Le langage C

Version du 15 septembre 2009
Compléments aux notes de cours

Historique du langage C

1963 CPL (Combined Programming Language) (Université de Londres, UK)

- équivalent américain du langage Algol 60;
- tentative de rallier l'Amérique à Algol 60.

1967 BCPL (Basic CPL) (Richard)

- sous-ensemble de CPL;
- plus facile à comprendre;
- les ambiguïtés du langage CPL demeurent.

1970 B (Thompson de Bell Lab)

- simplification de BCPL;
- essentiellement, il s'agit d'un langage sans types;
- difficile à porter sur des machines disparates, car le langage est trop près la machine.

1972 C (Kernighan et Ritchie de Bell Lab)

- sur-ensemble de B (inclusion de types de données);
- permet d'adresser les composantes de la machine par un accès de haut niveau.

1973 UNIX est traduit en C sur PDP-11

- des 13 kloc, 800 sont en assembleur;
- tous les utilitaires d'UNIX (éditeurs, compilateurs, etc.) sont écrits en C.

1983 Création de C++ (Stroustrup de Bell Lab)

1986 Création de Objective C (Cox)

1989 ANSI-C (comité X3J11)

- assure la portabilité du langage par une norme.

1989 Création de Concurrent C (Gehani de Bell Lab)

1990 Création de C* (Thinking Machines)

1995 Création de Java (SUN MicroSystems)

Caractéristiques générales

- Il s'agit d'un langage de haut niveau avec les possibilités trouvées dans un langage d'assemblage.
- C possède un jeu d'instructions varié :
  - opérations arithmétiques, logiques et de manipulations de bits;
  - énoncés de contrôle.
- Il permet la récursivité.
- Le langage est très expressif : il dispose d'un jeu d'opération puissant.
- Il est modulaire :
  - il n'y a qu'une seule seule unité syntaxique pour définir des sous-programmes, la fonction, et pour laquelle l'imbrication de fonctions n'est pas permise;
  - l'encapulation (l'isolement) des variables se fait sur l'étendue d'un fichier;
  - supporte la compilation séparée.
- On le dit efficace. (Ce n'est pas vrai.)
- On le dit aussi portable.
  - Les E/S ne sont pas définies par le langage.
  - Les définitions des types élémentaires (ou de base) dépendent de la machine.

Quelques critiques

- C'est un langage peu lisible qui fait un usage abusif de certains caractères (par exemple, `'&'`, `'='` et `'*'`)
- Il n'y a pas de vérification des indices des tableaux.

Exemple d'un programme C

```
/*
 * Ecrit le carré des nombres 1 à 10
 */


#include <stdio.h>
void main(void)
{
    int i;



    i = 1;
    while (i <= 10) {
       printf("le carré de %d est %d\n",i,i*i);
       i = i + 1;
    }
    printf("\nFin du programme\n");


}
```

Quelques remarques

Un commentaire débute par la chaîne `"/*"` et se termine après la première apparition de `"*/"`.

Les compilateurs C++ permettent aussi de débuter un commentaire par la chaîne `"//"`. Dans ce cas, la fin du commentaire se termine à la fin de la ligne.

Exemples

1. `/* correct */`
2. `/* correct */`
3. `/* cor /* re /* ct */`
4. `/* problè*/me */`

Exemple d'un programme C

La fonction `printf()`

Cette fonction permet d'écrire à l'écran (sortie stdout).
syntaxe: `printf(format` [, <paramètres>] * `);`

Le premier paramètre indique le format qu'on désire utiliser pour l'impression. Il s'agit d'une chaîne de caractères contenant des caractères ordinaires (écrits tels quels à l'écran) et des codes de conversion introduits par le caractère `%`.
Chaque code de conversion est associé à un paramètre de l'appel, qui est converti selon le format demandé puis affiché.

