

Objectifs

Chaînes de  
caractères

Pointeurs sur  
fonctions

Récapitulation

Cast

Conclusion

# Programmation « orientée système »

## LANGAGE C – POINTEURS (3/5)

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle  
Faculté I&C

# Objectifs du cours d'aujourd'hui

Les pointeurs en pratique :

- ▶ chaînes de caractères
- ▶ pointeurs sur fonction
- ▶ forçage de type (casting)

# Chaînes de caractères

différent de Java !

À la différence d'autres langages, C n'offre pas de type de base pour la manipulation des chaînes de caractères.

**En C, une chaîne de caractères est codée dans un tableau de (ou pointeur sur des) caractères.**

**NOTE :** La fin de la chaîne de caractères est indiquée par le caractère nul (noté `'\0'` ou `(char) 0`)

Du point de vue organisation de la mémoire, on a donc strictement :

```
char nom[6] = { 'H', 'e', 'b', 'u', 's', '\0' };
```

**Attention !** Ce n'est pas comme cela qu'on écrit en pratique ! ("Hebus")

☞ il y a heureusement quelques facilités de programmation supplémentaires !



## Chaînes de caractères (2)

La déclaration d'une chaîne de caractères peut se faire

- ① par une **constante** littérale (entre guillemets) :

```
"Bonjour"
```

(Note : `\` pour représenter le caractère `"`)

Cette constante est un tableau de caractères *qui inclut le caractère nul à la fin*

- ② par une variable de taille fixe (tableau) :

```
char nom[25];
```

```
char nom_fichier[FILENAME_MAX];
```

```
char const welcome[] = "Bonjour";
```

- ③ par une allocation dynamique (pointeur) :

```
char* nom;
```

☞ Ne pas oublier de faire l'allocation (`calloc/malloc`)

☞ penser que pour stocker une chaîne de  $n$  caractères il faut **allouer de la place pour  $n+1$  caractères** (en raison du `'\0'` final).



# Affectation de chaînes de caractères



**Attention !** Voici une erreur courante concernant l'affectation de chaînes de caractères :

```
char* s = "bonjour";
```

N'est en général **PAS CORRECT !** (bien que cela fonctionne ! Mais que se passe-t-il en réalité ?)

La seule bonne façon de faire est  
soit d'utiliser la fonction `strncpy` (si `s` doit être modifiée) :

```
strncpy(s, "bonjour", TAILLE);
```

Note : il faut bien sûr avoir **alloué `s` avant** (à `TAILLE+1` éléments),  
mais aussi mettre le `'\0'` final que `strncpy` ne garantit pas :

```
char* s = calloc(TAILLE+1, 1);
```

L'utilisation du `=` avec une valeur littérale ("`blablabla`")  
n'est sans risque **QUE** lors de l'initialisation d'un tableau  
statique :

```
char s[] = "Bonjour";
```

soit d'utiliser `const`  
(si `s` ne doit pas être mo-  
difiée) :

```
const char* s  
= "bonjour";
```

# Fonctions de la bibliothèque string

```
char* strcpy(char* dest, char const* src);
```

copie la chaîne `src` dans la chaîne `dest`. Retourne `dest`.

**Attention !** aucune vérification de taille n'est effectuée ! **Préférez** `strncpy` !

```
char* strncpy(char* dest, char const* src, size_t n);
```

copie les `n` premiers caractères de `src` dans `dest`. Retourne `dest`.

**Attention !** n'ajoute pas le `'\0'` à la fin si `src` contient plus de `n` caractères !

```
char* strcat(char* dest, char const* src);
```

ajoute la chaîne `src` à la fin de la chaîne `dest`. Retourne `dest`.

**Attention !** aucune vérification de taille n'est effectuée ! **Préférez** `strncat` !

```
char* strncat(char* dest, char const* src, size_t n);
```

ajoute au plus `n` caractères de `src` à la fin de `dest`. Retourne `dest`.

```
int strcmp(char const* s1, char const* s2);
```

Compare (ordre alphabétique) les chaînes `s1` et `s2`. Retourne un nombre négatif si `s1 < s2`, 0 si les deux chaînes sont identiques et un nombre positif si `s1 > s2`.

**Préférez** `strncmp` !

## Fonctions de la bibliothèque `string` (2)

```
int strncmp(char const* s1, char const* s2, size_t n);
```

comme `strcmp`, mais ne compare au plus que les `n` premiers caractères de chacune des chaînes.

```
size_t strlen(char const * s);
```

Retourne le nombre de caractères dans `s` (**sans** le caractère nul de la fin).

```
char* strchr(char const* s, char c);
```

Retourne un pointeur sur la première occurrence de `c` dans `s`, ou `NULL` si `c` n'est pas dans `s`

```
char* strrchr(char const* s, char c);
```

idem que `strchr` mais en partant de la fin. Retourne donc la dernière occurrence de `c` dans `s`.

```
char* strstr(char const* s1, char const* s2);
```

retourne le pointeur vers la première occurrence de `s2` dans `s1` (ou `NULL` si `s2` n'est pas incluse dans `s1`).



