Programmer en C

S2 - dept info

18 août 2021

Table des matières

1 De C++ à C

Quand on regarde superficiellement un programme C, on trouve que la syntaxe est identique à celle de C++. C'est normal, historiquement C++ a été conçu [1] par Bjarne Stroustrup des Bell Labs, dans les années 1983-1985, comme une extension du langage C, pour y intégrer les classes et la programmation objet. Au départ, C++ s'appelait « C with classes ».

Le langage C a été conçu 10 ans plus tôt (1971-1972) par Dennis Ritchie des Bell Labs, qui travaillait avec Ken Thompson sur la conception d'un nouveau système d'exploitation appelé UNIX.

D'où deux nouvelles :

— la bonne : vous ne serez pas dépaysé par la syntaxe [2] ;
— la mauvaise : il va falloir apprendre à se passer de mécanismes que vous connaissez en C++, mais qui n'existent pas en C.

1.1 Ce qui n'existe pas en C

Citons par exemple

— le passage de paramètres par référence,
— la surcharge,
— les chaînes de caractères,
— les opérations `<<` et `>>` pour lire et écrire sur des *streams*,
— new et delete
— les classes, l'héritage,
— le transtypage `dynamic_cast`, `static_cast`), et le polymorphisme etc.
— les templates,
— et plein d'autres choses.

1.2 Idées directrices du langage C

Source `http://fr.wikipedia.org/wiki/Unix`

En 1969, K. Thomson a commencé à programmer le système UNIX en langage d'assemblage, spécifique au mini-ordinateur PDP-7 sur lequel il travaillait.

Voici un extrait du code d'UNIX (commande d'administration `dsw`, qui servait à détruire des fichiers indiqués par les interrupteurs de la console) [3]

```
" dsw

    lac djmp
    dac .-1
    oas cla
    cma
    tad d1
    dac t1
    sys open; dd; 0
1:
    lac d2
    sys read; dir; 8
    sna
    sys exit
    lac dir
    sna
    jmp 1b
    isz t1
    jmp 1b

wr:
    lac d1
    sys write; dir+1; 4
    lac d1
    sys write; o12; 1
    sys save
do:
    sys unlink; dir+1
    sys exit

d1: 1
d2: 2
o12: 012
t1: 0
djmp: jmp do
dd: 056056; 040040; 040040; 040040
dir: .=.+8
```

Évidemment, ce genre de programme permet d'utiliser au mieux les possibilités de la machine, mais il est peu maintenable, et n'est pas *portable* : pour l'utiliser sur une autre machine que le PDP-7, il faut le réécrire complètement.

1. Voir document `http://www.hitmill.com/programming/cpp/cppHistory.html`
2. qui a aussi fortement imprégné celles de Java, C#, PHP, Javascript, Python, Perl etc.)
3. `http://www.informatica.co.cr/unix-source-code/research/pups-mail/eml.1208.html`

En 1971, K. Thomson décide de réécrire UNIX pour le rendre plus facilement maintenable, et portable sur d'autres machines. Il pense pour cela d'abord aux langages TMG et FORTRAN, mais TMG est aussi spécifique au PDP-7, et FORTRAN n'est pas vraiment adapté. Il décide, avec l'aide de D. Ritchie, de définir un nouveau langage B, inspiré de BCPL qu'il utilisait sur son projet précédent (MULTICS).

Le langage B est non typé : toutes les données sont des *mots*, de même taille, pouvant contenir indifféremment des entiers, des adresses etc.

Cette absence de typage s'est révélé ennuyeuse sur le PDP-11, parce qu'elle ne permettait pas de travailler facilement sur des caractères (octets). D. Ritchie a alors commencé à modifier le langage B pour y introduire des types, c'est "new B", qui devient "C".

Ce nouveau langage permet toujours de travailler à bas niveau : les types de données de base correspondent de façon triviale à ceux qui sont disponibles sur la machine (un entier ou un pointeur = un mot, un caractère = un octet, etc) et les instructions C se traduisent facilement en instructions élémentaires, et inversement. Par exemple, l'auto-incrémentation (opérateur "++") était disponible dans le jeu d'instructions de beaucoup de machines de l'époque.

Voici par exemple la commande `cp`, telle que la trouve dans les sources UNIX de juin 1972[4]

```
1  main(argc,argv)
2  char **argv;
3  {
4  char buf[512];
5  int fold, fnew, n;
6  char *p1, *p2, *bp;
7  int mode;
8          if(argc != 3) {
9                  write(1,"Usage: cp oldfile newfile\n",26);
10                 exit();
11         }
12         if((fold = open(argv[1],0)) < 0){
13                 write(1,"Cannot open old file.\n",22);
14                 exit();
15         }
16         fstat(fold,buf);
17         mode = buf[2] & 037;
18         if((fnew = creat(argv[2],mode)) < 0){
19                 stat(argv[2], buf);
20                 if((buf[3] & 0100) != 0){
21                         p1 = argv[1] - 1;
22                         p2 = argv[2] - 1;
23                         bp = buf - 1;
24                         while(*++bp = *++p2);
25                         *bp = '/';
26                         p2 = bp;
27                         while(*++bp = *++p1)
28                                 if(*bp == '/')
29                                         bp = p2;
30                         if((fnew = creat(buf,mode)) < 0){
31                                 write(1,"Cannot creat new file.\n",23);
32                                 exit();
33                         }
34                 }else{
35                 write(1,"Cannot creat new file.\n",23);
36                 exit();
37                 }
38         }
39         while(n = read(fold, buf, 512))
40         if(n < 0){
41                 write(1,"Read error\n",11);
```

4. http://code.google.com/p/unix-jun72/source/browse/trunk/src/cmd/cp.c

```
42                      exit();
43              }else
44                      if(write(fnew,buf,n) != n){
45                              write(1,"Write error.\n",13);
46                              exit();
47                      }
48              fstat(fnew,buf);
49              exit();
50      }
```

La réécriture du noyau UNIX dans ce nouveau langage C est achevée à l'été 1973.

