Chapitre 4
Chaînes de caractères

Plan du chapitre 4

1. Définition et exemples

Introduction

- Les chaînes de caractères sont essentielles pour créer des interfaces utilisateurs
- Les messages textuels peuvent être affichés ou saisis
- En C, une chaîne de caractères est simplement une liste de caractères terminée par un caractère nul ‘\0’

```
// tableau de caractères
char chaine[] = "Hello";
```

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’
0x12340	0x12341	0x12342	0x12343	0x12344	0x12345

Définition des chaînes en C

- Tableaux de caractères
  - Adresse du premier caractère : `char*` ou `const char*`
  - Il faut suffisamment de place en mémoire aux adresses suivantes sinon le résultat est indéterminé !
- Se terminant par \0
  - Une chaîne de caractère est terminée par un caractère « \0 »
  - Ajouté implicitement aux littéraux mais ceci n'est pas vérifié par le compilateur
  - Si le « \0 » manque (pas de terminaison), les résultats seront indéterminés !
    - note : \0 est le tout premier caractère de la table ASCII

Exemples et contre-exemples

```
// char chaine0[];
// ne compile pas, car la taille ne peut être déterminée

char chaine1[] = {'E', 'r', 'r', 'o', 'r'};
// il manque '\0' : le tableau est correct, mais inutilisable
// comme chaîne C : l'affichage par
// printf("%s", chaine1) est indéterminé après Error

char chaine2[] = {'H', 'e', 'l', 'l', 'o', '\0'};
// correct mais lourd, d'où le raccourci suivant :

char chaine3[] = "Hello";
// les tailles de chaine2 et de chaine3 valent 6 (\0 ajouté)

char chaine4[5] = "Hello";
// même effet que chaine1, pas de \0

char chaine5[10] = "Bonjour";
// correct, toute la place n'est pas utilisée, taille 10 |Bonjour\0\0\0|
```

Exemples et contre-exemples

```
char chaine5[10] = "Bonjour";

for (int i = 0; i < 12; ++i)
   printf("%c", chaine5[i]);
// Bonjour□□□?? ... dépassement de la zone mémoire réservée

char* chaine6;     // correct, pointeur sur un caractère

chaine6 = chaine5;
// affecte à chaine6 l'adresse de chaine5
// affichée comme "Bonjour" mais sa taille est celle d'un pointeur

for (int i = 0; i < 6; ++i)
   chaine6[i] = chaine2[i];
// on modifie chaine6, attention à garder/copier le \0
// l'affichage de chaine6 par printf("%s", chaine6)
// donne "Hello", tout comme chaine2 et chaine5
```

2. Littéraux et variables de type chaîne

Littéraux (ou constantes chaîne)

- Un littéral de type chaîne en C est simplement
une série de caractères délimitée par des doubles guillemets " "

```
printf("Hello"); // un string littéral
```

- Ces chaînes littérales peuvent contenir des caractères d'échappement
au moyen de la barre oblique inversée

```
printf("Hello\nworld"); // ... avec un échappement
```

- Le caractère de code ASCII zéro (*null character*) est un octet
avec tous les bits à zéro, représenté par \0.

Littéraux (ou constantes chaîne)

- Un littéral peut occuper plusieurs lignes grâce au symbole \

```
printf("Hello\
␣ ␣ ␣ ␣ World");  // affiche Hello ␣ ␣ ␣ ␣ World
```

- Si plusieurs littéraux se suivent, séparés par des espaces, le compilateur les joint en une seule chaîne (concaténation)

```
printf("*p = %" PRId64 " *q = %" PRId64 "\n", *p, *q);
```

Littéraux (ou constantes chaîne)

- Lorsqu'une chaîne littérale (ou « constante chaîne ») apparaît dans un programme, le compilateur lui alloue une zone mémoire, avec une adresse sur le début
- L'allocation est faite en mémoire statique avant l'exécution du programme
- Si plusieurs littéraux identiques apparaissent, le compilateur peut ou non utiliser une seule zone (c'est souvent le cas)

```
const char* ptr1 = "Hello";
const char* ptr2 = "Hello";
printf("%p - %p", (void*)ptr1, (void*)ptr2);
// 0x100003fa2 - 0x100003fa2
```

Littéraux (ou constantes chaîne)

- Un littéral de type chaîne est stocké comme un tableau de caractères constants interdisant sa modification, raison du `const char*`
  L'erreur est relevée à l'exécution ou à la compilation selon les compilateurs

```
// chaîne littérale
const char* s = "Hello";
s[1] = 'a'; // erreur
```

```
// tableau de caractères
char s[] = "Hello";
s[1] = 'a'; // pas de problème
```

- Il peut être utile d'utiliser des indices avec les littéraux

```
char digitToHexa(uint8_t digit) {
  assert(digit < 16);
  return "0123456789ABCDEF"[digit];
}
```