Principaux codes de conversion

- `%d` entier affiché en décimale
- `%c` caractère
- `%s` chaîne de caractère terminée par un NULL
- `%x` entier non-signé affiché en hexadécimale avec les minuscules `a` à `f`
- `%X` idem mais avec les majuscules `A` à `F`
- `%o` entier non-signé affiché en octale
- `%%` affiche le caractère %

Types élémentaires

Il y a 3 types élémentaires :

les entiers	int
les réels	float
les caractères	char

Les entiers se décomposent en sous catégories:

- les entiers courtsshort int ou short
- les entiersint
- les entiers longslong int ou long

La capacité des différents types d'entiers dépend de l'ordinateur utilisé et aussi du compilateur. Mais, indépendamment de la machine, nous avons toujours la règle

short int δ int δ long int

Machine	short	int	long
Macintosh	16	16	32
VAX	16	32	32
IBM PC (sous WIN16)	16	16	32
IBM PC (sous WIN32)	16	32	32
IBM RISC 6000	32	32	64

Types élémentaires (suite)

Les nombres en virgule flottante se décomposent aussi en trois sous catégories :

- courts(6 chiffres significatifs)float
- longs(16 chiffres significatifs)double
- très longs(ANSI-C)long double

Généralement les courts tiennent sur 32 bits alors que les longs tiennent sur 64 bits. Les long double possèdent une taille uniforme d'un ordinateur à l'autre car ils implantent le format de l'IEEE qui tient sur 80 bits.

Le type caractère est généralement utilisé pour conserver un caractère. Il peut aussi servir comme entier sur 8 bits.

Représentation des valeurs numériques

Toute valeur numérique assignée à un entier (int) se fait par des chiffres 0 à 9.

Pour indiquer qu'il s'agit d'un long ou d'une valeur non signée, on utilise les suffixes `L` et `U`.

Exemple
`65000U` peut être représenté dans un unsigned int de 16 bits mais pas dans un int de 16 bits.

Par défaut, la base 10 est utilisée, mais on peut aussi employer les bases 8 et 16. La base 8 s'introduit en préfixant le nombre par 0 et la base 16 par la chaîne `0x`.

Exemple

```
057
 057L
```

`47` et `0x2F` représentent la valeur 47 sur un int
`47L` et `0x2FL` représentent la valeur 47 sur un long

Une valeur numérique de type virgule flottante est composée d'une partie entière, d'un point, d'une partie fractionnaire, de la lettre `E` (ou `e`), et d'une partie exposant. La partie entière et la partie fractionnaire peuvent être omises, mais pas les deux en même temps. Le point décimal et le `E` peuvent aussi être omis, mais pas en même temps.

Toute constante en virgule flottante est considérée double, à moins qu'elle ne soit préfixée des lettres `F` ou `L` (minuscule ou majuscule), auquel cas elle est respectivement float ou long double.

Exemple

`0.3` `.3` et `3E-1` représentent la même valeur
`.3` `.3F` et `.3L` sont différents

Représentation des valeurs numériques (suite)

Les caractères sont des valeurs numériques. La valeur numérique est équivalente à un nombre décimal, octal ou hexadécimal qui serait le code interne du caractère dans la représentation interne de la machine utilisée.

Exemples

`'A'` et `0101` sont équivalents en ASCII

`'0'` et `48` le sont aussi en ASCII.

Certains caractères particuliers doivent être précédés du caractère `'\'` (barre oblique inverse). Ce sont habituellement des caractères de contrôle :

début d'une nouvelle ligne	`'\n'`
retour de chariot	`'\r'`
recul vers la gauche	`'\b'`
saut de page	`'\f'`
tabulateur horizontal	`'\t'`
tabulateur vertical	`'\v'`
valeur numérique octale	`'\ooo'`
valeur numérique hexadécimale	`'\xhhh'`
sonnerie	`'\a'`
apostrophe	`'\''`
barre oblique inverse	`'\\'`

où ooo représente un nombre octal de 1, 2 ou 3 chiffres, et hhh un nombre hexadécimale de 1 à 3 chiffres.

Exemple

`'A', '\x41'` et `'\101'` représentent le même caractère.

Définitions des variables

<définition> ::= <type simple> <liste d'identificateurs> ;
<type simple> ::= [<signe>] int | [<signe>] char | [<signe>] short [int] |
[<signe>] long [int] | float | [long] double
<signe> ::= unsigned | signed

<liste d'identificateurs> ::= <identificateur> [, <identificateur>]*
<identificateur> ::= [_]* <lettre> [<lettre> | <chiffre> | _]*

Exemples

```
int c, _c, i, j;
float r;
long int I;
signed long int R;
```

Il est important de remarquer que les minuscules et les majuscules sont considérées comme différentes. Ainsi, les identificateurs `Nombre, NOMBRE` et `nombre` sont tous trois différents.

