Le Langage C

Luc Courtrai

Institut Universitaire Professionnalisé de Vannes
Université de Bretagne Sud
Tohannic, rue Yves Mainguy - 56000 Vannes (France)
E-mail : Luc.Courtrai@univ-ubs.fr

30 septembre 2013

Bibliographie :
– Le Langage C, Norme ANSI , Dennis Ritchie, Brian Kernighan (Dunod)
– La Bibliothèque C standard, P.J. Plauger (Masson, Prentice hall)

Table des matières

Ces quelques pages présentent les principales caractéristiques du langage C. Ce n'est pas un manuel de programmation mais plutôt une aide à l'apprentissage du langage. Tous les exemples sont écrits dans la norme ANSI.

Introduction

Le langage C a été développé, dans le début des années 1970 par Dennis Ritchie dans les laboratoire BELL.

Le C est très lié au système d'exploitation Unix, développé dans le même laboratoire. Unix est développé à plus de 95 pourcent en C (le reste est en assembleur).

Qualités du langage

- Le C est un langage procédural (classe de langage : procédural, déclaratif, fonctionnel,). Il est modulaire, structuré et typé.
  La principale qualité du C est l'efficacité des programme exécutable produit par son compilateur (Échelle du temps d'exécution d'une même application ecríte en : Assembleur 1, C 1,5, Pascal 5).
- Le C possède de nombreuses librairies spécialisées (système, graphique, entrée-sortie, mathématique, date, réseau ..)
- Le C, dans sa norme ANSI, est portable et donc non lié à une architecture matérielle et le compilateur.

Inconvénients du langage

- lisibilité de certaines expressions
  `(((~(~0<<b)&n) << (sizeof(unsigned) - b)) | (n >> b)`
  C'est un décalage circulaire à droite de b bits sur l'entier n.
- Peu de contrôle sur les types (limite des tableaux , accès mémoire, affectation, type par défaut (int et float), ...)

1 Notions de base du langage

Cette première partie présente les notions minimales nécessaires pour commencer la programmation C. Elle reprend la programmation classique "à la Pascal" et permet le développement rapide de quelques programmes.

Déclarations des types de base (int, float, char)

Exemples :

```
int  i; /* déclaration d'une variable de type entier */
float a1,a2; /* déclaration de 2 variables de type réel */
char c1,c2; /* déclaration de 2 variables de type caractère  */
```

NB : Chaque ligne correspond dans l'exemple à la déclaration d'une ou plusieurs variables de même type. Les noms des variables sont séparés par des virgules.

NB : Le texte encadré par les symboles "/" et "*/ " est un commentaire du programme. Il est impossible d'emboîter des commentaires sur plusieurs niveaux.*

De plus en plus les compilateurs acceptent les commnentaires sur une fin de ligne, le commentaire commence par un // (commentaire C++.

Le nom d'une variable (ou d'un autre objet *C*) doit obligatoirement commencer par une lettre (minuscule ou majuscule) ou par un tiret bas " _".

Initialisation d'une variable

Exemples :

```
int i =4; /* déclaration d'une variable entière initialisée à la valeur 4 */
float a =2.1, b=3.; /* déclaration et initialisation d'une variable réelle  */
char c1='A',c2=32; /* déclaration et initialisation de 2 variables caractères */
/* la variable c2 est initialisée avec un entier correspondant à sa valeur Ascii  */
```

L'affectation (=)

Exemples :

```
int i1, i2 ; /* déclaration de 2 variables entières */
i1 = 4 ; /* la constante 4 est affectée à la variable i1 */
i2 = i1 ; /* la valeur dans la variable i1 est affectée à la variable i2 */
```

L'instruction conditionnelle (if)

Syntaxe :

```
if ( expression )
   instruction;
```

```
if ( expression )
   instruction;
else
   instruction;
```

```
if ( expression ) {
   instruction1;
   instruction2;
   ...
   instructionN;
} else {
   instruction1;
   instruction2;
   ...
   instructionN;
}
```

L'expression est calculée à l'exécution ; si elle retourne une valeur différente de zéro elle est considérée comme vraie.

Exemple :

```
int sup;
int x=4,y=12;
if (x > y)
  sup = x ;
else
  sup = y ;
```

Les opérateurs de base (+, -, /, *, %, >, >=, <, <=, ==, !=, &&, | |, !)

```
+, -, /, *        /* opérateurs arithmètiques */
%                 /* modulo */
>, <, >=, <=      /* opérateurs de comparaison */
= =, !=           /* opérateurs d'egalité et d'inegalite */
&&                /* et logique ex : if ((a>='0') && (a<='9')) */
| |               /* ou logique ex : if ((a>='0') | | (a<='9')) */
!                 /* opérateur not, ex : if (! (a==2) )  */
```

*NB : La liste complète des opérateurs est donnée en Annexe. Les priorités entre les opérateurs restent classiques (par exemples les * sur les +) et les parenthèses permettent de forcer certaines priorités. Le && ou le || étant prioritaires sur les opérateurs <,<=,>,>=,==,|=, un bon parenthésage est nécessaire (voir exemple).*

Les opérateurs sont les mêmes quelque soit le type (*int* ou *float*). Si les deux opérandes sont entiers, le résultat sera entier même si la variable devant récupérer ce résultat est réel. Par contre si un des deux opérandes est entier, il y a un "équilibrage" des opérandes et le résultat sera réel. Il faut donc éventuellement forcer le type d'un opérande (cast). Exemple de "forcage de type" (transtypage) : (float)2 où la constante entière 2 est transformée en constante réelle.

Exemples :

```
int i1 = 3, i2 = 2 ;
float f1 ;
f1 = i1/2 ;                    /* 2 est une constante entière donc f1 = 3/2 ->1 */
f1 = i1/2. ;                   /* 2. est une constante réelle donc f1 = 3/2. ->1.5 */
f1 = i1/(float)2 ;             /* La constante entière est tranformée en réel f1 = 1.5 */
f1 = i1/i2 ;                   /* i1,i2 sont entiers donc f1 = 1 */
f1 = i1/(float)i2 ;            /* la valeur de i2 est forcée en réel,donc f1 = 1.5 */
f1 = (float)(i1/i2) ;          /* la valeur est forcée en réel mais apres le calcul ,donc f1 = 1 */
```

L'instruction itérative (while), boucle tant que

Syntaxe :

```
while ( expression )
        instruction ;
```

```
while ( expression ) {
        instruction1 ;
        instruction2 ;
        ...
        instructionN ;
}
```

Tant que la valeur de l'expression est vraie (valeur différente de zéro), le corps de la boucle est exécuté.

Exemple :

```
// factorel fac = val!
int val = 3; // valeur de la fonction
int fac = 1; // resultat du factorel
             // initialise a 1  element neutre de la mutipli
while (val > 1) {
     fac = fac * val;
     val = val - 1;
}
```

Les entrées/sorties formatées (printf et scanf)

Syntaxe :

sortie écran.
printf (“format”,liste d’expressions) ;
Le format permet d’imprimer les variables de la liste, il peut contenir des caractères spéciaux :
%d valeur de type entier
%c valeur de type caractère
%f valeur de type réel
\n passage à la ligne
printf (**“** valeurs de i = **%d,** de i + j = **%d \n”,** i, i + j) ; /* *affiche deux entiers et un retour chariot* */

entrée clavier
scanf(“format”,liste d’adresses de variables) ;
Le format est identique au printf
scanf(“%d”,&i) ; /* *le & devant le i permet a la fonction scanf d’accèder au contenu de la variable* */

exemple de saisie d’une variable
printf(“entrer une valeur : “) ;
scanf(“%d”,&i) ;

Le format peut être précisé en donnant une taille.
%nd *valeur entière sur n caractères (chiffres)*
%n1.n2f *valeur réelle sur n1 caractères avec n2 chiffres derrière le point d*
les nombres sont complétés par des blancs (cadrage à droite)

printf(“**%10d**,**%10.2f**\n”,100,2/3.) /* *bbbbbb*100,*bbbbb*0.66 */

La séquence (,)

Toutes les instructions C retournent une valeur.
Exemples :

int a ;
a ; /* *l’instruction sans effet, mais correcte* */
if (a) intruction ; /* *si a possède une valeur différente de 0, l’instruction est exécutée* */

La séquence permet comme le **“ ;”** d’écrire une suite d’instructions ; mais la valeur retournée par la séquence est la valeur retournée par la dernière instruction de la séquence.
Exemple :

int a **;**
if (**scanf(“%d”,&a) ,a**) /* *ici l’expression retourne la valeur de a* */
while (**a = a -1, printf(“%d”,a), a**) ; /**dans l’expression du tant que on décrémente la valeur de a, on l’affiche et on teste sa valeur pour sortir de la boucle. Le corps de la boucle est ici “vide” (;)* */

La séquence permet alors d’insérer des instructions dans des expressions.

Exemple complet de programme C

Voici un programme C qui affiche la valeur maximale d’une liste de naturels saisis au clavier :

```
# /*Affiche le max d'une suite de naturels*/
#include<stdio.h> /* accès à la librairie des entrées/sorties printf et scanf */
int main(void) { /* fonction principale, lancée au début du programme */
    int max = 0 , cour ;
    printf ("Entrer un : liste de naturels terminee par un 0 \n" ) ;
    while (scanf("%d",&cour),cour != 0) {
        if (cour > max)
            max = cour ;
    } ;
    if (max == 0)
        printf("Liste vide\n") ;
    else
        printf("Le max est %d\n",max) ;
}
```

Voici la ligne de commande pour compiler le programme précédent :

```
cc -o max max.c
```

Cette compilation génère à partir du fichier source "max.c" un exécutable "max".
Exécution du programme

```
> max
Entrez une liste de naturels terminee par un 0 :
101 234 56 45 0
Le max est 234
```

2 Les différents types numériques

Les autres types de base (unsigned, short, long, double)

Le langage C possède plusieurs types d'entier et de réel. Ils se différencient par la taille de leur représentation en mémoire. Ils permettent une meilleure gestion de l'espace mémoire.

Le type *unsigned* définit des entiers non-signés (les naturels). Il sont codés directement dans la base 2.

Exemple : (unsigned) 19 -> 00000000 00010011 (représenté ici sur 2 octets)

Le type *char* est codé sur 1 octet. Le type *char* étant compatible avec les entiers signés, il constitue les entiers les plus petits (en taille).

```
char car = 'a' ; //code ascci 97
car = car + 1 ; // 97 + 1 = 98
printf("%c",car) ;
```

Le type *short* permet des représentations plus petites en mémoire (en général de moitié) ; à l'opposé, le type *long* permet des représentations plus grandes (en général double). Relativement à leur taille on a :

unsigned char <= short unsigned <= unsigned <= long unsigned <= long long unsigned
char <= short int <= int <= long int <= long long int

NB : Par défaut, le type short (respectivement long) correspond au short int (respectivement long int)

short <= int <= long

Pour les réels, le C propose deux types *float* et *double* (en général les doubles occupent 2 fois plus de mémoire).

float <= double <= long double

Les tailles de ces entiers ou réels sont dépendantes de la machine cible. Voici les tailles mémoires en octet de ces différents types sur les Sparc Stations Sun, IBM rs6000, Intel pentium (linux) :

type	Sparc	PowerPc	Pentium
char	1	1	1
short int	2	2	2
int	4	4	4
long int	4	4	4
long long int	8	8	8
unsigned char	1	1	1
short unsigned	2	2	2
unsigned	4	4	4
long unsigned	4	4	4
long long unsigned	8	8	8
float	4	4	4
double	8	8	8
long double	12	12	12

```
unsigned char 1 (0 - 255)
short unsigned 2 (0 - 65535)
unsigned 4 (0  - 4 294 967 295)
long unsigned  4 (0  - 4.294 10(9))
long long unsigned 8 ( 0 - 1.844 10(19))
```

NB : La "fonction" **sizeof(type)** *retourne la taille réelle utilisée en mémoire pour un type.*

Initialisation des variables :

```
int main(void) {
        printf("taille d'un float %d\n",sizeof(float)) ;
}
/*affiche "taille d'un float 8" sur un pentium 32 bit*/
```

Le C permet d'initialiser directement des variables par des valeurs exprimées en octal ou en héxa-décimal. Si la constante débute par un 0 (respectivement 0x), elle est exprimée en octal (respectivement en hexadécimal).