1.3 Caractéristiques du langage C

En résumé, le langage C a été conçu comme un langage de programmation

- **de bas niveau**, pour écrire des programmes efficaces qui utilisent au mieux les possibilités du matériel ;
- **simple**, pour pouvoir écrire des compilateurs efficaces. Par exemple les opérations d'entrée-sortie, bien qu'indispensables, ne font pas partie du langage C : ce sont des fonctions définies dans une bibliothèque ;
- **portable**, pour écrire des programmes qui tournent sur des machines de types différents ;
- **de programmation structurée**, avec des notions de blocs, de boucles (`for`, `while`), d'alternatives (`if`, `switch`), comme Algol et Pascal, plutôt que de branchements et d'étiquettes (comme Fortran IV et l'assembleur), afin de faciliter l'écriture et la maintenance ;
- **concis**, pour accélérer le codage. Utilisation de symboles à la place de mots-clés (il n'y en a que 32, contre 50 en Java, 77 en C# et 357 en Cobol).

La concision a souvent été reprochée au langage C, parce que les programmeurs débutants ont du mal à décoder les formulations parfois cryptiques qu'ils rencontrent dans les programmes.

Par exemple la séquence (lignes 22 à 24)

```
p2 = argv[2] - 1;
bp = buf - 1;
while(*++bp = *++p2);
```

a pour effet de copier la chaîne de caractères donnée en second paramètre (y compris le caractère nul qui la termine) dans dans le tampon `buf`, en laissant à la fin `bp` pointer sur le caractère nul. [5]

En réalité, il se dégage assez rapidement des *tournures idiomatiques* pour réaliser les opérations courantes : au bout de quelques jours de pratique n'importe quel programmeur débutant, en C ou C++, sait reconnaître d'un simple coup d'oeil des boucles comme

```
for (i=0; i<10; i++) {
   ...
}
```

sans avoir besoin de réfléchir aux détails de fonctionnement de la boucle `for`.

1.4 Évolution du langage

Depuis 1972, le langage C a connu quelques évolutions, qui vont globalement vers

- une *standardisation des types* : le langage C ne précisait par exemple pas la taille des entiers utilisés, ce qui posait des problèmes de portabilité (dans B, il était supposé qu'un mot était assez grand pour contenir un entier ou un pointeur) ; ajout de nouveaux types (complexes, booléens).
- un *renforcement des vérifications* de types (dans la philosophie initiale, le programmeur était supposé "assez grand pour savoir ce qu'il fait et le faire correctement", ce qui n'a pas manqué de causer quelques soucis), adjonction de `const` pour interdire des modifications malencontreuses.
- des ajouts de facilités parfois déjà implémentées dans les compilateurs : tableaux de taille variable,
- un rapprochement de la syntaxe avec C++ : commentaires commençant par `//`, possibilité de déclarer des variables ailleurs qu'en début de bloc.

5. Sur cet exemple on voit que l'auteur a mis à profit sa connaissance des opérations du PDP-7, pour essayer de convaincre le compilateur C d'utiliser les instructions avec pré-incrémentation, quitte à "reculer" les pointeurs au départ pour y arriver.

La normalisation :

— 1978, livre *The C Programming Language* de Brian Kernighan et Dennis Ritchie. Sert de référence pour ce qu'on appelle C "K&R" :
— En 1989, norme dite ANSI C ou C89 (officiellement ANSI X3.159-1989) par l'ANSI (american national standard organization). Norme adoptée aussi par l'Organisation internationale de normalisation (C ISO, ISO/CEI 9899 :1990).
— C95 de l'ISO, apporte quelques correctifs.
— C99 (formellement ISO/CEI 9899 :1999), amène en particulier les tableaux dynamiques et facilite les calculs numériques intensifs ;
— C11, alias ISO/IEC 9899 :2011
— C17 est la dernière version à ce jour.
— la prochaine version, nom de code C2X, devrait être adoptée fin 2021.

Dans ce cours les exemples seront compilés avec `gcc -std=c17` ou `-std=gnu17` pour utiliser quelques spécificités.

2 Premiers exemples

2.1 Hello, world !

Le langage C a inauguré une tradition qui se perpétue de nos jours : toute présentation d'un langage de programmation commence par un exemple qui fait écrire un message de bienvenue.