Opérateurs arithmétiques

Par ordre de priorité décroissante, ils sont

plus unaire `+`moins unaire `-` incrément `++`décrément `--` multiplication `*`division `/` addition +soustraction `-` affectations `= += *= /= -=` et `%=`	modulo `%`

- Ces opérateurs sont tous binaires (ils agissent sur 2 opérandes), sauf les 2 opérateurs unaires, l'incrément et le décrément, qui n'agissent que sur un seul opérande.
- Le plus unaire est un ajout du standard ANSI qui permet d'imposer l'évaluation qui le suit avant celle des autres.
- La division de 2 nombres entiers donne un résultat entier et tronqué. Ainsi `5 / 3` donne `1`.
- L'opérateur `%` est le reste de la division. Par exemple, `5 % 3` donne `2`.
-
- Les opérateurs d'incrément et de décrément affectent une variable avant ou après sa référence.

Exemples

```
int i=4, a, b, c, d;
i++;/* équivalent à i = i + 1;    */
i--;/* équivalent à i = i - 1;    */
a = i++;/* équivalent à a = i; i++;   */
b = i--;/* équivalent à b = i; i--;   */
c = ++i;/* équivalent à i++; c = i;   */
d = --i;/* équivalent à i--; d = i;   */
```

Opérateurs arithmétiques (suite)

- L'affectation d'une valeur à une variable est un opérateur. Le résultat de cet opérateur est la valeur affectée. Ainsi `i = 3;` met 3 dans `i` et retourne 3. Aussi,

  `i = (j = 5) + 9;` affecte la valeur 14 à `i`.

- Les opérateurs <op>= sont des raccourcis.

  <variable> <op>= <expression>;

  est équivalent à

  <variable> = <variable> <op> `(<expression>);`

La conversion implicite de types

Le type le plus faible est toujours converti dans le type le plus fort :
char □ short □ int □ long □ float □ double □ long double

Exemples

```
i = j - 'A' * r++;
x = --a + (b + 3) - c++;
```

Remarquons que rien ne garantit que `--a` sera évalué avant `c++`. Deux compilateurs différents pourraient les évaluer dans un ordre différent. C'est ce problème que règle le plus unaire de la norme ANSI.

Conversion explicite

La conversion explicite de type prend la forme

```
(<type>)<expression>
```

Exemples

```
int i;
i = (int)2.5;                    /* i = 2 */
```

```
int i = 1, j = 3;
double x;
x = i / 3;                       /* x = 0.0*/
x = (double)i / j;               /* x = 1.0 / 3 = 0.333333 */
x = (double)(i / j)              /* x = (double)(0) = 0.0 */
```

Exemple complet

```
double d = 3.2, x;
int i=2, y;


x = d * ( y = ( (int)2.9 + 1.1) / d );
                                  /* que valent x et y? */


x=   d *(   y=   ( ( (int)2.9) + 1.1) / d );
(x   = d*   (y   = ( 2 + 1.1) / d ) );
(x   = d*   (y   = 3.1 / d ) );
(x   = d*   0.0 );/* y = 0 */
(x   = 0.0 );
0.0
```

Opérateurs de manipulation de bits

Par ordre décroissant de priorité, ils sont

complément à un ~
décalage à gauche <<décalage à droite >>
et &
ou exclusif ^
ou |
affectations <<= >>= &= ^= |=

- Tous ces opérateurs sont binaires sauf le complément à un qui est unaire.
- Le décalage à gauche équivaut à multiplier par 2. La forme que prend cet opérateur est
  valeur << nombre_de_positions.

- Le décalage à droite est difficilement portable. Les compilateurs réagissent différemment selon qu'ils interprètent un décalage comme étant logique ou comme étant arithmétique.
  Le décalage logique insert toujours des 0 à gauche. Par contre, le décalage arithmétique insert la valeur du bit de signe à gauche.

