TABLEAUX, POINTEURS ET FONCTIONS EN C

2012-10-31

© Emmanuel Chieze
Département d'informatique, UQAM
INF3135

Plan

- Tableaux
- Pointeurs
- Fonctions
- Entrées-Sorties
- Exercices

Tableaux

3

- Collection de données de même type
- Stockée de façon contiguë en mémoire
- Définition (vecteurs)

```
int chiffres[10];
```

- Définition et initialisation

```
int toto[] ={12,-12,34,13,43};
```

- Taille fixée par la définition

Tableaux

4

- Accès à un élément du tableau

```
int toto[] ={12,-12,34,13,43};
int a;
a=toto[2];   /* a vaut 34 */
b=toto[5];   /* valeur de b ? */
```

- Le premier élément correspond à l'indice 0
- Pas de vérification de dépassement de borne à la compilation

Tableaux

5

- Initialisation partielle d'un tableau

```
int titi[4] ={12,-12};
```

  - titi[0] vaut 12
  - titi[1] vaut -12,
  - titi[2] et titi[3] ne sont pas initialisés

Exemple2.1.c : Fonction calculant le produit scalaire de deux vecteurs

Chaînes de caractères

6

- Chaînes de caractères = tableau de caractères
- Chaînes constantes représentées entre " "

```
char chaine[]="toto"  est l'abréviation de
char chaine[]={'t','o','t','o','\0'}
```

- Terminées par la caractère spécial \0
  - Longueur de la chaîne "toto" : 4
  - Taille du tableau représentant la chaîne "toto" : 5
- Autres caractères spéciaux : \n \t \\ \"
  - s'écrivent avec deux caractères
  - ne représentent qu'un seul caractère

Chaînes de caractères

7

```
char chaine[] = "toto";
int i;
for (i=0;i<=4;i++)
  printf ("%d ",chaine[i]);
/* affiche 116 111 116 111 0 */
```

- Exercice : écrire une fonction déclarée par

```
unsigned longueur (char[] chaine)
```

retournant la longueur d'une chaîne de caractères. L'intégrer à un programme pour la tester. (Exemple2.2.c)

Plan

8

- Tableaux
- Pointeurs
- Fonctions
- Entrées-Sorties
- Exercices

Pointeurs

9

- Pointeur =
  - adresse du début d'une donnée en mémoire
  - associé à un type de donnée
    - pour savoir combien d'octets lire à partir de l'adresse spécifiée
    - et pour savoir quel schéma de décodage leur appliquer
- & retourne l'adresse d'une donnée en mémoire

```
int toto = 234;
printf ("toto vaut %d et est stocké à l'adresse hexa
  %p\n",toto, &toto);
```


Pointeurs

10

- Déclaration du pointeur pi pointant vers un entier

```
int *pi, toto = 234; /* Déclaration du pointeur */
pi = &toto;          /* Initialisation du pointeur */
printf ("toto (adresse %p) vaut %d \n",&toto, toto);
printf ("pi = %p et pointe vers %d\n",pi, *pi);
*pi = 350;
printf ("toto (adresse %p) vaut %d \n",&toto, toto);
printf ("pi = %p et pointe vers %d\n",pi, *pi);
```

Affiche :

```
toto (adresse ffbefb54) vaut 234
pi = ffbefb54 et pointe vers 234
toto (adresse ffbefb54) vaut 350
pi = ffbefb54 et pointe vers 350
```

- *pi désigne le contenu pointé par pi

Pointeurs

11

- Affectation d'une valeur à un pointeur : impossible directement

```
int *pi;
pi = 0xdff1;              /* NON */
```

- Utiliser l'opérateur cast pour affecter une valeur à un pointeur

```
pi = (int*) 0xdff1;       /* OK pour le compilateur*/
                          /* NON en pratique */
```

- Utiliser l'opérateur cast pour associer la même adresse à des pointeurs de types distincts

```
int *pi;
char *pc;
pi = (int*) 0xdff1;
pc = (char*) pi;
```


Tableaux et Pointeurs

12

- Un tableau d'entités de type X est un pointeur **constant** vers des entités de type X
- `int a[3]` définit un pointeur a vers des entiers
- a désigne l'adresse du 1er élément du tableau

```
int a[3]={1,2,3}, *pi;
pi = a;     /* Initialisation de pi */
printf("%p %p %d %d %d %d\n", a, pi, a[0], a[1], a[2],
  *pi);
```

affiche

```
ffbefb18 ffbefb18 1 2 3 1
```

- `pi = a` est valide
- `a = pi` n'est pas valide, car a est un pointeur constant

