = 2, &t[1] = 0x7ffebfb89394
t[2] = 3, &t[2] = 0x7ffebfb89398
t[3] = 4, &t[3] = 0x7ffebfb8939c
```

Arithmétique des pointeurs

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

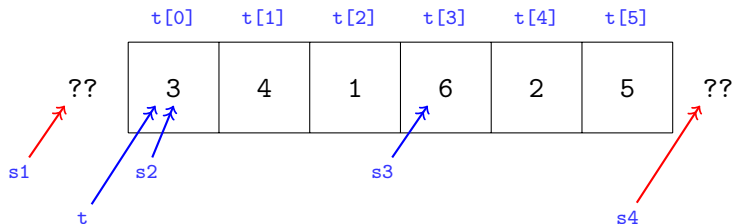


- `t`: pointe la première valeur du tableau
- `t + i`: pointe la *i*-ème valeur
- `t - 1`: pointe à « gauche » du tableau
- `t + 6`: pointe à « droite » du tableau
- Bref, `(t + i) == &t[i]`, ou `*(t + i) == t[i]`

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 -

- 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;
}
```

Résultat:

```
values = [3.140000, 2.710000, 1.410000, -0.500000]
p      = 0x7ffea139f2c8, *p      = 2.710000
p + 1 = 0x7ffea139f2d0, *(p + 1) = 1.410000
p + 2 = 0x7ffea139f2d8, *(p + 2) = -0.500000
p - 1 = 0x7ffea139f2c0, *(p - 1) = 3.140000
```

Opérateurs ++ et --

- ++: 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 (double *p = &values[2]; // On initialise à la fin de values
        p >= values; // On répète tant qu'on n'est pas au début
        --p) // On décrémente de sizeof(double) octets
        printf("%lf ", *p);
    return 0;
}
```

Résultat:

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


Chaînes de caractères

Chaînes de caractères en C

- Cas **particulier** de tableau
- Ses éléments sont de type `char`
- Chaînes **littérales** délimitées par des guillemets " "
- Chaîne **bien formée**: doit terminer par le caractère `\0`
- Plusieurs **types** peuvent désigner une chaîne:

```
char s1[10];    // Tableau de caractères de taille fixe
char *s2;       // Pointeur vers début d'une chaîne
const char *s2; // Chaîne en lecture seule
```

Bibliothèques utiles

- `ctype.h`: manipulation de caractères
- `string.h`: manipulation de chaînes

La bibliothèque ctype.h

- `int isalpha(c)`: retourne une valeur $\neq 0$ ssi `c` est **alphabétique**
- `int isupper(c)`: retourne une valeur $\neq 0$ ssi `c` est **majuscule**
- `int islower(c)`: retourne une valeur $\neq 0$ ssi `c` est **minuscule**
- `int isdigit(c)`: retourne une valeur $\neq 0$ ssi `c` est un **chiffre**
- `int isalnum(c)`: retourne `isalpha(c) || isdigit(c)`
- `int isspace(c)`: retourne une valeur $\neq 0$ ssi `c` est un **espace**, un **saut de ligne**, un caractère de **tabulation**, etc.
- `int isprint(c)`: retourne une valeur $\neq 0$ ssi `c` est **affichable**
- `char toupper(c)`: retourne la lettre **majuscule** correspondant à `c`
- `char tolower(c)`: retourne la lettre **minuscule** correspondant à `c`

Remarque

Les fonctions `toupper` et `tolower` sont définies sur les **caractères** et non sur les **chaînes**

La bibliothèque `string.h`

Plusieurs **fonctions** disponibles:

- `strcat`: concatène une chaîne à la suite d'une autre
- `strncat`: concatène une chaîne à une autre en tronquant
- `strcpy`: copie une chaîne dans une autre
- `strncpy`: copie une chaîne dans une autre en tronquant
- `strlen`: longueur d'une chaîne
- `strcmp`: compare deux chaînes
- `strncmp`: compare deux chaînes en tronquant
- `strchr`: cherche un caractère dans une chaîne de gauche à droite
- `strrchr`: cherche un caractère dans une chaîne de droite à gauche
- `strstr`: cherche une chaîne dans une autre de gauche à droite
- `strrstr`: cherche une chaîne dans une autre de droite à gauche
- `strtok`: segmente une chaîne en morceaux (*tokens*)

Les fonctions strcat et strncat (1/3)

```
// Concatène la chaîne `src` à la suite de la chaîne `dest`  
char *strcat(char *dest, const char *src);
```

- Commence à écrire sur le '\0' à la fin de dest
- Puis insère un caractère '\0' à la toute fin
- **Dangereuse**: attaque par dépassement de tampon (*buffer overrun*)