Variables de type chaîne en C

- La suite de caractères doit se terminer par le caractère '\0'
- Toutes les fonctions sur les chaînes en C s'attendent à trouver un '\0', sinon undefined behavior

```
char chaine1[5] = "1234";
char chaine2[5] = "Hello";
printf("%s", chaine2); // Hello1234
```

- Bonnes pratiques pour déclarer une chaîne pouvant contenir 80 caractères

```
#define TAILLE 80
char chaine[TAILLE + 1];
```

```
#define TAILLE 80
typedef char Chaine[TAILLE + 1];
Chaine chaine;
```

Initialisation des variables de type chaîne

- Déclaration et initialisation

```
char s[6] = "Hello";
```

  - ici, "Hello" n'est pas un littéral, mais l'abréviation d'un initialiseur de tableau avec un \0 implicitement ajouté
  - il équivaut à `char s[6] = {'H','e','l','l','o','\0'};`

- Si l'initialiseur est plus court que l'espace déclaré le compilateur mettra les autres caractères à zéro

```
char s[10] = "Hello"; // 5 fois \0 ajouté à la fin
```

- Un initialiseur d'une chaîne ne peut pas être plus long que l'espace déclaré, mais pourrait avoir la même taille ☹ (en C)
  - si c'est le cas, le caractère \0 n'est pas ajouté (chaîne sans terminaison)

Tableau ou pointeur

```
char tab[]       = "Hello";  // tableau de caractères
const char* ptr = "Hello";  // pointeur sur une chaîne littérale
```

- Utilisation d'un tableau (tab) : permet de modifier les caractères de *tab* mais pas *tab* lui-même (voir slide suivant)
- Utilisation d'un pointeur (ptr) : le pointeur *ptr* pointe vers un littéral de type chaîne, qui ne devrait pas être modifié, mais *ptr* est une variable qui peut aussi pointer vers d'autres chaînes (attention à la perte du littéral !!)
- Grâce à la similarité entre tableaux et pointeurs en C, on peut utiliser les deux versions comme chaînes de caractères
- Note : l'allocation dynamique s'applique aussi aux chaînes

Tableau ou pointeur

L'affectation globale d'un tableau n'est pas possible

```
char chaine1[10] = "Hello";
char chaine2[10] = chaine1; // erreur
chaine2 = chaine1;          // erreur
// chaine1 et chaine2 sont des tableaux
// donc des adresses constantes !
```

… mais possible si dans une structure

(à l'initialisation comme à l'affectation)

```
typedef struct {
   char s[10];
} Chaine;

Chaine chaine1 = {"Hello"};
Chaine chaine2 = chaine1;

chaine2 = chaine1;
```

Tableau ou pointeur

Que donne la comparaison globale de deux tableaux ?

```
char chaine1[10] = "Hello";
char chaine2[10] = "Hello";

// légal mais toujours faux
if (chaine1 == chaine2)
   ...
```

… et de deux chaînes ?

```
char  chaine1[10] = "Hello";
char* chaine2     = chaine1;

// légal mais comparaison des adresses
if (chaine1 == chaine2)
   ...
```

```
char  chaine1[10] = "Hello";
char* chaine2     = chaine1;

// légal mais compare les 1er caractères
if (*chaine1 == *chaine2)
   ...
```

3. Les chaînes en entrée et sortie

Affichage d'une chaîne sur `stdout`

Deux options fournies par la librairie `stdio.h`

```
int puts(const char* s);
```

en cas de succès retourne une valeur >= 0 sinon le code EOF

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

int main(void) {
   char nom[] = "World";
   printf("Hello %s\n", nom);
   puts(nom);
   return EXIT_SUCCESS;
}
```

Et bien sûr la fonction `printf` déjà étudiée au chapitre no 1

```
int printf(const char* format, ...);
```

Saisie d'une chaîne

- La lecture d'une chaîne (depuis `stdin` ou un fichier) peut être dangereuse si ce qui est lu par le programme est trop long et dépasse la réservation prévue pour la variable de type chaîne

```
#define TAILLE 80
char nom[TAILLE + 1];
scanf("%s", nom); // N.B. : nom et pas &nom !
// l'input ne doit pas dépasser 80 caractères
```

Saisie avec scanf

- Lors de l'appel de `scanf`
  - le programme ignore les premiers blancs du buffer d'entrée
  - il lit des caractères et les stocke dans l'argument de `scanf`
    - jusqu'au premier espace, tab, ou saut de ligne (retour chariot)
    - les caractères suivants restent dans le *buffer* d'entrée
  - ajoute un caractère nul ('\0') à la fin de la chaîne lue
- On peut rendre `scanf` plus sûr avec le descripteur de format `"%ns"`
  - *n* indiquera le nombre maximal de caractères à stocker
  - les caractères au-delà de *n* restent dans le buffer d'entrée

