Langage C

Pascal Fares : ISAE Cnam Liban ©

October 11, 2014

pfares@cofares.net

Part I

Le C

Le langage C est un langage de programmation impératif et structuré permettant de définir des programmes pour des applications très diverses. Le langage C est donc un langage de programmation très général. En ce sens il appartient à la famille des langages dits universels comme Algol, Pascal, Ada, etc... La spécificité du langage C vient de sa définition proche d'une structure de machine et de son traitement des pointeurs (adresse mémoire). Il comprend aussi des notions de compilation conditionnelle et de traitement de macro-instruction qui sont essentielles dans la programmation de projet important. La structure interne de la mémoire centrale d'un ordinateur comprend des assemblages fixes de bits et une interprétation de ces regroupements. Ainsi nous avons dans la plupart des cas des octets, des mots et des mots longs. Leur interprétation peut étre caractère, entier, flottant ou pointeur (adresse). Ces éléments fondamentaux sont manipulés par les instructions de la machine; ils formeront les outils de base du langage C. Un programme C se réduit toujours à un ensemble de fonctions non imbriquées.

Chapter 1

Les déclarations

1.0.1 La définition des éléments

Le langage C est un langage impératif déclaratif. Tout élément doit étre déclaré avant d'étre utilisé. Une seule exception à cette règle concerne les fonctions. Une fonction peut étre déclarée mais elle peut également étre définie par défaut.

Chaque nom d'un élement est composé de chiffres ou de lettres et doit commencer par une lettre. Le caractère souligné "_" est considéré comme une lettre. Le nombre de caractères significatifs dépend du compilateur. Historiquement seuls les huit premiers caractères du nom sont pris en considération; les autres caractères sont ignorés (vérifier la documentation du compilateur que vous utilisez). Un nom ne doit pas étre un mot clé; il existe 28 mots clés comme int, static, integer, etc....

Les élements de bases des variables sont au nombre de sept et correspondent aux différentes représentations machines (voir table 1.0.1):

char	caractère (octet)	8 bits
short	entier court signé	16 bits
int	entier signé	32 bits
long	entier long signé	64 bits
unsigned	entier non signé	
float	flottant simple précision	32 bits
double	flottant double précision	64 bits

Table des types et leur taille probable

Ces éléments représentent les variables élémentaires de la mémoire. Ils correspondent également à une certaine interprétation. La distinction sur la longueur (nombre de bits) d'un entier dépend de la machine sur laquelle le langage est implanté. Elle dépend également du compilateur. Le langage précise simplement qu'un entier long est de taille supérieure ou égale à un entier court. Pour les machines à octets un choix fréquent est celui donnée en (Tab 1.0.1):

Char et int sont des variables arithmétiques alors que "float" et "double" sont des variables flottantes. On peut définir des entiers non signés sur toutes les longueurs:

unsigned short
unsigned long.

Le langage ne prévoit rien sur la nature signée ou non signée du caractère (octet). Cela dépend de la machine (extension ou non du bit de signe).

Les constantes associées à ces variables sont définies de manière naturelle:

- un entier est une suite de chiffres précédée éventuellement d'un signe.
- un flottant utilise le point pour commencer la partie décimale et la lettre "e" ou "E" pour l'exposant: +1234.56e-7
- un caractère est mis entre apostrophes: 'a'
- certains caractères ne sont pas représentables. On utilise alors une forme spécifique pour déterminer leur code: '\ddd' où "ddd" est le code octal du caractère.
  Exemple: '\014' est le caractère "form feed" '\040' est le caractère espace " "
  Certains caractères sont prédéfinis:
  n nouvelle ligne (line feed)
  t tabulation
  b espace arière (back space)
  r retour chariot (carriage return)
  f nouvelle forme (form feed)
  \\ caractpre \
  \' caractère '
  Le caractère '**0**' représente le caractère nul (le nombre 0).
- Une constante entière peut étre écrite en octal ou en hexadécimal. Si le premier chiffre d'une constante entière est 0 alors cette constante est en octal ou en hexadécimal. Si le caractère "x" ou "X" suit le 0 alors l'entier est en hezadécimal; sinon il est en octal:
  10 dix en base dix
  012 douze en base huit soit dix en base dix
  OxA A en base seize soit dix en base dix

Un pointeur est un entier de type indéfini (dépendant de la machine). On peut définir un pointeur sur n'importe quel élément. Le pointeur représente en fait l'adresse d'une donnée en mémoire.

Pour déclarer une variable on précise son type puis son nom. Si ce nom est précédé d'une étoile alors la variable est un pointeur sur un élément du type donné:

```
int I; // I est un entier.
short int I; //I est un entier court.
int *P; //P est un pointeur sur un entier.
```

Une variable peut étre initialisée par une constante ou une expression constante. Cette constante suit la définition de la variable et est placée éventuelleroent entre accolades :

```
int k = 2;
int l = { 3 };
int *p = &k; ( adresse de k ).
```

A partir de ces variables élémentaires il existe trois modes de constuction de variables complexes: le tableau, la structure et l'union.

1.0.1.1 Le tableau

Un tableau est un ensemble d'éléments de méme type. On définit un tableau par le nombre n de ces éléments et l'indice d'un tableau varie toujours de 0 à n-1:

int t[10]; Tableau de 10 entiers

Les éléments du tableau sont toujours rangés de maniére consécutive en mémoire et l'index d'adressage doit toujours être une valeur entière. Le nom du tableau est une constante pointeur et représente l'adresse du premier élément du tableau. On pourra ainsi écrire:

int *p;
p = t; affectation à P de l'adresse de t[0], qui est l'adresse du premier élément du tableau

Il est bien sûr possible de définir des tableaux à plusieurs dimensions. Dans ce cas les bornes respectives se suivent:

int T[10][8]; //tableau à deux dimensions.

Le nom du tableau est dans ce cas une constante pointeur sur un tableau:

int (*P)[8]; P = T; *P est un pointeur et pointe sur le sous-tableau T[0][]. **P est un entier, le premier élément T[0][0].

Remarque:

La définition "int **P" serait incorrecte. Elle signifierait pointeur sur un pointeur d'entier et serait donc équivalente à la déclaration "int *P[S]". I1 faut donc définir un pointeur sur un élément du type "T[i]". Cet élément est un tableau de 8 entiers. Les parenthèses de la définition sont donc obligatoires car les crochets sont prioritaires par rapport à l'étoile.