Exemples

```
int i = 2, j = -2;
i = i >> 1;          /* i = 1 */
j = j << 1;          /* j = -4 */
```

Sur une machine de 16 bits, que réalisent les opérations ci-dessous?

```
int i, j;


i = i & 0177;
j = (i << 8) | ((i >> 8) & 0377);
```

Les opérateurs logiques

Dans cette catégorie, les opérateurs disponibles sont les suivants :

non logique `!`	
comparaisons `< <= >= >`	
égalité `==`	différent `!=`
et logique `&&`	
ou logique `||`	

- Ces opérateurs sont dits logiques car ils retournent une valeur booléenne : 0 (zéro) si le résultat est faux, et 1 s'il est vrai.
- Même si les opérateurs logiques retournent les valeurs 0 et 1, ces opérateurs agissent sur toute valeur entière. Le langage C stipule que toute valeur nulle (0) équivaut à un faux et que toute autre valeur équivaut à un vrai.
  Ainsi, l'opérateur `!` transforme toute valeur nulle en un 1 et toute valeur non nulle en 0.
- Il est important de remarquer que les opérateurs `&&` et `||` n'ont absolument pas le même comportement que leurs homologues `&` et `|`. Alors que les opérateurs de manipulation de bits opèrent sur tous les bits un à un, les opérateurs logiques ne regardent que la valeur globale pour déterminer si l'expression est nulle ou pas.
- Malgré ce qui a été dit sur l'ordre d'évaluation des différentes parties d'une expression, l'évaluation d'une expression logique faisant intervenir les opérateurs `&&` et `||` se fait toujours de gauche à droite et se termine dès que le résultat est connu.

Exemple

```
int a, b;

a || b++;          /* si a est vrai, b n'est pas incrémenté */
a && b++;          /* si a est faux, b n'est pas incrémenté */
```

Autres opérateurs

Opérateur `sizeof`

Cet opérateur unaire retourne l'occupation mémoire en octet de son opérande. Le résultat de cet opérateur peut être utilisé dans une expression arithmétique.

syntaxe sizeof <expression>
sizeof (<type>)

Opérateur conditionnel

Il s'agit de l'opérateur ternaire (à trois opérandes) semblable à l'énoncé conditionnel.

syntaxe <condition> ? <expression 1> : <expression 2>

La condition est d'abord évaluée. Si elle est vraie, le résultat est la valeur de l'expression 1; dans le cas contraire, le résultat est la valeur de l'expression 2.

Exemple

```
printf("Le résultat est %d pomme%s.",a,a > 1 ? "s" : "");
```

Priorité des opérateurs

Tous les opérateurs que nous avons vus sont classés par ordre décroissant de priorité. Ce tableau aide à déterminer quel opérateur appliquer avant l'autre dans une expression qui en possède plusieurs. Ainsi, `a = b + c * d` doit être interprété comme `a = ( b + (c * d))` à cause des priorités.

Que faire lorsque plusieurs opérateurs ayant la même priorité se trouvent dans la même expression?

C'est à ce moment-là qu'il faut tenir compte de l'associativité. Ainsi, l'expression `a = c - d + e` fait intervenir 2 opérateurs arithmétiques. Bien que l'addition soit mentionnée en premier lieu sur la ligne du tableau, il ne faut pas nécessairement l'appliquer en premier pour autant. Lorsque l'associativité des opérateurs est de gauche à droite, il faut, parmi tous les opérateurs de même priorité, utiliser d'abord celui qui se trouve le plus à gauche dans l'expression.
Donc, `c - d + e` signifie `(c - d) + e`.

Quand l'associativité est de droite à gauche, nous devons d'abord appliquer celui qui se trouve le plus à droite dans l'expression. Par exemple, `a = b = c = 0` est équivalent à `(a = (b = (c = 0)))`.

opérateur	associativité
1. `() [] . ->`	gauche à droite
2. `! ~ ++ -- - + (<type>)` sizeof	droite à gauche
3. `* / %`	gauche à droite
4. `+ -`	gauche à droite
5. `<< >>`	gauche à droite
6. `< <= > >=`	gauche à droite
7. `== !=`	gauche à droite
8. `&`	gauche à droite
9. `^`	gauche à droite
10. `|`	gauche à droite
11. `&&`	gauche à droite
12. `||`	gauche à droite
13. `? :`	droite à gauche
14. `= += -= etc.`	droite à gauche
15. `,`	gauche à droite

Énoncés de contrôle

Un énoncé est n'importe quelle expression suivie du caractère `';'` (point-virgule).

Exemples

```
x++;
 6;
```

Les énoncés peuvent être regroupés en bloc pour former une instruction composée.