```
#define TAILLE 80
char nom[TAILLE + 1];
scanf("%80s", nom);
```

Saisie avec scanf "%ns"

Comment éviter le codage "en dur" de n ?

Variante 1

- Procéder en deux étapes (`xstr` puis `str`) permet de s'assurer que `TAILLE` est d'abord remplacé par sa valeur avant d'être transformée en chaîne de caractères

```
#define str(x) #x
#define xstr(s) str(s)
#define TAILLE 80

char nom[TAILLE + 1];
scanf("%" xstr(TAILLE) "s", nom);
```

Saisie avec scanf "%ns"

Variante 2

- Pour construire le descripteur de format, on peut également utiliser `sprintf`, qui écrit dans une chaîne (il existe aussi `sscanf`)
  - il faut allouer assez de place pour la chaîne générée

```
int sprintf(char* str, const char* format, ...);
int sscanf(const char* str, const char* format, ...);
```

```
#define TAILLE 80
char nom[TAILLE + 1];
char format[5];
sprintf(format, "%%%ds", TAILLE);
scanf(format, nom);
```

Manipulation des tampons E/S

- L'affichage avec `printf` peut ne pas se faire tout de suite après l'instruction : l'OS n'écrit pas immédiatement le tampon de sortie à chaque modification
  - `int fflush(FILE* flux)` vide le tampon associé au flux
  - `fflush(stdout)` force l'écriture du tampon de sortie
  - `fflush(stdin)` sous Windows et avec certains compilateurs, vide le tampon d'entrée ; pas recommandé toutefois car pas portable
    - meilleure solution car toujours portable : `while (getchar() != '\n');`
- Constantes utiles dans `<stdio.h>`
  - `EOF` : valeur entière indiquant qu'une fin de fichier ou d'entrée-sortie a été atteinte (souvent -1)
  - `stdout, stdin, stderr` = pointeurs de type FILE* associés aux E/S standard

4. Manipulations de chaînes

Opérateurs sur les chaînes

- En C++ (et d'autres langages), on dispose de nombreux opérateurs et fonctions sur les chaînes, pour copier, comparer, concaténer, extraire, etc.
  - mais en C, les opérateurs de base ne fonctionnent pas sur les chaînes
  - les chaînes étant des tableaux, on a les mêmes limitations que pour ceux-ci
  - on ne peut pas copier ou comparer des chaînes avec les opérateurs habituels du C (opérateurs `=` `==` `!=`)

- Rappel : dans la déclaration `char nom[] = "Hello";`
  '=' n'est pas un opérateur d'affectation, mais une initialisation

Copie d'une chaîne vers une autre

```
#include <string.h>
char* strcpy (char* dest, const char* src);
char* strncpy(char* dest, const char* src, size_t n);
```

- `strcpy` copie la chaîne qui se trouve à l'adresse `src` vers la zone mémoire commençant à l'adresse `dest`
  - la copie inclut le caractère final '\0'
  - les deux zones mémoire ne doivent pas se superposer
  - la zone `dest` doit avoir une taille suffisante (sinon le comportement est indéterminé)
- `strncpy` est semblable, mais elle copie au plus n caractères de la chaîne `src`
- Rappel : il existe également `memcpy` et `memmove`

Cas particulier de `strncpy`

- Si la chaîne `src` a plus que `n` caractères (ou exactement `n` caractères), la copie dans `dest` n'aura pas de caractère '\0' de fin

- Si la chaîne `src` a moins de `n` caractères, `strncpy` écrit dans `dest` des caractères '\0' jusqu'à arriver à `n`

- Le comportement est indéterminé si :
  - la zone `dest` n'est pas suffisamment grande pour stocker les caractères copiés
  - les zones `src` et `dest` se superposent
  - si `dest` ne pointe pas vers un tableau de caractères
  - si `src` ne pointe pas vers une chaîne terminée par '\0'

Longueur d'une chaîne

```
#include <string.h>
size_t strlen(const char* s);
```

- `strlen` calcule la longueur de s, excluant le dernier '\0'

```
strlen("Hello");
// retourne 5
```

- Exemple d'implémentation de `strlen`

```
size_t strlen(const char* s) {
   const char* p = s;
   while (*p) ++p;
   return (size_t)(p - s);
}
```

- Implémentations réelles et plus efficaces
  - https://github.com/esmil/musl/blob/master/src/string/strlen.c
  - https://github.com/lattera/glibc/blob/master/string/strlen.c

Concaténation de deux chaînes

