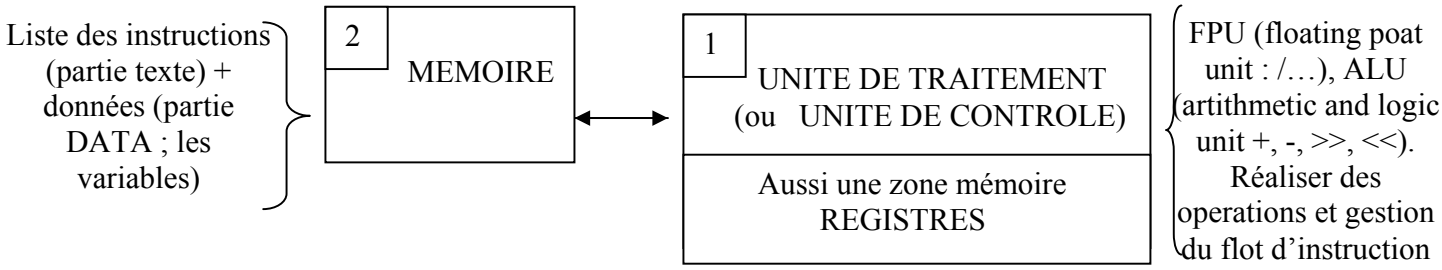
Notes sur le Langage C

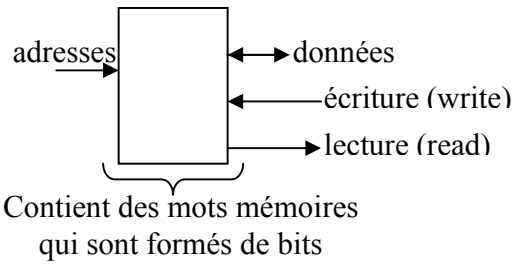
D'après livre "Le Langage C" de Aitken & Jones, et les cours de Gilles Menez
menez@unice.fr 04 92 94 27 10
Ouvrages conseillés : Méthodologie de la programmation en langage C (Braquelaire)
Langage C Ansi (Kernighan et Ritchie), Langage C (Claude Delannoy)

« Programmation » est un néologisme (mot nouveau) introduit en 1962. Il s'agit d'un traitement automatique de l'information. Il y a plusieurs paradigmes (façons de concevoir ce traitement) : O.O (orienté objet : C++, Java), impératif (comme le C), déclaratif (Prolog), fonctionnels (Lisp). Le paradigme impératif vient de la constitution des machines. On s'inspire du modèle de Von Neuman (années 40) qui définit des états (valeur mémoire) et les instructions pour les modifier. Un des inconvénients de ce paradigme est pour la recherche du parallélisme.



Ceci correspond à une définition d'automate : on va chercher des instructions, on les traite, on ramène le résultat.

La mémoire est une sorte de bibliothèque servant à mémoriser des informations. Ses temps d'accès sont différents : à chaque éloignement du microprocesseur, on peut considérer qu'il y a un facteur 10 (la mémoire est 10 fois plus longue à accéder que les registres, les disques durs sont 10 fois plus long que la mémoire, etc. etc.).



En général les mots sont de 8 bits (un octet) car le plus petit type char est logé sur 8 bits. Il n'y a pas d'organisation spécifique des données : ASCII, binaire naturel, IEEE754 (flottants)…
On a deux phases importantes : adressage (nombre d'adresses disponible, entraînant la contenance de la mémoire avec n fils pour contrôler 2^n fils) et accès.

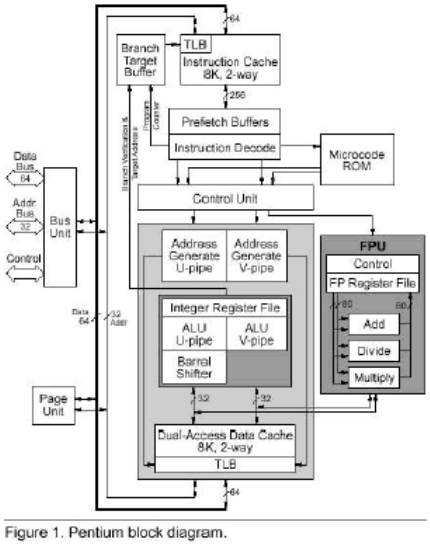
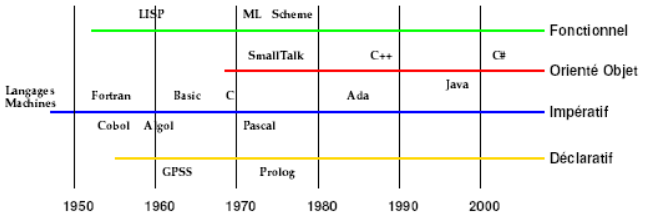


Figure 1. Pentium block diagram.

La représentation de la zone texte en mémoire (i.e. la partie instruction) induit une organisation et un codage. La définition dépend de la partie contrôle (les instructions du microprocesseur), donc du constructeur. C'est pourquoi les exécutables (par défaut a.out) ne vont pas fonctionner si on les passe d'un X86 à une machine sur processeur alpha.
Le codage peut-être de différentes formes pour faire la différence entre les instructions arithmétiques et les instructions d'accès à la mémoire. On peut les représenter par des mnémoniques (utilisation de l'assembleur avec le langage du microprocesseur : ADD pour additionner, LOAD pour charger, STORE pour stocker…).

I. Le Langage C, Historique et Caractéristiques

Le langage C fait partie de la lignée des paradigmes structuré, procédural et impératif.



En 1970, Ritchie (laboratoire Bell chez AT&T) met au point le langage C pour écrire UNIX (évolution de BCPL et B). Il y a les contraintes matérielles de l'époque : on devait tourner sur le PDP 11 (de Digital, racheté Compaq puis Hewlett-Packard) avec 24Ko de RAM pour le compilateur.
A l'époque, le C était un langage de haut-niveau :

- des flots (des boucles, des tests…)
- le typage

Maintenant, on pourrait surtout dire que le C est un assembleur de haut niveau. Il n'est pas orienté-objet et ne bénéficie donc pas de l'héritage.

Le premier C est le C de Kernigan et Ritchie (K&R, 1983). Exemple d'une fonction dans cette norme :

```
F(a, b) ;
int a ;
int b ;
```

La seconde norme est l'ANSI C (C89) avec f(int a, float b).
Enfin, il y a eu le C99 (complexe, booléens, tableau de taille dynamique, long long de 64 bits, long double de 128 bits).