Exemples :

```
unsigned char u = 0x11 ; /* déclare un entier non-signé sur 1 octet initialisé avec la
valeur 11 exprimée en héxadécimal (cad 17(10)) */
int i = 036 ; /* déclare un entier initialisé avec la valeur 36 en octal (cad 30(10)) */
long unsigned ul = 134 /* déclare un entier non-signé long initialisé avec 134(10)) */
```

NB : une variable caractère peut ainsi être directement initialisée avec une valeur ASCII

Les formats d'entrées/sorties *scanf* et *printf* :

Le format pour imprimer *printf* ou saisir *scanf* les variables peut contenir les séquences suivantes :
%d type entier
%u type entier non-signé
%o type entier converti en octal
%x type entier converti en héxadécimal
%c type caractère
%s type chaîne de caractères
%f type réel (format fixe avec un point décimal) *(à éviter)*
%lf type réel (format fixe avec un point décimal)
%e type réel (format exposant)
Pour les formats %o, %u, %x, %d, %f on peut y ajouter l (pour long) oui h (pour sHort) -> %lo, %lu, %lx, %ld, %lf, %ho, %hu, %hx, %hd, ...
Une taille peut être précisée dans les formats (ex %10.2f). Les nombres sont cadrés à droite et les chaînes de caractères à gauche. Le moins permet d'inverser le cadrage (ex "%- 10s" est un format pour une chaîne de 10 caractères cadrée à droite).
Le format peut être paramétré. Exemple :

```
int f=10,val=104;
printf(''%*d'',f,val);
```

Ici l'étoile est remplacée par le contenu de la variable f donc "%10d".
Les saisies correspondantes au *scanf* utilisent le flot d'entrée (suite de caractères). Les valeurs sont séparées par des blancs, tabulations, ou retour chariots ; sauf pour les caractères (format %c) puisque les blancs, tabulations, ou retour chariots sont aussi des caractères.
Il est possible absorder ces caratères en précédant le format d'un blanc. Par exemple **scanf(" %c",&car) ;** saisit un caractère autre qu'un blanc et ne prend pas le retour chariot correspondant à la saisie précédente.

Les opérateurs (~, | , &, ^)

Un ensemble d'opérateurs permet de travailler directement sur la représentation binaire des variables. Dans les exemples suivants, nous considérerons une représentation binaire sur 1 octet.

```
~                       /* complément binaire */
&                       /* et binaire */
|                       /* ou binaire */
^                       /* ou exclusif binaire */
```

Le complément binaire (~)

L'opérateur ~ effectue le **complément binaire** d'une valeur entière. Les 0 sont transformés en 1 et les 1 en 0.

exemple : (unsigned) 19 -> 00010011
~(unsigned) 19 -> 11101100

NB : Cet opérateur est souvent utilisé pour construire des masques.

Le et binaire (&)

L'opérateur & effectue un **et binaire** entre deux entiers :

```
exemple :       (26)        00011010
                (23)    &   00010111
                            =======
                            00010010
```

Le ou binaire (|)

L'opérateur | effectue un **ou binaire** entre deux entiers :

```
exemple :       (18)        00010010
                (23)    |   00010111
                            =======
                            00010111
```

Le ou exclusif binaire (^)

L'opérateur ^ effectue un **ou exclusif binaire** entre deux entiers :

```
exemple :       (18)        00010010
                (23)    ^   00010111
                            =======
                            00000101
```

Les opérateurs de décalage (<<, >>)

```
>>                      /* décalage à droite */
<<                      /* décalage à gauche */
```

Le décalage à droite (>>)

L'opérateur >> effectue un décalage à droite des bits d'un entier. Les bits à l'extrême droite sont perdus et des 0 complètent la partie gauche. x>>n signifie n décalages de bit sur l'entier x.

```
exemple :       (unsigned)(19)      00010011 >> 2
                                    ___________
                                    00000100
```

Le décalage à gauche (<<)

L'opérateur << effectue un décalage à gauche des bits d'un entier. Les bits à l'extrême gauche sont perdus et des 0 complètent la partie droite.

exemple : (unsigned)(19) 00010011 << 2

01001100

3 Les tableaux et les fonctions

Les tableaux (int[ ], float[ ], char[ ])

Les tableaux en C sont une suite de cases mémoire contigües.

Voici la première approche : **les tableaux constants** "à la Pascal" :

La taille d'un tableau est définie à la déclaration de celui-ci. La première case du tableau est **toujours** la case d'indice 0. Donc, pour un tableau de n éléments, les indices vont de 0 à n-1.

```
int  t[10] ;  /* déclaration d'un tableau t de 10 cases de type entier */
/* Le premier élement est accessible par t[0] et le dernier par t[9] */
int t[3][10] ; /* déclaration d'un tableau à deux dimensions */
/* l'accès aux eléments s'effectue par t[i][j] */
/* Le premier élement est accessible par t[0] et le dernier par t[9] */
int t[3][10][5] ; /* déclaration d'un tableau à trois dimensions */
int t[ ] = {0,1,2,3,4,5,6,7,8,9} ; /*déclaration et initilisation d'un tableau de 10 entiers */
int t[5] = {0,1,2} ;  /* déclaration d'un tableau de 5 éléments dont les 3 premiers sont
initialisés*/
float a[50] ; /* déclaration d'un tableau a contenant 50 réels */
char ch[10] ; /* déclaration d'un tableau de 10 caractères */
int t[2][3] = {{1,2,3},{4,5,6}} ;
```

Accés aux élements d'un tableau

```
int main(void) {
  int tab[2] = {1,2};
  printf("%d %d\n",tab[0],tab[1]);
  tab[0] =  tab[1] + 1; // 3
}
```

NB : Attention, en C le compilateur ne vérifie pas les indices du tableau. Dans un tableau de n cases, vous pouvez accéder à la case d'indice >= à n (ou < 0). Vous allez alors accéder à une zone mémoire en dehors du tableau (souvent une autre variable).

La boucle (for)

La conception de la boucle for est proche en C du principe de la boucle *while*. Elle est surtout utilisée pour un confort d'écriture.

Syntaxe :

```
for( partie_initialisation ; expression_de_continuation ; action_en_fin_d'itération )
    instruction ;

for( partie_initialisation ; expression_de_continuation ; action_en_fin_d'itération ) {
    instruction1 ;
    instruction2 ;
    ...
    instructionN ;
} ;
/* le for précédent correspond exactement au while suivant */
partie_initialisation ;
while (expression_de_continuation) {
    instruction1 ;
    instruction2 ;
    ...
    instructionN ;
    action_en_fin_d'itération
} ;
```

Les instructions *partie_initialisation* et *action_en_fin_d'itération* sont optionnelles. Le *for* peut être réduit comme suit : **for(; ;)**

NB : Si l'expression de continuation est absente, la valeur retounée est toujours vraie.

exemples de *for* :

```
int i ;
for( i = 0 ; i <= 10 ; i = i+1 )
     instruction ;
/*pour étendre la partie_initialisation et action_de_fin_d'itération, on utilise la séquence " , " */
for( i = 0, j = 100 ; (i <= 10) && (j >= 0) ; i +=1, j-= 1 )
    instructions ;

for( scanf("%d",&i) ; i ; printf("%d",i), i-=1) ; /* pas de corps dans le for */
```

Les blocs ({,})

Une séquence d'instructions peut constituer un bloc. Les blocs sont encadrés par les accolades "{", "}" et peuvent contenir des déclarations de variables locales. Ils sont utilisés principalement dans les structures de contrôle (*if, while* ,...) mais peuvent aussi être n'importe où dans le programme.

Exemple :

```
int main(void) {
    int x=1,y=2 ;
    ...
    {           int tmp ; /*variable temporaire pour l'échange des variables x et y */
                tmp = x ;
                x = y ;
                x = tmp ;
    }
    ...
}
```

Les fonctions ()

Les fonctions C permettent le découpage d'un programme en "sous programmes". Il n'y a pas de procédure dans le langage ; mais une fonction peut ne pas retourner de résultat (le type du résultat est alors déclaré *void*).

Les paramètres sont déclarés entre les parenthèses suivant le nom de la fonction. Chaque nom de paramètre est précédé par son type (norme ANSI). Les différents paramètres sont séparés par des virgules. Si la fonction n'a pas de paramètre, il faut mettre le mot clé *void* entre les parenthèses de la fonction.

```
type_du_résultat nom_fonction( déclaration_des_paramètres_formels ) {
    déclaration_des_variables_locales
    corps_de_la_fonction
    return résultat ;
} ;
```

Le mot clé *return* permet de renvoyer un résultat (expression). La fonction est alors "abandonnée" même si il reste des instructions après le *return*.

Appel de la fonction

Exemple :

```
variable =nom_fonction(liste_des_paramètres_effectifs) ;
```

Les paramètres effectifs de la liste sont séparés par des virgules et chaque paramètre est une expression.

NB : même si la fonction n'a pas de paramètre, il faut mettre des parenthèses, celles-ci préviennent le compilateur de l'appel de fonction. (Sinon c'est l'adresse de la fonction qui est manipulée comme un objet)

Voici un exemple complet utilisant les fonctions C.

```
#include<stdio.h>
int fac(int n) { /* fonction factorielle
/* calcul it */
    int cumul =1 ;
    for ( ;n > 1 ; n -=1) cumul = cumul * n ;
    return cumul ;
}

int arrangement (int m, int p) {  /* arrangement m p */
    return fac(m) / fac(m-p) ;
}

int main(void ) { /* fonction principale */
    int m,p ;
    printf ("Arrangement m p \n" ) ;
    printf ("saisir m et p ") ;
    scanf("%d%d",&m,&p) ;
    if (m > p)
                printf("\nArrangement m%d p%d = %d \n",m,p,arrangement(m,p)) ;
}
```

La taille du tableau dans la déclaration d'un paramètre tableau n'est pas nécessaire. Dans une fonction ayant un tableau en paramètre, la taille est souvent passée dans un autre paramètre :

Exemple :

```
*

/* la fonction retoune la moyenne les valeurs du tableau */
float moyenne(int tab[ ] , int n) {
    int cumul = 0, i ;
    for (i = 0 ; i < n ; i+=1)
                cumul += tab[i] ;
    return cumul / (float) n ;
}
/* exemple d'utilisation */
int main(void){
    int t1={1,7,6} ;
    int t2={3,5,7,13,5,7} ;
    printf("moyenne du tableau T1 %f \n",moyenne(t1,3)) ;
    printf("moyenne du tableau T2 %f \n",moyenne(t2,6)) ;
}
```

Pour un paramètre, tableau à N dimensions (avec N > 1), il faut fixer N-1 dernières dimensions.

Les fonctions des librairies

Les fonctions préexistantes en C comme le *printf* et *scanf* sont compilées dans une librairie spécifique de nom *libC.a ou libC.so*. Cette librairie est automatiquement liée lors de l'édition de liens. Il est un ensemble d'autres librairies C contenant par exemple les fonctions mathématiques (cos,sin,tan,power,exp, ..). Ces librairies doivent être explicitement liée lors de lédition de liens.

Exemple la fonction cosinus (cos) est dans la librairie *libm.a*.

La compilation et lédition de liens séffectue par la commande :

```
gcc -o main main.c -lm
```

L'option l précise de lier la librairie m (implicitement libm.a).

Les macro-instructions (#define)

Les macro-instructions sont des instructions pour le préprocesseur C. Par convention, les macros sont désignées par des noms écrits en majuscules.

Exemple :

```
#define MAX 100

int main(void){
    int tab[MAX];
    ...
}
```

Dans ce cas la macro-instruction définit une "constante". Mais une macro peut contenir tout autre source C.

Exemple :

```
#define INIT        i = 0;\
                    j = 0;\
                    printf("init\n")

int main(void){
    INIT;
    ...
}
```

NB : Si une macro est définie sur plus d'une ligne, chaque ligne doit se terminer par le caractère "\".

Une macro-instruction peut contenir des arguments

Exemple :

```
#define INC(X, N)          (X)+= (N)
#define VALCAR(C)            ((C) - '0')

int main(void){
    int i = 1;
    char c = '1';
    INC(i,10);
    printf("%d\n",i); /* 11 */
    printf(" caractere %c%d%d \n",c,c,VALCAR(c)); /* 1 49 1 */
}
```

Les macro-instructions sont traitées par substitution de chaîne de caractères dans un fichier. A chaque rencontre d'une macro, le préprocesseur remplace celle-ci par le source correspondant, ceci avant d'effectuer la compilation proprement dite.

Une macro peut appeler une seconde macro , mais l'appel récursif est interdit, (le préprocesseur étendrait la chaîne à l'infini).

Les directives de compilation

```
// pr.c
int main(void){
#ifdef TRACE
      printf("main\n")
#endif
      ...
}
```

La partie de code entre le ifdef et endif est compilée que si TRACE est défini.

```
cc -o pr -DTRACE pr.c  // le code entre les marqueurs est compile

cc -o pr  pr.c  // le code entre les marqueurs n'est pas compile
```

Autre exemple : pour éviter une multiple inclusion de fichier .h

```
// fichier interface.h
#ifndef INTERFACE
#define INTERFACE


..

#endif
```

4 Compléments

Les constantes ANSI

Les constantes permettent de nommer une valeur (par ex pi = 3.14). La valeur ne pourra plus être modifiée.

```
const int max = 100;
const float pi = 3.14156 ;
const char space = ' '; // espace
..
int tab[max];  // tableau dans la dimension
               // est une constante
```

L'affectation (=)

Toutes les instructions C retournent une valeur. L'affectation a = 10 ; retourne la valeur de a (cad10). Cette propriété permet de mettre en cascade les affectations.

Exemples :

```
i = j = k = 0 ;          /* i = ( j = ( k = 0 )) ; */
i = 10 + (j = 10) ;      /* i = (1 + (j = 10)) ; */
```

Les affectations peuvent être associées avec les opérateurs simples. Les résultats de l'expression utilisant une variable est réaffectée à cette variable.