<énoncé> ::= <expression> `;` | <bloc>
<bloc> ::= `{` <définition>* <énoncé>* `}`

Exemple

```
{

    i1;
    i2;

}
```

- Remarquons que, même s'il remplace un énoncé, un bloc ne se termine pas par un point-virgule.
- Un bloc peut être compris dans un autre bloc.
- De nouvelles variables peuvent être définies au début de tout bloc.

Portée des identificateurs

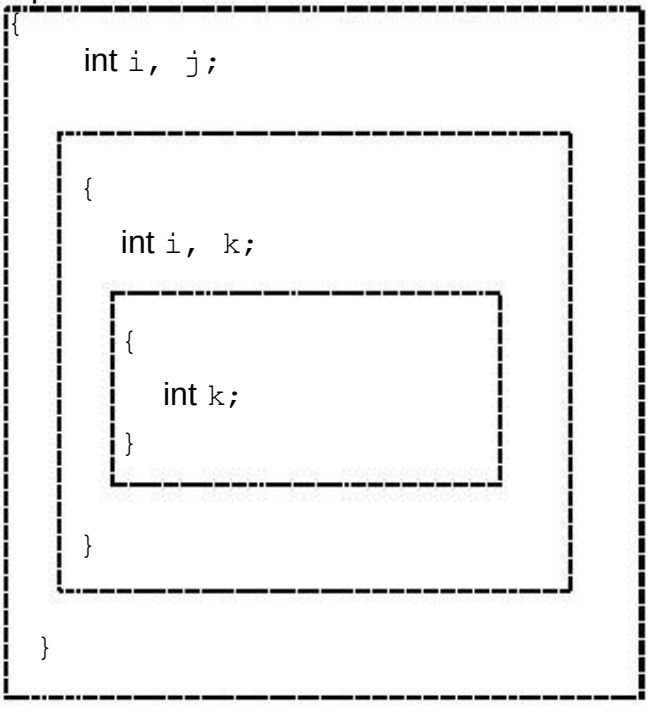
La portée des identificateurs définit la visibilité de ces identificateurs.
Il y a trois règles :

1. Toute variable cesse d'exister lorsqu'on rencontre la fin d'un bloc dans lequel elle est définie.

2. Toute variable définie à l'intérieur d'un bloc est connue partout dans le bloc.

3. Le compilateur utilise toujours la définition qui se trouve au niveau le plus rapproché.

Exemple



Fonctions

- La définition d'une fonction désigne l'endroit où cette fonction est définie.
- La déclaration (ou ligne de prototype) d'une fonction ne fait que spécifier les types de la liste de ses arguments et le type de sa valeur de retour.

Exemple

```
int Somme(int, int);          /* déclaration ou ligne de prototype */


void main(void)
{
    ...
    ... = Somme(3,4);
}



int Somme(int a, int b)       /* définition de la fonction */
{
  return a + b;
}
```

- La déclaration permet d'utiliser la fonction sans la connaître.

- Les fonctions reçoivent toujours une copie des paramètres; elles ne travaillent jamais avec les originaux (passage des paramètres par valeur).

- Toute fonction qui ne retourne aucune valeur retourne le type void. Si la fonction ne reçoit aucun paramètre, on met alors le mot-clé void dans sa liste de paramètres lors de sa définition et de sa déclaration.

Fonctions (suite)

- Il n'est pas possible de définir une fonction à l'intérieur d'une autre. (Pas d'imbrication de fonction.)
- Les variables définies dans une fonction ne sont ni visibles ni accessibles depuis d'autres fonctions. Le compilateur maintient un espace de noms disjoint pour chaque fonction d'un programme. Seules les variables globales sont visibles à partir de plusieurs fonctions.

Énoncé return

Cet énoncé permet de terminer l'exécution d'une fonction, et facultativement de retourner une valeur.

syntaxe return [<expression>];

- Le résultat de l'évaluation de l'expression est converti dans le type retourné par la fonction (comme dans le cas d'une affectation) avant d'être retourné à l'appelant.

Énoncé conditionnel if - else

syntaxe if (<condition>) <énoncé> [else <énoncé>]

La condition est une expression quelconque donnant un résultat entier interprété comme une valeur booléenne. La différence entre une condition et une expression n'est pas syntaxique mais sémantique.