II. Compilation

Le compilateur habituel gcc masque une chaîne de compilation (toolchain), qu'on peut afficher avec gcc –v (mode verbose) :
1) Précompilation : le préprocesseur. Réécriture de toutes les lignes commençant avec un #.
2) Compilation. Traduction de code source vers de l'assembleur (Intel…). Le fichier produit contient donc des instructions directement destinées au microprocesseur ; contrairement à Java (la JVM, Java Virtual Machine), ce n'est pas un programme qui exécute le code.
3) Assemblage : traduire l'assembleur en code objet (les .o).
4) Edition de liens (l'outil ln) : créer les liens entre les différentes fonctions car elles ne sont pas forcément situées dans le même fichier. Exemple : f.c → f.o qui utilise la fonction cos (libm.a).

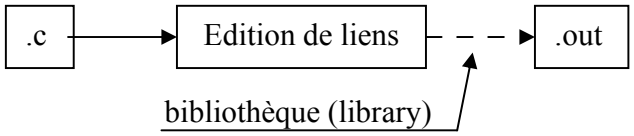
Quelques options de gcc :

- -lm, qui fait l'édition de lien avec la bibliothèque mathématique
- -E, le code tel qu'il a été réécrit par le préprocesseur
- -C, arrêt du compilateur après la création du .o. Pratique pour une bibliothèque (absence de main) ou une compilation séparée (réduire la phase de compilation). Si on arrête la compilation de tous les fichiers au .o, lorsqu'on modifie un fichier il faudra juste refaire son .o (pas besoin de toucher aux autres ; bien les gros projets). Compléter avec les liens.
- -ansi (choisir le dialecte C89), -pedantic (tout C non ANSI provoquera un warning)
- -fpic (fonction independant code) : impose au moment de la compilation le fait que le module objet va être relogeable ; il peut-être déplaçable n'importe où en mémoire et doit pouvoir fonctionner (adressage relatifs : exemple la var est à 4 + le début de la section)
- gcc –Wall –Werror fahr.c : Wall signale tous les risques, Werror arrête en cas de warning

Ainsi, pour compiler deux fichiers f.c et g.c :
1) gcc –C f.c 2) gcc –C g.c
3) gcc g.o f.o –O nom (ou gcc g.o f.o –O nom –lm, si besoin de faire le lien avec une bibliothèque)

III. Les Bibliothèques

Il s'agit d'un regroupement de fonctions utilisables par des programmes.



Il y a un gain de temps en réutilisant du code, de la fiabilité, et un code plus compréhensible (langage de la bibliothèque).

Le fonctionnement des bibliothèques et ce qui est inclus diffère selon les langages : en pascal, l'affichage avec print fait partie du langage, tandis qu'en C il est dans la bibliothèque.

Il y a deux formes de bibliothèques : statiques (static library : .a) et partagées (share library : .sh).

Le .out contient notre code et celui de la bibliothèque

Le .out contient uniquement notre code. Il y a partage de la présence du code de la bibliothèque en mémoire.

La naissance de bibliothèques partagées vient de l'utilisation de programmes en parallèle : avec les static on se retrouvait avec plusieurs versions de la bibliothèque dupliquée en mémoire. Avec les share, la bibliothèque est présente une seule fois en mémoire ; c'est cette forme qui est utilisée par défaut avec unix. Pour l'autre, il faut y faire référence de façon explicite avec

gcc g.o f.o –static –lm (référence à /usr/lib/libm.a)
gcc g.o f.o –static –lx11 (référence à /usr/x11lib/lib/libx11.a)

Commandes importantes :

ar –t /usr/lib/libm.a → acos.o … pow.o (Bien décomposer (une fonction pour un .o))

nm –s /usr/lib/libm.a → Table des symboles
x0 : 000000 T –ieee <blabla…>
nom ; adresse relative à la mémoire ; Référence à la section

(à noter : dans AIX pour faire une bibliothèque, il faut rajouter à la main la table des symboles)

La normalisation POSIX définit un standard des systèmes d'exploitation.

L'utilisation d'une bibliothèque se fait en 2 phases :

- #include <….h> : les headers ne sont pas la bibliothèque mais une interface (pour couvrir seulement une partie des fonctions offertes). Exemple, stdio.h (/usr/include) est l'interface de /usr/lib/libc.c permettant d'avoir les entrées/sorties. Les headers forment les déclarations mais les définitions sont dans la bibliothèque.
- édition de lien qui donnera la bibliothèque. On peut compiler sans l'option –lm en utilisant
- gcc f.c usr/lib/libm.a (libc.a et libc.so ne nécessitent pas d'apparaître : il est considéré comme d'utilisation évidente et placé par défaut avec l'édition de liens).

IV. Variables

1) Les types de variable

Au début du développement d'Unix (B ou BCPL), la notion de type n'existait pas ; on programmait en « cellule » ou « mot » mémoire. Avec l'évolution du matériel, des problèmes immédiats : les flottants n'occupaient pas forcément le même nombre de cellules d'une machine à l'autre, il fallait donc réécrire le code pour chaque machine.

On cherche donc à organiser la mémoire indépendamment de la structure de la machine. Le typage permet de définir un ensemble de données et d'opérations. D'où un intérêt :

- pour le pouvoir d'expression (haut-niveau par rapport à la mémoire)
- pour la compilation (procéder à des contrôles de cohérence)

Une instruction comme float x ; commence par réserver la zone en mémoire puis la nomme.
Cependant, en C, deux choses permettent de débrayer le typage :

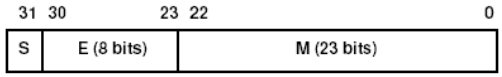
- le cast (ou coercition, transformation de type). C'est un problème lorsqu'on fait de la coercition sur des pointeurs.
- void*, des pointeurs génériques

Il y a deux catégories : les types scalaires (dont void) et les flottants.
Le tableau suivant récapitule les différents types pour une architecture 16 bits :

Type de variable	Mot clé	Octets	Intervalle
Caractère	char	1	-128 à 127
	wchar_t	2	-32768 à 32767
Caractère non signé	unsigned char	1	0 à 255
Entier	int	2	-32768 à 32767
Entier non signé	unsigned int	2	0 à 65535
Entier court	short	2	-32768 à 32767
Entier court non signé	unsigned short	2	0 à 65535
Entier long	long	4	-2147483648 à 2147438647
Entier long non signé	unsigned long	4	0 à 4 294 967 295
Simple précision virgule flottante	float	4	1,2E-38 à 3,4E38 (précis. à 7)
Double précision virgule flottante	double	8	2,2E-308 à 1,8E308 précis. 19

