Notes de cours C

Thibaut Marmey

November 16, 2018

Contents

1 Bases

1.1 Général

- **int main(int argc, char **argv)** : *argc* représente le nombre d'arguments lors de l'exécution du programmes. Sans aucun argument *argc = 1*. ***argv* représente quant à lui la liste des arguments déclarés lors de l'exécution du programme.

- Si l'on veut passer un int en paramètre du programme, utliser : *atoi(argv[indice])* Cela permet de convertir le *char* récupéré comme paramètre en *int*.

- Taille des différents types de données

type	size	value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or ...
unsigned int	2 or 4 bytes	0 to 65,535 or ...
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	...
unsigned long	4 bytes	...
double	8 bytes	...

- La condition *if* est *True* si la condition est différente de 0.
  Ainsi si l'on a *if(1/10)* la condition est *False*. *Explication :* 1 et 10 sont deux int, le résultat est de 1/10 est 0.1 mais le résultat est aussi un int donc la troncature du int donne 0.
  if(-1) est quant à lui *True* car -1 est différent de 0.

- L'opérateur ternaire **?** qui permet de compresser les conditions *if*.

```
// Exp1 ? Exp2 : Exp3;
// Si Exp1 est vrai Exp2 est evalue, sinon Exp3 est evalue
return (a > b) ? a : b;
```

- **Scope rules** : 3 places différentes où les variables peuvent être déclarées : local, global, formal.

data type	initial default value
int	1 byte
char	'\0'
float	0
double	0 bytes
pointer	NULL

- **Unions** : type de données qui permet de stocker différents types de variables dans la même zone mémoire.

```
union Data {
   int i;
   float f;
   char str[20]
} data;
```

  La taille mémoire de l'union est la taille de la variable la plus grande (ici 20 bytes par le char).
  Dans une union un seul membre peut être utilisé à la fois. On accède à un élément à la fois sinon l'élément le plus ancien est corrompu.
  On accède aux éléments comme dans une structure

```
data.i = 1;
// On utilise la variable tout de suite pour ne pas la corrompre
data.f = 20.2;
```

- **C header files** and **once-only headers**

```
#ifndef HEADER_FILE
#define HEADER_FILE
   blabla function
#endif
```

- Use of macros :
  - inline small function, work like template

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

– token expansion, to name multiple variables adding prefixes

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void) {
   int token34 = 40;
   tokenpaster(34);
   return 0;
}

#define DEF_PAIR_OF(dtype) \
  typedef struct pair_of_##dtype { \
      dtype first; \
      dtype second; \
  } pair_of_##dtype##_t

 DEF_PAIR_OF(int);
 DEF_PAIR_OF(double);
 DEF_PAIR_OF(MyStruct);
 /* etc */
```

1.2 Les structures

- *typedef* permet d'assigner des nouveaux noms aux types de données. C'est le plus souvent utilisé quand l'utilisateur créé lui-même un nouveau type.

```
// typedef <data_type> <alias_name>
struct Foo { ... }
typedef struct Foo Foo;
//or with the abbrevation
typedef struct Foo { ... } Foo
```

  Ici le *data_type* serait *struct Foo*. Il n'y a donc plus besoin d'appeler *struct Foo* à chaque fois mais seulement *Foo*.

- Calculer la taille d'une structure : sizeof(structure) **!=** sum sizeof(elements)
  Il y a un allignement de mémoire c'est à dire que comme le processeur lit des mots de 4 octets il rajoute des bits dits de "bourrage". Ainsi il faut calculer la taille d'une structure en multiple de 4.
  Il faut donc calculer la somme des sizeof(elements) et arrondir le résultat au multiple de 4 supérieur. *ex : si la somme des sizeof est égal à 14 la taille de la structure est : sizeof(structure) = 16 octets*

1.3 Pointeur

- ***lien internet :*** **doc cplusplus.com sur les pointeurs**

- **Déclaration d'un pointeur** : la valeur d'un pointeur est une adresse. Initialisation à NULL. Où s'il pointe sur une variable :