Exemple :

```
i += j ;                 /* i = i + j ; */
```

Ce type d'affectation est valable pour les cas suivants :

```
+=, -=, /=, *=, %=
|=, &=, ^=
>>=, <<=
```

L'incrémentation (++, - -)

L'opérateur ++ (respectivement - -) permet d'incrémenter (respectivement de décrémenter) une variable.

Exemple

```
i ++ ;                   /* i +=1 ; */
++ i ;                   /* i +=1 ; */
```

Le placement du ++ par rapport au nom de la variable détermine le moment ou l'incrémentation est effectuée. Si le ++ est placé après la variable, l'expression contenant le ++ est évaluée avant l'incrémentation. Et inversement si le ++ est placé avant la variable, l'incrémentation s'effectue avant l'évaluation de l'expression.

Exemple :

```
i = 0 ;
if ( ! ++i )
      instruction ;
```

```
i = 0 ;
if ( ! i++ )
      instruction ;
```

Dans le premier cas, l'instruction n'est pas exécutée puisque l'incrémentation s'effectue avant l'évaluation de i. Dans le second cas l'instruction est exécutée. Dans les deux cas i vaut 1 à la sortie de l'instruction *if* .

NB : Le compilateur optimise alors le code généré en chargeant une seule fois la variable.

L'instruction itérative (do), boucle répéter

syntaxe :

```
do
    instruction ;
while ( expression ) ;
```

```
do {
    instruction1 ;
    instruction2 ;
    ...
    instructionN ;
} while ( expression ) ;
```

Le corps de la boucle est exécuté avant l'évaluation de la condition de continuation. Le programme passe donc au moins une fois dans le corps de la boucle (contrairement au *while*).

L'instruction conditionnelle à cas multiple (switch)

syntaxe :

```
switch ( expression ) {
    case const1 :
                    instruction1 ;
                    ...
                    instructionN ;
                    break ;
    case const2 :
                    ...
                    break ;
    default :
                    ...
}
```

En fonction de la valeur retournée par l'expression, le programme se branche sur l'une des séquences d'instructions. Le cas *default* (autre cas) est optionnel. L'instruction *break* permet de sortir du *switch*. Si il n'est pas mis le programme exécute les instructions de la branche *case* suivante. Cette possibilité permet d'exécuter la même séquence d'instructions pour plusieurs constantes.

Exemple :

```
char c ;
scanf(%c,&c) ;
switch ( c ) {
    case 'a' :
    case 'A' :
                    instruction1 ;
                    ...
                    instructionN ;
                    break ;
    case 'b' :
    case 'B' :
                    instruction1 ;
                    ...
                    instructionN ;
                    break ;
    ...
```

L'opérateur conditionnel (? :)

Le langage C propose un opérateur conditionnel. Dans une expression, on peut donc effectuer des tests.

syntaxe :

```
expression ?
    expression1 :
    expression2
```

Si l'expression est vraie l'expression1 est évaluée (exécutée) sinon le programme évalue l'expression2.

Exemples :

```
i > j ? 0 : -1 ; /* si i>j l'expression retourne 0 sinon -1 */
k = i > j ? 0 : -1 ; /* le résultat de l'expression est affectée à k */
if (i > j ? 0 : -1 ) k ++ ; /* l'expression contrôle le test */
i > j ? /*opérateurs ? : imbriqués */
    k > l ?
        0
        : -1
    : 1 ;
```

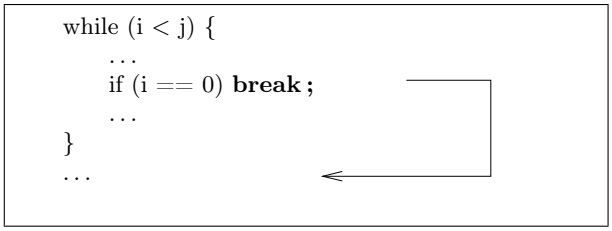
Plusieurs instructions peuvent être insérées dans l'opérateur en utilisant la séquence ",". La séquence doit se terminer par une expression (pas de " ;").

Le break, continue et exit

L'instruction break

L'instruction *break* permet de sortir d'une structure de contrôle sans exécuter le reste des instructions (voir l'instruction *switch*). Il peut être utilisé dans les *while, do, for* et *switch*. Dans une boucle, le *break* sort de cette boucle (sans évaluer l'expression de contrôle de la boucle).

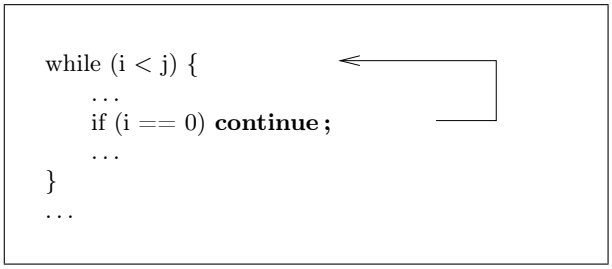
Exemple :



L'instruction continue

L'instruction *continue* permet dans une structure de contrôle (*while, do, for*) de sortir d'une étape de la boucle et de revenir sur le test de contrôle de la boucle.

Exemple



L'instruction exit

l'instruction *exit* provoque la fin immédiate du programme quelleque soit la position de l'exit dans ce programme. La valeur donnée en paramètre de l' *exit* peut être récupérée par l'utilisateur du programme (sous Unix (en Csh pour la variable locale $status)) (en Sh par la variable $?)) ;

Les instructions *break*, *continue* et *exit* sont à utiliser avec précaution. Elles vont à l'encontre de la programmation structurée. Elles sont souvent utilisées dans une programmation "système" pour des raisons d'efficacités.

5 Les pointeurs et tableaux

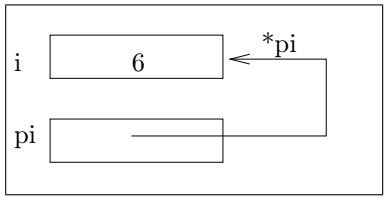
Notion de pointeur

Les pointeurs en C sont des adresses de cases mémoire. Ils permettent entre autres le passage par adresse des paramètres d'une fonction.

La déclaration d'un pointeur utilise le symbole "*****". Exemple : *int * pi*; déclare un pointeur sur une variable de type entier. L'adresse d'une variable est le nom de la variable précédée par le symbole " **&**" et le contenu de cette variable pointée est le nom de la variable pointeur précédé par le symbole "*****".

```
int i = 5 ;                          /*déclaration d'une variable entière */
int *pi ;                            /* déclaration d'un pointeur d'entier */
pi = &i ;                            /* l'adresse de i est affectée au pointeur pi */
*pi = 6 ;                            /* modification de la variable pointée par pi */
printf("%d %d\n", *pi, i) ;          /*affiche 2 fois le contenu de i (cad 6) */
```

Les variables *i* et *pi* peuvent se schématiser comme suit :



Les paramètres d'une fonction

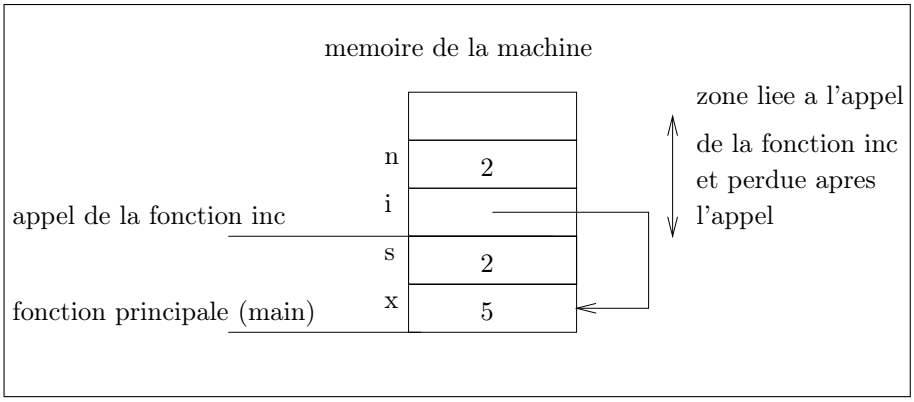
Les paramètres d'une fonction sont passés par valeur. Pour modifier un paramètre, il faudra donc utiliser l'adresse de la variable comme paramètre : "l'adresse de la variable sera alors passée par valeur".

Dans la fonction, le paramètre correspondant à une variable "par adresse" est déclaré comme un pointeur et il faudra dans le corps de la fonction dépointer le paramètre pour accéder au contenu de la variable pointée.

Exemples : La fonction *inc* incrémente de *n* une variable entière passée en paramètre.

```
void inc(int *i, int n) { /* i pointeur sur la variable à incrémenter */
     *i += n ; /*accès au contenu de i */
}

int main(void) { /*exemple d'utilisation de la fonction */
     int x= 5, s = 2 ;
     inc(&x, s) ; /* passage de l'adresse de i */
     printf("%d\n",x) ;
}
```

schématisation de l'appel de inc :



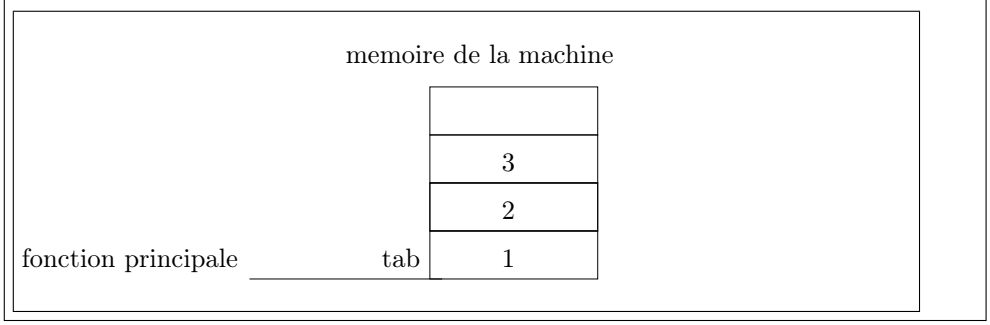
Lors de l'appel de la fonction *inc*, l'adresse de la variable *x* est affectée au paramètre de nom *i* et la valeur contenue dans la variable *s* est recopiée dans le paramètre *n*. Dans la fonction *inc*, la case dépointée par la variable *i* est le contenu de variable *x* de la fonction principale ; par contre l'accès à la variable *n* utilise la copie de *s*.

A la fin cet appel, la région de pile (*i* et *n* après la variable *s de la fonction principale)* correspondant à l'appel n'est plus accessible.

Les tableaux

Nous avons vu précédemment que les tableaux "constants" étaient une suite de cases mémoire contigues. Une variable de type tableau (*int tab[3]*) est en réalité une constante indiquant l'adresse de la première case du tableau. La variable *tab* ne pourra jamais être modifiée :

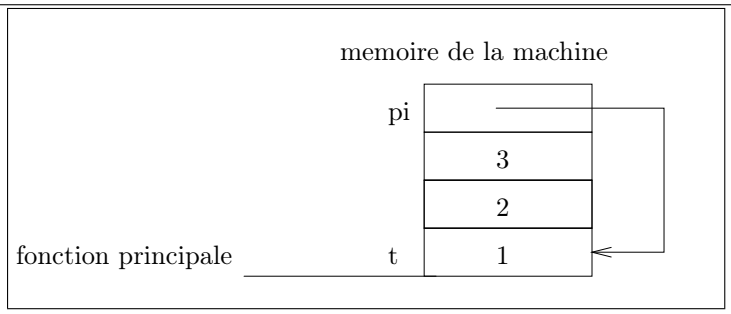
```
int main(void) {
    int tab[]={1,2,3} ;
    printf("tab = %x &t[0] = %x\n",tab,&tab[0]) ; /*affiche 2 fois la même "adresse" */
}
```



NB : Lorsque l'on passe un tableau constant en paramètre d'une fonction, l'adresse de la première case est donc copiée. La fonction peut alors modifier directement les valeurs contenues dans le tableau. Les tableaux sont donc passés par adresse.

On peut alors déclarer des variables de type pointeur que l'on pourra initialiser avec l'adresse d'un tableau.

```
int t[ ] = {1,2,3} ;          /*déclaration d'un tableau de 3 entiers  */
int *pi ;                     /* déclaration d'un pointeur d'entier */
pi = t ;                      /* l'adresse de i est affectée au pointeur pi */
printf("%d\n",*pi) ;          /*affiche le contenu de tab[0] (cad 1) */
```



Calcul d'adresse sur les pointeurs.

Le symbole "**&**" retourne l'adresse d'une variable et le symbole "*****" permet accéder au contenu d'une variable pointée.

Exemple : l'expression ***&i** équivaut à *i*.

On peut aussi effectuer des calculs sur des adresses mémoire

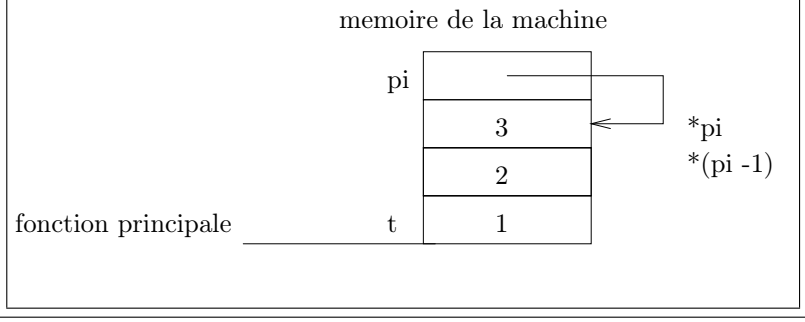
Exemple : l'expression ***(&i + 1)** retourne le contenu de la case dont l'adresse est : l'adresse de *i* incrémentée de la taille d'un élément du type pointé (cad suivant la variable **i**).