Arithmétique des pointeurs

13

- a représente l'adresse de a[0],
  - a+1 celle de a[1],
  - a+n celle de a[n],
  - peu importe le type du pointeur
- différence de pointeurs de même type
- incrémentation/décrémentation de pointeurs
- comparaison de pointeurs

Arithmétique des pointeurs

14

	a[0]	a[1]	a[2]
Vecteur a	0	4	8
	*a	*(a+1)	*(a+2)

```
int a[3]={0,4,8}, *pi, *pi2;
pi = a;                    /* Initialisation de pi */
pi2 = &a[2];               /* Initialisation de pi2 */
printf("%d\n", pi2-pi);        /* Affiche 2 */
printf("%d\n", *(--pi2));      /* Affiche 4 */
printf("%d\n", *(pi+1));       /* Affiche 4*/
if (pi+1 == pi2) printf("pi et pi2 pointent vers la
  même case mémoire\n");       /* Le message s'affiche
  */
```

Tableaux et Pointeurs

15

```
int *pi, ti[10];
```

- La déclaration `ti[10]` réserve l'espace mémoire nécessaire pour stocker le tableau
- La déclaration `*pi` ne réserve aucun espace mémoire
- `*pi = 6, *(pi+1) = 5` est valide, mais les emplacements pi et pi+1 pourraient être utilisés par le compilateur ultérieurement

Plan

16

- Tableaux
- Pointeurs
- Fonctions
- Entrées-Sorties
- Exercices

Les fonctions en C

17

- Unité de base de programmation en C
  - Effectuent un travail précis
  - Permettent un découpage du code
    - pour une meilleure maintenance du code
    - pour en permettre la réutilisation

Catégorisation des fonctions

18

- Il existe 2 catégories générales de fonctions
  - fonctions pures
    - fonction au sens mathématique du terme
    - résultat ne dépend que des arguments
    - pas d'effet de bord
    - exple : sqrt, log, …
  - autres fonctions
    - fonction dont le résultat dépend de l'environnement
    - ou fonction modifiant l'environnement
    - ou fonction ayant des effets de bord
    - exple : getchar, malloc, …

Arguments et paramètres

- Distinguer
  - paramètres (formels) : variables contenant les valeurs transmises par l'extérieur (dans le code de la fonction)
  - arguments : valeurs spécifiques passées lors d'un appel à une fonction
- Unité de base de programmation en C
  - Reçoivent généralement des données en entrée (paramètres)
  - Renvoient au plus un résultat, nécessairement élémentaire (nombre, pointeur, caractère, structure)

Arguments et paramètres

Définition	Appel	Commentaires
`int fact (int n)`	`toto = fact(5)`	
`int affiche_bonjour()`	`toto = affiche_bonjour()` `toto = affiche_bonjour(1, 2, "cd")`	
`int affiche_bonjour(void)`	`toto = affiche_bonjour()` `toto = affiche_bonjour(1,2,"cd")`	OK Erreur de compilation
`void affiche_message(char * message)`	`affiche_message("Salut")`	

Exécution d'une fonction

21

- Lors de l'appel d'une fonction
  - l'exécution du bloc appelant est suspendue
  - le fonction est exécutée
    - jusqu'à ce que l'instruction return soit exécutée
    - ou jusqu'à la fin de la fonction sinon
  - la valeur retournée par la fonction est retournée au programme appelant, qui peut l'utiliser et reprendre son exécution là où elle avait été suspendue.

Passage des arguments

22

- Les arguments de type atomique sont passés **par valeur**
  - une copie de la valeur est transmise à la fonction
  - la modification de cette valeur au sein de la fonction n'affecte pas le programme principal
  - Cf. Exemple1.3.c

Passage des arguments

23

- Qu'affiche ce programme ?

```
void echange_valeurs(int a, int b);
int main ()
{
   int a=5,b=6;
   echange_valeurs(a,b);
   printf("a=%d b=%d\n", a, b);
   return 0;
}

void echange_valeurs(int a, int b)  {
   int z =a;
   a=b;
   b=z;
}
```


Passage des arguments

24

- Passage des arguments par adresse : utiliser des pointeurs (cf. Exemple2.3.c)