```
int *ptr = &var
//ou int var = 1;
int *ptr = NULL;
ptr = &var;
```

- **Changer valeur d'une variable via pointeur :**

```
int a = 1;
int *ptr = &var;
*ptr = 2;
```

  La variable a est alors égale à 2;

- Si ptr est un pointeur pointant sur a = 1. *ptr est la valeur de a.

-

objet	valeur	adresse
i	3	AAAAA
p (pointeur sur i)	AAAAA	CCCCC
*p	3	AAAAA
pp (pointeur sur p)	CCCCC	DDDDD
*pp	AAAAA	CCCCC
**pp	3	AAAAA

- Créer un pointeur et lui donner seulement la possibilité de lire une variable et non la modifier : *const* int *p = &a

- void pointers est un pointeur qui n'a pas de type associé. Il peut prendre tout type de variable (le cas de malloc() ou calloc()). Dans un printf par exemple, le void pointer ptr doit être caster *ex : printf("%d", *ptr)* ne marche pas. L'expression *printf("%d", *(int *)ptr)* quant à elle marche.

- Pointeur d'une fonction généralement pour passer une fonction en argument d'une autre fonction.

```
int operation (int x, int y, int (*functocall)(int,int)) {
   int g;
   g = (*functocall)(x,y);
   return (g);
   }
// pour l'utiliser
m = operation (7, 5, addition);
int (*minus)(int,int) = subtraction; // Pointeur de fonction
n = operation (20, m, minus);
```

1.4 Fonctions

- *lien internet :* Utilisation de malloc

```
int *ptr = (int *)malloc(sizeof(int));
*ptr = var
// Ou bien
int *ptr;
ptr = malloc(sizeof(int))
```

- *free(ptr)* si on ne se sert plus du pointeur
- Si p pointe sur a[0] alors *(p+1) = a[1]
- Utiliser *calloc(nElements, nBytes)* pour les tableaux. Cette fonction initialise toutes les cellules du tableau à zero
- Fonction permettant de commuter deux nombres (int).

```
void commuterNombre(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
   }
// Appel de la fonciton
commuterNombre(&var1, &var2);
```

- Fonction permettant de commuter deux pointeurs (on change seulement les adresses sur lesquelles ils pointent et on ne change pas les valeurs des variables pointées).

```
void commuterPointeur(int **a, int **b) {
   int temp = *a;
   *a = *b;
   *b = temp;
   }
// Appel de la fonction
commuterPointeur(&ptr1, &ptr2);
```

- Méthode pour qu'une fonction renvoie un tableau de int. Tout d'abord il faut savoir que le type de valeur dans le return doit être static c'est pourquoi le tableau déclaré sera de type *int static tableau*. La fonction s'écrit : *int *fonction()*
- Utilisation de *static* data dans une fonction : la *static* data présente dans une fonction a la capacité de garder sa valeur (persistence) même après la sortie de la fonction. Ainsi, s'il y a un *static* compteur dans une fonction, à chaque appel sa valeur va s'incrémenter.
- Fonction à paramètres variables. Ce n'est pas très recommendé car le compilateur n'a pas de contrôle et les erreurs ne sont pas anticipées par celui-ci. Un exemple connu de fonction à paramètres variables est *printf()*.
  Liste des particularités de ces fonctions :
  - La fonction admet n'importe quel type de retour.

- Elle a obligatoirement un paramètre fixe
- Les paramètres variables sont obligatoirement les derniers.

Le rôle du dernier paramètre fixe est de fournir un moyen de déterminer le nombre et éventuellement le type des paramètres variables (donc passés après).
Implémentation :

- Inclure *stdarg.h* pour accéder aux fonctions et macros de manipulation des paramètres variables.
- Définir une variable de traitement des variadics de type va_list = va
- Initialiser cette variable correctement avec va_start(va, compteurDeParamVar) et le nom du dernier paramètre fixe
- Récupérer et traiter les paramètres selon la spécification de la fonction va_arg(va, *type*)
- Terminer l'analyse proprement avec va_end().

Généralement on va utiliser une boucle *for* relié à *compteurDeParamVar* qui va permettre de récupérer tous les paramètres variables placés en argument de la fonction.