```
// Concatène au plus `n` caractères de la chaîne `src` à la suite  
de la chaîne `dest`  
char *strncat(char *dest, const char *src, size_t n);
```

- Même idée que strcat
- Mais utilise au plus les *n* premiers caractères de src
- Puis insère un caractère '\0' à la toute fin
- **Sécuritaire**: si *n* est choisie correctement

Les fonctions strcat et strncat (2/3)

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\\0') printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else printf(" ? ");
    printf("\n");
}

int main(void) {
    // strcat: résultat pas toujours bien formé
    char s[10] = "Linux "; printf("s[10] avant\n"); print_string(s, 10);
    printf("s[10] après\n"); strcat(s, "Mint"); print_string(s, 10);
    // strncat: sécuritaire
    char t[10] = "Linux "; printf("t[10] avant\n"); print_string(t, 10);
    unsigned int m = 10 - strlen(t) - 1;
    printf("t[10] après\n"); strncat(t, "Mint", m); print_string(t, 10);
    return 0;
}
```

Les fonctions strcat et strncat (3/3)

Résultat:

```
s[10] avant
 0  1  2  3  4  5  6  7  8  9
L  i  n  u  x      \0 \0 \0 \0
s[10] après
 0  1  2  3  4  5  6  7  8  9
L  i  n  u  x      M  i  n  t
t[10] avant
 0  1  2  3  4  5  6  7  8  9
L  i  n  u  x      \0 \0 \0 \0
t[10] après
 0  1  2  3  4  5  6  7  8  9
L  i  n  u  x      M  i  n  \0
```

- La chaîne s est **mal formée**
- Et on a écrit à un endroit **non réservé**
- La chaîne t est **bien formée**
- Car on a recopié seulement les $n - \text{strlen}(t) - 1$ premiers caractères de "Mint", où n est la capacité de t

Les fonctions strcpy et strncpy (1/3)

```
// Copie la chaîne `src` dans la chaîne `dest`  
char *strcpy(char *dest, const char *src);
```

- Puis insère un caractère `'\0'` à la toute fin
- **Dangereuse**: comme strcat, dépassement de tampon possible

```
// Copie au plus `n` caractères de la chaîne `src` dans la chaîne `dest`  
char *strncpy(char *dest, const char *src, size_t n);
```

- Copie au plus les `n` premiers caractères de `src`
- Puis insère un caractère `'\0'` à la toute fin
- **Sécuritaire**: si la capacité de `dest` est au moins `n + 1`

Les fonctions strcpy et strncpy (2/3)

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\\0')        printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else                    printf(" ? ");
    printf("\n");
}

int main(void) {
    // strcpy: résultat pas toujours bien formé
    char s[6]; printf("s[6] avant\n"); print_string(s, 6);
    printf("s[6] après\n"); strcpy(s, "CentOS"); print_string(s, 6);
    // strncpy: plus sécuritaire
    char t[6]; printf("t[6] avant\n"); print_string(t, 6);
    printf("t[6] après\n"); strncpy(t, "CentOS", 5); print_string(t, 6);
}
```

Les fonctions strcpy et strncpy (3/3)

Résultat:

```
s[6] avant
 0  1  2  3  4  5
 ?  U  \0 \0  ?  ?

s[6] après
 0  1  2  3  4  5
 C  e  n  t  0  S

t[6] avant
 0  1  2  3  4  5
 \0 ?  ?  ? \0 \0

t[6] après
 0  1  2  3  4  5
 C  e  n  t  0  \0
```

- La chaîne s est **mal formée**
- Et on a écrit à un endroit **non réservé**
- La chaîne t est **bien formée**
- Car on a recopié seulement les $n - 1$ premiers caractères de "Mint", où n est la capacité de t

La fonction strlen

```
// Retourne la longueur de la chaîne `s`  
size_t strlen(const char *s);
```

– **Remarque:** parcourt toute la chaîne (complexité linéaire)

```
#include <stdio.h>  
#include <string.h>  
  
int main(void) {  
    char s[40] = "Cette chaine est bien formee";  
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));  
    strcpy(s, "Celle-ci aussi");  
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));  
    strcat(s, ", meme plus longue");  
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));  
    return 0;  
}
```

Résultat:

s = Cette chaine est bien formee	strlen(s) = 28
s = Celle-ci aussi	strlen(s) = 14
s = Celle-ci aussi, meme plus longue	strlen(s) = 32

Les fonctions strcmp et strncmp (1/3)

```
// Compare les chaînes s1 et s2
int strcmp(const char *s1, const char *s2);
```

- Implémente l'ordre **lexicographique** (dictionnaire)
- Induit par l'**alphabet ASCII**
- C'est une relation d'ordre **total**
- La valeur **retournée** est

$$\begin{cases} 0 & \text{si les chaînes s1 et s2 sont égales} \\ <0 & \text{si s1 précède s2 dans le dictionnaire} \\ >0 & \text{si s1 succède s2 dans le dictionnaire} \end{cases}$$

- Aucun problème si chaînes **bien formées**

