Cours / Travaux Pratiques

Systèmes d'Exploitation

Programmation système en langage C

A. Lebret, L3 SPI, © Université Paris-Est Créteil, 2011/2015

Table des matières

AU SUJET DES SÉANCES DE TRAVAUX PRATIQUES

L'objectif de ces séances de travaux pratiques d'une durée totale de 48 heures en salle est d'introduire les différents aspects du langage C avant d'attaquer la programmation système. Certaines parties ne sont d'ailleurs valables que sous le système Linux.
Sachez que votre maîtrise de ce langage ne sera proportionnelle qu'au temps que vous y aurez consacré en toute **autonomie**.

Vous effectuerez un compte-rendu électronique au format « ODT » (*Open Document Text*) à l'aide de la suite OpenOffice. Vous y inclurez vos codes sources **correctement commentés** ainsi que la réponse aux questions. Des captures d'écran pourront être incluses à l'aide d'outils tels que `gnome-panel-screenshot` (capture d'écran) et `gimp` (retouche d'images).

La note du TP sera constituée par :
- la vérification hebdomadaire de l'état d'avancement du rapport pour les différentes parties ;
- un entretien individuel de 10mn.

et tiendra compte du comportement et du degré d'autonomie à chaque séance.

Typographie :

répertoire, fichier, **commande**, code, **concept/vocabulaire**, **mot réservé**.

Pour vos recherches documentaires, je vous conseille :
- KERNIGHAN, Brian, RITCHIE, Dennis. *Le langage C : norme ANSI*. 2ème édition. Éditions Dunod, 2004, 280 p., ISBN 2100487345 ;
- RIFFLET, Jean-Marie, YUNES, Jean-Baptiste. *UNIX : Programmation et communication*. Éditions Dunod, 2003, 800 p., ISBN 2100079662.

1. PRÉSENTATION DU LANGAGE C

Notions : langage interprété vs. langage compilé, `gcc`, `-Wall`, `a.out`, *shell*.

1.1. Historique

Le langage C a été créé par Brian Kernighan et Dennis Ritchie au début des années 70 afin d'éviter autant que possible l'utilisation de l'assembleur dans l'écriture du système UNIX. L'avantage du C sur l'assembleur est qu'il permet aux programmes d'être plus concis, plus simples à écrire et plus facilement portables sur d'autres architectures. En effet, en C, tout ce qui pose problème de portabilité est répertorié sous forme de **`fonction`**.
C est un langage :

- qui permet de tout faire ou presque ;
- qui permet de créer des exécutables très rapides (compilés) ;
- qui possède peu d'instructions.

1.2. Un premier programme

```
1. /*
2.  * Mon premier programme C.
3.  */
4.  #include <stdio.h>
5.  int main()
6.  {
7.    printf("Mon premier programme en C\n");
8.    return 0;
9.  }
```

Ce programme se compose de deux ensembles :
1) L'en-tête
Les lignes 1, 2, 3 fournissent un commentaire sur plusieurs lignes encadré par les caractères `/*` et `*/`. Il est possible depuis 1999 d'écrire un commentaire mono-ligne C++ à l'aide des caractères `//`.
Suit une information pour le compilateur :

`#include <sdtio.h>`

- `#` indique une information destine en propre au compilateur qui est le programme qui traduit le code que vous écrivez en langage machine.
- **`include <stdio.h>`** : le compilateur devra inclure la bibliothèque `<stdio.h>` qui fournit des outils standardisés (*std.*.). Ce sont des fonctions, comme `printf()` et `scanf()`, qui gèrent les entrées et les sorties d'informations (*...io: in/out*) et permettent donc le dialogue avec la machine. Nous trouverons cette ligne dans presque tous nos programmes.

Nous disposons en C de nombreuses bibliothèques, qui rassemblent un nombre impressionnant de fonctions. On les appelle par la directive **`#include <*.h>`**. Nous en aurons sans cesse besoin.

2) Le programme principal (`main()` en anglais signifie « le plus important »).
On y trouve aussi deux parties: la fonction `main()` et le texte du programme lui même entre deux accolades `{ }`.

- `main()` est une fonction.

On reconnaît une fonction aux parenthèses `()` qui suivent son nom. Ici commence le programme principal.

- Entre les accolades `{}` nous trouvons le corps de la fonction `main()` qui est constitué ici de deux lignes de code:
  `printf("Mon premier programme en C\n");`
  C'est une fonction car son nom est suivie de parenthèses. Son rôle est d'afficher à l'écran le message qui se trouve entre les parenthèses (...). Nous pouvons utiliser cette fonction, comme toutes celles de la bibliothèque `<stdio.h>`, que nous avons activée avec la directive `include` deux lignes plus haut. Ce message est une chaîne de caractères qui doit être écrite entre des guillemets anglais `""`. Le caractère `\n` signifie un saut de ligne suivi d'un retour chariot. Enfin, et cela n'est pas le moins important, au bout de la ligne il se trouve un point-virgule `;`. Il indique la fin de la déclaration de l'instruction `printf()`, c'est un délimiteur.
  `return 0;`
  Qui indique que la fonction renvoie la valeur `0`, ici comme il s'agit de la fonction principale : au *shell* qui l'a appelé, cela dès la fin de son exécution.

1.3. Mise en oeuvre

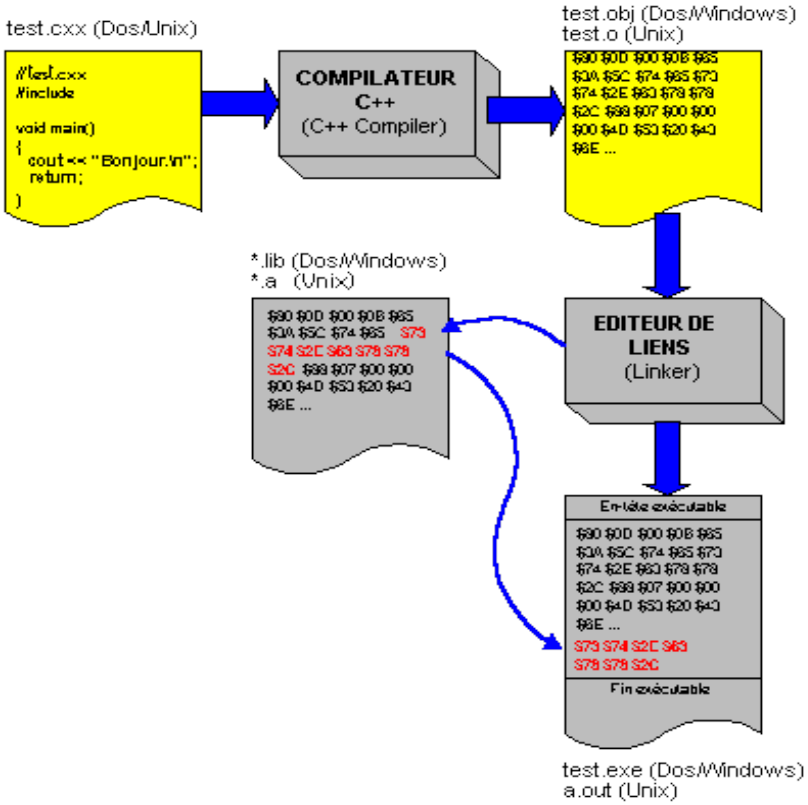
Pour compiler et exécuter ce programme sous GNU/Linux, il faut d'abord le copier dans un fichier se terminant par l'extension `.c` à l'aide d'un éditeur de texte (**`vi`** ou **`gedit`**). Appelons ce fichier `intro.c`. Il s'agit du fichier « source ». On le compile sous GNU/Linux avec **`gcc`** (sous d'autres UNIX, le compilateur de base fourni avec le système s'appelle **`cc`**) :

```
gcc -o intro intro.c
```

Le compilateur procède en deux temps comme le montre l'illustration 1, mais de manière bien souvent transparente pour l'utilisateur :

1) La compilation du fichier source en code machine. Le pré-processeur ira à ce moment-là chercher dans le fichier `<stdio.h>`, le « prototype » de la fonction `printf()`, puis l'incorporera au source, avant de générer un fichier « objet » en code machine. Ce dernier portera le nom que vous avez donné au fichier source avec l'extension `*.o` (`.obj` sous Ms-Windows).
2) L'appel de l'éditeur de liens, chargé d'intégrer le code machine de la fonction `printf()` à celui précédemment obtenu.

Pour que le programme devienne exécutable, le compilateur lui ajoutera du code propre au système informatique utilisé.
Un fichier exécutable qui s'appelle **`intro`** est alors généré. Si l'option `-o` n'avait pas été spécifiée, le nom du fichier exécutable produit aurait été : **`a.out`**.



Pour le lancer :

```
./intro
Mon premier programme en C
```

L'utilisation de l'option `-Wall` permet de s'assurer que le code est syntaxiquement correct (tolérance du compilateur **gcc**) :

```
gcc -Wall -o intro intro.c
```

1.4. Remarques

1) La directive `#include` doit se placer le plus gauche possible.
2) Pour faciliter la lecture d'un programme, il est indispensable de lui apporter un maximum d'explications en clair. JAMAIS, VOUS NE FEREZ TROP DE COMMENTAIRES !

```
/* Il est possible en C de faire des
          commentaires sur plusieurs lignes.*/
/*  condition: /* de ne pas
                les imbriquer comme ça! */ ERREUR */

// Par contre avec ceux-ci, c'est le reste de la ligne,
// derrière les premiers caractères //, qui sera considéré
    // comme un commentaire (valable depuis 1999).
```

3) Les conventions d'écriture sont peu nombreuses :
Les mots utilisés doivent être séparés par un espace.

```
printf("Mon premier programme en C\n");
```

D'abord `printf`, puis la parenthèse qui joue le rôle se séparateur, ensuite le texte entre guillemets, enfin la parenthèse de fermeture qui est aussi un séparateur et le point-virgule qui clôt la ligne. Mais on peut ajouter les espaces qu'on veut:

```
printf  ("Mon premier programme en C\n"
                  )   ;
```

Manipulation

1) « `intro.c` » : recopier le programme précédent, le compiler puis l'exécuter. Ensuite, le modifier afin qu'il produise le résultat suivant :

```
./intro
Mon premier
programme en C
```

2) Modifier la valeur retournée par la fonction `main()` et une fois le programme compilé et exécuté, tester la commande *shell* suivante :

```
echo $?
```

2. VARIABLES ET TYPES FONDAMENTAUX

Notions : entiers (`char`, `int`, `short`, `long`), réels en virgule flottante (`float`, `double`), nombres signés et non-signés (`signed`, `unsigned`).

2.1. Déclarer une variable pour pouvoir l'utiliser

En C, une variable est un emplacement destiné à recevoir une donnée. Cet emplacement se situe dans une portion de la zone de la mémoire allouée au programme lors de son exécution et appelée **pile** (*Stack*) (voir le chapitre 15). Un programme en cours d'exécution quant a lui, est appelé **processus**.
Le nom de la variable (par exemple `maVariable`) est une **étiquette** (appelée aussi « label ») qui identifie l'**adresse relative** de la variable, par rapport à l'**adresse absolue** qu'aura le processus en mémoire lors de son exécution. Le compilateur se charge de l'association étiquette-adresse relative.

En C, toute variable doit donc être déclarée avant d'être utilisée !

Un nom de variable doit commencer par une lettre ou bien par le caractère _. La suite peut être composée de chiffres, de lettres ou de _.

Attention, le compilateur fait la distinction entre minuscules et MAJUSCULES !

Pour déclarer une variable, on écrit son type suivit de son nom. Le type de la variable spécifie le format et la taille des données qui pourront être stockées grâce à elle en mémoire.
Il est possible de déclarer plusieurs variables d'un même type sur la même ligne en les séparant par des virgules.

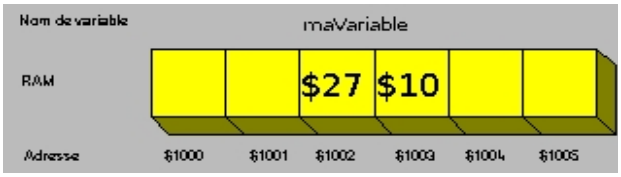
Exemple de déclarations de variables :

```
int   age;
float prix, taux;
long  numero_secu;
```

Par exemple, si l'on définit sur une architecture 32 bits :

```
long maVariable;
```

Alors, le compilateur pourra par exemple associer l'étiquette `maVariable` aux adresses relatives `$1002` et `$1003`, puisqu'un entier long comme nous le verrons plus loin est défini sur 8 octets, et qu'une mémoire de 32 bits nécessite deux emplacements pour le stocker.
Si la valeur `10000` est affectée à `maVariable`, alors les valeurs `$27` et `$10` seront stockées en mémoire comme le montre l'illustration 2 ($10000 = 2710_{16}$) :



Une variable doit toujours être déclarée à l'intérieur d'une fonction (exceptées les variables dites **globales**, voir un peu plus loin) et avant toute instruction. La fonction `main()` ne fait pas exception à la règle :

```
int main()
{
    int   a, b;
    float x = 5;

    b = 0; /* une première instruction */
    return 0;
}
```

2.2. Types fondamentaux

Les types des variables que nous allons utiliser appartiennent à trois familles:

- les caractères uniques ;
- les nombres entiers ;
- les nombres réels.

2.2.1. Les variables de type caractère uniquement

On parle de variable à caractère unique pour dire : variable ne contenant qu'un seul caractère, la lettre `'a'` par exemple. Et ceci par opposition aux chaînes de caractères comme `"Mon premier ..."`.
Elles sont déclarées avec le mot réservé **`char`** et occupent 8 bits (*BInary digiT*) de mémoire c'est-à-dire un octet (*byte*).
La déclaration par défaut d'une variable **`char`** suppose qu'elle possède un signe (+) ou (-), on dit parfois qu'elle est « signée », traduction de l'expression **`signed char`**.
En pratique, il est exceptionnel d'avoir besoin d'un caractère qui ait une valeur algébrique. Il est possible d'utiliser une option de compilation qui permet de changer cette valeur par défaut. Les 8 bits peuvent alors tous être utilisés à la représentation de caractères, ce qui permet d'en définir $2^8 = 256$.

2.2.1.1. Déclaration et initialisation de variables caractères

Exemple :

```
#include <stdio.h> /* nécessaire pour printf(). */

int main()
```

```
{
  /* déclaration et initialisation de 2 variables char. */
  char car1 = 'A', car2;
  car2 = 'a';
  printf(" 1ère affichage:    car1: %c\t car2: %c\n",
           car1, car2);

  /* modification des variables avec les codes
     ASCII de Z et z.
   */
  car1 = 90;
  car2 = 122;

  printf(" 2ème affichage:    car1: %c\t car2: %c",
           car1, car2);
  return 0;
}

/*  Résultats:

 1ère affichage:    car1: A    car2: a
 2ème affichage:    car1: Z    car2: z                    */
```

Nous déclarons et initialisons dans ce programme deux variables **char**. Nous remarquons plusieurs choses nouvelles:

- Les déclarations des deux variables sont séparées par une virgule.
- On utilise le signe d'affectation (=) pour initialiser aussitôt la première variable car1 avec la lettre A entre deux apostrophes 'A', qu'il ne faut pas confondre avec les guillemets anglais "A". Notez que nous pouvons faire la déclaration de car2 sur la première ligne et ne l'initialiser que par la suite.
- Dans printf(), remarquez la séquence \n que nous connaissons déjà et la séquence \t qui commande une tabulation horizontale.
- A l'intérieur de la chaîne qui va être affichée, le symbole (%), nous le savons, indique l'emplacement d'une variable. Il est suivi de la lettre c qui indique le type **char** de la variable affichée.
- Dans notre exemple, nous affichons deux variables à l'intérieur de la même chaîne. Nous réservons pour chacune un emplacement avec %c. Il faut indiquer au C les variables destinées à ces réservations. Nous le faisons après la virgule en indiquant dans l'ordre "...", car1, car2); sans oublier le point-virgule final.
- Puis nous modifions directement nos deux variables en utilisant le code ASCII des lettres Z et z (90 et 122) et nous vérifions avec printf().
- L'accolade } ferme le domaine de la fonction main() et le programme.

2.2.1.2. Utilisation de l'équivalence entre les caractères et leur représentation ASCII

Exemple :

```
#include <stdio.h>

int main()
{
  /* déclaration et initialisation simultanées. */
  char car1 = 'A', car2 = 'a';
  printf(" 1ère affichage:    car1: %c\t car2: %c\n",
```

```
          car1, car2);
  /* modification des variables caractères:
     on affecte à car1 sa propre valeur + 25     */
  car1 = car1 + 25;
  car2 = car2 + 25;

  printf(" 2ème affichage:    car1: %c\t car2: %c",
          car1, car2);
  return 0;
}

/*  Résultat:  le même que celui de l'exercice précédent

 1ère affichage:    car1: A   car2: a
 2ème affichage:    car1: Z   car2: z                    */
```

Il ressemble au précédent. En examinant un tableau ASCII, vous remarquerez que le `A` vaut 65 et que le `Z` vaut 90. Soit une différence de 25, différence que l'on retrouve entre `a` et `z`. Nous modifions donc nos variables en leur ajoutant 25.
Surtout ne pensez pas que le mélange entre le type **`char`** et des nombres soit quelque chose de général en C ! Ce serait même plutôt exactement le contraire et une exception quasi unique dans ce langage. Cela tient uniquement au fait, que pour le C, les caractères sont complètement assimilés à leur code ASCII, c'est-à-dire à des nombres entiers.

2.2.1.3. Séquences dites d'échappement.

Certain caractères non affichables peuvent être représentés par des séquences particulières de caractères, toujours précédés de la barre gauche (\).
Nous connaissons déjà `\n` (*new line*) qui est le retour à la ligne et `\t` qui commande un tabulation horizontale. Il en existe d'autres:

Séquence d'échappement	***Code ASCII***	***Signification***
`\a`	7	sonnerie
`\b`	8	retour arrière
`\t`	9	tabulation horizontale
`\n`	10	retour à la ligne
`\v`	11	tabulation verticale
`\f`	12	nouvelle page
`\r`	13	retour chariot
`\"`	34	guillemet anglais
`\'`	39	apostrophe
`\?`	63	point d'interrogation
`\\`	92	barre gauche (anti-slash)
`\0`	0	caractère nul

Le caractère nul est utilisé pour marquer la fin d'une chaîne de caractères. Ce n'est pas l'équivalent du caractère zéro qui se note `'0'` ou 48 en code ASCII. Quand un autre caractère est précédé de `\`, le résultat est totalement imprévisible!

2.2.2. Les trois types de variables entières

- le type **short** (**signed** ou **unsigned**) qui est codé avec 2 octets
- le type **int** (**signed** ou **unsigned**) qui est codé avec 2 ou 4 octets,
- le type **long** (**signed** ou **unsigned**) qui est codé avec 4 ou 8 octets.

Il faut noter que le type **int** correspond toujours, pour une machine, à la longueur de l'unité ou mot de mémoire.
Sur une machine 16 bits, le type **int** est codé sur 2 octets, sur une machine 32 bits actuelle, il est codé sur 4 octets, et sur les futures architectures 64 bits, il pourra atteindre 8 octets.

2.2.2.1. Les entiers représentés avec 16 bits (2 octets)

a) *Le type* `short` *ou* `signed short`

La variable de type **short** prend 2 octets de mémoire, soit 16 bits. Le premier bit est appelé bit de poids fort, c'est lui qui représente le signe (+) quand il vaut 0 et le signe (-) lorsqu'il vaut 1. Les 15 autres servent au codage de la valeur.
La plus grande valeur que nous pourrons coder est composée du zéro, qui donne à l'entier une valeur positive, et de 15 fois le 1:

`0111 1111 1111 1111` <=> 32767 en décimal

Pour le convertir en décimal on multiplie chacun des quinze 1 par une puissance croissante de 2, en commençant par 0:

$2^0+ 2^1+ 2^2+ 2^3+...+ 2^{14} = 32767 = 2^{15} - 1$ (comptez il y en a 15)

```
0111 1111 1111 1110     <=>         32766= 2^1+...+ 2^14
.... .... .... ....                 et ainsi jusqu'à
0000 1000 0000 0000     <=>         2048= 2^11
.... .... .... ....
0000 0000 1000 0000     <=>         128= 2^7
.... .... .... ....
0000 0000 0000 1000     <=>         8= 2^3
.... .... .... ....
0000 0000 0000 0001     <=>         1= 2^0    puis
0000 0000 0000 0000     <=>         0
```

Maintenant se pose le problème de la représentation des entiers négatifs. On a choisi le système du complément à 2. Qu'est-ce à dire?

```
Prenons la valeur  de 1 sur 2 octets:       0000 0000 0000 0001
Changeons les 1 en 0 et les 0 en 1  :       1111 1111 1111 1110
Ajoutons la valeur binaire 1  :         +   0000 0000 0000 0001
On additionne  :                        =   1111 1111 1111 1111
```

Nous obtenons la valeur inverse de 1, c'est-à-dire -1.

```
1111 1111 1111 1111     <=>         -1    et de même
1111 1111 1111 1110     <=>         -2
.... .... .... ....
1111 1111 1111 1000     <=>         -8
.... .... .... ....
```

```
1111 1111 1000 0000     <=>      -128
.... .... .... ....
1111 1000 0000 0000     <=>     -2048
.... .... .... ....
1000 0000 0000 0001     <=>    -32767
1000 0000 0000 0000     <=>    -32768
```

Nous avons pu ainsi représenter l'ensemble des entiers entre -32768 et 32767. C'est le domaine des **short**. Si vous en sortez lorsque vous utilisez une variable déclarée **short** que se passe-t-il ?
Exemple : **Sortie du domaine de définition des** short

```
#include <stdio.h>

int main()
{
  short test0 = 32766, test1;
  printf(" 1ère affichage:    test0: %d\n", test0);
  test1 = test0 + 1;
  printf(" 2ème affichage:    test1: %d\n", test1);
  test1 = test0 + 2;
  printf(" 3ème affichage:    test1: %d\n", test1);
  test1 = test0 + 3;
  printf(" 4ème affichage:    test1: %d\n", test1);
  return 0;
}

/*  Résultat:

 1ère affichage:    test0:  32766
 2ème affichage:    test1:  32767
 3ème affichage:    test1: -32768
 4ème affichage:    test1: -32767                          */
```

On déclare deux variables **short** et on initialise aussitôt la première test0 à 32766, à peine au-dessous de la valeur limite supérieure. Puis on affecte successivement test1 avec la valeur de test0 augmentée de 1, 2, 3.
Le premier affichage donne la valeur initiale, le second est normal. Par contre le 3ème nous donne une valeur inattendue, l'autre extrémité du domaine de validité des **short** signés, -32768 .
Binairement cela est parfaitement logique:

```
     0111 1111 1111 1111     ( en binaire 1 + 1= 10 je pose 0
+    0000 0000 0000 0001     ( et je retiens 1...cela vous
=    1000 0000 0000 0000     ( rappelle-t-il quelque chose ?

32767 + 1 = -32768             C'est correct!
```

Pour nous, utilisateurs de nombres décimaux, cela peut générer quelques bizarreries dans nos programmes! C'est d'autant plus dangereux que cette sortie du domaine du type choisi pour la variable, peut se produire à l'intérieur d'une chaîne de calculs.
Exemple : **Sortie du domaine de définition des** short

```
#include <stdio.h>

int main()
{
  short t0, t1, t2, test3,
```

```
  entier0 = 440, entier1 = 80, entier2 = 2;

  t0 = entier0;       /* t0 = 440                */
  t1 = t0 * entier1;  /* t1 = 35200              */
  t2 = t1 / entier2;  /* t2 = 17600 en décimal   */

  printf(" Le résultat:\t t0= %d\n\t\t"
                  " t1= %d\n\t\t"
                  " t2= %d", t0, t1, t2);
  return 0;
}

/*  Le programme nous affiche:

 Le résultat:     t0= 440
           t1= -30336     =  (35200-32767-1)-32767-1
           t2= -15168

 En fait l'opération s'écrit:            t2= 440* 80/ 2
 si on change l'ordre des opérations:    t2= 440/ 2* 80
 ou si on utilise des parenthèses: t2= 440* ( 80/ 2)
 il n'y a plus de débordement                          */
```

Il faut donc que le programmeur veille, à l'intérieur de ses chaînes de calcul, à ce que les valeurs prises par les variables ne sortent pas de leur domaine de définition. Cela peut souvent se faire, comme dans ce programme, en optimisant l'ordre des opérations.

domaine des `short` `{ -32768, 32767 }`

L'affichage avec `printf()` utilise après le symbole `%` le suffixe `d` :

```
printf("Affichage de test1 de type short: %d", test1);
```

b) *Le type* `unsigned short`

Nous disposons de 16 bits puisque le bit de poids fort ne contient plus de signe. Nous pouvons donc représenter $2^0 + 2^1 + ... + 2^{15} = 2^{16} - 1 = 65535$ valeurs.

domaine des `unsigned short` `{ 0, 65535 }`

L'affichage avec `printf()` utilise après le symbole `%` le suffixe `u`:

```
printf("Affichage de test2 de type unsigned short: %u",
test2);
```

Et nous retrouvons, bien sûr, les mêmes problèmes en cas de sortie du domaine de définition.

2.2.2.2. Les entiers représentés avec 32 bits et plus (4 octets)

a) *Le type* `int` *ou* `signed int`

Avec un espace de 32 bits, en gardant le bit de poids fort pour le signe, il est possible de représenter $2^{31} - 1 = 2\ 147\ 483\ 647$ valeurs positives et autant + 1 valeurs négatives, ce qui nous donne:

domaine des `int` `{ -2 147 483 648, 2 147 483 647 }`

L'affichage avec printf() utilise après le symbole % le suffixe d :

```
printf("Affichage de test3 de type int: %d", test3);
```

b) Le type unsigned int

Nous pouvons utiliser les 32 bits pour la codification, ce qui donne: $2^{32} - 1 = 4\ 294\ 967\ 295$ valeurs possibles.

domaine des unsigned int { 0, 4 294 967 295 }

L'affichage avec printf() utilise après le symbole % le suffixe u:

```
printf("Affichage de test4 de type unsigned int: %u",
test4);
```

c) Le type long ou signed long

domaine des long { -4 294 967 296, +4 294 967 295 }

L'affichage avec printf() utilise après le symbole % le suffixe ld:

```
printf("Affichage de test5 de type long: %ld", test5);
```

d) Le type unsigned long

domaine des unsigned long { 0, 8 589 934 591 }

L'affichage avec printf() utilise après le symbole % le suffixe lu:

```
printf("Affichage de test6 de type unsigned long: %lu",
test6);
```

2.2.2.3. Quelques erreurs à éviter avec les entiers

Exemple :

```
#include <stdio.h>

int main()
{
  int z0 = 100, z1, z2, z3, z4;

  z1 = z0 / 2;        /* 50                */
  z2 = z0 / 2.4;      /* 41.6666           */
  z3 = z0 / 3;        /* 33.3333           */

  printf(" Le résultat:\t z0= %d\n\t\t"
         " z1= %d\n\t\t"
         " z2= %d\n\t\t"
         " z3= %d\n\t\t", z0, z1, z2, z3);

  z4 = z1 / z2* z3;   // 40
  printf(" z4= %d", z4);
  return 0;
```

```
}

/*  Le programme nous affiche:

 Le résultat:      z0= 100
                   z1= 50
                 z2= 41
                 z3= 33
                 z4= 33                                          */
```

Nous avons déclaré des variables entières, z1,.. Sans entrer dans le calcul binaire, regardons comment calcule la machine:

`100 / 2 = 50` : OK

`100 / 2.4 = 41` : au lieu de `41.6666`

Les décimales de l'opération `(100 / 2.4)` sont simplement tronquées. Notez que le résultat avec `2.4` tronqué serait `100 / 2 = 50`.

`100 / 3 = 33` : au lieu de `33.3333`. C'est la même chose.

Pour le calcul de `z4`, partons des valeurs obtenues par la machine, notez que les opérations se font de gauche à droite :

`z4 = 50 / 41* 33` `50 / 41 = 1` au lieu de `1.21951`

`1 * 33 = 33`

Là aussi, c'est au programmeur de surveiller son code.

2.2.2.4. Les notations hexadécimales et octales

- Format octal. Il faut ajouter un `0` devant la valeur : `010` (8 décimal)
- Format hexadécimal. Il faut ajouter `0x` ou `0X` devant la valeur : `0xF` (15 décimal)

ATTENTION : ne faîtes donc pas précéder les valeurs décimales par un `0`, le compilateur les prendrait pour des valeurs en base 8.

2.2.3. Les variables réelles

2.2.3.1. Pourquoi les machines calculent-elles toujours faux ?

Les nombres réels sont ceux que nous utilisons en algèbre. On les dit, en virgule flottante, puisqu'un nombre réel peut toujours s'écrire en représentation décimale, de façon unique sous la forme approchée :

$0.xxxxxxxxx * 10^{y}$

Exemple :

$1024.000 = 0.1024 * 10^{4}$ ou bien

$0.001024 = 0.1024 * 10^{-3}$ la virgule « flotte ».

`y` est l'**exposant**, ici `y = -3`,

la suite des `x` derrière le `0.` est la **mantisse**: `1024`.

`0.1024 = 1 * 1/10 + 0 * 1/100 + 2 * 1/1000 + 4 * 1/10000`

La mantisse sera formée, derrière le `0`, des 4 premiers `x`: `1024`. Chacun des chiffres de la

mantisse est compris entre 0 et 9 dans la numérotation décimale. Il est associé à une puissance de 1/10.

Dans les machines, en représentation binaire ce nombre réel est converti en base 2 et devient:
0.zzzzzzzzz * 2^t chacun des nombres tant lui même binaire.
Chacun des chiffres z est un 1 ou un 0 et il est associé à une puissance de 1/2.
L'exposant est t, une puissance négative de 2.
nombre = (1^{er} z) * $1/2^1$ + ($2^{ème}$ z) * $1/2^2$ + ($3^{ème}$ z) * $1/2^3$ +...

Exemples :
1 = 1/2 * 2 = $(1/2)^1$* 2^1
mantisse: le 0. et 1 fois 1/2 soit = 0.100000000
exposant: la puissance de 2 qui est ici = 1

de la même façon:
2 = 1/2 * 4 = $(1/2)^1$* 2^2
mantisse: 0.100000000
exposant: 11 qui est la traduction de 2 en binaire

3.5 = 0.875 * 2^2 on décompose 0.875 en puissances de 1/2:
0.875 = 0.50+ 0.25+ 0.125= $(1/2)^1$+ $(1/2)^2$+ $(1/2)^3$
mantisse en binaire: 0.1110 0000 0000
exposant: 11 qui traduit le 2 décimal.

Nous avons choisi pour nos exemples des cas simples. En général, la suite des puissances de 1/2 nécessaire pour recomposer la mantisse est infinie alors que la capacité des machines ne l'est pas !
Exemple :
1/20 = 0.8 * 2^{-4} en binaire la mantisse est une suite infinie:
0.1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100...

Il faut donc tronquer cette suite et nécessairement introduire une erreur d'arrondi.

Si, dans une boucle de calcul, on refait 1000 fois une opération, l'erreur sera en général égale à 1000 fois l'arrondi.
Il faut par exemple faire très attention lorsqu'on compare deux variables théoriquement égales car elles seront généralement différentes à cause des arrondis. En pratique on comparera leur différence à une valeur *epsilon* négligeable par rapport aux variables. Dans ce cas, *epsilon* est le « zéro numérique ».
Il faut aussi surveiller les opérations avec des nombres de grandeur très différentes:

```
#include <stdio.h>

int main()
{
  double x0 = 50, x1 = 1e30, x2 = -1e30, y0, y1;
  y0 = x1 + x0 + x2;
  printf(" y0= %lf\n", y0);

  y1 = x1 + x2 + x0;
  printf(" y1= %lf", y1);
  return 0;
}
```

```
/*  Le programme nous affiche:

 y0= 0.000000
 y1= 50.000000                                              */
```

Remarquez la déclaration très classique du type **double**, une variable réelle que nous allons définir par la suite, ainsi que la déclaration de son format dans printf() : %lf. Notez aussi la manière d'écrire sous une forme condensée un nombre comme 1 suivi de 30 zéros : 1e30.
Nous avons insisté sur ces difficultés en pensant surtout à ceux, techniciens ou gestionnaires, qui ont besoin d'utiliser des fonctions mathématiques et qui doivent connaître les limites de fiabilité de leurs calculs. Mais en général, les types de variables utilisées permettent couramment d'obtenir 14 chiffres significatifs avec des réels codés sur 64 bits et 18 avec des variables sur 80 bits, ce qui suffit largement dans la plupart des applications.

2.2.3.2. Les deux types de variables réelles : float, double

a) Conditions aux limites des domaines de définition

Lorsqu'il y a sortie du domaine de définition par dépassement de la borne supérieure, le programme s'arrête et affiche un message.
Quand il s'agit d'un dépassement de la borne inférieure, $2.23E^{-308}$ par exemple, le programme continuera avec un résultat arrondi à zéro, ce qui peut dans certains cas induire des erreurs. Le zéro peut être +0 ou -0, selon le signe utilisé pour l'approcher.

b) Le type float

Il occupe 4 octets de mémoire soit 32 bits (1+ 8+ 23).

signe (1)	exposant (8)	mantisse 23

domaine des float $\{ -3.40 * 10^{38}, -1.21 * 10^{-38} \} + \{ 0 \} + \{ 1.21 * 10^{-38}, 3.40 * 10^{38} \}$

L'affichage avec printf() utilise après le symbole % le suffixe f:

```
printf("Affichage de test7 de type float: %f", test7);
```

La longueur de la mantisse permet, dans des conditions normales, de garder 6 chiffres significatifs.

c) Le type double

Il occupe 8 octets de mémoire soit 64 bits (1+ 11+ 52).

signe (1)	exposant (11)	mantisse 52

domaine des double $\{ -1.79*10^{308}, -2.23*10^{-308} \} + \{ 0 \} + \{ 2.23*10^{-308}, 1.79*10^{308} \}$

L'affichage avec printf() utilise après le symbole % le suffixe lf:

```
printf("Affichage de test8 de type double: %lf", test8);
```

La longueur de la mantisse permet, dans des conditions normales, de garder 14 chiffres significatifs.

2.2.4. Mélange de variables entières et réelles dans une même expression

Il faut examiner son code avec une extrême attention quand on utilise des variables de types différents dans une même expression.

Exemple : **Mélange de types dans une même expression**

```
#include <stdio.h>

int main()
{
  int a = 3, b = 4, c, d;
  double t = 3.0, x, y;

  c = b / a;
  x = b / a;
  y = b / t;

  printf(" c= %d\t x= %lf\t y= %lf",c, x, y);
  x = 1.0 * b / a;
  y = b / a * 1.0;
  printf("\n\n x= %lf\t y= %lf", x, y);
  d = 1.0 * b / a;
  printf("\n\n d= %d", d);
  return 0;
}

/*

 c= 1 x= 1.000000 y= 1.333333

 x= 1.333333      y= 1.000000 ( y= 1 * 1.0 = 1.000000  )

 d= 1                                                  */
```

`c = b / a;` `a / b = 1.333333` tronqué à 1. Est-ce parce que `c` est un **`int`**? NON puisque dans:

`x = b / a;` `x` est un **`double`** et le résultat est tronqué à `1.000000` C'est donc le type des variables `a` et `b` qui déterminent celui de leur division. On confirme ce principe:

`y = b / t;` `t` est un **`double`**, le résultat de la division n'est pas tronqué: `w = 1.333333`

EN PRATIQUE, pour ne pas avoir de problèmes, on utilise un procédé simple :

`x = 1.0*b/a;` (et non `x = b/a*1.0;`)

La première opération est `1.0*b` dont le résultat est un réel et impose ce type à toute l'expression.

Nous faisons une chose très différente avec l'écriture:

```
int d;
d = 1.0 * b / a;
```

`1.0*b/a = 1.333333` nous l'avons vérifié plus haut, le résultat est pourtant: 1. C'est donc l'affectation d'une valeur **`double`** `1.333333` à `d` qui est un **`int`** qui force le résultat 1.

3. OPÉRATEURS

Notions : opérateur d'affectation (=), opérateurs arithmétiques (+, -, *, /, %), opérateurs de manipulation de bits (&, |, ~, ^, >>, <<), opérateurs logiques (&&, ||, !), opérateurs spéciaux (?:, (), **sizeof**, *, &).

Le C est très riche en opérateurs. Il utilise ceux des langages évolués mais aussi ceux de l'assembleur pour manipuler des bits.

3.1. L'opérateur d'affectation =

Ce n'est surtout pas un symbole d'égalité!

```
test = 5; /* signifie que la valeur 5 est affecté à la
variable test. */
5 = test; /* est illégal, on n'affecte pas une variable au
nombre 5.*/
```

- Le membre à droite doit être une valeur ou une expression calculée
- Le membre à gauche doit être impérativement un élément modifiable comme par exemple une variable, on dit qu'il s'agit d'une « *lvalue* » (contraction de *left value*).

```
test + 5 = y - 2; /* est rejeté, le membre de gauche n'est
pas une lvalue */
```

```
x = y = z = 0;  /* est correct si les variables sont du même
type. L'associativité se fait de droite à gauche. */
```

Mais on ne peut pas user de cette facilité dans une déclaration :

```
double x = y = z = 3;  /* est rejeté */
```

3.2. Les opérateurs arithmétiques

3.2.1. Les opérateurs dits « unaires »

Les opérateurs **unaires** n'agissent que sur un seul élément.

- + et - qui déterminent le signe d'une variable : -test;
- les opérateurs d'incrémentation et de décrémentation:
  - on incrémente i quand on écrit : i = i+ 1;
  - on décrémente j : j = j- 1;

Proche en cela de l'assembleur, le C offre des opérateurs qui permettent d'accélérer ces opérations:

- i++ ou ++i incrémente la variable i de 1.
- j-- ou --j décrémente j de 1.

Les 2 formes ne sont pas équivalentes:

```
z = 10;          /* signifie: test = 10 et z = 11 */
test = z++;      /* 10 est affecté à test puis on */
                 /* incrémente z. */

z = 10;          /* signifie: test = 11 et z = 11, */
test = ++z;      /* on incrémente z puis 11 est affecté */
                 /* à test. */

z = 10;
test = z--;      /* test = 10 et z = 9. */

z = 10;
test = --z;      /* test = 9 et z= 9. */
```

Il est possible d'écrire :

```
test = 5+ ++z;
```

on incrémente z, on l'ajoute à 5 et le résultat est affecté à test. L'inconvénient est de rendre le code plus difficile à lire. Ne vaut-il pas mieux écrire sur 2 lignes?

```
++z;             /* ou: z++; */
test = 5+ z;
```

On ne peut pas incrémenter ou décrémenter une expression :

```
(test * 3/z)++  /* est rejeté. */
```

Les opérateurs unaires ont la plus forte priorité dans les calculs.