Les additions (ou éventuellement soustractions) d'adresse se font toujours par rapport à la taille du type de case pointée.

NB : Ici le calcul d'adresse est à première vue dangereux, mais si la variable pi pointe sur une case d'un tableau, l'expression ***(pi + 1)** *accède à la case suivante du tableau.*

Autre exemple : l'instruction *pi += 2* déplace le pointeur de deux cases.

```
int t[ ] = {1,2,3} ;          /*déclaration d'un tableau de 3 entiers  */
int *pi ;                     /* déclaration d'un pointeur d'entier */
pi = t ;                      /* l'adresse de t est affectée au pointeur pi */
pi += 2 ;                     /* déplace le pointeur de deux cases */
printf("%d\n", * (pi -1))     /*affiche le contenu de la case précédente */
```



Les chaînes de caractères.

Les chaînes de caractères peuvent en C être des tableaux constants.

Exemple :

char ch[] = {'a', 'b', 'c','\n'} ;

ou

char ch[] = "abc";

Mais les chaînes sont le plus souvent définies comme des pointeurs sur des zones mémoires contenant des caractères. Une chaîne se définit alors comme suit :

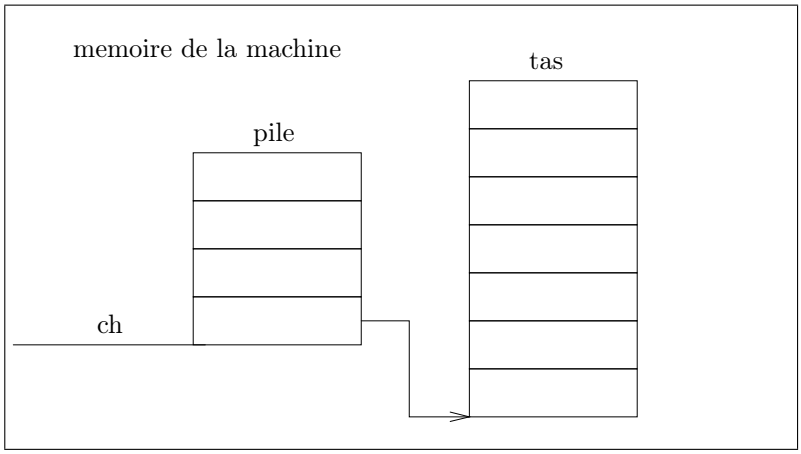
*char *ch ;*

Deux fonctions (*malloc* et *free*) de la librairie C permettent d'allouer et de désallouer un bloc mémoire. L'espace est pris sur le tas (espace mémoire du processus). Lea fonction retourne NULL en cas d'echec.

Exemple :

*char *ch = (char *)malloc(10) ;*

définit une chaîne de caractères, alloue une zone mémoire pouvant contenir 10 caractères et affecte l'adresse de cette zone à la variable *ch* :



La fonction *free()* libère l'espace mémoire alloué par un *malloc()*. Ces deux fonctions doivent être utilisées après avoir inclus le fichier <malloc.h>.

Plusieurs autres fonctions de la librairie standard permettent directement de travailler sur des chaînes de caractères : (utilisables par inclusion du fichier <string.h>.

Exemples :

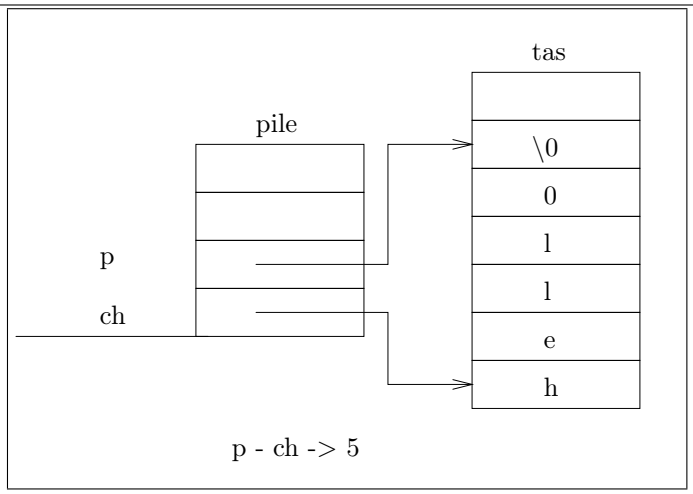
- char * strcpy(char *ch1,char *ch2) copie la chaîne ch2 dans ch1 et retourne la chaine ch1.
- int strlen(char *ch) retourne le nombre de caractères contenus dans ch.
- int strcmp(char *ch1,char *ch2) compare les deux chaînes ch1 et ch2.
- char * strcat(char *ch1,char *ch2) concatène la chaîne ch2 en bout de la chaîne ch1 et retourne la chaine ch1.

Par convention une chaîne de caractères se termine par le caractère nul (**0**). Ce caractère permet aux fonctions standards de détecter la fin de la chaîne.

Exemple de fonction : *strlen()* :

NB : L'opérateur ++ appliqué sur une variable pointeur incrémente l'adresse contenue dans cette variable de la taille d'un objet du type pointé.

```
int strlen(char * ch ) { /* retourne la taille de la chaine ch */
      char *p = ch ; /*déclare un 2ème pointeur inialisé sur la même chaine */
      while (*p) p ++ ; /*test sur le caractère nul \0 */
      return (p - ch) /*retourne la différence entre les deux adresses */
} /*ici sizeof(char) =1 */
```



Les arguments de la ligne de commande

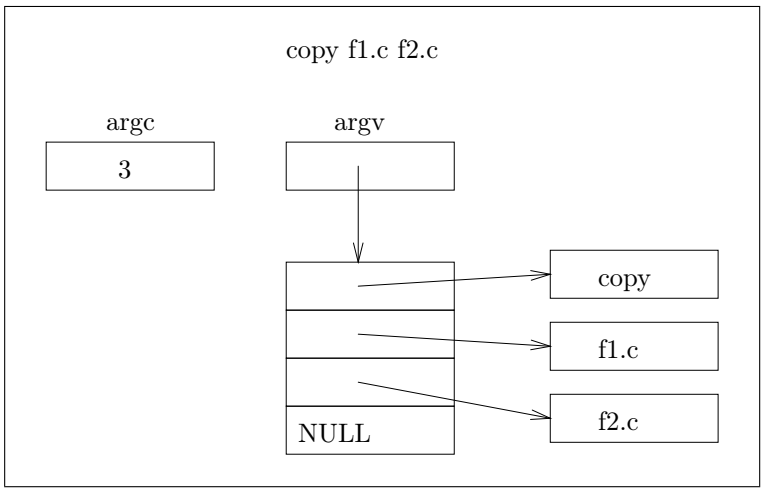
Le langage C permet de récupérer les arguments passés lors du lancement d'un programme. Cette fonctionnalité permet entre autres, d'écrire des commandes systèmes.

syntaxe :

```
int main (int argc, char **argv, char **env)
```

Les arguments sont les paramètres de la fonction principale *main*. *argc* contient le nombre d'arguments (y compris le nom du programme) et *argv* est un tableau de chaînes de caractères. Tous les arguments sont en format chaîne de caractères. Le nom du programme (nom de l'exécutable) est la première chaine du tableau. Le paramètre *env* convient les informations sur environnement du processus.

Exemple avec un exécutable de nom *copy* :



exemple d'utilisation :

```
int main (int argc, char **argv)  {
     /*ce programme affiche les arguments d'appel */
     int i ;
     for (i=0, i<argc ; i++)
          printf("%s\n", *argv++) ;
}
```

Les variables d'environnement peuvent être plus simplement est accedées par la fonction

`char * getenv(char * uneVariable);`

qui retourne la valeur de la variable de nom uneVariable.

6 Les structures et la définition de type

Déclaration des structures (struct)

Les structures en C comme dans les autres langages, permettent de regrouper plusieurs informations pouvant être de types différents. L'entité regroupant ces informations est nommable dans le langage.

syntaxe :

```
struct [ nom_de_la_structure ] {
    type champ1 ;
    type champ2 ;
    ...
    type champN ;
} [ liste_de_variables ] ;
```

Le nom de la structure est optionnel. Il n'est utile que si l'utilisateur veut par la suite déclarer d'autres variables ayant la même structure. La liste des variables est aussi optionnelle, les variables pouvant être déclarées par la suite et de la façon suivante :

```
struct nom_de_la_structure liste_de_variables ;
```

Voici un exemple de structure : une fiche *client* comprend le nom, le prénom et le solde d'un client.

Voici sa déclaration :

```
struct s_client {
    char        nom[25] ;
    char        prenom[15] ;
    float       solde ;
} client, t_client[10] ;
```

Dans cet exemple, nous avons défini une variable (*client*) de type structure *s-client* ainsi qu'un tableau *t_client* de 10 structures *s_client*.

Une variable structure peut être initialisée comme suit :

```
struct s_client cl =
    { "courtrai","luc",0.} ;
```

L'accès aux différents champs d'une structure via le nom de la variable s'exprime par le symbole ".".

Exemple :

```
void affiche_client(struct s_client cl){
    printf("nom                 :%25s\n",cl. nom) ;
    printf("prenom              :%15s\n",cl. prenom) ;
    printf("solde               :%10.2f\n",cl.solde) ;
} ;
```

L'accès aux champs via une variable de type pointeur utilise le symbole (->).

Exemple :

```
struct s_client t_client[10] , *p_client ;
p_client = &t_client[2] ;          /*positionne le pointeur sur le 3ème client */
p_client->solde = 0 ;              /* accès au champ <=>  (*p_client).solde */
```

Exemple de structure “récursive” :

```
struct s_node {
    float valeur ;                 /* information contenue dans le noeud */
    struct s_node *fd,*fg ;        /* lien sur le fils droit et fils gauche  */
} ;
```

L'allocation dynamique d'un bloc mémoire pouvant contenir ce type de structure se fait par la fonction *malloc*. Le type retourné par la fonction est un *int **, il faut donc forcer le type (cast) du résultat pour récupérer un pointeur de structure. L'argument passé à la fonction *malloc* est un nombre d'octets correspondant au nombre de caractères voulus.

Voici un exemple d'allocation :

```
struct s_client *p_client ;
p_client =(struct s_client *)malloc(sizeof(struct s_client)) ;
```

Structure = champs de bits

Une struture C peut être définie comme une suites de bits. Les champs peuvent alors être plus petits que les types de bases. Le compilateur n'effecte pas d'alignement sur les mots mémoire.

Voici l'exemple d'une structure de la taille d'un octet découpée en deux champs.

```
int main(void) {
  struct Stoto {
     unsigned char part1:3; // le trois pemiers bits
     unsigned char part2:5; // les cinq bits suivants
  } toto;

  toto.part1 = 7; // 7 -> trois bits
  toto.part2 = 16; // 16 -> quatre bits

  printf("%u \n",sizeof(struct Stoto)); // 1
}
```

Le compilateur doit garantir les limites des champs lors des affectations.

Déclaration d'un type (typedef)

Le programmeur d'une application C peut définir de nouveaux types. Cela permet entre autres de faciliter l'écriture d'un programme et d'uniformiser les déclarations des variables (par exemple pour éviter le mot clé *struct* dans la déclaration des variables structures).

syntaxe :

```
typedef                    type
nom_du_type ;
```

exemple : voici comment redéfinir les types de base en français :

```
typedef int entier;
typedef float reel;
typedef char caractere;

entier e; reel r; caractere
c;
```

La définition de type est bien sûr plus utile pour les tableaux, pointeurs, structures et même les unions.

Exemple :

```
typedef struct {
    char nom[25];
    char prenom[15];
    float solde;
} client;

typedef client clients10[10];

client cl;
clients10 t_cl;
```

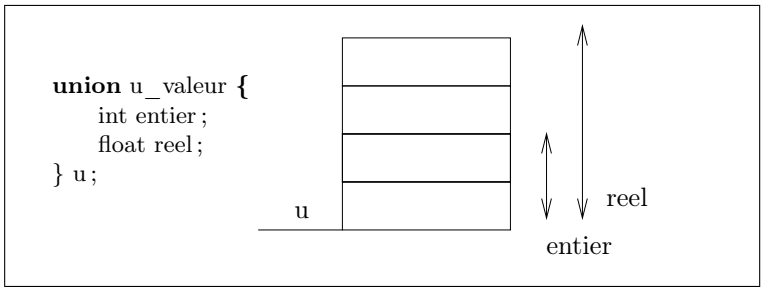
Les unions

Les unions en C permettent d'utiliser le même espace mémoire pour stocker des variables peuvant être de type différent. Le programmeur doit gérer explicitement le type de l'information contenue dans l'union.

syntaxe :

```
union nom_de_l_union {
    type champ1;
    type champ2;
    ...
    type champN;
} liste_de_variables;
```

Voici un exemple d'union : un espace *u_ valeur* peut contenir soit un entier soit un réel :



L'accès aux différents champs de l'union via le nom de la variable s'exprime par le symbole "**.** ".