Les variations de l'indice dans l'ordre de rangement des éléments du tableau s'effectuent toujours de droite à gauche.

Un tableau peut étre initialisé. On définit l'ensemble des valeurs entre deux accolades:

int T[10] = { 0,1,2,3,4,5,6,7,8,9 };
int S[5][2] = { 0,1 , 2,3 , 4,5 , 6,7 , 8,9 }; variation des indices de droite à gauche.
ou
int S[5][2] = { {0,1},{2,3},{4,5},{6,7},{8,9}};

exemple:

```
main () {
      int S[5][2] = {0,1,2,3,4,5,6,7,8,9};
      int i,j;
      /**
        * afficher tous les éléments
        */
```

```
   for (i=0; i < 5; i++)
      for (j=0; j <2; j++)
         printf("S[%d][%d]=%d\n",i,j,S[i][j]);
}
/** résultat de l'execution
compiler : gcc testtab.c
lancer le programme ./a.out
S[0][0]=0
S[0][1]=1
S[1][0]=2
S[1][1]=3
S[2][0]=4
S[2][1]=5
S[3][0]=6
S[3][1]=7
S[4][0]=8
S[4][1]=9

*/
```

Un cas particulier de tableau est celui des chaînes de caractères. Une chaine de caractéres est un tableau de caractères à une dimension. I1 existe une forme abrégée pour définir les éléments du tableau d'une chaine de caractères: on place la chaîne entre deux guillemets. Le compilateur construit alors un tableau de caractpres contenant les caractères de la chaine et ajoute le caractère '\0" en fin de tableau. Les deux déclarations suivantes ont le même effet:

char CH[3] = { 'u','n','\0' };
char CH[3) = "un";

Remarque: Le termanateur de chaîne ('\0') est utilisé pour reconnaître la fin d'une chaîne, son absence est grâve des fonctions tel que `printf` ne saurait plus fonctionner.

Bien sûr un tableau est une constante pointeur. On pourra définir plus simplement la chaine comme:

char *CH="un";
ou
char CH[] = "un";

La définition d'une table de messages pourra prendre la forme d'une table de pointeurs sur des chaines de caractpres :

char *MES[3] = { "messagel","message2","message3" };

cette définition correspond 8 la définition:

```
char MES1[]="message1"; //premier tableau contenant "messagel"
char MES2[] = "message2";
```

```
char MES3[] = "message3";
char *MES[3] = { MES1, MES2, MES3 };
/** Les éléments de ce tableau sont des
  * pointeurs sur des caractères. Ils
  * peuvent étre initialisés avec les
  * constantes pointeurs MES1, MES2 et
  * MES3.
  */
```

1.0.1.2 La structure

La structure est un ensemble hétérogène d'éléments regroupés pour des raisons fonctionnelles. Une structure porte un nom qui identifie sa nature. Ce nom pourra servir à la définition de plusieurs variables de ce type. Une structure est introduite par le mot clé "struct" suivi du nom d'identification et enfin, entre accolades, de la description des éléments composant cette structure:

```
struct exp {
    int I;
    int *P;
    int TAB[10];
};
La structure "exp" est composée d'un entier, d'un pointeur sur un entier et d'un tableau de dix
```

La déclaration d'une structure ne définit aucune variable. Elle définit seulement la forme d'une structure et identifie cette forme par un nom. Dans l'exemple précédent la forme de cette structure porte le nom "exp". On peut alors définir des variables ayant cette forme:

```
struct exp A, *B, C[10];
A est une structure de type "exp".
B est un pointeur sur une structure de type "exp".
C est un tableau de dix structures de type "exp".
```

L'initialisation d'une structure est similaire à l'initialisation d'un tableau. Les éléments sont rangés de gauche à droite ou de haut en bas:

```
struct dex {
    int I;
    char A;
    short C[4];
};
struct dex E = { 2,'a',3,4,5,6 };
```

La déclaration d'une variable structure peut suivre directement la définition de type:

```
struct trx {
    int T[10];
    char L[S];
} F; /* définit une structure F de type trx. */
```

Les structures peuvent étre imbriquées.

On réfère à un élément d'une structure par le nom pointé:

```
E.I représente l'élément I de la structure E de type dex. F.L[0] ou F.(L[0]) représente l
```

Dans le cas ou il s'agit d'un pointeur sur une structure il faut utiliser le nom fléché "->" (symbole "-" suivi du symbole ">"):

```
B->I représente l'élément I de la structure pointée par B.
```

Ces expressions peuvent étre d'une complexité quelconque (limites imposées par le compilateur).

```
struct cexp {
    struct exp K[10];
    int A;
    struct exp *V;
};
struct cexp L,*R;
```

On peut alors avoir par exemple les constructions suivantes:

```
L.K[0].TAB[1]   deuxième élément du tableau "TAB" de la
                structure premier élément du tableau K
                de la structure L.
R->K[1].I       variable I de la structure deuxième élément du tableau K dans la structur
*(L.V->P) valeur de la variable pointée par le pointeur P de la structure pointé par la v
R->V->P variable P de la structure pointée par la variable V de la structure pointée par
```

Un cas particulier concerne les structures se référant à elles-mémes (type récursif):

```
struct cel {
    int R;
    struct cel *Suivant;
};
```

Ce type de structure permet de définir des listes de façon simple. Une cellule d'une liste du type précédent est constituée d'un entier et d'un pointeur sur la cellule suivante.