3.2.2. Les opérateurs « binaires »

Ils portent sur deux termes. Parmi eux :

- + l'addition, - la soustraction,
- * la multiplication, / la division.
  On ajoute les opérateurs parenthèses () pour fixer l'ordre des calculs.
- L'opérateur modulo, noté %, qui donne le reste de la division entière d'un entier par un autre:
  `25 % 5= 0, 25 % 7= 4, 25 % 19= 6.`
- Les opérateurs raccourcis. Ils sont utilisés à cause de leur rapidité.
  `test = test + z;` s'écrit en raccourcis : `test += z;`
  Dans le premier cas, le C réserve une place en mémoire pour préserver la variable test, alors qu'avec l'écriture raccourcie l'opération se fait dans le registre qui contient test dont on ne conservera pas l'ancienne valeur. On a selon le même principe:

```
x += y; ⇔  x = x + y;          x -= y; ⇔  x = x - y;
x *= y; ⇔  x = x * y;          x /= y; ⇔  x = x / y;
x %= y; ⇔  x = x % y;
```

  Comme avec l'opérateur d'affectation, le membre à gauche doit être impérativement une « *lvalue* », un élément modifiable, et le membre à droite doit être une expression calculée ou une valeur.

3.2.3. Les opérateurs de manipulation de bits

Ils agissent directement sur le code binaire des mots machine, ce qui accélère les opérations simples. On qualifie ces opérations de « bas niveau ». Ces opérateurs ne s'appliquent qu'à des

variables de type entier ou assimilé: int, char, short, long int, signés ou non.

3.2.3.1. L'opérateur unaire d'inversion bit à bit : ~ [Alt] 126

Il inverse un à un tous les bits d'un mot :

```
                0111 1111 1111 1110     ⇔     32766
     ~32566 =>  1000 0000 0000 0001     ⇔   - 32767 = -( 32766+ 1)
```

C'est à 1 près la valeur inverse en décimal.

Exemple :

```
unsigned char a, b;
a = '\x0A';      // a = 00001010 (VRAI)
b = !a;          // b = 00000000
b = ~a;          // b = 11110101
b = -a;          // b = 11110110
```

3.2.3.2. Opérateurs logiques binaires bit à bit : & , | , ^

Ils s'appliquent à des valeurs entières :

Opérateur	*Signification*
&	ET logique bit à bit
|	OU logique bit à bit
^	OU logique exclusif bit à bit

Table de vérité :

X	Y	X & Y	X | Y	X ^ Y
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Exemples :

```
X           0000 1111 0000 1111     ⇔     3855
Y           0001 0000 1111 1001     ⇔     4345
X & Y       0000 0000 0000 1001     ⇔        9

X           0000 1111 0000 1111     ⇔     3855
Y           0001 0000 1111 1001     ⇔     4345
X | Y       0001 1111 1111 1111     ⇔     8191

X           0000 1111 0000 1111     ⇔     3855
Y           0001 0000 1111 1001     ⇔     4345
X ^ Y       0001 1111 1111 0110     ⇔     8182
```

```
unsigned char a, b, c;
a = '\x0A';       // a = 00001010
b = '\xC3';       // b = 11000011
c = a & b;        // c = '\x02'
```

```
c = a | b;          // c = '\xCB'
c = a ^ b;          // c = '\xC9'
```

3.2.3.3. Les opérateurs de décalage binaire vers la droite et vers la gauche

Ils permettent des opérations sur des entiers, les bits sortants sont perdus, les bits entrants sont des 0.

- **Décalage à gauche :** `x << y`

Exemple :

```
4345 << 2
```

La valeur binaire de 4345 est décale 2 fois gauche. C'est la multiplication par 2 autant de fois qu'il y a de décalages, mais à l'intérieur du domaine de définition des variables.

Voici 3 exemples où l'on utilise des entiers de 16 bits :

```
4345 << 2    0001 0000 1111 1001  ⇔  4345
             0100 0011 1110 0100  ⇔ 17380 = 4* 4345
             2 décalages
-15756 << 1  1100 0010 0111 0100  ⇔ -15756
             1000 0100 1110 1000  ⇔ -31512= 2* (-15756)
             1 décalage
-8250 << 2   1101 1111 1100 0110  ⇔  -8250
             0111 1111 0001 1000  ⇔  32536
             On est sorti du domaine des int :
             32536 = 32768 + (-8250* 4+ 32768) = limite + dépassement algébrique
```

- **Décalage à droite :** `x >> y`

Exemples :

```
4345 >> 2     0001 0000 1111 1001  ⇔   4345
              0000 0100 0011 1110  ⇔   1086 = 4345/ 4
              On divise par 2 à chaque décalage.
```

```
unsigned char a, b, c;
char x, y;
a = '\x0A';      // a = 00001010
b = '\xC3';      // b = 11000011
c = b << 6;      // c = '\xC0'
c = a >> 3;      // c = '\x01'
c = b >> 4;      // c = '\x0C'
x = '\xC3';
y = x >> 4;      // y = '\xFC' car x étant signé
                 // négativement (char), on réinjecte des
                 // 1 et non des 0
```

Pour ces opérateurs binaires on dispose d'expressions raccourcies :

```
x &= y    ⇔    x = x & y
x ^= y    ⇔    x = x ^ y
x >>= y   ⇔    x = x >> y
x |=  y   ⇔    x = x | y
x <<= y   ⇔    x = x << y
```

3.2.4. Les opérateurs logiques

Ils font des comparaisons ou des opérations logiques sur des variables en utilisant une convention : si la réponse à la question posée est :

« FAUX » l'opérateur renvoie 0

« VRAI » l'opérateur renvoie une valeur différente de 0.

3.2.4.1. Les opérateurs de comparaison (<, >, <=, >=, !=, ==)

Exemple :

si x = 3 et y = 0

```
x < y    renvoie    0      et      x > y    renvoie      1
x <= y              0      et      x >= y               1
```

!= est le test « différent de » :

```
x != y   renvoie    1
```

== est le test « d'égalité » :

```
x == y   renvoie    0
```

Il ne faut surtout pas confondre l'opérateur d'égalité == avec l'opérateur d'affectation =

3.2.4.2. Les opérateurs logiques (&&, ||, !)

Opérateur	*Signification*
!	NON logique
&&	ET logique (intersection)
||	OU logique inclusif (union)

Table de vérité :

X	Y	X && Y	X || Y	!X
0	0	0	0	1
1	0	0	1	0
0	1	0	1	1
1	1	1	0	0

Il ne faut pas les confondre avec les opérateurs binaires : &, | et ^ qui ne portent que sur des entiers.

Exemple :

```
#include <stdio.h>

int main()
{
  int t0 = 0, t1 = 1, t2 = 3,
```

```
  test0, test1, test2, test3, test4, test5;
  test0 = (t0 < t1) && (t1 < t2);     /* (1) && (1) => 1  */
  test1 = !(t0 < t1) && !(t1 < t2);   /* (0) && (0) => 0  */
  test2 = (t0 < t1) || (t1 < t2);     /* (1) || (1) => 1  */
  test3 = !(t0 < t1) || !(t1 < t2);   /* (0) || (0) => 0  */
  test4 = !((t0 < t1) && (t1 < t2)); /* test4=!test0 => 0 */
  test5 = !((t0 < t1) || (t1 < t2)); /* test5=!test2 => 0 */

  printf(" Le résultat:\t test0= %d\n\t\t"
         " test1= %d\n\t\t"
         " test2= %d\n\t\t"
         " test3= %d\n\t\t"
         " test4= %d\n\t\t"
         " test5= %d",
         test0, test1, test2, test3, test4, test5);
  return 0;
}

/*
 Le résultat:
 test0= 1
 test1= 0
 test2= 1
 test3= 0
 test4= 0
 test5= 0

 vérification des lois de Morgan:
    test4 = test3   !(X && Y)  = !X || !Y
    test5 = test1   !( X || Y) = !X && !Y
*/
```

Il faut noter plus particulièrement qu'il existe un ordre de priorité entre ces opérateurs :
`!` possède une priorité supérieure à `&&` et `&&` à `||`. Dans le calcul de `test1` la machine value d'abord `!(t0 < t1)` et `!(t1 < t2)` puis fait l'opération `&&`. Vous trouverez plus loin un tableau qui précise ces priorités.

3.2.5. Les opérateurs spéciaux

Nous étudierons :

- l'opérateur ternaire `?:`
- l'opérateur séquentiel `,`
- l'opérateur de conversion de type `()`
- l'opérateur **`sizeof`**`()`
- les opérateurs unaires de référence `&` et de déférencement `*`

3.2.5.1. L'opérateur ternaire conditionnel `?:`

```
C ? A : B
```

Il est dit « ternaire » puisque nous avons 3 expressions : `C`, `A` et `B`.

1) Il teste l'expression `C`.
2) Si cette expression est « `VRAI` », il value `A` et renvoie sa valeur.
3) Si cette expression est « `FAUX` », il value `B` et renvoie sa valeur.

Exemple :

```
#include <stdio.h>

int main()
{
  float a = -5.43, b = 3.21, test0, test1, test2, min, max;
  test0 = (a > 0) ? (a) : (-a);
  /* renvoie la valeur absolue. */
  printf(" test0= %f", test0);

  test1 = (a> b) ? (a - b) : (b - a);
  /* renvoie la valeur absolue de la différence. */
  printf("\n test1= %f", test1);

  test2 = (test0 > b) ? (test0 - b) : (b - test0);
  /* a est remplacé par sa valeur absolue avant le test. */
  printf("\n test2= %f", test2);

  min = (a < b) ? a : b;
  /* renvoie la valeur du plus petit. */
  printf("\n min= %f", min);

  max = (a > b) ? a : b;
  /* renvoie la valeur du plus grand. */
  printf("\n max= %f", max);
  return 0;
}

/*
 test0= 5.430000
 test1= 8.639999 au lieu de 8.64 c'est une erreur d'arrondi!
 test2= 2.220000
 min= -5.430000
 max= 3.210000
 */
```

Remarque : Remplacez dans la déclaration **float** par **double** et les %f par %lf dans les quatre printf(). L'erreur d'arrondi disparaît.

3.2.5.2. L'opérateur séquentiel ,

Il permet d'enchaîner plusieurs instructions, la dernière donne sa valeur à l'expression. On l'utilise par exemple pour :

- Des déclarations multiples : double x, y = 2.36e-5, z;
- Dans la partie initialisation et celle de fin de la boucle for comme nous le verrons plus loin.

3.2.5.3. L'opérateur de transtypage ()

Rencontré au paragraphe 2.1.4, l'opérateur de **transtypage** (*cast*) permet de forcer la conversion d'un type de variable en un autre type.

```
( nouveau type qu'on force )expression à convertir
```

Par exemple :

Nous cherchons à obtenir la valeur entière d'une expression `z` :

```
double x = 1.3e-4, y = 25.3, z;
int a;
z = (x - 3.5)/ (7 * y + 5.1) + 18;    // z = 17.980791
a = ( int )z;                         // a = 17
```

On a forcé la variable réelle `z`, à se convertir en un entier.
ATTENTION, cette opération présente des dangers quand on sort du domaine de définition des variables :

```
#include <stdio.h>

int main()
{
  int a0, b0;
  long int b1;  /* ou en abrégé: long b1;  */
  double x = 1.3e-4, y = 25.3, z;

  z = (x - 3.5) / (7 * y + 5.1) + 18;
  a0 = ( int )z;
  printf("\n z= %lf\t\t a0= %d", z, a0);
  z *= 2000;    /* équivalent de z= z* 2000; */
  b0 = ( int )z;
  printf("\n z= %lf\t b0= %d", z, b0);
  b1 = ( long int )z;  /* ou: b1= ( long) z;*/
  printf("\n z= %lf\t b1= %ld", z, b1);  /* %ld long  */
  return 0;
}

/*
 z= 17.980791     a0= 17         OK
 z= 35961.582108  b0= -29575     sortie du domaine des int
                                 -29575= (35961-32768)-32768
 z= 35961.582108  b1= 35961      OK
 */
```

Dans ce programme on passe aux **`long int`** pour pallier cet inconvénient. Notez que l'opération de conversion ne se fait qu'après le calcul de l'expression. Celui ci se fait dans les conditions normales :

```
a = ( int )((x - 3.5) / (7 * y + 5.1) + 18);
```

3.2.5.4. L'opérateur `sizeof()`

Il permet de calculer la taille en octets d'une variable et donc de son type.

```
sizeof(nom de la variable)
```

Exemple :

```
short int a = 25;
printf(" taille de la variable a= %d", sizeof(a));
/* Nous obtiendrons :  taille de la variable a= 2 */
```

L'écriture `sizeof(int)` aurait donné 4 et `sizeof(double)` aurait donné 8.

3.2.5.5. Les opérateurs unaires d'adresse et d'indirection `&` et `*`

Nous avons vu qu'au moment de la déclaration d'une variable, le compilateur réserve un espace mémoire pour stocker la valeur qui sera ensuite affectée à cette variable quand on l'initialise.
Le compilateur conserve dans une table le nom de la variable avec l'adresse de cette zone.

- L'opérateur unaire d'adresse (ou de référencement) `&` permet d'obtenir cette adresse (ou cette référence). On utilise le même symbole que celui du ET bit à bit qui lui, est un opérateur binaire. Cet opérateur s'applique à une variable : Si `a` est une variable, `&a` est une adresse, la référence de `a`.
- Inversement, on peut obtenir la valeur stockée à cette adresse avec l'opérateur unaire d'indirection (ou de déréférencement) qui utilise le même symbole que l'opérateur binaire de multiplication. Cet opérateur s'applique à une adresse : Si `&a` est une adresse, `*(&a)` est la valeur de la variable situe à cette adresse, donc `a`.

Exemples :

```
#include <stdio.h>

int main()
{
  int a = 16; /* déclaration et initialisation de a. */
  /* 1. Opérateur unaire d'adresse ou de référencement:
        Quelle est l'adresse, la référence de la variable
        a ?
   */
  printf(" adresse de a:    &a= $%d", &a);
  /* cette adresse est fixée au moment de la déclaration de
     la variable. Elle est associée à son nom dans une
     table.
   */

  /* 2. Opérateur d'indirection ou de déréférencement:
        Quelle est la valeur de la variable stockée à
        l'adresse (&a)? */
  printf("\n\n  valeur de a:     a= %d   ou *(&a)= %d",
      a, *(&a));
  /* on retrouve la valeur de la variable a  */
  return 0;
}

/*
  adresse de a:    &a= $6660     (ou une autre valeur)

  valeur de a:     a= 16   ou *(&a)= 16
 */
```

```
#include <stdio.h>

int main()
{
  int a = 16;
  *(&a) = 32;
  printf("\n\n valeur de a: a= %d ou *(&a)= %d", a, *(&a));
  return 0;
}

/* valeur de a: a= 32 ou *(&a)= 32 */
```

Ce dernier programme est presque identique au précédent. Une ligne est ajoutée, elle est capitale :
`*(&a) = 32;` ce qui a entraîné `a = 32`

L'adresse déréférencée a été modifiée. La variable a suivi. C'est lourd de conséquences pour l'évolution du langage. Une variable a été modifiée indirectement à partir de son adresse. Les développeurs du langage, pour généraliser ce comportement ont créé une nouvelle catégorie d'objets qui sont par leur nature des adresses. Si un de ces nouveaux objets est une variable `b`, sa valeur est donc une adresse. Si nous affectons à `b` l'adresse de la variable `a`,

`b = &a;`

Alors `*b`, la valeur déréférencée de `b`, est égale à la valeur de `a`. Si nous modifions `*b`, alors la variable `a` sera modifiée.
On appelle l'objet `*b`, valeur déréférencée de la variable adresse `b`, un **pointeur** car il semble qu'il montre, qu'il pointe la variable `a`.

Exemple :

```
#include <stdio.h>

int main()
{
  int a, *b;
  /* déclaration de a et de la valeur déréférencée *b,
     d'une adresse b. */

  b = &a;
  /* initialisation de b: l'adresse de a est affectée à la
     variable adresse b. */

  printf(" adresse de a contenue dans b: $%d", b);
  /* cette valeur est fixée au moment de la déclaration
     de a, elle ne change plus jusqu'à la fin du
     programme. */

  a = 10; /* initialisation de a. */

  printf("\n *b: %d", *b);
  /* on affiche *b, c'est le contenu de a ! */

  *b = 11; /* on modifie *b */

  printf("\n  a: %d", a); /* a est également modifiée! */
  return 0;
}

/*
 adresse de a contenue dans b: $6656
 cette adresse peut varier à chaque compilation.
 *b= 10
 a= 11
 *b, valeur déréférencée de b, est appelée un pointeur
 car on dit que b pointe la variable a.
 ATTENTION: pour que &a existe il faut déclarer la variable
 a et pour que b existe il faut déclarer le pointeur *b
*/
```

3.2.6. Ordre de priorité des opérateurs en C

Opérateur	*Signification*	*Priorité descendante*
`()`	appel de fonction	gauche->droite
`[]`	élément de tableau	
`.`	membre d'une structure ou d'une union	
`->`	pointeur sur un membre	
`sizeof`	taille d'un objet en octets	
`sizeof`	taille d'un type en octets	
`++`	post-incrémentation (lvalue++)	droite -> gauche
`++`	pré-incrémentation (++lvalue)	
`--`	post-décrémentation (lvalue--)	
`--`	pré-décrémentation (--lvalue)	
`~`	inversion de bits	
`!`	NON logique	
`-`	moins unaire	
`+`	plus unaire	
`&`	adresse	
`*`	indirection	
`( type )`	conversion de type	
`*`	multiplication	gauche -> droite
`/`	division	
`%`	modulo	
`+`	addition	
`-`	soustraction	
`<<`	décalage à gauche	
`>>`	décalage à droite	
`<`	plus petit que	
`<=`	plus petit ou égal	
`>`	plus grand que	
`>=`	plus grand ou égal	
`==`	égalité	
`!=`	inégalité	
`&`	ET logique bit à bit	

Opérateur	*Signification*	*Priorité descendante*
`^`	OU logique exclusif bit à bit	
`|`	OU logique bit à bit	
`&&`	ET logique	
`||`	OU inclusif	
`?:`	opérateur ternaire	
`=`	affectation	
`+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=`		
`,`	opérateur séquentiel	

4. SAISIR ET AFFICHER DES DONNÉES

Notions : putchar(), getchar(), gets(), puts(), fgets(), printf(), scanf().

Les fonctions qui permettent de dialoguer avec une machine sont regroupées dans la bibliothèque <stdio.h> appelée en tête des programmes par la directive #include.
Quand nous employons les mots « lire » et « écrire », nous nous mettons par convention à la place de l'unité centrale de la machine. Nous écrivons une phrase au clavier et la machine « lit » cette phrase et la place dans un coin de sa mémoire. Puis la machine pourra « écrire » cette phrase sur votre écran ou sur un morceau de disque.
Nous étudierons le couple d'instructions putchar(), getchar() qui permet de gérer l'entrée et la sortie de caractères isolés. Puis le couple de fonctions printf() et scanf(), d'utilisation plus générale.

4.1. Lecture et écriture de caractères isolés avec getchar et putchar

Ce sont deux macro-instructions, très rapides d'exécution, qui sont copiées directement par le compilateur dans votre code.
printf() et scanf() permettent aussi de gérer des caractères, mais ce sont des fonctions. Nous verrons qu'elles emploient une procédure de fonctionnement un peu plus lente.
Exemple :

```
#include <stdio.h>

int main(void)
{
  char car;
  car = getchar(); /* Entrez un caractère: x   */
  putchar(car);    /* Il est affiché.  */
  putchar('Z');    /* affiche Z à la suite. */
  return 0;
}

/*
  x   caractère entré
  xZ  caractères affichés
 */
```

Ces macros sont utiles dans des cas particuliers que nous rencontrerons par la suite :
- Il est possible de lire et d'écrire une chaîne de caractères en traitant ces caractères un par un, au moyen d'une boucle répétée plusieurs fois.
- On peut les utiliser pour détecter un caractère \n, retour à la ligne, qui clôt la saisie d'une chaîne de caractères: « si la valeur renvoyée par la macro est \n, alors nous sommes au bout de la chaîne ».
  - getchar() renvoie un **int**, ce qui lui permet de renvoyer un caractère, sur 8 bits, en cas de succès ou -1 en cas d'échec.
  - putchar() prend en argument un **int**.

Remarque : `fgetchar()` et `fputchar()` existent aussi et nécessitent une action sur <Entrée>.
Exemple :

```
char c;
...
c = fgetchar();
...
fputchar(c);
```

4.2. E/S de chaînes de caractères avec `gets`, `fgets` et `puts`

- `gets()` est une fonction à proscrire absolument (dangers d'explosion du tampon / *buffer overflow*). Cette fonction lit une ligne de l'entrée standard `stdin` et la copie dans le tampon passé en argument jusqu'à ce que `\n` ou `EOF` soit rencontré.
- `fgets()` permet de limiter le nombre de caractères à lire et à copier dans le tableau passé en argument.
- `puts()` écrit la chaîne passée en argument et rajoute `\n` à la fin.

Exemple :

```
char zone[70];
...
gets(zone); // voir remarque au-dessus si plus de 70
...         // caractères sont entrés au clavier.
puts(zone);
```

4.3. Afficher avec la fonction `printf()`

La fonction `printf()` permet l'affichage d'informations contenues dans des arguments qui sont des variables ou des expressions calculer. Elle « écrit » donc pour nous, des informations qui « sortent » du système.

```
printf("chaîne de caractères", des arguments);
```

Parmi ces caractères on peut inclure une séquence qui permet d'afficher des variables. Elle débute par le symbole `%`. Une suite de codes permet de définir avec précision le format d'affichage des variables.

```
%x.y (code du type)
```

1) `x` est la longueur de l'espace alloué pour la variable.

Si x est un nombre positif, la variable est cadrée à droite.
Si x est un nombre négatif, la variable est cadrée à gauche.

```
int a = 16;
printf(" a= %10d", a);      //  a= ........16
printf(" a= %-10d", a);     //  a= 16........
```

Si la variable à afficher dépasse la taille allouée, elle est tout de même affichée en totalité selon un format standard.

2) `.y` (précédé du point) indique le nombre de décimales à écrire après la virgule

```
double b = 9.87654321;
printf(" b= %.12lf", b);     // b= 9.876543210000
```

3) Il est possible de définir « dynamiquement », c'est-à-dire pendant le déroulement du programme, les paramètres `x` et `y`.
Dans la suite, on les remplace par des symboles `*`.

```
p = 20;
q = 14;
printf(" b= %*.*lf", p, q, b);
```

Les paramètres `p` et `q` dans l'ordre remplacent les deux symboles `*`.
Exemple :

```
#include <stdio.h>

int main()
{
  int a = 16, p, q;
  double b = 9.87654321;

  printf(" Voici une chaîne de caractères."); /* pas
                                                 d'arguments */
  printf("\n\n Formats standards: a=%d\t b=%lf", a, b);
  printf("\n\n a1= %10d\t\t a2= %-10d\n\n"
       " b1= %14.8lf\t b2= %14.12lf", a, a, b, b);
  p = 20;
  q = 14;
  printf("\n\n b3= %*.*lf", p, q, b);

  printf("\n\n Affichage du résultat d'une expression:"
       " Z= %18.14lf",  1.0* a/ p* q+ b);
  return 0;
}
/*
 Voici une chaîne de caractères.
 Formats standards: a=16     b=9.876543
 a1=         16         a2= 16


 b1=     9.87654321     b2= 9.876543210000


 b3=     9.87654321000000


 Affichage du résultat d'une expression: Z=
21.07654321000000     */
```

Notez que l'on peut obtenir directement le résultat du calcul d'une expression sans avoir à créer une variable pour contenir ce résultat. La chaîne de calculs est dans ce cas un argument.

4) Codification du type de la variable à afficher.

Type	*Codification*
caractère (**`char`**)	`%c`
entier signé (**`short`**, **`int`**)	`%d`
entier non signé (**`unsigned short`**, **`unsigned int`**)	`%u`

Type	*Codification*
entier en hexadécimal	`%x` ou `%X`
entier en octal	`%o`
entier long signé (`long int`)	`%ld`
entier long non signé (`unsigned long int`)	`%lu`
entier long en hexadécimal	`%lx` ou `%lX`
réel simple précision (`float`)	`%f`
réel simple précision en notation scientifique	`%e` ou `%E`
réel double précision (`double`)	`%lf`
réel double précision en notation scientifique	`%le` ou `%lE`
chaîne de caractères (`char []`)	`%s`
pointeurs (valeur en hexadécimal)	`%p`

5) Quelques particularités

```c
#include <stdio.h>

int main()
{
  int n;
  char car = 'm';
  printf(" car1= %c\t car2= %d\n", car, car);
  car -= 32;
  n = printf(" car1= %c\t  car2= %d", car, car);
  printf("\n n= %d", n);
  return 0;
}

/*
 car1= m    car2= 109     la valeur ASCII de m
 car1= M    car2=  77     109- 32= 77 valeur ASCII de M
 n= 19                    longueur de la zone d'affichage  */
```

- Le code `%c` qui n'affiche qu'un seul caractère utilise pourtant deux octets comme pour `%d`. On retrouve l'équivalence entre les caractères et leur représentation ASCII.
- Quand le code format n'est pas conforme avec le type de la variable, l'affichage est incorrect.
- Si vous donnez à la fonction `printf()` un nombre de codes de format inférieur à celui des arguments, la priorité reste au format et les arguments en trop sont ignorés. Si vous n'avez pas donné suffisamment d'arguments, la machine affiche n'importe quoi pour les formats en trop!
- Comme souvent avec les fonctions standards, `printf()` nous renvoie une valeur: `n= printf(...);` où `n`: nombre de CARACTERES enregistrés avec succès

4.4. Lire avec la fonction `scanf()`

La fonction `scanf()` permet la saisie au clavier d'informations qui seront affectées à une variable. Elle « lit » donc pour le système, des informations qui y entrent.

4.4.1. Saisie de valeurs numériques.

Pour établir un dialogue avec une machine on combine l'utilisation de printf() et de scanf().
Exemple : **Utilisation de scanf() avec 1 variable par fonction**

```
#include <stdio.h>

int main(void)
{
  int entier0;
  double reel0;

  /* Saisie d'une seule variable. */
  printf(" Entrez un entier du domaine des int: ");
  scanf("%d", &entier0); /* &entier: adresse de entier0.*/
  printf(" Entrez un nombre réel: ");
  scanf("%lf", &reel0);  /* &reel: adresse de reel0. */
  /* contrôle: */
  printf(" entier0= %d\t reel0= %lf", entier0, reel0);
  return 0;
}

/*
 Entrez au clavier un entier du domaine des int: 123
 Entrez au clavier un nombre réel:  45.67
 entier0= 123     reel0= 45.670000                          */
```

1) Notez que les paramètres utilisés sont les mêmes que ceux de printf(), %d pour des entiers et %lf pour des doubles.
2) Si la longueur du code format est plus grande que celle de la variable, il y aura empiètement sur la zone de mémoire qui suit avec des conséquences totalement imprévisibles.
3) On donne à la fonction l'adresse &entier0, &reel0 des variables. En l'absence du symbole &, la valeur lue sera écrite n'importe où.

Exemple : **Utilisation de scanf() avec plusieurs variables**

```
#include <stdio.h>

int main()
{
  int entier1, n;
  double reel1;

  /* Saisie de plusieurs variables. */
  printf("\n\n Entrez un entier et un réel"
       " séparés par un espace: ");
  n = scanf("%d %lf", &entier1, &reel1);
  /* n: nombre de valeurs entrées avec succès. */

  /* contrôle: */
  printf("\n n= %d", n);
  printf("\n entier1= %d\t reel1= %lf", entier1, reel1);
  return 0;
}

/*
 Entrez un entier et un réel séparés par un espace: 89 1.23
 n= 2
```

```
 Entier1= 89      Reel1= 1.230000
 */
```

1) Notez qu'il est possible d'introduire une chaîne de caractères à l'intérieur de scanf() entre deux codes formats. Ici cette chaîne est un simple espace => "%d %lf". La condition est que cette chaîne soit strictement reproduite au moment de la saisie, sinon il y a un blocage pur et simple de la fonction.
2) Nous avons mis en évidence une propriété de scanf(). Elle renvoie une valeur: n = scanf(...) ; n: nombre de VARIABLES enregistrées avec succès

4.4.2. Saisie de caractères ou de chaînes.

Exemple :

```
#include <stdio.h>

int main()
{
  char car, chaine[16];
  int n;

  /* Saisie d'un caractère: */
  printf(" Entrez un caractère: ");
  scanf("%c", &car);
  printf("    Ce caractère est: %c", car);

  /* Saisie d'une chaîne de caractères: */

  printf("\n\n Entrez une chaîne de moins de 16 caract.: ");
  n = scanf("%s", chaine);
  printf(" n= %d\t\t\t    Cette chaîne est: %s", n, chaine);
  return 0;
}

/*
 Entrez un caractère: Q
    Ce caractère est: Q

 Entrez une chaîne de moins de 16 caractères: Tintin Milou
 n=1                 Cette chaîne est: Tintin           */
```

1) La saisie d'un seul caractère est l'équivalent de ce que nous avons vu avec le couple getchar(), putchar().
2) Anticipons l'étude que nous ferons des chaînes de caractères.
   * En C, une chaîne est une séquence de caractères, placés dans une suite de zones contigus de mémoire, qu'on appelle un tableau.
   chaine[7] = "Tintin";
   initialise un tableau qui peut contenir une chaîne de 6 caractères, le 7ème est un \0, le caractère nul (différent du zéro), qui indique la fin du tableau.
   * Chacun de ces caractères est contenu dans une zone individualisée,
   chaine[0] = 'T', chaine[1] = 'i',..., chaine[5]= 'n', chaine[6]= '\0'.
   * Ces tableaux ont, entre autres, une particularité, leur nom est équivalent à une adresse. Le mot chaine est l'adresse du 1er terme: chaine équivaut à &chaine[0], c'est une adresse et donc un nombre, ce qui fait que &chaine n'a pas de sens.

* Vous aurez aussi remarqué que le nom `Milou` n'a pas été pris en compte. En effet, `scanf()` prend les caractères de la chaîne en compte jusqu'à ce qu'elle rencontre un espace, une tabulation ou `\n` le retour à la ligne. C'est ce qui s'est passé dans notre programme: « `Tintin` », puis un espace et `scanf()` a décroché !
L'ennui, c'est que « `Milou` » est resté dans le « tampon » !

Exemple : **Mise en évidence du tampon avec** `scanf()`

```
#include <stdio.h>

int main()
{
  char chaine0[16], chaine1[16];
  int n;

  printf("\n\n Entrez une chaîne de moins de 16 caract.: ");
  n = scanf("%s", chaine0);
  printf(" n= %d\t\t\t Cette chaîne est: %s", n, chaine0);
  scanf("%s", chaine1);
  printf("\n Cette chaîne est: %s", chaine1);
  return 0;
}

/*
 Entrez une chaîne de moins de 16 caract.: Tintin Milou
 n=1                  Cette chaîne est: Tintin
  Cette chaîne est: Milou    */
```

Nous avons un peu modifié le programme en ajoutant `chaine1` et un nouveau `scanf()`. Exécutez le programme et vous verrez que le nom de `Milou` est allé TOUT SEUL dans `chaine1` ! Il était bien dans le « tampon » !

4.4.3. Qu'est-ce que ce « tampon » ?

Le mécanisme de cette fonction est caractérisé par l'existence d'un tampon, une zone de mémoire intermédiaire, entre le clavier et l'unité centrale.

1) A l'appel de la fonction `scanf()`, le tampon intermédiaire est interrogé. Si le tampon est vide, le clavier est sollicité pour fournir des caractères. S'il n'est pas vide, c'est son contenu qui est proposé!
2) La séquence de caractères entrés au clavier est stockée dans le tampon jusqu'à la réception d'un `\n`.
3) Les caractères sont alors retirés du tampon, convertis dans le format de la variable, puis affectés à cette variable dont on aura fourni l'ADRESSE.
4) La fonction renvoie une valeur qui est le nombre d'enregistrements réalisés avec succès.
5) MAIS cette affectation ne se fera que si le type de la variable correspond bien à celui annoncé dans la fonction. Sinon l'erreur restera dans le tampon et sera proposée à toutes les demandes suivantes de caractères pour d'autres `scanf()` avec les conséquences désastreuses que vous pouvez imaginer.
6) Quand le tampon est vide, il y a de nouveau interrogation du clavier.

IL N'Y A PAS DE VERIFICATION au cours de l'exécution de la fonction.

4.4.4. Quelques règles de sécurité

I. Ne saisir qu'une variable par fonction `scanf()`
(mais il est parfois nécessaire de transgresser la règle).
II. Vider le tampon après un appel de `scanf()`.
III. Contrôler la présence du `&` pour les variables dont le nom n'est pas déjà par lui même équivalent à une adresse (tableaux).
IV. Vérifier la compatibilité entre le code formaté et la variable.

4.5. Quelques compléments utiles

4.5.1. Vider le tampon après un appel à `scanf()`

Il suffit de demander `getchar()` de lire un à un les caractères qui seraient encore dans le tampon jusqu'à obtenir le `\n` de fin de ligne:

```
scanf(...);
while (getchar() != '\n');
```

« Tant que le caractère lu par `getchar()` est différent de `\n`, je passe au suivant. »

4.5.2. Saisie de chaînes où figurent des espaces

Il est possible de remplacer le symbole `s` par une séquence de caractères entre crochets, redéfinissant à notre usage les caractères autorisés pour `scanf()`.
Exemple : **Redéfinition des caractères autorisés pour `scanf()`**

```
#include <stdio.h>

int main()
{
  char chaine0[16], chaine1[16];
  printf("\n\n Entrez une chaîne de moins de 16 caract.: ");
  scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", chaine0);
  /* entre crochets: les majuscules + l'espace en 1ère
     position. */

  printf("\t\t\t    Cette chaîne est: %s", chaine0);
  scanf("%s", chaine1);
  printf("\n Cette chaîne est: %s", chaine1);
  return 0;
}

/*
 Entrez une chaîne de moins de 16 caract.: OBELIX Idefix
                      Cette chaîne est: OBELIX I
 Cette chaîne est: defix                                  */
```

1) A la demande du programme, entrez la chaîne `OBELIX Idefix` qui mêle majuscules, minuscules et un espace. Cette chaîne doit contenir moins de 16 caractères pour être conforme à la déclaration de `chaine0[16]`. Le 16ème est `chaine[15]='\0'` le caractère nul qui termine la chaîne.

2) On autorise les majuscules et l'espace qui est le premier caractère entre les crochets. Dès que la fonction rencontre un caractère non autorisé elle décroche ; `chaine0` contient : `OBELIX I`, avec un espace entre les noms.
3) Les caractères qui sont restés dans le tampon sont repris par le second `scanf()` ; `chaine1` contient la suite: `defix` en minuscules.

Exemple : **Vidage du tampon**

```
#include <stdio.h>

int main()
{
  char chaine0[16], chaine1[16];
  printf("\n\n Entrez une chaîne de moins de 16 caract.: ");
  scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", chaine0);
  while(getchar() != '\n');
  /* vide le tampon:
     Tant que le caractère lu par getchar() est
     différent de \n,
      je l'élimine et continue ma lecture */

  printf("\t\t\t    Cette chaîne est: %s\n", chaine0);
  scanf("%s", chaine1);
  printf("\n Cette chaîne est: %s", chaine1);
  return 0;
}

/*
 Entrez une chaîne de moins de 16 caract.: OBELIX Idefix
                      Cette chaîne est: OBELIX I          */
```

On a ajouté une ligne de code après `scanf()` pour vider le tampon: Puisque le tampon est vide, `scanf()` sollicite maintenant le clavier pour de nouveaux caractères. Vous pouvez saisir un texte dans `chaine1`.

Exemple : **Redéfinition des caractères avec** `scanf()`

```
#include <stdio.h>

int main()
{
  char chaine0[16];
  printf("\n\n Entrez une chaîne de moins de 16 caract.: ");
  scanf("%[^\n]", chaine0);
  while (getchar() != '\n');
  printf("\t\t\t    Cette chaîne est: %s", chaine0);
  return 0;
}

/*
 Entrez une chaîne de moins de 16 caract.: Diderot Iris
                      Cette chaîne est: Diderot Iris    */
```

Plaçons entre les crochets l'expression `[^\n]`. Elle signifie que tous les caractères ASCII sont autorisés sauf `\n` le retour la ligne. On retrouve dans l'expression le symbole de négation `^`. Il est maintenant possible de saisir un texte quelconque jusqu'au retour la ligne.

Il existe dans la bibliothèque `stdio.h` une fonction d'utilisation très générale qui permet de vider les tampons associés aux flux de données. Elle s'applique avec `scanf()` au tampon associé au flux de données que vous avez composé au clavier et qui entre dans la mémoire.
Ce flux se nomme `stdin`, flux d'entrée (*in*) standard (*std*).
On peut donc remplacer :

```
scanf(...);
while( getchar() != '\n');
```

par :

```
scanf(...);
fflush(stdin); // Vide le tampon associé au flux d'entrée
```

4.5.3. Limitation du nombre de caractères saisis par `scanf()`

On utilise un « spécificateur de largeur ».

```
scanf("%20s", chaine);
```

La fonction ne lira que 20 caractères avant de les affecter à la chaîne.

```
scanf("%20[^\n]", chaine);
```

est également correct.

Pour fiabiliser la saisie avec `scanf`, voici un modèle universel :

```
while (1) {
   fgets(tampon, taille, stdin);
   ret = sscanf(tampon, "%c", &c);
   if (ret != 1)
      /* instructions à exécuter en cas d'erreur */
   if ((strlen(tampon) == taille-1) &&
       (tampon[taille-2] != '\n'))
      do c = getchar(); while (c != '\n');
}
```

5. STRUCTURES DE CONTRÔLE

5.1. Branchements conditionnels

Notions : `if`, `else`, `switch`.
Dans la vie, on a souvent besoin de décider du déroulement des choses en fonction de certaines conditions. Par exemple : « S'il pleut, je prend mon parapluie » (ceci n'a d'ailleurs de sens que si je souhaite sortir !). De même en C, il est possible de décider de n'exécuter une action que si une condition particulière est remplie.
Pour tester si une condition est vraie avant d'exécuter une action, on dispose d'opérateurs de comparaison : `==` `!=` `<` `<=` `>=` `>` et d'opérateurs logiques `!` `||` `&&`. Une expression utilisant ces opérateurs retourne 0 si la condition testée est fausse et 1 si elle est vraie. Une condition peut consister en une expression. Dans ce cas, si la valeur renvoyée par l'expression égale zéro, la condition est fausse. Elle est vraie pour toute autre valeur.