Il existe plusieurs autres fonctions dans `string`. Pour en savoir plus : `man 3 string`

# Lecture/Écriture

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

Écriture d'une chaîne de caractères `s` :

```
printf("...%s...", s);
```

ou

```
puts(s);    (qui ajoute un retour à la ligne à la fin)
```

ou

```
fputs(s, stdout);    (lui n'ajoute rien)
```

Lecture d'une chaîne de caractères `s` :

```
scanf("%s", s);
```

ou (mieux ! car fixe une taille limite)

```
fgets(s, taille, stdin);
```

☞ cf `printf` et `scanf` dans le cours sur les entrées/sorties (semaine 4).

# [hors cours] Monsieur, et en C++... ? [hors cours]

La manipulation des chaînes de caractères est beaucoup plus simple en C++. Il existe en effet dans la bibliothèque standard la classe `string`, très similaire à `StringBuffer` en Java.

Exemple :

```
#include <string>
// [...]
string chaine ;           // -> chaine vaut ""
string chaine2("test") ;
chaine = "test3" ;
chaine = chaine2 ;       // -> chaine vaut "test"
// mais attention a la semantique de = en C++ :
chaine = 'a' ;           /* -> chaine vaut "a", mais
                        *   chaine2 vaut toujours "test"
                        */
chaine = "Un petit " + chaine2 + " !";
chaine.insert(2, "joli ");
// etc...
```



# Les chaînes de caractères



Valeur littérale : `"valeur"`

Déclarations :

```
char* nom;  
char nom[taille];  
char nom[] = "valeur";
```

Écriture : `printf("...%s...", chaine);` ou `puts(chaine);`

Lecture : `scanf("%s", chaine);` ou `gets(chaine);`

Quelques fonctions de `<string.h>` :

<code>strlen</code>	<code>strcat</code>
<code>strcpy</code>	<code>strncat</code>
<code>strncpy</code>	<code>strchr</code>
<code>strcmp</code>	<code>strrchr</code>
<code>strncmp</code>	<code>strstr</code>

Objectifs

Chaînes de  
caractères

Pointeurs sur  
fonctions

Récapitulation

Cast

Conclusion

# Plan

- ▶ chaînes de caractères
- ▶ **pointeurs sur fonction**
- ▶ forçage de type (casting)

# Pointeurs sur fonctions

En C, on peut en fait pointer sur n'importe quel endroit mémoire.

On peut en particulier **pointer sur des fonctions** (cf exemple d'il y a 2 cours).

La syntaxe consiste à mettre *(\*ptr)* à la place du nom de la fonction.

Par exemple :

`double f(int i);` est une fonction qui prend un `int` en argument et retourne un `double` comme valeur

`double (*g)(int i);` est un **pointeur sur une fonction** du même type que ci-dessus.

On peut maintenant par exemple faire : `g=f;` (identique à `g=&f;`)

puis ensuite : `z=g(i);` (identique à `z=(*g)(i);`)

Note : pas besoin du `&` ni du `*` dans l'utilisation des pointeurs de fonctions.

Pour un exemple complet, voir l'exemple du début du cours sur les pointeurs (il y a 2 cours).

## Pointeurs sur fonctions (2)

Ces pointeurs sur fonctions sont donc un moyen très utile de **passer des fonctions en arguments d'autres fonctions**

Exemple précédent :

```
typedef double (*Fonction)(double);  
...  
double integre(Fonction f, double a, double b) { ... }  
...  
aire = integre(sin, 0.0, M_PI);
```

Plus généralement, on construit des **fonctions « génériques »** ayant comme arguments des pointeurs génériques (`void*`).

On peut ensuite passer ces fonctions génériques à des fonctions très générales.

L'exemple typique est celui du tri `qsort` ([man 3 qsort](#))

# Utilisation de `qsort`



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

`base` est un pointeur sur la zone à trier

`nb_el` est le nombre d'éléments à trier

`size` est la taille d'un élément (utiliser `sizeof` ici)

et `compar` est **la fonction utilisée pour comparer deux arguments** :

cette fonction doit retourner un entier

- ▶ nul en cas d'égalité ;
- ▶ négatif si le premier argument est « plus petit » (vient avant) le second argument ;
- ▶ positif s'il est « plus grand » (vient après) .

