CS–202 Computer Systems – C Lectures

C03 – POINTERS 2: COMPLEMENTS

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Chaînes de caractères

différent de Java !

À la différence d'autre 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 !

Jean-Cédric Chappelier

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).

Jean-Cédric Chappelier

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 modifiée) :

```
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` !

Jean-Cédric Chappelier

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 `strchar` 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 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).

Jean-Cédric Chappelier

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>` :

strlen	strcat
strcpy	strncat
strncpy	strchr
strcmp	strrchr
strncmp	strstr

Jean-Cédric Chappelier

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).

Jean-Cédric Chappelier

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`)

Jean-Cédric Chappelier

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) .

Jean-Cédric Chappelier

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

`int i`	un entier
`int* p`	un pointeur sur un entier
`int** p`	un pointeur sur un pointeur sur un entier
`int* f()`	une fonction qui retourne un pointeur sur un entier
`int (*f)()`	un pointeur sur une fonction qui retourne un entier
`int* (*f)()`	un pointeur sur une fonction qui retourne un pointeur sur un entier
`int** f();`	une fonction qui retourne un pointeur sur un pointeur sur un entier
`int* t[]`	un tableau de pointeurs sur des entiers
`int (*p)[]`	un pointeur sur un tableau d'entiers
`int (*f())[]`	une fonction qui retourne un pointeur sur un tableau d'entiers
`int* (*f())()`	une fonction qui retourne un pointeur sur une fonction retournant un pointeur sur un entier
`int (*(*(*f())[])())[]`	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)

Jean-Cédric Chappelier

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 :

```c
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 ...
  ...
}
```


Jean-Cédric Chappelier

`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*`).

Pointeurs et tableaux

On a vu dans les cours et exercices précédents qu'on pouvait par exemple allouer un pointeur sur une zone de 3 `double` :

```
double* ptr;
ptr = calloc(3, sizeof(double));
```

Pourtant `ptr` en tant que tel ne pointe que sur **un** `double` !
(regardez son type : `double*`)

Que vaut `*ptr` ?
☞ la valeur du **premier** `double` stocké dans cette zone.

Comment accéder aux 2 autres ?

☞ avec une syntaxe identique aux tableaux : `ptr[1]` et `ptr[2]`

Jean-Cédric Chappelier

Pointeurs et tableaux (2)

En C, un tableau est en fait très similaire à un **pointeur** (on l'a déjà vu lors du passage d'argument à une fonction) **constant** sur une zone allouée **statiquement** (lors de la déclaration du tableau).

Ainsi `int[]` est pratiquement identique à « `int* const` »
et `*p` est strictement équivalent à `p[0]`

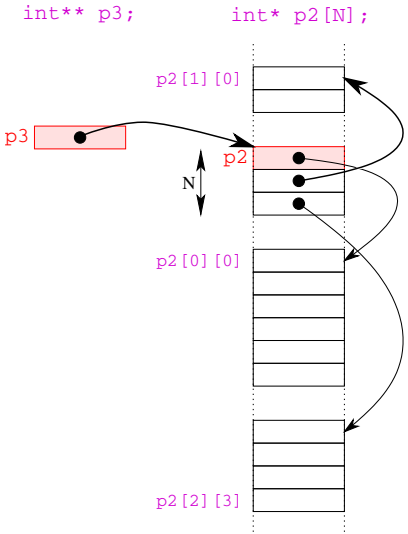
MAIS
`int**` ou `int*[]` sont **très différents** de `int[][]` (qui d'ailleurs n'existe pas en tant que tel ! `int[][M]`, oui)

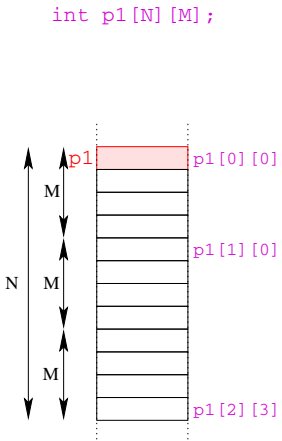
`int**` :

- n'est pas continu en mémoire ;
- n'est pas alloué au départ ;
- les lignes n'ont pas forcément le même nombre d'éléments.

Jean-Cédric Chappelier

Pointeurs et tableaux (2)





Pointeurs et tableaux (3)

Qu'affiche la portion de code suivant ?

```
#define N 2
#define M 14
// ...
  double   p1[N][M];
  double*  p2[N];
  double** p3;

  p3 = calloc(N, sizeof(double*)); // usual checks...

  for (size_t i = 0; i < N; ++i) {
    p2[i] = calloc(M, sizeof(double)); // ...
    p3[i] = calloc(M, sizeof(double)); // ...
  }

  printf("&(p1[1][2]) - p1 = %u doubles\n",
      (unsigned int) &(p1[1][2]) - (unsigned int) p1) / sizeof(double));

  printf("&(p2[1][2]) - p2 = %u doubles\n",
     ((unsigned int) &(p2[1][2]) - (unsigned int) p2) / sizeof(double));

  printf("&(p3[1][2]) - p3 = %u doubles\n",
     ((unsigned int) &(p3[1][2]) - (unsigned int) p3) / sizeof(double));

// ... // en particulier les free() !!
```

Pointeurs et tableaux (4)

Réponse (sur ma machine à un moment donné) :

```
&(p1[1][2]) - p1 = 16 doubles
&(p2[1][2]) - p2 = 151032928 doubles
&(p3[1][2]) - p3 = 49 doubles
```


Jean-Cédric Chappelier

Arithmétique des pointeurs

On peut facilement déplacer un pointeur en mémoire à l'aide des opérateurs + et - (et bien sûr leurs cousins ++, +=, etc.)

« Ajouter » 1 à un pointeur revient à le déplacer « en avant » dans la mémoire, d'un emplacement égal à *une* place mémoire **de la taille de l'objet pointé**.

Exemples (très pratiques) :