5.1.1. La structure de contrôle `if`

Si la condition de la structure `if` est vraie, on peut exécuter une instruction unique ou bien un ensemble d'instruction entre accolades. Dans ce dernier cas, attention, on ne met pas de `;` après l'accolade fermante !

```
if (condition)     /* Si la condition est vraie... */
   instruction;  /* ...on exécute une instruction unique */

if (condition) /* Si la condition est vraie... */
{              /* ...on exécute un bloc d'instructions */
   instruction;
   ...
   instruction;
}
```

Exemples :

```
if (n > 20)
    printf("Vous avez droit au tarif de groupe\n");
```

```
if (x == 3) {
    printf("une instruction !\n");
    printf("une autre instruction !\n");
}
```

5.1.2. La structure de contrôle `if...else`

L'instruction `else` permet d'exécuter une instruction au cas où la condition est fausse. C'est l'équivalent du « sinon ». Par exemple : « s'il fait beau, je pars à la campagne, sinon je reste à la maison ».

```
if (condition)
```

```
    instruction;
 else
    instruction;
```

Exemple :

```
if (choix == 'o' || choix == 'O')
    printf("Je pars à la campagne.\n");
else
    printf("Je reste à la maison.\n");
```

5.1.3. La structure de contrôle `switch`

Remplace les **`if`**...**`else`** les uns à la suite des autres.

```
 switch (expression) {
    case constante : instructions;
    case constante : instructions;
    ...
    default : instructions;
 }
```

Exemple :

```
char choix;
...
switch (choix) {
    case 't' : printf("vous voulez un triangle"); break;
    case 'c' : printf("vous voulez un carre"); break;
    case 'r' : printf("vous voulez un rectangle"); break;
    default  : printf("erreur. recommencez !");
}
```

L'instruction **`break`** est indispensable pour sortir du **`switch`**. Si on l'oublie, une fois le branchement exécuté, toutes les instructions qui suivent sont exécutées.

5.2. Les répétitions

Notions : **`while`**, **`for`**, **`do`**..**`while`**, **`break`**, **`continue`**.

5.2.1. La boucle `while`

L'instruction **`while`** permet d'exécuter un bloc d'instructions tant qu'une condition est vraie.

```
 while (condition) {
    instructions;
 }
```

Exemple :

```
int i=0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

5.2.2. La boucle `for`

```
for (initialisation ; condition ; opération) {
   instruction;
}
```

Voici ce qui se passe :

- La première fois qu'on rentre dans la boucle, on effectue généralement une initialisation (par exemple, `i=0`).
- On teste ensuite la condition. Si elle est vrai, on effectue la ou les instructions du corps de la boucle. Sinon, on sort de la boucle (la boucle est terminée).
- Après cela, on effectue une opération qui est en principe l'incrémentation d'un compteur.
- On re-teste la condition, etc..

Exemple :

```
int i;
for (i=0; i<10; i++)
   printf("%d\n", i);
```

Il est possible de procéder à plusieurs initialisations et à plusieurs opérations en les séparant par des virgules. Pour procéder à plusieurs tests, il faut utiliser les connecteurs logiques `&&` ou `||` :

```
int i, j;
for (i=0, j=1 ; i<10 && j<100 ; i++, j=j*2)
   printf("%d %d\n", i, j);
```

5.2.3. La boucle `do...while`

C'est une variante de la boucle **`while`** vue précédemment avec cette fois-ci l'obligation de passer au moins une fois dans la boucle puisque la condition n'est testée qu'à la fin..

```
do {
   instruction;
} while (condition);
```

5.2.4. Les instructions `break` et `continue`

Les instructions **`break`** et **`continue`** permettent de sortir d'une itération (cf. **`switch`**).
Exemple :

```
int i = 0;
while (1) { /* la boucle est infinie : toujours vraie */
   printf("%d\n", i);
   i++;
   if (i > 10) /* La sortie se fait pour n > 10 */
      break;
}
```

5.2.5. Comment faire `n` fois quelque chose

Chaque boucle du code suivant affiche 10 fois la même phrase :

```
/* en utilisant 'while' */
int i = 0;
```

```
while (i < 10) {
   printf("Bonjour à tous !\n");
   i++;
}

/* exactement la même chose en utilisant 'for' */
for (i = 0; i < 10; i++)
   printf("Bonjour à tous !\n");
```

On remarque que `i` est, par convention d'écriture, toujours initialisé à zéro.

5.2.6. Les pièges infernaux des boucles

Certaines erreurs sont très courantes quand on débute la programmation en C. En voici un certain nombre.
Ici, `i` n'est pas initialisé, ce qui rend le déroulement de l'itération imprévisible :

```
/* oups ! on a oublie d'initialiser 'i' */
while (i < 10) {
   printf("Bonjour à tous !\n");
   i++;
}
```

Une autre erreur possible, se tromper dans la comparaison. Ici, la condition ne sera jamais vraie, donc la boucle ne sera pas exécutée :

```
int i = 0;
while (i > 10) {    /* on a mis '>' ou lieu de '<' */
   printf("Bonjour à tous !\n");
   i++;
}
```

Attention, l'erreur la plus fréquente est sans doute d'oublier d'incrémenter `i`, il en résulte une boucle infinie :

```
int i = 0;
while (i < 10) {
       printf("Bonjour à tous !\n");
}
```

Attention aux '`;`' en trop. La boucle suivante ne fait rien pendant 10 fois puis affiche la phrase « `Bonjour à tous !` » une seule fois :

```
int i;
for (i = 0; i < 10; i++);
   printf("Bonjour à tous !\n");
```

Par ailleurs, attention à la syntaxe des boucles **`for`**. A l'intérieur du **`for`**, on sépare les parties d'initialisation, de test et d'opération par des '`;`'. Attention donc à ne pas confondre avec les '`,`' :

```
int i;
for (i = 0, i < 10, i++)  /* ERREUR : des ',' au lieu
                             des ';' */
   printf("Bonjour à tous !\n");
```

6. LES FONCTIONS

Notions : déclaration, prototype, arguments, valeur renvoyée, `return`, `void`, passage par valeur.

Jusqu'à présent, on a utilisé des fonctions prédéfinies (`printf()`, `scanf()`, etc.). Il est aussi possible d'écrire ses propres fonctions.

6.1. Qu'est-ce qu'une fonction ?

Un regroupement d'instructions
Une fonction est un regroupement d'instruction. Imaginons un robot Lego™ qui sait seulement se déplacer tout droit et tourner à droite ou à gauche. On lui apprend à dessiner un carré en la faisant aller tout droit, puis tourner à droite, puis continuer tout droit, etc, quatre fois. Ces actions (en programmation, on parle d'instructions) peuvent être regroupées au sein d'une procédure que l'on appellerait « dessiner_un_carre ». Ainsi, quand on dirait au robot « dessiner_un_carre », il effectuerait les instructions de la procédure les unes à la suite des autres et dessinerait un carré.
Des arguments pour des résultats différents
Une fonction peut prendre des arguments. En reprenant la métaphore précédente, on pourrait apprendre au robot à dessiner un carré d'une taille spécifiée. On dirait par exemple celui-ci « dessiner_un_carre de 18 cm ». On a là un exemple de fonction qui prend un argument.
Des fonctions qui renvoient un résultat
Une fonction peut retourner une valeur. On peut très bien imaginer une boite noire dans laquelle on peut insérer des jetons rouges par une fente et des jetons bleus par une autre. Selon le nombre de jetons insérés, la boite nous renvoie un certains nombre de jetons noirs. Nous avons là une fonction, la boîte, qui prend des arguments, les jetons de couleur, et qui renvoie une valeur, des jetons noirs.

6.2. Prototype d'une fonction

Le prototype d'une fonction permet de connaître le nombre et le type de ses arguments. Il nous donne aussi une idée de ce qu'elle renvoie.
Exemple :

```
float taxe(float);
int putchar(int); /* Fonction prédéfinie */
```

On peut souhaiter créer une fonction qui ne prend aucun argument et qui ne renvoie pas de valeur. En C, `void` est un type spécial qui signifie « rien ». Il est utilisable à la place des arguments et de la valeur renvoyée.
Exemple :

```
void afficher_menu(void);
int getchar(void); /* Fonction prédéfinie */
```

6.3. Définition d'une fonction

Une fonction est définie de la façon suivante :

```
type nom(type var1, type var2, ..., type varn)
```

```
{
   instruction;
   ...
   return valeur;
}
```

Une fonction doit se terminer par l'instruction **return** pour se terminer. Cette instruction **return** permet de retourner une valeur.
Par exemple, la fonction suivante prend en argument deux entiers et renvoie leur somme :

```
int additionner(int a, int b)
{
   int c;
   c = a + b;
   return c;
}

int main()
{
   int x = 2, y;
   y = additionner(x, 5)
   return 0;
}
```

L'appel à la fonction `additionner()` fonctionne en plusieurs étapes :

1. Dans le `main`, on fait appel à la fonction `additionner()` qui prend en argument la valeur de `x` et la valeur `5`.
2. Ces valeurs sont copiées dans les variables `a` et `b` définies dans l'en-tête de la fonction.
3. La fonction calcule et retourne la valeur de `c`.
4. On revient dans le `main()`. La fonction est remplacée par la valeur qu'elle retourne. Cette valeur est mise dans `y`.

6.4. Visibilité des variables dans un programme

Les variables ne sont pas visibles **en dehors** de la fonction où elles ont été définies.
Quand je suis dans une fonction, je n'ai accès qu'**aux variables de cette fonction**, c'est tout !
La fonction **`main()`** ne fait pas exception à la règle.

Pour comprendre cela, on peut prendre la métaphore suivante. Une fonction est comme une boite noire avec des fentes pour y insérer des jetons. Examinons comment la boite est faite.
Quand on insère des jetons, ceux-ci sont recueillis dans des petits casiers étiquetés qui portent un nom. En informatique, ces casiers sont appelés « variables ». Ces casiers peuvent avoir n'importe quel nom, il n'y a aucune confusion possible avec les casiers en dehors de la boite.
Par ailleurs, moi qui suit en dehors de la boite, je ne peux pas manipuler les casiers dans la boite. De la même façon, il y a dans la boite un petit rat qui fait tourner une roue. Et bien ce petit rat ne peut manipuler que les casiers qui sont dans sa boite.
Si une variable peut être définie en dehors de toute fonction, on dit alors d'elle qu'elle est « globale » ou « externe ». Une telle variable est visible de toutes les fonctions, à n'importe quel endroit du

programme.

6.5. Quelques exemples de fonctions

Cette fonction affiche n fois une lettre. Attention, putchar() prend en argument un int :

```
void afficher_car(int c, int n)
{
   while (n--)
      putchar(c);
   /* return implicite */
}
```

Une fonction qui ne prend aucun argument et qui ne renvoie rien :

```
void afficher_bonjour(void)
{
   printf("Bonjour à tous !\n");
}
```

6.6. Déclaration d'une fonction

Si une fonction est définie dans le code source avant la portion de code qui l'utilise, il n'est pas nécessaire de la déclarer. Dans le cas contraire il faut la déclarer en mettant son prototype dans l'entête du programme (c'est ce qu'on fait d'une manière particulière avec les fichiers d'inclusion « .h » afin de pouvoir utiliser des fonctions prédéfinies).

Exemple :

```
void afficher_car(int, int); /* déclaration */

int main()
{
   char c = 'x';
   afficher_car('-', 10);    /* appel de la fonction */
   afficher_car((int) c, 5);
   return 0;
}

void afficher_car(int c, int n) /* définition */
{
   /* Fonction qui affiche n fois une lettre. */
   while (n--)
       putchar(c);
   /* return implicite */
}
```

6.7. Comprendre le passage par valeur

Une fonction ne prend en argument que des valeurs. Attention, si on passe en argument une variable, on passe en réalité une copie de sa valeur en argument. Une variable ou une expression compliquée passées en argument sont toujours remplacées par leur valeur :

```
#include <stdio.h>

int main()
{
```

```
    int a = 2, b = 3;
    printf("%d\n", 5);
    printf("%d\n", a);
    printf("(%d+%d)^2 = %d\n", a, b, (a*a+b*b+2*a*b));
    return 0;
}
```

Ce programme affiche :

```
./a.out
5
2
(2+3)^2 = 25
```

Le passage par valeur implique qu'on ne peut changer la valeur d'une variable passée en argument. Par exemple, dans le programme suivant, la fonction ne modifie pas la valeur de `a` :

```
#include <stdio.h>

void mafonction(int x)
{
    x = 0;
}

int main()
{
    int a = 1;
    mafonction(a);  /* valeur de 'a' n'est pas modifiée */
    printf("%d", a); /* affiche 1 */
    return 0;
}
```

6.8. Comprendre la notion de valeur retournée

L'instruction **return**, qui termine toujours une fonction, permet de retourner une valeur. Cela signifie que dans un programme, une fonction est remplacée par la valeur qu'elle renvoie.
Par exemple, dans le programme suivant, l'appel à la fonction `additionner()` est tout à fait autorisé au sein d'une expression mathématique ou dans une fonction :

```
#include <stdio.h>
int additionner(int, int);

int main()
{
    int x = 1, y = 2, z;
    /* affecte a 'z' la valeur retournée par la fonction */
    z = additionner(x+y);

    /* possible aussi */
    printf("%d\n", additionner(x+y));

    /* également possible ! */
    z = additionner(additionner(x+3), 5);

    return 0;
}
```

```
int additionner(int a, int b)
{
   return (a + b);
}
```

6.9. Erreurs courantes

Oublier de déclarer un nom de variable dans la définition d'une fonction :

```
int additionner(int, int) /* oubli des noms de variables */
{
   int c;
   c = a + b;
   return c;
}
```

Il y a aussi les ';' en trop :

```
int additionner(int a, int b);  /* un ';' en trop !!! */
{
   return a + b;
}
```

Une autre erreur très fréquente est de redéclarer les noms de variables :

```
int additionner(int a, int b)
{
   int a, b; /*  variables déjà déclarées au dessus */
   return a + b;
}
```

Si une fonction n'est pas de type **void**, attention à bien retourner une valeur :

```
int additionner(int a, int b)
{
   int c;
   c = a + b;
   return; /* on oublie de retourner un entier ! */
}
```

6.10. Récursivité

Une fonction **récursive** est une fonction qui **s'appelle elle-même**. La récursivité permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.
La récursivité utilise toujours la **pile** du programme en cours.
On appelle **pile** une zone mémoire réservée à chaque programme ; sa taille peut être fixée manuellement par l'utilisateur. Son rôle est de stocker les variables locales et les paramètres d'une fonction. Supposons que nous sommes dans une fonction `fonc1()` dans laquelle nous avons des variables locales. Ensuite nous faisons appel à une fonction `fonc2()` ; comme le microprocesseur va commencer à exécuter `fonc2()` mais qu'ensuite il reviendra continuer l'exécution de `fonc1()`, il faut bien stocker quelque part les variables de la fonction en cours `fonc1()` ; c'est le rôle de la pile !

Tout ceci est géré de façon transparente pour l'utilisateur. Dans une fonction récursive, toutes les

variables locales sont stockées dans la pile, et empilées autant de fois qu'il y a d'appels récursifs. Donc la pile se remplit progressivement, et si on ne fait pas attention on arrive à un « débordement de pile ». Ensuite, les variables sont désempilées (et on dit « désempilées », pas « dépilées »).

Une fonction récursive comporte un appel à elle-même, alors qu'une fonction non récursive ne comporte que des appels à d'autres fonctions.

Toute fonction récursive comporte une instruction (ou un bloc d'instructions) nommée « point terminal » ou « point d'arrêt », qui indique que le reste des instructions ne doit plus être exécuté.

Exemple :
Pour calculer un nombre `a` à la puissance `b` nous pouvons utiliser la fonction itérative classique suivante :

```
int puissance(int a, int b)
{
  int r = 1;        /* déclaration d'un entier r */
  while (b > 0)     /* tant que b est supérieur à 0 */
  {
     r = r * a;     /* 1 étape de a * a * ... * a, b fois */
     b = b - 1;     /* une multiplication de moins à faire */
  }
  return r;
}
```

Et si nous utilisons la récursivité, la fonction devient :

```
int puissance(int a, int b)
{
  if (b > 0)
    return a * puissance(a, b - 1);
  else
    return 1;
}
```

7. LES TABLEAUX

Notions : définition, initialisation, affectation, indices de `0` à `n-1`, débordements, opérateur crochets [ ], tableaux à `n` dimensions, conversion des noms de tableaux en pointeurs, passage de tableaux en argument, pointeurs de tableaux.

7.1. Déclaration et initialisation

Un tableau permet de regrouper plusieurs données de même type en une entité. Les tableaux en C se déclarent avec l'opérateur crochets `[ ]`. Tout comme avec les variables simples, il est possible d'initialiser un tableau lors de sa déclaration en mettant les valeurs du tableau entre accolades.

Exemples de déclarations de tableaux :

```
/* déclaration d'un tableau de 10 caractères */
char tab[10];

/* déclaration et init. d'un tableau de 3 entiers */
int a[3] = {1, -1, 0};

/* déclaration et init. d'un tableau de 4 caractères */
char msg[] = {'a', 'b', 'c', '\0'};
```

7.2. Affectation

On accède à un élément d'un tableau à l'aide de son indice entre crochets. Attention, le premier élément a pour indice `0` et le dernier `n-1`. Le (n+1)ième élément s'écrit donc `tab[n]`.
Les éléments d'un tableau peuvent être affectés lors de la déclaration de celui-ci. Il n'est pas possible d'affecter directement un tableau à un autre tableau. On ne peut affecter un tableau qu'élément par élément.
Exemple :

```
tab[i] = val;
```

Attention, l'exemple suivant montre une affectation correcte et une autre incorrecte :

```
int tab[5];
int tab2[5] = {1, 2, 3, 5, 8}; /* ok */
tab = tab2; /* incorrect */
```

De la même manière, il est incorrect d'utiliser les opérateurs de comparaison pour comparer deux tableaux.
Affectation, comparaison... toutes ces opérations ne peuvent être réalisées qu'élément par élément. L'explication de tout cela est qu'un nom de tableau utilisé dans une expression est converti en pointeur sur le premier élément de ce tableau. Ainsi, l'expression `tab1 == tab2` revient à comparer les adresses en mémoire où sont implantés ces tableaux.
Attention, une fonction ne peut jamais renvoyer un tableau.

Exemple : initialisation de tous les membres d'un tableau à 0

```
for (i = 0; i < 100; i++) t[i] = 0;
```

7.3. Les débordements

Attention, quand on affecte un élément d'un tableau, il n'y a aucun contrôle fait pour savoir si on déborde ou non du tableau. Voici un exemple de débordement :

```
int tab[5];
tab[5] = 0; /* débordement : les indices vont de 0 a n-1 */
```

Une telle erreur ne sera pas repérée par le compilateur et ne se manifestera qu'à l'exécution par un message du type « **Bus error** » ou « **Segmentation fault** ».

7.4. Passage en argument de fonctions

Il est tout à fait possible de passer des tableaux en argument d'une fonction. Mais attention, un tableau n'est pas une valeur et le passage en argument est très particulier. Au niveau de la fonction tout d'abord. La déclaration de l'argument se fait en rajoutant des crochets pour indiquer que l'argument est un tableau.
Exemple :

```
void mafonction(int[]); /* le prototype */

void mafonction(int tab[]) /* la fonction */
{
   /* code de la fonction ... */
   return;
}
```

Au niveau de l'appel de la fonction, on passe juste en argument le nom du tableau.
Exemple :

```
int main()
{
   int t[6];
   mafonction(t);
   return 0;
}
```

Attention, le passage de tableaux en argument présente des particularités.

- Quand on est dans la fonction, on n'a aucun moyen de savoir quelle est la taille du tableau passé en argument. Si on a besoin de savoir quelle est cette taille, il faut la passer par un deuxième argument.
- Un tableau est passé non par valeur mais par « adresse ». Cela signifie que si le tableau est modifié dans la fonction, alors, comme par magie, le tableau passé en argument est réellement modifié ! Bon, ça n'est pas vraiment de la magie, mais nous verrons cela plus tard.

7.5. Les tableaux à plusieurs dimensions

Exemple :

```
int matrice[2][3]; // 2 lignes et 3 colonnes d'entiers
matrice[0][0] = 1;
```

```
matrice[0][1] = 2;
...
```

L'initialisation lors de la déclaration peut se faire de la manière suivante :

```
int matrice[2][3] = { 1, 2,  3,  4, 5, 6 };
```

ou encore :

```
int matrice[2][3] = { { 1, 2, 3 } ,
                      { 4, 5, 6 } };
```

La manipulation d'un tableau à plusieurs dimensions peut se faire par une imbrication de boucles. Par exemple, l'obtention d'une matrice identité[1] 3x3 peut se faire de la manière suivante :

```
int matrice[3][3];
int ligne, colonne;
for (ligne = 0; ligne < 3; ligne++)
  for (colonne=0; colonne < 3; colonne++)
    matrice[ligne][colonne] = 0;
for (ligne = 0; ligne < 3; ligne++)
  matrice[ligne][ligne] = 1;
```

La même chose en plus rapide :

```
int matrice[3][3];
int ligne, colonne;
for (ligne = 0; ligne < 3; ligne++)
  for (colonne=0; colonne < 3; colonne++)
    matrice[ligne][colonne] = (ligne == colonne) ? 1 : 0;
```

Ou encore :

```
int matrice[3][3];
int ligne, colonne;
for (ligne = 0; ligne < 3; ligne++)
  for (colonne=0; colonne < 3; colonne++)
    matrice[ligne][colonne] = ligne == colonne;
```

Remarque : Attention la réduction de deux boucles imbriquées en une seule n'est pas possible.

1 Matrice qui ne comporte des 1 que sur sa diagonale et des 0 partout ailleurs.

8. CHAÎNES DE CARACTÈRES

Notions : chaînes de caractères, \0, fonctions de manipulation de chaînes (strlen(), strcpy(), strcmp(), etc.).

8.1. Définition

Une chaîne de caractère est tout simplement une suite de caractères stockés dans un tableau. Une chaîne doit impérativement se terminer par le caractère \0.
Une chaîne littérale correspond à une suite de caractères entre guillemets.

```
char ch[] = "Bonjour"; /* tab. de 7 caractères avec '\0' */
char ch2[8] = "Bonjour"; /* plus risqué ! */
```

Attention les chaînes littérales sont assimilées à des constantes. Le code suivant est incorrect :

```
char ch[] = "Bonjour";
char ch[3] = 'X'; /* incorrect */
```

8.2. Fonctions de manipulation de chaînes

8.2.1. Afficher une chaîne

On utilise la fonction printf() avec l'indicateur de format %s et on met en argument le nom du tableau qui contient la chaîne :

```
char ch[] = "Bonjour";
printf("%s à tous !\n", ch);
```

8.2.2. Saisir une chaîne

On utilise la fonction scanf() avec l'indicateur de format %s et on met en argument le nom du tableau qui contient la chaîne. Attention, les tableau sont des cas particuliers et il n'est pas besoin de mettre le caractère & devant le nom :

```
char ch[512];
scanf("%s", ch);
```

8.2.3. Copier, comparer et mesurer

Une chaîne est un tableau, on ne peut donc réaliser directement une affectation ou une comparaison. Pour copier une chaîne dans un tableau, on utilise la fonction strcpy() :

```
char ch[] = "Bonjour";
char ch2[512];
strcpy(ch2, ch);
printf("%s\n", ch2);
```

Pour comparer, on utilise la fonction strcmp(). Une autre fonction utile est la fonction strlen()

qui sert à mesurer la longueur d'une chaîne (caractère nul non compris).

9. LES POINTEURS

Notions : adresse mémoire, opérateurs `*` et `&`, passage d'arguments par adresse, pointeurs de pointeurs de ..., types `void*` (pointeurs génériques), `const int *p` et `int * const p`.

9.1. Définition

Les pointeurs sont des variables qui ont pour valeur une « adresse mémoire ». On les utilise surtout lors du passage d'arguments à une fonction ou lors d'allocation dynamique de mémoire.

9.2. Déclaration

On déclare un pointeur d'un type donné en ajoutant le signe `*` avant le nom du pointeur :

```
char *ptr;
int* ptr2; /* possible aussi ! */
```

9.3. Les opérateurs `&` et `*`

L'opérateur `&` sert à récupérer l'adresse d'une variable. Exemple :

```
int var = 4;
int *ptr; /* ptr pointeur sur 'int' */
ptr = &var; /* 'ptr' pointe sur la variable 'var' */
```

L'opérateur `*` sert à accéder au contenu de la case mémoire pointée. Un pointeur est une adresse en mémoire, et grâce à l'opérateur `*`, on peut accéder à ce qu'il y a à cette adresse (soit pour modifier ce qui y est stocké, soit pour voir son contenu) :

```
int var = 5;
int *p;
p = &var; /* 'p' pointe sur 'var' */
printf("%d\n", *p);
*p = 4;    /* 'var' est modifie ! */
printf("%d\n", *p); /* on récupère la valeur */
```

Il est possible de pointer sur l'élément d'un tableau :

```
int tab[5];
int *ptr;
ptr = &tab[4]; /* pointe sur le dernier élément de tab */
```

9.4. Manipulation de pointeurs

Les pointeurs peuvent être incrémentés, décrémentés, additionnés ou soustraits. Dans ce cas, leur nouvelle valeur dépend de leur type. Incrémenter un pointeur de `char` ajoute 1 à sa valeur. Incrémenter un pointeur de `int` ajoute 2 ou 4 (cela dépend de l'architecture).
Tout comme avec les tableaux, il est possible d'utiliser les crochets pour accéder au contenu d'un élément pointé.

Les écritures suivantes sont équivalentes :

```
int tab[5], *p = tab;
*(p+1) = n;
p[1] = n;    /* identique à la ligne précédente */
```

9.5. Pointeurs, tableaux et chaînes littérales

Il est possible de pointer sur un élément particulier d'un tableau :

```
int tab[5];
int *ptr;
ptr = &tab[2]; /* pointe sur le 3eme élément du tableau */
```

Un nom de tableau utilisé dans une expression est converti en une adresse du début de ce tableau :

```
int tab[5];
int *ptr;
ptr = tab; /* équivalent à : "ptr = &tab[0]" */
```

Quand une chaîne littérale est utilisée dans une expression, comme pour un tableau, elle est convertie en une adresse sur son début :

```
int *ptr = "abcd";
printf("%s\n", ptr);
```

Attention, les chaînes littérales sont assimilées à des constantes et ne peuvent pas être modifiées.

9.6. Pointeurs génériques

Les pointeurs de type `void *` sont utilisés pour pointer sur quelque chose dont on ne connaît pas le type. Les seuls opérateurs autorisés avec les pointeurs génériques sont :

- l'affectation : =
- la conversion de type

Les autres opérateurs sont interdits. Parmi eux :

- l'indirection : `*p`
- l'addition : `p + i`
- la soustraction : `p - i`

9.7. Une utilisation des pointeurs : le passage par adresse

Le passage d'arguments par adresse ne peut se faire qu'avec des pointeurs. Il permet de changer la valeur d'une variable dont l'adresse est passée en argument de la fonction :

```
void incrementer(int *n)
{
   *n = *n + 1;
}

int main()
{
   int a = 3;
   incrementer(&a);
   printf("%d\n", a);
```

```
    return 0;
}
```

L'appel à la fonction incrementer() fonctionne en plusieurs étapes :

1. Dans le **main()**, on fait appel à la fonction incrementer() qui prend en argument l'adresse de la variable a (l'opérateur & retourne l'adresse d'une variable).
2. L'adresse de a est stockée dans la variable n de la fonction.
3. Grâce à l'opérateur *, on accède au contenu de la case pointée par n.
4. On revient dans le main().

Attention, nous avons vu que le nom d'un tableau utilisé seul dans une expression, était converti en un pointeur sur celui-ci. Cela signifie qu'un tableau est toujours passé par adresse à une fonction. Pour reprendre la métaphore, quand une fonction prend en argument un tableau, elle travaille toujours sur l'original.

9.8. Utilisation avancée

```
int *p; /* pointeur sur un entier */
int *t[10]; /* tableau de 10 pointeurs sur des entiers */
int (*pt) [5]; /* pointeur sur un tableau de 5 entiers */
int *fonc(); /* fonction renvoyant pointeur sur entier */
int (*pfonc) (); /* pointeur sur fonction retournant un
entier */
```

Et les listes chaînées...

10. PASSER DES ARGUMENTS À UN PROGRAMME

Notions : argc, argv, atoi().
Le *shell* permet de transmettre des arguments au lancement d'un programme. Un programme C est capable de récupérer ces arguments qui sont stockés sous forme de chaînes de caractères. Un tableau habituellement nommé argv qui doit être déclaré dans la fonction main() contient des pointeurs sur ces chaînes. L'entier argc indique le nombre d'arguments passés au programme dans le tableau argv. Attention, le premier argument est toujours le nom du programme lui-même (comme en *shell*).
Exemple :

```
int main(int argc, char *argv[])
{
   printf("%s\n", argv[0]); /* nom du programme lancé */
}
```

Convertir les arguments récupérés
La principale conversion est celle d'une chaîne de caractères en un nombre entier. On utilise pour cela la fonction atoi() :

```
char ch[] = "1234";
int n;
n = atoi(ch); /* 'n' a maintenant pour valeur 1234 */
```

11. Types de données composés et types dérivés

Notions : `struct`, `enum`, `union`, `typedef`, initialisation, déclaration, utilisation, opérateurs `.` et `->`.

11.1. Structures

11.1.1. Créer de nouveaux types de données

Très souvent, nous avons à manipuler des objets (au sens de « choses », qu'elles soient réelles ou abstraites) qui possèdent plusieurs caractéristiques. Il est possible d'associer ces caractéristiques pour créer un nouveau type de variable que l'on appelle structures.

11.1.2. Des représentations plus proches de la réalité : un code plus clair et un premier pas avant la programmation objet

Admettons que nous voulions réaliser une base de donnée de musiciens qui permettra de stocker et de manipuler plusieurs de leurs caractéristiques telles que leur nom, leurs dates de naissance et de décès. Admettons que nous souhaitions une base de 200 musiciens. Jusqu'à présent, la seule méthode que nous connaissons pour implémenter cela est de créer autant de tableaux qu'il y a de caractéristiques et de dire qu'à chaque indice correspond un musicien différent :

```
char nom[200][20];
unsigned short int annee_de_naissance[200];
unsigned short int annee_de_mort[200];
```

Cette implémentation présente l'inconvénient de produire un code source éclaté dont la structure ne reflète pas la réalité du problème. Les caractéristiques d'un musicien sont éclatées sur plusieurs tableaux. La création de structures permet heureusement une implémentation plus proche de la réalité en regroupant au sein d'une même entité, différentes données.
Par exemple, la structure `Musicien` :

```
struct Musicien
{
   char nom[20];
   unsigned short int naissance;
   unsigned short int mort;
};
```

11.1.3. Déclarer une structure

La déclaration d'une structure se fait par le mot réservé `struct` suivi d'une liste de déclaration de variables entre accolades. Attention car il y a un `';'` à la fin des accolades.
La syntaxe est :

```
struct Nom
{
```

```
    type champ1;
    type champ2;
    ...
    type champn;
};
```

Attention!

Les structures créent des trous en mémoire. Sur les architectures 16 bits et plus encore sur les architectures 32 puis 64 bits, la taille des trous augmente. En effet, les groupes de 2, 4 et 8 octets se placent à partir d'adresses mémoires paires. On se croirait dans la plaisanterie de Coluche sur les trous dans le gruyère : « *Plus il y a de gruyère et plus il y a de trous. Mais plus il y a de trous et moins il y a de gruyère!* ».

Exemple :

```
struct S {
   int x;
   char a;
   long y;
   char b;
   float z;
} alpha;
```

sera stockée en mémoire de la manière suivante sur une architecture 32 bits :

Champ	**x**		**a**				**y**		**b**		
Octets	4	4	4	4			8		1		
Adresse	$406	$407	$408	$409	$410	$411			$412		

Remarque : Si une option de compression des structures existent sur chaque compilateur, elle opère en général au détriment des performances de la machine.

11.1.4. Affectation et manipulation des structures

Il est possible d'affecter une structure lors de sa déclaration :

```
struct Musicien le_musicien = {"Bach", 1685, 1750};
```

Contrairement aux tableaux, il est possible d'utiliser l'opérateur '=' pour affecter une structure à partir d'une autre :

```
struct Musicien le_musicien = {"Bach", 1685, 1750};
struct Musicien l_autre_musicien;
l_autre_musicien = le_musicien;
```

Pour accéder aux membres de la structure, on utilise l'opérateur ' **.** ' concaténé au nom de la structure. Cela permet d'affecter une structure élément par élément :

```
struct Musicien le_musicien;
strcpy(le_musicien.nom, "Bach");
le_musicien.naissance = 1685;
le_musicien.mort = 1750;
printf("Le musicien s'appelle %s\n", le_musicien.nom);
```

Attention, on ne peut utiliser les opérateurs arithmétiques ou les opérateurs de comparaison sur des structures !

11.1.5. Les champs de bits

Les éléments d'une structure peuvent aussi être des portions d'entiers, ce qui permet de stocker plusieurs variables sur un seul entier (économie de place). On parle de champs de bits pour qualifier ces données plus petites. Par exemple, sur des ordinateurs fonctionnant en mode console sous Ms-Dos, avec des écrans permettant d'afficher 25 lignes de 80 caractères, on pouvait utiliser ce genre d'artifice pour manipuler les caractères affichés tout en respectant l'espace mémoire :

```
struct Caractere
{
    char code_ascii;
    unsigned int couleur : 3;
    unsigned int luminosite : 1;
    unsigned int couleur_fond : 3;
    unsigned int clignote : 1;
} ECRAN[25][80];
...
ECRAN[10][23].code_ascii = 'A';
...
ECRAN[10][23].clignote = 1;
```

Voir le chapitre suivant consacré au champs binaires.

11.1.6. Pointeurs de structures

Il est possible d'accéder aux membres d'une structure à partir d'un pointeur sur celle ci :

```
struct Musicien *ptr;
ptr = &le_musicien;

printf("Le musicien s'appelle %s\n", ( * ptr ).nom);
```

Mais l'écriture précédente est complexe. Une facilité est offerte par l'opérateur '->'. Ainsi, les deux écritures ci-dessous sont équivalentes :

```
( * ptr ).nom et ptr->nom
```

Pour reprendre notre exemple :

```
struct Musicien *ptr;
ptr  = &le_musicien;

printf("Le musicien s'appelle %s\n", ptr->nom);
```

11.2. Les énumérations

Une énumération permet de définir des constantes pour une liste de valeurs. Le compilateur assigne une valeur par défaut à chaque élément de la liste en commençant par 0 et en incrémentant à chaque fois.

Exemple :

```
enum Couleur { rouge, vert, bleu, jaune };
```

Il est aussi possible d'assigner explicitement des valeurs :

```
enum Couleur { rouge = 2, vert = 3,
               bleu = 1, jaune = 0 };
```

La variable s'utilise comme suit :

```
enum Couleur couleur_pixel;
couleur_pixel = rouge;
```

11.3. La directive `typedef`

Cette directive permet d'assigner un nouveau nom à un type de données. On l'utilise par commodité. Dans l'exemple suivant, on peut utiliser `Navire` à la place de `struct bateau` :

```
typedef struct Bateau Navire;
```

De même pour `COULEUR` à la place de `enum couleur` :

```
typedef enum Couleur {
   rouge = 2,
   vert = 3,
   bleu = 1,
   jaune = 0 } COULEUR;

COULEUR couleur_pixel = rouge;
```

11.4. Les unions

Dans les structures, une suite de membres occupait un espace contigu de mémoire. Dans les unions ce sont des membres différents qui occuperont successivement et selon vos besoins la même zone. On pourra ainsi utiliser dans les mêmes zones de mémoire des variables de type différents, **`int`**, **`double`**, **`struct`**,... un seul des membres pouvant être actif. La **taille** d'une union reste celle de son membre le **plus grand**.

Exemple 1 :

```
#include <stdio.h>

int main()
{
  union Exemple
  {
    int i;
    double d;
    char ch;
  } mu = {90}, alpha, *ptr= &mu;

  printf("\n taille des objets   mu: %d"
         "\n\t\t     mu.i: %d , mu.d: %d , mu.ch: %d\n",
         sizeof( mu), sizeof( mu.i),
         sizeof( mu.d), sizeof( mu.ch));

  printf("\n mu.i= %d\n mu.d= %.3lf\n mu.ch= %c\n",
         mu.i, mu.d, mu.ch);

  mu.d = 89.012;
  printf("\n mu.i= %d\n mu.d= %.3lf\n mu.ch= %c\n",
           mu.i, mu.d, mu.ch);

  mu.ch = 'A';
```

```
  printf("\n mu.i= %d\n mu.d= %.3lf\n mu.ch= %c\n",
         mu.i, mu.d, mu.ch);

  ptr->i = 88;
  printf("\n mu.i= %d\n mu.d= %.3lf\n mu.ch= %c\n",
         ptr->i, ptr->d, ptr->ch);

  alpha = mu;
  printf("\n alpha.i= %d\n alpha.d= %.3lf\n alpha.ch= %c\n",
         alpha.i, alpha.d, alpha.ch);
  return 0;
}

/*
 taille des objets   mu: 8
                     mu.i: 4 , mu.d: 8 , mu.ch: 1

 mu.i= 90
 mu.d= 0.000
 mu.ch= Z

 mu.i= -1683627180
 mu.d= 89.012
 mu.ch= T

 mu.i= -1683627199
 mu.d= 89.012
 mu.ch= A

 mu.i= 88
 mu.d= 89.012
 mu.ch= X

 alpha.i= 88
 alpha.d= 89.012
 alpha.ch= X
*/
```

Quelles remarques pouvons-nous faire ?

1) La première série d'affichages donne la taille des différents objets. Remplacez le mot **union** par le mot **struct** et vous aurez le même résultat mais la taille de l'objet mu passe de 8 (la taille du double) à 13 octets (la somme des tailles de chaque type).
2) L'initialisation n'est possible avec la déclaration que pour le premier membre de la liste de définition de l'union: union Exemple { int i; double d; } mu = {90}; donne mu.i= 90 Si on avait écrit 90.5 la valeur aurait été convertie en 90. Mais il vaut peut être mieux ne pas jouer avec ça! L'initialisation peut se faire par affectation, comme pour les structures: alpha= mu;
3) L'accès aux membres se fait aussi comme avec les structures: alpha.i ou ptr->i
4) Analysez les résultats affichés et vous constaterez que malgré certaines coïncidences, on ne peut se fier qu'à la valeur introduite en dernier lieu dans l'union, car il n'y a pas de manière simple permettant de tester le type de la variable vivante dans l'union. Dans un programme complexe, on peut rapidement ne plus trop savoir où on en est!

Quand il faut impérativement utiliser une union, on peut maintenir une certaine sécurité en lui associant une numération qui précise l'état où se trouve l'union:

```
union { int i; double d; char ch; } mu;
enum etat_union { Vide, car, entier, reel= 8 } e_mu= Vide;
```

Les nombres entiers associés à l'énumération `0`, `1`, `2`, `8` correspondent à la longueur des types des variables de l'union.