```
#include <string.h>
char*  strcat(char* dest, const char* src);
char* strncat(char* dest, const char* src, size_t n);
```

- `strcat` prolonge la chaîne se trouvant à `dest` par la chaîne `src`
  - cherche et écrase le caractère '\0' à la fin de `dest`, mais rajoute un '\0' à la fin du résultat
  - `src` et `dest` ne doivent pas se superposer
  - la zone de `dest` doit être de taille suffisante
- `strncat` est semblable, mais utilise au plus n caractères de `src`.
- Contrairement à `strncpy`, un '\0' est toujours ajouté à la fin de la chaîne, quel que soit le critère qui arrête la concaténation.

Concaténation de deux chaînes

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

char* concat(const char* lhs, const char* rhs) {
   char* chaine = (char*)calloc(strlen(lhs) + strlen(rhs) + 1, sizeof(char));
   if (chaine)
      strcat(strcpy(chaine, lhs), rhs);
   return chaine;
}

int main(void) {
   char* chaine = concat("Hello", " world");
   printf("%s", chaine);
   free(chaine);
   return EXIT_SUCCESS;
}
```

Comparaison de deux chaînes

```
#include <string.h>
int  strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);
```

- `strcmp` compare les deux chaînes données en argument et retourne un entier
  - positif si `s1` est supérieure à `s2` (différence des deux caractères aux mêmes positions)
  - nul si les deux chaînes sont identiques
  - négatif si `s1` est inférieure à `s2` (différence des deux caractères aux mêmes positions)
- `strncmp` compare les n premiers caractères (au plus) de `s1` et `s2`
- Exemple d'implémentation très concise
  - https://github.com/esmil/musl/blob/master/src/string/strcmp.c

Recherche dans les chaînes

```
char* strchr(const char* str, int c);
```

- cherche le caractère `c` (donné comme `int`) dans la chaîne `str`
- retourne un pointeur sur la 1ère occurrence de `c`, ou NULL si pas trouvé

```
char* strrchr(const char* str, int c);
```

- même fonction, mais depuis la fin (« reverse »)

```
char* strstr(const char* str, const char* sub);
```

- cherche la chaîne `sub` dans la chaîne `str`
- retourne un pointeur sur la 1ère occurrence de `sub`, ou NULL si pas trouvée

Décomposition d'une chaîne

```
char* strtok(char* string, const char* delims);
```

- Décompose la chaîne `string` en ses éléments lexicaux (*tokens*), où `delims` contient les caractères séparateurs
- Fonctionnement assez particulier, « à mémoire statique »
  - lors d'un premier appel, on obtient un pointeur sur le début du premier *token*, sa fin est marquée avec '\0', et un pointeur caché est positionné sur cette fin
  - pour obtenir les *tokens* suivants, on appelle `strtok` avec NULL en 1er argument elle fait les mêmes choses que ci-dessus
    - la fin de `string` est signalée en retournant NULL
- La chaîne originale est modifiée : les séparateurs sont remplacés par '\0'
  → stocker la chaîne originale si nécessaire

Exemple avec strtok

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

int main(void) {
   char str[] = "- This, a sample string.";
   const char* delims = " ,.-";
   char* pch = strtok(str, delims);
   while (pch != NULL) {
      printf("%s\n", pch);
      pch = strtok(NULL, delims);
   }
   return EXIT_SUCCESS;
}
```