La structure suivante permet la définition d'un arbre binaire:

```
struct bin {
    int val;
    struct bin *gauche;
    struct bin *droite;
};
```

La construction de type structure sert également à définir des champs. Un champ est une suite continue de bits. Les champs doivent tous être définis à l'intérieur d'une variable élémentaire. Les bits de cette variable sont pris dans le même ordre mais le langage ne précise pas cet ordre (gauche-droite ou droite-gauche). Nous supposerons pour les exemples que l'on utilise l'ordre gauche-droite. Un champ est défini en précisant le nombre de bits qu'il contieni:

```
struct champ {
    unsigned prem : 4; /* les quatre premiers bits. */
    unsigned deux : 2; /* les deux bits suivants.   */
    unsigned fin : 16; /* et les 16 derniers        */
};
```

Une structure de ce type fait référence à 32 bits (regroupement minimum existant pour contenir les trois champs). Cette structure sera donc rangée dans un mot de 32 bits sur une machine classique. Dans le cas ou un int sur cette machine occupe 32 bits la structure sera rangée dans un "int". Dans le cas ou les "int" de cette machine serait aeulement de 16 bits cette structure serait rangée dans un tableau de deus "int": Le premier mot contiendrait les deux premiers champs et le deuxième mot le champ suivant. I1 ne doit pas y avoir de chevauchement entre les données élémentaires pour les champs). Un champ peut ne pas Atre nommé; dans ce cas on utilise les deux poinis suivis de la largeur du champ anonyme. Ils sont utilisés pour le remplissage. Ainsi dans le cas précédent et en supposant qu'un int occupe 16 bits la structure champ est équivalente H la structure:

```
struct champ {
    unsigned prem: 4;
    unsigned deux: 2;
    : 10;
    unsigned fin: 16;
};
```

Les champs sont des entiers. I1s peuvent étre utilisés comme tel.

1.0.1.3 L'Union

La mémoire est une suite de bits avec certains regroupements. Les regroupements peuvent étre interprétés de différentes façons. Ainsi un octet peut étre interprété comme un entier ou comme un caractère. Seul l'utilisation du regroupement détermine l'interprétation machine de ce regroupement. Les langages de programmations associent des noms à ces regroupements et définissent une interprétation par 1e type. Cette interprétation fixe associée à un regroupement peut étre quelquefois source de difficultés inexistantes dans la programmation assembleur. La définition d'union permet de définir une vision différente du méme espace mémoire. Les contraintes dues aux types sont donc ainsi levées. La définition d'une union suit le méme principe que celui d'une siructure mais chaque membre définit une interprétation du méme espace mémoire:

```
union exp {
    int I;
    char T[2];
}
```

Si un entier est sur 16 bits et un caractère sur 8 bits 1a définition précédente donnera deux interprétations de 16 bits mémoire: soit une suite de deux caractères, soit un entier signé. Les règles de traitement de l'union sont les mémes que celle de la structure. La zone mémoire occupée par une union est toujours la zone maximum correspondant à l'un de ces membres.

Exemple: Un registre sur le micoprocesseur 68000 possède 32 bits. Il peut étre utilisé sur 16 bits et sur 8 bits. On peut définir toutes les interprétations d'un registre:

```
traitement des bits:
struct bitrg {
unsigned bit31:1; unsigned bit30:1; unsigned bit29:1; unsigned bit28:1; unsigned bit27:1;
Définition du registre:
union reg {
    struct bitrg H;
    char CR[4];
    short SR[2];
    long LR;
};
ainsi avec la déclaration "struct reg R";
R.B.bit31 représentera le bit de poids fort.
R.CR[3] le caractère manipulé dans les instructions
        machines.
R.SR[2] le mot traité par les instructions machines.
R.LR le mot long ou le registre en entier.
```

1.0.2 La déclaration des fonctions

Il ne faut pas confondre la *définition d'une fonction* et sa *déclaration*. La déclaration indique le nom de cette fonction et précise la nature du résultat de cette fonction.

Dans le langage C toutes les fonctions produisent un résultat. L'utilisation d'un résultat n'est pas obligatoire et une procédure peut étre vue comme une fonction dont on utilise pas le résultat. Dans le langage C la déclaration des fonctions ne donne aucune indication sur les paramètres (nombre et nature) qui doivent être passés à cette fonction. Les paramètres seront définis seulement au moment de la définition de la fonction. La cohérence entre la nature des paramètres à l'appel d'une fonction et celle attendue par la fonction est de la responsabilité du programmeur. Il existe des programmes (lint) qui permettent de vérifier cette cohérence.

de même le compilateur gcc de GNU augmente C de la possibilité de signer les fonctions, il permet de préciser les paramètre et leurs type au moment de la déclaration. La norme initiale du langage C imposait qu'un paramètre soit de nature simple (contenu dans un mot mémoire). Les versions actuelles acceptent des extensions et permettent de passer une structure comme paramètre (dans la norme initiale il était seulement possible de passer un pointeur sur une structure, ce qui reste meilleur car consomme mois de place dans la pile). La même contrainte existe sur la nature de l'objet retourné par une fonction. Ces restrictions initiales ne sont pas des limitations car comme il est possible de transmettre dans les deux sens des pointeurs sur des objets des objets (le programme appelant ou appelé doit simplement comporter une recopie de l'objet pointé). *Ce paragraphe ne traite que de la déclaration des fonctions.* Une fonction définit une valeur d'un objet quelconque. On définit une fonction en plaçant deux parenthèses derrière son nom. La fonction doit étre précédée du type d'objet qu'elle produit:

un exemple:

```
int f(); /* signature d'une fonction renvoyant un entier */

int *g(); /* signature d'une fonction renvoyant un pointeur sur entier */

int w();
int (*k)(); /* pointeur sue une fonction renvoyant un entier */

/**
  * definition de la fonction f
  */
int f(int x, int y) {
    return (x+y);
}

int w(int (*s)(), int x, int y) {
   return ((*s)(x,y));
   /* dans gnu gcc on peur ecrite s(x,y) au lieu (*s)(x+y)
    * c'est peut-être plus clair!
    */
}

main () {
 k=f; /* on dit que k pointe sur la fonction f */

 printf("%d+%d=%d\n",3,5,k(3,5));
     /* l'utilisation de k est alors equivalente à f*/
 printf("%d+%d=%d\n",3,5,w(f,3,5));
}

/** resultat de l'execution (test)
compilation:
[pfares@pportable MPS]$ gcc testdecfonc.c
execution:
[pfares@pportable MPS]$ ./a.out
3+5=8
3+5=8
*/


     int f(); //fonction renvoyant un entier.
     int *f(); //fonction renvoyant un pointeur sur un entier.
     int (*f)(); pointeur sur une fonction renvoyant un entier.
                 La parenthèse est ici nécessaire. Elle précise
                 que l'étoile se rapporte à "f" et non à "int".
                 Donc au lieu d'avoir un pointeur sur "int" on a
                 un pointeur sur "f".
     int *(*f)(); pointeur sur une fonction renvoyant un pointeur
                  sur un entier.
```

```
struct exp *f(); fonction renvoyant un pointeur sur une
                 structure de type "exp".
```

Avec une fonction il est possible soit de l'appeler, soit de caluler son adresse.