Tableaux et fonctions

25

- Un tableau peut être l'argument d'une fonction

```
float produit_scalaire (float a[], float b[],
  unsigned taille);
```

ou

```
float produit_scalaire (float *a, float *b, unsigned
  taille);
```

  - un tableau équivaut à un pointeur constant
  - un tableau est donc passé par adresse
  - cf. exemple2.4.c
  - utiliser `const` pour limiter le risque de modification du tableau dans la fonction
  - cf. exemple2.5.c

Tableaux et fonctions

26

- Une fonction ne peut retourner un tableau comme résultat
  - faire du tableau résultat l'un des arguments de la fonction, puisque les tableaux sont passés par adresse
  - cf. exemple2.6.c
- Une fonction ne peut pas retourner un pointeur créé dans la fonction
  - Sauf avec allocation dynamique

Déclaration des fonctions

27

- Déclaration des fonctions :
  - au début du fichier où elles sont définies et/ou utilisées.
  - on peut omettre la spécification du nom des variables
- Définition des fonctions :
  - les variables doivent être présentes
- Impossible de définir 2 fonctions de même nom avec des signatures différentes (*conflicting types*)

```
float produit_scalaire (float [], float [], unsigned);
/* ou
float produit_scalaire (float a[], float b[], unsigned taille); */

int main()
…

float produit_scalaire (float a[], float b[], unsigned taille){
    …
}
```


Variables automatiques

28

- On a vu les **variables locales** (éventuellement constantes), dites **variables automatiques**
  - visibles de leur déclaration jusqu'à la fin du bloc où elles sont définies
  - accessibles au sein de leur bloc uniquement
  - supprimées lorsque l'on sort du bloc
  - valeur quelconque si pas d'initialisation explicite

Variables globales

29

- **Variables globales**
  - déclarées en dehors de fonctions
  - visibles de leur déclaration jusqu'à la fin du fichier où elles sont définies
  - utilisables jusqu'à la fin du programme
  - Initialisation par défaut à 0
- Fonctions : même visibilité, accessibilité et durée de vie que les variables globales
- cf. exemple2.7.c

Variables statiques

30

- Variables et fonctions statiques

```
static char tampon[TAILLETAMPON];
static int toto;
static int factorielle(int n);
```

  - Initialisation par défaut à 0
  - Variables locales statiques
    - associées à un espace de stockage permanent
    - existent même lorsque la fonction n'est pas appelée
    - cf. **exemple2.8.c**
  - Variables globales statiques et fonctions statiques
    - identiques aux variables globales et aux fonctions
    - mais **ne peuvent être utilisées en dehors du fichier où elles sont définies**

Variables externes

31

- Variables externes
  - définies dans un autre fichier comme variables globales non statiques
  - déclaration dans le fichier important ces variables

```
extern int toto, titi[];
```

  - pas nécessaire d'indiquer la taille du tableau dans la déclaration externe

La fonction main

32

- Déclaration de base de main

```
int main ()
```

  - n'autorise pas le passage d'arguments
    - les arguments fournis par l'utilisateur seront ignorés
    - Attention : `int main (void)` se comporte comme `int main ()` (puisque les arguments de main ne sont pas connus à la compilation)
  - par convention, main () retourne
    - 0 si le traitement est OK
    - un code d'erreur distinct de 0 sinon

La fonction main

33

- Déclaration plus complète de main

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

  - Permet de récupérer les arguments passés sur la ligne de commande et de les traiter
  - argc : nombre d'arguments (incluant le nom du programme, argv[0])
  - argv : tableau de pointeurs sur les arguments, vus comme chaînes de caractères
  - cf. exemple2.9.c

La fonction main

34

- Conversion de chaînes en nombres (stdlib.h)
  - Fonctions conservées pour des raisons de compatibilité

```
double atof(const char *chaine);
int atoi(const char *chaine);
long int atol(const char *chaine);
```

  - cf. exemple2.10.c

La fonction main

35

- Conversion de chaînes en nombres (stdlib.h)
  - Fonctions recommandées

```
double strtod(const char *chaine, char **fin);
float strtof(const char *chaine, char **fin);
long double strtold(const char *chaine, char **fin);
unsigned long strtoul(const char *chaine, char **fin,
  int base);
long strtol(const char *chaine, char **fin, int base);
```