```
  mu.i = 90;
  e_mu = entier;
```

Il faut reconnaître que c'est assez lourd à gérer.

Exemple 2 :

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

typedef union { int i; double d; } Exemple;
typedef enum { vide, entier = 2, reel = 8 } Etat;
typedef enum { Echec = -1, Succes, Vrai = 0, Faux } Logique;

void lireInt(int *);
void lireDouble(double *);
Logique lireExemple(Exemple * const, const Etat);

int main()
{
  Exemple ex;
  Etat e_ex = vide;
  Logique test;

  e_ex = entier;
  test = lireExemple(&ex, e_ex);
  if (test == Succes)
    printf(" ex. i= %d ", ex.i);
  else
    printf("\n Echec de l'allocation de mémoire!");

  e_ex = reel;
  test = lireExemple(&ex, e_ex);

  if (test == Succes)
    printf(" ex. d= %.3lf ", ex.d);
  else
    printf("\n Echec de l'allocation de mémoire!");
  return 0;
}

Logique lireExemple(exemple *const e, const etat e_e)
{
  exemple *ptr = ( Exemple * ) malloc(sizeof(Exemple));
  if (ptr == NULL) return Echec;

  if (e_e == entier)
  {
    printf(" \n Entrez une valeur entière: ");
    lireInt(&ptr->i);
    e->i = ptr->i;
  }
  else
  {
    printf(" \n Entrez une valeur réelle: ");
    lireDouble(&ptr->d);
```

```
    e->d = ptr->d;
  }
  free(ptr); /* libération de la mémoire    */
  return Succes;
}

void lireInt(int *entier)
{
  double dble;
  do
  {
    while (scanf("%lf", &dble)!= 1)
      while (getchar() != '\n') ;
    while (getchar() != '\n') ;
  } while (dble < -32768.0 || dble > 32767.0);
  *entier = ( int )dble;
}

void lireDouble(double *reel)
{
  while (scanf("%lf", reel)!= 1)
    while (getchar() != '\n') ;
  while (getchar() != '\n') ;
}
```

L'exemple précédent rassemble un petit paquet de difficultés. L'objectif est de réaliser une opération simple dans une fonction qui accepte indifféremment des variables entières ou réelles. L'opération sera ici la saisie d'une valeur. Pour corser un peu les choses, nous créerons dynamiquement dans la fonction une variable qui pourra alternativement avoir le type **int** ou **double**. Nous utiliserons ensembles des énumérations, des unions, et la directive **typedef**.

1) Nous définissons une union et deux numérations que nous connaissons déjà en utilisant la directive **typedef**.
2) La fonction lireExemple() est une fonction d'initialisation du type Exemple. Elle reçoit un pointeur de type union Exemple et une variable entière de type enum Etat. Elle renvoie une variable entière de type enum Logique.
3) Dans le programme principal on déclare l'union ex de type Exemple et l'énumération Etat qui lui est liée qu'on initialise aussitôt avec le mot vide qui correspond à son état actuel. On déclare également l'énumération test de type Logique qui sera dans tous les cas initialisée par la valeur de type Logique renvoyée par la fonction. On l'initialiserait tout de même, pour des raisons de sécurité, dans un programme d'application.
4) On donne la valeur entier à l'énumération et on appelle la fonction qui renverra une valeur affectée à la variable test. Le retour de la fonction et la variable sont bien sûr du même type Logique. L'adresse de la variable ex de type Exemple est &ex comme avec une structure. On passe également à la fonction l'énumération e_ex de type Etat dont la valeur actuelle est entier ou 2.
5) Pour ne pas travailler directement avec la variable qui lui a été passée, la fonction commence par créer dynamiquement une variable intermédiaire. Cette variable de type Exemple contiendra les deux types **int** et **double**. Elle servira aux manipulations nécessaires dans la fonction et sera finalement affectée à e. Notez que le pointeur utilisé avec malloc() doit être du type de la variable manipule, qui est ici Exemple.
6) Une autre solution aurait consisté à créer dans la partie qui leur est propre une variable de type **int** et une de type **double**, la variable e_e ayant successivement les valeurs 2 et 8. int

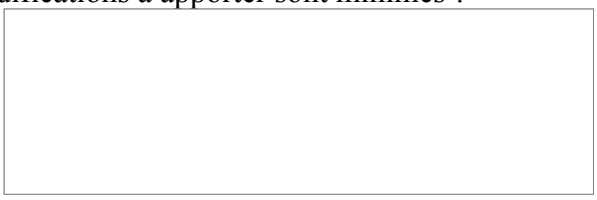
`*ptr = ( int * )malloc(e_e);` et plus loin `double *ptr = ( double * ) malloc(e_e);` Notez que les pointeurs sont bien du type de la variable manipulée. La fonction est virtuellement dédoublée!

7) En cas d'échec de l'allocation, `malloc()` renvoie un pointeur `NULL`, la fonction est fermée et renvoie un entier de type `Logique` avec la valeur `Echec`. A la fin de la fonction la zone de mémoire allouée est libérée.
8) La saisie se fait de la même manière que pour les structures avec l'adresse passée `&ptr->i` . Finalement on affecte la valeur saisie dans `ptr` à la variable `ex` par pointeurs et la fonction renvoie `Succes`.

11.5. Les structures chaînées

Le principal problème des données stockées sous forme de tableaux est que celles-ci doivent être ordonnées : le « suivant » doit toujours être stocké physiquement derrière.
Imaginons gérer une association. Un tableau correspond à une gestion dans un cahier : un adhérent par page. Supposons désirer stocker les adhérents par ordre alphabétique. Si un nouvel adhérent se présente, il va falloir trouver où l'insérer, gommer toutes les pages suivantes pour les re-écrire une page plus loin, puis insérer le nouvel adhérent. Une solution un peu plus simple serait de numéroter les pages, entrer les adhérents dans n'importe quel ordre et disposer d'un index : un feuille où sont indiqués les noms, dans l'ordre, associés à leur « adresse » : le numéro de page. Toute insertion ne nécessitera de décalages que dans l'index. Cette méthode permet l'utilisation de plusieurs index (par exemple un second par date de naissance). La troisième solution est la liste chaînée : les pages sont numérotées, sur chaque page est indiquée la page de l'adhérent suivant, sur le revers de couverture on indique l'adresse du premier. L'utilisation d'une telle liste nécessite un véritable « jeu de piste », mais l'insertion d'un nouvel adhérent se fera avec le minimum d'opérations.
Appliquons cela, de manière informatique, à une liste d'entiers, avec pour chaque valeur l'adresse (numéro de mémoire) du suivant :

Si l'on veut insérer une valeur dans la liste, les modifications à apporter sont minimes :



En C on définira un type structure regroupant une valeur entière et un pointeur :

```
struct Page {
  int val;
  struct page *suivant;
};
```

Un pointeur (souvent global) nous indiquera toujours le début de la liste:

```
struct Page *premier;
```

Au fur et à mesure des besoins on se crée une nouvelle page :

```
nouveau = ( struct Page * )malloc(sizeof(struct Page));
```

En n'oubliant pas de préciser le lien avec le précédent :

```
precedent->suivant = nouveau;
```

le dernier élément ne doit pas pointer sur n'importe quoi, on choisit généralement soit le pointeur NULL, soit le premier (la liste est dite bouclée).

Exemple :

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
struct Page {int val; struct Page *suivant; };
struct Page *premier;

int encore(void) /* demande si on en veut encore*/
{
  printf("encore (O/N) ? ");
  return (toupper(getchar())=='O');
}

void lire(void)
{
  struct Page *precedent,*nouveau;
  premier = ( struct Page * )malloc(sizeof(struct Page));
  puts("entrez votre premier entier");
  scanf("%d",&premier->val);
  precedent = premier;
  while (1)
  {
    nouveau = ( struct Page * )malloc(sizeof(struct Page));
    precedent->suivant = nouveau;
    precedent = nouveau;
    puts("\nentrez votre entier");
    scanf("%d",&nouveau->val);
  }
  precedent->suivant = NULL;
}

void afficher(struct Page *debut)
{
  printf("\nliste : ");
  while(debut != NULL)
  {
    printf("%d ",debut->val);
    debut = debut->suivant;
  }
  printf("\n");
}

int main(void)
{
  lire();
  afficher(premier);
}
```

12. LES CHAMPS BINAIRES

Le C permet de travailler directement avec les éléments des mots binaires qui sont actuellement employés par les machines. C'est particulièrement utile pour écrire, dans des cas particuliers, des procédures de dialogue avec des périphériques ou des processeurs ou tout simplement pour rendre plus rapides certaines séquences névralgiques à l'intérieur de nos programmes. Ces choses sont familières à nos amis qui programment en assembleur. On appelle parfois ces opérations de la « programmation de bas niveau », terme assez peu heureux en la circonstance.
Nous donnerons un aperçu des moyens de transformation d'un mot binaire par une opération de masquage avant de parler des champs. Pour vous mettre en forme, revoyez le chapitre sur les opérateurs binaires.

12.1. Le masquage

C'est une opération qui permet de modifier un mot binaire. L'application qui permet la modification s'appelle le masque:

```
x  ==> masque ==> y
y = x (masque)   on applique à x un masque et le
                 résultat est affecté à y.
```

Voici quelques exemples de masques. Nous avons ajouté deux exemples de décalage pour préciser la nature des caractères de remplacement.
Par commodité nous utilisons dans ces exemples des entiers cours signés de 16 bits (**short**).

12.1.1. Exemples de masquages

```
#include <stdio.h>

int main()
{
  int x = 0x5555, y;
  /* exemple 1    */
  y = x & 0xff ;
  printf("\n y= %x , %d", y, y);

  /* exemple 2    */
  y = x & 0xff00;
  printf("\n\n y= %x , %d", y, y);

  y = x & ~0xff;  /*  0xff = ~0xff00 */
  printf("\n y= %x , %d", y, y);

  /* exemple 3    */
  y = x | 0xff ;
  printf("\n\n y= %x , %d", y, y);

  /* exemple 4    */
  y = x | 0xff00;
```

```
  printf("\n\n y= %x , %d", y, y);

  y = x | ~0xff;
  printf("\n y= %x , %d", y, y);

  /* exemple 5      */
  y = x ^ 0xff;
  printf("\n\n y= %x , %d", y, y);

  /* exemple 6      */
  y = x ^ 0xff00;
  printf("\n\n y= %x , %d", y, y);

  y = x ^ ~0xff;
  printf("\n y= %x , %d", y, y);
  return 0;
}
/*

 y= 55 , 85

 y= 5500 , 21760
 y= 5500 , 21760

 y= 55ff , 22015

 y= ff55 , -171
 y= ff55 , -171

 y= 55aa , 21930

 y= aa55 , -21931
 y= aa55 , -21931
*/
```

Exemple 1: `y = x  & 0xFF;`

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
(masque)	& 0000 0000 1111 1111	255	FF
= y	0000 0000 0101 0101	85	55

Le masque est dans ce cas l'opérateur logique binaire `&` et la valeur 55 en hexadécimal. Son effet est de masquer les 8 premiers bits de la variable x, ou si vous préférez d'extraire les 8 bits de droite pour les affecter à la variable y, en annulant les 8 bits de gauche.
Si le mot comportait 32 bits au lieu de 16 nous aurions le même résultat car les bits ajoutés à gauche seraient des zéros. Ce masque n'est donc pas lié à la taille des mots.

Exemple 2: `y = x & 0xFF00;`

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
(masque)	& 1111 1111 0000 0000	-256	FF00
= y	0101 0101 0000 0000	21760	5500

Le masque est dans ce cas l'opérateur logique binaire `&` et la valeur `FF00`. Son effet est de masquer les 8 derniers bits de la variable `x`.
Si on utilise des mots de plus de 16 bits, les bits de remplissage à gauche seraient des 1 puisque le bit de poids fort est un 1. Le masque est donc lié à la longueur des mots utilisés. Pour pallier cet inconvénient, on utilise plutôt le complément à 1 du nombre `FF00` qui est `~FF00`:

```
 FF00       1111 1111 0000 0000       -256         FF00
~FF00       0000 0000 1111 1111        255           FF
```

d'où: `~FF00 = FF <=> ~FF = FF00` et finalement:

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
& ~00FF	& 1111 1111 0000 0000	-256	FF00
= y	0101 0101 0000 0000	21760	5500

Soit: `y= x & ~0xFF;`
Ce qui donne le même résultat mais le masque reste utilisable avec des mots plus longs. Ceci est possible puisque la priorité de l'opérateur d'inversion de bit `~` est plus forte que celle de l'opérateur `&`.

Exemple 3: `y = x | 0xFF;`

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
(masque)	| 0000 0000 1111 1111	255	FF
= y	0101 0101 1111 1111	22015	55FF

Le masque est dans ce cas l'opérateur logique binaire `|` et la valeur `FF`. Son effet est de mettre les 8 derniers bits de la variable `x` à 1.

Exemple 4: `y = x | 0xFF00;`

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
(masque)	| 1111 1111 0000 0000	-256	FF00
= y	1111 1111 0101 0101	-171	FF55

Le masque est dans ce cas l'opérateur logique binaire `|` et la valeur `FF00`. Son effet est de mettre les 8 premiers bits de la variable binaire `x` à 1. Là aussi on utilisera plutôt le complément à 1 de `FF00` qui est `~FF00 = FF` pour ne pas être limités par la taille des mots.

```
y = x | ~0xFF;
```

Exemple 5: `y = x ^ 0xFF;`

Opération	***binaire***	***décimal***	***hexadécimal***
x	0101 0101 0101 0101	21845	5555
(masque)	^ 0000 0000 1111 1111	255	FF
= y	0101 0101 1010 1010	21930	55AA

Le masque est dans ce cas l'opérateur logique binaire `|` et la valeur `FF`. Son effet est d'inverser les 8 derniers bits de la variable `x`.

Exemple 6: `y = x ^ FF00;` ou plutôt: `y = x ^ ~0xFF;`

Opération	*binaire*	*décimal*	*hexadécimal*
x	0101 0101 0101 0101	21845	5555
(masque)	^ 1111 1111 0000 0000	-256	FF00
= y	1010 1010 0101 0101	43605	AA55

Le masque est dans ce cas l'opérateur logique binaire `^` et la valeur `FF00`. Son effet est d'inverser les 8 premiers bits de la variable `x`.

12.1.2. Exemples de décalages

```
#include <stdio.h>

int main()
{
  short x0 = 0x5555, x1 = 0xaaaa, y;

  /* exemple 1  */
  y = x0 << 4 ;
  printf(" y= %x , %d", y, y);

  y = x0 << 5 ;
  printf("\n y= %x , %d", y, y);

  /* exemple 2  */
  y = x0 >> 4 ;
  printf("\n\n y= %x , %d", y, y);

  y = x1 >> 4 ;
  printf("\n y= %x , %d", y, y);
  return 0;
}

/*

y= 5550 , 21840
y= aaa0 , -21856

y= 555 , 1365
y= faaa , -1366
*/
```

Exemple de décalages à gauche: `y = x0 << 4;` et `y = x0 << 5;`

Opération	*binaire*	*décimal*	*hexadécimal*
x0	0101 0101 0101 0101	21845	5555
<< 4			
= y	0101 0101 0101 0000	21840	5550

Les bits de gauche sont perdus et les positions libres à droite sont remplies par des zéros. Si on décale de 5 le résultat décimal est négatif.

Opération	*binaire*	*décimal*	*hexadécimal*
x0	0101 0101 0101 0101	21845	5555
<< 5			
= y	1010 1010 1010 0000	-21856	AAA0

Exemple de décalages à droite: `y = x0 >> 4;` et `y = x1 >> 4;`

Opération	*binaire*	*décimal*	*hexadécimal*
x0	0101 0101 0101 0101	21845	5555
>> 4			
= y	0000 0101 0101 0101	1365	555

Les bits de droite sont perdus et les positions libres à gauche sont remplies par des zéros, comme le bit de poids fort.

Opération	*binaire*	*décimal*	*hexadécimal*
x1	1010 1010 1010 1010	-21846	AAAA
>> 4			
= y	1111 1010 1010 1010	-1366	FAAA

Les bits de droite sont perdus et les positions libres à gauche sont remplies par des 1, comme le bit de poids fort.

12.2. Les champs binaires

Lorsqu'il faut travailler avec des données qui ne comprennent que quelques bits, le C permet de rassembler ces données dans un ou plusieurs mots mémoire. On divise ces mots en plusieurs champs binaires qui sont les membres d'une structure, d'une union ou en C++ d'une classe.

```
une structure ou une union, ou une classe.
{
  1er  mot; chaque mot se partage en champs binaires,
  2me mot;  chaque champ est membre de la structure.
  .........
};
```

Bien évidemment, il ne sera pas possible d'utiliser des champs construits pour des mots de 16 bits avec des programmes qui emploient des mots de 32 bits.

12.2.1. Définition

On peut placer dans les mots des champs binaires dont on définit la taille. Le type des variables est celui des entiers signés ou non, court ou long. En C++ on utilise aussi les variables de type **`signed`** ou **`unsigned char`**.

```
type      :    taille en bits point-virgule ou virgule
```

unsigned : 5 ;

```
struct champ_binaire
{
  short          i      :  9,
                        :  2; /* 1er mot de 16 bits */
  unsigned short u      :  5;
  int e0                : 12; /* second mot de 16 bits
                                 nous avons omis la fin. */
};
```

- La largeur d'un champ doit donc toujours être inférieure à la taille du mot mémoire. Elle peut être omise quand il s'agit de la fin de la déclaration mais pas à l'intérieur d'un mot sinon les champs suivants seront décalés.
- Par contre on peut comme dans notre exemple omettre le nom d'un champ. La zone correspondante restera inaccessible.
- Il n'est pas possible de placer un champ unique à cheval sur 2 mots.
- Si le dernier champ déborde des limites du mot, il est automatiquement reporté en entier au mot suivant.
- Les champs sont alloués en partant des bits de poids faible vers les bits de poids fort. Attention, certains compilateurs font le contraire, FAITES DES ESSAIS!

Une structure construite avec le premier mot de notre exemple serait disposée de la manière suivante:

```
  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
<         u         >  vide <               i                   >
```

12.2.2. Déclaration et initialisation

```
struct cb { /* définition. */
  short i               : 2,
                        : 4;
  unsigned short u      : 6;
};

/* déclaration d'un tableau tab_cb de
   2 structures de type cb et initialisation.
 */
struct cb tab_cb[2]= {{1, 32u}, {3}};
```

L'initialisation se fait comme avec les structures. Selon les compilateurs il faut parfois initialiser les membres dont on a omis le nom, ce qui est le cas du second **int**. Là encore, FAITES DES ESSAIS! Ce sont les bits de droite, de poids faibles, qui sont affectés au membre. Les bits en trop sont masqués. Les valeurs affectées aux membres doivent être entières.

Exemple : **Initialisation de champs binaires**

```
#include <stdio.h>

#define DIM 3
```

```
int main()
{
  int i;
  struct champ_bin {
    short i             :  1,
          j             :  2,
                        :  2;
    unsigned short u    :  4,
                  v     :  6,
                  w     :  1;
  };

  struct champ_bin cb[DIM]= {{ 0, 1, 14u, 30u, 0},
                             { 1, 2, 30u, 76u, 1u}};

  printf(" Etat des membres des structures  cb[]:\n\n");

  for (i = 0; i < DIM; i++)
  {
    printf("\n cb[%d].i= %2d , cb[%d].j= %2d ,"
     " cb[%d].u= %2u , cb[%d].v= %2u , cb[%d].w= %2u",
     i, cb[i].i, i, cb[i].j, i, cb[i].u, i, cb[i].v, i,
     cb[i].w);
  }

  cb[2]   = cb[0];
  cb[1].i = 0;
  cb[1].j = cb[0].j;
  cb[1].u = cb[2].u;
  cb[1].v = 30u;
  cb[1].w = 0;

  printf("\n\n");
  for (i = 0; i < DIM; i++)
  {
    printf("\n cb[%d].i= %2d , cb[%d].j= %2d ,"
      " cb[%d].u= %2u , cb[%d].v= %2u , cb[%d].w= %2u",
      i, cb[i].i, i, cb[i].j, i, cb[i].u, i, cb[i].v, i,
      cb[i].w);
  }
  return 0;
}

/*
 Etat des membres des structures  cb[]:

 cb[0].i=  0 , cb[0].j=  1 , cb[0].u= 14 , cb[0].v= 30 ,
cb[0].w=  0
 cb[1].i= -1 , cb[1].j= -2 , cb[1].u= 14 , cb[1].v= 12 ,
cb[1].w=  1
 cb[2].i=  0 , cb[2].j=  0 , cb[2].u=  0 , cb[2].v=  0 ,
cb[2].w=  0


 cb[0].i=  0 , cb[0].j=  1 , cb[0].u= 14 , cb[0].v= 30 ,
cb[0].w=  0
 cb[1].i=  0 , cb[1].j=  1 , cb[1].u= 14 , cb[1].v= 30 ,
cb[1].w=  0
 cb[2].i=  0 , cb[2].j=  1 , cb[2].u= 14 , cb[2].v= 30 ,
```

```
cb[2].w=  0
*/
```

1) Nous définissons la structure champ_bin qui contient un mot de 16 bits, partagé en 6 champs, dont un n'est pas nommé.

```
 15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
* w *           v           *       u       *-------*   j   * i *
```

2) Nous déclarons le tableau cb de 3 structures du type champ_bin et nous les initialisons aussitôt à la manière des tableaux de structures. Nous supposerons que le compilateur n'exige pas l'initialisation du champ qui n'est pas nommé. L'affichage se fait en accédant aux membres comme avec des structures.

3) La 1ère et la 3ème structure s'affichent sans surprises. Par contre la seconde mérite d'être analysée de près.

a) Nous avions donné à la variable signée i la valeur : 1. Cette valeur est traduite par la valeur binaire: 000 0000 0000 0001. Comme nous ne disposons en i que d'un bit, la partie de gauche est tronquée et il ne reste que: 1, qui se traduit en décimal, pour un entier court signé par: -1. Le même espace de 1 bit occupé plus loin par la variable w pour un entier non signé, traduit la même valeur binaire 1 par le 1 décimal.

b) Nous avions donné à la variable signée j la valeur : 2. Cette valeur est traduite par la valeur binaire: 0000 0000 0000 0010. Comme nous ne disposons en i que de deux bits, la partie de gauche est tronquée et il ne reste que : 10. Pour l'évaluer en décimal, la machine le remet à 16 bits en respectant la règle du bit de poids fort pour un entier signé, donc ici 1. Et elle nous renvoie: 1111 1111 1111 1110, qui se traduit en décimal, pour un entier signé par: -2.

c) Nous avions donné à la variable non signée u la valeur : 30. Cette valeur est traduite par la valeur binaire: 0000 0000 0001 1110. Comme nous ne disposons en i que de 4 bits, la partie de gauche est tronquée et il ne reste que : 1110, complété pour un entier non signé en : 0000 0000 0000 1110, qui se traduit en décimal, pour un entier non signé par: 14.

d) Nous avions donné à la variable non signée v la valeur : 76. Cette valeur est traduite par la valeur binaire: 0000 0000 0100 1100. Comme nous ne disposons en i que de 6 bits, la partie de gauche est masquée et il ne reste que: 00 1100, complété en : 0000 0000 0000 1100, qui se traduit en décimal, pour un entier non signé par: 12.

Pour des entiers non signés:
La conclusion est qu'il n'est possible d'employer normalement dans les champs binaires que des valeurs inférieures ou égales à un maximum qui dépend du nombre n de bits qu'il contient:

MAX(n)= $2^{n-1}+ 2^{n-2}+...+ 2^1+ 2^0$

Par exemple, si n = 4 alors MAX(4)= $2^3+ 2^2+ 2^1+ 2^0 = 8+ 4+ 2+ 1 = 15$

Pour des entiers signés, c'est plus délicat:

- Un champ de 1 bit ne peut contenir que 0 ou -1, puisque la valeur binaire 1 est interprétée comme -1.
- Un champ de 2 bits peut contenir les valeurs binaires: 00, 01, 10, 11, qui seront interprétées comme: 0 1 -2 -1.
- Un champ de 3 bits peut contenir les valeurs binaires: 000, 001, 010, 011, 100, 101, 110, 111, qui seront interprétées comme: 0 1 2 3 -4 -3 -2 -1

- et ainsi de suite...0, 1, 2,..., n, -(n+ 1), -n, -(n- 1),..., -2, -1.

Cela explique sans doute que la plupart des applications se font avec des `unsigned int`.

4) Comme pour les structures nous pouvons affecter `cb[0]` à `cb[2]`.

5) Nous pouvons aussi modifier directement un membre: `cb[1].i = 0;` ou lui affecter une variable du même type: `cb[1].j = cb[0].j;`

12.2.3. Utilisation

- On ne peut pas travailler avec l'adresse d'un membre d'un champ binaire car les adresses ne peuvent être celles de fractions d'octet. On ne peut donc pas manipuler des membres avec des pointeurs.
- On peut affecter une valeur une variable d'un champ binaire ou lui affecter une variable du même type.
- On ne peut pas forcer le type d'une variable.
- On peut sur chaque membre faire toutes opérations sur des entiers qui peuvent être représentés avec le nombre de bits du membre.
- Des champs binaires peuvent être associés comme membres à des structures ou des unions ordinaires. Exemple :

```
struct melange
{
  short entier;          /* un membre entier signé.    */
  unsigned short i: 4,   /* un membre qui est un champ binaire.*/
                 j: 8,
                 k: 3;
  double dble;           /* un membre de type double.  */
  struct date d0;        /* une structure de type différent.   */
};
```

Exemple : **Opérations de masquage dans des champs binaires avec des entiers non signés**

```
#include <stdio.h>

typedef struct {
  unsigned short a: 8,
                 b: 4,
                 c: 3;
} champ_bin;

typedef union {
  champ_bin cb;
  double dble;
} champ_U;

int main()
{
  champ_bin cb0 = {85, 4};
  champ_U ucb;
  ucb.cb = cb0;

  /* 1. masquage d'un champ. */
  printf(" cb0.a= %d", cb0.a);
  cb0.a &= 0xf;   /* ou cb0.a= cb0.a & 0xf; */
```

```
  printf("\n cb0.a= %d", cb0.a);

  /* 2. masquage sur un champ binaire membre d'une union. */
  printf("\n\n ucb.cb.a= %d , ucb.cb.b= %d , ucb.cb.c= %d",
          ucb.cb.a, ucb.cb.b, ucb.cb.c);

  ucb.cb.a &= 0xf;
  printf("\n ucb.cb.a= %d", ucb.cb.a);
  return 0;
}

/*
 cb0.a= 85
 cb0.a= 5

 ucb.cb.a= 85 , ucb.cb.b= 4 , ucb.cb.c= 0
 ucb.cb.a= 5
*/
```

Les opérations de masquage s'appliquent directement aux variables des champs binaires. Dans cet exercice nous allons extraire les 4 bits de poids faible à droite d'un champ et les lui affecter ou si vous préférez, masquer les bits de gauche au delà de 4.
Dans un second exemple, on place donc le champ binaire dans une union, associé à un membre de type quelconque et on réalise la même opération.

1) Nous définissons une structure champ_bin et une union ou un champ binaire de ce type est associé à une variable réelle. On déclare et initialise simultanément la structure de champs binaires cb0 puis on déclare une union qu'on initialise en lui affectant la structure que l'on vient de créer.
2) Nous faisons l'opération de masquage sur le champ a de cb0 après avoir contrôlé le contenu de cb0.a.

Opération	***binaire***	***décimal***	***hexadécimal***
cn0.a	0101 0101	85	AAAA
(masque)	& 0000 0000 0000 1111	15	0xF
= cb0.a	0000 0000 0000 0101	5	0x5

3) Nous faisons la même opération sur le champ correspondant de l'union, après en avoir vérifié le contenu, et nous obtenons le même résultat.

13. LES FONCTIONS DE MANIPULATION DE FICHIERS

Notions : `FILE`, `fopen()`, `fread()`, `fwrite()`, `fclose()`, `open()`, `read()`, `write()`, `close()`.

Les données stockées en mémoire sont perdues dès la sortie du programme. Les fichiers sur support magnétique (bande, disquette, disque) ou électronique (clé USB) peuvent par contre être conservés, mais au prix d'un temps d'accès aux données très supérieur. On peut distinguer :

- les fichiers séquentiels pour lesquels on accède au contenu dans l'ordre du stockage.
- les fichiers à accès indexé encore appelé accès aléatoire, pour lesquels on peut directement accéder à n'importe quel endroit du fichier.

Le C distingue deux catégories de fichiers :

- Les fichiers « bruts ». Ces fichiers sont des fichiers dits « binaire » dans lesquels
- Les fichiers « bufferisés » utilisant les flux de données standards. Ceux-ci peuvent peuvent être « binaires » ou « texte »,

Les fichiers étant dépendants du matériel, ils ne sont pas prévus dans la syntaxe du C mais par l'intermédiaire de fonctions spécifiques qui diffèrent suivant les systèmes d'exploitation et les compilateurs.

13.1. Fichiers bufferisés

Les opérations d'entrée/sortie sur ces fichiers se font par l'intermédiaire d'un « tampon mémoire » géré automatiquement. Cela signifie qu'une instruction d'écriture n'impliquera pas une écriture physique sur le disque mais dans le tampon, avec une écriture sur le disque uniquement lorsque celui-c sera plein. Les fichiers bufferisés peuvent être utilisés :

- Soit en accès direct, en lecture et écriture, avec tous les éléments de même type et même taille (souvent une structure en format binaire), ceci permettant d'accéder directement à un élément donné (le 10ème, le précédent, l'avant-dernier, etc.).
- Soit en accès séquentiel, avec des éléments de types différents, tous formatés sous forme « texte » (c'est-à-dire qu'un `float` binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre), en lecture seule ou écriture seule, ces fichiers seront compréhensibles par n'importe quel autre programme (éditeur de textes, imprimante, etc.). Un tel fichier s'utilise comme l'écran et le clavier, par des fonctions similaires.

Les fichiers sont identifiés par un pointeur sur une structure `FILE`.

13.1.1. Ouverture et fermeture d'un fichier

Avant de pouvoir lire ou écrire dans un fichier, il faut que celui-ci soit ouvert. C'est ce que réalise la fonction `open()` qui obéit à la syntaxe suivante :

```
FILE * fopen(char * chemin, char * mode);
```

Cette fonction ouvre le fichier de nom `chemin` suivant le mode :

- r : lecture seule)
- w : écriture, si le fichier existe il est d'abord vidé
- a : (*append*) écriture à la suite du contenu actuel, création si inexistant
- r+ : lecture et écriture, le fichier doit exister
- w+ : lecture et écriture mais effacement au départ du fichier si existant
- a+ : lecture et écriture, positionnement en fin de fichier si existant, création sinon

On peut rajouter t ou b au mode pour des fichiers texte (gestion des CR/LF, option par défaut) ou binaires, ou encore le définir par défaut en donnant à la variable _fmode la valeur O_TEXT ou O_BINARY.
fopen() retourne un pointeur qui nous servira pour accéder au fichier. En cas d'erreur, le pointeur NULL est retourné, le type d'erreur est donné dans une variable errno, détaillée dans errno.h.
La fonction void perror(char *s_mess) permet d'afficher un message correspondant à l'erreur. En général on lui passe le nom du fichier.

On referme un fichier avec la fonction fclose() :

```
int close(FILE * fd);
```

Cette fonction referme le fichier fd, en y recopiant le reste du tampon si nécessaire. Cette fonction est obligatoire pour être sur d'avoir l'intégralité des données effectivement transférées sur le disque. En cas de réussite, elle retourne 0.

13.1.2. Lecture d'un fichier

La lecture dans un fichier se fait par la fonction fread qui obéit à la syntaxe suivante :

```
int fread(void *tampon, int taille, int n, FILE *fd);
```

Cette fonction lit n éléments dont la taille unitaire est taille dans le fichier dont le descripteur est fd et les place dans tampon. En cas de réussite, elle renvoie le nombre d'éléments lus (< n si fin de fichier), sinon elle retourne 0.

13.1.3. Écriture d'un fichier

L'écriture dans un fichier se fait par la fonction fwrite :

```
int fwrite(void *tampon, int taille, int n, File * fd);
```

Cette fonction écrit n élément dont la taille unitaire est taille et contenus dans tampon dans le fichier dont le descripteur est fd. Cette fonction retourne le nombre d'octets écrits ou un nombre différent en cas d'erreur.

13.1.4. Autres fonctions

```
int fflush(FILE * fd)
```

Transfère effectivement le reste du tampon sur disque, sans fermer le fichier (à appeler par exemple avant une instruction qui risque de créer un « plantage »).

```
int fseek(FILE * fd, long combien, int mode)
```

Déplace le pointeur de fichier de combien octets, à partir de : début du fichier (mode=0), position actuelle (mode=1) ou fin du fichier (mode=2). Retourne 0 si tout c'est bien passé. Cette fonction n'est utilisable que si l'on connaît la taille des données dans le fichier (impossible d'aller directement à une ligne donnée d'un texte si on ne connaît pas la longueur de chaque ligne).

```
int feof(FILE * fd)
```

Indique si la fin de fichier est atteinte ou non (0).

- Les fichiers bufferisés permettent aussi des sorties formatées au niveau caractère :

```
char fgetc(FILE * fd)
char fputc(char c, FILE * fd)
```

et même :

```
char ungetc(char c, FILE * fd)
```

qui permet de reculer d'un caractère. Cette fonction correspond donc à {fseek(fd,-1,1);c=fgetc(fd)}.

- Mais aussi au niveau chaîne de caractères :

```
char * fgets(char * s, int max, FILE * fd)
```

Lit une chaîne en s'arrêtant au \n ou à max-1 caractères lus, résultat dans la zone pointée par s, et retour du pointeur s ou NULL en cas d'erreur.

```
char fputs(char * s, FILE * fd)
```

Écrit la chaîne dans le fichier sans ajouter de \n, rend le dernier caractère écrit ou EOF si erreur.

```
int fprintf(FILE * fd, char *format, listearguments)
```

Retourne le nombre d'octets écrits, ou EOF en cas d'erreur. Les \n sont transformés en CR/LF si fichier en mode texte (spécifique PC).

```
int fscanf(FILE * fd, char * format, listeadresses)
```

Retourne le nombre de variables lues et stockées, 0 en cas d'erreur.

Exemple :

```
#include <stdio.h>

int main()
{
  FILE * pfichier;
  char nom_fichier[30], nom_personne[30];
  int  compteur, nb_enregistrements;

  /* Première partie :
     Créer et remplir le fichier */
  printf("Entrez le nom du fichier à créer : ");
  scanf("%s", nom_fichier);
  pfichier = fopen(nom_fichier, "w");
  printf("Nombre d'enregistrements à créer : ");
  scanf("%d", &nb_enregistrements);
  compteur = 0;
  while (compteur < nb_enregistrements)
```

```
  {
    printf("Entrez le nom de la personne : ");
    scanf("%s", nom_personne);
    fprintf(pfichier, "%s\n", nom_personne);
    compteur++;
  }
  fclose(pfichier);

  /* Deuxième partie :
     Lire et afficher le contenu du fichier */
  pfichier = fopen(nom_fichier, "r");
  compteur = 0;
  while (!feof(pfichier))
  {
    fscanf(pfichier, "%s\n", nom_personne);
    printf("NOM : %s\n", nom_personne);
    compteur++;
  }
  fclose(pfichier);
  return 0;
}
```

13.2. Fichiers « bruts »

C'est la méthode la plus efficace et rapide pour stocker et récupérer des données sur fichier, mais aussi la moins pratique. On accède au fichier par lecture ou écriture de blocs d'octets dont la taille est définie par le programmeur. C'est ce dernier qui est chargé de préparer et gérer ses blocs. On choisira en général une taille de bloc constante pour tout le fichier, et correspondant à la taille d'un enregistrement physique (secteur, *cluster*, etc.). On traite les fichiers par l'intermédiaire de fonctions, prototypées dans `stdio.h` (ouverture et fermeture) et dans `unistd.h` (les autres), disponibles sur la plupart des compilateurs.

13.2.1. Ouverture et fermeture d'un fichier

Avant de pouvoir lire ou écrire dans un fichier, il faut que celui-ci soit ouvert. C'est ce que réalise la fonction `open` qui obéit à la syntaxe suivante :

```
int open(const char *chemin, int oflag, mode_t mode);
```

Cette fonction ouvre le fichier de nom `chemin` et retourne un descripteur de fichier (*handle* en anglais) qui permettra de l'identifier dans toutes les autres fonctions (lecture, écriture, déplacement, etc.). En cas d'échec, elle retourne `-1`.
Le paramètre `oflag` précise le mode d'ouverture du fichier :

- `O_RDONLY` : ouverture en lecture seule
- `O_WRONLY` : ouverture en écriture seule
- `O_RDWR` : ouverture en lecture et écriture
- `O_NDELAY` : ouverture non bloquante
- `O_APPEND` : positionnement en fin de fichier avant chaque écriture
- `O_CREAT` : création du fichier si il n'existe pas
- `O_TRUNC` : ouverture avec troncature si le fichier existe
- `O_EXCL` : ouverture exclusive (retourne un code d'erreur si le fichier existe déjà lors d'une création)

Ces constantes peuvent être combinées à l'aide de l'opération OU binaire.
Le paramètre mode définit les droits d'accès au fichier en cas de création (dans les autres cas, il n'est pas nécessaire de positionner ce paramètre). Il peut prendre les valeurs suivantes

- S_ISUID, S_IGID, S_ISVTX : activation respective des bits setuid, setgid et sticky ;
- S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU : activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le propriétaire du fichier ;
- S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXG : activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le groupe du propriétaire du fichier ;
- S_IROTH, S_IWOTH, S_IXOTH, S_IRWXO : activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour les autres.

Là aussi, ces constantes peuvent être combinées à l'aide de l'opération OU binaire.

On referme un fichier avec la fonction close() :

```
int close(int fd);
```

Cette fonction referme le fichier dont le descripteur est fd. En cas de réussite, elle retourne 0, sinon elle retourne -1.

13.2.2. Lecture d'un fichier

La lecture dans un fichier se fait par la fonction read() qui obéit à la syntaxe suivante :

```
ssize_t read(int fd, void *tampon, size_t n);
```

Cette fonction lit n octets dans le fichier dont le descripteur est fd et les place dans un tampon. En cas de réussite, elle renvoie le nombre d'octets transférés, sinon elle retourne -1.
Exemple :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
   char c;
   int fd;
   fd =  open("/etc/passwd", O_RDONLY);
   if (fd == -1){
     perror("impossible d'ouvrir le fichier\n");
     exit(1);
   }
   while ( read(fd, &c, 1) > 0 )
       putchar(c);
   close(fd);
   return 0;
}
```

13.2.3. Écriture d'un fichier

L'écriture dans un fichier se fait par la fonction write() :

```
ssize_t write(int fd, const void *tampon, size_t n);
```

Cette fonction écrit n octets dans le fichier dont le descripteur est fd à partir d'un tampon. Cette

fonction retourne le nombre d'octets écrits ou `-1` en cas d'erreur.
Exemple :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
   char tampon[]="Bonjour à tous !";
   int fd;
   fd = open("salut.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
   write(fd, tampon, sizeof(tampon));
   close(fd);
   return 0;
}
```

13.2.4. Autres fonctions

```
int eof(int fd)
```

Indique si on se trouve (`1`) ou non (`0`) en fin du fichier.

Le fichier peut aussi être utilisé séquentiellement, c'est-à-dire que le « pointeur de fichier » est toujours placé derrière le bloc que l'on vient de traiter, de manière à pouvoir traiter le suivant. Pour déplacer le pointeur de fichier en n'importe que autre endroit, on appelle la fonction :

```
long lseek(int fd, long combien, int code)
```

Elle déplace le pointeur de fichier de `combien` octets, à partir de : début du fichier si `code`=0, position actuelle si `1`, fin du fichier si `2`. La fonction retourne la position atteinte (en nombre d'octets), `-1L` si erreur.