```
// Compare au plus les `n` premiers caractères de s1 et s2
int strncmp(const char *s1, const char *s2, size_t n);
```

Les fonctions strcmp et strncmp (2/3)

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

int main(void) {
    char *words[] = {"Ubuntu", "4+5=0", "uname", "lolcat"};
    for (unsigned int i = 0; i < 4; ++i) {
        for (unsigned int j = i; j < 4; ++j) {
            int c = strcmp(words[i], words[j]);
            printf("strcmp(%-6s, %-6s) = %-3d ==> ",
                words[i], words[j], c);
            printf("%-6s %s %-6s\n", words[i],
                c == 0 ? "==" : (c < 0 ? "<" : ">"),
                words[j]);
        }
    }
    return 0;
}
```

Résultat:

```
strcmp(Ubuntu, Ubuntu) = 0      ==> Ubuntu == Ubuntu
strcmp(Ubuntu, 4+5=0 ) = 33     ==> Ubuntu > 4+5=0
strcmp(Ubuntu, uname ) = -32    ==> Ubuntu < uname
strcmp(Ubuntu, lolcat) = -23     ==> Ubuntu < lolcat
strcmp(4+5=0 , 4+5=0 ) = 0       ==> 4+5=0 == 4+5=0
strcmp(4+5=0 , uname ) = -65     ==> 4+5=0 < uname
strcmp(4+5=0 , lolcat) = -56     ==> 4+5=0 < lolcat
strcmp(uname , uname ) = 0       ==> uname == uname
strcmp(uname , lolcat) = 9       ==> uname > lolcat
strcmp(lolcat, lolcat) = 0       ==> lolcat == lolcat
```

Les fonctions strcmp et strncmp (3/3)

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

bool is_square(const char *s) {
    size_t n = strlen(s);
    if (n % 2 != 0) return false;
    size_t h = n / 2;
    return strncmp(s, s + h, h) == 0;
}

int main(void) {
    char *strings[] = {"abab", "123123", "1122", "aaabaaa", "aaabaaac"};
    for (unsigned int i = 0; i < 5; ++i) {
        unsigned int freq[10];
        printf("Is \"%s\" a square? %s\n",
              strings[i], is_square(strings[i]) ? "yes" : "no");
    }
    return 0;
}
```

Résultat:

```
Is "abab" a square? yes
Is "123123" a square? yes
Is "1122" a square? no
Is "aaabaaa" a square? no
Is "aaabaaac" a square? no
```

La fonction strchr

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

int main(void) {
    const char *p = "abracadabra";
    while (true) {
        p = strchr(p, 'a');
        if (p == NULL) break;
        printf("%s\n", p);
        ++p;
    }
    return 0;
}
```

Résultat:

```
abracadabra
acadabra
adabra
abra
a
```

La fonction strtok (1/2)

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\0')           printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else                       printf(" ? ");
    printf("\n");
}

int main(void) {
    char s[] = "-4,3a,2"; char *token;
    printf("Chaîne s = \"%s\" avant segmentation\n", s);
    print_string(s, 8);
    printf("\nSegmentation...\n");
    token = strtok(s, ","); // Premier appel de strtok sur s
    while (token != NULL) {
        printf("token = %s\n", token);
        token = strtok(NULL, ","); // Appels suivants sur NULL
    }
    printf("\n");
    printf("Chaîne s = \"%s\" après segmentation\n", s);
    print_string(s, 8);
}
```


La fonction strtok (2/2)

Résultat:

Chaîne s = "-4,3a,2" avant segmentation

0	1	2	3	4	5	6	7
-	4	,	3	a	,	2	\0

Segmentation...

token = -4

token = 3a

token = 2

Chaîne s = "-4" après segmentation

0	1	2	3	4	5	6	7
-	4	\0	3	a	\0	2	\0

- La fonction strtok **modifie** la chaîne qu'elle segmente
- En écrasant les délimiteurs avec des caractères '\0'
- Pour des raisons d'**efficacité**
- **Copier** la chaîne avant segmentation pour la conserver

Autres fonctions

Standard

- `memcpy`, `memmove`, `memchr`, `memcmp`, `memset`: manipule des octets
- `strcoll`: compare deux chaînes selon la locale
- `strerror`: chaîne de caractère correspondant à une erreur
- `strspn`: longueur d'un préfixe contenant certains caractères
- `strcspn`: longueur d'un préfixe ne contenant pas certains caractères
- `strpbrk`: recherche d'octets
- `strxfrm`: transforme une chaîne

Non standard

`memccpy`, `mempcpy`, `strcat_s`, `strcpy_s`, `strdup`, `strerror_r`, `strerror_r`, `strlcat`, `strncpy`, `strsignal`, `strsep`, `strtok_r`, etc.

Pointeurs de fonctions

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

```
#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;
}
```

Résultat:

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

Tableau de fonctions

```
#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

```
#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 i, 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;
}
```

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
```