```
This
a
sample
string
```

Source : http://www.cplusplus.com/reference/cstring/strtok/

5. Fonctions sur les caractères et fonctions de conversion

Test de caractères : <ctype.h>

- Fonctions de la forme `int isxxx(int c);`
  qui indiquent à quelle catégorie appartient le caractère `c`
  - le résultat est interprété comme ‘true’ ou ‘false’
  - on parle de ‘lettre’ au sens ASCII, sans accents

- Fonctions de transformation : `int tolower(int c);`
  `int toupper(int c);`
  - retournent la minuscule / majuscule du caractère `c`
  - si ce n’est pas possible, retournent le caractère `c`

- Retournent des `int` pour des raisons historiques
  - «caster» en un `char` si besoin d’afficher

Test de caractères : <ctype.h>

- `int isalnum(int c);`
  est une lettre ou un chiffre
- `int isalpha(int c);`
  est une lettre (minuscule ou majuscule)
- `int islower(int c);`
  est une lettre minuscule
- `int isupper(int c);`
  est une lettre majuscule
- `int isdigit(int c);`
  est un chiffre
- `int isxdigit(int c);`
  est un chiffre hexadécimal

- `int ispunct(int c);`
  est un caractère de ponctuation
- `int isspace(int c);`
  est un espace, un tab, une fin de ligne ou un retour chariot
- `int isgraph(int c);`
  est affichable et non blanc (code ASCII de 33 à 126)
- `int isprint(int c);`
  est un caractère affichable (code ASCII de 32 à 126)
- `int iscntrl(int c);`
  est un caractère de contrôle code ASCII de 0 à 31 et 127 (DEL)

Conversions en nombres

- Définies dans `<stdlib.h>` et non dans `<string.h>`
- Fonctions anciennes (pour rétrocompatibilité)
  - `atof`, `atol`, `atoi`, …
- Fonctions après C99 : plus générales, plus complexes
  - `strtod`, `strtol`, `strtoul`, …
- Exemple : `double atof(const char* str);`
  - convertit une chaîne `str` en un nombre réel (de type double) comme si la chaîne était lue au clavier
  - s'arrête sur le 1er caractère qui ne peut être interprété
  - si aucune valeur ne peut être construite, renvoie zéro (0.0)

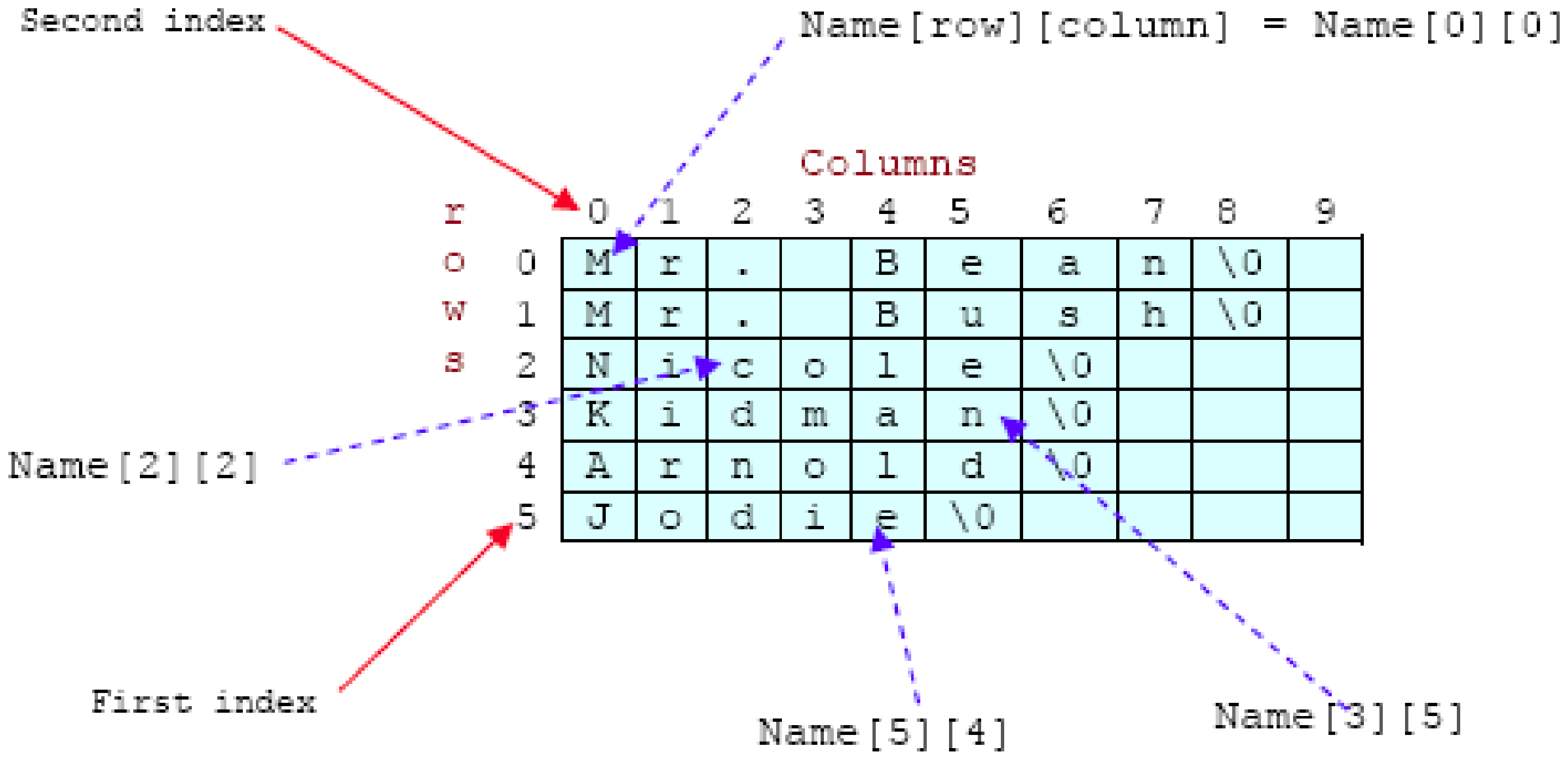
Conversions en nombres

Exemple : `double strtod(const char* str, char** end);`

- conversion d'une chaîne vers un réel (double)
- les espaces en tête sont ignorés
- la conversion s'arrête au 1er caractère ne faisant pas partie d'un nombre réel, et `end` prend l'adresse de ce caractère (pointeur par adresse)
- si la conversion n'est pas possible : 0.0 en retour, et `end` sera l'adresse de début de `str`
- si `end` est initialement NULL, il est ignoré mais alors il est impossible de savoir où la lecture s'est terminée

6. Tableaux de chaînes

Tableaux à tailles fixes




