I3 – Algorithmique et programmation
Introduction à la programmation en langage C

Cours n°5
Gestion de la mémoire en C
Structures de données et types dérivés

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2011 – 2012

1. Allocation de tableaux
   - Allocation statique
   - Allocation dynamique
   - Utilisation

2. Déclaration de nouveaux types
   - Structure
   - Énumération
   - Remarque : taille
   - Type dérivé

Tableaux

Rappel :

Tableaux

Un tableau est un type particulier de variable. Il contient un **ensemble de variables** de même type, stockées de façon contiguë en mémoire.

Exemple : tableau d'entiers

3	5	2	1	12	5	2	9

Parcours d'un tableau :

```
1 début
2     monTableau[10] : Tableau d'Entiers
3     pour i ⟵ 0 à 9 pas 1 faire
4         monTableau[i] ⟵ 2 * i
5     finpour
6 fin
```

Déclaration d'un tableau

En C :

- Déclaration d'un tableau
- Type des données contenues
- Taille fixe
- On ne peut pas dépasser les limites du tableau (SEGFAULT)

On peut parcourir le tableau avec une boucle `for` (par exemple) :

```
1    int i;
2  /* on a un tableau tab de N entiers */
3    for( i = 0 ; i < N ; i++ ) {
4      printf( "tab[%d] : %d\n", i, tab[i] );
5    }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1   int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1   #define TAILLETAB 10
2   int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1   #define TAILLETAB 10
2   int tab[TAILLETAB];
3
4   for( i = 0 ; i < TAILLETAB ; i++ ) {
5     tab[i] = 0;
6   }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1   int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1   #define TAILLETAB 10
2   int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1   #define TAILLETAB 10
2   int tab[TAILLETAB];
3
4   for( i = 0 ; i < TAILLETAB ; i++ ) {
5     tab[i] = 0;
6   }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1    int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1    #define TAILLETAB  10
2    int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1    #define TAILLETAB  10
2    int tab[TAILLETAB];
3
4    for( i = 0 ; i < TAILLETAB ; i++ ) {
5      tab[i] = 0;
6    }
```

Déclaration statique et initialisation

Remplissage du tableau :

```
1 int tab[3] = { 0, 1, 2};
```

Le compilateur :

- Réserve l'emplacement en mémoire pour 3 cases
- Les remplit avec les valeurs données entre accolades (séparées par des virgules)

Autre possibilité :

- Déclaration sans la taille
- Initisalisation **tout de suite**
- Le compilateur compte le nombre d'éléments donnés pour l'initialisation et il réserve la taille nécessaire

```
1 int tab[] = { 0, 1, 2};
```

Allocation dynamique

La taille du tableau **n'est pas** connue **à la compilation**

- On ne peut pas réserver d'espace mémoire à la compilation
- On déclare un tableau et on l'alloue plus tard

Utilisation d'un **pointeur**. Rappel :

Définition

Un pointeur est une **adresse** vers une zone mémoire.

```
int* i;
```

int* i;

Non-initialisé, un pointeur ne pointe sur **rien**

- Il peut pointer vers une variable : `int* i = &k;`
- On peut **allouer une zone mémoire** vers laquelle il pointe

Allocation dynamique

La taille du tableau **n'est pas** connue **à la compilation**

- On ne peut pas réserver d'espace mémoire à la compilation
- On déclare un tableau et on l'alloue plus tard

Utilisation d'un **pointeur**. Rappel :

Définition

Un pointeur est une **adresse** vers une zone mémoire.

```
int* i;
```

int* i;

Non-initialisé, un pointeur ne pointe sur **rien**

- Il peut pointer vers une variable : `int* i = &k;`
- On peut **allouer une zone mémoire** vers laquelle il pointe

Allocation dynamique

Allocation d'une zone mémoire : fonction `malloc()`

- Prototype : void *malloc(size_t size);
- Définie dans stdlib.h : `#include <stdlib.h>`
- Retourne un pointeur vers l'espace mémoire alloué
- L'espace mémoire alloué est un tampon de mémoire contigüe