Les flottants : ANSI/IEEE 754 (1985)

Les flottants sont généralement implantés selon cette norme :



La valeur d'un float est donné par la formule (en simple précision) :

➢ $-1^{Signe} * 1.Mantisse * 2^{Exposant-127}$

Par exemple :

$$\begin{aligned} -0,40625 &= 0*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4} + 1*2^{-5} \\ &= -0,01101 * 2^0 \\ &= -1,101 * 2^{-2} \end{aligned}$$

➢ Signe = 1 car valeur négative
➢ Mantisse = 10100000 car le « 1, » est implicite en IEEE
➢ Exposant : -2 + 127 (le biais) soit 125 (11000000 en base 2)

L'énumération est équivalente à un unsigned int. Le domaine de valeur correspond à des constantes symboliques.

```
enum couleur {rouge, vert, bleu} c ;
/* On définit deux choses : un type qui porte le nom « enum couleur » et une variable c de type enum couleur */
enum couleur x ;
/* une variable x de type enum couleur */

enum L2 {martin, durand, dupont} ;
/* Implicitement on associe 0 à martin, 1 à durand, 2 à dupont */
enum admis {durand, martin} ;
/*On ne peut plus faire ça, car ici on associerait 0 à durand, 1 à martin */
```

Un nom de variable ne peut pas commencer par un chiffre, doit éviter de faire plus de 31 caractères (car seuls ceux-là sont significatifs), contient des lettres ou des chiffres avec respect de la casse, et éventuellement _ qui compte pour une lettre.

On peut définir un synonyme pour un type de données existant avec l'instruction typedef :
Typedef int matrix[10][10] ; /*Type 2 : matrix ; Type 1 : int[10][10]… usage valide : matrix m;*/
entier var; // valide à présent car le type entier a été défini

2) Constructions d'expressions : autour du symbole =
Le = réalise une affection (appelée 'effet de bord'). Son rôle est différent selon les langages : il s'agit d'un opérateur en C (car on retourne une valeur), mais c'est une instruction := en Pascal. Attention tout particulièrement à if(a=1) qui équivaut à if(1), la condition toujours vérifiée !

On parle de lvalue (left value) pour ce qui est à gaucher dans une affectation. Les constantes n'en sont pas, car une lvalue est une variable, quelque chose disposant d'une adresse.

```
char s[10] ;
char t[10] ;
s = t ; /* Attention, cela ne fait pas une recopie caractère par caractère dans la cible (utiliser pour cela la fonction strcpy(_,_) mais un échange de pointeur */

i = (j=k) + 1 ; /* Un gain de place… */
f((i=j+1)*(j=2*h)) ; /* Divinatoire… dépend de l'ordre d'évaluation de l'expression */
```

3) La portée des variables
a. Variables externes : extern
Quand une variable est déclarée hors d'un bloc de fonction, on dit qu'elle est externe (ou globale). On peut alors y accéder en tous points du programme. Lorsque le programme est plus compliqué, on peut parfois signaler qu'on fait référence à la variable globale :

```
int x;
void exemple(void);

void exemple(void){
        extern int x; /* Juste pour signaler, inutile…*/
        puts(x);      /* affichage de la valeur */
```

Le qualificatif volatile sert à signaler qu'une variable peut-être modifiée indépendamment du programme. Idéal pour les périphériques, qui sont assimilés à une variable de machine, mais où on peut lire et non piloter leurs valeurs (capteur…).

b. Utiliser le registre : register
Le processeur contient quelques emplacements mémoires nommés registres (pour addition, division...). On peut enregistrer une variable directement dans un registre afin que les opérations qui la concerne soient plus rapides (exemple : variable de boucle).
Dans ce cas on utilise le mot clé register. Cependant, c'est une demande et non un ordre : s'il n'y a pas de place dans le registre, on traitera cette variable comme une locale (aussi nommée "automatique" parce que le mot-clé 'auto' est implicite), simplement.
Exemple :

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

L'utilisation de register est en train de chuter avec le perfectionnement des compilateurs. C'est lié au problème d'optimisation : consommation d'énergie, donc lié à l'architecture.

c. Variables statiques : static

On utilise des variables statiques lorsqu'on veut se rappeler la dernière valeur de la variable.

Exemple :

```
#include <stdio.h>
void func1(void);
main(){
        for(int count =0;count<6;count++){
                printf("Iteration n° %d :",count);
                func1();
        }
        return 0;
}
void func1(void){
        static int x=0;
        int y=0;
        printf("x=%d, y=%d\n",x++,y++);
}
```

Ce qui nous amène à :

Iteration n°0 : x=0, y=0
Iteration n°1 : x=1, y=0
Iteration n°2 : x=2, y=0
Iteration n°3 : x=3, y=0
Etc. etc.

d. Les constantes : const

On a deux types de constantes :

- Littérales, introduites directement dans le code source. Exemple : const int count = 38.

A noter qu'une constante commençant par 0 est exprimée en octal, et avec 0x en héxadécimal.

- Symboliques, représentées par un nom (symbole) dans le programme.

Syntaxe possible : #define CONSTNAME litteral, instruction donnée au préprocesseur.

On définit alors un nom de constante avec la valeur littérale associée.

Exemple : #define PI 3.14

Ou en utilisant le mot clé const, tout à fait équivalent.

Exemple : const PI 3.14

int f(const int x) : la variable x ne doit pas être modifiée

int * const y = &a : le pointeur est constant sur a, ce qui est pointé peut-être modifié

const int * y : on peut modifier, le pointeur pas la valeur

const int * const y : rien ne peut être modifié

e. Adresses différentes : restrict

Le qualificatif restrict indique au compilateur que les adresses de deux pointeurs ne sont pas les mêmes (donc disjoints). On nomme aliasing le cas de deux pointeurs faisant référence à la même adresse.