  - chaine = chaine à analyser
  - fin = reste de la chaine après extraction de la valeur numérique
  - base = base dans laquelle le nombre est exprimé dans la chaîne (pour les entiers seulement)
  - cf. exemple2.11.c

Plan

36

- Tableaux
- Pointeurs
- Fonctions
- Entrées-Sorties
- Exercices

Canaux sous UNIX

37

- UNIX distingue 3 canaux standard de communication entre commandes
  - l'entrée standard (canal 0)
  - la sortie standard (canal 1)
  - le canal d'erreur (canal 2)

Canaux sous UNIX

38

- cat affiche le contenu d'un fichier
  - source :
    - si aucun argument, utilise l'entrée standard, jusqu'à ce que ^D soit saisi (EOF)
    - sinon, affiche le contenu des fichiers mentionnés
  - le résultat s'affiche sur la sortie standard
  - les messages d'erreurs s'affichent sur le canal d'erreur
- ls donne le contenu d'un répertoire
  - l'entrée standard n'est pas utilisée
  - le résultat s'affiche sur la sortie standard
  - les messages d'erreurs s'affichent sur le canal d'erreur

Canaux sous UNIX

39

- Redirections dans des fichiers (tcsh)
  - redirection de l'entrée standard
    - `cat <toto.c`
    - (même comportement que `cat toto.c`, mais processus distinct)
  - redirection de la sortie standard vers le fichier res
    - `ls >res`
    - `ls >>res`
  - redirection de la sortie standard et du canal d'erreur vers le fichier res
    - `ls >& res`
    - `ls >>& res`

Canaux sous UNIX

40

- Redirections dans des fichiers (ksh)
  - redirection de l'entrée standard
    - `cat <toto.c`
    - (même comportement que `cat toto.c`, mais processus distinct)
  - redirection de la sortie standard vers le fichier res
    - `ls >res`
    - `ls >>res`
  - redirection du canal d'erreur vers le fichier res
    - `ls 2> res`
    - `ls 2>> res`
  - redirection du canal d'erreur vers la sortie standard
    - `ls 2>&1`
    - `ls >res 2>&1`

Canaux sous UNIX

41

- Pipelines : la sortie standard d'une commande devient l'entrée standard de la suivante
  - `ls -al | more`
- Exemples
  - `cat <toto >titi`
  - `cat <toto | wc`
  - `cat toto | sort | more`

E/S caractère par caractère (stdio.h)

42

- int getchar()
  - renvoie le prochain caractère lu sur l'entrée standard
  - renvoie EOF en fin de fichier (définie dans stdio.h, vaut généralement -1)
  - => utiliser un type int pour traiter le code ASCII étendu (au lieu de unsigned char)
- int putchar (int c)
  - envoie un caractère sur la sortie standard
- int ungetc(int c, stdin)
  - replace un caractère sur l'entrée standard
  - le 2nd paramètre est une constante ici
  - n'utiliser qu'une fois après un getchar()
- cf. exemple2.12.c

E/S ligne par ligne (stdio.h)

43

- char *gets(char *ligne)
  - renvoie la prochaine ligne lue sur l'entrée standard
    - supprime \n en fin de ligne
    - ajoute \0 en fin de chaîne
  - renvoie NULL en fin de fichier
  - aucun contrôle sur la taille de la ligne lue
- int puts (const char *ligne)
  - envoie une ligne sur la sortie standard
  - en ajoutant \n en fin de ligne
- cf. exemple2.13.c

 2012-10-31

E/S formatées (stdio.h)

44

- printf envoie un texte formaté à la sortie standard : printf(format,arg1,arg2…)
- sprintf envoie un texte formaté dans une chaîne : sprintf(chaine,format,arg1,arg2…)
- format contient un emplacement par argument, spécifié par % suivi d'un code de formatage
  - Caractères spéciaux : \t (tabulation), \n (retour chariot), \", %% (signe de %)

 2012-10-31

E/S formatées

45

- Formats applicables aux entiers
  - %c : affichage comme caractère
  - %h : affichage comme short int décimal
  - %d : affichage comme entier décimal (int)
  - %ld : affichage comme long int décimal
  - %H : affichage comme unsigned short int
  - %u : affichage comme entier décimal non-signé (unsigned int)
  - %lu : affichage comme unsigned long int
  - %o : affichage comme octal
  - %x (resp. %X) : affichage comme hexadécimal : d1 (resp. D1))