```
char Name[6][10] = {"Mr._Bean", "Mr._Bush",
                    "Nicole", "Kidman", "Arnold", "Jodie"};
```

Tableau de pointeurs

- Pour ne pas imposer une taille fixe à chaque ligne (économie de mémoire) on peut déclarer un tableau de chaînes comme tableau de pointeurs

```
const char* planets[] = {
  "Mercury",
  "Venus",
  "Earth",
  "Mars",
  "Jupiter",
  "Saturn",
  "Uranus",
  "Neptune",
  "Pluto"
};
```

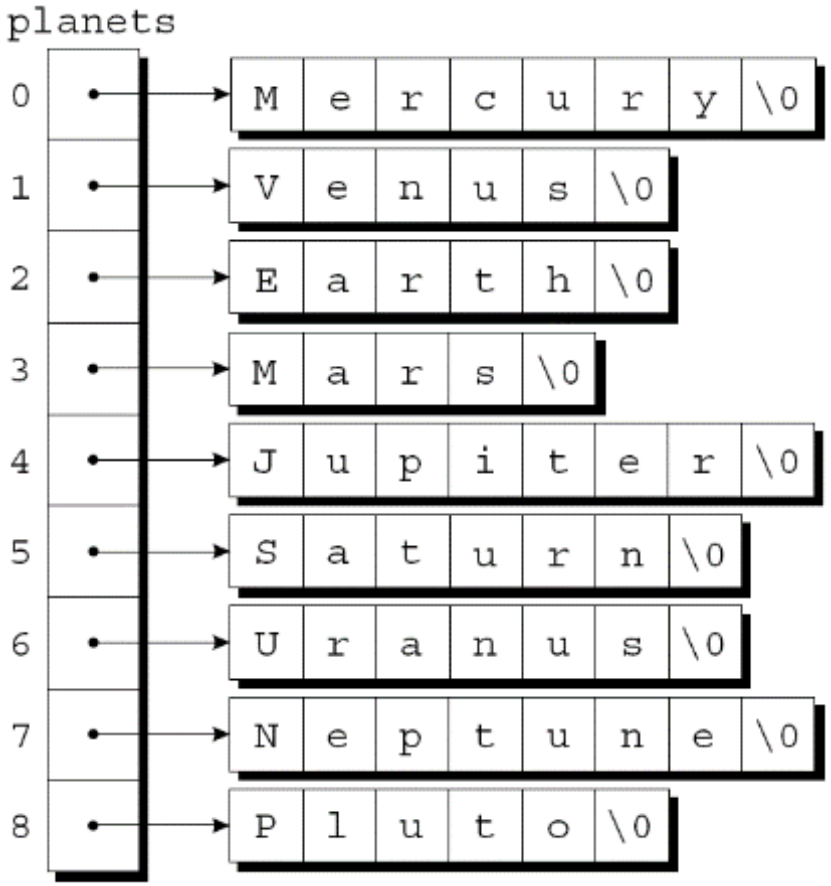
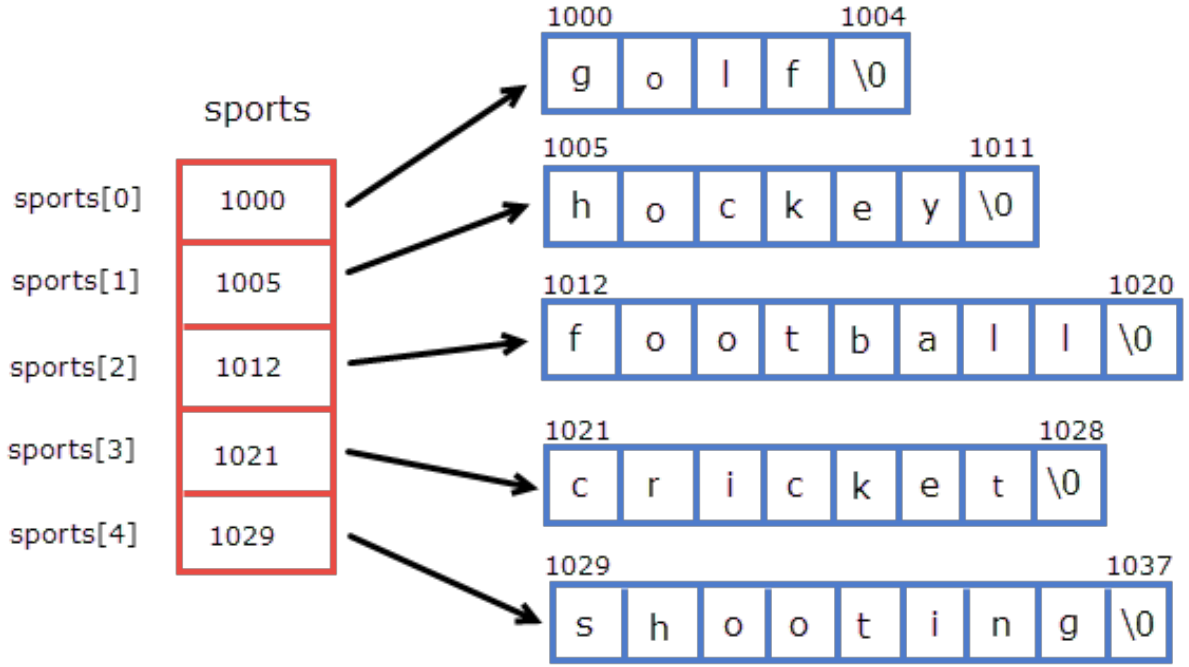


Tableau de pointeurs (mémoire)

- Les tableaux de chaînes de longueur variable (*jagged arrays*) occupent seulement la place nécessaire pour le tableau de pointeurs + les caractères des chaînes + les '\0'

- Pas de garantie que les chaînes se suivent dans la mémoire



https://overiq.com/c-programming/101/array-of-pointers-to-strings-in-c/

7. Les arguments de la ligne de commande

(un avant-goût de la programmation système)

Rôle de l'OS

- Une fois un code C compilé, le programme s'exécute dans un « environnement » préparé et fourni par le système d'exploitation
  - l'OS contrôle l'accès des exécutables au processeur, à la mémoire, et aux périphériques (cf. cours SYE)
  - l'OS permet de lancer une exécution
    - par double-clic s'il y a une interface graphique
    - via des scripts (p.ex. au démarrage)
    - via une ligne de commande (interface texte) : nom de l'exécutable + paramètres
- Exemple de paramètres fournis à l'exécutable gcc
  - `gcc -O3 -o test test.c` = 4 chaînes ("-O3", "-o", "test", "test.c") mais 3 sens différents : argument principal (test.c), fichier de sortie (option -o, argument test) et niveau d'optimisation (option -O, argument collé 3)

Arguments en ligne de commande

- On souhaite gérer dans un programme en C les arguments fournis par l'OS
- Solution prévue : utiliser deux paramètres de « main »