Pour de problèmes de sûreté de fonctionnement et de sécurité, le compilateur peut ainsi être optimisé.

```
float * restrict a1, * restrict a2 ;
void init(int n){
        float * t = malloc(2*n*sizeof(float));
        a1 = t;          /* référence à la première moitié de t */
        a2 = t + n ;     /* référence à la seconde moitié de t */
}

void f1(int n, float * restrict a1, const float * restrict a2){
        int i;
        for(i=0;i<n;i++)
                a1[i] += a2[i]; /* a1 et a2 ne référencent pas les mêmes zones mémoires */
}
```

4) Pointeurs

Avec l'opérateur indirect * on indique qu'on a un pointeur sur une variable d'un type donné. Exemple : int *p_nb

Pour que le pointeur contienne l'adresse de la variable, on utilise l'opérateur d'adresse &. Exemple : p_nb = &nb

Quand l'opérateur indirect * précède un pointeur, il fait référence à la variable pointée.

Exemple : printf("%d",*p_nb); est équivalent à printf("%d",nb);

L'adresse d'une donnée est celle du premier octet qu'elle occupe. Exemple :

```
int entiers[3];
float flottants[3];
double precis[3];
printf("\t\tInt\t\tFloat\t\tDouble");
for(int x=0;x<3;x++)
        printf("\nElement %d:\t%ld\t\t%ld\t\t%ld",x,&entiers[x],&flottants[x],&precis[x]);
```

Avec un résultat possible à l'exécution :

	Int	Float	Double
Element 0:	1392	1400	1410
Element 1:	1394	1402	1418
Element 2:	1396	1404	1424
[clôture]	1398	1408	1432

On peut utiliser les pointeurs arithmétiques pour parcourir des tableaux. Si le pointeur nous envoit sur le premier élément du tableau, alors on peut l'incrémenter pour le faire passer sur l'élément suivant.

Autrement dit, si p_nb pointe sur un élément de tableau de type int (2 bits), alors p_nb++; va incrémenter le pointeur de 2 afin qu'il pointe sur l'élément suivant.

De la même façon, p_nb+=4; va le faire avancer de 4 éléments et donc de 8 bits.

Autrement dit, le pointeur modifie sa valeur automatiquement selon la taille des données pointées.

On peut faire deux autres types d'opérations sur les pointeurs :

- la différence. p_tab2 - p_tab1 donne le nombre d'éléments qui séparent les deux pointeurs. Si l'un pointe sur le début du tableau et l'autre sur la fin, on obtient le nombre d'éléments du tableau.
- une comparaison classique, comme !=, >, >=, <, <=, ==.

Le nom d'un tableau équivaut à un pointeur sur le premier élément. Par exemple, pour afficher tous les éléments d'un tableau, on peut faire :

```
int tab[5];
for(int i=0;i<5;i++)
        printf("Element %d: %d",i,*(tab+i)); // *(tab+i) est équivalent à tab[i]
```

Voire même :

```
int tab[5];
int *p_tab = tab;       /* car le nom du tableau seul est un pointeur de type entier ici */
for(int i=0;i<5;i++){
        printf("Element %d: %d",i,*p_tab);
        p_tab++;
}
```

On peut donc proposer une implémentation naïve d'une pile :

```
/* pile.h */
#define SIZE 10
int pile[SIZE];
int *init,*courant;
typedef int bool;
bool estVide();
bool estPlein();
void put(int o);
int get();

main(){
        init = courant = pile;
        /* Série d'opérations ... */
        return 0;
}

int get(){
        if(estVide()){
                printf("l'adresse %ld n'est pas dans le tableau",&courant);
                return 0;
        }
        int o = *courant;
        courant--;
}
```

Ceci illustre simplement les déplacements d'un pointeur, mais ne doit pas être utilisé. On se déplace sur des zones mémoires sans avoir réservé…

```
bool estVide(){ return courant==init; }
bool estPlein(){ return (courant-init == SIZE); }

void put(int o){
        if(estPlein()) return;
        *courant = o;
        courant++;
}
```

Pour passer des tableaux à une fonction, il est simple de transmettre le tableau lui-même, mais on va avoir un problème : on ne connait pas sa taille. On peut donc en profiter pour transmettre sa taille dans le même élan. Exemple :

```
triBulle(int tab[], int nbEl){
/* int tab[] équivaut à int *tab puisque tab est un pointeur de type int vers le premier élément */
        for(int i=0;i<nbEl;i++){
                for(int j=0;j<nbEl-1;j++){
                        if(tab[j]>tab[j+1])
                                int tmp=tab[j];
                                tab[j]=tab[j+1];
                                tab[j+1]=tab[j];
                        }
                }
        }
}
```

5) <u>Chaines de caractères</u>

Une chaîne est un tableau de caractères qui se termine par le caractère nul \0.
Exemple : char chaine[6]={'H','e','l','l','o','\0'}
On peut aussi donner la chaîne littérale avec les guillemets : char chaine[6]="Hello" ; le caractère nul est automatiquement rajouté.
Enfin, le compilateur peut-même calculer automatiquement la taille : char chaine[]="Hello".

Si la chaîne doit s'étendre sur plusieurs lignes, on termine chaque ligne par \ : « _____\
_____ »

On peut aussi utiliser des symboles particuliers dans la chaîne : « bonjour\ta\0x41oi ».

Principales fonction de manipulation des chaînes de caractère :

- char *strcpy(char *but, const char *source)
  /* Copie la chaine d'adresse source à l'adresse but, y compris le \0 de fin.
  Fournit en retour l'adresse but. */
- char *strncpy(char *but, const char *source, size_t longueur)
  /* Copie au maximum 'longueur' caractères de la chaine d'adresse source à l'emplacement d'adresse but, en complétant par des caractères de code nul si la longueur maximale n'est pas atteinte. Si la longueur est atteinte, il n'y aura pas \0 */
- char *strcat(char *but, const char *source)
  /* Recopie la chaîne d'adresse source avec son \0 à la fin de la chaîne d'adresse but, c'est-à-dire à partir de son zéro de fin qui se trouve donc remplacé par le premier caractère de la chaîne d'adresse source. Fournit en retour l'adresse but. */
- char *strncat(char *but, const char *source, size_t longueur)
  /* Idem que ci-dessus en recopiant au maximum 'longueur' caractères de source */
- int strcmp(const char *chaine1, const char *chaine2)
  /* Compare les chaînes en se basant sur la valeur du code de leur caractère. Fournit :
  une valeur négative si chaine1 < chaine2
  une valeur positive si chaine1 > chaine2
  zero si chaine1 = chaine 2 */
- int strncmp(const char *chaine1, const char *chaine2, size_t longueur)
  /* pareil en limitant la comparaison à un maximum de 'longueur' caractères */