```
long filelength(int fd)
```

Retourne la taille d'un fichier sans déplacer le pointeur de fichier.

Exemple : **Émulation simplifiée de la commande** `cp`

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#define TAILLE_BLOC 1024

int main(int argc, char * argv[])
{
  int source, destination;
  char buffer[TAILLE_BLOC];
  int nb_lus, nb_ecrits;
  if (argc != 3)
  {
    puts("erreur arguments");
    return 1;
  }
  if ((source=open(argv[1],O_RDONLY))<0)
```

```
  {
    puts("erreur ouverture");
    return 2;
  }
  if ((destination=open(argv[2],O_WRONLY|O_CREAT|
O_TRUNC,S_IREAD|S_IWRITE|S_IEXEC))<0)
  {
    puts("erreur ouverture");
    return 2;
  }
  do
  {
    nb_lus = read(source,( char * )buffer, TAILLE_BLOC);
    if (nb_lus > 0)
      nb_ecrits=write(destination,( char * )buffer, nb_lus);
  }
  while ((nb_lus==TAILLE_BLOC)&&(nb_ecrits>0)) ;
  close(source);
  close(destination);
  return 0;
}
```

13.3. Fonctions propres au VFS (*Virtual File System*) de Linux

Ces fonctions, que l'on appellent « primitives », permettent d'interagir avec le système d'exploitation. Leurs prototypes se trouvent dans les fichiers <unistd.h>.

13.3.1. Fonctions de gestion des fichiers

13.3.1.1. Modification du nom du fichier

```
int rename(const char *ancien_nom, const char *nouv_nom)
```

13.3.1.2. Modification des droits d'accès à un fichier

```
int chmod(const char *nom_fichier, mode_t mode)
```

mode prend les mêmes valeurs que pour la fonction open().

13.3.1.3. Modification du propriétaire d'un fichier

```
int chown(const char *nom_fichier, uid_t id_utilisateur,
          gid_t id_groupe)
```

13.3.2. Fonctions de gestion des répertoires

13.3.2.1. Création d'un répertoire

Un répertoire nom_rep est créé par un appel de la fonction :

```
int mkdir(const char *nom_rep, mode_t mode)
```

mode prend les mêmes valeurs que pour la fonction open().
En cas de succès, la fonction retourne le descripteur associé au répertoire. En cas d'échec, elle retourne la valeur -1 et la variable errno prend l'une des valeurs suivantes :

- EACESS : l'accès n'est pas possible ;
- EEXIST : un fichier de même nom existe déjà ;
- EFAULT : le chemin spécifié n'est pas valide ;
- ENAMETOOLONG : le chemin spécifié comprend un nom trop long ;
- ENOTDIR : l'un des composants du chemin d'accès n'est pas un répertoire ;
- ELOOP : un cycle de liens symboliques est détecté ;
- EMFILE : le nombre maximal de fichiers ouverts par le processus utilisant la fonction est atteint ;
- ENOMEM : la mémoire est insuffisante pour réaliser l'opération ;
- EROFS : le système de gestion de fichiers est en lecture seule ;
- ENOSPC : le système de fichiers est saturé.

13.3.2.2. Destruction d'un répertoire

Un répertoire nom_rep est détruit par un appel de la fonction :

```
int rmdir(const char *nom_rep)
```

En cas d'échec, elle retourne la valeur -1 et la variable errno prend l'une des valeurs suivantes :

- EACESS : l'accès n'est pas possible ;
- EBUSY : le répertoire à détruire est un répertoire courant de processus ou un répertoire racine ;
- EFAULT : le chemin spécifié n'est pas valide ;
- ENAMETOOLONG : le chemin spécifié comprend un nom trop long ;
- ENOENT : le répertoire désigné n'existe pas ;
- ELOOP : un cycle de liens symboliques est détecté ;
- ENOTEMPTY : le répertoire à détruire n'est pas vide ;
- ENOMEM : la mémoire est insuffisante pour réaliser l'opération ;
- EROFS : le système de gestion de fichiers est en lecture seule.

13.3.2.3. Exploration d'un répertoire

Trois fonctions permettent aux processus d'accéder aux entrées d'un répertoire. Ces sont les appels systèmes opendir(), readdir() et closedir(), qui permettent respectivement d'ouvrir un répertoire, de lire une entrée d'un répertoire et de fermer un répertoire. Voici leurs prototypes :

```
DIR *opendir(const char * nom_rep);
```

La fonction ouvre le répertoire nom_rep en lecture et renvoie un descripteur pour le répertoire ouvert. Le type DIR défini dans <dirent.h> représente un descripteur de répertoire ouvert.

```
struct dirent *readdir(DIR * rep);
```

Lit une entrée du répertoire désigné par le descripteur rep et la retourne dans une structure de type struc_dirent. La structure struct_dirent correspond à une entrée de répertoire. Elle comprend les champs suivants :

- long d_ino, le numéro d'inode de l'entrée ;
- unsigned short d_reclen, la taille de la structure retournée ;
- char [] d_name, le nom de fichier de l'entrée considérée.

```
int closedir(DIR * rep);
```

Ferme le répertoire désigné par le descripteur rep.

Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>

int main()
{
  char nom[50];
  struct dirent *entree;
  DIR *fd;
  getcwd(nom, 50);
  printf("Mon répertoire courant est %s\n", nom);
  fd = opendir(nom);
  entree = readdir(fd);
  while (entree != NULL)
  {
    printf("Le numéro d'inode de l'entrée est %d et le
           nom de fichier correspondant est %s\n",
           entree->d_ino, entree->d_name);
    entree = readdir(fd);
  }
  closedir(fd);
  return 0;
}
```

14. LE PRÉPROCESSEUR

Notions : préprocesseur, `#define`, `#include`, `#if`, `#elif`, `#ifdef`, `#endif`, `#undef`, `##`.

Le préprocesseur est appelé au début d'une phase de compilation pour modifier le source.

- **La directive** `#define`

```
#define symbole chaine
```

remplace un symbole par une chaîne dans le code source du programme à chaque fois qu'il apparaît.

Exemple :

```
#define MAX 256
```

Le symbole peut être paramétré :

```
#define max(a, b) ( (a)>(b) ? (a) : (b) )
```

Il est recommandé de parenthèser les arguments de la chaîne. Expliquez pourquoi l'exemple suivant produit un résultat erroné :

```
#define carre(a) (a*a)
...
carre(x+1);
```

- **La directive** `#include`

```
#include <fichier>
```

ou

```
#include "fichier"
```

permet d'inclure le code d'un autre fichier dans le code source de notre programme.

15. ALLOCATIONS DYNAMIQUES

Notions : `malloc()`, `calloc()`, `realloc()`, `free()`, listes chaînées, tableaux dynamiques.

15.1. Implantation des variables en mémoire

Dans les langages de bas niveau comme le langage C ou l'assembleur, le programmeur doit se préoccuper de l'implantation en mémoire de ses données.
Les variables locales des fonctions sont rangées sur la **pile** du processeur. Lorsque l'on entre dans la fonction, le pointeur de pile est remonté pour faire de la place aux variables locales, et il est diminué d'autant lorsque l'on quitte la fonction. Ce mécanisme simple évite de se préoccuper de savoir où ranger les variables en mémoire, assure que l'espace utilisé est libéré après usage, et permet que les fonctions soient « réentrantes », c'est-à-dire puissent être appelées plusieurs fois simultanément (fonctions récursives ou programmes multi-tâches).

L'allocation des variables sur la pile a aussi quelques inconvénients :

- La pile est de taille limitée (segment de pile de taille fixe), on ne peut y créer des variables de grandes tailles (par exemple un tableau représentant une image).
- Le compilateur doit connaître la taille de toutes les variables locales lorsqu'il génère le code de la fonction, afin de déterminer de quelle quantité il faut augmenter le pointeur de pile Ce point est très gênant lorsque l'on ne connaît pas à l'avance la taille des données à traiter (tableaux de taille variable).
- Dans certains cas, il est nécessaire de créer des variables qui « survivent » à la fonction.

Pour éviter ces limitations, le programme peut allouer de l'espace mémoire en dehors de la pile, dans une zone mémoire réservé à cet usage que l'on appelle souvent le **tas** (*heap*).

Un programme utilise donc 4 zones mémoires :

- le segment de code, qui contient ses instructions;
- le segment de pile (adresses de retour des fonctions, paramètres et variables locales);
- le segment de données (variables globales « statiques », que nous n'étudions pas dans ce cours);
- le tas, ou mémoire dynamique, pour stocker des données. La taille de cette zone varie au cours de l'exécution du programme.

La fonction `malloc()` (*memory allocation*) permet de réserver un espace mémoire dans le tas. Le résultat de cette fonction est l'adresse de début de l'espace alloué.
Lorsque le programme n'a plus besoin de l'espace alloué par `malloc()`, il doit le libérer en appelant la fonction `free()`.

15.2. Allocation dynamique de mémoire

Il y a allocation dynamique de mémoire quand une zone de mémoire n'est allouée que par une demande du programme au cours de son exécution. Un pointeur est alors créé, dont la référence

contient l'adresse de la zone de mémoire allouée où la variable pourra être stockée.
On utilise surtout les variables dynamiques pour créer des tableaux dont les dimensions ne sont pas connues au moment de la compilation, mais seulement lors de l'exécution du programme en fonction des paramètres qui lui sont alors donnés.
Les fonctions utilisées pour gérer l'allocation dynamique se trouvent dans la bibliothèque <alloc.h>, mais plusieurs parmi ces fonctions ne sont pas implémentées en C ANSI.

15.2.1. Les fonctions `malloc()` et `free()`

15.2.1.1. La fonction `malloc()`

Elle permet de faire une demande de réservation d'espace mémoire dans le tas:

```
void * malloc(unsigned int taille);
```

L'argument du type entier non signé est la taille du bloc que l'on demande dans le tas. Si l'allocation réussit, la fonction renvoie un pointeur sur le bloc alloué. Si la place disponible d'un seul tenant dans la mémoire est insuffisante ou que la taille demande est nulle, la fonction renvoie un pointeur NULL.
Le prototype de la fonction indique qu'elle renvoie un pointeur **void**. Il faudra donc, au moment de la demande, lui donner le type du pointeur qui correspond à celui que l'on créé.

```
char * chaine;
chaine = ( char * )malloc(dimension);
```

L'opérateur de conversion permet d'obtenir le type désiré, **char**. De même, si on a définit au préalable une structure Date:

```
struct Date *d0;
d0 = ( Date * )malloc(sizeof(Date)); //pointeur type Date
```

15.2.1.2. La fonction `free()`

Elle libère la mémoire quand elle a été allouée avec malloc() ou calloc() et realloc() que nous allons bientôt étudier.
ATTENTION, il faut surtout éviter de libérer deux fois de suite la même zone de mémoire ou de libérer une zone qui n'est pas allouée. Cela peut conduire sans problème à un superbe plantage du programme.

Exemple : **Allocation de mémoire pour une chaîne de caractères**

```
#include <stdio.h>
#include <string.h>   /* pour strncpy()          */
#include <stdlib.h>    /* pour malloc() et free() */
#include <process.h>  /* pour exit()             */

void lireTaille(unsigned *);
void lireChaine(char*, unsigned);

int main()
{
  char *chaine;
  unsigned dim;
```

```
  printf(" Entrez la dimension de la chaîne de
           caractères: ");
  lireTaille(&dim);
  chaine = ( char * )malloc(dim * sizeof(char));
  if (chaine == NULL)
  {
    printf("\n La mémoire disponible est insuffisante!\n");
    exit( 1);
  }
  printf(" Entrez la chaîne:  ");
  lireChaine(chaine, dim);
  printf("\n chaîne:  %s", chaine);
  free(chaine);
  return 0;
}

void lireChaine(char *chaine0, unsigned dim_chaine0)
{
  char *chaine1;
  if (( chaine1 = ( char * )malloc(2 * 256) ) == NULL)
  {
    printf("\n La mémoire disponible est insuffisante!\n");
    exit( 1);
  }
  *( chaine0 + dim_chaine0 - 1 ) = '\0';
  while (scanf("%255[^\n]", chaine1)!= 1)
    while (getchar() != '\n') ;
  while (getchar() != '\n') ;
  strncpy(chaine0, chaine1, dim_chaine0- 1);
  free(chaine1);
}

void lireTaille(unsigned* entier_u)
{
  double dble;
  do
  {
    while (scanf("%lf", &dble)!= 1)
      while (getchar() != '\n');
    while( getchar() != '\n');
  } while (dble < 0.0 || dble > 65535.0);
  *entier_u = (unsigned)dble;
}

/*
 Entrez la dimension de la chaîne de caractères: 16
 Entrez la chaîne:  OBELIX Idefix!!!
 Affichage de la chaîne:  OBELIX Idefix!                */
```

Le programme permet de créer dynamiquement une chaîne dont on vient de saisir la dimension, de l'initialiser et de l'afficher.

1) Nous avons respecté, pour la dimension de la chaîne, le type `unsigned int` de l'argument de `malloc()`. Vous pouvez passez un argument de type `int` à cette fonction, le compilateur fera la conversion et si par malheur l'argument est négatif, `malloc()` renverra un pointeur `NULL`.
2) Nous avons utilisé l'opérateur de conversion pour rendre le pointeur renvoyé conforme au type de la variable. La boucle **`if`** permet de sortir du programme sans trop de casse grâce à

l'instruction exit(1) si l'allocation ne s'est pas faite et donc que le pointeur renvoyé est NULL. La fonction free(chaine) libère la mémoire à la fin du programme. C'est une bonne habitude à prendre, bien que dans ce cas particulier ce ne soit pas nécessaire puisque le programme se ferme aussitôt. Dans une application normale ce programme ne serait qu'un module parmi d'autres et il serait alors indispensable de libérer la mémoire allouée.

3) La fonction lireChaine() utilise elle aussi l'allocation dynamique pour la chaîne source et pouvoir employer strncpy() sans surprises avec une chaîne cible créée dynamiquement.

Exemple : **Allocation de mémoire pour un tableau**

```
#include <stdio.h>
#include <stdlib.h>   /* pour malloc() et free()  */
#include <process.h> /* pour exit()             */

void lireTaille(unsigned*);
void lireDouble(double*);
void lireTab(double*, unsigned);
void ecrireTab(double*, unsigned);

int main()
{
  double *tab; /* Nom du tableau           */
  unsigned n;  /* Dimension du tableau     */
  printf(" Entrez la dimension du tableau: ");
  lireTaille(&n);
  tab = ( double * )malloc(n * sizeof(double));
  if (tab == NULL)
  {
    printf("\n La mémoire disponible est insuffisante!\n");
    exit( 1);
  }
  lireTab(tab, n);
  ecrireTab(tab, n);
  free(tab);
  return 0;
}

void lireTab(double *tableau, unsigned dim_tab)
{
  unsigned i;
  double *ptr;
  ptr = tableau;
  printf("\n (tableau)= \n");
  for (i = 0; i < dim_tab; i++)
  {
    printf("\t   [%u]= ", i);
    lireDouble(ptr++);
  }
}

void ecrireTab(double *tableau, unsigned dim_tab)
{
  unsigned i;
  double *ptr;
  ptr = tableau;
  printf("\n (tableau)= ");
  for (i = 0; i < dim_tab; i++)
```

```
    printf("\n\t   [%u]= %lf", i, *( ptr++));
}

void lireTaille(unsigned* entier_u)
{
  double dble;
  do
  {
    while (scanf("%lf", &dble)!= 1)
      while (getchar() != '\n');
    while (getchar() != '\n');
  } while (dble < 0.0 || dble > 65535.0);
  *entier_u = (unsigned)dble;
}

void lireDouble(double *reel)
{
  while (scanf("%lf", reel)!= 1)
    while (getchar() != '\n');
  while (getchar() != '\n');
}
```

C'est un second exemple type. Il n'y a rien de bien nouveau, par soucis d'homogénéité nous n'avons utilisé que des variables entières non signées. Dans `lireTab()` et `ecrireTab()` nous avons créé un pointeur de transit `*ptr` pour ne pas avoir à modifier la valeur de `tab`.

15.2.1.3. La fonction `calloc()`

```
void * calloc(unsigned nombre, unsigned taille);
```

Elle permet d'allouer dans le « tas » un bloc de (`nombre*_taille`) octets. Le bloc est initialisé à zéro. Comme `malloc()`, elle renvoie un pointeur sur le bloc réservé, ou un pointeur `NULL`, si la place manque, ou que les valeurs `nombre` ou `taille` sont nulles.
Dans le premier exemple, remplacez la ligne de code qui appelle la fonction `malloc()` par:

```
chaine = (char *)calloc(dim, sizeof(char));
```

La fonction réservera un bloc de (`dim*_2`) octets et initialisera le bloc à zéro ce qui est parfois un avantage déterminant pour l'usage de cette fonction.

15.2.1.4. La fonction `realloc()`

```
void * realloc(void * espace, unsigned taille);
```

Cette fonction permet de modifier la taille du bloc `espace` déjà créé avec `malloc()`, `calloc()` ou `realloc()`, pour lui donner la nouvelle dimension `taille`. Si le nouveau bloc est plus petit que l'ancien, elle renverra l'adresse de l'ancien bloc. S'il est plus grand, elle pourra l'ajuster quand la place nécessaire est disponible ou bien le recopier ailleurs et renvoyer dans ce cas une nouvelle adresse. En cas d'échec de cette réallocation, si aucune zone mémoire d'un seul tenant n'est disponible ou si l'argument `taille` est nul, la fonction renvoie un pointeur `NULL`, et elle est donc alors équivalente à un nouveau `malloc()`.

16. DÉBOGAGE / *DEBUG* À L'AIDE DE GDB

16.1. Commandes principales

16.1.1. La pose de points d'arrêt

(b**)reak** `[line]` : place un point d'arrêt à la ligne indiquée.

(b**)reak** `[fonc]` : place un point d'arrêt sur la fonction spécifiée.

`info break` : indique où sont définis les point d'arrêts.

`clear [line|fonc]` : supprime un point d'arrêt.

16.1.2. Exécution du programme

`run < file` : lance le programme avec une redirection de l'entrée standard.

(n**)ext** : exécute une instruction sans rentrer dans le code des fonctions.

(s**)tep** : exécute une instruction en entrant dans le code des fonctions.

(c**)ount** : continue l'exécution du programme.

`jump [line]` : saute a la ligne indiquée (modifie le compteur ordinal).

(l**)ist** : liste le code source du programme.

16.1.3. Examen des données

(p**)rint** `[exp]` : affiche la valeur de l'expression.

(p**)rint** `[*tab@num]` : affiche num valeurs du tableau tab.

`display [exp]` : affiche la valeur de l'expression après chaque arrêt.

`undisplay [num]` : supprime un affichage.

`set [exp=value]` : modifie la valeur d'une variable.

17. Programmation système sous Linux

Par Jérôme Geydan (ENSTA) et Alain Lebret (Lycée Diderot)

Nous avons déjà abordé dans le chapitre 13 les accès aux entrées/sorties sous Linux. Nous allons à présent nous tourner vers la notion de processus déjà aperçue dans la première partie, et terminerons par la programmation des ports parallèle et série.

17.1. Gestion des processus

17.1.1. Les processus

Les ordinateurs dotés de systèmes d'exploitation modernes peuvent exécuter « en même temps » plusieurs programmes pour plusieurs utilisateurs différents : ces machines sont dites **multitâches** et multi-utilisateurs. Un **processus** (ou *process* en anglais) est un ensemble d'instructions se déroulant séquentiellement sur le processeur : par exemple, un de vos programmes en C, ou encore le terminal de commandes qui interprète les entrées au clavier.
Sous UNIX, la commande **ps** permet de voir la liste des processus existant sur une machine : **ps -ux** donne la liste des processus que vous avez lancé sur la machine, avec un certain nombre d'informations sur ces processus. En particulier, la première colonne donne le nom de l'utilisateur ayant lancé le processus et la dernière le contenu du tableau `argv` du processus. **ps -aux** donne la liste de tout les processus lancés sur la machine.
Chaque processus, à sa création, se voit attribuer un numéro d'identification (le PID). C'est ce numéro qui est utilisé ensuite pour désigner un processus. Ce numéro se trouve dans la deuxième colonne de la sortie de la commande **ps -aux**.
Un processus ne peut être créé qu'à partir d'un autre processus, sauf le premier, **init**, qui est créé par le système au démarrage de la machine. Chaque processus a donc un ou plusieurs fils, et un père, ce qui crée une structure arborescente. À noter que certains systèmes d'exploitation peuvent utiliser des processus pour leur gestion interne, dans ce cas le PID du processus **init**, le premier processus en mode utilisateur, sera supérieur à 1.

17.1.2. Les fonctions d'identification des processus

Ces fonctions sont au nombre de deux :

- `pid_t getpid(void)`
  Cette fonction retourne le PID du processus.
- `pid_t getppid(void)`
  Cette fonction retourne le PID du processus père. Si le processus père n'existe plus (parce qu'il s'est terminé avant le processus fils, par exemple), la valeur retournée est celle qui correspond au processus **init**, en général de PID 1, ancêtre de tous les autres et qui ne se termine jamais.

Le type `pid_t` est en fait équivalent à un type **int** et a été défini dans un fichier d'en-tête par un appel à **typedef**.
Le programme suivant affiche le PID du processus créé ainsi que celui de son père :

```
#include <sys/types.h>
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf("Mon PID est : %d\n", (int)getpid());
    printf("Le PID de mon père est : %d\n", (int)getppid());
    exit(EXIT_SUCCESS);
}
```

La commande **echo $$** entrée au clavier retourne le PID du *shell* interprétant cette commande. Le programme suivant peut donc donner le résultat suivant (les numéros de processus étant uniques, deux exécutions successives ne donneront pas le même résultat) :

```
echo $$
189
./ex21
mon PID est : 3162
le PID de mon père est : 189
```

17.1.3. La création de processus

La bibliothèque C propose plusieurs fonctions pour créer des processus avec des interfaces plus ou moins perfectionnées. Cependant toute ces fonctions utilisent l'appel système `fork()` qui est la seule et unique façon de demander au système d'exploitation de créer un nouveau processus.

- `pid_t fork(void)`
  Cet appel système crée un nouveau processus. La valeur retournée est le PID du fils pour le processus père, ou 0 pour le processus fils. La valeur -1 est retournée en cas d'erreur. `fork()` se contente de dupliquer un processus en mémoire, c'est-à-dire que la zone mémoire du processus père est recopiée dans la zone mémoire du processus fils. On obtient alors deux processus identiques, déroulant le même code en concurrence. Ces deux processus ont les mêmes allocations mémoire et les mêmes descripteurs de fichiers. Seuls, le PID, le PID du père et la valeur retournée par `fork()` sont différents. La valeur retournée par `fork()` est en général utilisée pour déterminer si on est le processus père ou le processus fils et permet d'agir en conséquence.

L'exemple suivant montre l'effet de `fork()` :

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int i;

  printf("[processus %d] avant le fork\n", getpid());
  i = fork();
  printf("[processus %d] après le fork retournant %d\n",
         getpid(), i);
  exit(EXIT_SUCCESS);
}
```

Le résultat de l'exécution est :

```
./ex22
```

```
[processus 1197] avant le fork
[processus 1197] après le fork retournant 1198
[processus 1198] après le fork retournant 0
```

Le premier **printf()** est avant l'appel à fork(). À ce moment-là, il n'y a qu'un seul processus, donc un seul message « avant le fork ». En revanche, le second printf() se trouve après le fork() et il est donc exécuté deux fois par deux processus différents. Notons que la variable i, qui contient la valeur de retour du fork() contient deux valeurs différentes.
Lors de l'exécution, l'ordre dans lequel le fils et le père affichent leurs informations n'est pas toujours le même. Cela est dû à l'ordonnancement des processus par le système d'exploitation.

17.1.4. L'appel système wait()

Il est souvent très pratique de pouvoir attendre la fin de l'exécution des processus fils avant de continuer l'exécution du processus père (afin d'éviter que celui-ci se termine avant ses fils, par exemple). La fonction : pid_t wait(int *status) permet de suspendre l'exécution du père jusqu'à ce que l'exécution d'un des fils soit terminée. La valeur retournée est le PID du processus qui vient de se terminer ou -1 en cas d'erreur. Si le pointeur status est différent de NULL, les données retournées contiennent des informations sur la manière dont ce processus s'est terminé, comme par exemple la valeur passée à exit() (voir le manuel en ligne pour plus d'informations, **man 2 wait**).
Dans l'exemple suivant, nous complétons le programme décrit précédemment en utilisant wait() : le père attend alors que le fils soit terminé pour afficher les informations sur le fork() et ainsi s'affranchir de l'ordonnancement aléatoire des tâches.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
  int i,j;

  printf("[processus %d] avant le fork\n", getpid());
  i = fork();
  if (i != 0) { /* i != 0 seulement pour le père */
    j = wait(NULL);
    printf("[processus %d] wait retourne %d\n",getpid(),j);
  }
  printf("[processus %d] après le fork retournant %d\n",
         getpid(), i);
  exit(EXIT_SUCCESS);
}
```

L'exécution donne :

```
[processus 1203] avant le fork
[processus 1204] après le fork retournant 0
[processus 1203] wait retourne 1204
[processus 1203] après le fork retournant 1204
```

Notons que cette fois-ci, l'ordre dans lequel les informations s'affichent est toujours le même.

17.2. Recouvrement de processus

17.2.1. Le recouvrement de processus

Dans le chapitre précédent, nous avons vu comment créer de nouveaux processus grâce à l'appel système `fork()`. Cet appel système effectue une duplication de processus, c'est-à-dire que le processus nouvellement créé contient les mêmes données et exécute le même code que le processus père. Le recouvrement de processus permet de remplacer par un autre code le code exécuté par un processus. Le programme et les données du processus sont alors différents, mais celui-ci garde le même PID, le même père et les même descripteurs de fichiers.
C'est ce mécanisme qui est utilisé lorsque, par exemple, nous tapons la commande **`ls`**, ou que nous lançons un programme après l'avoir compilé en tapant **`./a.out`** :

- le terminal de commande dans lequel cette commande est tapée fait un `fork()` ;
- le processus ainsi créé (le fils du terminal de commande) est recouvert par l'exécutable désiré ;
- pendant ce temps, le terminal de commande (le père) attend que le fils se termine grâce à `wait()` si la commande n'a pas été lancée en tâche de fond.

17.2.2. Les appels système de recouvrement de processus

Ces appels système sont au nombre de 6 :

- `int execl(char *path,char *arg0,char *arg1,...,char *argn,(char *)0)`
- `int execv(char *path, char *argv[])`
- `int execle(char *path,char *arg0,char *arg1,...,char *argn,(char *)0,char **envp)`
- `int execlp(char *file,char *arg0,char *arg1,...,char *argn,(char *)0)`
- `int execvp(char *file, char *argv[])`
- `int execve(char *path, char *argv[], char **envp)`

Notons que le code présent après l'appel à une de ces fonctions ne sera jamais exécuté, sauf en cas d'erreur. Nous décrirons ici uniquement les appels système `execv()` et `execlp()` :

- `int execv(char *path, char *argv[])`
  Cette fonction recouvre le processus avec l'exécutable indiqué par la chaîne de caractère `path` (`path` est le chemin d'accès à l'exécutable). `argv` est un tableau de chaînes de caractères contenant les arguments à passer à l'exécutable. Le dernier élément du tableau doit contenir `NULL`. Le premier est la chaîne de caractères qui sera affichée par **`ps`** et, par convention, on y met le nom de l'exécutable.

  Exemple :
  ```
  ...
  argv[0] = "ls";
  argv[1] = "-l";
  argv[2] = NULL;
  execv("/bin/ls", argv);
  ```

- `int execlp(char *file, char *arg0, char *arg1, ..... , char *argn, (char *)0);`
  Cette fonction recouvre le processus avec l'exécutable spécifié par `file` (le fichier utilisé est

cherché dans les répertoires contenus dans la variable d'environnement `$PATH` si la chaîne `file` n'est pas un chemin d'accès absolu ou relatif). Les arguments suivants sont les chaînes de caractères passées en argument à l'exécutable. Comme pour `execv()`, le premier argument doit être le nom de l'exécutable et le dernier `NULL`.

Exemple :
```
execlp("ls","ls","-l",NULL);
```

Notons qu'il n'est pas nécessaire ici de spécifier le chemin d'accès (**`/bin/ls`**) pour l'exécutable.

Pour recouvrir un processus avec la commande **`ps -aux`**, nous pouvons utiliser le code suivant :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *arg[3];
  arg[0] = "ps";
  arg[1] = "-aux";
  arg[2] = NULL;
  execv("/bin/ps", arg); /* l'exécutable est dans /bin */
  fprintf(stderr, "erreur dans execv\n");
  exit(EXIT_FAILURE);
}
```

Dans le chapitre 10, nous utilisions les variables passées à la fonction `int main(int argc, char **argv)` pour récupérer les arguments de la ligne de commande. Ces valeurs sont en fait déterminées par le terminal de commande et c'est lui qui les « passent » à la commande exécutée, grâce à un appel à `exec()`.
Ainsi, lorsque nous tapons une commande comme **`ps -aux`**, le terminal de commande exécute (de manière simplifiée) :

```
int res;
...
res = fork();

if (res == 0) { /* on est le fils */
  execlp("ps", "ps", "-aux", NULL);
  fprintf(stderr, "erreur\n");
  /* si execlp retourne, quelque chose ne va pas */
  exit(EXIT_FAILURE);
}
```

17.2.3. Les signaux

17.2.3.1. Introduction

Les signaux constituent la forme la plus simple de communication entre processus. Un signal est une information atomique envoyée à un processus ou à un groupe de processus par le système d'exploitation ou par un autre processus. Lorsqu'un processus reçoit un signal, le système d'exploitation l'informe : « tu as reçu tel signal », sans plus. Un signal ne transporte donc aucune autre information utile.

Lorsqu'il reçoit un signal, un processus peut réagir de trois façons :

- Il est immédiatement dérouté vers une fonction spécifique, qui réagit au signal (en modifiant la valeur de certaines variables ou en effectuant certaines actions, par exemple). Une fois cette fonction terminée, on reprend le cours normal de l'exécution du programme, comme si rien ne s'était passé.
- Le signal est tout simplement ignoré.
- Le signal provoque l'arrêt du processus (avec ou sans génération d'un fichier `core`).

Lorsqu'un processus reçoit un signal pour lequel il n'a pas indiqué de fonction de traitement, le système d'exploitation adopte une réaction par défaut qui varie suivant les signaux :

- soit il ignore le signal ;
- soit il termine le processus (avec ou sans génération d'un fichier `core`).

Vous avez certainement toutes et tous déjà utilisé des signaux, consciemment, en tapant `<Ctrl-C>` ou en employant la commande **`kill`**, ou inconsciemment, lorsqu'un de vos programme a affiché

```
segmentation fault (core dumped)
```

17.2.3.2. Liste et signification des différents signaux

La liste des signaux dépend du type d'Unix. La norme POSIX.1 en spécifie un certain nombre, parmi les plus répandus. On peut néanmoins dégager un grand nombre de signaux communs à toutes les versions d'Unix :

- `SIGHUP` **:**
  rupture de ligne téléphonique. Du temps où certains terminaux étaient reliés par ligne téléphonique à un ordinateur distant, ce signal était envoyé aux processus en cours d'exécution sur l'ordinateur lorsque la liaison vers le terminal était coupée. Ce signal est maintenant utilisé pour demander à des démons (processus lancés au démarrage du système et tournant en tâche de fond) de relire leur fichier de configuration.
- `SIGINT` **:** interruption. C'est le signal qui est envoyé à un processus quand on tape `<Ctrl-C>` au clavier.
- `SIGFPE` **:** erreur de calcul en virgule flottante, le plus souvent une division par zéro.
- `SIGKILL` **:** tue le processus.
- `SIGBUS` **:** erreur de bus.
- `SIGSEGV` **:** violation de segment, généralement à cause d'un pointeur nul.
- `SIGPIPE` **:** tentative d'écriture dans un tuyau qui n'a plus de lecteurs.
- `SIGALRM` **:** alarme (chronomètre).
- `SIGTERM` **:** demande au processus de se terminer proprement.
- `SIGCHLD` **:** indique au processus père qu'un de ses fils vient de se terminer.
- `SIGWINCH` **:** indique que la fenêtre dans lequel tourne un programme a changé de taille.
- `SIGUSR1` **:** signal utilisateur 1.
- `SIGUSR2` **:** signal utilisateur 2.

Il n'est pas possible de dérouter le programme vers une fonction de traitement sur réception du signal `SIGKILL`, celui-ci provoque toujours la fin du processus. Ceci permet à l'administrateur système de supprimer n'importe quel processus.
À chaque signal est associé un numéro. Les correspondances entre numéro et nom des signaux se trouvent généralement dans le fichier `/usr/include/signal.h` ou dans le fichier `/usr/include/sys/signal.h` suivant le système.

17.2.3.3. Envoi d'un signal

a) *Depuis un interpréteur de commandes*

La commande **kill** permet d'envoyer un signal à un processus dont on connaît le numéro (il est facile de le déterminer grâce à la commande **ps**) :

```
kill -HUP 1664
```

Ici, on envoie le signal SIGHUP au processus numéro 1664 (notez qu'en utilisant la commande **kill**, on écrit le nom du signal sous forme abrégée, sans le SIG initial).
On aurait également pu utiliser le numéro du signal plutôt que son nom :

```
kill -1 1664
```

On envoie le signal numéro 1 (SIGHUP) au processus numéro 1664.

b) *Depuis un programme en C*

La fonction C kill() permet d'envoyer un signal à un processus :

Ici, on envoie le signal SIGHUP au processus numéro 1664.
On aurait pu directement mettre 1 à la place de SIGHUP, mais l'utilisation des noms des signaux rend le programme plus lisible et plus portable (le signal SIGHUP a toujours le numéro 1 sur tous les systèmes, mais ce n'est pas le cas de tous les signaux).

17.2.3.4. Interface de programmation

L'interface actuelle de programmation des signaux, qui respecte la norme POSIX.1, repose sur la fonction sigaction(). L'ancienne interface, qui utilisait la fonction signal(), est à proscrire pour des raisons de portabilité. Nous en parlerons néanmoins rapidement pour indiquer ses défauts.

a) *La fonction `sigaction()`*

La fonction sigaction() indique au système comment réagir sur réception d'un signal. Elle prend comme paramètres :

1. Le numéro du signal auquel on souhaite réagir.
2. Un pointeur sur une structure de type sigaction. Dans cette structure, deux membres nous intéressent :
   - sa_handler, qui peut être :
     - un pointeur vers la fonction de traitement du signal ;
     - SIG_IGN pour ignorer le signal ;
     - SIG_DFL pour restaurer la réaction par défaut.
   - sa_flags, qui indique des options liées à la gestion du signal. Étant donné l'architecture du noyau UNIX, un appel système interrompu par un signal est toujours avorté et renvoie EINTR au processus appelant. Il faut alors relancer cet appel système. Néanmoins, sur les UNIX modernes, il est possible de demander au système de redémarrer automatiquement certains appels système interrompus par un signal. La constante SA_RESTART est utilisée à cet effet.

   Le membre sa_mask de la structure sigaction indique la liste des signaux devant être bloqués pendant l'exécution de la fonction de traitement du signal. On ne veut généralement bloquer aucun signal, c'est pourquoi on initialise sa_mask à zéro au moyen de la fonction

sigemptyset().

3. Un pointeur sur une structure de type sigaction, structure qui sera remplie par la fonction selon l'ancienne configuration de traitement du signal. Ceci ne nous intéresse pas ici, d'où l'utilisation d'un pointeur nul.

La valeur renvoyée par sigaction() est :

0 si tout s'est bien passé.