Exemple :

```
union u_valeur u ;
scanf("%f",&u.reel) ;
u.entier= 1 ; /* la valeur réelle est écrasée */
printf("%d", u.entier ) ;
```

Les unions sont le plus souvent associées à une structure qui encapsule un indicateur et une union, le tag permettant de distinguer de champ de l'union.

Exemple :

```
struct {
    int tag ; /* indicateur */
    union {
        int entier ;
        float reel ;
    } valeur ;
} v ;
...
if (v. tag)
    printf("%d",v.valeur.entier) ;
else
    printf("%f",v.valeur.reel) ;
```

7 Les fichiers bufferisés

La gestion des fichiers en C fait partie d'une bibliothèque de fonctions d'entrées/sortie (librairie standard). Nous verrons ici les fichiers bufferisés (appelés *stream*) où, à chaque fichier est associée une zone tampon en mémoire. Ces fichiers sont gérés caractère par caractère par le système. Le fichier <stdio.h> doit être inclus. Nous présentons ici les principales fonctions.

Trois *stream* sont prédéfinis (stdin (l'entrée standard), stdout (la sortie standard), stderr (la sortie d'erreur standard))

Déclaration d'un fichier (FILE *)

Chaque fichier est associé à un descripteur logique. Le descripteur (structure **FILE**) contient différentes informations "système", dont un lien vers la zone tampon. La déclaration d'un fichier s'effectue en définissant un pointeur de FILE.

```
FILE * fic ;
```

Ouverture et fermeture de fichier (fopen, fclose, fflush)

L' ouverture d'un fichier (fopen)

L'ouverture d'un fichier associe un fichier physique à un descripteur logique interne.
syntaxe :

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

Le paramètre *nom* est le nom externe (physique) du fichier. Il peut contenir un chemin d'accès au fichier.

Le paramètre *mode* définit le mode d'ouverture du fichier. Il peut prendre les valeurs suivantes :

```
"w"     /* écriture le fichier est créé ou écrasé */
"r"     /* lecture  */
"a"     /* ajout (écriture), positionne en fin de fichier */

"w+"    /* idem w mais possibilité de lecture */
"r+"    /* idem r mais possibilité d'écriture */
"a+"    /* idem a mais possibilité de lecture */
```

En mode *w*, si le fichier existe, il est détruit puis ouvert en mode écriture. En cas de problème d'ouverture, (par exemple si le fichier n'existe pas en mode *r*), la fonction *fopen* retourne NULL.

Exemple : ouverture d'un fichier de donnée existant en lecture/écriture :

```
FILE * fic ;
if ((fic = fopen("/home/dupont/fichier.data","r+" ) == NULL)
    printf("probleme d'ouverture\n") ;
else {
    ...
```

NB : Sous Msdos où les chemins d'accès sont décrits avec des "\", caractères spéciaux en C, les "\" doivent être doublés "\\".

Les fichiers *stdin, stdout, stderr* sont prédéfinis comme descripteurs sur respectivement l'entrée standard, la sortie standard et la sortie d'erreur. Ces descripteurs sont toujours ouverts.

La fermeture d'un fichier (fclose).

L'instruction fclose a pour effet de vider le tampon, fermer le fichier et libérer le descripteur. Syntaxe :

int **fclose**(FILE * f)

La fonction retourne 0 si la fermeture a fonctionné correctement (-1 sinon).

L'instruction(fflush)

L'instruction *fflush* permet de vider le tampon sans attendre la fermeture du fichier.

int **fflush**(FILE * f)

Lecture, écriture dans un fichier

Pour toutes les fonctions de lecture ou d'écriture, la position dans le fichier après l'appel est automatiquement modifiée.

mode caractère (fgetc, fputc)

La fonction *fgetc* lit un caractère dans un fichier ouvert :

int **fgetc**(FILE * f)

En fin de fichier le caractère retourné est EOF.

*NB : La fonction getc(FILE*f) est plus souvent utilisée. Sa fonctionnalité est identique. La fonction (macro-instruction) getchar correspond à getc(stdin)*

La fonction *fputc()* écrit un caractère dans un fichier.

int **fputc**(char c, FILE * f)

La fonction retourne le caractère écrit, en cas de problème, la valeur retournée est négative.

*NB : La fonction (macro-instruction) putc(char c, FILE *f) a les mêmes fonctionnalités. La macro-instruction putchar(char c) correspond à putc(c,stdout).*

La fonction *ungetc(char c,FILE *f)* permet de remettre un caractère dans le tampon d'un fichier. Il sera lu à la prochaine lecture.

Exemple : voici une commande qui affiche le contenu d'un fichier dont le nom est passée en argument de la commande

```
#include<stdio.h>
int main(int argc, char *argv[]){
  FILE *fic;
  int c;
  if (argc != 2)
    printf("Syntaxe : affiche fichier\n");
  else
    if ((fic = fopen(argv[1],"r")) == NULL)
      printf("remplace : probleme d'ouverture de fichier\n");
    else {
      while ((c=fgetc(fic))!=EOF) /* -1 c doit etre de type int */
        putchar(c);
      fclose(fic);
   }
}
```

mode chaîne de caractères (fgets, fputs)

La fonction *fgets* lit une chaîne de caractères dans un fichier ouvert :

char * **fgets**(char *s, int n, FILE * f)

La zone pointée par *s* doit être allouée avant l'appel de *fgets* et le paramètre *n* est le nombre maximum de caractères pouvant être lus (souvent calculé en fonction de la taille de la zone de réception). La lecture s'arrête dès qu'une fin de ligne ou fin de fichier (le caractère "\n" ou EOF est recontrée, elle est alors remplacée dans *s* par "\0"). La fonction retourne la chaîne de caractères ou NULL en cas de fin de fichier ou en cas de problème de lecture.

La fonction *gets(char *s)* est similaire pour l'entrée standard :

char * **gets**(char *s)

La fonction *fputs()* écrit une chaîne de caractères dans un fichier.

int **fputs**(char *s, FILE *f)

Le caractère "\0" est remplacé par une fin de ligne dans le fichier. En cas de problème, la valeur retournée est négative.

La fonction *puts(char *s)* est similaire pour la sortie standard :

int **puts**(char *s)

Exemple :

```
/* affice toutes les lignes d'un fichier    */
/* les lignes de plus de 80 carateres sont  */
/* decoupees en troncon de 80               */
#include<stdio.h>
#define SIZE 80
int main(int argc, char *argv[]){
  FILE *fic;
  char *buf=(char *)malloc(SIZE +1);
  if (argc != 2)
    printf("Syntaxe : affiche fichier\n");
  else
    if ((fic = fopen(argv[1],"r")) == NULL)
      printf("remplace : probleme d'ouverture de fichier\n");
    else {
      while (fgets(buf,SIZE,fic)!=NULL)
        puts(buf);
      fclose(fic);
   }
}
```

mode bloc de données (fread, fwrite)

La fonction *fread* lit un bloc dans un fichier.

int **fread**(char *ptr, int size, int nitems, FILE *f)

La fonction lit *nitems* éléments de la taille *size* du fichier *f*. Le premier élément sera écrit à l'adresse *ptr*. La fonction retourne le nombre d'éléments réellement lus.

Exemple

```
// liste un fichier d'entier
// le fichier fic doit etre ouvert
{
int val;
//tant que l'on peut lire un entier
while (fread(&val,sizeof(int),1,fic) == 1)
  printf("%d\n",val);
}
```

Exmple pour un tableau

```
//type complexe
typedef struct {
 double im,re;
} Tcomplexe;
// liste un tableau de complexe

{
Tcomplexe tab[10];
//on essaie de lire 10 complexe
nb = fread(&tab,sizeof(Tcomplexe),10,fic);
printf("nombre de complexe lus %d\n",nb);
}
```

La fonction *fwrite* écrit un bloc dans un fichier.

int **fwrite**(char *ptr, int size, int nitems, FILE *f)

La fonction écrit *nitems* éléments, chacun de taille *size*, dans le fichier *f*. Le premier élément se trouve à l'adresse *ptr*. La fonction retourne le nombre d'éléments réellement écrits.

En utilisant le forçage de type (cast), ces deux fonctions permettent de lire (ou d'écrire) directement des blocs de données correspondant à des structures, unions, tableaux ...

mode formaté (fprintf, fscanf)

En mode formaté, les fichiers sont traduits en une suite de caractères lisibles par un éditeur texte. Les données sont converties en caractères avant d'être écrites dans le fichier. Inversement les données sont reconverties à leur lecture. Ces fonctions utilisent les mêmes formats que les fonctions *printf* et *scanf*.

La fonction *fscanf* lit une suite de données dans un fichier (ouvert) en fonction d'un format donné.

int **fscanf**(FILE *f, char *format, liste_d_adresses_de_variable)

La fonction *fprintf* écrit une suite de données dans un fichier en fonction d'un format donné.

int **fprintf**(FILE *f, char *format, liste_de_variables)

En cas de problème, la valeur retournée par l'une des fonctions est négative ; sinon la fonction *fscanf* retourne le nombre de conversions réussies et la fonction *fprinf* le nombre de caractères écrits.

Positionnement dans un fichier (fseek, ftell, rewind)

L'instruction *fseek* permet de forcer la position courante du fichier sur un endroit donné :

int **fseek**(FILE * f, long offset, int origine)

Le paramètre *offset* correspond à un déplacement en nombre d'octets par rapport à l'origine. Cette origine peut prendre les valeurs suivantes :

SEEK_SET **0** /* *début du fichier* */
SEEK_CUR **1**/* *position courante* */
SEEK_END **2**/* *fin du fichier* */

Exemple pour se positonne sur le Nième entier d'un fichier.

```
//positione sutr le Nene entier
// le fichier (fic) doit biensur etre ouvert
fseek(fic,(N -1) * sizeof(int),SEEK_SET);
```

exemple : debut du fichier

```
fseek(fic,0,SEEK_SET);
```

fin du fichier

```
size = fseek(fic,0,SEEK_END); //retourne la taille du fichier
```

La fonction*ftell* retourne la position dans le fichier. (donnée en nombre d'octets par rapport au début du fichier.)

long **ftell**(FILE * f)

La fonction *rewind* repositionne le pointeur sur le premier caractère du fichier.

int **rewind**(FILE * f)

8 La programmation modulaire

La compilation séparée

Le langage C permet la compilation séparée. Le source d'une application est alors réparti sur plusieurs fichiers. Ce type de compilation permet de diminuer la taille des fichiers et surtout la réutilisation des fonctions existantes.

Le mot clé *extern* permet de déclarer une fonction ou variable non définie dans le fichier. (Ces objets seront décrits dans d'autres fichiers). Les fonctions *extern* doivent être déclarées avec leur entête (prototype de fonction).

A l'édition des liens, il faut qu'une et une seule fonction *main* soit définie dans l'un des fichiers.

Voici un exemple comprenant deux fichiers et les commandes de compilation séparée.

fichier factoriel.h

```
/* Interface du module factoriel */
extern long unsigned fac(unsigned) ; /*calcul de la factorielle d'un entier */
```

fichier factoriel.c

```
/* Implantation module factoriel */
#include"factorielle.h"
long unsigned fac(unsigned n) { /*fonction factorielle */
     /* calcul recursif du factoriel(n) */
     return (n ? n * fac(n - 1) : 1) ;
}
```

fichier programme.c

```
/* module principal (utilisation du module factorielle) */
#include<stdio.h>
#include"factorielle.h"
int main(void ) { /*fonction principale */
     unsigned n ;
     printf ("Factorielle de n \n" ) ; scanf("%u",&n) ;
     if (n > 0)
          printf("Factorielle n%u = %lu \n",n,fac(n)) ;
}
```

commande de compilation et d'edition des liens

```
cc -c -o factoriel.o factoriel.c
cc -o programme programme.c factoriel.o
```

NB : L'option -c compile sans effectuer l'édition des liens et l'option -o permet de nommer le fichier résultat de compilation.

Sous Unix la comande *nm* permet de lister les symboles contenus dans les fichiers objets (.o) ou archives (.a ou .so).

```
> nm factoriel.o
00000000 T fac
```

Le carctères T indique que le symbole est terminal et U inconnu.

Si des noms de structures ou de types (ou macro-instructions) doivent être partagés par plusieurs fichiers, il est préférable de les décrire dans un fichier *"include"* qui sera inclus dans tous les fichiers.

Syntaxe :

#include “nom_de_fichier.h”

Exemple d'utilisation de prototype.

```
// exemple de fonctions croisees

int pair(unsigned v); // prototype de la fonction
                      // appele dans la deuxieme fonction
int impair(unsigned v) {
  return (v == 0 ? 0 : pair(v-1));
}
int pair(unsigned v) {
  return (v == 0 ?  ! 0 : impair(v-1));
}
int main(void) {
  printf("pair(5) %d pair(6) %d \n",pair(5),pair(6));
}
```

La commande make

Le fichier *makefile* ou *Makefile* permet de décrire les liens de dépendance des fichiers d'une application pour la compilation séparée. A partir de ce fichier, la commande *make* ne compile que les fichiers modifiés depuis la dernière compilation, ainsi que les fichiers “.o” ou exécutables les utilisant (La commande utilise les dates des fichiers).