Remarques :

- Le type `size_t` est un entier
- `malloc()` retourne un pointeur de type non spécifié (`void*`)
  - Transtypage pour utiliser un pointeur du type désiré

Exemple :

```
1  #define TAILLETAB   10
2  int* i;
3  i = (int*) malloc( TAILLETAB * 4 );
```

Allocation dynamique

Allocation d'une zone mémoire : fonction `malloc()`

- Prototype : void *malloc(size_t size);
- Définie dans stdlib.h : `#include <stdlib.h>`
- Retourne un pointeur vers l'espace mémoire alloué
- L'espace mémoire alloué est un tampon de mémoire contigüe

Remarques :

- Le type `size_t` est un entier
- `malloc()` retourne un pointeur de type non spécifié (`void*`)
  - Transtypage pour utiliser un pointeur du type désiré

Exemple :

```
1  #define TAILLETAB   10
2  int* i;
3  i = (int*) malloc( TAILLETAB * 4 );
```

Utilisation de malloc()

Exemple précédent : non portable !

- Taille transmise en dur `TAILLETAB * 4`
- Et si un entier est codé sur autre chose que 4 octets ?

Fonction qui retourne la taille d'un élément :

- `size_t sizeof( type )`
- Retourne un entier (`int` ou `size_t`)
- Prend le nom du type en argument

```
1 #define TAILLETAB    10
2 int* tab;
3 tab = (int*) malloc( TAILLETAB * sizeof( int ) );
```

Attention : `malloc()` retourne **toujours** un pointeur utilisable (contrairement à ce que dit le man)

- Si le système n'a pas pu allouer la mémoire : `bus error` lors de l'utilisation (plantage sur SIGBUS)

Modification de la taille d'un espace alloué

Utilisation de la fonction **realloc()**

- Prototype : `void* realloc(void *ptr, size_t size);`
- Modification de la taille d'un tampon mémoire
- Concrètement : réallocation d'un nouvel espace mémoire, copie du contenu de l'ancien dans le nouveau (ce qui tient dans le nouveau)

Attention : le pointeur vers le tableau change

- On récupère un nouveau pointeur en retour de la fonction

```
1 #define TAILLETAB    10
2 int* tab;
3 int taille;
4 taille = TAILLETAB;
5 tab = (int*) malloc( taille * sizeof( int ) );
6 taille = taille * 2;
7 tab = (int*) realloc( tab, taille * sizeof( int ) );
```

Libération de l'espace mémoire

La mémoire allouée doit être **libérée** !!

- Utilisation de la procédure **free()**
- Prototype : `void free( void* ptr );`
- Le système libère le tampon mémoire pointé par le pointeur

```
1 #define TAILLETAB    10
2 int* tab;
3 tab = (int*) malloc( TAILLETAB * sizeof( int ) );
4 free( tab );
```

Après avoir libéré une zone mémoire, il est plus sûr de mettre le pointeur qui pointait dessus à la valeur `NULL`

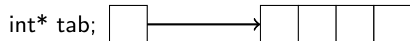
- Éviter de l'utiliser ultérieurement par erreur

```
1 if( NULL != tab ) {
2   free( tab );
3   tab = NULL;
4 }
```

Représentation mémoire d'un tableau

Début d'un tableau = adresse contenue dans le pointeur

- Puis décalage case par case suivant la taille des éléments contenus dans le tableau