```
int tab[N];
...
const int* const end = tab + N;
for (int* p = tab; p < end; ++p) { ... *p ... }
```

```
char* s;  char* p;  char lu;
...
p = s;
while (lu = *p++) { ... lu ... }
```

Explication de l'exemple précédent

```
char* s;  char* p;  char lu;
...
p = s;
while (lu = *p++) { ... lu ... }
```

- Que veut dire `*p++` ?
  Est-ce `*(p++)` ou `(*p)++` ?
- Que fait l'autre (`(*p)++`) ?
- Est-ce que `*p++` est pareil que `*++p` ?
- Pourquoi une variable `lu` plutôt que `*p` directement dans le corps de la boucle ?
  Par exemple : `while(*p++) { ... *p ... }`
- Erreur dans la condition d'arrêt de la boucle ? (`==` au lieu de `=`) ?

Attention !

Attention ! Le résultat de « `p = p + 1` » **dépend du type de `p`** !
(Et c'est souvent là une source d'erreur !)

Le plus simple (avant ce qui va suivre) est de comprendre « `p = p + 1` » comme « *passe à l'objet (pointé) suivant* ».

En clair :

> Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

Il faut éviter de penser aux vraies valeurs (adresses, en tant que nombres entiers), mais si l'on y tient vraiment, on aura donc :

```
(int) (p+1) == (int) p + sizeof(Type)
```

pour `p` un pointeur de type « `Type*` »

Note : on ne peut donc pas faire d'arithmétique des pointeurs sur des `void*` !

Soustraction de pointeurs

On a vu qu'il existait les opérateurs `ptr + int` et `ptr - int` (chacun de type `ptr`).

Il existe aussi `ptr - ptr`

« `p2 - p1` » retourne le nombre d'objets stockés entre `p1` et `p2` (de même type).

Attention ! Le type de cet opérateur (soustraction de pointeurs) est `ptrdiff_t` (défini dans `stddef`).
CE N'EST PAS `int` **!** (ceci est une grave erreur !)

```
ptrdiff_t dp = p2 - p1;
```


Jean-Cédric Chappelier

Pointeurs et tableaux (synthèse)

(Pour `int* p;` `int t[N];` et `int i;`)

`t[i]` est en fait exactement `*(t+i)`

À noter que c'est symétrique... (...et on peut en effet écrire `3[t]` ! !)

`t` est en fait exactement `&t[0]` (et est un `int* const`)

`int t2[N][M]` n'a **rien** à voir avec un `int**` (et est plus proche d'un `int* const`)

`void f(int t[N])` (ou `void f(int t[])`) sont en fait exactement `void f(int* t)` :

- attention à la sémantique de `t` (et en particulier à sa taille) dans le corps de `f` ;
- nécessité absolue de toujours passer la taille de `t` comme argument supplémentaire.

Jean-Cédric Chappelier

Complément (et mise en garde) sur `sizeof`

L'opérateur `sizeof` accepte comme argument soit un type, soit une expression C (laquelle *n'est pas* évaluée)
(et retourne la taille mémoire nécessaire à stocker le type de l'expression en question)

Exemples :

```
int i;
int tab[N];
... sizeof(double) ...
... sizeof(i) ...                    // = sizeof(int)
... sizeof(tab)/sizeof(tab[0]) ... // donne N, mais ATTENTION !!
```

Mais il faut faire attention à ne pas mal l'employer :

```
int tab[1000];
int* t = tab;
... sizeof(t) ... /* combien ca vaut ? */
```


Jean-Cédric Chappelier

Complément (et mise en garde) sur `sizeof`

Attention ! PIRE !

```
#define N 1000

void f(int t[N]) {
  ... sizeof(t)/sizeof(int) ... /* combien ca vaut ? */
}
```

☞ Je répète qu'un tableau passé en argument de fonction n'a **AUCUNE** connaissance de sa taille ! !

Autre (**mauvais**) exemple, plus subtil : où est le bug ? :

```
int tab[1000];
...
const int* const end = tab + sizeof(tab);
for (int* t = tab; t < end; ++t) {
    utiliser(*t, ...);
}
```

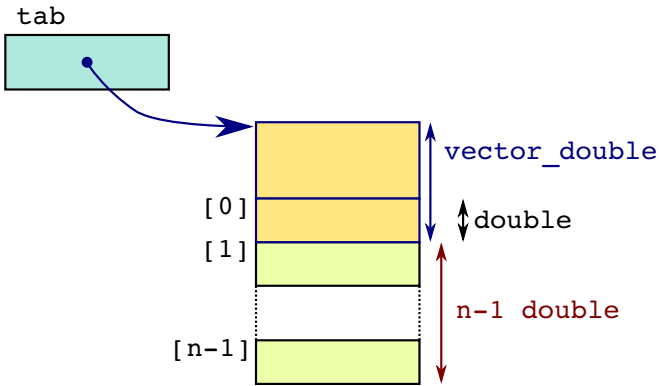
Rappel : flexible array member

```c
struct vector_double {
  size_t size; // nombre d'éléments
  double data[1];
};
```

Ceci permet d'avoir des tableaux dynamiques en C, via l'allocation dynamique :

```c
const size_t N_MAX = (SIZE_MAX - sizeof(struct vector_double)) / sizeof(double) + 1;
if (nb <= N_MAX) {
  struct vector_double* tab = malloc(sizeof(struct vector_double)
                                     + (nb-1)*sizeof(double) );
  if (tab != NULL) {
    tab->size = nb;
  }
}
```

Flexible array member



Utilisation :

```
tab->data[i]
```