Dans ce fichier les entités manipulées sont des entrées (associées en général à des noms de fichiers). Les entrées dans ce fichier *makefile* sont suivies du caractère “ **:**”. La liste des dépendances (noms de fichier *.o, .c, .h*) suit alors chaque entrée. La commande *make* analyse la première entrée du fichier (souvent appelée *all*). Les commandes à exécuter pour chaque entrée sont décrites sous la ligne de cette entrée. Chaque ligne de commande est alors précédée d'une tabulation.

Voici le fichier *makefile* de l'application ci-dessus :

fichier makefile

```
# commentaire
# all : point de départ
all : programme
#==========================
# edition des liens
#==========================
programme : factoriel.o programme.o
	cc -o programme programme.o factoriel.o
#==========================
# compilation
#==========================
# module programme
programme.o : programme.c factoriel.h
	cc -c -o programme.o programme.c
#==========================
# module factoriel
factoriel.o : factoriel.c factoriel.h
	cc -c -o factoriel.o factoriel.c
```

Le système utilise la date système (de dernière modification) pour recompiler ou pas les modules.

Exemple conséquant de fichier Makefile.

```
#  Makefile
#  application Gsstion des segments
#
#  L Courtrai
#  12 /3/ 99

#Definition des macros
# syntaxe NOM= chaine

CC=gcc  # Definition d'un macro CC qui contient le nom du compilation

# option pour les inclusions (repertoire de recherche)
INC_FLAGS =  -I./include/

# mode normal
# CFLAGS=
# mode Debug
CFLAGS= -DDEBUG

# recherche des librairie pour l'editions des liens
LDLIBS= -L${LIB}


# repertoire de projet
LIB = ./lib/$(ARCH)/  #$(XXX) varaible d'environnement du shell
SRC = ./src/
OBJ = ./obj/$(ARCH)/
BIN = ./bin/$(ARCH)/


# entrre de depart
all : ${LIB}/libSegment.a ${BIN}/test

# contruction de la librairie
#
${LIB}/libSegment.a : ./${OBJ}/tableSegment.o
        ar rcv ${LIB}/libSegment.a ./obj/$(ARCH)/tableSegment.o
        ranlib ${LIB}/libSegment.a

#
#
# Compilation des modules


# gestion de la table des segment
${OBJ}/tableSegment.o:${SRC}/tableSegment.c
        ${CC} -c ${INC_FLACS} ${CFLAGS} -o ${OBJ}/tableSegment.o ${SRC}/tableSegment.c



# Module de test

${BIN}/test: ./exemple/test.c ${LIB}/libSegment.a ./include/segment.h
        ${CC} -o ${BIN}/test ${INC_FLAGS} ${CFLAGS} \
```

```
  ./exemple/test.c ${LDLIBS}  ${LIB}/libSegment.a


# nettoie

clean :
        -rm ${LIB}/*.a ${OBJ}/*.o  ${BIN}/* *~ src/*~


# installation du logiciel
#    par root
install :
        -cp ${LIB}/*.a /usr/local/lib
```

Les librairies static

```
// fichier carre.c
// fonction carre
int icarre(int i){
  return i*i;
}

// fichier cube.c
// fonction cube
int icube(int i){
  return i*i*i;
}

// compilation partielle
gcc -c -o cube.o cube.c
gcc -c -o carre.o carre.c
// archive les fichier .o dans la librairie cubeEtCarre.a
> ar rcv cubeEtCarre.a *.o
a - carre.o
a - cube.o

v verbose
r Insérer les fichiers dans l'archive (avec remplacement).
c Creation de l'achive
// met a jour le fichier d'index
// ranlib  génère  un  index du contenu d'une archive, et le stocke dans
       l'archive. L'index liste chaque  symbole  défini  par  un  membre  de
       l'archive, çàd un fichier objet.
> ranlib cubeEtCarre.a
// visualise le contenu de la librairie
> ar tv cubeEtCarre.a *.o
rw-r--r-- 261/200    759 Jan 15 13:08 2001 carre.o
rw-r--r-- 261/200    757 Jan 15 13:08 2001 cube.o
// extrait d'une archive un fichier
> ar xv cubeEtCarre.a carre.o
x - carre.o
// liste des symboles de la librairie
>nm cubeEtCarre.a
carre.o:
```

```
00000000 t gcc2_compiled.
00000000 T icarre
cube.o:
00000000 t gcc2_compiled.
00000000 T icube
// Liste de l'index dans la librairie
> nm -s cubeEtCarre.a

Archive index:
icarre in carre.o
icube in cube.o
```

La portée des objets C (extern, static)

Dans un programme *C*, tous les objets déclarés au premier niveau sont globaux. Les variables d'une fonction ou d'un bloc sont par opposition locales.

Par défaut, chaque variable ou fonction est exportable et pourra donc être utilisée dans les autres fichiers avec le mot clé *extern*. Dans ces fichiers, et uniquement pour les fonctions, le mot clé *extern* est optionnel, seule la déclaration du prototype de la fonction suffit.

Le corps de la fonction sera, soit déclaré dans un autre fichier, soit dans le même fichier. Cela permet d'écrire des procédures croisées où chaque procédure appelle l'autre.

Le mot clé *static* permet de limiter la portée d'une variable ou fonction à un module (fichier .o) (notion de variables privées au module). Ces objets ne sont alors plus exportables.

Les variables locales d'une fonction peuvent être *static*. Dans ce cas, la variable n'est pas détruite à la fin de l'appel de la fonction et conserve sa valeur au prochain appel.

Exemple :

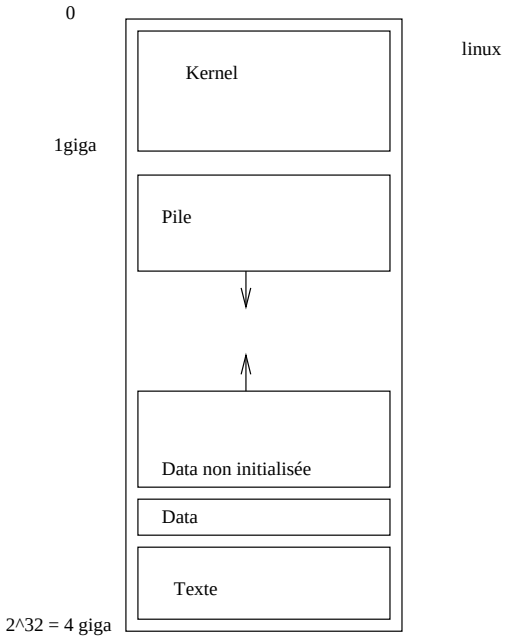
```
#define MAX 128
static int tab[MAX];
static cptF =0;

int f(void) {
    static int i = 0;
    tab[i] = 1;
    cpt++;
    return(++i);
}
int main(void ) { /* fonction principale */
    printf("%d,%d\n",f(),f()); /* affiche 1,2*/
    printf("%d,%d\n",tab[0],tab[1]); /* affiche 1,1*/
    printf("%d\n",cpt /*2*/
}
```

Dans cet exemple, le tableau est privé au module et ne sera donc pas accessible d'un autre module. Par contre il est vu dans toutes les fonctions de ce module. La variable i est elle privée à la fonction f, elle est initialiée au premier appel de la fonction est converve sa valeur d'un appel à l'autre.

8.1 mémoire d'un programme C Unix

Tous processus Unix possède un espace d'adressage constitué de trois (ou quatre) segments : texte (code), donnée et pile.

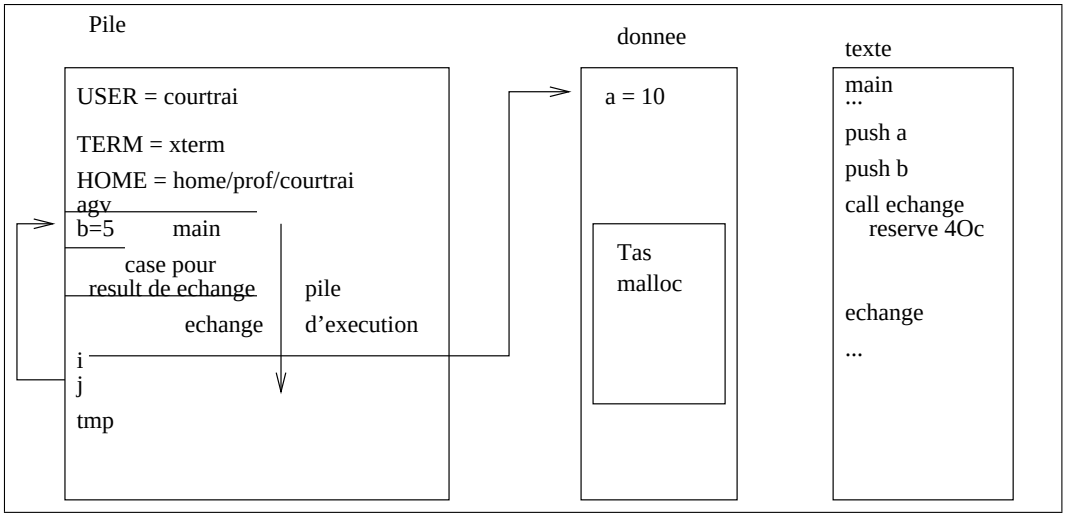
La memoire sur linux



– Le segment de texte contient les instructions en langage machine qui forment le code exécutable du programme. C'est un segment accédé en lecture et peut donc à priori être partagé par plusieurs processus.
– Le segment de donnée contient l'espace de stockage des variables du programme : chaîne de caractères, tableaux, ... En C, ce segment contient les variables globales et les espaces alloués dynamiquement (par la primitive *malloc*). Le système permet d'étendre un segment en lui ajoutant une page par un appel système. Cet appel est souvent utilisé par la primitive *malloc*.
– Le segment de donnée non initialisé. En C, ce segment contient les espaces alloués dynamiquement (tas). Le système permet d'étendre un segment en lui ajoutant une page par un appel système. Cet appel est souvent utilisé par la primitive *malloc*.
– Le segment de pile représente l'exécution d'un programme. En C il contient l'évaluation des expressions, l'empilement des appels de fonction avec leurs variables locales. Un programme ne peut pas gérer à priori sa taille de pile (par exemple à cause de la récursivité possible). Ce segment doit donc pouvoir s'étendre.
  A départ d'une exécution le segment ne contient que les variables d'environnement du shell et la ligne de commande.

exemple :

```
int echange(int *i, int *j) {
   int tmp;

   tmp =*i; *j=*i; *j= tmp;

   return 1;
}

int a=10;

int main() {

  int b=5;
  char * str = (char *) malloc (100);
  strcpy(str,''bonjour'');
  if (echange (&a, &b))

     printf("%d %d \n", a, b);
}
```



9 La librairie standard

Mise à par les fichiers (stdio.h) le C en standard propose un ensemble de fonctions :

9.1 Les fonctions sur les caractères (ctype.h)

```
int islower(int c);  // vrai si c appartient a..z
int isupper(int c);  // vrai si c appartient A..Z
int isalpha(int c);  // vrai si islower(c) ou isalpha(c)
int isdigit(int c);  // vrai si c appartient 0..9
int isxdigit(int c); // vrai si c appartient 0..9,A..F
int isalnum(int c);  // return isalpha(c) ou isdigit(c)

int ispace(int c);   // vrai si c est un ' ',\t,\n,\f,\r,\v
int isprint(int c);  // vrai si x est affichable
int toupper(in c);   // convertit le caractere en son equivalent majuscule
                     // si islower(c) est faux c est inchangé
int tolower(in c);   // convertit le caractere en son equivalent minuscule
                     // si isupper(c) est faux c est inchangé
```

9.2 Les fonctions mathématiques (math.h,stdlib)

fonction sur les réels

```
double cos(double x); //cosinus de x (x en radians (entre 0 et 2PI))
double sin(double x); //cosinus de x
double tan(double x); //tan de x

double exp(double x); //exponentiel de x
double log(double x); //logarithme naturel de x
double log10(double x); //logarithme en base 10 de x

double pow(double x, double y); // x élevé à la puissance y
double sqrt(double x) ; // racine carré de x

double fabs(double x); // valeur absolue de x
double floor(double x); // valeur entiere inférieur la plus proche de x

double ceil(double x); // valeur entiere supérieure la plus proche de x
double fmod(doube x, double y); // reste de x/y
```

fonction sur les entiers

```
int abs(int x); // valeur absolue de x
div_t div(int x,int y) // quotient et reste de x / y
                       // typedef struct {
                       //    int quot;int rem
                       // } div_t
long int labs(long x); // valeur absolue de x
ldiv_t ldiv(long x,long y) // quotient et reste de x / y
                       // typedef struct {
                       //    int quot;int rem
                       // } ldiv_t
```

9.3 La gestion mémoire (malloc.h,strlib.h,string.h

Les fonctions d'allocation et de déallocation d'un espace dans le tas

```
#include <stdlib.h>
void *malloc(size_t size);
```

alloue size octets (le contenu n'est pas initialisé)

```
void free(void *ptr);
```

après un malloc realloc,calloc ;

```
void *calloc(size_t nb, size_t size);
```

alloue nb éléments de size octets (Leur contenu est initialsé à 0.)

```
void *realloc(void *ptr, size_t size);
```

modifie la taille du bloc ptr à la nouvelle taille size (et recopie le contenu). Attention l'espace peut être deplacé.

malloc,calloc et realloc retunr NULL en cas d'echec.

```
 void *memcpy(void *dest, const void *src, size_t n);
 void bcopy (const void *src, void *dest, int n)
```

copie les n premiers octets de src dans dest. (return un poiteur sur dest (bcopy , ancienne version (deconseillée)

```
 void *memmove(void *dest, const void *src, size_t n);
```

copie les n premier octets de src dans dest. (return un poiteur sur dest) elle garantit les eventuels chevauchements des deux zones.

```
 void *memmove(void *dest, const void *src, size_t n);
```

```
void *memset (void *s, int c, size_t n);
```

remplit les n premiers octets de s à la valeur c ;

```
void *bzero (void *s,  size_t n);
```

remplit les n premiers octets de s à la valeur 0 ; (deconseillée).