-1 si une erreur est survenue. Dans ce cas l'appel à sigaction() est ignoré par le système.

Ainsi, dans l'exemple ci-dessous, à chaque réception du signal SIGUSR1, le programme sera dérouté vers la fonction traiterSignal(), puis reprendra son exécution comme si de rien n'était. En particulier, les appels système qui auraient été interrompus par le signal seront relancés automatiquement par le système.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* prototype de la fonction gestionnaire de signal */
/* le paramètre est de type int (imposé), et il n'y */
/* a pas de valeur de retour */
void traiterSignal(int sig);

int main(int argc, char *argv[]) {
  struct sigaction act;

  /* une affectation de pointeur de fonction */
  /* c'est équivalent à écrire : */
  /* act.sa_handler = &traiterSignal */
  act.sa_handler = traiterSignal;
  /* le masque des signaux non pris en compte est mis */
  /* à l'ensemble vide (aucun signal n'est ignoré) */
  sigemptyset(&act.sa_mask);
  /* Les appels systèmes interrompus par un signal */
  /* seront repris au retour du gestionnaire de signal */
  act.sa_flags = SA_RESTART;
  /* enregistrement de la réaction au SIGUSR1 */
  if ( sigaction(SIGUSR1,&act,NULL) == -1 ) {
    /* perror permet d'afficher la chaîne avec */
    /* le message d'erreur de la dernière commande */
    perror("sigaction");
    exit(EXIT_FAILURE);
  }

  printf("Je suis le processus numéro %d.\n" ,getpid());

  for(;;) { /* boucle infinie équivalente à while (1) */
    sigset_t sigmask; /* variable locale à cette boucle */
    sigemptyset(&sigmask); /* mask = ensemble vide */
    /* on interrompt le processus jusqu'à l'arrivée */
    /* d'un signal (mask s'il n'était pas vide */
    /* correspondrait aux signaux ignorés) */
    sigsuspend(&sigmask);
    printf("Je viens de recevoir un signal et de le
           traiter\n");
  }
```

```
  exit(EXIT_SUCCESS);
}

void traiterSignal(int sig) {
  printf("Réception du signal numéro %d.\n", sig);
}
```

La fonction sigsuspend() utilisée dans la boucle infinie permet de suspendre l'exécution du programme jusqu'à réception d'un signal. Son argument est un pointeur sur une liste de signaux devant être bloqués. Comme nous ne désirons bloquer aucun signal, cette liste est mise à zéro au moyen de la fonction sigemptyset(). Les programmes réels se contentent rarement d'attendre l'arrivée d'un signal sans rien faire, c'est pourquoi la fonction sigsuspend() est assez peu utilisée dans la réalité.

b) *La fonction `signal()`*

La vieille interface de traitement des signaux utilise la fonction signal() au lieu de sigaction().

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void traiterSignal (int sig);

int main(int argc, char *argv[]) {
  if ( signal(SIGUSR1,traiterSignal ) == SIG_ERR ) {
    perror("signal");
    exit(EXIT_FAILURE);
  }
  printf("Je suis le processus numéro %d.\n", getpid());
  for (;;) {
    pause();
  }
  exit(EXIT_SUCCESS);
}

void traiterSignal (int sig) {
   printf("Réception du signal numéro %d.\n", sig);
}
```

La fonction signal() indique au système comment réagir sur réception d'un signal. Elle prend comme paramètres :

1. Le numéro du signal auquel on s'intéresse.
2. Une valeur qui peut être :
   - un pointeur vers la fonction de traitement du signal ;
   - SIG_IGN pour ignorer le signal ;
   - SIG_DFL pour restaurer la réaction par défaut.

De manière évidente, la fonction signal() est plus limitée que la fonction sigaction() parce qu'elle ne permet pas d'indiquer les options de traitement du signal, en particulier le comportement que l'on souhaite pour les appels système interrompus par la réception du signal.
Elle a aussi un effet de bord très vicieux par rapport à la fonction sigaction, c'est pourquoi il ne faut plus utiliser la fonction signal().

La norme POSIX.1, à laquelle la fonction `sigaction()` est conforme, spécifie que, si une fonction de traitement a été indiquée pour un signal, elle doit être appelée à chaque réception de ce signal et c'est bien ce qui se passe lorsqu'on utilise `sigaction()`.
En revanche, lorsqu'on utilise la fonction `signal()`, le comportement du système peut être différent :

- Sur les UNIX BSD, tout se passe comme si l'on avait utilisé `sigaction()` et la fonction de traitement est bien appelée chaque fois qu'on reçoit le signal.
- Sur les UNIX System V, en revanche, la fonction de traitement est bien appelée la première fois qu'on reçoit le signal, mais elle ne l'est pas si on le reçoit une seconde fois. Il faut alors refaire un appel à la fonction `signal()` dans la fonction de traitement pour rafraîchir la mémoire du système.

Cette différence de comportement oblige à gérer les signaux de deux manières différentes suivant la famille d'UNIX.

17.2.3.5. Conclusion

Les signaux sont la forme la plus simple de communication entre processus. Cependant, ils ne permettent pas d'échanger des données.
En revanche, de par leur traitement asynchrone, ils peuvent être très utiles pour informer les processus de conditions exceptionnelles (relecture de fichier de configuration après modification manuelle, par exemple) ou pour synchroniser des processus.

17.2.4. Les tuyaux

17.2.4.1. Introduction

Les tuyaux (ou *pipes* en anglais) permettent à un groupe de processus d'envoyer des données à un autre groupe de processus. Ces données sont envoyées directement en mémoire sans être stockées temporairement sur disque, ce qui est donc très rapide.
Tout comme un tuyau de plomberie, un tuyau de données a deux côtés : un côté permettant d'écrire des données dedans et un côté permettant de les lire. Chaque côté du tuyau est un descripteur de fichier ouvert soit en lecture soit en écriture, ce qui permet de s'en servir très facilement, au moyen des fonctions d'entrée / sortie classiques.
La lecture d'un tuyau est bloquante, c'est-à-dire que si aucune donnée n'est disponible en lecture, le processus essayant de lire le tuyau sera suspendu, il ne sera pas pris en compte par l'ordonnanceur et n'occupera donc pas inutilement le processeur jusqu'à ce que des données soient disponibles. L'utilisation de cette caractéristique comme effet de bord peut servir à synchroniser des processus entre eux (les processus lecteurs étant synchronisés sur les processeur écrivains).
La lecture d'un tuyau est destructrice, c'est-à-dire que si plusieurs processus lisent le même tuyau, toute donnée lue par l'un disparaît pour les autres. Par exemple, si un processus écrit les deux caractères `ab` dans un tuyau lu par les processus **A** et **B** et que **A** lit un caractère dans le tuyau, il lira le caractère `a` qui disparaîtra immédiatement du tuyau sans que **B** puisse le lire. Si **B** lit alors un caractère dans le tuyau, il lira donc le caractère `b` que **A**, à son tour, ne pourra plus y lire. Si l'on veut donc envoyer des informations identiques à plusieurs processus, il est nécessaire de créer un tuyau vers chacun d'eux.
De même qu'un tuyau en cuivre a une longueur finie, un tuyau de données à une capacité finie. Un processus essayant d'écrire dans un tuyau plein se verra suspendu en attendant qu'un espace suffisant se libère.

Vous avez sans doute déjà utilisé des tuyaux. Par exemple, lorsque vous tapez

```
ls | wc -l
```

l'interprète de commandes relie la sortie standard de la commande **ls** à l'entrée standard de la commande **wc** au moyen d'un tuyau.
Les tuyaux sont très utilisés sous UNIX pour faire communiquer des processus entre eux. Ils ont cependant deux contraintes :

- les tuyaux ne permettent qu'une communication unidirectionnelle ;
- les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un ancêtre commun.

17.2.4.2. L'appel système `pipe()`

Un tuyau se crée très simplement au moyen de l'appel système `pipe()` :
L'argument de `pipe()` est un tableau de deux descripteurs de fichier (un descripteur de fichier est du type **int** en C) similaires à ceux renvoyés par l'appel système `open()` et qui s'utilisent de la même manière. Lorsque le tuyau a été créé, le premier descripteur, `tuyau[0]`, représente le côté lecture du tuyau et le second, `tuyau[1]`, représente le côté écriture.
Un moyen mnémotechnique pour se rappeler quelle valeur représente quel côté est de rapprocher ceci de l'entrée et de la sortie standard. L'entrée standard, dont le numéro du descripteur de fichier est toujours 0, est utilisée pour lire au clavier : 0 lecture. La sortie standard, dont le numéro du descripteur de fichier est toujours 1, est utilisée pour écrire à l'écran : 1 écriture.
Néanmoins, pour faciliter la lecture des programmes et éviter des erreurs, il est préférable de définir deux constantes dans les programmes qui utilisent les tuyaux :

17.2.4.3. Mise en place d'un tuyau

La mise en place d'un tuyau permettant à deux processus de communiquer est relativement simple. Prenons l'exemple d'un processus qui crée un fils auquel il va envoyer des données :

1. Le processus père crée le tuyau au moyen de `pipe()`.
2. Puis il crée un processus fils grâce à `fork()`. Les deux processus partagent donc le tuyau.
3. Puisque le père va écrire dans le tuyau, il n'a pas besoin du côté lecture, donc il le ferme.
4. De même, le fils ferme le côté écriture.
5. Le processus père peut dès lors envoyer des données au fils.

Le tuyau doit être créé avant l'appel à la fonction `fork()` pour qu'il puisse être partagé entre le processus père et le processus fils (les descripteurs de fichiers ouverts dans le père sont hérités par le fils après l'appel à `fork()`).
Comme indiqué dans l'introduction, un tuyau ayant plusieurs lecteurs peut poser des problèmes, c'est pourquoi le processus père doit fermer le côté lecture après l'appel à `fork()` (il n'en a de toute façon pas besoin). Il en va de même pour un tuyau ayant plusieurs écrivains donc le processus fils doit aussi fermer le côté écriture. Omettre de fermer le côté inutile peut entraîner l'attente infinie d'un des processus si l'autre se termine. Imaginons que le processus fils n'ait pas fermé le côté écriture du tuyau. Si le processus père se termine, le fils va rester bloqué en lecture du tuyau sans recevoir d'erreur puisque son descripteur en écriture est toujours valide. En revanche, s'il avait fermé le côté écriture, il aurait reçu un code d'erreur en essayant de lire le tuyau, ce qui l'aurait informé de la fin du processus père.
Le programme suivant illustre cet exemple, en utilisant les appels système `read()` et `write()` pour la lecture et l'écriture dans le tuyau :

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
  int tuyau[2], nb, i;
  char donnees[10];

  if (pipe(tuyau) == -1) { /* création du pipe */
    perror("Erreur dans pipe()");
    exit(EXIT_FAILURE);
  }

  switch (fork()) { /* les 2 processus partagent le pipe */
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]); /* on ferme l'écriture */
      /* on peut alors lire dans le pipe */
      nb = read(tuyau[LECTURE], donnees, sizeof(donnees));
      for (i = 0; i < nb; i++) {
         putchar(donnees[i]);
      }
      putchar('\n');
      close(tuyau[LECTURE]);
      exit(EXIT_SUCCESS);
    default :   /* processus père, écrivain */
      close(tuyau[LECTURE]); /* on ferme la lecture */
      strncpy(donnees, "bonjour", sizeof(donnees));
      /* on peut écrire dans le pipe */
      write(tuyau[ECRITURE], donnees, strlen(donnees));
      close(tuyau[ECRITURE]);
      exit(EXIT_SUCCESS);
   }
}
```

17.2.4.4. Utilisation des fonctions d'entrées / sorties standard avec les tuyaux

Puisqu'un tuyau s'utilise comme un fichier, il serait agréable de pouvoir utiliser les fonctions d'entrées / sorties standard (`fprintf()`, `fscanf()`...) au lieu de `read()` et `write()`, qui sont beaucoup moins pratiques. Pour cela, il faut transformer les descripteurs de fichiers en pointeurs de type `FILE *`, comme ceux retournés par `fopen()`.
La fonction `fdopen()` permet de le faire. Elle prend en argument le descripteur de fichier à transformer et le mode d'accès au fichier (`"r"` pour la lecture et `"w"` pour l'écriture) et renvoie le pointeur de type `FILE *` permettant d'utiliser les fonctions d'entrée/sortie standard.
L'exemple suivant illustre l'utilisation de la fonction `fdopen()` :

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1
```

```
int main (int argc, char *argv[]) {
   int tuyau[2];
   char str[100];
   FILE *mon_tuyau ;

   if ( pipe(tuyau) == -1 ) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      /* ouvre un descripteur de flot FILE * à partir */
      /* du descripteur de fichier UNIX */
      mon_tuyau = fdopen(tuyau[LECTURE], "r");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      /* mon_tuyau est un FILE * accessible en lecture */
      fgets(str, sizeof(str), mon_tuyau);
      printf("Mon père a écrit : %s\n", str);
      /* il faut faire fclose(mon_tuyau) ou à la rigueur */
      /* close(tuyau[LECTURE]) mais surtout pas les deux */
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
    default :   /* processus père, écrivain */
      close(tuyau[LECTURE]);
      mon_tuyau = fdopen(tuyau[ECRITURE], "w");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      /* mon_tuyau est un FILE * accessible en écriture */
      fprintf(mon_tuyau, "petit message\n");
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   }
}
```

Il faut cependant garder à l'esprit que les fonctions d'entrées / sorties standard utilisent une zone de mémoire tampon lors de leurs opérations de lecture ou d'écriture. L'utilisation de cette zone tampon permet d'optimiser, en les regroupant, les accès au disque avec des fichiers classiques mais elle peut se révéler particulièrement gênante avec un tuyau. Dans ce cas, la fonction `fflush()` peut se révéler très utile puisqu'elle permet d'écrire la zone tampon dans le tuyau sans attendre qu'elle soit remplie.
Il est à noter que l'appel à `fflush()` était inutile dans l'exemple précédent en raison de son appel implicite lors de la fermeture du tuyau par `fclose()`.

L'exemple suivant montre l'utilisation de la fonction `fflush()` :

```
#include <sys/types.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];
   char str[100];
   FILE *mon_tuyau;

   if ( pipe(tuyau) == -1 ) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      mon_tuyau = fdopen(tuyau[LECTURE], "r");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      fgets(str, sizeof(str), mon_tuyau);
      printf("[fils] Mon père a écrit : %s\n", str);
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   default :   /* processus père, écrivain */
      close(tuyau[LECTURE]);
      mon_tuyau = fdopen(tuyau[ECRITURE], "w");
      if (mon_tuyau == NULL) {
         perror("Erreur dans fdopen()");
         exit(EXIT_FAILURE);
      }
      fprintf(mon_tuyau, "un message de test\n");
      printf("[père] Je viens d'écrire dans le tuyau,\n");
      printf("     mais les données sont encore\n");
      printf("     dans la zone de mémoire tampon.\n");
      sleep(5);
      fflush(mon_tuyau);
      printf("[père] Je viens de forcer l'écriture\n");
      printf("     des données de la mémoire tampon\n");
      printf("     vers le tuyau.\n");
      printf("     J'attends 5 secondes avant de fermer\n");
      printf("     le tuyau et de me terminer.\n");
      sleep(5);
      fclose(mon_tuyau);
      exit(EXIT_SUCCESS);
   }
}
```

17.2.4.5. Redirection des entrée et sorties standard

Une technique couramment employée avec les tuyaux est de relier la sortie standard d'une commande à l'entrée standard d'une autre, comme quand on tape

```
ls | wc -l
```

Le problème, c'est que la commande **ls** est conçue pour afficher à l'écran et pas dans un tuyau. De même pour **wc** qui est conçue pour lire au clavier.
Pour résoudre ce problème, UNIX fournit une méthode élégante. Les fonctions dup() et dup2() permettent de dupliquer le descripteur de fichier passé en argument :

```
#include <unistd.h>
int dup(int descripteur);
int dup2(int descripteur, int copie);
```

Dans le cas de dup2(), on passe en argument le descripteur de fichier à dupliquer ainsi que le numéro du descripteur souhaité pour la copie. Le descripteur copie est éventuellement fermé avant d'être réalloué.
Pour dup(), le plus petit descripteur de fichier non encore utilisé permet alors d'accéder au même fichier que descripteur.
Mais comment déterminer le numéro de ce plus petit descripteur ? Sachant que l'entrée standard a toujours 0 comme numéro de descripteur et que la sortie standard a toujours 1 comme numéro de descripteur, c'est très simple lorsqu'on veut rediriger l'un de ces descripteurs (ce qui est quasiment toujours le cas). Prenons comme exemple la redirection de l'entrée standard vers le côté lecture du tuyau :

1. On ferme l'entrée standard (descripteur de fichier numéro 0 ou, de manière plus lisible, STDIN_FILENO, défini dans <unistd.h>). Le plus petit numéro de descripteur non utilisé est alors 0.
2. On appelle dup() avec le numéro de descripteur du côté lecture du tuyau comme argument. L'entrée standard est alors connectée au côté lecture du tuyau.
3. On peut alors fermer le descripteur du côté lecture du tuyau qui est maintenant inutile puisqu'on peut y accéder par l'entrée standard.

La façon de faire pour connecter la sortie standard avec le côté écriture du tuyau est exactement la même.
Le programme suivant montre comment lancer **ls | wc -l** en utilisant dup2() :

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

int main(int argc, char *argv[]) {
   int tuyau[2];

   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
   case -1 :   /* erreur */
```

```
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
   case 0 :   /* processus fils, ls , écrivain */
      close(tuyau[LECTURE]);
      /* dup2 va brancher le côté écriture du tuyau */
      /* comme sortie standard du processus courant */
      if (dup2(tuyau[ECRITURE], STDOUT_FILENO) == -1) {
         perror("Erreur dans dup2()");
      }
      /* on ferme le descripteur qui reste pour */
      /* << éviter les fuites >> ! */
      close(tuyau[ECRITURE]);
      /* ls en écrivant sur stdout envoie en fait dans le */
      /* tuyau sans le savoir */
      if (execlp("ls", "ls", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
   default :   /* processus père, wc , lecteur */
      close(tuyau[ECRITURE]);
      /* dup2 va brancher le côté lecture du tuyau */
      /* comme entrée standard du processus courant */
      if (dup2(tuyau[LECTURE], STDIN_FILENO) == -1) {
         perror("Erreur dans dup2()");
      }
      /* on ferme le descripteur qui reste */
      close(tuyau[LECTURE]);
      /* wc lit l'entrée standard, et les données */
      /* qu'il reçoit proviennent du tuyau */
      if (execlp("wc", "wc", "-l", NULL) == -1) {
         perror("Erreur dans execlp()");
         exit(EXIT_FAILURE);
      }
   }
   exit(EXIT_SUCCESS);
}
```

La séquence utilisant dup2() est équivalente à celle-ci en utilisant dup().

17.2.4.6. Synchronisation de deux processus au moyen d'un tuyau

Un effet de bord intéressant des tuyaux est la possibilité de synchroniser deux processus. En effet, un processus tentant de lire un tuyau dans lequel il n'y a rien est suspendu jusqu'à ce que des données soient disponibles. Donc, si le processus qui écrit dans le tuyau ne le fait pas très rapidement que celui qui lit, il est possible de synchroniser le processus lecteur sur le processus écrivain.
L'exemple suivant met en oeuvre une utilisation possible de cette synchronisation :

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LECTURE  0
#define ECRITURE 1

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

```
   int tuyau[2], i;
   char car;

   if (pipe(tuyau) == -1) {
      perror("Erreur dans pipe()");
      exit(EXIT_FAILURE);
   }

   switch (fork()) {
    case -1 :   /* erreur */
      perror("Erreur dans fork()");
      exit(EXIT_FAILURE);
    case 0 :   /* processus fils, lecteur */
      close(tuyau[ECRITURE]);
      /* on lit les caractères un à un */
      while (read(tuyau[LECTURE], &car, 1) != 0 ) {
         putchar(car);
         /* affichage immédiat du caractère lu */
         fflush(stdout);
      }
      close(tuyau[LECTURE]);
      putchar('\n');
      exit(EXIT_SUCCESS);
    default :   /* processus père, écrivain */
      close(tuyau[LECTURE]);
      for (i = 0; i < 10; i++) {
         /* on obtient le caractère représentant i */
         /* en prenant le i-ème caractère à partir de '0' */
         car = '0'+i;
         /* on écrit ce seul caractère */
         write(tuyau[ECRITURE], &car, 1);
         sleep(1); /* et on attend 1 sec */
      }
      close(tuyau[ECRITURE]);
      exit(EXIT_SUCCESS);
   }
}
```

17.2.4.7. Le signal SIGPIPE

Lorsqu'un processus écrit dans un tuyau qui n'a plus de lecteurs, parce que les processus qui lisaient le tuyau sont terminés ou ont fermé le côté lecture du tuyau, ce processus reçoit le signal SIGPIPE. Comme le comportement par défaut de ce signal est de terminer le processus, il peut être intéressant de le gérer afin, par exemple, d'avertir l'utilisateur puis de quitter proprement le programme.

17.2.4.8. Autres moyens de communication entre processus

Les tuyaux souffrent de deux limitations :

- ils ne permettent qu'une communication unidirectionnelle ;
- les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un ancêtre commun.

Ainsi, d'autres moyens de communication entre processus ont été développés pour pallier ces inconvénients :

- **Les tuyaux nommés**
  (FIFOs) sont des fichiers spéciaux qui, une fois ouverts, se comportent comme des

tuyaux : les données sont envoyées directement sans être stockées sur disque. Comme ce sont des fichiers, ils peuvent permettre à des processus quelconques, pas nécessairement issus d'un même ancêtre, de communiquer.

- **Les *sockets***
  permettent une communication bidirectionnelle entre divers processus, fonctionnant sur la même machine ou sur des machines reliées par un réseau.

Les tuyaux nommés s'utilisent facilement et sont abordés dans le paragraphe suivant. Quant aux *sockets*, elles sont étudiées plus loin.

17.2.4.9. Les tuyaux nommés

Comme on vient de le voir, l'inconvénient principal des tuyaux est de ne fonctionner qu'avec des processus issus d'un ancêtre commun. Pourtant, les mécanismes de communication mis en jeu dans le noyau sont généraux et, en fait, seul l'héritage des descripteurs de fichiers après un `fork()` impose cette restriction.
Pour s'en affranchir, il faut que les processus désirant communiquer puissent désigner le tuyau qu'ils souhaitent utiliser. Ceci se fait grâce au système de fichiers.
Un tuyau nommé est donc un fichier :

```
mkfifo fifo
ls -l fifo
prw-r--r-- 1 frosch74 frosch74 0 2006-10-19 14:35 fifo
```

Il s'agit cependant d'un fichier d'un type particulier, comme le montre le `p` dans l'affichage de **`ls`**.
Une fois créé, un tuyau nommé s'utilise très facilement :

```
echo coucou > fifo &
[1] 11090
cat fifo
[1]  + done       echo coucou > fifo
coucou
```

Si l'on fait abstraction des affichages parasites du *shell*, le tuyau nommé se comporte tout à fait comme on s'y attend. Bien que la façon de l'utiliser puisse se révéler trompeuse, un tuyau nommé transfère bien ses données d'un processus à l'autre en mémoire, sans les stocker sur disque. La seule intervention du système de fichiers consiste à permettre l'accès au tuyau par l'intermédiaire de son nom.
Il faut noter que :

- Un processus tentant d'écrire dans un tuyau nommé ne possédant pas de lecteurs sera suspendu jusqu'à ce qu'un processus ouvre le tuyau nommé en lecture. C'est pourquoi, dans l'exemple, **`echo`** a été lancé en tâche de fond, afin de pouvoir récupérer la main dans le *shell* et d'utiliser **`cat`**. On peut étudier ce comportement en travaillant dans deux fenêtres différentes.
- De même, un processus tentant de lire dans un tuyau nommé ne possédant pas d'écrivains se verra suspendu jusqu'à ce qu'un processus ouvre le tuyau nommé en écriture. On peut étudier ce comportement en reprenant l'exemple mais en lançant d'abord **`cat`** puis **`echo`**.

En particulier, ceci signifie qu'un processus ne peut pas utiliser un tuyau nommé pour stocker des données afin de les mettre à la disposition d'un autre processus une fois le premier processus terminé. On retrouve le même phénomène de synchronisation qu'avec les tuyaux classiques.
En C, un tuyau nommé se crée au moyen de la fonction `mkfifo()` :
Il doit ensuite être ouvert grâce à `open()` ou à `fopen()`, puis s'utilise au moyen des fonctions

d'entrées / sorties classiques.

17.2.5. Les sockets

17.2.5.1. Introduction

Les *sockets* représentent la forme la plus complète de communication entre processus. En effet, elles permettent à plusieurs processus quelconques d'échanger des données, sur la même machine ou sur des machines différentes. Dans ce cas, la communication s'effectue grâce à un protocole réseau défini lors de l'ouverture de la *socket* (IP, ISO,...).
Ce document ne couvre que les *sockets* réseau utilisant le protocole IP (*Internet Protocol*), celles-ci étant de loin les plus utilisées.

17.2.5.2. Les RFC

Il sera souvent fait référence dans la suite de ce chapitre à des documents appelés « *Request For Comments* » (RFC). Les RFC sont les documents textuels décrivant les standards de l'Internet, en particulier, les protocoles de communication de l'Internet. Les RFC sont numérotées dans l'ordre croissant au fur et à mesure de leur parution. Elles sont consultables sur de nombreux sites dont :

http://abcdrfc.free.fr/ (site en français)

17.2.5.3. Technologies de communication réseau

Il existe rarement une connexion directe par câble entre deux machines. Pour pouvoir circuler d'une machine à l'autre, les informations doivent généralement traverser un certain nombre d'équipements intermédiaires. La façon de gérer le comportement de ces équipements a mené à la mise au point de deux technologies principales de communication réseau :

a) La commutation de circuits

nécessite, au début de chaque connexion, l'établissement d'un circuit depuis la machine source vers la machine destination, à travers un nombre d'équipements intermédiaires fixé à l'ouverture de la connexion. Cette technologie, utilisée pour les communications téléphoniques et par le système ATM (*Asynchronounous Transfert Mode*) présente un inconvénient majeur. En effet, une fois le circuit établi, il n'est pas possible de le modifier pour la communication en cours, ce qui pose deux problèmes :

- si l'un des équipements intermédiaires tombe en panne ou si le câble entre deux équipements successifs est endommagé, la communication est irrémédiablement coupée, il n'y a pas de moyen de la rétablir et il faut en initier une nouvelle ;
- si le circuit utilisé est surchargé, il n'est pas possible d'en changer pour en utiliser un autre.

En revanche, la commutation de circuits présente l'avantage de pouvoir faire très facilement de la réservation de bande passante puisqu'il suffit pour cela de réserver les circuits adéquats.

b) La commutation de paquets

repose sur le découpage des informations en morceaux, dits paquets, dont chacun est expédié sur le réseau indépendamment des autres. Chaque équipement intermédiaire, appelé **routeur**, est chargé d'aiguiller les paquets dans la bonne direction, vers le routeur suivant. Ce principe est appelé routage. Il existe deux types de routage :

- **Le routage statique,**
  qui stipule que les paquets envoyés vers tel réseau doivent passer par tel routeur. Le routage statique est enregistré dans la configuration du routeur et ne peut pas changer suivant l'état du réseau.
- **Le routage dynamique,**
  qui repose sur un dialogue entre routeurs et qui leur permet de modifier le routage suivant l'état du réseau. Le dialogue entre routeurs repose sur un protocole de routage (tel que OSPF (RFC-2328), RIP (RFC-1058) ou BGP (RFC-1771) dans le cas des réseaux IP).
  - Le routage dynamique permet à la commutation de paquets de pallier les inconvénients de la commutation de circuits :
    - ✗ si un routeur tombe en panne ou si un câble est endommagé, le routage est modifié dynamiquement pour que les paquets empruntent un autre chemin qui aboutira à la bonne destination (s'il existe un tel chemin, bien entendu) ;
    - ✗ si un chemin est surchargé, le routage peut être modifié pour acheminer les paquets par un chemin moins embouteillé, s'il existe.
    - ✗ En revanche, contrairement à la commutation de circuits, il est assez difficile de faire de la réservation de bande passante avec la commutation de paquets. En contrepartie, celle-ci optimise l'utilisation des lignes en évitant la réservation de circuits qui ne seront que peu utilisés.

17.2.5.4. Le protocole IP

IP (*Internet Protocol*, RFC-791 pour IPv4, RFC-2460 pour IPv6) est un protocole reposant sur la technologie de commutation de paquets.
Chaque machine reliée à un réseau IP se voit attribuer une adresse, dite « adresse IP », unique sur l'Internet, qui n'est rien d'autre qu'un entier sur 32 bits. Afin de faciliter leur lecture et leur mémorisation, il est de coutume d'écrire les adresses IP sous la forme de quatre octets séparés par des points, par exemple : `172.16.112.115`.
Les noms de machines sont cependant plus faciles à retenir et à utiliser mais, pour son fonctionnement, IP ne reconnaît que les adresses numériques. Un système a donc été mis au point pour convertir les noms en adresses IP et vice versa : le DNS (*Domain Name System*, RFC-1034 et RFC-1035).
Chaque paquet IP contient deux parties :

- **L'en-tête,**
  qui contient diverses informations nécessaires à l'acheminement du paquet. Parmi celles-ci, on peut noter :
  - l'adresse IP de la machine source ;
  - l'adresse IP de la machine destination ;
  - la taille du paquet.
- **Le contenu du paquet**
  proprement dit, qui contient les informations à transmettre.

IP ne garantit pas que les paquets émis soient reçus par leur destinataire et n'effectue aucun contrôle d'intégrité sur les paquets reçus. C'est pourquoi deux protocoles plus évolués ont été bâtis au-dessus d'IP : UDP et TCP. Les protocoles applicatifs couramment utilisés, comme SMTP ou HTTP, sont bâtis au-dessus de l'un de ces protocoles (bien souvent TCP). Ce système en couches, où un

nouveau protocole offrant des fonctionnalités plus évoluées est bâti au-dessus d'un protocole plus simple, est certainement l'une des clés du succès d'IP.

17.2.5.5. Le protocole UDP

UDP (*User Datagram Protocol*, RFC-768) est un protocole très simple, qui ajoute deux fonctionnalités importantes au-dessus d'IP :

- l'utilisation de numéros de ports par l'émetteur et le destinataire ;
- un contrôle d'intégrité sur les paquets reçus.

Les numéros de ports permettent à la couche IP des systèmes d'exploitation des machines source et destination de faire la distinction entre les processus utilisant les communication réseau.
En effet, imaginons une machine sur laquelle plusieurs processus sont susceptibles de recevoir des paquets UDP. Cette machine est identifiée par son adresse IP, elle ne recevra donc que les paquets qui lui sont destinés, puisque cette adresse IP figure dans le champ destination de l'en-tête des paquets. En revanche, comment le système d'exploitation peut-il faire pour déterminer à quel processus chaque paquet est destiné ? Si l'on se limite à l'en-tête IP, c'est impossible. Il faut donc ajouter, dans un en-tête spécifique au paquet UDP, des informations permettant d'aiguiller les informations vers le bon processus. C'est le rôle du numéro de port destination.
Le numéro de port source permet au processus destinataire des paquets d'identifier l'expéditeur ou au système d'exploitation de la machine source de prévenir le bon processus en cas d'erreur lors de l'acheminement du paquet.
Un paquet UDP est donc un paquet IP dont le contenu est composé de deux parties :

- **l'en-tête UDP,**
  qui contient les numéros de ports source et destination, ainsi que le code de contrôle d'intégrité ;
- **le contenu du paquet UDP**
  à proprement parler.

Au total, un paquet UDP est donc composé de trois parties :

1. l'en-tête IP ;
2. l'en-tête UDP ;
3. le contenu du paquet.

UDP ne permet cependant pas de gérer la retransmission des paquets en cas de perte ni d'adapter son débit à la bande passante disponible. Enfin, le découpage des informations à transmettre en paquets est toujours à la charge du processus source qui doit être capable d'adapter la taille des paquets à la capacité du support physique (par exemple, la taille maximale (MTU, *Maximum Transmission Unit*) d'un paquet transmissible sur un segment Ethernet est de 1500 octets, en-têtes IP et UDP compris, voir RFC-894).

17.2.5.6. Le protocole TCP

TCP (*Transmission Control Protocol*, RFC-793) est un protocole établissant un canal bidirectionnel entre deux processus.
De même qu'UDP, TCP ajoute un en-tête supplémentaire entre l'en-tête IP et les données à transmettre, en-tête contenant les numéros de port source et destination, un code de contrôle d'intégrité, exactement comme UDP, ainsi que diverses informations que nous ne détaillerons pas ici.
Un paquet TCP est donc un paquet IP dont le contenu est composé de deux parties :

- **l'en-tête TCP,**
  qui contient, entre autres, les numéros de ports source et destination, ainsi que le code de contrôle d'intégrité ;
- **le contenu du paquet TCP**
  à proprement parler.

Au total, un paquet TCP est donc composé de trois parties :
1. l'en-tête IP ;
2. l'en-tête TCP ;
3. le contenu du paquet.

Contrairement à UDP, qui impose aux processus voulant communiquer de gérer eux-même le découpage des données en paquets et leur retransmission en cas de perte, TCP apparaît à chaque processus comme un fichier ouvert en lecture et en écriture, ce qui lui confère une grande souplesse. À cet effet, TCP gère lui-même le découpage des données en paquets, le recollage des paquets dans le bon ordre et la retransmission des paquets perdus si besoin est. TCP s'adapte également à la bande passante disponible en ralentissant l'envoi des paquets si les pertes sont trop importantes (RFC-1323) puis en ré-augmentant le rythme au fur et à mesure, tant que les pertes sont limitées.

17.2.5.7. Interface de programmation

L'interface de programmation des *sockets* peut paraître compliquée au premier abord, mais elle est en fait relativement simple parce que faite d'une succession d'étapes simples.

a) La fonction `socket()`

La fonction `socket()` permet de créer une *socket* de type quelconque :
- Le paramètre `domaine` précise la famille de protocoles à utiliser pour les communications. Il y a deux familles principales :
  PF_UNIX
  pour les communications locales n'utilisant pas le réseau ;
  PF_INET
  pour les communications réseaux utilisant le protocole IP.

  Nous n'étudierons par la suite que la famille `PF_INET`.
- Le paramètre `type` précise le type de la *socket* à créer :
  - `SOCK_STREAM`
    pour une communication bidirectionnelle sûre. Dans ce cas, la *socket* utilisera le protocole TCP pour communiquer.
  - `SOCK_DGRAM`
    pour une communication sous forme de datagrammes. Dans ce cas, la *socket* utilise le protocole UDP pour communiquer.
  - `SOCK_RAW`
    pour pouvoir créer soi-même ses propres paquets IP ou les recevoir sans traitement de la part de la couche réseau du système d'exploitation, ce qui est indispensable dans certains cas très précis.

- Le paramètre `protocole` doit être égal à zéro.

La valeur renvoyée par la fonction `socket()` est -1 en cas d'erreur, sinon c'est un descripteur de fichier qu'on peut par la suite utiliser avec les fonctions `read()` et `write()`, transformer en

pointeur de type `FILE *` avec `fdopen()`, etc..
Nous n'étudierons que les *sockets* de type `SOCK_STREAM` par la suite, celles-ci étant les plus utilisées.

b) Exemple de client TCP

La programmation d'un client TCP est fort simple et se déroule en cinq étapes :

1. création de la *socket* ;
2. récupération de l'adresse IP du serveur grâce au DNS ;
3. connexion au serveur ;
4. dialogue avec le serveur ;
5. fermeture de la connexion.

i) Création de la socket

Elle se fait simplement à l'aide de la fonction `socket()` :

ii) Interrogation du DNS

Le protocole IP ne connaît que les adresses IP. Il est donc nécessaire d'interroger le DNS pour récupérer l'adresse IP du serveur à partir de son nom. Il peut aussi être utile de récupérer le nom du serveur si le client est lancé avec une adresse IP comme argument. Pour cela, on utilise deux fonctions : `gethostbyname()` et `gethostbyaddr()`.

iii) La fonction `gethostbyname()`

La fonction `gethostbyname()` permet d'interroger le DNS de façon à obtenir l'adresse IP d'une machine à partir de son nom :
Cette fonction prend en argument une chaîne de caractères (pointeur de type `char *`) contenant le nom de la machine et renvoie un pointeur vers une structure de type `hostent` (défini dans le fichier d'en-tête `<netdb.h>`). Cette structure contient trois membres qui nous intéressent :

- `h_name` : est le nom canonique de la machine (pointeur de type `char *`) ;
- `h_addr_list` : est la liste des adresse IP de la machine (pointeur de type `char **`).
- `h_length` : est la taille de chacune des adresses stockées dans `h_addr_list` (ceci sert à en permettre la recopie sans faire de suppositions sur la taille des adresses).

iv) La fonction `gethostbyaddr()`

La fonction `gethostbyaddr()` permet d'interroger le DNS de façon à obtenir le nom canonique d'une machine à partir de son adresse IP :
La fonction `inet_addr()` permet de transformer une chaîne de caractères représentant une adresse IP (comme 192.168.1.1) en entier long.
La fonction `gethostbyaddr()` prend en argument :

1. un pointeur de type `char *` vers l'adresse IP sous forme numérique, d'où la nécessité d'utiliser `inet_addr()` ;
2. la taille de cette adresse ;
3. le type d'adresse utilisée, `AF_INET` pour les adresses IP.

Elle renvoie un pointeur vers une structure de type `hostent`, tout comme `gethostbyname()`.

v) Connexion au serveur TCP

La connexion au serveur TCP se fait au moyen de la fonction `connect()` :

Cette fonction prend en argument :

1. le descripteur de fichier de la *socket* préalablement créée ;
2. un pointeur vers une structure de type `sockaddr` ;
3. la taille de cette structure.

Dans le cas des communications IP, le pointeur vers la structure de type `sockaddr` est un fait un pointeur vers une structure de type `sockaddr_in`, définie dans le fichier d'en-tête `<netinet/in.h>` (d'où la conversion de pointeur).
Cette structure contient trois membres utiles :

- `sin_family` : est la famille de protocoles utilisée, `AF_INET` pour IP ;
- `sin_port` : est le port TCP du serveur sur lequel se connecter ;
- `sin_addr.s_addr` : est l'adresse IP du serveur.

Le port TCP doit être fourni sous un format spécial. En effet les microprocesseurs stockent les octets de poids croissant des entiers courts (16 bits) et des entiers longs (32 bits) de gauche à droite (microprocesseurs dits *BIG ENDIAN* tels que le Motorola 68000) ou de droite à gauche (microprocesseurs dits *LITTLE ENDIAN* tels que les Intel). Pour les communications IP, le format *BIG ENDIAN* a été retenu. Il faut donc éventuellement convertir le numéro de port du format natif de la machine vers le format réseau au moyen de la fonction `htons()` (*Host To Network Short*) pour les machines à processeur *LITTLE ENDIAN*. Dans le doute, on utilise cette fonction sur tous les types de machines (elle est également définie sur les machines à base de processeur *BIG ENDIAN*, mais elle ne fait rien).
L'adresse IP n'a pas besoin d'être convertie, puisqu'elle est directement renvoyée par le DNS au format réseau.

La fonction `memset()` permet d'initialiser globalement la structure à zéro afin de supprimer toute valeur parasite.

vi) Dialogue avec le serveur

Le dialogue entre client et serveur s'effectue simplement au moyen des fonctions `read()` et `write()`.
Cependant, notre client devra lire au clavier et sur la *socket* sans savoir à l'avance sur quel descripteur les informations seront disponibles. Il faut donc écouter deux descripteurs à la fois et lire les données sur un descripteur quand elles sont disponibles.
Pour cela, on utilise la fonction `select()`, qui est extrêmement pratique et qui peut s'utiliser avec tout type de descripteurs de fichier (*socket*, tuyaux, entrée standard...).
On commence par indiquer, au moyen d'une variable de type `fd_set`, quels descripteurs nous intéressent :

La fonction `FD_ZERO()` met la variable `rset` à zéro, puis on indique son intérêt dans l'écoute de la *socket* et de l'entrée standard au moyen des fonctions `FD_SET()`.
On appelle ensuite la fonction `select()` qui attendra que des données soient disponibles sur l'un de ces descripteurs :
La fonction `select()` prend cinq arguments :

1. Le plus grand numéro de descripteur à surveiller, plus un. Dans notre exemple, c'est `client_socket + 1`, puisque l'entrée standard a 0 comme numéro de descripteur.
2. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels on veut lire.
3. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur

lesquels on veut écrire. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.

4. Un pointeur vers une variable de type `fd_set` représentant la liste des descripteurs sur lesquels peuvent arriver des conditions exceptionnelles. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.
5. Un pointeur vers une structure de type `timeval` représentant une durée après laquelle la fonction `select()` doit rendre la main si aucun descripteur n'est disponible. Dans ce cas, la valeur retournée par `select()` est 0. N'étant pas intéressés par cette possibilité, nous avons utilisé un pointeur nul.

La fonction `select()` renvoie -1 en cas d'erreur, 0 au bout du temps spécifié par le cinquième paramètre et le nombre de descripteurs prêts sinon.
La fonction `FD_ISSET` permet de déterminer si un descripteur est prêt :

vii) Fermeture de la connexion

La *socket* se ferme simplement au moyen de la fonction `close()`. On peut aussi utiliser la fonction `shutdown()`, qui permet une fermeture sélective de la *socket* (soit en lecture, soit en écriture).

viii) Client TCP complet