9. Erreurs à ne pas commettre dans les chaînes de caractère

```
int n;          /* Interdit car la taille du tableau doit-être connue      */
int T[n];       /* statiquement, i.e. avant l'exécution du programme */
```

```
int[] f(int n){
        int T[n] ;
        ...
        return T ;
}
```

```
int T[10] ;
int f(int T[]){
        T[1] = ...
}
```

Détail de l'appel :

```
int main(...){
  int T[10] ;
  f(T) ;   /* effets de bord */
```

```
char s[10] = « bonjour » ;
for(i=0 ;i<strlen(s) ;i++) { ... }
```

strlen va chercher le '\0' à chaque fois.
La complexité est donc en n², très mauvais.

while(s[i] != '\0') OU for(i=0 ; i<10 & s[i] != '0'; i++) si pas sûr que chaine caractère

V. Expressions

Une expression doit respecter des règles de construction sur le nombre d'opérandes qu'accepte un opérateur (i.e. son arité), et les types autorisés pour chaque opérande.

1) Incrémentation

Soit le code suivant :
x=10; y = x++;
Alors x vaut 11 et y vaut 10. C'est une incrémentation postfixe.
Tandis que dans le code x=10; y = ++x; on aura x et y à 11. Incrémentation préfixe.

2) Tests

Le 0 représente "faux", tous ce qui n'est pas à 0 est donc vrai.
Autrement dit : (expression) est équivalent à (expression != 0)
(!expression) est équivalent à (expression == 0)

L'opérateur de condition est le seul opérateur ternaire. Sa syntaxe est : exp1 ? exp2 : exp3;
Si exp1 est vrai, on fait exp2 ; sinon, exp3.
Donc z = (x>y) ? x : y ;
Est équivalent à :
if (x>y) z=x;
else z=y;

Opérateur	Symbole	Exemple	Associativité
indexation	[]	t[i][j]	→
champ de structure	. →	c.reel, p → imag	→
appel de fonction	( )	max(tab)	→
négation logique	!	!x	←
négation bit à bit	~	~x	←
incrémentation	++	i++, ++i	←
décrémentation	--	i--, --i	←
moins unaire	-	-2.0	←
plus unaire	+	+2.0	←
coercition	(type)	(char *)	←
taille	sizeof	sizeof(int), sizeof(p)	←
adresse de	&	&x	←
indirection	*	*p	←
multiplication	*	a*b	→
division	/	a/b	→
modulo	%	a%b	→
addition	+	a + b	→
soustraction	-	a - b	→
décalage à gauche	<<	x << 4	→
décalage à droite	>>	x >> 4	→
opérateurs relationnels	<, <=, >, >=	n < 3, a >= 1	→
opérateurs d'(in)égalité	==, !=	a == 5, b! = 3	→
et bit à bit	&	a&b	→
ou exclusif bit à bit	^	a^b	→
ou bit à bit	|	a|b	→
et logique	&&	a&&b	→
ou logique	||	a||b	→
expression conditionnelle	? :	a(x > 0)?x : -x	←
affectation simple	=	a = 5	←
affectation étendues	+=, -=, *=, /=, %= &=, ^=, |=, <<=, >>=	x+ = 1 x << 1	←
composition d'expressions	,	a = 2, b = a + 3, z+ = 5	→

3) La virgule

Dans certains cas, la virgule agit comme un opérateur. On peut former une expression en séparant deux sous-expressions par une virgule. Ce qui a pour résultat :
- évaluation des deux expressions en commençant par celle de gauche
- l'expression entière prend la valeur de la sous-expression de droite

Exemple : x=(a++,b++);
En résultat, on attribue la valeur de b à x, puis on incrémente a, et on incrémente b.

On peut aussi faire un peu n'importe quoi : i = (j=2,1); au lieu de i=1; j=2

4) Instructions continue, goto

continue provoque l'arrêt de l'itération courante et le passage au début de l'itération suivante.
Exemple :

```
for(int i=0; i<N; i++){
        if((t[i]%2)==0) continue;
        sum += t[i];
}
```

Autre exemple, qui affiche tous les caractères qui ne sont pas des voyelles minuscules :

```
char buffer[81];
gets(buffer);
for(int i=0; buffer[i] != '\0'; i++){
        if(buffer[i]=='a' || buffer[i]=='o' || buffer[i]=='e' || buffer[i]=='i' || buffer[i]=='u')
                continue;
        printf("%c",buffer[i]);
}
```

On peut aller par l'instruction goto sur une étiquette positionnée dans la fonction. Exemple :

```
debut:
puts("Voulez-vous terminer ?");
scanf("%c",&reponse);
if(reponse=='o') goto fin;
goto debut;
fin:
puts("Bye !");
```

5) L'opérateur sizeof

Cet opérateur nous donne la taille en octets d'éléments primitifs (un entier...) ou plus complexes (un tableau).

Exemple :

```
int tableau[100];
printf("Un entier fait %d octects",sizeof(int));
printf("\ndonc un tableau de 100 entiers fait %d",sizeof(tableau));
```

6) Allocation dynamique

Les variables du tas (heap) sont des objets alloués dynamiquement au travers de la fonction malloc ; le programmeur gère entièrement leur cycle de vie.

La fonction malloc est définie dans stdlib par void *malloc(size_t taille).
On retourne un pointeur sur le début de la zone allouée en cas de succès et NULL sinon.

Il y a d'autres fonctions importantes pour l'allocation :

- calloc(size_t nbElements, size_t tailleElement) initialise à 0 chacun des octets alloués.
- realloc(void *ptr, size_t taille) change la taille de la zone allouée au pointeur ptr.
  Il y a allocation, recopie et libération. Le contenu de la nouvelle zone n'est pas initialisée.
  La valeur de ce pointeur est supposé avoir été obtenu par malloc, calloc ou realloc.
- Free(void *ptr) permet de libérer l'espace correspondant à l'adresse ptr.