9.4 Générateur aléatoire

```
// exemple d'utilisation du generateur de nombres aleatoires
//
// Luc Courtrai

#include<sys/time.h>


int main(void) {

  int i =0;
```

```
  int init; // valeur pour initialiser le generateur

  // recupere l'heure systeme pour initialiser le generateur
  time_t t;  //strut
  init= time(&t);

  srandom(init);  // initialisation du generateur

  while ( i++ <10)
    printf("%ld \n",random());  // genere un nombre aleatoire
```

9.5 format des scanf et printf

```
 printf("%%"); // %

printf("%*d",nbC, 100);
```

```
  0      indique  le  remplissage  avec  des zéros. Pour les
         conversions d, i, o, u, x, X, a, A, e, E, f, F,  g,
         et  G,  la  valeur  est complétée à gauche avec des
         zéros plutot qu'avec des espaces.  Si les attributs
         0  et  -  apparaissent  ensemble,  l'attribut 0 est
         ignoré.  Si une précision est fournie avec une con
         version numérique (d, i, o, u, x, et X), l'attribut
         0 est ignoré.  Pour les autres conversions, le com
         portement est indéfini.

       -      indique  que  la  valeur doit être justifiée sur la
              limite gauche du champ (par  défaut  elle  l'est  à
              droite).  Sauf  pour  la  conversion n, les valeurs
              sont complétées à droite par  des  espaces,  plutôt
              qu'a  gauche  par  des  zéros  ou  des  blancs.  Un
              attribut - surcharge un attribut 0 si les deux sont
              fournis.

       ´ ´    (un  espace)  indique qu'un espace doit être laissé
              avant un nombre positif (ou une chaîne  vide)  pro
              duit par une conversion signée

       +      indique  que  le  signe  doit toujours être imprimé
              avant un nombre produit par une conversion  signée.
              Un attribut + surcharge un attribut 'espace' si les
              deux sont fournis.
          signé.  La précision, si elle est mentionné, corre
         spond au nombre minimal  de  chiffres  qui  doivent
         apparaître.  Si  la  conversion  fournit  moins  de
         chiffres, le résultat est rempli à gauche avec  des
         zéros.  Par  défaut la précision vaut 1.  Lorsque 0
         est converti avec une précision valant 0, la sortie
         est vide.
```

sur les longeurs des types

hh La conversion entière suivante correspond à un signed char ou unsigned char, ou la conversion n suivante correspond à un argument pointeur sur un signed char.

h La conversion entière suivante correspond à un short int ou unsigned short int, ou la conversion n suivante correspond à un argument pointeur sur un short int.

l (elle) La conversion entière suivante correspond à un long int ou unsigned long int, ou la conversion n suivante correspond à un pointeur sur un long int, ou la conversion c suivante correspond à un argument wint_t, ou encore la conversion s suivante correspond à un pointeur sur un wchar_t.

ll (elle-elle) La conversion entière suivante corre spond à un long long int, ou unsigned long long int, ou la conversion n suivante correspond à un pointeur sur un long long int.

sur les types

d, i L'argument int est convertie en un chiffre décimal signé. La précision, si elle est mentionné, corre spond au nombre minimal de chiffres qui doivent apparaître. Si la conversion fournit moins de chiffres, le résultat est rempli à gauche avec des zéros. Par défaut la précision vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.

o, u, x, X
L'argument unsigned int est converti en un chiffre octal non-signé (o), un chiffre décimal non-signé (u), un chiffre héxadécimal non-signé (x et X). Les lettres abcdef sont utilisées pour les conver sions avec x, les lettres ABCDEF sont utilisées pour les conversions avec X. La précision, si elle est indiquée, donne un nombre minimal de chiffres à faire apparaître. Si la valeur convertie nécessite moins de chiffres, elle est complétée à gauche avec des zéros. La précision par défaut vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.

e , E L'argument réel, de type double, est arrondi et présenté avec la notation scientifique [-]c.ccce± cc dans lequel se trouve un chiffre avant le point, puis un nombre de décimales égal à la précision demandée. Si la précision n'est pas indiquée, l'affichage contiendra 6 décimales. Si la précision vaut zéro, il n'y a pas de point décimal. Une con version E utilise la lettre E (plutôt que e) pour

introduire l'exposant. Celui-ci contient toujours au moins deux chiffres. Si la valeur affichée est nulle, son exposant est 00.

f, F L'argument réel, de type double, est arrondi, et présenté avec la notation classique [-]ccc.ccc, où le nombre de décimales est égal à la précision réclamée. Si la précision n'est pas indiquée, l'affichage se fera avec 6 décimales. Si la précision vaut zéro, aucun point n'est affiché. Lorsque le point est affiché, il y a toujours au moins un chiffre devant.

SUSv2 ne mentionne pas F et dit qu'il existe une chaîne de caractères représentant l'infini ou NaN. Le standard C99 précise '[-]inf' ou '[-]infinity' pour les infinis, et une chaîne commençant par 'nan' pour NaN dans le cas d'une conversion f, et les chaînes '[-]INF' '[-]INFINITY' 'NAN*' pour une conversion F.

c S'il n'y a pas de modificateur l, l'argument entier, de type int, est converti en un unsigned char, et le caractère correspondant est affiché. Si un modificateur l est présent, l'argument de type wint_t (caractère large) est converti en séquence multi-octet par un appel à wctomb, avec un état de conversion débutant dans l'état initial. La chaîne multi-octet résultante est écrite.

s S'il n'y a pas de modificateur l, l'argument de type const char * est supposé être un pointeur sur un tableau de caractères (pointeur sur une chaîne). Les caractères du tableau sont écrits jusqu'au car actère NUL final, non compris. Si une précision est indiquée, seul ce nombre de caractères sont écrits. Si une précision est fournie, il n'y a pas besoin de caractère nul. Si la précision n'est pas donnée, ou si elle est supérieure à la longueur de la chaîne, le caractère NUL final est nécessaire.

p L'argument pointeur, du type void * est affiché en héxadécimal, comme avec %#x ou %#lx.

9.6 Les conversions en mémoire

Les fonctions scanf et printf s'appliquent sur l'entrée et la sortie standard , les fonctions fscanf et fprint sur les fichiers FILE *,les fonctions sscanf er sprintf s'applique sur un espace en mémoire,

int **sscanf**(char * adr, char *format, liste_d_adresses_de_variable)

exemple

```
char *ch = "13 13.4 coucou 14 14";
int i;float f; char s[64];
if (sscanf(ch,"%d %f %s",&i,&f,s) == 3)
```

```
    printf("%d %f %s\n",i,f,s);
  }
```

int **sprintf**(char * adr, char *format, liste_de_variables)

La fontion le nombre de caractères écrits dans adr.
exemple

```
int i = 13 ; float f = 13.4; char *s = "coucou";
char ch[512];
if (sprintf(ch,"%d %f %s",i,f,s) < 512)
   printf("%s\n",ch);
  }

int atoi(const char * str);
                            // convertit le debut de la chaine str
                            // dans un int
long int atol(const char * str);
                            // convertit le debut de la chaine str
                            // dans un long int
long int strtol(const char * ptr, char ** endptr,int base)
                            // convertit le debut de la chaine str
                            // dans un long
                            // la base doit etre comprsie en 0 et 36
                            // 0 =  base 8
                            // en cas d'echec errno vaut ERANGE
                            // si endptr n'est pas NULL
                            // endptr pointe sur le caractere
                            // suivant l 'entier
long unsigned strtoul(const char * ptr, char ** endptr,int base)
                            // convertit le debut de la chaine str
                            // dans un long unsigned
                            // la base doit etre comprsie en 0 et 36
                            // 0 =  base 8
                            // en cas d'echec errno vaut ERANGE
double atof(const char * str);
                            // convertit le debut de la chaine str
                            // dans un double
double strtod(const char * ptr, char ** endptr)
                            // convertit le debut de la chaine str
                            // dans un double
```

9.7 Générateur aléatoire

```
// exemple d'utilisation du generateur de nombres aleatoires
//
// Luc Courtrai

#include<sys/time.h>
```

```
int main(void) {

  int i =0;

  int init; // valeur pour initialiser le generateur

  // recupere l'heure systeme pour initialiser le generateur
  time_t t;  //strut
  init= time(&t);

  srandom(init);  // initialisation du generateur

  while ( i++ <10)
    printf("%ld \n",random());  // genere un nombre aleatoire
```

9.8 Gestion de la date

La fonction *ftime* ou *time* permet de récupére la date système de la machine.

```
  struct tm {
       int tm_sec; // seconde
       int tm_min; // minute
       int tm_hour; // heure
       int tm_mday; // hour du mois
       int tm_mon; //  numero de mois (janvier 0)
       int tm_year; // annee
       int tm_wday; // jour dans la semaine ( dimanche 0)
       int tm_yday; // jour de l'annee
       int tm_isdst; // heure d'ete
  };

time_t time(time_t *timer); //lit l'heure systeme
                            // nombre de secondes depuis
                            // le 1 janvier 1900
char * ctime(time_t *timer); // converti un time_t en ascii (char *)
struct tm *localtime(time_t *timer); //decompose la date en
                      // seconde,minute,heure;jour mois,annee ...
struct tm *gmtime(time_t *timer); //decompose la date en
                      // seconde,minute,heure;jour mois,annee ...
                      // en heure universel
char*  asctime(const strcut tm*);
                      // formate une date decomposee
                      // jour -> 0 Sun 1 Mon
                      // mois -> 0 Jan 1 Feb
double difftime(time_t *t1,time_t *t2);
                      // calcule le nombre de seconde entre deux dates
time_t mktime(struct tm *)
                      // construit un time-t a partir d'une date
                      // deconposee

#include <time.h>
```

```
int main(){

  time_t tps;    // temps

  tps = time(0); // recupere la date systeme

  printf("date : %s", ctime(&tps)); // conversion ascii du temps


    {
    struct tm * dateDecomposee;
    dateDecomposee = localtime(&tps);

    printf("%d:%d:%d   %d %d %d \n%d jour de la semaine \n\
%d jour de l'annee \n%d indicateur d'heure d'ete \n",
              dateDecomposee->tm_hour,
              dateDecomposee->tm_min,
              dateDecomposee->tm_sec,
              dateDecomposee->tm_mday,
              dateDecomposee->tm_mon + 1,
              dateDecomposee->tm_year + 1900,
              dateDecomposee->tm_wday,
              dateDecomposee->tm_yday,
              dateDecomposee->tm_isdst
     );
     printf("date : %s\n",asctime(dateDecomposee));

    }

}
```

strtime

```
#include<time.h>
#include <locale.h>

int main(){
  char date[512];

  time_t t =time(0); //recupere la date systeme


  struct tm * tmDate =  localtime(&t);

  char * result=setlocale(LC_TIME, "fr_FR");

  printf("%s\n", result);

  strftime(date,512,"%A %d %B %Y",tmDate);

  printf("%s \n",date);

  // mardi 19 novembre 2002
}
```

10 C avancé

10.1 assert (assert.h)

```
void assert(int expr); // place un test dans un programme
                       // si epr est fausse
                       // afficher sur stderr le fichier et le ligne de assert
                       // effectue un abort = (exit(0));
```

```
#include"stdio.h"
#include<assert.h>
int main(int argc, char *argv[]) {
  FILE * fic;
  int c;
  if (argc < 2) {
     fprintf( stderr,"usage: %s file x \n ",argv[0]);
     exit(1);
  }
  fic = fopen(argv[1],"r");
  assert(fic != NULL);
  while ( (c = fgetc(fic)) != EOF )
          putchar(c);
  fclose(fic);
}
```

```
> a.out toto
a.out: assert.c:11: main: Assertion ‘fic != ((void *)0)’ failed.
Abort
```

10.2 La gestion des erreurs

Outre la possibilité d'utiliser la sortie d'erreur standard (stderr), les fonctions C peuvent utiliser une variable globale *errno* pour prévenir qu'une erreur de survenue. De plus la fonction *perror(char *)* affiche un message d'erreur en fonction de la valeur de l'erreur.

```
       EILSEQ pour les erreurs  de  codage  dans  les  caractères
       étendus, ou multi-octets.

       E2BIG  liste d'arguments trop longue

       EACCES Interdiction d'accès

       EAGAIN Ressource indisponible temporairement

       EBADF  Mauvais descripteur de fichier.

       EBADMSG
              Mauvais message

       EBUSY  Ressource en cours d'utilisation.

       ECANCELED
              Opération annulée.
```

ECHILD Pas de processus fils.