Pour gagner du temps, nous passons de suite à une version plus sophistiquée, puisqu'elle intègre des opérations d'entrées-sorties, et des calculs :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6    char nom[100];
7    int  annee;
8
9    printf ("Quel est votre nom ? ");
10   scanf("%s", nom);
11   printf ("Et votre annee de naissance ? ");
12   scanf("%d", &annee);
13   printf ("Bonjour %s, vous avez %d ans !\n", nom, 2013-annee);
14
15   return EXIT_SUCCESS;
16 }
```

Explications :

1. Comme en C++, l'exécution d'un programme commence par l'appel de sa fonction `main()`.
   Les arguments de `main()` servent à récupérer les paramètres qui figurent sur la ligne de commande qui lance le programme. Ce sont
   — le nombre d'arguments (arg. count),
   — un tableau de chaînes de caractères (le nom du programme et les paramètres, arg. values) ;
   nous en verrons l'usage plus loin.
2. Ce programme fait des écritures sur l'écran (appels de la fonction `printf()`) et des lectures au clavier (`scanf()`).
3. Les deux premiers appels à `printf` sont faciles à comprendre, le dernier vous permet de voir que le premier paramètre est un *chaîne de format*, un "modèle à trous" avec des spécifications qui indique où doivent apparaître les paramètres suivants : la `%s` correspond à la chaîne `nom` et `%d` à la valeur décimale de `2013-annee`.
4. On retrouve ces formats pour l'instruction `scanf` qui sert à la lecture.

5. Pour la lecture de l'année de naissance, vous remarquez certainement que le paramètre était "`&annee`" et non "`annee`" : la fonction `scanf()` prend comme paramètre l'`adresse` des données qui doivent être saisies.

   C'est parce qu'en C,

   > le passage de paramètre se fait par valeur

   il n'y a pas de passage de paramètres par référence. Pour transmettre une donnée qu'une fonction doit modifier, on passe explicitement en paramètre *son adresse*. Pour saisir quelque chose dans une variable entière (ou réelle), on précède donc le nom de la variable par l'opérateur `&`.

6. Les chaînes de caractères étant des tableaux, on passe simplement le nom du tableau. En effet, les déclarations

   ```
   char nom[100];
   int  annee;
   ```

   servent à réserver de l'emplacement pour deux données en mémoire : l'une pour stocker 100 caractères, l'autre pour un entier. l'identificateur `nom` représente la position de la première zone, c'est déjà une adresse, alors que l'identificateur `annee` désigne la seconde donnée. Son adresse s'obtient en lui appliquant l'opérateur "`&`".

2.2 Utilisation des paramètres de la ligne de commande

Exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      printf("Ceci␣est␣le␣programme␣%s␣avec␣%d␣arguments.\n",
7             argv[0], argc);
8      for (int i = 0; i < argc; i++) {
9          printf("argv[%d]␣=␣%s\n", i, argv[i]);
10     }
11     return EXIT_SUCCESS;
12 }
```

Voici ce que donne une exécution

```
$ ./salutations bienvenue chez les programmeurs C
Ceci est le programme ./salutations avec 6 arguments.
argv[0] = ./salutations
argv[1] = bienvenue
argv[2] = chez
argv[3] = les
argv[4] = programmeurs
argv[5] = C
```

3 La fonction `printf()`

La fonction `printf()` sert, comme le nom l'indique, à afficher des données selon un certain **f***ormat*. Exemple

```
printf ("Bonjour %s, vous avez %d ans !\n", nom, 2013- annee) ;
```

Le format, premier argument, est un modèle de ce qu'il faut afficher, avec des "trous" pour y faire apparaître des valeurs en suivant une *spécification*. On trouve ici

— `%d` pour faire afficher un nombre entier en décimal,

— `%s` pour faire afficher une chaîne.

Les spécificateurs (d, s, etc.) doivent correspondre au type des paramètres. Afficher une chaîne avec `%d` fait apparaître son adresse, pas son contenu.

Autres spécificateurs de conversion : (parmi les plus courants)

- `%x` et `%o` pour faire afficher un nombre entier en hexadécimal ou en octal
- `%u` pour un nombre non signé,
- `%c` pour faire afficher le caractère correspondant à un nombre (par exemple, le caractère A pour la valeur 65).
- `%f` pour un nombre en virgule flottante (`float` ou `double`)
- `%p` pour un pointeur.

Le spécificateur peut être précédé d'indicateurs et de champs pour préciser la présentation voulue.

Exemples :

- `%4d` demande l'affichage d'un nombre sur au moins 4 caractères. La valeur 42 sera donc précédée de deux espaces.
- idem pour `%04d` qui affiche un nombre sur 4 chiffres, mais avec des zéros en tête.
- `%10s` affiche une chaîne cadrée à droite sur 10 caractères, en remplissant éventuellement les premiers caractères par des espaces.
- `%-10s` fait un cadrage à droite
- `%10.3f` affiche un nombre réel sur 10 caractères dont 3 après la virgule.

Pour les conversions d'entiers (spécifications `d`, `u`,`x`, ...) la longueur de la donnée est précisée par

- `hh` pour les `char` (signés ou pas),
- `h` pour les `short int`,
- `l` pour les `long int`,
- `ll` pour les `long long int`
- `z` pour les `size_t`

Enfin, pour les caractères larges (`wchar_t`) on emploie `%lc`, et `%ls`, pour les chaînes de caractères larges, ce qui provoquera leur conversion en chaînes de caractères "multibyte" (UTF-8).

Documentation : voir `man 3 printf` pour plus de détails sur les spécifications.

Exercice : écrire un programme qui affiche la table de multiplication

```
      1  2  3  4  5   .... 9
     -- -- -- -- --       --
 1 :  1  2  3  4  5        9
 2 :  2  4  6  8 10       18
      .....
 9 :  9 18 27  .......    81
```

4 Passage de paramètres

En C, le passage de paramètres se fait par valeur uniquement. Depuis C89, on peut préciser (`const` qu'un paramètre ne doit pas être modifié dans la fonction.