Exemple de réservation en mémoire d'un tableau de 20 entiers, où chaque case allouée est à 0 :

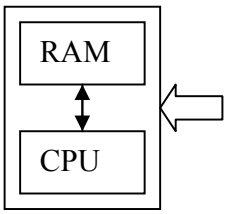
```
int *tableau ;
tableau = (int *)calloc(20,sizeof(int)) ;
```

Création dynamique d'une matrice de taille n :

```
int **m ;
mat=(int **)malloc(n*sizeof(int *));          /* C'est un pointeur de pointeurs ...*/
for(i = 0; i<n; i++)
        mat[i] = (int *)malloc(n*sizeof(int));      /* Et on initialise chacun des pointeurs. */
```

VI. Les entrées/sorties

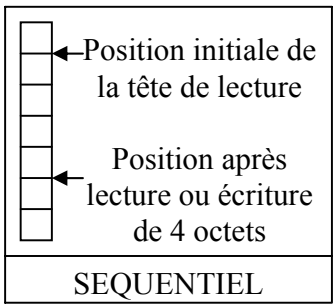
1) Généralités



On cherche à échanger de l'information entre le monde extérieur (disque, périphériques, cartes sons, imprimantes, réseau…) et la mémoire.

On utilise systématiquement le concept de fichier (fichier ordinaire pour le disque dur, driver pour le matériel comme /dev/audio, socket pour le réseau). Le fichier ordinaire est une entité logique qui sert d'argument à toutes les opérations d'entrée/sortie.

On peut considérer le fichier comme une borne magnétique avec des successions d'octets. D'où :
- Le fichier n'a pas de structure. Exemple : dans le traitement d'image, la première chose que l'application fait consiste à donner une structure au fichier.
- Il n'y a pas de marqueur de fin de fichier. Cette notion est issue de la taille.



Il y a deux modes d'accès :
- Séquentiel. L'accès suit l'ordre des octets. La tête de lecture avance (par les opérations de lecture/écriture) en passant par tous les octets.
- Direct. On fait sauter la tête de lecture à l'octet donné (utilisation de la fonction fseeks pour se déplacer en nombre d'octets). Surtout utilisé pour l'image, quand on veut sauter l'en-tête.

2) Printf, affichage de texte et de variables

Exemple : printf("\nBonjour utilisateur %d %s", numero, nom);
On a plusieurs instructions de contrôles disponibles : \n (retour chariot), \t (tabulation), \\ (antislash), \a (bip), \b ('back', retour arrière), \" (guillemet).
Lorsqu'on veut afficher des variables, on spécifie les conversions.

Le tableau pour la correspondance est le suivant :

Conversion	Signification	Types convertis
%c	Un caractère	char
%d	Entier décimal signé	int, short
%ld	Idem, mais en long	long
%f	Nombre virgule flottante	float, double
%s	Chaine de caractères	Tableau de char (String)
%2s	Chaine "contrainte"	Tableau de char contraint à une dimension de 2
%u	Entier déc. non signé	unsigned int, unsigned short
%lu	Idem, mais en long	unsigned long

Exemple de code :

```
int a=6;
char[] chaine="Bonjour";
printf("%s chez vous, numero %d",chaine,a);
```

Attention : le retour chariot n'est pas automatique avec printf. Si on ne vide pas le buffer soi-même, le texte peut ne pas s'afficher (exemple : printf sans \n en fin de main, le texte n'y est pas).

3) puts, affichage de texte seul

Une fonction d'affichage élémentaire, donc faible consommation de ressources par rapport à la précédente. Exemple : puts("Hello, world");

Le retour chariot est automatique en fin de chaîne.

Ou un exemple plus long avec des string :

```
char *msg1="Bonjour";
char *msg2="à tous";
puts(msg1);
puts(msg2);
/* résultat, affichage de Bonjour à tous */
```

4) scanf, lecture de données

On peut lire une ou plusieurs données avec cette fonction. Comme pour printf, dans la première partie on spécifie la conversion, et dans la deuxième partie on expédie les données aux variables concernées. Attention toutefois, scanf détache les variables lorsqu'il rencontre des blancs !

Exemple :

```
int nb;
char c;
printf("Entrez un nombre puis un caractere");          /* sous-entendu : séparés par des blancs */
scanf("%d %c",&nb,&c);
```

Autre exemple :

```
scanf("%3s%3s%3s",s1,s2,s3);
```

Utilisé en entrant septembre, on récupérera s1 = sep, s2 = tem, s3 = bre.

5) gets, lecture d'une chaine de caractères

Cette fonction lit la chaine de caractères tapées par l'utilisateur, blanc compris, jusqu'à ce que l'utilisateur appuie sur entrée.

Exemple :

```
char input[81];
printf("Saisissez le texte\n");
gets(input);
```

Il peut être intéressant de tester une ligne blanche dans une boucle de saisie :

```
char input[81], *ptr;
while (*(ptr=gets(input)) != NULL)
        printf("Vous avez tapé %s\n", input);
puts("Vous avez donné une ligne blanche");
```

6) Notion de flot

Le flot corrsepond en C au type FILE * var; défini dans stdio.h. Il y a 3 flots standards : stdin (initialisé vers le clavier), stdout (sortie écran habituelle), stderr (sortie écran signalant une erreur).

Pour toutes les fonctions utilisant un flot, il faudra le préciser. Exemple : fprintf(stdout, «…», ..), fputs, fgets, fscanf…

7) Utilisation élémentaire des fichiers

```
FILE *fich ;                                    /* cette variable pointeur représente le flux */
fich = fopen(« c:\\donnees\\essai », « wt ») ;   /* NULL si l'ouverture a échouée */
```

/* On associe le fichier au flux. On l'ouvre en mode 'write text' ce qui correspond à une ouverture en mode texte et en écriture (si le fichier n'existe pas, il sera crée, s'il existe déjà on remplace son contenu) */

```
int id ;
char nom[80] ;
fscanf(fich, « Utilisateur %d : %s », &id, &nom) ; /* Lecture dans le fichier */
fprintf(fich, « -> Le numero %id correspond a %s », id, martin) ; /* On écrit dans le fichier, la tête se déplace */
fseek(fich, 0, SEEK_SET) ;
```

/* place le pointeur de fichier sur un octet demandé. On spécifie le flux, puis la distance par rapport au point qui nous intéresse parmi SEEK_CUR (position actuelle), SEEK_END (fin du fichier), SEEK_SET (début du fichier). Ici on s'est donc remis au début */

```
fputs(« blablabla », fich) ;    /* on écrit blablabla au début du fichier. Attention, contrairement à puts, le caractère de fin de ligne n'est pas mit automatiquement avec fputs ! */
fclose(fich) ;        /* fermeture du flot ; gestion des données 'pending' : les données en attente d'écriture vers le buffer seront transférées vers leur destination avant la fermeture finale */
```

Pour connaître la taille d'un fichier :

```
FILE *fich ;
fich = fopen(« c:\\test », « wt ») ;
fseek(fich, 0, SEEK_END);
long taille;
taille = ftell(fich);
/* ftell donne la position courante en octets par rapport au début du fichier*/
```

Accès	Paramètre	Position	si le fichier existe	si il n'existe pas
lecture	"r"	début	.	erreur
écriture	"w"	début	mis à zéro	création
concaténation	"a"	fin	.	création
lecture	"r+"	début	.	erreur
et	"w+"	début	mis à zéro	création
écriture	"a+"	fin	.	création
Suffixe b : entrée/sortie binaire				

Sous UNIX, le suffixe b est ignoré car il n'existe aucune différence entre un fichier binaire et un fichier de données quelconques.