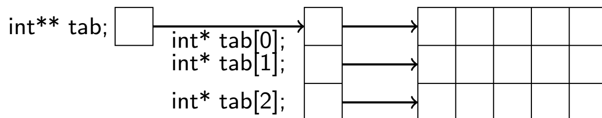

```
1  int* tabI;
2  double* tabD;
3  int i;
4
5  tabI = (int*) malloc( TAILLETAB \
6                      * sizeof( int ) );
7  tabD = (double*) malloc( TAILLETAB \
8                      * sizeof( double ) );
9
10 printf( "Tableau d'entiers:\n" );
11 for( i = 0 ; i < TAILLETAB ; i++ ) {
12   printf( "%d : %p\n", i, &(tabI[i]) );
13 }
14
15 printf( "Tableau de doubles:\n" );
16 for( i = 0 ; i < TAILLETAB ; i++ ) {
17   printf( "%d : %p\n", i, &(tabD[i]) );
18 }
```

Affichage obtenu :

```
Tableau d'entiers:
0 : 0x100100080
1 : 0x100100084
2 : 0x100100088
3 : 0x10010008c
4 : 0x100100090
Tableau de doubles:
0 : 0x1001000a0
1 : 0x1001000a8
2 : 0x1001000b0
3 : 0x1001000b8
4 : 0x1001000c0
```

Tableau à plusieurs dimensions

Utilisation d'un **tableau de pointeurs** sur des tableaux :



Exemple : tableau d'entiers à 2 dimensions

- Déclaration d'un pointeur sur un tableau de pointeurs sur des entiers
- Alloction du tableau de pointeurs
- Allocation de tous les tableaux sur entiers

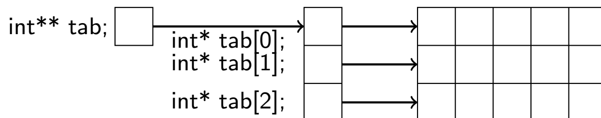
```
#define LIGNES    3
#define COLONNES  5
int** tab2d;
int i;

tab2d = (int**) malloc( LIGNES \
                  * sizeof( int* ) );
for( i =  ; i < LIGNES ; i++ ) {
  tab2d[i] = (int*) malloc( COLONNES \
                  * sizeof( int ) );
}
```

Accès :

- `tab[i]` désigne la i-ème ligne → pointeur vers le tableau correspondant à la ligne
- `tab[i][j]` désigne la j-ème case du tableau de la ligne i

Tableau à plusieurs dimensions

Utilisation d'un **tableau de pointeurs** sur des tableaux :



Exemple : tableau d'entiers à 2 dimensions

- Déclaration d'un pointeur sur un tableau de pointeurs sur des entiers
- Alloction du tableau de pointeurs
- Allocation de tous les tableaux sur entiers

```
#define LIGNES    3
#define COLONNES  5
int** tab2d;
int i;

tab2d = (int**) malloc( LIGNES \
                 * sizeof( int* ) );
for( i =  ; i < LIGNES ; i++ ) {
  tab2d[i] = (int*) malloc( COLONNES \
                  * sizeof( int ) );
}
```

Accès :

- `tab[i]` désigne la i-ème ligne → pointeur vers le tableau correspondant à la ligne
- `tab[i][j]` désigne la j-ème case du tableau de la ligne i

Parcours d'un tableau

Utilisation de l'indice entre crochets

- `tab` désigne l'**adresse du début** d'un tableau
- `tab[i]` désigne la **i-ème case** du tableau
- `&(tab[i])` désigne l'**adresse de la i-ème case** du tableau

Arithmétique de pointeurs

Si on déclare un tableau `int* tab`

- `tab` désigne l'adresse du 1er élément du tableau
  - Équivalent à `&( tab[0] )`
- `*tab` désigne l'élément pointé par `tab`
  - Équivalent à `tab[0]`
- On accède à l'élément suivant en ajoutant 1 au pointeur
  - `tab+1` pointe sur l'élément qui suit l'élément pointé par `tab`
  - `*(tab+1)` désigne l'élement pointé par `tab+1`, donc `tab[1]`

On peut effectuer des **opérations arithmétiques** sur un pointeur. Exemple :

```
1  #define TAILLETAB    10
2  int tab[TAILLETAB];
3  int* p;
4  int i;
5  p = tab;
6  for( i = 0 ; i < TAILLETAB ; i++ ) {
7    printf( "Adresse : %p ", p );
8    printf( "Valeur : %d\n", *p );
9    p++;
10 }
```