Lorsqu'une fonction n'est pas déclarée elle retourne implicitement un entier. Lorsque l'on veut définir une procédure on définit une fonction et on n'utilise pas sa valeur. Certains compilateurs utilisent le type `"void"` pour indiquer que la valeur de la fonction ne doit pas étre utilisée.

Exemple de définition de fonctions:

Soit une fonction "somme" qui retourne la somme de deux éléments. Les écritures suivantes ont le méme effet:

```
x + y;      évaluation de la somme
somme(x,y); sans utilisation du résultat.
z = x + y;      évaluation de la somme
z = somme(x,y); avec utilisation du résultat.
```

Lorsque l'on fait référence à une fonction comme paramètre d'une autre fonction on utilise implicitement son adresse:

```
int f();
g(f);
```

Dans g() on définit le paramètre comme un pointeur sur une fonction:

```
g(P)
int (*p)();
{ ....
   (*p)(); appel de la fonction paramètre.
   ....
}
```

Bien sûr les pointeurs sur des fonctions peuvent apparaitre n'importe où comme membre d'une structure ou élément de tableau.

Exemple: Sur le microprocesseur 68000 il existe un tableau de vecteurs d'interruptions. Ce tableau définit pour chaque entrée une procédure d'exception. Nous pouvons le déclarer de la façon suivante:

```
   int (*exp)()[256];
ou
   void (*exp)()[256];
```

La déclaration de type : typedef

La pseudo-fonction "typedef" permet la définition de nouveau type. Elle est très utile pour les programmes importants. Cette fonction ne définit qu'une abzéviation et n'est pas associée à la définition du traitement du type cozrespondant comme dans le langage ADA ou C++ par exemple. Néanmoins elle est très employée dans les programmes volumineux et permet de classifier les différents objets utilisés (un nom par type d'objet). Le mot clé "typedef" précède la déclaration:

```
typedef int Now; Le mot "Now" devient un type nouveau
                 synonyme de "int".
```

On pourra alors écrire:

```
Now x,y; x et y sont deux entiers.
```

Bien sûr cette pseudo-fonction est valable quelle que soit la déclaration:

```
typedef struct {
    char NOM[8];
    int IX;
} TAB;
```

Le mot "TAB" définit un type d'élément qui peut étre utilisé pour former de nouveaux objets:

```
TAB T[100],*P: tableau de structure et pointeur sur une
               structure.
T[4].IX détinit l'entier IX de 1a structure en cinquième
        position dans le tableau T.
```

1.1 Les expressions

Une expression définit une valeur. La valeur d'une expression esi très souvent une valeur élémentaire (dans la norme initiale une expression définit toujours une valeur élémentaire). Pour définir une expression on utilise des variables, des constantes et des opérateurs. Une fonction est bien sdr considérée comme un opérateur. Ces opérateurs permettent de calculer une nouvelle valeur à partir d'un ensemble d'éléments.

Exemple: L'opérateur "&" détermine l'adresse de quelque chose. "&I" sera donc l'adresse de la variable I.

L'accès à une valeur élémentaire détermine une expression primaire. Une expression primaire utilise des opérateurs d'accés et est définie par:

- une constante.

- ⊢ un identificateur.
- ⊢ une chaine.
- ⊢ une expression entre parenthèses.
- ⊢ une expression primaire suivie d'une expression enire crochets.
- ⊢ une expression primaire suivie d'une suite d'expressions entre parenthpses.
- ⊢ une valeur primaire suivie d'un identificateur précédé d'un point.
- ⊢ une expression primaire suivie d'un identificateur précédé d'une flèche.

Dans chaque cas la signification est différente. Elle correspond l'interprétation suivante:

- ≙ constante : constante.
- ≙ identificateur : valeur de la variable définie par cet identificateur.
- ≙ chaine : une chaine est un tableau de caractères. La valeur associée à une chaine est donc un pointeur sur le premier caractère de la chaine.
- ≙ (expression) : valeur de l'expression.
- ≙ expression_primaire [expression] : accès à un tableau. l'expression E1[E2] à la méme signification que *((E1) + (E2))
- ≙ expression_primaire(liste d'expression) : appel de fonction.
- ≙ valeur_primaire.identificateur : accès à un élément de structure.
- ≙ expression_primaire->identificateur : accès à un élément d'une structure pointée.

L'évaluation d'une expression utilise toujours les mémes formes de données. I1 y a une conversion préalable (casting) avant toute évaluation. Cette conversion suit des règles simples:

- ✓ Les opérandes de type caractère et entier court sont converties en entiers.
- ✓ Les opérandes de type flottant sont converties en double.
- ✓ Si une opérande est de type double, les autres opérandes du mème opérateur sont converties en double.
- ✓ Sinon si une opérande est de type long, les sutres opérandes sont converties en long.
- ✓ Sinon si une opérande est de type unsigned, les autres opérandes sont converties en unsigned.
- ✓ Sinon les opérandes sont de type int.

Le résultat de l'expression est du type ainsi déterminé et peut être converti pour une affectation par exemple. Les différents opérateurs accessibles dans le langage C sont les suivants:

Les opération arithmétique			
	-	mois ou opposé	-2 ou 3 - 5
	+	plus	5+8
	*	multiplier	5*8
	/	division	5/2 ≙ (2)
	%	reste ou modulo	5%2≙ 1
Opérateurs booléens		*bit à bit*	
	~	compléments	~0≙ 255
	|	union	0x0f|0xf0 ≙0xff
	^	union exlusive	0xff^0xff\triangleq 0x00
	&	intersection	0x0f&0xf0\triangleq0x00
Opérateurs de décalage			
	>>	décalage droit	111000>>2≙001110
	<<	décalage gauche	
Opérateurs logiques			
	!	non	
	!=	différent	3!=2 ≙vrai
	<	inférieur	
	<=	inférieur ou égal	
	>	supérieur	
	>=	supérieur ou égal	
	&&	et	
	||	ou	
Opérateur d'affectation		changement de l'état mémoir	
	=	affectation	
	+=	ajout afectation	x≙1,x+=2→x≙3
	-=	soustraire	x≙1,x-=2→x≙-1
	*=	multiplier	x≙1,x*=2→x≙2
	/=	diviser	x≙1,x/=2→x≙0
	%=	reste	x≙1,x%=2→x≙1
	>>=	decaler droite	x≙32,x>>=2→x≙8
	<<=	decaler gauche	x≙1,x<<=2→x≙4
	&=	et binaire	x≙1,x&=2→x≙0
	|=	ou binaire	x≙1,x|=2→x≙3
	^=	xor binaire	x≙1,x+=2→x≙3
Opérateur d'incrémentation			
	++exp	pré-incrémentation	x≙1,y = ++x→x=2,y=2
	exp++	post-incrémentation	x≙1,y = x++→x≙2,y≙1
	–exp	pré-décrémentation	
	exp–	post-incrémentation	
Opérateur calcul d'adresse			
	&	adresse de	y≙1,x=&y
Opérateur d'accés			
	*	valeur dont l'adresse est	*x≙1
Opérateur de taille			
	sizeof		sizeof(int)≙2
Opérateur de type		casting	
	(type)		
Opérateur eval en séquence			
	, 20		y=(x=1,x=x+2)→y≙3,x≙3
opérateur conditionel			
	e0 ? e1 : e2		(2>1?'a':'b')≙'a'

Les booléens en C Il n'y a pas de booléen au sens propre du terme en C, par contre toute valeur numérique nulle est considérée fausse et toute valeur non nulle est considérée vraie

Exemple

```
main () {
   int x=0;
   int y=-1;

   if (x) printf("x vrai\n");
   else  printf("x faux\n");
   if (y) printf("y vrai\n");
   else  printf("y faux\n");
}

/** résultat de l'execution
 gcc bool.c
[pfares@pportable MPS]$ ./a.out
x faux
y vrai
*/
```

1.2 Le calcul sur les pointeurs

Les pointeurs sont de type "int" avec quelques nuances. Par contre on peut ajouter ou retrancher un entier à un pointeur. De même si l'on retranche deux pointeurs on obtient un entier. Ce sont les seules combinaisons permises. Lorsque l'on ajoute ou l'on retranche un entier à un pointeur l'opération s'effectue relativement à la dimension (en octets) de l'objet sur lequel le pointeur pointe. Cela correspond à un déplacement dans un tableau d'éléments de même type:

```
struct A
{
    long K;
    long L;
} T[10],*P;
```

Chaque élément de T à 8 octets. Lorsque l'on ajoute 1 au pointeur P on lui ajoute en fait 8 :

```
char *C; P = T;
C = (char *) P;
```

Après les deux affectations suivantes les pointeurs P et C auront la même valeur:

```
++P; C += 8;
```

1.3 Les structures de contrôle

Le langage C possède une structure de blocs. Ces blocs permettent de définir un programme structuré. Plusieurs instructions permettent le traitement d'une instruction complexe ou d'un bloc. Ces instructions sont celles que l'on retouve plus ou moins dans tout langage structuré.

1.3.1 Le bloc

Un bloc est délimité par deux accolades. Il permet la définition de variables locales au bloc et elles sont déclarées dès le début de bloc. Ces variables cachent éventuellement la visibilité de variables appartenant à des blocs supérieurs. Un bloc constitue une instruction.

Exemple:

```
{
    int I,J; /* premier bloc.
              * Ce bloc contient deux variables
              * locales I et J.
              */
    {
        int I; /* bloc imbriqué.
                * Ce bloc comporte une déclaration d'une
                * variable locale I qui occulte la première
                * variable locale I.
                * Dans ce bloc on accède
                * donc au deuxième I et sur J bien sûr
                *
    }
    ........
    /* on accède de nouveau au premier I
     */
}
```

1.3.2 La structure if-else

Cette structure classique prend la forme:

```
if ( expression ) <instructionV>else <instructionF>
```

L'expression est évaluée et si elle est non nulle on applique l'instructionV. Sinon on applique l'instructionF.

Lorsque plusieurs "if" sont imbriqués un "else" est toujours associé à l'"if" sans "else" le plus près. C'est à dire le dernier qui précède ce "else".

Exemple:

if (n > 0) if (a > b)

z = a; else z = b;

Le "else" se rapporte au "if" de la condition "a>b". Cette instruction peut définir un choix multiple :

1.3.3 1.3.3 La structure switch

Cette instruction permet un choix multiple. Ici une expression est évaluée et sa valeur détermine l'endroit du programme où l'on doit commencer l'exécution. Sa forme générale est la suivante:

```
switch ( expression) {
    case val_1 : instruction
    case val_n : instruction
    default : instruction
}
```

Il faut remarquer plusieurs conventions à propos de cette instruction "switch":

- La valeur de l'expression doit être de type "int". Toutes les valeurs associées aux différents cas sont des constantes converties en "int".
- La valeur de l'expression est comparée à chaque cas et le premier ayant la valeur de l'expression détermine le point de départ de l'exécution qui se déroule ensuite en séquence jusqu'à la fin de l'instruction "switch".
- La constante "default" permet de définir un point d'entrée quelle que soit la valeur de l'expression de façons à avoir des traitements mutuellement explusifs.

1.3.4 L'instruction while

Cette instruction de boucle permet de répéter une instruction tant qu'une condition est vérifiée:

```
while ( expression ) instruction
```

Exemple:

Lecture d'un mot séparé par un blanc avec conservation des huit premiers caractères:

```
int i = 0;
while((c = getchar()) != ' ') if (I < 8) NOM[i++] = c;
```

1.3.5 L'instruction do-while

Cette instruction est similaire à la précédente. Dans le cas de l'instruction précédente le test est d'abord effectué et par conséquent l'instruction peut ne jamais être exécutée. Dans le cas de cette instruction le test est réalisé après l'exécution de l'instruction qui est donc toujours effectuée au moins une fois.

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

est équivalent à:

```
instruction
WhilE( expression) instruction
```

1.3.6 L'instruction for

Cette dernière instruction de boucle du langage C possède trois composantes: l'initialisation, le test et la progression. Sa forme est la suivante:

```
for( expression~1; expression~2; expression~3) instruction
```

Cette instruction est équivalente à la suite:

```
expression~1 while(expression~2) {
    instruction
    expression~3;
}
```

Chaque expression est facultative mais dans ce cas on doit laisser les points virgules. Lorsque l'expression 2 est absente elle est réputée vraie par convention (jamais nulle).