Étudions l'exemple suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int somme(const int x, const int y)
5  {
6    return x+y;
7  }
8
9  int main()
10 {
11   int a, b;
12   printf ("Donnez deux nombres : ");
13   scanf("%d %d", &a, &b);
14   printf("la somme %d + %d vaut %d\n", a, b, somme(a,b));
```

```
15    return EXIT_SUCCESS;
16  }
```

La définition de la fonction `somme()` ne devrait pas vous causer de surprise particulière, pas plus que son appel depuis le `printf()` du `main()`.

Ce qui est plus intéressant, c'est la fonction `scanf()`, qui sert à la lecture des données `a` et `b`. Globalement, cette fonction reçoit une chaîne de format de lecture (similaire à celle de `printf`), et une indication des variables à remplir.

Mais si on écrivait `scanf("...", a,b);` ce qu'on transmettrait à `scanf`, c'est les *valeurs* contenues dans `a` et `b` à ce moment-là, parce que `scanf`, comme toutes les fonctions C, utilise un passage par valeur.

On transmet donc les *adresses* des variables, qui se notent `&a` et `&b`, à la fonction `scanf()`.

5 Adresses et pointeurs

Les **pointeurs** sont des variables susceptibles de contenir des adresses. On les déclare en précédant leur nom par une étoile.

Par exemple dans

```
int a, *pa;
char message[10], *ptr;
```

on voit la déclaration de deux pointeurs, l'un vers des entiers, l'autre vers des caractères.

On peut leur affecter un contenu, qui sera respectivement l'adresse d'un entier, et celle d'un caractère

```
pa = & a;
ptr = & ( message[3] ); // ou message+3, voir plus loin
```

Il existe aussi une valeur spéciale, `NULL`, qui peut être affectée à n'importe quel pointeur.

On pourrait donc utiliser des pointeurs dans `scanf` :

```
int pa = &a,  pb = &b;
...
scanf("%d %d", pa, pb);
```

Mais ici c'est plus simple de faire directement :

```
scanf("%d %d", &pa, &pb);
```

Déréférencement : Quand un pointeur contient l'adresse d'une donnée, on peut le *déréférencer*, c'est-à-dire accéder à la donnée pointée par le pointeur. Le déréférencement, ou indirection, se note par une étoile.

Exercice : dans l'exemple ci-dessous, étudiez l'utilisation des pointeurs comme paramètres :

Exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void saisirDeuxNombres(int *px, int *py)
5  {
6    printf("Donnez deux nombres : ");
7    scanf("%d %d", px, py);
8  }
9
10 void calculerSomme(const int x, const int y, int *pz)
11 {
12   *pz = x+y;
13 }
14
```

```
15 void afficherSomme(const int x, const int y, const int z)
16 {
17   printf("la somme %d + %d vaut %d\n", x, y, z);
18 }
19
20 int main()
21 {
22   int a, b, c;
23   saisirDeuxNombres( &a, &b    );
24   calculerSomme    ( a,  b, &c);
25   afficherSomme    ( a,  b,  c);
26   return EXIT_SUCCESS;
27 }
```

Quand une action doit modifier un de ses paramètres, cette action se traduit par une fonction C qui reçoit comme paramètre l'adresse de la donnée.

Exercice : écrire un programme C qui
- demande trois nombres entiers,
- appelle une fonction qui ordonne ces trois nombres,
- les affiche dans l'ordre (du plus petit au plus grand).

6 Les chaînes des caractères

En C il n'y a pas de chaînes de caractères au sens des `string` de C++, qui sont des objets qui peuvent contenir un nombre indéfini de caractères, et qui possèdent des méthodes pour réaliser diverses opérations.

En C les chaînes sont des suites de caractères consécutifs en mémoire, dont la fin est marquée par un caractère nul `'\0'`. Elles sont désignées par l'adresse de leur premier caractère.

Une chaîne est en fait un tableau : la notation `"abc"` correspond à une suite de 4 octets : 'a', 'b', 'c' et le caractère nul '\0' qui marque la fin.

6.1 Opérations sur les chaînes :

Dans la mesure où ce qu'on appelle chaîne est simplement l'adresse de son premier octet, on peut accéder individuellement à ses caractères par un simple indiçage.

```
char message [] = "Hello, world !";

for (int i=0; message[i] != '\0'; i++) {
  printf("_%c_", message[i]);
}
```

Et on peut aussi la parcourir par des pointeurs ;

```
char message [] = "Hello, world !";
for (char *p = message;  *p != '\0'; p++) {
  printf("_%c_", *p);
}
```

Les autres opérations, comme par exemple évaluer la longueur d'une chaîne ou déterminer si elle contient un caractère donnée, comparer deux chaînes etc. se font par l'intermédiaire de fonctions de bibliothèque.

Quelques-unes parmi les plus importantes :

```
#include <string.h>

size_t strlen(const char *s);

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

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

1. `strlen` retourne la longueur d'une chaîne, est calculée par une boucle de parcours. Le type de retour est `size_t` ce qui correspond à des entiers non signés.
2. `strcpy` copie les octets de `src`, jusqu'au caractère nul final (compris), dans `dst`.
3. `strcat` détermine la position du caractère nul de `c`, et copie à partir de là les octets de `src`, jusqu'au caractère nul final (compris).
4. `strcmp` compare les chaînes `s1` et `s1`, et renvoie un nombre négatif si `s1` précède `s2` dans l'ordre lexicographique, 0 si les chaînes sont égales, et détermine la position du caractère nul de `c`, et un nombre positif si `s1` est après `s2`.

Il faut bien noter que les fonctions `strcpy` et `strcat` peuvent causer des violations d'accès mémoire si le programmeur a réservé des tableaux trop petits pour y loger le résultat : C est un langage de bas niveau, et la responsabilité des "détails techniques" incombe au programmeur.

6.2 Opérations sur les caractères

On a souvent besoin de savoir à quel genre de caractère on a affaire : alphabétique, numérique, minuscule, etc. Il existe de nombreuses fonctions définies par l'entête `ctype.h`