- p est initialisé à l'adresse de début du tableau (ligne 5)
- On *incrémente* `p` ligne 9 : on passe à l'élément suivant

Application aux chaînes de caractères

Chaîne de caractères = tableau de caractères

Exemple : Coucou

C	o	u	c	o	u	“0

Le dernier élément est le caractère spécial “0

- Caractère de fin de chaîne de caractères
- Attention : pour une chaîne de N caractères, il faut donc un tableau de taille N+1

Déclaration d'une chaîne de caractères : `char* chaine;`

Opérations sur des chaines de caractères : la plupart déclarées dans string.h

- `#include <string.h>`

Opérations sur les chaînes de caractères

Longueur d'une chaine de caractères :

- `size_t strlen( char* str );`
- Prend en paramètre une chaîne de caractères
- Retourne un entier de type `size_t`

Copie d'une chaîne de caractères :

- `char* strcpy( char* dest, char* orig );`
- Copie `orig` dans `dest`
- Retourne `dest` : souvent inutilisé
- La chaîne `dest` doit être allouée

Attention : `strcpy` n'est pas sûr (risques de débordement de tampons, exploitable comme une faille de sécurité)

- Utilisation de `strncpy( )`
- Prototype : `char* strncpy( char* dest, char* orig, size_t taille );`
- Copie `taille` caractères de `orig` dans `dest`

Copie brute d'un tampon dans un autre :

- `void *memcpy( void *dest, void *orig, size_t n );`

Opérations sur les chaînes de caractères

Longueur d'une chaine de caractères :

- `size_t strlen( char* str );`
- Prend en paramètre une chaîne de caractères
- Retourne un entier de type `size_t`

Copie d'une chaîne de caractères :

- `char* strcpy( char* dest, char* orig );`
- Copie `orig` dans `dest`
- Retourne `dest` : souvent inutilisé
- La chaîne `dest` doit être allouée

Attention : `strcpy` n'est pas sûr (risques de débordement de tampons, exploitable comme une faille de sécurité)

- Utilisation de `strncpy( )`
- Prototype : `char* strncpy( char* dest, char* orig, size_t taille );`
- Copie `taille` caractères de `orig` dans `dest`

Copie brute d'un tampon dans un autre :

- `void *memcpy( void *dest, void *orig, size_t n );`

Opérations sur les chaînes de caractères

Longueur d'une chaine de caractères :

- `size_t strlen( char* str );`
- Prend en paramètre une chaîne de caractères
- Retourne un entier de type `size_t`

Copie d'une chaîne de caractères :

- `char* strcpy( char* dest, char* orig );`
- Copie `orig` dans `dest`
- Retourne `dest` : souvent inutilisé
- La chaîne `dest` doit être allouée

Attention : `strcpy` n'est pas sûr (risques de débordement de tampons, exploitable comme une faille de sécurité)

- Utilisation de `strncpy( )`
- Prototype : `char* strncpy( char* dest, char* orig, size_t taille );`
- Copie `taille` caractères de `orig` dans `dest`

Copie brute d'un tampon dans un autre :

- `void *memcpy( void *dest, void *orig, size_t n );`

1. Allocation de tableaux
   - Allocation statique
   - Allocation dynamique
   - Utilisation

2. Déclaration de nouveaux types
   - Structure
   - Énumération
   - Remarque : taille
   - Type dérivé

Définition d'une structure

Définition d'une structure de données contenant des éléments de types différents

```
1 struct maStruct {
2   type1 champ1;
3   type2 champ2;
4   ...
5 };
```

Syntaxe

- Utilisation du mot-clé **struct**
- Nom de la structure
- Définition des champs
- Attention au point-virgule final