E/S formatées

46

- Format applicable aux chaînes de caractères : %s
- Format applicable aux pointeurs : %p
- Formats applicables aux nombres flottants
  - %e (resp. %E) : notation scientifique : 1.87e03
  - %f : notation régulière (float)
  - %g : notation la plus courte des deux
  - %lf : notation régulière (double)
  - %L : notation régulière (long double)

E/S formatées

47

Instructions additionnelles de formatage %cnm<format>

- c = combinaison de diverses options (facultatives)
  - - : cadrage à gauche (défaut = cadrage à droite)
  - + : résultat préfixé par + si nombre positif (pour %d, %i, %e, %f, %g)
  - espace : résultat préfixé par espace si nombre positif (pour %d, %i, %e, %f, %g)
  - # :
    - pour %o, %x, %X : résultat préfixé par 0, 0x ou 0X
    - pour %e, %E, %f, %g : point décimal toujours présent
- n = largeur minimale du champ
  - remplissage par des espaces
  - ou par des 0 si n est précédé de 0 et si cadrage à droite
- m = un . suivi d'un entier
  - pour formats %e, %f, %g : nombre de chiffres après la virgule
  - pour %s : nombre max. de caractères à imprimer

printf()

48

```
char chaine1[] = "zoo";
int decimal = 233;
float flottant = 23.411;

printf("%c %d %o %u %x\n",
   decimal,decimal,decimal,decimal,decimal);
printf("D%sF\n", chaine1);
printf("D%10sF\n", chaine1);
printf("D%-10sF\n", chaine1);
printf("D%10.2sF\n", chaine1);
printf("D%-10.2sF\n", chaine1);
printf("D%.2sF\n", chaine1);
printf("D%dF\n", decimal);
printf("D%10dF\n", decimal);
printf("D%-10dF\n", decimal);
printf("D%010dF\n", decimal);
printf("D%fF\n", flottant);
printf("D%10fF\n", flottant);
printf("D%-10.2fF\n", flottant);
printf("D%010.2fF\n", flottant);
printf("D%.2fF\n", flottant);
```

```
é 233 351 233 e9
DzooF
D      zooF
Dzoo      F
D       zoF
Dzo       F
DzoF
D233F
D      233F
D233      F
D0000000233F
D23.410999F
D 23.410999F
D23.41     F
D0000023.41F
D23.41F
```

E/S formatées (stdio.h)

49

- scanf (format, var1, var2 …)
- Lit l'entrée standard, interprète les données selon le format spécifié, et place leur valeur dans les variables fournies en arguments
- Une forme simple de reconnaissance de patrons (pattern-matching)
- Le processus s'arrête quand les deux chaînes ne correspondent pas
- sscanf (chaîne à analyser, format, var1, var2 …) : identique mais lit la chaîne à analyser

E/S formatées (stdio.h)

50

- Spécification (partielle) du format du patron : %cndt
  - c = * : on ignore le patron reconnu
  - n = nombre de caractères max du patron
  - d = h (short/unsigned short), l (long/unsigned long/double), L (long double)
  - t = c, u, o, x, X, d, i (décimal ou octal ou hexadécimal), f, e, E, g, G, s

scanf() (stdio.h)

```
char chaine1[] = "Bonjour Toto" ;
Char chaine2[] = "x=23+34\ny=280*12.3";
char ch[50], ch1[50], ch2[50];
int i, j;
float k, l;

if (sscanf(chaine1, "Bonjour %s", ch))
        printf("Test1 : %s\n", ch);
if (sscanf(chaine1, "Bonjour%s", ch))
        printf("Test2 : %s\n", ch);
if (sscanf(chaine1, "Bonjour %2s", ch))
        printf("Test3 : %s\n", ch);
if (sscanf(chaine2, "x=%s y=%s", ch, ch1))
        printf("Test4 : %s %s\n", ch, ch1);
if (sscanf(chaine2, "X=%s y=%s", ch, ch1))
        printf("Test5 : %s %s\n", ch, ch1);
if (sscanf(chaine2, "x=%d+%d y=%f*%f", &i, &j, &k, &l))
        printf("Test6 : %d %d %f %f\n", i, j, k, l);
if (sscanf(chaine2, "x=%o+%o", &i, &j))
        printf("Test7 : %d %d\n", i, j);
if (sscanf(chaine2, "x=%*d+%d",&i))
        printf("Test8 : %d \n", i);
```