```
#include <ctype.h>

int isalpha(int c);
int isdigit(int c);
int isalnum(int c);
int isspace(int c);

int islower(int c);
int isupper(int c);
 ...
```

Remarquez que
— elles prennent comme caractère un `int`, et non un `char`, mais les deux sont compatibles : un caractère est un petit entier.
— elles renvoient un `int` et non un `bool`, type qui n'existait pas dans les débuts de C (mais les booléens sont aussi de petits entiers).

7 Lecture et écriture dans des fichiers textes

Pour lire et écrire dans des fichiers textes autres que l'entrée et la sortie standard, on dispose de fonctions `fscanf` et `fprintf` analogues aux fonctions vues plus haut.

Elles prennent comme premier paramètre un "`FILE *`" obtenu par la fonction `fopen()` et comme second paramètre un format.

Exemple

```
1  /*
2    Lecture d'un fichier de notes
3    Affichage du nombre de notes, de la somme, et de la moyenne.
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  int main()
10 {
11     float somme  = 0.0;
12     int   nombre = 0;
```

```
13      float note;
14
15      FILE *fnotes = fopen("notes.txt", "r");
16
17      while ( fscanf(fnotes, "%f", &note) == 1) {
18          somme += note;
19          nombre += 1;
20      }
21
22      fclose(fnotes);
23
24      printf("%d notes lues, total=%.2f, moyenne=%05.2f\n",
25              nombre, somme, somme/nombre);
26
27      return EXIT_SUCCESS;
28  }
```

Remarques :

- le second paramètre de `fopen()` est une chaîne qui indique le mode d'ouverture du fichier : ici en lecture, ce serait `"w"` en écriture ;
- la boucle est répétée tant que le `fscanf()` réussit à lire un nombre sur `fnotes` ;
- `fclose()` fait évidemment le pendant à `fopen()` ;
- l'affichage de la somme se fait avec 2 chiffres après la virgule (en fait un point décimal) celui de la moyenne sur 5 chiffres, dont 2 après la virgule, avec éventuellement un ou plusieurs zéros non significatifs en tête.

Exemple d'exécution :

```
4 notes lues, total=50.00, moyenne=12.50
```

Un peu plus : les fonctions `scanf(...)` et `printf(...)` sont en fait des appels à `fscanf(stdin,....)` et `fprintf(stdout,....)`, les variables prédéfinies `FILE *stdin, *stdout *`" correspondent aux flots d'entrée et de sortie standards [6].

Il faut aussi signaler les deux fonctions

```
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

qui servent à lire respectivement un caractère ou une ligne de texte.

La fonction `fgets` nécessite quelques commentaires :

- elle prend comme second paramètre le nombre maximum de caractères qu'on peut mettre dans le tampon `s`, ce qui assure une protection contre les lignes trop longues (débordement de buffer)
- si le tampon est assez grand il reçoit la ligne, le retour-chariot de fin de ligne, ainsi que le caractère nul de fin de chaîne.

Last but not least, `int feof(FILE *stream);` teste l'indicateur de fin d'un fichier.

Exercice : écrire un programme qui fait afficher, ligne par ligne, un fichier texte dont le nom est passé en paramètre. Les lignes apparaîtront numérotés sur 4 chiffres, en commençant par 0001.

Prévoir le cas des lignes trop longues qui apparaîtront sans numérotation.

```
0001 ceci est une premiere ligne
0002 ceci est une seconde ligne qui e
     st trop longue pour apparaitre e
     n une seule fois
0003 ceci est la troisieme ligne
....
```

6. la sortie d'erreurs est `stderr`

8 Chaînes UNICODE et caractères larges

Si vous testez le programme précédent avec des textes accentués, vous risquez d'avoir quelques surprises.

Vous travaillez en effet dans un environnement moderne qui permet l'emploi simultané de jeux de caractères multiples : alphabet latin, grec, arabe, idéogrammes, etc. Tous ces jeux de caractères sont intégrés dans le standard UNICODE, dans lequel les *caractères larges* sont codés sur 32 bits (et non sur 8 bits comme dans le code ISO-LATIN). Cependant les caractères UNICODE sont transmis, dans le codage UTF-8, sous forme d'un nombre variable d'octets : 1 octet pour ceux qui correspondent à l'ancien code ASCII 7 bits, 2 octets pour les caractères accentués des langues dérivées du latin, du grec, etc, et jusqu'à 4 pour les idéogrammes.

Exercice : pour vous en convaincre, écrivez un programme qui
- lit une chaîne par `fgets()`
- calcule et affiche sa longueur (déduire le caractère de fin de ligne)
- fait afficher la chaîne caractère par caractère, avec des espaces entre deux caractères consécutifs (`hello -> h e l l o`).

et essayez-le avec des chaînes accentuées ou pas.

Conclusion : La représentation de chaînes de caractères par des tableaux d'octets fonctionnait correctement dans des environnements où les caractères étaient codés sur un octet (ANSI, ISO-LATIN), mais ce n'est plus le cas aujourd'hui.

C'est pour cela qu'ont été introduits en C les *caractères larges* (type `wchar_t`) et les fonctions associées.