```
#include <sys/types.h>

#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

/*-------------------------------------------- */
int socket_client ( char *serveur , unsigned short port ) {
   int                  client_socket ;
   struct hostent       *hostent ;
   struct sockaddr_in   serveur_sockaddr_in ;

   client_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
   if ( client_socket == -1 ) {
      perror ( "socket" ) ;
      exit ( EXIT_FAILURE ) ;
   }

   if ( inet_addr( serveur ) == INADDR_NONE )   /* nom */ {
      hostent = gethostbyname ( serveur ) ;
      if ( hostent == NULL ) {
         perror ( "gethostbyname" ) ;
         exit ( EXIT_FAILURE ) ;
      }
   }
   else   /* adresse IP */ {
      unsigned long addr = inet_addr ( serveur ) ;
      hostent = gethostbyaddr((char *)&addr, sizeof(addr),
                              AF_INET ) ;
      if ( hostent == NULL ) {
         perror ( "gethostbyaddr" ) ;
```

```
         exit ( EXIT_FAILURE ) ;
      }
   }

   memset(&serveur_sockaddr_in, 0,
          sizeof(serveur_sockaddr_in));
   serveur_sockaddr_in.sin_family = AF_INET ;
   serveur_sockaddr_in.sin_port = htons(port) ;
   memcpy ( &serveur_sockaddr_in.sin_addr.s_addr ,
            hostent->h_addr_list[0] ,
            hostent->h_length) ;
   printf ( ">>> Connexion vers le port %d de %s [%s]\n" ,
            port , hostent->h_name ,
            inet_ntoa ( serveur_sockaddr_in.sin_addr ) ) ;

   if ( connect(client_socket,
                (struct sockaddr *)&serveur_sockaddr_in,
                sizeof(serveur_sockaddr_in)) == -1 ) {
      perror ( "connect" ) ;
      exit ( EXIT_FAILURE ) ;
   }
   return client_socket ;
}

/*--------------------------------------------- */

int main (int argc , char **argv)
{
   char             *serveur ;
   unsigned short   port ;
   int              client_socket ;

   if ( argc != 3 ) {
     fprintf(stderr, "usage: %s serveur port\n",
             argv[0]);
     exit ( EXIT_FAILURE ) ;
   }
   serveur = argv[1] ;
   port = atoi ( argv[2] ) ;
   client_socket = socket_client ( serveur , port ) ;

   for ( ; ; )  {
      fd_set rset ;

      FD_ZERO ( &rset ) ;
      FD_SET  ( client_socket , &rset ) ;
      FD_SET  ( STDIN_FILENO  , &rset ) ;

      if (select(client_socket+1, &rset, NULL, NULL, NULL)
          == -1 ) {
         perror ( "select" ) ;
         exit ( EXIT_FAILURE ) ;
      }

      if (FD_ISSET(client_socket, &rset)) {
        int             octets ;
        unsigned char   tampon[BUFSIZ] ;

        octets = read(client_socket,tampon,sizeof(tampon));
```

```
        if ( octets == 0 ) {
            close ( client_socket ) ;
            exit ( EXIT_SUCCESS ) ;
        }
        write ( STDOUT_FILENO , tampon , octets ) ;
      }

      if ( FD_ISSET ( STDIN_FILENO , &rset ) ) {
         int             octets ;
         unsigned char   tampon[BUFSIZ] ;

         octets = read(STDIN_FILENO,tampon,sizeof(tampon));
         if ( octets == 0 ) {
            close ( client_socket ) ;
            exit ( EXIT_SUCCESS ) ;
         }

         write ( client_socket , tampon , octets ) ;
      }
   }

   exit ( EXIT_SUCCESS ) ;
}
```

c) Exemple de serveur TCP

La programmation d'un serveur TCP est somme toute fort semblable à celle d'un client TCP :

1. création de la *socket* ;
2. choix du port à écouter ;
3. attente d'une connexion ;
4. dialogue avec le client ;
5. fermeture de la connexion.

i) Création de la *socket*

La création d'une *socket* serveur se déroule strictement de la même façon que la création d'une *socket* client :
On appelle cependant une fonction supplémentaire, `setsockopt()`. En effet, après l'arrêt d'un serveur TCP, le système d'exploitation continue pendant un certain temps à écouter sur le port TCP que ce dernier utilisait afin de prévenir d'éventuels client dont les paquets se seraient égarés de l'arrêt du serveur, ce qui rend la création d'un nouveau serveur sur le même port impossible pendant ce temps-là. Pour forcer le système d'exploitation à réutiliser le port, on utilise l'option `SO_REUSEADDR` de la fonction `setsockopt()` :

ii) Choix du port à écouter

Le choix du port sur lequel écouter se fait au moyen de la fonction `bind()` :

Cette fonction prend en argument :

1. le descripteur de fichier de la *socket* préalablement créée ;
2. un pointeur vers une structure de type `sockaddr` ;
3. la taille de cette structure.

La structure de type `sockaddr_in` s'initialise de la même façon que pour le client TCP (n'oubliez pas de convertir le numéro de port au format réseau au moyen de la fonction `htons()`). La seule

différence est l'utilisation du symbole INADDR_ANY à la place de l'adresse IP. En effet, un serveur TCP est sensé écouter sur toutes les adresses IP dont dispose la machine, ce que permet INADDR_ANY.
La fonction listen() permet d'informer le système d'exploitation de la présence du nouveau serveur :

Cette fonction prend en argument :
1. le descripteur de fichier de la *socket* préalablement créée ;
2. le nombre maximum de connexions pouvant être en attente (on utilise généralement le symbole SOMAXCONN qui représente le nombre maximum de connexions en attente autorisé par le système d'exploitation).

iii) Attente d'une connexion

La fonction accept() permet d'attendre qu'un client se connecte au serveur :

Cette fonction prend trois arguments :
1. le numéro de descripteur de la *socket* ;
2. un pointeur vers une structure de type sockaddr, structure qui sera remplie avec les paramètres du client qui s'est connecté;
3. un pointeur vers un entier, qui sera rempli avec la taille de la structure ci-dessus.

En cas de succès de la connexion, la fonction accept() renvoie un nouveau descripteur de fichier représentant la connexion avec le client. Le descripteur initial peut encore servir à de nouveaux clients pour se connecter au serveur, c'est pourquoi les serveurs appellent ensuite généralement la fonction fork() afin de créer un nouveau processus chargé du dialogue avec le client tandis que le processus initial continuera à attendre des connexions.
Après le fork(), chaque processus ferme le descripteur qui lui est inutile (exactement comme avec les tuyaux).

iv) Dialogue avec le client

Le dialogue avec le client s'effectue de la même façon que dans le cas du client.

v) Fermeture de la connexion

La fermeture de la connexion s'effectue de la même façon que dans le cas du client.

vi) Serveur TCP complet

```
#include <sys/types.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

/* ------------------------------------------- */
int socket_serveur ( unsigned short port ) {
   int                  serveur_socket , option = 1 ;
   struct sockaddr_in   serveur_sockaddr_in ;

   serveur_socket = socket ( PF_INET , SOCK_STREAM , 0 ) ;
```

```
   if ( serveur_socket == -1 ) {
      perror ( "socket" ) ;
      exit ( EXIT_FAILURE ) ;
   }

   if ( setsockopt(serveur_socket, SOL_SOCKET, SO_REUSEADDR,
                   &option, sizeof(option)) == -1 ) {
      perror ( "setsockopt" ) ;
      exit ( EXIT_FAILURE ) ;
   }

   memset(&serveur_sockaddr_in, 0,
          sizeof (serveur_sockaddr_in));
   serveur_sockaddr_in.sin_family      = AF_INET ;
   serveur_sockaddr_in.sin_port        = htons ( port ) ;
   serveur_sockaddr_in.sin_addr.s_addr = INADDR_ANY ;

   if ( bind ( serveur_socket ,
               (struct sockaddr *) &serveur_sockaddr_in,
               sizeof(serveur_sockaddr_in) ) == -1 ) {
      perror ( "bind" ) ;
      exit ( EXIT_FAILURE ) ;
   }

   if ( listen ( serveur_socket , SOMAXCONN ) == -1 ) {
      perror ( "listen" ) ;
      exit ( EXIT_FAILURE ) ;
   }

   return serveur_socket ;
}

/* ------------------------------------------------------ */
void serveur ( int client_socket ) {
   for ( ; ; ) {
      fd_set rset ;
      FD_ZERO ( &rset ) ;
      FD_SET  ( client_socket , &rset ) ;
      FD_SET  ( STDIN_FILENO  , &rset ) ;

      if ( select(client_socket+1, &rset,
           NULL, NULL, NULL ) == -1 ) {
         perror ( "select" ) ;
         exit ( EXIT_FAILURE ) ;
      }

      if ( FD_ISSET ( client_socket , &rset ) )  {
        int             octets ;
        unsigned char   tampon[BUFSIZ] ;

        octets = read(client_socket,tampon,sizeof(tampon));
        if ( octets == 0 ) {
          close ( client_socket ) ;
          printf ( ">>> Déconnexion\n" ) ;
          exit ( EXIT_SUCCESS ) ;
        }

        write ( STDOUT_FILENO , tampon , octets ) ;
      }
```

```
      if ( FD_ISSET ( STDIN_FILENO , &rset ) ) {
         int             octets ;
         unsigned char   tampon[BUFSIZ] ;

         octets = read(STDIN_FILENO,tampon,sizeof(tampon));

         if ( octets == 0 ) {
           close ( client_socket ) ;
           exit ( EXIT_SUCCESS ) ;
         }
         write ( client_socket , tampon , octets ) ;
      }
   }
}

/* ------------------------------------------------ */

int main ( int argc , char **argv ) {
   unsigned short   port ;
   int              serveur_socket ;

   if ( argc != 2 )  {
      fprintf ( stderr , "usage: %s port\n" , argv[0] ) ;
      exit ( EXIT_FAILURE ) ;
   }

   port = atoi ( argv[1] ) ;
   serveur_socket = socket_serveur ( port ) ;

   for ( ; ; )   {
      fd_set rset ;
      FD_ZERO ( &rset ) ;
      FD_SET  ( serveur_socket , &rset ) ;
      FD_SET  ( STDIN_FILENO   , &rset ) ;

      if (select(serveur_socket+1, &rset,
                 NULL, NULL, NULL) == -1 ) {
         perror ( "select" ) ;
         exit ( EXIT_FAILURE ) ;
      }

      if ( FD_ISSET ( serveur_socket , &rset ) ) {
         int client_socket ;
         struct sockaddr_in client_sockaddr_in ;
         socklen_t taille=sizeof (client_sockaddr_in);
         struct hostent *hostent ;

         client_socket = accept ( serveur_socket ,
                   (struct sockaddr *) &client_sockaddr_in ,
                   &taille ) ;
         if ( client_socket == -1 ) {
            perror ( "accept" ) ;
            exit ( EXIT_FAILURE ) ;
         }

         switch ( fork ( ) )  {
          case -1 :   /* erreur */
            perror ( "fork" ) ;
```

```
            exit ( EXIT_FAILURE ) ;
          case 0 :   /* processus fils */
            close ( serveur_socket ) ;
            hostent = gethostbyaddr(
               (char *)&client_sockaddr_in.sin_addr.s_addr,
               sizeof(client_sockaddr_in.sin_addr.s_addr),
               AF_INET) ;
            if ( hostent == NULL ) {
               perror ( "gethostbyaddr" ) ;
               exit ( EXIT_FAILURE ) ;
            }
            printf (">>> Connexion depuis %s [%s]\n",
               hostent->h_name,
               inet_ntoa ( client_sockaddr_in.sin_addr ) ) ;
            serveur ( client_socket ) ;
            exit ( EXIT_SUCCESS ) ;
         default :   /* processus père */
            close ( client_socket ) ;
          }
       }
   }

   exit ( EXIT_SUCCESS ) ;
}
```

17.3. Les interfaces parallèle IEEE 1284 et série RS-232

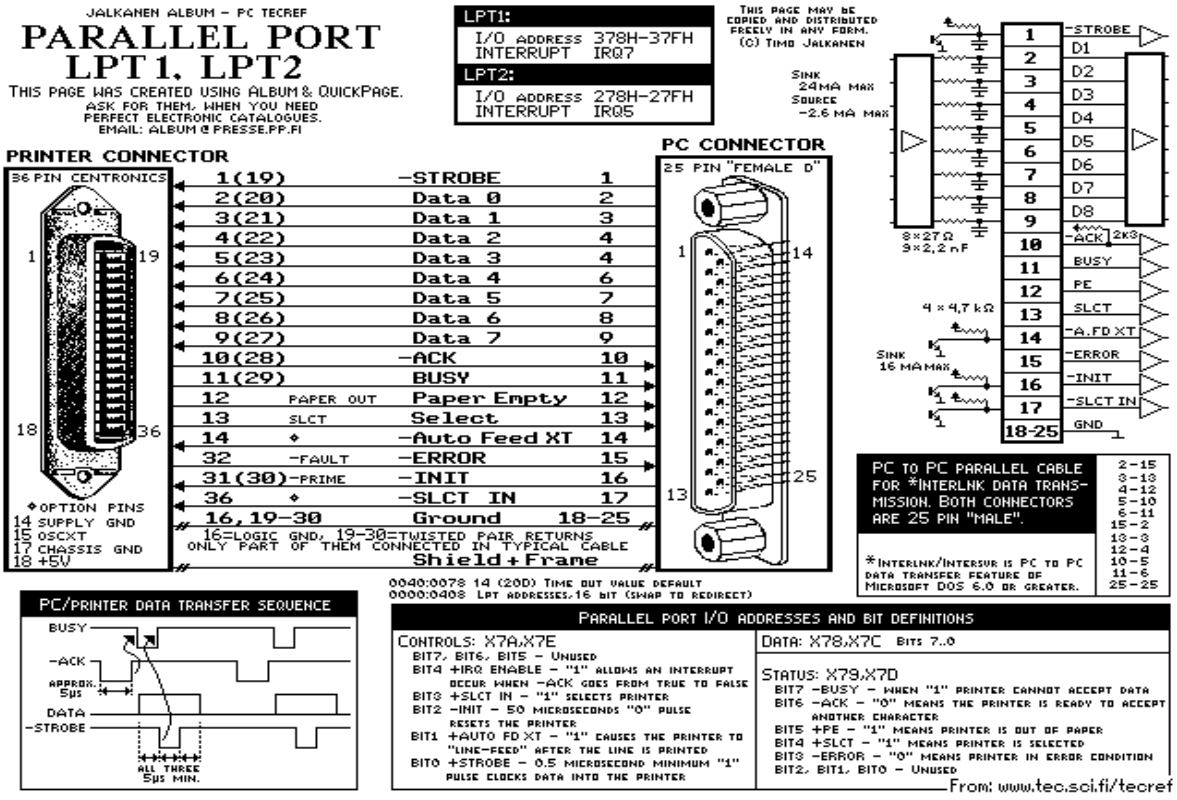
Des interfaces d'entrées/sorties séries et parallèles équipent tous les PC. L'interface série RS-232, de plus en plus supplantée par le port USB, permet l'échange d'informations à faible débit, avec des périphériques tels qu'un modem ou encore un autre PC, sur des distances inférieures à quelques dizaines de mètres. L'interface parallèle IEEE 1284 permet aussi un échange d'informations avec des périphériques comme une imprimante, mais avec un débit plus élevé et sur des distances beaucoup plus faibles.

17.3.1. L'interface parallèle IEEE 1284

Présente en 1981 sur l'IBM PC, l'interface parallèle était prévue pour piloter les toutes nouvelles imprimantes matricielles et compenser la lenteur du port série de l'époque. L'interface permettait la transmission de 8 bits de données en une seule fois de manière parallèle sur 8 lignes spécifiques. Ce n'est qu'en 1994 que le standard 1284 fut proposé afin de définir une interface parallèle bidirectionnelle devant faire face à la montée en puissance des ordinateurs et des configurations.

17.3.1.1. Structure de l'interface IEEE 1284

L'IEEE 1284 se compose de 17 lignes de signaux et de 8 lignes connectées à la masse pour former les 25 broches d'un connecteur DB25 femelle à l'arrière d'un PC. A chaque ligne correspond un bit dans un registre donné. Le premier registre est accessible à l'adresse d'entrée/sortie propre à chaque interface parallèle. Un certain nombre d'adresses sont standardisées comme l'adresse `0x378` pour la première interface parallèle du PC, mais des cartes additionnelles peuvent utiliser des adresses totalement différentes. Le premier registre `DATA` (adresse `0x378`) est celui contrôlant les lignes de données, le deuxième `STATUS`, 8 bits plus loin (adresse `0x379`) concerne les lignes d'état et enfin, le troisième `CONTROL` (adresse `0x37A`) les lignes de contrôle.



Le tableau ci-dessous résume les différents signaux du port parallèle :

Broche DB25	***Broche Centronics***	***Signal***	***Sens***	***Bit***	***Fonction***
1	1	/STROBE	S	/C0	Indique au périphérique que des données sont présentées sur les lignes D0..D7 et qu'il doit les prendre en compte. Ce signal est actif au niveau bas. Donc pour indiquer au périphérique que des données sont prêtes, on passe brièvement (~5µs) le signal à la masse.
2	2	DATA 0	S/E	D0	Ligne de donnée 0
3	3	DATA 1	S/E	D1	Ligne de donnée 1
4	4	DATA 2	S/E	D2	Ligne de donnée 2
5	5	DATA 3	S/E	D3	Ligne de donnée 3
6	6	DATA 4	S/E	D4	Ligne de donnée 4
7	7	DATA 5	S/E	D5	Ligne de donnée 5
8	8	DATA 6	S/E	D6	Ligne de donnée 6
9	9	DATA 7	S/E	D7	Ligne de donnée 7
10	10	/ACK	In	S6	*Acknowledge*. Le périphérique met cette ligne à l'état bas brièvement (~5µs) pour indiquer au PC qu'il a bien reçu le caractère.
11	11	BUSY	In	/S7	Active au niveau haut, cette ligne indique que le périphérique ne peut plus recevoir de données pour les traiter.
12	12	PAPER	In	S5	*Paper out*. Cette ligne indique que le périphérique n'a plus de papier.

13	13	ONLINE	In	S4	Indique que le périphérique est en ligne.
14	14	/AUTOFEED	In	/CI	Ce signal demande au périphérique de faire un saut de ligne. A l'époque des imprimantes matricielles, certaines configurations d'imprimante ne faisaient pas de retour à la ligne (CR/LF), mais simplement un retour de la tête d'impression en début de ligne (CR). Il fallait donc logiciellement avancer le papier d'une ligne (LF).
15	32	/ERROR	In	S3	Indique au PC que le périphérique est en erreur. Ce signal est actif au niveau bas et le registre au niveau haut.
16	31	/INIT	Out	C2	L'ordinateur peut demander une réinitialisation de l'imprimante.
17	36	/SELECT	Out	/C3	Ce signal permet une sélection de périphérique.
18-25	16, 17, 19-30, 33	Masse	N/A	N/A	Masse.

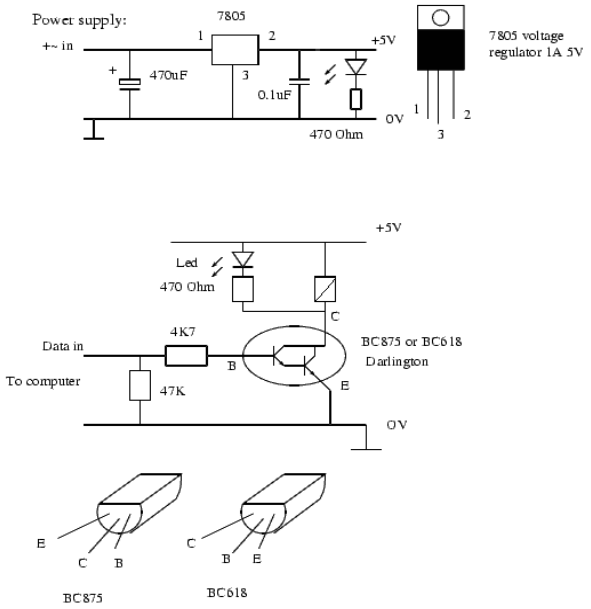
17.3.1.2. Les modes de l'interface IEEE 1284

L'interface IEEE 1284 peut fonctionner dans trois modes :

- **SPP** (*Standard Parallel Port)* : Les ligne de données de l'interface ne peuvent être utilisées qu'en mode unidirectionnel, i.e. de l'ordinateur vers le périphérique ;
- **EPP** (*Enhanced Parallel Port*) : Amélioration du mode SPP. Le bit 5 du registre de contrôle permet d'inverser le sens écriture/lecture pour les lignes de données. En le plaçant à 1, il est alors possible de lire des données en provenance du périphérique ;
- **ECP** (*Extended Capabilities Port*) : Amélioration du mode SPP. Utilise des registres de configuration complémentaires ainsi qu'une structure FIFO pour accélérer les transferts.

17.3.1.3. Exemple de connexion avec l'extérieur

Le schéma suivant, à reproduire pour toutes les broches de données D0 à D7, permet de piloter jusqu'à huit relais depuis le port parallèle.



Une alimentation stabilisée électroniquement est utilisée par ce pilotage. Elle assure une puissance constante et stable et protège le port parallèle. L'alimentation extérieure peut provenir de n'importe quelle source de courant continu entre 6 et 24V. Le 7805 est un régulateur commun. Il faudra juste faire attention aux deux condensateurs (470µF et 0.1µF) qui sont très proches du 7805 afin de l'empêcher d'osciller, ce qui pourrait le détruire.
L'interface peut être répétée jusqu'à huit fois suivant le nombre de relais à piloter. Nous utilisons un transistor Darlington NPN pour fournir des courants conséquents : En effet, le BC875 comme le BC618 peuvent commuter à environ 500mA. La résistance de 47K en entrée garantit qu'un circuit ouvert (par exemple, l'ordinateur qui n'est pas connecté) est toujours équivalent à un état « inactif ». La tension sur le port parallèle est supérieure à 4V si l'on se trouve dans un état « actif » et est inférieure à 1V pour un état « inactif ». Le transistor sert donc d'interrupteur et la LED indique l'état de l'interface (actif/inactif).

17.3.1.4. Programmation de l'interface IEEE 1284

a) Accès direct à l'interface parallèle

L'exemple de code suivant permet d'accéder de manière *barbare* à l'interface parallèle placée à l'adresse `0x378` :

```
#include <unistd.h>
#include <stdio.h>
#include <asm/io.h>
#define BASE_IEEE_1284 0x378

int main(int argc, char **argv)
{
```

```
    /* La fonction ioperm() permet de positionner les
       autorisations d'entrée/sortie sur les ports à
       commencer de 0x378 et ce sur 3 octets.
       ATTENTION, l'exécutable devra posséder les droits
       du root pour fonctionner.
    */
    if (ioperm(BASE_IEEE_1284, 3, 1)) {
        perror("ioperm");
        exit(1);
    }
    /* La valeur 10 ou 00001010 en binaire est écrite sur le
       port 0x378. Les broches de données D1 et D3 sont
       donc à 1 et les autres à 0 */
    outb(10, BASE_IEEE_1284);
    /* La valeur lue en 0x379 est celle du registre
       d'état STATUS. */
    printf("lu : 0x%X\n", inb(BASE_IEEE_1284 + 1));

    /* Fermeture du port par le programme */
    if (ioperm(BASE_IEEE_1284, 3, 0)) {
        perror("ioperm");
        exit(1);
    }
    return 0;
}
```

b) Accès via le pseudo-fichier `/dev/port`

Le second exemple que nous proposons est déjà plus élégant. En effet, sous GNU/Linux, il est possible d'utiliser le pseudo-fichier `/dev/port` qui réalise une copie mémoire (*mappage*) des adresses des ports d'E/S. Pour écrire dans le registre `DATA` de l'interface IEEE 1284, il suffit de positionner le pointeur dans le fichier, à l'adresse `0x378`, et d'écrire ou de lire un octet. Ce sont les permissions du fichiers `/dev/port` qui vont définir le bon fonctionnement du programme. Nous vous renvoyons au chapitre 13 pour les fonctions d'accès fichiers (`open()` pour l'ouverture, `lseek()` pour rechercher l'adresse `0x378` dans le fichier, et `read()` ou `write()` pour les lectures / écritures).

```
#include ...

int main(int argc, char * argv[])
{
  int port_parallele;
  if ((source=open("/dev/port",O_RDWR))<0) {
      fprintf(stderr,"erreur ouverture: %s(%d)\n",
              strerror(errno), errno);
      exit(3);
  }
  lseek(port_parallele, 0x378, 0);
  // ...
  write(port_parallele, 11, 1);
  // ...
  close(port_parallele);
  return 0;
}
```

c) *Accès via l'interface* `ppdev`

Enfin le dernier exemple utilise l'interface `ppdev` qui permet l'accès aux différents ports parallèles d'un ordinateur par l'intermédiaire des pseudo-fichiers `/dev/parportX` où `X` est le numéro du port parallèle considéré (`0` pour `LPT1`, etc.). Il s'agit de la solution optimale pour y accéder et les méthodes classiques d'accès aux fichiers sont utilisables de la même manière que ci-dessus. La fonction `ioctl()` est utilisée pour les accès.

```
#include <sys/ioctl.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/errno.h>
#include <linux/ppdev.h>
#include <linux/parport.h>

int main(int argc, char **argv)
{
    int errno, fd;

    /* Ouvre le fichier et demande un accès exclusif au port
       parallèle */
    if ((fd = open("/dev/parport0", O_RDWR)) < 0) {
        fprintf(stderr, "erreur ouverture\n");
        exit(2);
    }
    if (ioctl(fd, PPCLAIM) < 0) {
        fprintf(stderr,"Acces exclusif refusé: %s(%d)\n",
                strerror(errno), errno);
        exit(3);
    }

    // ...
    /* Ecriture de la donnée 0x0A sur le port */
    unsigned char valeur = 0x0A;
    if (ioctl(fd, PPWDATA, &valeur) < 0) {
        fprintf(stderr, "Impossible d'écrire: %s (%d)\n",
                strerror(errno), errno);
        exit(4);
    }

    // ...
    /* Lecture de données sur le port */
    /* Redonne la main au système et ferme le fichier */
    unsigned char valeur_lue;
    int bit5_registre_control = 1; // lecture activée

    if (ioctl(fd, PPDATADIR, &bit5_registre_control) < 0) {
        fprintf(stderr, "Bit 5 de CONTROL non positionné:
                %s (%d)\n", strerror(errno), errno);
        exit(5);
    }

    if (ioctl(fd, PPRDATA, &valeur_lue) < 0) {
        fprintf(stderr, "Impossible de lire:
                %s (%d)\n", strerror(errno), errno);
```

```
        exit(6);
    }
    // Faire quelque chose avec valeur_lue....