Compter le nombre de lignes :

```
FILE *fich ;
fich=fopen(« C:\\test », « wt ») ;
char c = getc(fich);
int compteur = 0;
while(c!=EOF){ if(c=='\'){ compteur ++ ; } }
```

Gestion générale des fichiers :

```
if(rename(« C:\\test.txt », « C:\\test ») == 0){ puts("le fichier a ete renomme !"); }
if(remove("C:\\test" != 0){ puts("Impossible de supprimer le fichier"); }
```

Nom de la fonction	*Effets*	*Nom de la fonction*	*Effets*
fopen(char * fichier)	Ouverture de fichier	fseek(flux, nombre d'octets , pos. init.)	Se placer à une position spécifique
fclose(char * fichier)	Fermeture du fichier	ftell(flux)	Position courante
fscanf(flux,format,var)	Lecture dans le fichier	getc(flux)	Lecture d'un char
fprintf	Ecriture dans le fichier	rename(old, new)	Renommer
fputs	idem, mais juste une chaine	remove(char * fich)	Supprimer

On peut redéfinir un flot déjà initialisé en changeant le fichier associé (i.e. une redirection) :

```
FILE *fp ;
[…]
fp = freopen(« pgm.log », « w », stdout);
/* on considère maintenant que le flot stdout est le fichier pgm.out ; i.e. toutes les sorties seront redirigées vers ce fichier de log */
```

On peut aussi vider les buffers associés à différents fichiers.

```
fflush(fp) ;
/* On vide le buffer associé à fp, si ce fichier a été ouvert en écriture ou en mise à jour.
En cas d'échec on renvoit EOF. En cas de succès on renvoit 0 */
```

Utilisations particulières :

- fflush(stdin) : l'utilisateur a écrit quelque chose au clavier, peut-être trop par rapport à ce qui est demandé, on remet le buffer à 0.
- fflush() : on vide tous les buffers disponibles

Ecriture caractère par caractère :

```
int c ;     /* la valeur donnée par getc est celle du code ASCII du caractère */
FILE *fi, *fo ;
[…]
while((c=fgetc(fi)) !=EOF){
        fputc(c, fo);
}
```

La fonction feof permet de regarder si on est arrivé à la fin du fichier. La fonction ferror teste l'indicateur d'erreur et rend une valeur différente de 0 si une erreur s'est produite. Exemple :

```
int c ;
c = getchar() ;
while( !ferror(stdin) && !feof(stdin)){
        putchar(c);                     /* on réalise un écho */
        c = getchar();
}
```

On peut utiliser fgets et fputs pour lire et écrire des chaînes de caractère dans un flux.

```
FILE *fp ;
char buf[80] ;
if((fp = fopen(« echo.c », « r »)) = NULL){
        perror("fopen");   /* Report d'erreur */
        exit(1); }
fgets(buf, sizeof(buf), fp);
while(!ferror(fp) && !feof(fp)){
        fputs(buf, stdout); /* ou printf("%s", buf; */
        fgets(buf, sizeof(buf), fp) ;
}
fclose(fp);
return EXIT_SUCCESS;
}
```

Pour une lecture formatée dans un fichier on utilisera fscanf, exemple :

```
FILE *fp ;
fp = fopen(« C:\\test.txt");
int nombre;
fscanf(fp, "%d", &nombre);
```

```
int i;            scanf("%d",&i);     printf("%d",i);   décimal
int i;            scanf("%o",&i);     printf("%o",i);   octal
int i;            scanf("%x",&i);     printf("%x",i);   hexadécimal
unsigned int i;   scanf("%u",&i);     printf("%u",i);   décimal
short j;          scanf("%hd",&j);    printf("%d",j);   décimal
short j;          scanf("%ho",&j);    printf("%o",j);   octal
short j;          scanf("%hx",&j);    printf("%x",j);   hexadécimal
unsigned short j; scanf("%hu",&j);    printf("%u",j);   décimal
long k;           scanf("%ld",&k);    printf("%d",k);   décimal
long k;           scanf("%lo",&k);    printf("%o",k);   octal
long k;           scanf("%lx",&k);    printf("%x",k);   hexadécimal
unsigned long k;  scanf("%lu",&k);    printf("%u",k);   décimal
float l;          scanf("%f",&l);     printf("%f",l);   point décimal
float l;          scanf("%e",&l);     printf("%e",l);   exponentielle
float l;                              printf("%g",l);   la plus courte des 2
double m;         scanf("%lf",&m);    printf("%f",m);   point décimal
double m;         scanf("%le"&m);     printf("%e",m);   exponentielle
double m;                             printf("%g",m);   la plus courte
long double n;    scanf("%Lf"&n);     printf("%Lf",n);  point décimal
long double n;    scanf("%Le"&n);     printf("%Le",n);  exponentielle
long double n;                        printf("%Lg",n);  la plus courte
char o;           scanf("%c",&o);     printf("%c",o);   caractère
char p[10];       scanf("%s",p);      printf("%s",p);   chaîne de caractères
                  scanf("%s",&p[0]);
```

On peut vouloir ne prendre ses variables que dans une certaine partie des données lues. Pour ça, on utilise de séquences d'échappement : on absorbe des choses sans en tenir compte. Exemple :

```
while(l){
        printf(« Entrez un entier : ») ;
        if((args = scanf(« %d », &x)) == 0){
                printf("Erreur : pas un entier");
                scanf(« %*[^\n] ») ;        /* Aller jusqu'à la fin de ligne */
                scanf(« %*l[\n] ») ;        /* Sauter une ligne */
        } else {
                if(args == l)
                        printf(« Read in %d\n », x) ;
                else
                        break;
        }
}
```

La syntaxe des séquences d'échappement est %*<NOMBRE><CARAC>. Signification :

- * : la prochaine unité du flot d'entrée doit être ignorée
- nombre : longueur maximale de l'unité à reconnaître
- carac : l'interprétation de la prochaine unité (d : décimal, o : octal, x : héxadécimal…).