```c
int somme(int fixe1, int fixe2 ,int n,...){
    va_list va;
    va_start(va, n);
    // On fait commencer le va_start au dernier paramètre fixe

    int resultat = 0;

    for (int i = 0; i < n ; i++){
        resultat += va_arg(va, int);
    }
    va_end(va);
    return resultat;
}
```

1.5 Pointeur de pointeur

- Pointeur de pointeur :

```c
int a = 1;
int *ptr = &a;
int **pp = &ptr;
```

- *lien internet :* utilisation pointeur de pointeur
- Si pp est un pointeur de pointeur pointant sur le pointeur ptr. On initialise cette variable : int **pp = &ptr; Ainsi *pp correspond à la valeur de ptr qui est l'adresse de la variable pointée par celui-ci. Via pp, on accède à la valeur cette variable avec **pp.

1.6 Pointeur de char

- Pour déclarer un tableau de char modifiable :

```
char tab[] = "abc"
```

- En déclarant le pointeur de char : char *p = ”abc” il n'est plus possible de modifier le string ”abc”.
- Pour printf le pointeur p : *printf(”%s”,p);*
- Dans malloc il y a déjà un free(). Donc pour changer le mot contenu dans un char * via une fonction dans laquelle il est un argument on peut faire de la façon suivante:

```
void changerMot(char **pp, char *mot){
   // permet de free() le pointeur qui a la chaine de caractere
   *p = malloc(sizeof(mot))
   *p = mot; }
// Appel de la fonction
changerMot(&tableauDeChar, "unMot")
```

1.7 Les listes chaînées

- lien tutorial 1 sur openclassroom
- programme liste_chainee.c
- Une liste chaînée est un moyen d'organiser une série de données en mémoire. Cela consiste à assembler des structures en les liant entre elles à l'aide de pointeurs
- La structure est organisée de la manière suivante :
  1. Elément stocké 2. Un poiteur pointant sur la case d'après
- La liste est dite chainée car elle constitue une chaîne de pointeur qui relie les différents éléments.
- A la différence des tableaux, les éléments dans cette liste chaînées ne se suivent pas dans la mémoire. Tous ces éléments sont donc stockés dans différents endroit de la mémoire d'où la nécessité des pointeurs.
- Comment construire une chaîne de caractère ? On va d'abord créer une structure qui va constituer le schéma de chaque maillon dans cette chaîne. Il faut donc initialiser une structure. Considérons une liste chaînée d'entiers.
- Initialisation de la structure Element :

```
typedef struct Element Element;
struct Element {
   int nombre;
   Element *suivant;
   };
```

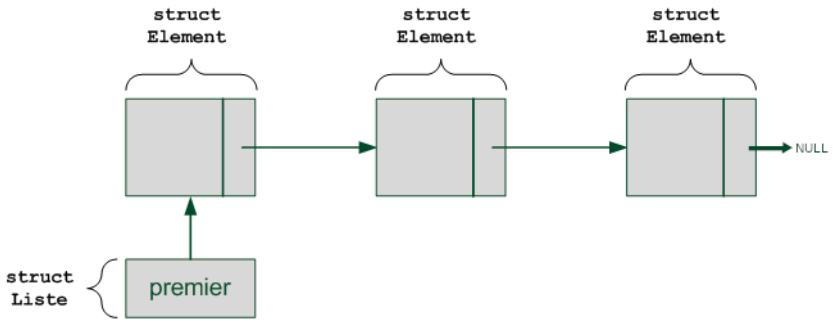
Ce schéma de maillon constitue va permettre de créer une liste dite "simplement chaînée" puisqu'on ne peut connaître que le maillon suivant. Lorsqu'une structure prend en compte le maillon précédent on parle de liste "doublement chaînée". Cela dit, l'utilisation de ce type de chaîne devient alors plus complexe.

- Il faut en plus créer une structure qui va permettre de contrôler l'ensemble de la liste chaînée. Elle contient ainsi le premier élément de la liste chaînée. On peut aussi stocker la taille de la liste chaînée.

- Initialisation de la structure Liste :