    /* Redonne la main au système et ferme le fichier */
    if (ioctl(fd, PPRELEASE) < 0) {
        fprintf(stderr,"Redonne la main: %s (%d)\n",
                strerror(errno), errno);
        exit(7);
    }
    if (close(fd) < 0) {
        fprintf(stderr, "erreur fermeture: %s (%d)\n",
                strerror(errno), errno);
        exit(5);
    }
    return (0);
}
```

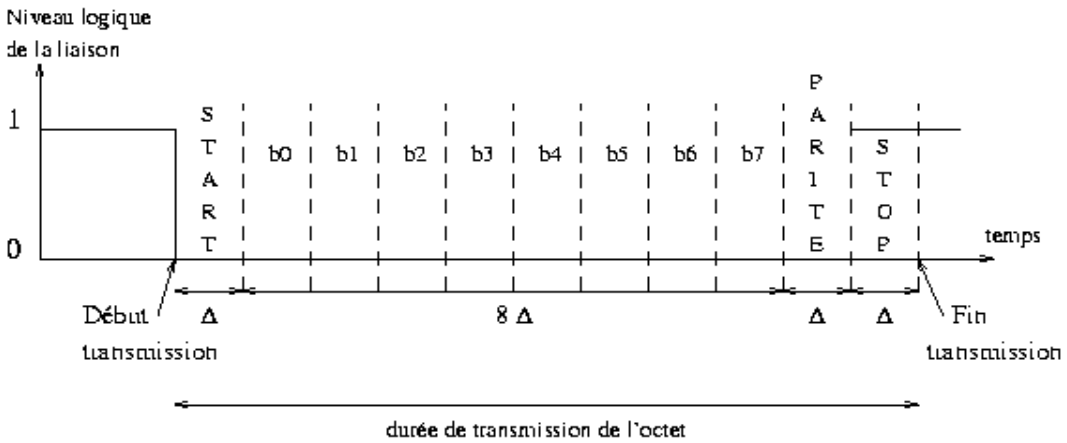
17.3.2. L'interface série RS-232

17.3.2.1. La nécessité d'une transmission série

Sur des distances supérieures à quelques mètres, il est difficile de mettre en oeuvre une transmission en parallèle : coût du câblage, mais surtout interférences électromagnétiques entre les fils provoquant des erreurs importantes. On utilise alors une liaison série, avec un seul fil portant l'information dans chaque sens.

Sur des distance supérieures à quelques dizaines de mètres, on utilisera des modems (MODulateur-DEModulateur) aux extrémités de la liaison et on passera par un support de transmission public (réseau téléphonique ou lignes spécialisées).

17.3.2.2. Principe de la transmission série asynchrone



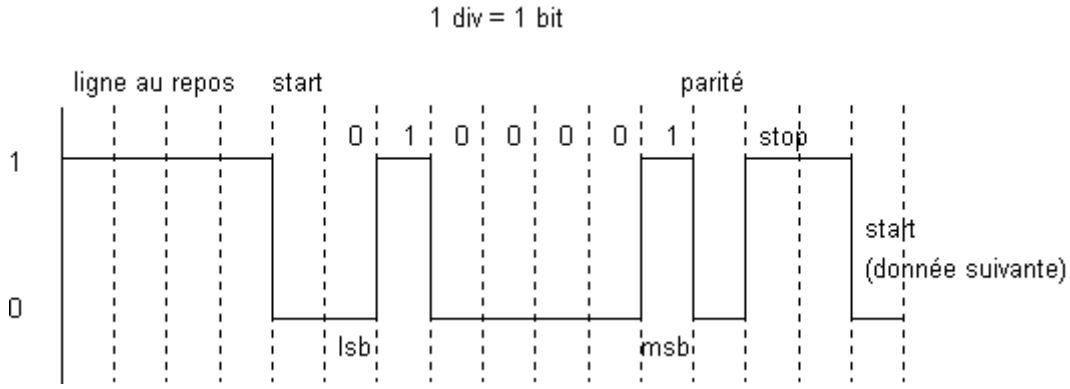
En l'absence de transmission, le niveau de la liaison est à `1` (niveau de repos). Les bits sont transmis les uns après les autres, en commençant par le bit de poids faible b_0. Le premier bit est précédé d'un

bit appelé `start` (niveau 0). Après le dernier bit de l'octet, on peut transmettre un bit de parité, puis un ou deux bits `stop` (niveau 1).
Chaque bit est transmis en une durée `D`, qui fixe le débit de la transmission. Le nombre de changements de niveaux par seconde est appelé **rapidité de modulation** (**RM**), et s'exprime en **bauds** (du nom de Baudot, l'inventeur du code TELEX). On a :

$$RM = \frac{1}{D}\ bauds \quad \text{et aussi :} \quad débit\ binaire = \frac{1}{D}\ bits/s$$

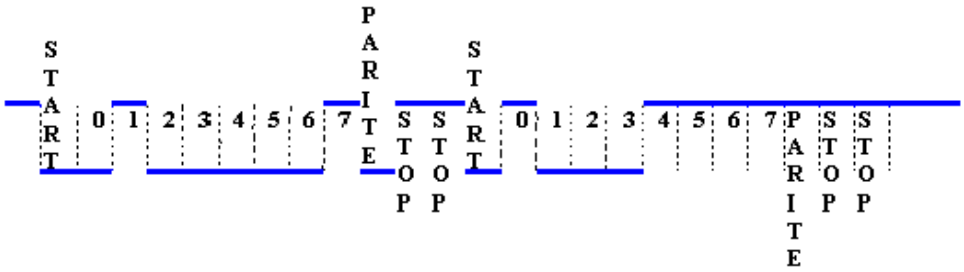
Le récepteur détecte l'arrivée d'un octet par le changement de niveau correspondant au bit `start`. Il **échantillonne** ensuite chaque intervalle de temps `D` au rythme de son horloge.
Comme les débits binaires de transmission série de ce type sont faibles (< 19600 bits/s) et que les horloges de l'émetteur et du récepteur sont suffisamment stables (horloges à quartz), il n'est pas nécessaire de les synchroniser. C'est la raison pour laquelle ce type de transmission série est qualifié d'**asynchrone**.
Lorsque les débits sont plus importants, la dérive des horloges entraînerait des erreurs et l'on doit alors mettre en oeuvre une transmission **synchrone** (voir le cours « réseaux »).
Il existe différentes vitesses normalisées : 115200, 9600, 4800, 2400, 1200... bauds. La communication peut se faire dans un sens (**simplex**) ou dans les deux sens (**duplex**). Dans ce dernier cas, on distingue le cas d'une alternance émission puis réception (**half-duplex**), ou celui d'une émission et réception simultanées (**full-duplex**).

Exemple : Soit à transmettre en parité paire, avec 2 bits de stop, le caractère `B` dont le codage ASCII est $01000010_{(2)}$ la trame sera la suivante d'un point de vue logique :



La **parité** est une technique de **détection d'erreurs** qui permet de vérifier que le contenu d'un mot n'a pas été modifié accidentellement lors de sa transmission.
L'émetteur compte le nombre de `1` dans le mot et met le bit de parité à `1` si le nombre trouvé est *impair*, ce qui rend le total pair : c'est la **parité paire**. On peut aussi utiliser la parité impaire.

Exemple: transmission de `$82`, puis `$F1`, avec parité paire et 2 bits de stop donne le logigramme suivant :



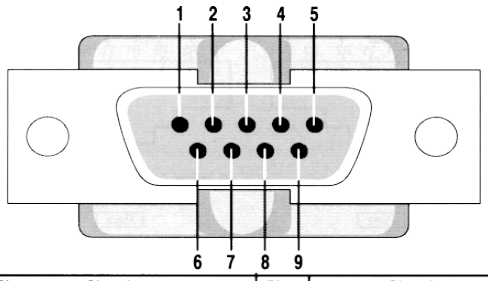
17.3.2.3. Normes RS-232 et V24

Ces normes spécifient les caractéristiques *mécaniques* (les connecteurs), *fonctionnelles* (nature des signaux) et *électriques* (niveaux des signaux) d'une liaison série asynchrone avec un longueur maximale de 15 m et une rapidité de modulation maximum de 20Kbauds.
L'**EIA** (*Electrical Industry Association*) a été à l'origine aux USA de la norme RS-232, dont la dernière version est RS-232C. Le CCITT (Comité Consultatif International pour la Téléphonie et la Télégraphie) a repris cette norme qu'il a baptisé V24. Deux autres normes permettent des débits plus élevés et des distances plus importantes : RS-423 (666m, 300kbauds), et RS-422 (1333 m, 10 Mbauds).

Tension	***État***
$-V_{MAX}$ à -3V	1 logique
-3V à 3V	zone interdite (élimination des problèmes de bruit)
3V à V_{MAX}	0 logique

Version de la norme	**V_{max}**
RS-232	48V
RS-232A	25V
RS-232B	12V
RS-232C	5V

La norme V24 utilise le connecteur DB25, de forme trapézoïdale à 25 broches, représenté figure suivante. Le connecteur DB9 à 9 broches, plus récent, remplace actuellement le DB25.



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Les niveaux électriques des bornes 2 et 3 (signaux d'information) sont compris entre +3V et $+V_{MAX}$ pour le niveau logique 0, et -3V et $-V_{MAX}$ pour le niveau logique 1 (niveau de repos).

Nom du Signal	*Abréviation*	*DB9*	*DB25*	
Donnée transmise	TD	3	2	Sortie
Donnée reçue	RD	2	3	Entrée
Demande d'émission	RTS	7	4	Sortie
Prêt à recevoir	CTS	8	5	Entrée
Connexion effectuée	DSR	6	6	Entrée
Détection de porteuse	CD	1	8	Entrée
Terminal prêt	DTR	4	20	Sortie
Indicateur sonnerie	RI	9	22	Entrée
Masse électrique	GND	5	7	

17.3.2.4. Contrôle de flux

a) Matériel (CTS/RTS)

Dans les explications suivantes, l'équipement qui envoie les données sera appelé l'**émetteur** et celui qui les reçoit le **récepteur**, quand bien même ils sont en fait émetteur et récepteur à tour de rôle.
L'émetteur envoie des données. Le récepteur les stocke dans une mémoire tampon. Lorsque cette mémoire atteint un seuil de remplissage défini, le récepteur supprime son signal CTS (passage au 1 logique). L'émetteur arrête immédiatement d'envoyer des données. Le récepteur continue de traiter les données qu'il a dans sa mémoire tampon. Lorsque sa mémoire tampon arrive au seuil d'espace libre suffisant, il réactive le signal CTS (passage au 0 logique). L'émetteur se remet à envoyer des données. Le cycle recommence jusqu'à ce que toutes les données aient été envoyées.

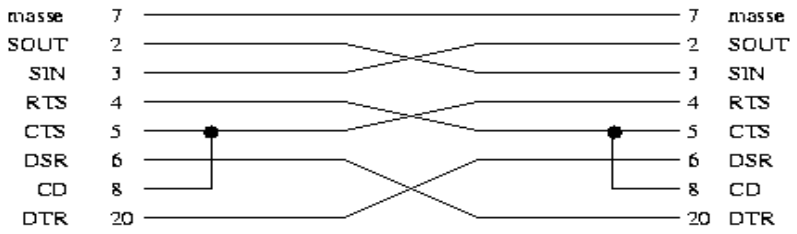
b) Logiciel (X_{ON}/X_{OFF})

Dans les explications suivantes, l'équipement qui envoie les données sera appelé l'émetteur et celui qui les reçoit le récepteur, quand bien même ils sont en fait émetteur et récepteur. L'émetteur envoie des données. Le récepteur les stocke dans une mémoire tampon. Lorsque cette mémoire atteint un seuil de remplissage défini, le récepteur envoie le code X_{OFF} (caractère de code décimal 17 / hexadécimal 0x11) à l'émetteur. L'émetteur arrête immédiatement d'envoyer des données. Le récepteur continue de traiter les données qu'il a dans sa mémoire tampon. Lorsque sa

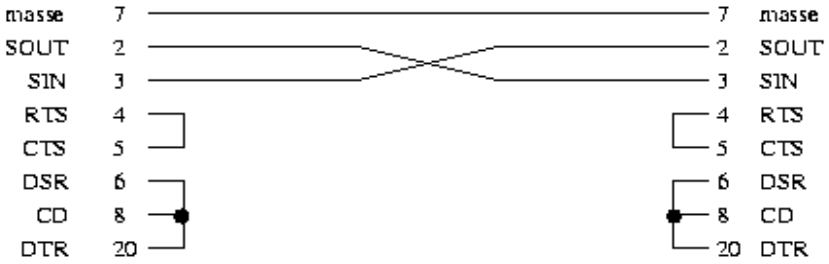
mémoire tampon arrive au seuil espace libre suffisant, par exemple 75%, il envoie le code X_{ON} (caractère de code décimal `19` / hexadécimal `0x13`) à l'émetteur. L'émetteur se remet à envoyer des données. Le cycle recommence jusqu'à ce que toutes les données aient été envoyées.

17.3.2.5. Câble NULL-MODEM

Il est possible de connecter deux PC par leurs interfaces séries. Si la distance est courte (inférieure à quelques dizaines de mètres), il n'est pas nécessaire d'utiliser un modem et on utilise alors un câble NULL-MODEM, qui croise certains signaux comme le montre la figure suivante.



Lorsque les signaux de dialogues ne sont pas nécessaires, il suffit de croiser les signaux SIN et SOUT, ce qui donne le câble NULL-MODEM simplifié (3 fils) représenté sur la figure suivante.



17.3.2.6. Autres interfaces séries

Un PC « ancien modèle » comportait au maximum quatre interfaces séries RS-232 appelées ports `COM1` à `COM4` sous Ms-Windows (ou `/dev/ttyS0` à `/dev/ttyS3` sous GNU/Linux) et accessibles généralement aux adresses :

N° port	*Adresse*	*IRQ*
1	3F8H	4
2	2F8H	3
3	3E8H	4
4	2E8H	3

Les nouvelles configuration d'ordinateurs personnels ont vu l'avènement des interfaces séries USB et *FireWire* (IEEE 1394) :

- **USB (*Universal Serial Bus*)**
  Contrairement au port série RS-232 qui accepte des vitesses n'allant que jusqu'au Mbaud,

l'USB quant à lui, possède un taux de transfert maximum de 12 Mbauds. Il permet de connecter à l'unité centrale jusqu'à 127 périphériques et de connecter ou déconnecter l'un de ces périphérique en mode « *Plug and Play* ».

– **IEEE 1394 ou *FireWire* (ligne de feu)**
Son taux de transfert maximum est de 400 Mbauds. Standard récent, il est utilisé pour la communication avec des périphériques graphiques ou vidéos tels que la caméra numérique. Il permet en outre de connecter jusqu'à 63 périphériques sur une même unité centrale et de connecter et déconnecter l'un de ces périphérique en mode « *Plug and Play* ».

– **RS-422 et RS-485**
Ces deux interfaces séries sont issues de la norme V11. Le support de transmission est ici `différentiel`. Deux conducteurs correspondant à des niveaux complémentaires sont donc utilisés pour chaque signal ce qui limite l'influence des bruits extérieurs et des masses. Pour la RS-485, des circuits trois états permettent des liaisons multi-points. Son taux de transfert maximum est de 10Mbits/s pour une longueur de câble maximum de 1km.

17.3.2.7. Programmation de l'interface série RS-232

Tout d'abord, pensez à consulter l'excellent document en ligne de Michael Sweet : *Serial Programming Guide for POSIX Operating Systems*, ou encore le HOWTO de Gary Frerking : *Linux Serial Programming HowTo*.

De la même façon que nous avons accéder à l'interface parallèle de différentes manières, voyons comment faire de même avec l'interface série.

a) Accès direct à l'interface série

L'exemple de code suivant permet d'accéder de manière *barbare* à l'interface série `/dev/ttyS0` placée à l'adresse `0x2F8` :. Elle nous oblige par contre à en connaître un peu plus sur la structure interne d'une carte d'interface série RS-232.
Le composant central d'une carte d'interface série RS-232 est un UART (*Universal Asynchronous Receiver Transmitter*). L'UART est composé d'un certain nombre de registres 8 bits :

- envoi des données : THR (*Transmission Holding Register*) pour la préparation des données à envoyer (rajout de la parité, des bits de stop et du start) et TSR (*Transmission Shift Register*) pour la sérialisation des données bit par bit sur la ligne de communication ;
- réception des données : RSR (*Receiver Shift Register*) pour la désérialisation des données en provenance de la ligne de communication et RDR (*Receiver Data Register*) pour l'épuration des bits de stop et de parité ;
- configuration : LCR (*Line Control Register*) pour la configuration de la communication (vitesse, parité, bits de stop), MCR (*Modem Control Register*) , IER (*Interrupt Enable Register*), etc.. Le registre LCR accessible à l'adresse `BASE + 3` est configurable de cette manière :

Bit 7	1	Bit `DLAB` (*Divisor Latch Access Bit*). Ce bit permet de configurer la vitesse de communication. Il permet le stockage du rapport de division `N` dans deux registres de 8 bits placés aux adresses `BASE` (poids faible) et `BASE+1` (poids fort). La valeur de `N` est telle que :
		`vitesse attendue = 115200 / N`
		115200 étant la vitesse maximale permise par le composant.
		Par exemple, pour une vitesse de 9600 bits/s :
		`N = 115200/9600 = 12 = 0x000C.`

		Il faudra donc écrire en `BASE` et `BASE+1` les valeurs `0x0C` et `0x00`.		
	0	Accès aux tampons de réception, transmission et au registre IER (*Interrupt Enable Register*)		
Bit 6	Active la commande *Break*			
Bits 3, 4 et 5	**Bit 5**	**Bit 4**	**Bit 3**	**Parité**
	X	X	0	Aucune
	0	0	1	Impaire (*odd*)
	0	1	1	Paire (*even*)
	1	0	1	Parité haute (*mark*) toujours à 1
	1	1	1	Parité basse (*space*) toujours à 0
Bit 2	**Nombre de bits de stop**			
	0	1 bit de stop		
	1	2 bits de stop pour les données de taille 6,7 ou 8 bits 1.5 bits de stop pour les données de taille 5 bits.		
Bits 0 et 1	**Bit 1**	**Bit 0**	**Longueur de la donnée**	
	0	0	5 bits	
	0	1	6 bits	
	1	0	7 bits	
	1	1	8 bits	

Voici un extrait du code :

```
#include <unistd.h>
#include <stdio.h>
#include <asm/io.h>

#define BASE_RS232_TTYS0 0x2F8 // adresse de base
#define LCR BASE_RS232_TTYS0+3 // registre LCR

int main(int argc, char **argv)
{
    /* La fonction ioperm() permet de positionner les
       autorisations d'entrée/sortie sur les ports à
       commencer de 0x2F8 et ce sur 8 octets.
       ATTENTION, l'exécutable devra posséder les droits
       du root pour fonctionner.
    */
    if (ioperm(BASE_RS232_TTYS0, 8, 1)) {
        perror("ioperm");
        exit(1);
    }

    /* Configuration du port série avec :
       9600 bauds, 8 bits de données, 1 bit de stop et
       parité paire */
    outb(0x80,LCR); /* Activation configuration vitesse */
    /* Ecriture du diviseur N=12 ==> vitesse = 9600bits/s */
    outb(0x0c,BASE_RS232_TTYS0);
    outb(0x00,BASE_RS232_TTYS0+1);
    outb(0x1b,LCR);  /* 8 bits / 1 stop / parité paire*/



    /* La valeur 41 (il s'agit du code ASCII du caractère
       'A') est écrite sur le port série /dev/ttyS0 */
    outb(41, BASE_RS232_TTYS0);
```

```
    // ...

    /* Fermeture du port par le programme */
    if (ioperm(BASE_RS232_TTYS0, 8, 0)) {
        perror("ioperm");
        exit(1);
    }
    return 0;
}
```

b) *Accès via le pseudo-fichier* `/dev/port`

Nous vous renvoyons à l'exemple fourni dans le paragraphe concernant l'accès à l'interface parallèle en utilisant le pseudo-fichier `/dev/port`. Avec l'exemple du dessus, la transposition devrait être aisée.

c) *Accès via l'interface POSIX* `termios`

Enfin pour l'accès utilisant l'interface POSIX `termios` définie dans le fichier `termios.h`, nous vous renvoyons aux pages de manuel `termios` ainsi qu'aux références Internet du dessus.

17.3.3. Conclusion

Malgré leur grand âge, les interface parallèles et séries sont encore largement utilisées dans de nombreuses applications industrielles (acquisition de données, pilotage d'équipements, etc.). La maîtrise des techniques d'exploitation de ces interfaces est souvent indispensable pour la réalisation d'un système industriel complexe.

18. EXERCICES

18.1. Exercices sur les variables

Écrire en langage C les programmes suivants :

1. « `taxe.c` » : programme qui à partir d'un prix H.T. affiche le prix T.T.C..
2. « `degresF.c` » : programme qui convertit des degrés Celsius en degrés Fahrenheit : $Fahrenheit = (\frac{9}{5}) \cdot Celsius + 32$
3. « `dec2hex.c` » : programme qui affiche en hexadécimal et en octal un nombre saisi en décimal. Faire aussi l'inverse en le nommant « `hex2dec.c` ».

18.2. Exercices sur les opérateurs et les structures de contrôle

Écrire en langage C les programmes suivants :

1. « `vote.c` » : programme qui demande l'âge de la personne. Si la personne a au moins 18 ans, alors on affiche « peut voter », sinon, on affiche « ne peut pas voter ».
2. « `divisible.c` » : programme qui indique si un nombre est divisible par 2. Il existe 2 méthodes, l'une utilisant l'opérateur modulo `%` et l'autre les masquages de bits.
3. « `decalage.c` » : programme qui prend un entier et opère un décalage de 1 bit à gauche. Qu'observe-t-on ? Et pour un bit à droite ?
4. « `simple.c` » : programme qui affiche 5 fois le caractère '`x`'.
5. « `boucles.c` » : programme qui affiche les nombres de 1 à 10 puis de 20 à 1 de trois en trois (faire deux fois l'exercice : avec « `for` » et avec « `while` »).
6. « `multiplications.c` » : programme qui affiche les tables de multiplication de 1 à 9.
7. « `puissance.c` » : programme qui saisit deux nombres entiers non signés et calcule le premier à la puissance du second.
8. « `calculette.c` » : faire une mini calculette proposant les 4 opérations :
```
$ ./calculette
> 3 4 +
7
> q
$
```
9. « `premiers.c` » : programme qui permet de saisir un nombre et d'indiquer si celui-ci est premier.

18.3. Exercices sur les fonctions et la récursivité

Écrire en langage C les programmes suivants :

1. « `fact.c` » : programme qui calcule la factorielle d'un nombre saisi au clavier. On utilisera deux méthodes différentes (itération et récursion) dans deux fonctions distinctes. On rappelle que $n! = n \cdot (n-1) \cdot (n-2) ... 3 \cdot 2 \cdot 1$

2. « `newton.c` » : programme qui calcule la racine carrée d'un nombre en utilisant la propriété de la série ci-après qui converge vers $\sqrt{a}$ lorsque $n \to \infty$:
   $U_{n+1} = \frac{1}{2} \cdot (U_n + \frac{a}{U_n})$ avec $U_0 = 5$
   Nous nous placerons dans le cas où :
   $\forall\ e > 0,\ \exists\ N(e)\ /\ |U_N - U_{N+1}| < e$ et fixerons e pour obtenir 10 chiffres significatifs.

3. « `arctangente.c` » : La série qui permet de calculer directement l'arc de cercle dont la tangente est `x` s'écrit :
   $\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots + \textit{terme général} + \ldots$
   Le terme général est celui qui permet de représenter tous les autres et est ici :
   $X(n) = \frac{(-1)^n \cdot x^{(2n+1)}}{(2n+1)}$
   Cette série converge très lentement. C'est dire qu'il faut calculer et additionner beaucoup de termes pour avoir une précision acceptable. Proposer la fonction de calcul sous deux formes :
   La première avec une boucle classique (`de i=0 à i=n, j'additionne les X(i)`) et la seconde en utilisant la récurrence. La bibliothèque `<math.h>` possède une fonction, `double pow(double x, double n)` qui renvoie un réel double égal à x^n. Remarque : l'utilisation de la bibliothèque mathématique impose que vous compiliez avec l'option `-lm`.
   A titre de contrôle sachez que `arctan(1) = 0.787873` pour 100 termes.

18.4. Exercices sur les tableaux

Écrire en langage C les programmes suivants :

1. « `notes.c` » : programme qui saisit les notes de 10 élèves, qui fait la moyenne des notes, puis qui affiche les notes au-dessus de la moyenne.
2. « `tri_nombres.c` » : programme qui saisit une liste de nombres (maximum 25), la trie puis la réaffiche.

18.5. Exercices sur les chaînes de caractères

Écrire en langage C les programmes suivants :

1. « `strlen.c` », « `strcpy.c` » et « `strcmp.c` » : programmes qui réimplémentent les fonctions du même nom.
2. « `renverse.c` » : programme qui saisit une chaîne de caractères au clavier et l'affiche à l'envers.
3. « `palindrome.c` » : programme qui saisit une chaîne de caractères au clavier et indique s'il s'agit ou non d'un palindrome. Exemple de palindrome : « `ET LA MARINE VA VENIR A MALTE` ».

18.6. Exercices sur les pointeurs

Écrire en langage C les programmes suivants :

1. « `carre.c` » : créer une fonction qui remplace la valeur de l'entier passé en argument par

son carré.
2. « `echange.c` » : fonction qui échange les valeurs de deux entiers passés en argument.
3. « `echangep.c` » : fonction qui échange les valeurs de deux pointeurs passés en argument.

18.7. Exercices sur le passage d'arguments

Écrire en langage C les programmes suivants :

1. « `params.c` » : programme qui affiche tous les arguments passés en paramètre en indiquant leur position :
```
$ ./params un pomme 1234 tapis
1 un
2 pomme
3 1234
4 tapis
```
2. « `additionne_params.c` » : programme qui additionne tous les nombres passés en argument et qui affiche le résultat.
3. « `encadre.c` » : programme qui encadre une phrase passée en argument de la façon suivante :
```
$ encadre + "Bonjour a tous"
++++++++++++++++++
+ Bonjour a tous +
++++++++++++++++++
```

18.8. Exercices sur les structures

Réaliser un programme « `complexe.c` » qui permet de manipuler les nombres complexes de la forme :

$$z = a + ib \text{, où } a \text{ et } b \in \mathbb{R} \text{ et } i^2 = -1$$

1. Créer une structure `Complexe` contenant les champs « partie réelle `a` » et « partie imaginaire `b` ».
2. Écrire les fonctions d'addition (`Complexe additionner(Complexe z1,Complexe z2);`) de deux nombres complexes et de multiplication (`Complexe multiplier(Complexe z,float n);`) d'un nombre complexe par un réel. Les tester.
3. Écrire et tester la fonction de calcul du module d'un nombre complexe `z` : `double calculerModule(Complexe z);` Rappel : $|z| = \sqrt{a^2 + b^2}$
4. Écrire et tester la fonction de calcul de l'argument d'un nombre complexe `z` : `double calculerArgument(Complexe z);` Rappel : $arg(z) = \arccos\left(\frac{a}{|z|}\right)$
5. Nous voulons maintenant représenter un nombre complexe sous sa forme exponentielle : $z = |z| \cdot e^{i \cdot arg(z)}$. Créer une structure `ComplexeExp` permettant cette représentation.
6. Écrire et tester une fonction de conversion d'un nombre complexe depuis sa forme algébrique vers sa forme exponentielle : `ComplexeExp convertir(Complexe z);`

18.9. Exercices sur les champs binaires

Réaliser en langage C les programmes suivants :

1. « `dec2bin.c` » : programme qui affiche un nombre entier en binaire à l'aide des opérateurs >> et `&`.
2. « `rvb.c` » : programme qui affiche les composantes « Rouge », « Vert » et « Bleu » d'un pixel de couleur, lorsque la valeur de celui-ci est saisie au clavier sous forme entière. Rappel : un pixel de couleur est défini par la suite des quatre octets : `<Alpha><Rouge><Vert><Bleu>` formant une valeur entière de 32 bits.

18.10. Exercices sur les fichiers

Réaliser en langage C les programmes suivants :

1. « `cat.c` » : programme qui émule la commande `cat`.
2. « `compte_lignes.c` » : programme qui compte les lignes d'un fichier.
3. « `wc.c` » : programme qui émule la commande `wc`.
4. « `accolade.c` » : programme qui compte les nombres d'accolades fermantes et ouvrantes d'un code source en C passé en argument. Il signale une erreur si les deux nombres obtenus ne sont pas identiques.
5. « `decommente.c` » : programme qui supprime les commentaires (`/* */`).
6. « `ls.c` » : programme qui émule la commande `ls`.

18.11. Exercices sur les allocations dynamiques

Réaliser en langage C les programmes suivants :

1. « `alloc.c` » : allouez dynamiquement de la mémoire pour un tableau de 100 entiers.
2. « `bigtab.c` » : faire un programme qui saisit un nombre non prédéterminé d'entiers, les stocke dans un tableau et les réaffiche.

18.12. Exercice sur le débogage avec gdb

Le polynôme suivant permet le calcul de la variable réelle `y` en fonction de la variable réelle `x` :

$$y=\frac{(x-1)}{x}+\frac{1}{2}\frac{(x-1)^2}{x^2}+\frac{1}{3}\frac{(x-1)^3}{x^3}+\frac{1}{4}\frac{(x-1)^4}{x^4}+\frac{1}{5}\frac{(x-1)^5}{x^5}$$

On peut simplifier ce calcul en posant : $u=\frac{(x-1)}{x}$

Par exemple, pour `x = 2.0` alors `u` vaut `0.5` et `y = 0,688`.
Le programme que l'on a écrit pour résoudre ce problème est le suivant :

Source polynome.c :

```
1 /* Polynome.c */
2 #include <stdio.h>
3 #include <math.h>
4 main()
5 {
6    float u = 0, x = 0, y = 0;
7    // lecture de x avec scanf
8    ...
9    // calcul des solutions
10   u = x - 1 / x;
11   y = u+pow((u/2),2.)+ pow((u/3),3.)+pow((u/4),4.)+pow((u/5),5.);
```

```
12    // affichage des résultats
13    printf("\nx = %f et y = %f\n", x, y);
14    return 0;
15 }
```

Effectuer, à l'aide de **gdb**, un relevé (sous forme de tableau) des valeurs prises par chaque variable aux différentes étapes du programme. Corriger le programme si nécessaire.

18.13. Exercices sur la programmation système Linux

18.13.1. Gestion des processus

1) Reprendre le programme affichant son PID et celui de son père, et ajouter un appel à la fonction `sleep()` pour attendre 10 secondes avant d'exécuter les appels `getpid()` et `getppid()`.

- `unsigned int sleep(unsigned int s)` suspend l'exécution du programme pendant `s` secondes.

Vérifier que le terminal de commande où le programme est lancé est bien le père du processus correspondant. Relancer ensuite le programme, mais en tâche de fond et en redirigant la sortie dans un fichier :

```
./exo > sortie &
```

et fermer la fenêtre en tapant **exit**. Attendre 10 secondes et regarder le contenu du fichier contenant la sortie du programme. Remarques ?

2) Écrire un programme créant un processus fils. Le processus père affichera son identité ainsi que celle de son fils, le processus fils affichera son identité ainsi que celle de son père.

3) Nous allons essayer de reproduire le phénomène du premier exercice. Pour cela, il faut ajouter un appel à la fonction `sleep()`, de façon à ce que le père ait terminé son exécution avant que le fils n'ait appelé `getppid()`.

4) Ajouter un appel à la fonction `wait(NULL)` pour éviter que le père ne se termine avant le fils.

5) Dans l'exercice précédent, utiliser `wait()` pour obtenir le code de retour du fils et l'afficher.

18.13.2. Recouvrement de processus

1) Créer un processus et le recouvrir par la commande `du -s -h`. Attention, ce programme n'a pas besoin d'utiliser l'appel système `fork()`.

2) Reprendre le premier exemple de la série d'exercices précédente (l'application affichant son PID et celui de son père). Nous appellerons ce programme **identite**. Écrire ensuite une application (semblable à celle de la question 2 de la série d'exercices précédente) créant un processus fils. Le père affichera son identité, le processus fils sera recouvert par le programme **identite**.

3) Écrire une application **monshell** reproduisant le comportement d'un terminal de commande. Cette application doit lire les commandes tapées au clavier, ou contenue dans un fichier, créer un nouveau processus et le recouvrir par la commande lue. Par exemple :

```
./monshell
ls
extp3_1.c       extp3_2.c       identite        monshell
soltp3_1.c      soltp3_2.c      soltp3_3.c
ps
  PID  TT  STAT      TIME COMMAND
  521 std  S      0:00.95 bash
```

```
  874  p3  S      0:00.02 bash
 1119  p3  S+     0:00.56 vi soltp3_3.c
 1280  p2  S      0:00.03 tcsh
 1283  p2  S+     0:00.04 vi
 1284 std  S+     0:00.01 ./monshell
exit
```

Pour la commodité d'utilisation, on pourra faire en sorte que le terminal de commande affiche une invite (un prompt) avant de lire une commande :

```
./monshell
monshell>ls
extp3_1.c       extp3_2.c       identite        monshell
soltp3_1.c      soltp3_2.c      soltp3_3.c
monshell>exit
```

On pensera à utiliser la fonction `wait()` afin d'attendre que l'exécution d'une commande soit terminée avant de lire et de lancer la suivante.
Que se passe-t-il si l'on tape à l'invite **`ls -l`** ?

---

4) Ajouter au terminal de commande de la question précédente la gestion des paramètres pour permettre l'exécution de **`ls -l`** par exemple. Est-ce que la ligne **`ls *.c`** se comporte comme un vrai terminal de commande ?

---

5) Ajouter au terminal de commande de la question précédente la gestion des exécutions en tâche de fond, caractérisée par la présence du caractère `&` en fin de ligne de commande.

18.13.3. Les signaux

1) Écrire un programme **`sigusr1`** qui, sur réception du signal `SIGUSR1`, affiche le texte « `Réception du signal SIGUSR1` ». Le programme principal sera de la forme :

```
for (;;) { /* boucle infinie équivalente à while (1) {} */
   sigset_t sigmask ;
   sigemptyset(&sigmask);
   sigsuspend(&sigmask);
}
```

et il serait judicieux de faire afficher au programme son numéro de processus à son lancement. La fonction `sigaction()` déclarée dans `signal.h` doit être utilisée.

---

2) La fonction `alarm()` permet de demander au système d'exploitation d'envoyer au processus appelant un signal `SIGALRM` après un nombre de secondes donné.
Écrire un programme **`alarm`** qui demande à l'utilisateur de taper un nombre. Afin de ne pas attendre indéfiniment, un appel à `alarm()` sera fait juste avant l'entrée du nombre pour que le signal `SIGALRM` soit reçu au bout de 5 secondes.
Dans ce cas, on veut que l'appel système de lecture du clavier soit interrompu lors de la réception du signal, il faut donc initialiser le membre `sa_flags` de la structure `sigaction` à 0.
Fonctions à utiliser :

```
#include <signal.h>
#include <unistd.h>
int sigaction(int sig, struct sigaction *act, struct sigaction *oact)
int alarm(int secondes)
```

---

3) Écrire un programme **`sigchld`** qui appelle `fork()`. Le père mettra en place une fonction pour traiter le signal `SIGCHLD` (qui affichera, par exemple, « `Réception du signal SIGCHLD` »),

attendra 10 secondes, puis affichera « `Fin du père` ». Le fils se affichera « `Fin du fils dans 2 sec` », puis se terminera deux secondes après.
Quel est l'intérêt du signal `SIGCHLD` ? Qu'est-ce qu'un zombie ?

18.13.4. Les tuyaux

1) Écrire un programme **`tuyau1`** qui met en place un tuyau et crée un processus fils. Le processus fils lira des lignes de caractères du clavier, les enverra au processus père par l'intermédiaire du tuyau et ce dernier les affichera à l'écran.
Le processus fils se terminera lorsqu'on tapera `<Ctrl-D>` (qui fera renvoyer `NULL` à `fgets()`), après avoir fermé le tuyau. Le processus fils se terminera de la même façon.
Cet exercice permet de mettre en évidence la synchronisation de deux processus au moyen d'un tuyau.
Fonctions à utiliser :

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int pipe(int tuyau[2])
pid_t fork()
FILE *fdopen(int fichier, char *mode)
char *fgets(char *chaine, size_t taille, FILE *fichier)
```

---

2) On se propose de réaliser un programme mettant en communication deux commandes par l'intermédiaire d'un tuyau, comme si l'on tapait dans un *shell* :

```
ls | wc -l
```

La sortie standard de la première commande est reliée à l'entrée standard de la seconde.
Écrire un programme **`tuyau2`** qui met en place un tuyau et crée un processus fils. Le processus père exécutera la commande avec option **`sort +4 -n`** grâce à la fonction `execlp()` et le processus fils exécutera la commande avec option **`ls -l`**. Pour cela, il faut auparavant rediriger l'entrée standard du processus père vers le côté lecture du tuyau et la sortie standard du processus fils vers le côté écriture du tuyau au moyen de la fonction `dup2()`.
Fonctions à utiliser :

```
#include <sys/types.h>
#include <unistd.h>

int pipe(int tuyau[2])
pid_t fork()
int dup2(int descripteur, int copie)
int execlp(const char *fichier, const char *arg, ...)
```

---

3) Reprenez le programme **`monshell`** précédent et modifiez-le afin qu'il reconnaisse aussi les commandes de la forme

```
ls -l | sort +4 -n
```

Fonctions à utiliser :

```
#include <sys/types.h>
#include <string.h>
#include <unistd.h>

char *strtok(char *chaine, char *separateurs)
```

```
int pipe(int tuyau[2])
pid_t fork()
int dup2(int descripteur, int copie)
int execlp(const char *fichier, const char *arg, ...)
```

La fonction `strtok()` vous sera utile pour analyser la ligne tapée au clavier.
Si vous avez le temps, essayez de modifier le programme pour qu'il reconnaisse un enchaînement quelconque de commandes :

```
cat toto | grep tata | wc -l
```

Dans quel ordre vaut-il mieux lancer les processus fils ?

18.13.5. L'interface parallèle IEEE 1284

Système d'accès à une salle sensible
Dans le cadre d'un système d'accès à une salle sensible, nous souhaitons mettre en oeuvre l'interface parallèle IEEE 1284 pour piloter l'ouverture et la fermeture d'une porte depuis l'ordinateur. Une carte TOR (Tout Ou Rien), reprenant le schéma de principe de l'illustration 4, permet d'envoyer un signal d'activation à une gâche électrique.
Écrire et tester le programme **gache** qui en permet le pilotage.

18.13.6. L'interface série RS-232

1ère partie : ***Mise en Oeuvre***
1) Rappeler la plage des tensions pour une liaison série RS-232B.
2) Tracer le chronogramme d'une transmission série asynchrone pour le message « BRAVO » avec la configuration suivante : vitesse de transmission de 9600 bauds, 8 bits de données, 1 bit de stop et parité paire.
3) Calculer la durée minimum de transfert du message précédent.
4) Sous GNU/Linux, configurer après l'avoir installé le programme **minicom** pour qu'il assure une communication à : 1200 bauds, 7 bits de données, parité impaire, 2 bits de stop et pas de contrôle de flux.
5) Le câble NULL-MODEM qui vous est fourni est-il simplifié ou complet ? Le vérifier.
6) Connecter le câble NULL-MODEM au PC du binôme voisin puis tester la communication. De quel type de communication s'agit-il (*simplex*, *full-duplex*, *half-duplex*) ? Que se passe-t-il en cas de transfert d'un gros fichier ? Est-il nécessaire d'effectuer un contrôle du flux ?

2ème partie : ***Système d'accès à une salle sensible***
Toujours dans le cadre de notre salle sensible, nous souhaitons effectuer l'analyse des badges RFID (voir description ci-après) portés par les personnels qui désirent accéder à cette salle. Ces badges sont détectés par un module RFID relié à un ordinateur via une liaison série RS-232.
Le module transmet une suite d'octets comprenant le numéro du badge, à la vitesse de 9600bps, avec 8 bits de données, 1 bit de stop et aucune parité.
1) Écrire le programme **rfid** chargé de lire le numéro du badge depuis le module, puis de vérifier sa présence dans un fichier texte contenant les champs :

```
date_accès   heure_accès   numero_rfid   numero_salle   nom   prenom
```

2) Intégrer le pilotage de la gâche réalisé précédemment puis valider dans des cas de figure positif et négatif.

La technologie RFID
RFID qui signifie *Radio Frequency IDentification* (Identification par Radio Fréquence) est un terme générique pour toutes les technologies qui utilisent les ondes radio pour identifier automatiquement des objets. L'utilisation la plus courante de la technologie RFID est le stockage d'un numéro de série sur une puce reliée à une antenne, cela afin d'identifier un produit de manière unique. L'ensemble puce-antenne s'appelle **transpondeur RFID** ou encore **étiquette RFID**. L'antenne permet à la puce de transmettre les informations d'identification à un lecteur distant qui convertit les ondes radio émises par cette étiquette RFID, en données numériques.

Fonctionnement
Le système comprend un transpondeur ainsi qu'un lecteur (avec antenne intégrée ou non). Le lecteur envoie un signal pour se mettre en connexion avec l'antenne de l'étiquette RFID et la puce retourne alors au lecteur un signal comprenant son identification. La distance de lecture des étiquettes dépend de plusieurs facteurs : la fréquence de fonctionnement, la puissance du lecteur, les interférences avec des objets métalliques, la présence d'eau, etc.. En moyenne, les étiquettes basse-fréquence peuvent être lues jusqu'à 30 cm. En Europe, les réglementations imposent une distance maximum d'un mètre pour ces technologies.

Différents types d'étiquettes
Une étiquette est constituée d'une puce avec une bobine formant l'antenne. Elle se présente sous deux aspects : intégrée dans une étiquette papier ou moulée dans un support (palet, boîte, capsule, etc.). Les puces des étiquettes peuvent être soit en lecture-écriture, soit en lecture seule. Avec les puces en lecture-écriture, l'utilisateur peut ajouter des informations sur l'étiquette

Fréquences utilisables
Les systèmes RFID utilisent plusieurs fréquences différentes, mais généralement les plus courantes sont en basse-fréquence autour de 125KHz, d'autres plafonnent à 13,56MHz. Les étiquettes basse-fréquence consomment moins et pénètrent plus dans les matières non métalliques que celles en haute-fréquence. Elles sont idéales pour scanner des objets à haute teneur en eau tels que le papier. Les étiquettes haute-fréquence offrent typiquement une meilleure portée et un transfert de données plus rapide, mais elles consomment plus, pénètrent moins dans la matière et nécessitent un chemin dégagé entre l'étiquette et le lecteur.

Matériel utilisé

- deux cartes transpondeurs RFID opérant sur la fréquence 125Khz et conçues sur la base d'un circuit EM4102 d'EM-Microelectronic. Elles disposent d'une capacité mémoire de 5 octets faisant office de numéro de série unique pré-configuré en usine (ces données sont associées à 14 bits de parité permettant un contrôle supplémentaire). Ces cartes à lecture seule, pourront être utilisées jusqu'à une distance maximum de 120mm. La documentation du transpondeur est disponible en suivant le lien : EM4102_DS.pdf ;

- un module RFID UM005 opérant à 125KHz et nécessitant une alimentation de 5V ainsi qu'une antenne externe. Il dispose d'une sortie LED, d'une sortie *buzzer* et d'une sortie série RS-232 (9600bps, 8 bits de données, 1 bit de stop,

aucune parité). Le passage d'un transpondeur devant l'antenne, provoque l'émission sur la liaison série d'une suite d'octets le représentant. La documentation du module est disponible en suivant le lien : UM-005_eng.pdf ;

- une platine d'évaluation pour le module UM005.

---

19. ANNEXE

19.1. Installer une application GNU sous UNIX

1. Décompresser l'archive à l'aide de la commande **tar** :
   ```
   tar xfz appli.tgz
   ```
2. Aller dans le répertoire créé :
   ```
   cd appli
   ```
3. Lire le fichier README et le fichier INSTALL qui accompagnent la distribution.
4. Compiler et installer :
   ```
   ./configure
   make
   make install
   ```

19.2. Compilation séparée

Notions : compilation séparée, librairies, .o, options de **gcc** (-l, -L, -I, etc.), **make**.

La compilation séparée permet de produire du code réutilisable (bibliothèques) et plus fiable. Elle est indispensable pour les gros projets.

19.2.1. Un premier exemple de compilation séparée

Prenons le code d'une fonction longueur qui renvoie la taille d'une chaîne de caractères. Cette fonction est très utile et devrait pouvoir être réutilisable dans n'importe quel autre programme.

19.2.2. Compiler un fichier objet qui contient les fonctions

Voici le code du fichier longueur.c :

```
#include <stdio.h>

int longueur(char tab[])
{
        int i=0;
        while(tab[i])
           i++;
        return i;
}
```

On compile ce fichier afin de produire un fichier « objet » dont l'extension est .o :

```
gcc -Wall -c longueur.c
```

On obtient le fichier longueur.o qui contient le code compilé de la fonction longueur. La

commande UNIX **nm** nous permet de vérifier que la fonction longueur est bien définie dans le fichier longueur.o :

```
nm longueur.o
00000000 t gcc2_compiled.
00000000 T longueur
```

19.2.3. Utiliser le fichier objet

Attention, le fichier objet obtenu ne peut être exécuté directement. Pour ce faire, il faut créer un programme, dans un autre fichier, qui va faire appel à la fonction que nous avons définie. Exemple, le fichier main.c :

```
#include <stdio.h>
extern int longueur(char *);
int main()
{
   char chaine[] = "Bonjour";
   printf("%s fait %d caractères\n", chaine,
                    longueur(chaine));
   return 0;
}
```

Pour pouvoir utiliser la fonction longueur, il faut déclarer son prototype au début du programme en mettant la directive extern devant. Cette directive indique au compilateur que la fonction est présente dans un autre fichier que dans celui du programme principal.
Pour compiler le programme contenu dans main.c :

```
gcc -Wall main.c longueur.o -o prog
```

19.2.4. Un code plus propre

A chaque fois que nous voulons utiliser une fonction externe, il faut déclarer son prototype. Il existe un moyen propre de faire cela : utiliser un fichier d'en-tête. Voici, dans le cas présent, ce fichier que nous appellerons fonc.h :

```
int longueur(char *);
```

On inclue le code du fichier d'en-tête dans le code source du programme avec la directive #include :

```
#include <stdio.h>
#include "fonc.h"

int main()
{
   char chaine[] = "Bonjour";
   printf("%s fait %d caractères\n", chaine,
           longueur(chaine));
   return 0;
}
```

19.2.5. Automatiser la compilation avec `make`

Exemple de fichier `Makefile` :

```
OBJ=prog

prog : main.c longueur.o
        gcc -Wall $^ -o $@

%.o : %.c
        gcc -Wall -c $^

clean :
        rm -f *.o $(OBJ)
```

Variables prédéfinies :

`$^` :	dépendances
`$@` :	cible
`$<` :	nom de la première dépendance
`$(basename filename)` :	préfixe de la cible courante