Exemple:

```
boucle infinie: for(;;) instruction
```

1.3.7 L'instruction break

Cette instruction arrête l'exécution d'une instruction "while", "dowhile", "for" ou "switch". Cet arrêt concerne l'instruction la plus interne lorsque plusieurs de celles-ci sont imbriquées.

Exemple:

Recherche dans une table d'un mot de huit caractères :

```
{
char T[100][8],NOM[8];
int I,J;
for(I = 0; I < 100; ++I)
    for(J = 0; J < 8; ++J)
        if (T[I][J] != NOM[J]) break;
        if (J == 8) break;
}
```

Le premier "break" arrête la comparaison des mots dès qu'il y a deux caractères différents entre le mot de la table et le mot recherché.

Le deuxième "break" arrête la recherche dans la table dès que le mot a été trouvé. Il a été trouvé lorsque J est égal à 8 et donc lorsque la boucle interne ne s'est pas arrêtée sur le "break".

1.3.8 L'instruction continue

Cette instruction définie une nouvelle itération d'une boucle "while" .

"do-while" ou "for". Elle est équivalente à une instruction "goto" ver la fin de boucle. Elle est équivalente à l'instruction "goto contin;" dans les cas suivants:

while(expression) do for(.....) contin: ; contin: ; contin: ;

1.3.9 L'instruction goto

Le langage C possède bien évidemment une instruction de saut. Elle es rédu.ite à sa plus simple expression. Chaque instruction peut ét= identifiée par une étiquette et l'instruction "goto" fait référence l'une de ces étiquettes:

goto etiquette;

Une étiquette est un identificateur placé devant une instruction:

etiquette : instruction

1.3.10 L'instruction vide

Une instruction peut être nulle, c'est à dire vide. Elle peut servir définir un corps de boucle nul par exemple, ou à positionner une étiquette.

Exemple:

boucle d'attente d'interruption:

```
for(;;) ;
```

boucle infinie où il n'y a rien à faire.

1.4 4 Les fonctions et la notion de classes

1.4.1 Les fonctions

Une fonction C est un bloc définissant une valeur. Ce bloc possède des paramétres d'appels et un type identifiant la nature de l'élément produit.

Une fonction a la forme suivante:

```
type <nom de la fonction> ( <liste des paramètres> )
description des paramètres
{
    <corps de la fonction>
}
```

üne fonction produit une valeur qui est définie par l'instruction "return". Si cette instruction n'est pas présente ou si elle ne définit pas de valeur la valeur de la fonction est indéfinie (mais elle existe).

1.4.1.1 L'instruction retum

L'instruction "return" permet de définir la valeur de la fonction. cette instruction à la forme suivante:

```
   return expression ;
ou
   return ;
```

La valeur de l'expression est convertie dans le type de la fonction. Dans le cas de l'expression vide la valeur transmise est indéfinie.

Exemple:

```
int max(a,b)
int a,b;
{
    return a>b?a:b;
}
```

Cette fonction retourne la valeur maximum des paramètres a et b.

les paramètres

Une particularité du langage C réside dans son mode d'appel des fonctions:

- Tout paramètre est appelé par valeur.
- Les paramètres sont en général simples (obligatoirement simple , dans la première version) c'est à dire de type "int" "double" ou "pointeur". Un argument de type "short" par exemple est converti (suivant le compilateur) en type "int". Un argument de type "float" est converti en "double". Bien sûr il est possible de passer en paramètre un pointeur sur un objet quelconque.
- Un paramètre qui n'est pas défini est réputé de type "int".
- Un tablesu est un pointeur sur le premier élément, il peut donc ètre passé comme paramètre. Lorsqu'il est passé comme paramètre seule sa première dimension peut ne pas étre précisée:

exp(T,U,V)

int T[],U[][2],V[][8][10];

Exemple:

main() int I;

Le programme C

Un programme C est par définition un fichier C qui comprend une ou plusieurs fonctions. Pour une liaison avec le système une de ces fonctions doit porter le nom particulier "main". Chaque programme débutera par la fonction "main().

1.4.1.2 les fonctions d'entrée/sortie

Le langage C ne possède pas dans sa définition de fonctions spécifiques d'entrée/sortie. A coté d'un compilateur C il existe toujours une bibliothèque de fonctions qui peuvent être appelées comme toute fonction. Certaines de ces fonctions ont été définies pour réaliser des tâches d'entrée/sortie. Lorsque l'on utilise C sous UNIX on peut utiliser les fonctions standard d'entrée/sortie suivantes:

read(), fread(), pour la lecture de fichier. write(), fwrite(), pour l'écriture de fichier. open(), fopen(), pour l'ouverture de fichier. creat() pour la création de fichier. close(),fclose() pour la fermeture de fichier.

scanf(),fscanf() pour une lecture de données suivant un format. printf(),fprintf() pour l'écriture de données suivant un format.

Bien sûr il existe un grand nombre d'autres fonctions. Ces fonctions seront examinées avec la bibliothèque standard.

1.4.2 La notion de classes

Notion essentielle d'un programme C cette notion concerne les variables et les fonctions.

1.4.2.1 Classe des variables

Une variable peut être dynamique ou statique, externe, locale ou définie dans un registre.

La classe naturelle d'une variable de bloc est dynamique ou sutomatique. Les deux déclarations suivantes sont équivalentes:

```
auto int I;
int I;
```

Une variable automatique n'a d'existence que dans le bloc et lorsque celui-ci est actif. Sa valeur initiale peut être définie par une expression quelconque et est positionnée à chaque activation du bloc (certains compilateurs n'acceptent pas l'initialisation des variables automatiques).

```
int I = f(K);
```

valeur initiale de la variable I est définie par l'appel de la La fonction f().

Une variable de type register signifie qu'elle doit utiliser autant que possible un registre. Elle peut également être initialisée de la même façon qu'une variable automatique. Seul un type simple (char, int, long) peut utiliser un type register et l'on ne peut pas utiliser l'adresse d'une telle variable:

Si I est défini par:

```
register int I;
```

alors l'instruction suivante est illégale:

```
P = &I;
```

Utilisation de l'adresse d'une variable de type "register".