EDEADLK
       Blocage d'une ressource évité.

EDOM   Erreur de domaine.

EEXIST Fichier existant.

EFAULT Mauvais adresse.

EFBIG Fichier trop grand.

EINPROGRESS
       Opération en cours.

EINTR Appel système interrompu.

EINVAL Argument invalide.

EIO   Erreur d'entrée/sortie.

EISDIR Est un répertoire.

EMFILE Trop de fichiers ouverts.

EMLINK Trop de liens symboliques.

EMSGSIZE
       Longueur du buffer de message inappropriée.

ENAMETOOLONG
       Nom de fichier trop long.

ENFILE Trop de fichiers ouverts sur le système.

ENODEV Périphérique inexistant.

ENOENT Fichier ou répertoire inexistant.

ENOEXEC
       Exécution impossible.

ENOLCK Pas de verrou disponible.

ENOMEM Pas assez de mémoire.

ENOSPC Plus de place sur le périphérique.

ENOSYS Fonction non implémentée.

ENOTDIR
       Pas un répertoire.

```
       ENOTEMPTY
              Répertoire non vide.

       ENOTSUP
              Opération non supportée.
        ENXIO  Périphérique ou adresse inexistant.

       EPERM  Opération interdite.

       EPIPE  Tube sans lecteur.

       ERANGE Résultat trop grand.

       EROFS  Système de fichiers en lecture-seule.

       ESPIPE Recherche invalide.

       ESRCH  Processus inexistant.

       ETIMEDOUT
              Délai maximal écoulé.

       EXDEV  Lien inapproprié.
```

Exemple

```
#include <math.h>
// division entiere
int div(int x,int y){
   if (y!=0)
       return x/y;
   else {
       errno = EDOM;
       return 0;
   }
}

// main
int main(void) {
  int x;
  errno = 0;
  // appel de la fonction div
  x = div(10,2);
  if (errno == EDOM) { // erreur
     perror("div ");
  }
  errno = 0;
  x = div(10,0);
   if (errno == EDOM) {
     perror("f ");
  }
  // appel de la fonction mathematique sqrt
  errno = 0;
  x = sqrt(-1);
```

```
  if (errno == EDOM) { // erreur de domaine
     perror("sqrt ");
  }
}
// gcc testErreur.c -lm
//
a.out
div : Numerical argument out of domain
sqrt : Numerical argument out of domain
```

Une liste d'erreurs (EDOM, ERANGE, EACCESS, ...) est prédéfinie. Les messages d'erreur aussi dans (sys_errlist[] et sys_nerr

Sortie de programme

```
void exit(int status);
       // termine le programme et retourne un code de retourne
       // à l'interpr'teur un code de retour status
void abort(void)
       // termine le programme
int atexit(void (*func)(void))
       // appel la fonction func  lors de la fin normale du programme
```

10.3 Les Fonctions comme paramètre de fontion

Le C permet de manipuler des fonctions comme des objets. On peut ainsi a l'exécution choisir une fonction plutôt qu'une autre.

exmple :

```
// pointeur de type double f(double);
typedef double (* p_double_Fonc_double)(double);

// applique un fonction passe en parametre sur un element

void apply(p_double_Fonc_double pf, double *val) {
  *val = pf(*val);
}

// exemple de fonction a applique
double carre (double i){
   return i*i;
}

// exemple d'utilisation

int main(void) {

  double x =5;

  apply(carre,&x); // appel d'apply avec la fonction carre
  printf("apply(carre)%lf\n",x);

  apply(sqrt,&x); // squrt de la librairie libmath.a
  printf("apply(sqrt)%lf\n",x);

  {
```

```
 //definition d'un tableau de pointeurs de fonction
  p_double_Fonc_double tabPf[2];
  int i;
  tabPf[0] = carre;
  tabPf[1] = sqrt;

  for(i =0;i <2;i++) {
    double v =5;
    v = tabPf[i](v);
    printf("%lf\n",v);
  }
  }
}

// compilation
// gcc test.c -lm  // la fonciotn sqrt
```

10.4 Fonction à nombre de paramètres variable

Le C permet d'écrire des fonctions à nombre de paramètres variable. Certaines fonctions comme *printf et scanf* ont deja cette particulatiré. Les paramêtres de la fonction sont alors gérer comme une liste et c'est au programmeur de spécifier les types de chaque paramètre lors du parcours de cette liste.

```
#include <stdarg.h>
 // foo nombre de parametres variable ...
 void foo(char *fmt, ...){
                   va_list ap;
                   int d;
                   char c, *p, *s;
                   va_start(ap, fmt); // position sur le debut de la liste
                                      // le premier argument decrit le
                                      // type des autre parametre
                   // analyse le contenu du premier argument
                   while (*fmt)
                        switch(*fmt++) {
                        case 's':           /* string */
                             // extrait un argument
                             s = va_arg(ap, char *);
                             printf("string %s\n", s);
                             break;
                        case 'd':           /* int */
                             d = va_arg(ap, int);
                             printf("int %d\n", d);
                             break;
                        case 'c':           /* char */
                             c = va_arg(ap, char);
                             printf("char %c\n", c);
                             break;
                        }
                   va_end(ap);
```

```
}
// utilisation
int main(){
  foo ("ddcs",12,13,'a',"coucou");
}
```

10.4.1 Gestion de l'ecran (ASCII)

Une suite de caratère ASCII permet de gérer le positionnement sur curseur sur l'écran ainsi que féffacemetn de lécran.

```
#define  CLEAR   printf("\033[2J")
#define  LOCATE(lig,col) printf("\033[%d;%dH",lig,col)

int main(){
  CLEAR;
  LOCATE(10,15);printf("*****  GESTION DE STOCK *****");
  LOCATE(12,5);printf("creation du fichier       c");
  LOCATE(13,5);printf("saisie d'une fiche        s");
  LOCATE(14,5);printf("liste des fiches          l");

  LOCATE(16,5);printf("sortie du logiciel        S");

  LOCATE(18,5);printf("votre choix              :");
  rep=getchar();
  ...
}
```

Gestion des couleurs

```
int main(){
  printf("\033[0m"); // noir
  printf("\033[1m"); // gras
  printf("\033[5m"); // gras
  printf("\033[7m"); // inverse video
  printf("\033[8m"); // fond
  printf("\033[31m"); // couleur 1
  printf("\033[32m"); // couleur 2
  printf("\033[33m"); // couleur 3
  printf("\033[34m"); // couleur 4
  printf("\033[35m"); // couleur 5
  printf("\033[36m"); // couleur 6
  printf("\033[37m"); // couleur 7
  printf("\033[41m"); // inverse couleur 1
  printf("\033[42m"); // inverse couleur 2
  printf("\033[43m"); // inverse couleur 3
  printf("\033[44m"); // inverse couleur 4
  printf("\033[45m"); // inverse couleur 5
  printf("\033[46m"); // inverse couleur 6
  printf("\033[47m"); // inverse couleur 7
}
```

10.4.2 La librairie curses (unix)

La librairie curses propose un ensemble de fonctions pour gérer un terminal Ecran,claveir et souris. (Effacer lécran,se positionner à un endroit, récupérer la position courante du curseur, ...)

Voici un petit exemple.

```
#include<curses.h>

int main(){
  int c,x,y;
  // definition du terminal
  WINDOW *win;
  // initialise m'ecran
  win=initscr();
  // efface la fenetre
  clear();

  // deplace le curseur a la position 5 0 (ligne colonne)
  move(5,0);

  mvaddstr(15,10,"texte a la position 15 10e : ");

  // on recupere la position du curserur
  x=getcurx(win);
  y=getcury(win);
  // equivalent a   getyx(win,y,x);


  //supprime  d'echo a l'ecran
  noecho();

  {
  // on lit un caractere sur le terminal sans DELAY
  char c= getch();
  }

  // remet l'echo
  echo();


  }

//cc -o curseur curseur.c -lcurses -ltermcap
```

10.5 Appel d'une commande du systeme d'exploitation

La fonction *system(char * ch)* permet à un programma C de lancer une commande système. exemple sur Unix :

```
// affiche le contenu d'un repertoire passe en argument
int main (int argc, char *argv[]) {
  if (argc != 1) {
    char commande [512];
    sprintf(commande,"ls \%s",argv[1]);
    system(commande);
  }
}
```

Sous unix le programme va créer un processus qui exécutera la commande. A la termaison de celui-ci le programme C reprend le controle.

10.6 Les librairies static

```
// fichier carre.c
// fonction carre
int icarre(int i){
  return i*i;
}

\begin{verbatim}
// fichier cube.c
// fonction cube
int icube(int i){
  return i*i*i;
}

\begin{verbatim}
// compilation partielle
gcc -c -o cube.o cube.c
gcc -c -o carre.o carre.c
// archive les fichier .o dans la librairie cubeEtCarre.a
> ar rcv cubeEtCarre.a *.o
a - carre.o
a - cube.o

v verbose
r Insérer les fichiers dans l'archive (avec remplacement).
c Creation de l'achive
// met a jour le fichier d'index
// ranlib  génère  un  index du contenu d'une archive, et le stocke dans
       l'archive. L'index liste chaque  symbole  défini  par  un  membre  de
       l'archive, çàd un fichier objet.
> ranlib cubeEtCarre.a
// visualise le contenu de la librairie
> ar tv cubeEtCarre.a *.o
rw-r--r-- 261/200    759 Jan 15 13:08 2001 carre.o
rw-r--r-- 261/200    757 Jan 15 13:08 2001 cube.o
// extrait d'une archive un fichier
> ar xv cubeEtCarre.a carre.o
x - carre.o
// liste des symboles de la librairie
>nm cubeEtCarre.a
carre.o:
00000000 t gcc2_compiled.
00000000 T icarre
cube.o:
00000000 t gcc2_compiled.
00000000 T icube
// Liste de l'index dans la librairie
> nm -s cubeEtCarre.a

Archive index:
```

```
icarre in carre.o
icube in cube.o
```

10.7 Les librairies dynamiques

Elles ne sont chargées à l'exécution que lors de l'appel de la prmière fonction et sont partagés par tous les programmes les utilisant (segment texte partagé).

Construction d'une librairie

```
// fichier carre.c
// fonction carre
int carre(int i){
  return i*i;
}

// gcc -shared  -c -o carre.so carre.c

// ou

// compilation des modules
// gcc -fPIC -c -o carre.o carre.c
// creation de la librairie
// gcc -shared -o carre.so carre.c
```

Utilisation

```
#include<dlfcn.h>
int main(void) {

  void *handle;
  int (*pcarre)(int);
  char *error;

  handle = dlopen ("./carre.so", RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(), stderr);
    exit(1);
  }

  pcarre = dlsym(handle, "carre");
  if ((error = dlerror()) != NULL)  {
    fputs(error, stderr);
    exit(1);
  }
  printf ("%d\n", (*pcarre)(2));
  dlclose(handle);
}
// gcc main.c -ldl
```

11 Annexe, la priorité des opérateurs

Opèrateur	Symbole	exemple	associa.
appel de fonction	( )	max(m,n)	–>
indexation	[ ]	tab[i][j][k]	–>
champ de structure	-> .	noeud->fg m.code	–>
négation logique	!	!a	<–
négation bit à bit	~	~a	<–
incrémentation	++	++i i++	<–
décrémentation	- -	- -i i- -	<–
moins unaire	-	-x	<–
coercition (cast)	(type)	(float)i (char *)p	<–
indirection	*	*p	<–
adresse	&	&x	<–
taille	sizeof	sizeof(int) sizeof(p)	<–
multiplication	*	a*b	–>
division	/	a/b	–>
modulo	%	a%b	–>
addition	+	a+b	–>
soustraction	-	a-b	–>
décalage	<< >>	x<<n x>>n	–>
relation	< <= > >=	a<3 b>=5	–>
[in]egalitée	!= ==	a==b c !=5	–>
et bit à bit	&	a&b	–>
ou exclusif bit à bit	^	a^b	–>
ou inclusif bit à bit	|	a|b	–>
conjonction (et)	&&	a&&b	–>
disjonction (ou)	||	a||b	–>
expression conditionnelle	? :	(c>0) ? x : -x	<–
affectation simple	=	a=10	<–
affectation étendue	+= -= /= *= %= etc.	x%=10	<–
expression composée	,	a=1,b=3,c=5	–>

Ce tableau indique l'échelle de priorité des différents opérateurs.Tous les opérateurs d'un groupe d'opérateurs (dans un cadre) ont la même priorité.

Le tableau est repris du livre de *Jean Marie Rifflet* "La programmation sous UNIX".