```
int main(int argc, char* argv[]) {...}
```

  - argc ("*argument count*") nombre d'arguments dans la ligne
  - argv ("*argument vector*") tableau de pointeurs vers les arguments

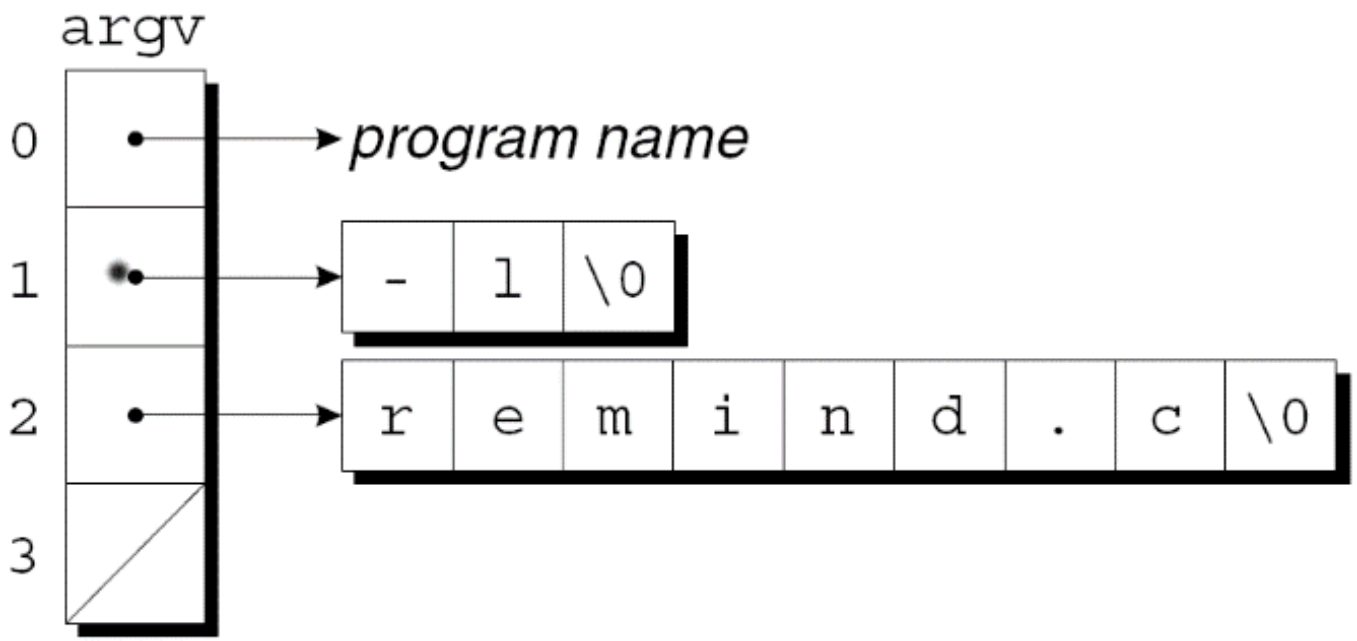
  N.B. Les deux noms `argc` et `argv` sont habituels, mais pas obligatoires

- Structure de argv[]
  - argv[0] = commande utilisée pour l'exécution (nom du programme)
  - argv[1] à argv[argc-1] = autres arguments (séparés par des espaces)
  - argv[argc] est le pointeur NULL

Exemple