```
typedef struct Liste Liste;
struct Liste {
   unsigned int taille;
   Element *premier;
   };
```

- Il faut aussi penser au dernier élément de la liste. Comment faire pour savoir que c'est le dernier ? La solution la plus efficace est de faire pointer le pointeur du dernier élément sur *NULL*.



- Il faut alors créer des fonctions pour manipuler la liste chaînée. Par exemple nous pouvons :

  1. initialiser la liste (on pourrait aussi utiliser : *void initialisation (Liste *liste)*)
  2. ajouter des éléments
  3. supprimer des éléments
  4. afficher les éléments de la liste chaînée
  5. supprimer la liste chaînée

```c
Liste *initialisation()
{
    Liste *liste = malloc(sizeof(*liste));
    Element *element = malloc(sizeof(*element));

    if (liste == NULL || element == NULL )
    {
        exit(EXIT_FAILURE);
    }

    element->nombre = 0;
    element->suivant = NULL;
    liste->premier = element;

    return liste;
}
```

(a) Liste *initialisation()

```c
void insertion(Liste *liste, int nvNombre)
{
    Element *nouveau = malloc(sizeof(*nouveau));

    if (liste == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }

    nouveau->nombre = nvNombre;
    nouveau->suivant = liste->premier;

    liste->premier = nouveau;
}
```

(b) void insertion(Liste*, int)

```c
void suppression(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    if (liste->premier != NULL)
    {
        Element *aSupprimer = liste->premier;
        liste->premier = liste->premier->suivant;
        free(aSupprimer);
    }
}
```

(c) void suppression(Liste*)

```c
void afficherListe(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    Element *actuel = liste->premier;

    while (actuel != NULL)
    {
        printf("%d -> ", actuel->nombre);
        actuel = actuel->suivant;
    }

    printf("NULL \n");
}
```

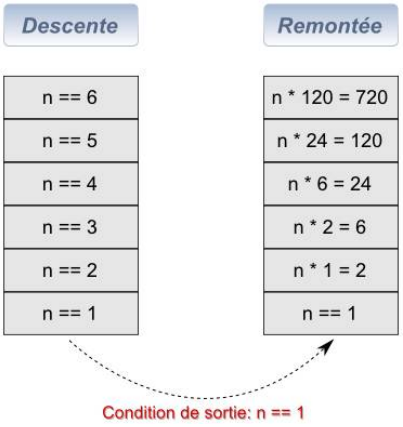
(d) void afficherliste(List*)

1.8 Récursivité

- Lien tutorial récursivité
- fichier programme récursivité
- Exemple de fonction récursive avec la fonction *factorielle* :

```c
unsigned int factorielle(int n){
   if (n == 1)
      return 1L;
   return n * factorielle(n-1)
   }
```

- Les appels des fonctions récursives sont ainsi empilés jusqu'à la condition de sortie ici $n == 1$.



- Cette méthode est appelée *récursivité normale*. Cette méthode est efficace mais peut poser problème si la récursivité est trop longue. En effet, il risque d'y avoir une explosition de lq pile ce qui entraînerait un plantage du programme. C'est pour cela que nous allons voir une autre méthode : *la récursivité terminale*

- Les fonctions récursives normales ont un argument en plus. Il n'y a alors qu'une descente de la pile et donne le résultat final à la fin.

```
unsigned long factoriel_terminal (int n, unsigned long result){
   if (n == 1)
      return result;
   else if (n == 0)
      return 1L;
   return factoriel\textunderscore terminal (n - 1, n * result);
   }
```