A développer

Voici un exemple de programme utilisant les caractères larges

```
1  #include <stdio.h>
2  #include <wchar.h>
3  #include <locale.h>
4
5  int main(void)
6  {
7          wchar_t prenom[20];
8
9          setlocale(LC_ALL,"");
10         wprintf(L"Ton prénom ?");
11         wprintf(L"prenom = %ls\n", prenom);
12         for (int i=0; i<wcslen(prenom); i++) {
13                 wprintf(L"_%lc", prenom[i]);
14         }
15         wprintf(L"_\n");
16         return 0;
17 }
```

9 Structures

Les classes de C++ ont été élaborées à partir du concept de *structure* présent dans C.

Une structure est une simple collection de champs. Pas de méthodes, pas d'héritage, pas d'attributs de visibilité.

Pour déclarer des types, on écrit

```
struct Date {
  int jour, mois, annee;
};

struct Personne {
  char nom[64];
  char prenom[64];
```

```
  struct Date naissance;
};

struct CarteGrise {
  char plaque[20];
  char modele[20];
  int annee;
  struct Personne * titulaire;
};
```

Remarquez que
— on utilise les noms de types en les précédant par le mot-clé `struct`.[7]
— la structure`Personne` *contient* une `Date`, alors que
— la structure`CarteGrise` *fait référence* à une `Personne` : elle contient un pointeur vers une `Personne`.
L'utilisation des structures ne pose pas de difficulté :

```
void saisir_date(struct Date *pDate)
{
  printf("Jour mois annee ? ");
  scanf("%d %d %d", pDate->jour, pDate->mois, pDate->annee);
}

void saisir_personne(struct Personne *pPersonne)
{
  printf("Nom :") ;
  .....
  printf("Date de Naissance, ");
  saisir_date(& (pPersonne->naissance));
}
...
int main(void)
{
  struct Personne conducteur;
  saisie_Personne(& conducteur);
  ....
}
```

9.1 Initialisation des structures

Une structure peut être initialisée, au moment de sa déclaration, par la liste des valeurs de ses champs, dans l'ordre, et entourée d'accolades.

```
struct Date prise_bastille = { 14, 7, 1789 };
```

Il est aussi possible[8] de désigner les champs à initialiser

```
struct Date noel = { .mois = 12, .jour = 25 }
```

Les champs manquants sont initialisés à 0.

9.2 Affectation, passage de paramètres

Les structures sont des données qui peuvent être contenues dans des variables, on peut les affecter

```
struct Date anniversaire ;

anniversaire = noel;
```

et elles sont passées en paramètres par valeur.

7. On verra plus loin (`typedef`) comment s'affranchir de cette contrainte.
8. mais pas en C++

```
void afficher_date(struct Date d)
{
  printf("%d/%d/%d", d.jour, d.mois, d.annee);
}
```

On peut également les comparer par `==` et `!=`.

10 Union

Soit un jeu, piloté par des évènements provenant du clavier ou de la souris

```
struct EvenementClavier {
   int type;              // type = 1
   char touche;
};

struct EvenementSouris {
   int type;               // type = 2
   int x, y;               // coordonnées
   int etat;               // état des boutons
};
```

L'union permet de faire un type général, assez grand pour contenir l'un ou l'autre de ses membres :

```
union Instruction {
   int type;
   struct EvenementClavier ec;
   struct EvenementSouris  es;
};
```

Ici,
— soit un entier appelé `type`,
— soit une structure `EvenementClavier` appelée `ec`, avec des champs `ec.type` et `ec.touche`,
— soit une structure `EvenementSouris` appelée `es`, avec des champs `ec.type`, `ec.touche`, etc.

Une *union* est donc une donnée en mémoire, de taille suffisamment grande pour stocker le plus grand de ses membres. Ici, probablement l'EvènementSouris. Chaque membre constitue une façon de voir le contenu de cette zone de données.

Notez qu'on a bien pris soin que le champ type se retrouve, au même endroit, dans les 3 membres ;

```
void traiterEvenement(union Evenement e)
{
   switch (e.type) {
   case  1 :  // evenement souris
            if (e.ec.touche == '\e') // escape
                SuspendrePartie();
            else if (e.ec.touche == '\n')
                TraiterSaisie();
            else
                AjouterCaractere(e.ec.touche);
            break;
   case 2 :  // evenement clavier
            DeplacerPointeur(e.es.x, e.es.y);
            ....
            break;
   ...
  }
}
```

L'initialisation d'une union peut se faire selon la même syntaxe que pour les structures, avec laquelle elle se combine

```
union Evenement e = {
                    .ec = { .type = 1,
                            .touche = '@';
                    };
```

11 Définition de noms de types : typedef

Le mot-clé `typedef` permet de définir des noms de types, au lieu de définit des noms de variables.
Exemple : les lignes

```
struct Point {
   int x, y
}  p, *pp;
```

déclarent à la fois
- un type nommé `struct Point`,
- une variable `p` de type `struct Point`,
- une variable `pp` de type `struct Point *`.

Si on écrit

```
typedef struct Point {
   int x, y
}  POINT, *PPOINT;
```

on définit deux noms de types
- `POINT` qui est synonyme de `struct Point`,
- `PPOINT` synonyme de type `struct Point *`.

ce qui permet de raccourcir ensuite les écritures : au lieu de

```
void tracerTrait( struct Point A, struct Point B)  {
   ....
}
```

on pourra écrire

```
void tracerTrait(POINT A, POINT B)  {
   ....
}
```

Ne pas en abuser. L'utilisation de `typedef` peut améliorer la lisibilité, mais aussi compliquer la compréhension, et il y a souvent des abus.