```
> ls -l remind.c
```

- `argc` vaut 3
- `argv` contient les pointeurs suivants



Analyse des paramètres de argv

`argv[]` est un tableau de pointeurs vers des chaînes

- l'accès aux valeurs des chaînes est aisé ☺
  mais l'identification correcte des options est à prévoir dans le programme ☹
  - elles viennent dans un ordre variable
  - syntaxe habituelle : nom d'attribut + valeur, séparés ou non par des espaces (`--X=1` ou `-x 1` ou `-x1`)

```
char** p;
for (p = &argv[1]; *p != NULL; ++p) {
   printf("%s\n", *p);
} // pour les afficher
```

Analyse des paramètres de argv

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

int main(int argc, char* argv[]) {

   printf("\nboucle for avec argc ...\n");
   for (int i = 0; i < argc; ++i)
      printf("argv[%d] = %s\n", i, argv[i]);

   printf("\nboucle while sans argc...\n");
   char** ptr = argv;
   while (*ptr) {
      printf("argv[%d] = %s\n", (int)(ptr - argv), *ptr);
      ++ptr;
   }

   return EXIT_SUCCESS;
}
```

Variables d'environnement

- Longue série de paires `attribut=valeur`
  p.ex. `SHELL=/bin/bash` ou `WINDIR=C:\WINDOWS`
  - visibles depuis l'OS avec la commande `SET` en mode console Windows ou la commande `env` sous Linux
- En C (standard ISO), la librairie `stdlib.h` fournit

  ```
  char* getenv(const char* name);
  ```

  retourne la valeur de la variable d'environnement `name`
  - sur des OS conformes à POSIX, on dispose de `putenv`
- Liste des variables d'environnement
  - la librairie `stdlib.h` la fournit via : `extern char** environ`

Variables d'environnement

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

int main(void) {

   char** ptr = environ;

   printf("liste des variables d'environnement\n");
   while (*ptr) {
      printf("- %s\n", *ptr);
      ++ptr;
   }
   printf("\n");
   printf("une variable d'environnement en particulier\n");
   printf("HOME = %s\n", getenv("HOME"));

   return EXIT_SUCCESS;
}
```

Valeur de retour d'un exécutable

- Le standard C spécifie que deux constantes doivent être disponibles, `EXIT_SUCCESS` et `EXIT_FAILURE`, que l'on peut passer à `exit` pour indiquer la fin (normale) réussie ou non

```
int main(void) {
  return EXIT_SUCCESS;  // ou EXIT_FAILURE
}  // cela appelle void exit(int status)
```

- l'appel de `return` cause un appel de `exit`
- cela entraîne la terminaison normale du processus
- retourne au processus parent (OS) la valeur `status & 255`
- toutes les fonctions enregistrées avec `atexit` sont appelées (voir PRG1)

8. Résumé

Résumé

- Bien comprendre la différence entre
  - tableau de caractères

```
char chaine[] = "Hello";
```

  - pointeur sur un littéral chaîne de caractères

```
const char* ptr = "Hello";
```

- Attention au danger des réservations mémoires insuffisantes (concaténation, lecture, …)
- Importance du caractère de terminaison '\0' nécessaire pour toutes les fonctions manipulant des chaînes de caractères

Résumé

- Faire la différence entre
  - Tableau de tableaux de caractères
  - Tableau de pointeurs

```
char  nom[3][10] = {"Jean", "Paul", "Joe"};
```

```
char* nom[3]     = {"Jean", "Paul", "Joe"};
```

- Comprendre les arguments en ligne de commande
  - Se souvenir que `argv[argc]` vaut `NULL`