Test1 : Toto
Test2 : Toto
Test3 : To
Test4 : 23+34 280*12.3
Test6 : 23 34 280.000 12.300
Test7 : 19 28
Test8 : 34

printf() et scanf()

- Comportement aberrant (pouvant aller au *segmentation fault* à l'exécution)
  - Si nombre d'emplacements dans le patron < nombre d'arguments restants
  - Si nombre d'emplacements dans le patron > nombre d'arguments restants
  - Si spec. de l'emplacement incompatible avec argument

Fonctions de manipulation de chaînes de caractères

53

- Définies dans <string.h>
  - Longueur d'une chaîne (excluant \0)

```
int strlen(const char *s)
```

  - Comparaison de chaînes : résultat négatif si s1<s2, 0 si s1==s2, positif si s1 > s2

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

  - Différence entre s1 == s2 et strcmp(s1, s2) ?

Fonctions de manipulation de chaînes de caractères

54

- Définies dans <string.h>
  - Concaténation de chaînes : orig concaténée à dest

```
char *strcat(char *dest, const char *orig)
char *strncat(char *dest, const char *orig, int n)
```

  - Copie de chaînes : orig copiée dans dest

```
char *strcpy(char *dest, const char *orig)
char *strncpy(char *dest, const char *orig, int n)
```

  - Les 4 fonctions retournent l'adresse de destination
  - les versions n copient/concatènent au plus n caractères
- Différence entre strcpy(p1,p2) et p1=p2 ?

Fonctions de manipulation de chaînes de caractères

55

- Exemple

```
char chaine1[] = "toto", chaine2[20] = "titi", chaine3[20];
printf("%s %s %s\n",chaine1, chaine2, chaine3);
strcat(chaine2,chaine1);
printf("%s %s %s\n",chaine1, chaine2, chaine3);
strcpy(chaine3,chaine1);
printf("%s %s %s\n",chaine1, chaine2, chaine3);
strncpy(chaine3,chaine2,2);
printf("%s %s %s\n",chaine1, chaine2, chaine3);
```

- Affiche

```
toto titi
toto tititoto
toto tititoto toto
toto tititoto tito
```


Fonctions de manipulation de chaînes de caractères

56

- Définies dans <string.h>
  - Recherche d'un caractère dans une chaîne
    - 1ère occurrence du caractère

```
char *strchr(char *cs, char c)
```

    - dernière occurrence du caractère

```
char *strrchr(char *cs, char c)
```

  - Les 2 fonctions retournent un pointeur vers l'occurrence cherchée du caractère, ou NULL si non trouvé

Fonctions de manipulation de chaînes de caractères

57

- Exemple

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

int main(void){
  char a[]= "tototi tititutu", *occurrence;

  for (occurrence = strchr(a, 't'); occurrence != NULL; occurrence =
   strchr(++occurrence, 't'))
        printf("t est trouve en position %d dans %s\n", occurrence - a,
   a);
}
```

- Affiche

```
t est trouve en position 0 dans tototi tititutu
t est trouve en position 2 dans tototi tititutu
t est trouve en position 4 dans tototi tititutu
t est trouve en position 6 dans tototi tititutu
t est trouve en position 8 dans tototi tititutu
t est trouve en position 10 dans tototi tititutu
```


Fonctions de manipulation de caractères

58

- Définies dans <ctype.h>
  - Tests de classe de caractère

```
int isalpha(int c)
int isdigit(int c)
int isupper(int c)
int islower(int c)
int isalnum(int c) …
```

    - c doit avoir une valeur égale à EOF ou représentable comme unsigned char
    - renvoie 0 si faux, autre chose que 0 sinon
  - Conversion de caractères

```
int tolower(int c)
int toupper(int c)
```

Plan

59

- Tableaux
- Pointeurs
- Fonctions
- Entrées-Sorties
- Exercices

Exercices

60

- Utiliser getchar directement pour écrire une fonction getline(char *ligne, int taille) qui renvoie la prochaine ligne de l'entrée standard
  - en limitant à taille -1 le nombre de caractères lus (les caractères excédentaires se retrouvent sur la ligne suivante)
  - en terminant chaque ligne par \0
  - et en supprimant les \n
- L'intégrer à un programme qui affiche l'entrée standard sur 30 colonnes (29 caractères affichés) (exemple2.14.c)