Par exemple, si on a défini le type `Evenement` comme une union d'`EvenementClavier`, `EvenementSouris`, etc. il est important de savoir, quand on relit le programme, que c'est une union et non pas une structure. Dans ce cas, utiliser un `typedef` cache de l'information qui est pourtant indispensable.

De même, dans l'exemple ci-dessus, il est assez probable qu'un aura besoin dans le code de dé-référencer les `PPOINT` : il est donc important de savoir que ce sont des pointeurs. Dans ce cas il serait plus raisonnable de déclarer un pointeur `pp` comme

```
POINT * pp;
```

plutôt que comme

```
PPOINT  pp;
```

Il y a malheureusement beaucoup d'abus. Le document `http://www.kernel.org/doc/Documentation/CodingStyle` (chapitre 5) définit une liste limitative d'usages raisonnables, dont en particulier
- Pour les *types opaques*, quand on veut absolument cacher les détails d'implémentation du type. Par exemple vous survivez très bien sans savoir que qu'est en réalité un `FILE`, du moment que vous pouvez l'employer à travers les fonctions prévues pour : `fopen`, `fscanf`, etc.
- Pour les types entiers, quand l'abstraction aide à éviter les confusions entre `int` et `long`. Exemple

```
typedef unsigned long EntierPositif;
```

Un bon principe : 'In general, a pointer, or a struct that has elements that can reasonably be directly accessed should *never* be a typedef."

12 Préprocesseur : usage des macros

Les directives du préprocesseur sont les lignes de votre programme C qui commencent par un dièse. Ces lignes sont traités avant la compilation proprement dite.

12.1 `#include`

En C++, vous avez déjà rencontré ce type de directive, qui sert à inclure le contenu d'un autre fichier. Il y a deux formes :

```
#include <nomdefichier>
#include "nomdefichier"
```

Les deux formes diffèrent par l'emplacement des fichiers ainsi inclus : avec la première forme ce sont des fichiers d'entête de bibliothèques système[9], avec la seconde des fichiers "personnels", généralement situés dans le même répertoire que le source. Le paramètre `-I` de `gcc` permet de citer des répertoires supplémentaires où le compilateur ira chercher les fichiers à inclure.

12.2 #define

Cette directive est généralement utilisée pour définir des constantes

```
#define  TAILLE_MAXIMUM 1000
#define  VERSION "Truc 3.14.16"
```

Le préprocesseur remplace les occurrences des identificateurs - si elles ne sont pas dans une chaîne par la chaîne correspondante, qui va jusqu'à la fin de la ligne. Par exemple

```
printf("VERSION = %s\n", VERSION);
```

est expansé en

```
printf("VERSION = %s\n", Truc 3.14.16);
```

Pour faire une expansion *dans* un chaîne on peut employer deux trucs

— l'opérateur `##` qui concatène des chaînes pendant le prétraitement

```
printf("VERSION = " ## VERSION ## "\n");
```

— le fait qu'une chaîne peut être une suite d'éléments entre guillemets

```
printf("VERSION = " VERSION "\n");
```

L'usage de `#define` ne se limite pas à définir des constantes, outre les macros que nous verrons plus loin, on peut s'en servir pour redéfinir n'importe quoi

```
#include <stdio.h>

#define SI  if (
#define ALORS ) {
#define SINON } else {
#define FINSI }

int main(int argc, char **argv)
{
        int a, b, max;
        printf("Donnez deux entiers ");
        scanf("%d %d", &&, &b);
        SI a > b
```

9. `/usr/local/include`, `libdir/gcc/target/version/include`, `/usr/target/include`, `/usr/include`

```
        ALORS
            max = a ;
        SINON
            max = b ;
        FINSI
        printf("max = %d\n", max);
}
```

ce qui n'est pas forcément une pratique recommandable.

12.3 #ifdef

Directive qui permet de savoir si une variable du préprocesseur a déjà été définie ou non. Permet en particulier d'éviter les problèmes de double inclusion.

```
/* fichier truc.h */
#ifndef TRUC_H
#define TRUC_H
    ...
#endif
```

ou d'inclure du code conditionnel

```
#ifdef MISE_AU_POINT
    printf("je suis passe dans truc(%d)\n", n);
#endif
```

qui sera compilé uniquement si la variable est définie plus haut, ou par l'option `-D` du compilateur :

```
gcc -DMISE_AU_POINT prog.c -o prog
```

12.4 Macros

Enfin, il y a la possibilité de réaliser des macros-définitions (macros), avec des paramètres. Exemple

```
#define CARRE(n)  ((n)*(n))
...

printf("2*2 = %s\n", carre(2));
```

Attention, les macros ne sont pas des fonctions. Si on avait écrit

```
#define CARRE(n)  n*n
```

le terme `CARRE(1+1)` se serait expansé en `1+1*1+1` qui vaut 3...

De même si on écrit

```
#define echanger(a,b)  {int c = a; a = b; b = c;}
```

on va avoir des problèmes avec

```
echanger((t[i++],t[j]);
```

qui s'expanse en

```
{int c = t[i++]; t[i++] = t[j]; t[j] = c;}
```

dans lequel i est incrémenté deux fois...

13 Allocation dynamique : malloc, free

La fonction `malloc()` permet de demander au système de réserver (allouer) en mémoire un espace d'une certaine taille.

La fonction retourne un pointeur (`void *`) vers cet espace.

```
#include <stdlib.h>

struct Date {
  int jour, mois, annee;
};

void truc() {
   struct Date * ptr;
   ...
   ptr = malloc(sizeof(struct Date));
   ...
}
```

Cet espace reste alloué jusqu'à ce que le programmeur en demande explicitement la libération (restitution), par l'appel de `free(ptr)`, ou jusqu'à la fin du programme.