Une variable statique est une variable qui est toujours présente dans le programme. Son initialisation ne peut être définie que par une expression constante et cette dernière n'est réalisée qu'une seule fois su chargement du programme.

Exemple:

```
exp1()
{
    int I=2;
    return I++;
}
```

```
exp2()
{
    static int I=2;
    return ++I;
}
```

Dans la premier cas chaque appel de la fonction "expl()" initialise la variable I. Cette fonction "expl()" a donc toujours la valeur 3.

Dans le deuxième cas la variable I n'est initialisée qu'une seule fois en début de programme et est toujours présente. Le premier appel de la fonction "exp2()" produit donc la valeur 3, mais le second appel produit la valeur 4, le troisième la valeur 5 et sinsi de suite.

Une variable externe est une variable qui est définie en dehors des fonctions et peut par conséquent étre référencée par plusieurs fonctions.

extern int I; autre référence à la variable I.

Une variable externe ne peut être automatique. Elle peut par contre être statique et dans ce cas la signification est particulière. Un programme C est un fichier qui comporte plusieurs fonctions et ce programme doit étre lié à d'autres programmes pour pouvoir être exécuté. Ces autres programmes peuvent bien sfzr faire référence à des variables externes d'un fichier quelconque. Dans le cas d'une variable externe statique seules les fonctions du même fichier peuvent faire référence à cette variable. L'attribut "statique" limite donc pour une variable la portée de sa référence potentielle:

Exemple:

fichier1:	fichier2	fichier3
static int j;	extern int i;	int i;

La variable "J" ne peut étre connue que dans le fichier 1, c'est à dire par les fonctions "fl()" et "f2()". La variable I peut étre connue de toutes les fonctions "fl()", "f2()", "f3()", "f4()" et "f5()".

1.4.2.2 Classe des fonctions

Par définition les fonctions sont de classe externe. Aussi une fonction peut être utilisée dans d'autres fichiers que celui où elle est définie sans avoir à la redéfinir.

Lorsqu'une fonction est de classe statique cette fonction n'est connue que dans le fichier où elle est définie (même restriction que pour les variables externes).

Exemple:

int fl()

static int f2()

La fonction "fl()" peut être appelée par une fonction de n'importe quel autre fichier alors que la fonction "f2()" ne peut être appelée que par les fonctions de ce fichier, ici que par "fl()".

int modif(); I = 4;

modif(I);

}

L'instruction "modif(I)" correspond à l'appel de la fonction "modif()". La variable "I" ne sera pas modifiée par cet appel. Elle sera donc toujours égale à 4.

modif(J) int J;

J = 3; return;

Pour modifier la valeur de I on peut utiliser l'affectation. La procédure retourne alors la valeur souhaitée:

main() int I;

int modif(); I = 4;

I = modif(I);

modif(J) int J;

J = 3; return J;

Ce premier cas est celui de l'appel traditionnel par valeur. On peut simuler l'appel par nom comme dans l'exemple suivant:

main() int I;

int modif2(); I = 4;

modif2(&I);

On a toujours un appel par valeur. Mais ici on passe en paramètre l'adresse de I. C'est donc une simulation d'un appel par nom. Après l'appel la variable I sera modifiée et aura la valeur 3.

int modif2(J) int *J;

*J = 3; return;

La récursivité

Contrairement à d'autres langages structurés il n'y a pas de précisions supplémentaires à donner pour définir une fonction récursive. Toute fonction C a la possibilité naturelle d'être récursive.

Exemple:

La fonction suivante imprime une chaîne de caractères dans l'ordre inverse (mot miroir). Cette fonction est définie récursivement pour les besoins de l'exemple:

```
imprr(s)
char *s;
{
    if (!*s) return;
    imprr(s + 1);
    printf("%c",*s);
    return;
}
```

Chapter 2

PARTAGE DE DONNEES; PRECOMPILATION

Lorsque plusieurs programmes doivent traiter les mêmes données il est très u‘˜ile que la définition de ces données soit unique. La modification de l'une de ces données est alors effectuée automatiquement pour tous les programmes concernés par cette donnée. Pour réaliser cet objectif il est nécessaire que la définition d'une donnée n'appartienne pas à un programme particulier. La définition de cette donnée doit donc alors être introduite automatiquement dans le programme utilisateur au moment de sa compilation. Cette opération est définie pour beaucoup de langage de programmation (PL/1, ADA, C, ...). Cette opération s'effectue avant la compilation et ajoute donc une phase supplémentaire à la production d'un programme.

Cette phase est dénommée "phase de précompilation":

2.1 La commande include

Cette commande permet d'ajouter au fichier à compiler le contenu d'un autre fichier. Ainsi toutes les définitions qui doivent être utilisées par plusieurs programmes seront défiaies dans un fichier qui lui même sera introduit dans le fichier à compiler su moment de la précompilation.

Exemple:

Le programme suivant réalise une liste des éléments d'un tablesu de structures. La structure est composée seulement de deux éléments : un nom et une valeur :

```
main() {
    struct elem {
       char nom[8]; int val;
    } tab[10]; int i;
    for (i=0; i< 10; ++i)
       scanf("%7s %d",tab[i].nom,&tab[ij.val);
    for(i = 0; i < 10; ++i)
       printf("%s Xd\n",tab[i].nom,tab[i].val);
}
```

Si dans un autre programme on utilise la même structure elle devra être redéclarée. En plaçant cette définition dans un fichier externe cette structure ne sera déclarée qu'une seule fois:

Le fichier de définition: elem.h

```
struct elem {
    char nom[8]; int val;
};
```

Le programme d'écho devient:

```
#include "exemp.h"
main()
{
    struct elem tab[10]; int i;
    for(i=0; i< 10; ++i)
        scanf("X7s Xd",tab[i].nom,&tab[iJ.val);
    for(i = 0; i < 10; ++i)
        printf("Xs Xd‘n",tab[iJ.nom,tab[iJ.val);
}
```

Un autre programme utilisant la même structure peut alors être défini:

```
#include "elem..h"
main() {
  struct elem tab[10];
  int i;
  for(i=0; i< 10; ++i)
    scanf("%7s %d",tab[i].nom,&tab[i].val);
  tri(tab);
  for(i = 0; i < 10; ++i)
    printf("%s %d\n",tab[i].nom,tab[i].val);
}
tri(tab)
struct elem *tab;
{
  int i,j,k; char c;
  j = l;
  while (j) {
    j = 0;
    for(i=0; i < 9; ++i) {
      for (k = 0; k < 7; ++k)
        if (tab[i].nom[k] != tab[i+1].nom[k]) break;
      if ((k < 7) && (tab[i].nom[k] > tab[i+1].nom[k])) {
        j = l;
        for(k = 0; k < 7; ++k) {
          c = tab[i].nom[k];
          tab[i].nom[k] = tab[i+1].nom[k];
```

```
            tab[i+l].nom[k] = c;
          }
          k = tab[iJ.val;
          tab[i].val = tab[i+l].val;
          tab[i+1].val = k;
        }
    }
  }
  return;
}
```

La modification de la structure doit être réalisée dans le fichier de définition "elem.h". Cette modification sera alors valable pour les deux programmes.