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
```

Utilisation d'une structure

Déclaration d'un élément du type de la structure :

- struct + nom de la structure + nom de la variable

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
6
7 struct etudiant et1;
```

Possibilité d'utiliser des structures dans les champs d'une structure

```
1 struct date {
2   int jour;
3   int mois;
4   int annee;
5 };
```

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   struct date dateDeNaissace;
5 };
```

Utilisation d'une structure

Déclaration d'un élément du type de la structure :

- struct + nom de la structure + nom de la variable

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
6
7 struct etudiant et1;
```

Possibilité d'utiliser des structures dans les champs d'une structure

```
1 struct date {
2   int jour;
3   int mois;
4   int annee;
5 };
```

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   struct date dateDeNaissace;
5 };
```

Utilisation d'une structure (suite)

Accès aux champs de la structure :

- nom de la variable + point (.) + nom du champ

Exemple :

```
1 struct etudiant et1;
2 et1.dateDeNaissace = { 0, 0, 0 };
```

Cas d'un pointeur

- Si on déclare un pointeur vers un élément du type de cette structure
- Utilisation d'une flèche (->) pour accéder à ses champs

```
1 struct etudiant* et2;
2 et2 = (struct etudiant*) malloc( sizeof( etudiant ) );
3 et2->dateDeNaissace = { 0, 0, 0 };
```

Déclaration d'une énumération

On déclare une **énumération** en énumérant les valeurs possibles de ses variables

- Utilisation du mot-clé **enum**
- Énumération de toutes les valeurs possibles
- Le compilateur se charge de convertir dans le type de données interne le plus adéquat (généralement un entier)

Exemple :

```
1 enum couleur { ROUGE, VERT, BLEU };
```

Possibilité de forcer la valeur interne prise par chaque valeur
Exemple :

```
1 enum booleen {
2     TRUE = 1,
3     FALSE = 0
4 };
```

Utilisation : `enum` + nom de l'énumération + nom de la variable

```
1 enum couleur c1 = ROUGE;
```

Remarque sur la taille des structures

La taille de la zone mémoire occupée par une structure n'est pas forcément égale à la somme des tailles des champs qui la composent !

```
1 struct str1 {
2     char c1;
3     int i1;
4     char c2;
5     int i2;
6 };
```

```
1 struct str2 {
2     char c1;
3     char c2;
4     int i1;
5     int i2;
6 };
```

Le compilateur aligne les données des champs des structures

- Blocs de 4 octets
- Bourrage entre les champs

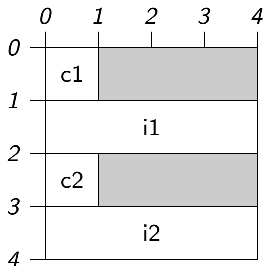
```
printf( "str1 : %d, str2 : %d\n",
        sizeof( struct str1 ),
        sizeof( struct str2 ) );
```

Affichage obtenu :

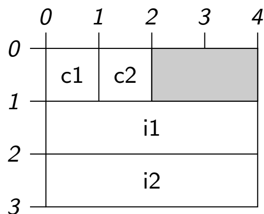
```
str1 : 16, str2 : 12
```

Occupation mémoire des structures

```
1 struct str1 {
2     char c1;
3     int i1;
4     char c2;
5     int i2;
6 };
```




```
1 struct str2 {
2     char c1;
3     char c2;
4     int i1;
5     int i2;
6 };
```



Éviter le bourrage à la compilation

NB : le bourrage a un intérêt

- Optimisation pour aligner les données sur les structures internes du CPU et la mémoire de la machine
- Transferts de données mémoire ↔ CPU plus rapides

Le compilateur gcc peut prévenir l'utilisateur en cas de bourrage dans une structure :

- Option `-Wpadded` : affiche un avertissement (warning) si une structure est bourrée (padded)

```
$ gcc -Wpadded -o taillesStruct taillesStruct.c
```

On peut demander à gcc de "packer" les structures

- Pas de bourrage utilisé
- Option `-fpack-struct`