C'est ce qu'on appelle l'*allocation dynamique*, par opposition à

— l'allocation **statique** des variables globales du programme

— l'allocation **automatique** des variables locales, qui sont crées et détruites automatiquement à l'entrée et à la sortie des fonctions.

A noter

— `malloc` prend comme paramètre le nombre d'octets désirés. Pour allouer un tableau de N dates, on appellera logiquement `malloc(N * sizeof(struct Date))`.

— elle retourne un pointeur de type `void*`, ce qui correspond en C à un pointeur générique qui ne peut pas être dé-référencé. Le type de ce pointeur doit être converti (on parle de *type cast*) en pointeur vers une structure Date.

Signalons l'existence d'une fonction `realloc()` qui change la taille mémoire d'une zone allouée, en la déplaçant et la recopiant éventuellement.

```
struct TableauExtensible {
    int taille;
    int *t;
};

void initialiser(struct TableauExtensible *tab, int tailleInitiale)
{
   tab->taille = tailleInitiale;
   tab->t      = malloc(tailleInitiale * sizeof(int));
}

void affecter(struct TableauExtensible *tab, int indice, int valeur)
{
   tab->t[indice] = valeur;
}

void valeur(struct TableauExtensible *tab, int indice)
{
   return tab->t[indice];
}

void redimensionner(struct TableauExtensible *tab, int taille)
{
   tab->taille = taille;
   tab->t      = realloc(tab, taille * sizeof(int));
}
```

14 Traitement d'exceptions : `setjmp`, `longjmp`

Ce couple d'instructions permet de réaliser des sauts d'une fonction à une autre. C'est une forme rudimentaire de mécanisme d'*exception*, utilisée principalement pour faire du rattrapage d'erreurs.

Lors d'un appel à la fonction `setjmp()`, le contexte (contenu des registres etc.) est sauvegardé dans un tampon (`jmp_buf`) et l'exécution continue en retournant la valeur 0.

Quand on appelle `longjmp()`, l'exécution se poursuit au "point de retour" indiqué dans le tampon (c'est-à-dire l'appel à la fonction `longjmp()`) , en transmettant une valeur différente de 0.

Il est donc possible, en regardant la valeur retournée par `setjmp()`, de voir si on est dans le cas normal ou dans le cas de traitement d'une exception.

Exemple :

```
#include <setjmp.h>

jmp_buf erreur_fatale;          // variables globales
jmp_buf commande_abandonnee;

voir executer(char commande[]) {
    ....
    if (....) {
        longjmp(commande_abandonnee, 42);
    }
    ....
}

int main() {
   if ( setjmp(erreur_fatale) == 0 ) {
     while (1) {
       char commande[128];
       lire(commande);
       int n = setjmp(commande_abandonnee);
       if (n == 0) {
           executer(commande);
       } else {              // exception ``commande abandonnée''
           printf("commande abandonnée (err %d)\n", n);
       }
    }
   } else {                       // exception ``erreur fatale''
          printf("erreur fatale\n");
        }
}
```

A Un exemple : simulateur de processeur

À mettre à jour

Le programme suivant simule le fonctionnement d'un ordinateur très simple, composé d'un processeur et d'une mémoire de 4096 mots de 16 bits.

Le processeur comporte 3 registres

- un *accumulateur* 16 bits
- un *compteur de programmes*, registre qui contient le numéro de la prochaine instruction à exécuter,
- un *registre d'instruction* qui contient l'instruction en cours d'exécution

Il ne possède que quelques instructions :

- `halt`, qui arrête le processeur ;
- `loadi`, qui charge une constante dans l'accumulateur,
- `load`, qui charge dans l'accumulateur un mot de la mémoire. Le numéro du mot est indiqué dans les 12 bits de droite de l'instruction.
- `loadx` et `storex`, utilisant l'adressage indirect : l'opérande effectif est le mot dont le numéro est contenu dans le mot dont l'adresse est codée dans l'instruction.
- `store`, qui copie le contenu de l'accumulateur dans un mot de la mémoire.
- `add` et `sub` qui ajoutent ou retranchent à l'accumulateur un mot de la mémoire
- `jmp` (jump) qui modifie le compteur de programme pour "sauter" à une autre instruction que celle d'après

— `jneg` qui fait sauter seulement si le contenu de l'accumulateur est négatif
— `jzero` qui fait sauter seulement si le contenu de l'accumulateur est négatif
— `call` qui appelle un sous-programme,
— `jmpx` qui fait un saut indirect.

A.1 Notes

1. `#define` définit une macro du préprocesseur. La chaîne "`TAILLE_MEMOIRE`" sera remplacée par l'expression `(1 << 12)` partout dans la suite du texte source.
2. Un `Mot` est une donnée sur 16 bits, qui peut être un entier signé ou non signé. Les deux types sont définis dans `stdint.h`.
3. L'énumération définit une suite de constantes : `HALT` vaut `0`, LOAD 1, etc.
4. La fonction utilitaire `nInstruction` fabrique un mot qui correspond à une instruction : les 4 premiers contiennent le code instruction (obtenu en décalant le code fourni de 12 positions vers la gauche), le reste l'adresse (à qui on applique un masque pour ne conserver que les 12 bits de droite).
5. La fonction `voir_donnes` affiche les valeurs signées contenues dans une partie de la mémoire du simulateur.
6. la fonction `voir_code` extrait la partie opération (4 bits) et l'adresse (12 bits), d'un mot contenant des instructions.