[: Dans la chaîne format, ce caractère introduit une séquence particulière destinée à définir un scanset. la séquence est formée du caractère [, suivi d'une suite de caractères quelconques, suivi du caractère].
Si le premier caractère après le crochet ouvrant n'est pas le caractère ^ , le scanset est l'ensemble des caractères entre crochets.
Si le caractère ^ est immédiatement après le crochet ouvrant, le scanset est l'ensemble des caractères ne se trouvant pas dans la chaîne entre crochets.
fscanf lit dans le fichier repéré par file_ptr jusqu'à la rencontre d'un caractère ne faisant pas partie du scanset.
Comme dans le cas du format c, il n'y a plus de notion de caractère séparateur, on ne considère que le scanset.
parami est interprété comme un pointeur vers une suite de caractères dans lesquels on met les caractères lus.

On peut vouloir examiner le prochain caractère que l'on rencontrera dans le flot, mais accéder à celui-ci revient à l'enlever du flot. On utilise donc la fonction int ungetc(int c, FILE *stream) pour replacer le dernier caractère lu dans le flot. Exemple :

```
#include <stdio.h>
#include <ctype.h>
void skip_whitespace(FILE *stream){
        int c;
        do{
                c = getc(stream);
        }while(isspace(c));
        ungetc(c, stream);
}
```

VII. Fonctions à nombre variable de paramètres

Une fonction avec paramètres variables doit avoir au moins un paramètre fixe (i.e. de type connu). Les paramètres variables sont alors parcourus comme une liste, le parcours débutant à partir du dernier paramètre fixe.

On utilise des pseudo-fonctions définies dans <stdarg.h> :

- va_list : déclaration de la structure de données utilisée pour parcourir la liste
- va_start : initialisation du parcours sur le premier paramètre variable
- va_arg : obtention d'un paramètre et passage au suivant
- va_end : terminaison du parcours

```c
#include <stdio.h>
#include <stdarg.h>

int max(char c, int first, ...) { /* LES ... signifient "arité variable" */
  va_list paramcourant;  /* structure permettant le parcours de la liste */
  int m=0, next=first;

  /* Initialisation de la structure */
  va_start(paramcourant,first);
  /* paramcourant repere l'adresse, dans la pile, du premier
     argument de la zone des parametres variables  */

  while(next>0) { /* On a un moyen de stopper dans la liste */

    if (next>m)
      m=next;    /* Calcul du max */

    /* On avance en precisant le type de ce qui est absorbé */
    next = (int)va_arg(paramcourant,int);
  }
  va_end(paramcourant); /* Destruction de la va_list */

  return m; /* On renvoit le max */
}
```

XII. Structures

Une structure contient plusieurs variables groupées sous le même nom, c'est en quelques sortes le pendant d'un objet, sans les méthodes propres.

```
struct coord {
        int x;
        int y;
};
```

On déclare une structure nommée "coord". On peut créer les deux structures dans la foulée :

```
struct coord {
        int x;
        int y;
} premier, second ;     /* Deux structures "coord" nommées "premier" et "second" sont créées */
```

On peut aussi les déclarer ultérieurement :

```
struct coord{
        int x;
        int y;
};
struct coord premier, second;
```

Voire même utiliser typedef :

```
typedef struct coord{
        int x;
        int y;
};
coord premier, second;
```

L'accès est simple, avec un point. Exemple : premier.x = 30.
Une structure peut contenir d'autres structures, comme un rectangle avec deux points :

```
struct rectangle{
        struct coord hautgauche;
        struct coord hautdroite;
};
```

On peut aussi faire des tableaux de structures. Exemple, une structure pour un nom. On en fait un tableau, c'est-à-dire un agenda :

```
struct entree{
        char nom[10];
        char prenom[12];
};
struct entree agenda[1000];    /* Un tableau de 1000 éléments */
```

Rien n'empêche d'utiliser des pointeurs dans les structures. Une autre façon de faire pour l'exemple précédent :

```
struct entree{
        char *nom;
        char *prenom;
}
```

Etudions l'exemple d'un pointeur vers une structure :

```
typedef struct exemple{
        int variable;
} hop;
exemple *p_exemple; /* création du pointeur */
p_exemple = &hop;    /* initialisation */
(*p_exemple).variable = 100; /* accès par opérateur indirect et pointeur à la variable */
p_exemple -> variable = 60;
printf("Valeur de variable : %d",p_exemple -> variable);
```

Les trois façons suivantes pour accéder sont donc équivalentes :
hop.variable (*p_exemple).variable p_exemple->variable

XIII. Unions

On déclare une union de la même façon qu'une structure. Le changement se situe sur le stockage en mémoire : les membres d'une union sont stockés au même emplacement ; la taille de l'emplacement mémoire réservé étant donc celui du plus grand membre. On ne peut donc pas accéder à plusieurs membres en même temps. Autrement dit, l'union ne peut recevoir qu'une valeur à la fois.

```
union exemple{
        char c;
        int i;
} hop;
hop.c = { '@' };
printf("char : %c ; nombre : %d",hop.c, hop.i);
/* En résultat on affichera bien le caractère @. Par contre, la valeur de i dépend de la machine. En effet, il partage son emplacement en mémoire avec c, et on vient de donner une valeur à c, ce qui a modifié l'emplacement mémoire. */
hop.i = 32000;
printf("char : %c ; nombre : %d",hop.c, hop.i);
/* Cette fois, le caractère sera incohérent puisqu'on vient de l'écraser en mettant le nombre. Le nombre lui sera bien 32000. */
```

Une utilisation pratique d'une union :

```
union date{
        char date_complete[9];
        struct partie_date{
                char mois[2];
                char separateur1;
                char jour[2];
                char separateur2;
                char annee[2];
        } partie;
} exemple = {"27/09/86"};
```

Autre utilisation pratique :

```
struct generic_tag{
        char type;
        union shared_tag{
                char c;
                int i;
                float f;
        }shared;
}exemple;
exemple.type = CARACTERE;
exemple.shared.c = '$';
/* Si on examine exemple à ce moment là, on sait qu'il travaille avec un caractère. */
exemple.type = FLOAT;
exemple.shared.f = 32.015;
/* Et maintenant, on saurait qu'il travaille avec un nombre à simple précision. */
```