```
$ gcc -fpack-struct -o taillesStruct taillesStruct.c
```

Toutes les structures du programme seront packées.

Éviter le bourrage à la compilation

NB : le bourrage a un intérêt

- Optimisation pour aligner les données sur les structures internes du CPU et la mémoire de la machine
- Transferts de données mémoire ↔ CPU plus rapides

Le compilateur `gcc` peut prévenir l'utilisateur en cas de bourrage dans une structure :

- Option `-Wpadded` : affiche un avertissement (warning) si une structure est bourrée (padded)

```
$ gcc -Wpadded -o taillesStruct taillesStruct.c
```

On peut demander à `gcc` de "packer" les structures

- Pas de bourrage utilisé
- Option `-fpack-struct`

```
$ gcc -fpack-struct -o taillesStruct taillesStruct.c
```

Toutes les structures du programme seront packées.

Éviter le bourrage à la compilation

NB : le bourrage a un intérêt

- Optimisation pour aligner les données sur les structures internes du CPU et la mémoire de la machine
- Transferts de données mémoire ↔ CPU plus rapides

Le compilateur gcc peut prévenir l'utilisateur en cas de bourrage dans une structure :

- Option `-Wpadded` : affiche un avertissement (warning) si une structure est bourrée (padded)

```
$ gcc -Wpadded -o taillesStruct taillesStruct.c
```

On peut demander à gcc de "packer" les structures

- Pas de bourrage utilisé
- Option `-fpack-struct`

```
$ gcc -fpack-struct -o taillesStruct taillesStruct.c
```

Toutes les structures du programme seront packées.

Éviter le bourrage en programmant

On peut définir les structures individuellement comme packées en donnant une directive de compilation

- `__attribute__((packed))`
- Seule cette structure sera packée
- Les structures qui sont définies sans cette directive de compilation ne le seront pas

```
1 struct str1 {
2     char c1;
3     int i1;
4     char c2;
5     int i2;
6 } __attribute__((packed));
```

Affichage obtenu en packant toutes les structures de données :

```
str1 : 10, str2 : 10
```

Définition d'un type dérivé

Définition d'un nouveau type :

- Utilisation du mot-clé **typedef**
- Ancien type + nouveau type

Redéfinition d'un type existant

```
1 typedef int entier;
```

Définition d'un type associé à une structure

```
1 typedef struct etudiant etudiant_t;
```

Définition d'un type dérivé

Définition d'un nouveau type :

- Utilisation du mot-clé **typedef**
- Ancien type + nouveau type

Redéfinition d'un type existant

```
1 typedef int entier;
```

Définition d'un type associé à une structure

```
1 typedef struct etudiant etudiant_t;
```

Définition d'un type dérivé

Définition d'un nouveau type :

- Utilisation du mot-clé **typedef**
- Ancien type + nouveau type

Redéfinition d'un type existant

```
1 typedef int entier;
```

Définition d'un type associé à une structure

```
1 typedef struct etudiant etudiant_t;
```

Définition d'un type dérivé (suite)

On peut définir le type en même temps que la structure :
Exemple avec une énumération :

```
1         typedef enum { TRUE, FALSE } booleen;
```

Utilisation :

```
1         booleen b1 = TRUE;
```

Exemple avec une structure :

```
1 typedef struct {
2   int jour;
3   int mois;
4   int annee;
5 } date_t;
```

Utilisation du type

- On utilise directement le nom du type

```
1 date_t d1;
```

Définition d'un type dérivé (suite)

On peut définir le type en même temps que la structure :
Exemple avec une énumération :

```
1        typedef enum { TRUE, FALSE } booleen;
```

Utilisation :

```
1        booleen b1 = TRUE;
```

Exemple avec une structure :

```
1 typedef struct {
2   int jour;
3   int mois;
4   int annee;
5 } date_t;
```

Utilisation du type

- On utilise directement le nom du type

```
1 date_t d1;
```