# Exemple d'utilisation de `qsort`

```
int compare_int(void const * arg1, void const * arg2) {  
    int const * const i = arg1;  
    int const * const j = arg2;  
    return ((*i == *j) ? 0 : ((*i < *j) ? -1 : 1)) ;  
}  
  
...  
    int tab[NB];  
  
...  
    qsort(tab, NB, sizeof(int), compare_int);
```

# Récapitulons

<code>int i</code>	.....	un entier
<code>int* p</code>	.....	un pointeur sur un entier
<code>int** p</code>	.....	un pointeur sur un pointeur sur un entier
<code>int* f()</code>	.....	une fonction qui retourne un pointeur sur un entier
<code>int (*f)()</code>	.....	un pointeur sur une fonction qui retourne un entier
<code>int* (*f)()</code>	.....	un pointeur sur une fonction qui retourne un pointeur sur un entier
<code>int** f();</code>	.....	une fonction qui retourne un pointeur sur un pointeur sur un entier
<code>int* t[]</code>	.....	un tableau de pointeurs sur des entiers
<code>int (*p)[]</code>	.....	un pointeur sur un tableau d'entiers
<code>int (*f())[]</code>	.....	une fonction qui retourne un pointeur sur un tableau d'entiers
<code>int* (*f())()</code>	.....	une fonction qui retourne un pointeur sur une fonction retournant un pointeur sur un entier
<code>int (*( *(*f()) [] ) () ) []</code>		une fonction qui retourne un pointeur sur un tableau de pointeurs pointant sur des fonctions retournant des pointeurs sur tableaux d'entiers

 **n'hésitez pas à utiliser typedef !**

# Forçage de type (ou «casting»)

En C, il est toujours possible d'interpréter avec un type différent une zone mémoire/variable déclarée dans un premier type.

Cela a pour effet de **convertir** la valeur désignée dans le nouveau type.

Il suffit pour cela de faire précéder la valeur par le type forcé entre `()` :

*(type) expression*

Exemple :

```
double x = 5.4;
int i = (int) x; /* i = 5 */
```

(suppression de la partie fractionnaire, c.-à-d. conversion vers 0)

# Forçage de type (2)



**Attention !** Dans le cas de pointeur, cela **ne change pas** le contenu de la zone/variable en question, mais **uniquement son interprétation**

Exemple :

```
double x = 5.4;
int* i = (int*) &x;

printf("%d\n", (int) x); /* affiche 5 */
printf("%d\n", *i);     /* affiche -1717986918 */
```

## Forçage de type (3)

On utilise le casting essentiellement pour :

- ▶ convertir facilement des valeurs (typiquement d'un type entier à un autre, ou d'un type `double` à un type entier) ;
- ▶ écrire du code « générique » via des pointeurs (`void*`).

Exemple : tri générique ([man 3 qsort](#))

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

```
Personne montab[TAILLE];
...
int compare_personnes(Personne const* p_quidam1,
                      Personne const* p_quidam2);
...
qsort((montab, TAILLE, sizeof(Personne),
      (int (*)(void const*, void const*))compare_personnes);
```

...

# qsort autre solution

... ou comme précédemment ;

les « cast », optionnels en C, étant alors **dans** la fonction `compar` :

```
...
int compare_personnes(void const* arg1, void const* arg2);
...
qsort(montab, TAILLE, sizeof(Personne), compare_personnes);
...
int compare_personnes(void const* arg1, void const* arg2) {
    Personne const* const p_quidam1 = arg1;
    Personne const* const p_quidam2 = arg2;

    ...
    ... *p_quidam1 ...
    ...
}
```

## void\* casts

Les casts depuis ou vers `void*` ont un statut un peu particulier (et différent entre C et C++) :

- ① dans les deux langages (C et C++), l'affectation **vers** `void*` est permise sans cast :

```
int* ptr1; void* ptr2; ... ptr2 = ptr1;
```

- ② En C, un `void*` peut être affecté, sans cast explicite, à un autre pointeur (de tout type).

En C++, par contre, l'affectation *depuis* un `void*` vers un pointeur « non void » n'est pas permise sans cast.

Le code suivant est donc **valide en C** mais *incorrect en C++* :

```
int* ptr1; void* ptr2; ... ptr1 = ptr2;
```



En C++, il faudrait écrire : `ptr1 = static_cast<int*>(ptr2);`

- ③ En C++, la *comparaison* entre `void*` et pointeur quelconque est par contre permise (conversion implicite vers `void*`).

Ceci dit...

# [hors cours] casting en C++ [hors cours]

...le sujet du casting en C++ est un sujet *avancé*, largement hors du cadre de ce cours.

Ce qu'il faut retenir en C++ :

- ▶ il existe **cinq** formes de casting (à ne pas confondre) :  
`const_cast`, `dynamic_cast`, `static_cast`, `reinterpret_cast` et cast « à la C »  
comme présenté précédemment.
- ▶ ne **jamais** utiliser de casting en C++ ; c'est pratiquement toujours le signe d'une  
*mauvaise conception* ;
- ▶ ne **JAMAIS, JAMAIS** utiliser le casting « à la C » en C++ !

# Ce que j'ai appris aujourd'hui

- ▶ ce que sont et comment utiliser les « chaînes de caractères » ;
- ▶ à utiliser les **pointeurs sur fonction** ;
- ▶ la notion de « casting » (forçage de type) :  
à éviter en général, utile pour les « pointeurs génériques ».