Lors de l'écriture d'un projet toutes les données communes à plusieurs unités de traitement de ce projet doivent être déclarées dans des fichiers externes Ces fichiers comprendront aussi les données _succentibles d'être utilisées lors des modifications ou extensions futures.

la commande "include" distingue deux catégories de fichiers. Les fichiers systèmes qui contiennent les définitions standard liées au système et les fichiers utilisateurs spécifiques a chaque application. Les fichiers systèmes sont placés dans un endroit précis de la machine (sous UNIX ils sont placés dans le répertoire "/usr/include". Les fichiers utilisateurs sont soit placés dans un emplacement précisé paz l'utilisateur (sous UNIX ils sont placés dans le répertoire courant ou dans un répertoire précisé à l'appel du compilateur). La distinction entre les deux types du fichier s'effectue par les caractères qui encadrent le nom du fichier à inclure:

Nom de fichier entre guillemets: fichier utilisateur.

Nom de fichier entre les symboles "<" et ">": fichier système. Exemple:

#include <stdio.h> /* fichier système */ #include "exemp.h" /* fichier utilisateur */

2.1.1 La commande define

Au cours de la phase de précompilation le langage C permet le traitement de macro-instruction. La première fonction des macroinstructions correspond au remplacement d'une chaîne de caractères par une autre. Ce remplacement s'effectue sur toutes les occurences possibles de la chaîne source.

Exemple:

La commande *#define EXP 1* transforme l'instruction *x += EXP * EXP* en *x += 1 * 1*

Cette possiblité permet de définir toutes les constantes machines par des noms symboliques. Ce procédé permet une plus grande aisance dans la portabilité du logiciel.

Ce remplacement peut comprendre tout un programme:

```
#define EXPI { printf("un exemple de remplacement~n"); }
```

Dans ce dernier cas la partie de programme:

```
i = 0; EXP1 j = 1;
```

sera remplacée avant la compilation par la partie:

```
i = 0;
printf("un exemple de remplacement~n"); } j = 1;
```

La deuxième fonction des macro-instructions concerne la définition d'une chaîne de caractères avec paramètres. Dans la chaîne de caractères résultante il y a substitution des chaînes paramètres.

Exemple:

La définition

```
#define max(a,b) (a < b)?  b :  a
```

transforme la partie de programme

```
x = max(z,t);
```

en

```
x = (z < t)?  t:  z;
```

Un nom défini dans une macro-instruction peut toujours être éliminé pour la suite de la compilation par la commande "undef". Cette commande annule la signification de la chaîne paramètre:

Exemple:

#define max(a,b) (a < b)? b : a

x = max(z,t); /* appel de la macro */ #undef max

x = max(z,t); /* appel de la fonction "max" *I

Dans le premier cas l'expression sera remplacée par l'extension de la macro-instruction. Dans le second cas aucun pré-traitement ne sera effectué et un appel à une fonction "max" sera réalisé.

2.1.2 La précompilation conditionnelle

La précompilation conditionnelle permet de définir différents codes pour le méme programme source.

....

l'utilisation directe du maximum par rapport à l'appel d'une fonction. La définition du maximum pourra étre alors la suivante dans le fichier MAX.h:

```
#ifdef DIRECT
#define max(a,b) (a < b)? b : a )
#else
max(a,b)
int a,b;
{
   return((a < b)? b : a);
}
#endif
```

Si au debut du programme utilisant max(a,b) la constante DIRECT est définie la fonction "max" sera traitée comme une macro-instruction. Dans le cas contraire cette fonction sera définie comme une fonction standard C.

Le programme:

```
#define DIRECT 1
#include "MAX.h"
main(){
  int x,y,z;
  x = 0;
  y = 1;
  z = max(x,y);
}
```

sera compilé comme:

```
#define max(a,b) ((a < b)? b : a )
main() {
  int x,y,z;
  x = 0
  z = ((a < b)? b : a );
}
```

par contre si

```
#include "MAX.h"
main(){
  int x,y,z;
  x = 0;
  y = 1;
  z = max(x,y);
}
```

DIRECT n'étant pas défini

le précompilateur fournit

```
max(a,b)
int a,b;
{
   return((a < b)? b : a);
}
main(){
  int x,y,z;
  x = 0;
  y = 1;
  z = max(x,y);
}
```

L'instruction de précompilation conditionnelle peut également vérifier la valeur d'une expression constante. Si cette expression est non nulle alors la partie conditionnelle est prise en compte. Dans le cas contraire, c'est à dire si l'expression est nulle la partie prise en compte est la partie encadrée par les préinstructions "#else" et "#endif".

Exemple: On souhaite placer dans une zone fixe d'une

longueur donnée des élements de longueur fixe. Si la longueur

des éléments à ranger est un sous-mutiple de la longueur de la zone

il faut utiliser un tableau. Dans le cas contraire il faut

utiliser une structure:

```
#define LGZONE 512 /* longueur de la zone */
typedef struct {
  zone
  char NOM[8];
  ïnt K;
}ELEM; /* définition de l'élément */
#define LGELEM 12 /* longueur de
#if LGZONE % LGELEM
struct ELSTR { /* tableau d'élément */
   ELEM TELEM[LGZONE / LGELEM];
   char CDR[LGZONE X LGELEM]; /* caractères de remplissage */
} PLELEM;
#else
ELEM PLELEM[LGZONE / LGELEM];
#endif
```