Exemple

```
if (i > 0)
        i = 0;
 else
    j = 5;
```

La clause else se rapporte toujours au dernier if le plus près du même niveau.

```
if ( condition 1 )
        if ( condition 2 )
            <énoncé>
        else
        <énoncé>
```

En pratique, on rencontre souvent les deux constructions ci-dessous.

```
if (i) .../* équivalent à if (i != 0) ... */
  if (! i) .../* équivalent à if (i == 0) ... */
```

- Ces constructions sont trop difficiles à saisir pour ceux qui n'utilisent pas fréquemment le langage C.
- De plus, elles n'accélèrent pas le calcul, car tout compilateur généra le même code selon que if `(i)` ou if `(i == 0)` est utilisé.

Énoncé conditionnel if - else (suite)

Une erreur fréquemment rencontrée est la suivante.

```
if (i = 0)
        ...
 else
     ...
```

Relation avec l'opérateur ternaire ? :

```
i3 = i1 > i2 ? i1 : i2;
```

est équivalent à

```
if (i1 > i2)
        i3 = i1;
  else
     i3 = i2;
```

Énoncé while

L'énoncé while permet de faire une boucle dont la condition de fin est vérifiée avant la première exécution du corps de la boucle.

syntaxe

```
while ( <condition> )
        <énoncé>
```

Exemple

```
int somme = i = 0;
 while (i < 10) {
      i++;
      somme += i;
 }
```

```
int somme = i = 0;
while (i < 10)
      somme += ++i;
```

Enoncé while (suite)

Autre exemple

•En C, il n'y a pas d'opérateur d'exponentiation, il faut utiliser la fonction `pow()` définie dans `math.h`.

• Réalisons une fonction plus efficace.

```
double puissance(double base, int exposant)
{
     double resultat;
     if (exposant < 0) {
          exposant = -exposant;
          base = 1.0 / base;
     }
     if ((exposant & 1) != 0)
          resultat = base;
     else
        resultat = 1.0;
     exposant >>= 1;
     while (exposant != 0) {
          base *= base;
          if ((exposant & 1) != 0)
              resultat *= base;
          exposant >>= 1;
     }
     return resultat;
}
```

Performance mesurée sur la machine RISC 6000 modèle 32H d'IBM avec 2 x 10^6 calculs identiques. (Les temps sont donnés en sec.)

exposant	3	6	12	18	21	24	32
puissance 1	92	2.3	2.69	3.07	3.07	3.07	3.48
pow	14.73	14.56	14.69	14.84	14.90	14.84	14.88

Énoncé do - while

Cette boucle ressemble à l'énoncé while, mais le corps de la boucle est toujours exécuté une fois. Ceci est très bien illustré par le fait que la condition s'écrit après le corps de la boucle.

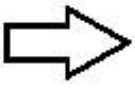
syntaxe	do <énoncé> while (<condition>);

Exemple

```
int somme = i = 0;
 do
   somme += i;
 while (i++ < 10);
```

Simplifions le code suivant

```
do {
   if (A) {
        B;
        C;
   }
} while (A);
```



```
while (A) {
     B;
     C;
}
```

Énoncé for

Cette boucle implante une boucle généralisée dans laquelle tous les éléments de contrôle sont rassemblés au même endroit.

syntaxe for ([<expression 1>] ; [<condition>] ; [<expression 2>])
<énoncé>

Cette boucle a la même sémantique que

```
<expression 1>;
while ( <condition> ) {
      <énoncé>
      <expression 2>
}
```

Exemples

```
int i, somme;
 somme = 0;
 for (i = 1; i <= 10; i++)
      somme += i;
```

ou encore

```
int i, somme;
  somme = 0;
  i = 1;
  for ( ; i <= 10; )
       somme += i++;
```

ou encore

```
int i, somme;
  somme = 0;
  i = 1;
  for ( ; i <= 10; somme += i++)
       ;
```

Énoncé for (suite)

Une instruction nulle (un énoncé vierge suivi de `;`) retourne toujours le résultat entier 1. Nous pouvons donc réaliser une boucle infinie par

for `( ; ; )`

Une erreur souvent commise est d'inclure un point-virgule devant le corps de la boucle. Dans ce cas, le corps de la boucle ne fait plus partie de la boucle; il sera exécuté une seule fois et ceci après avoir atteint la terminaison de la boucle.

Énoncé break

Cet énoncé interrompt le flot normal d'exécution et permet de sortir prématurément d'un énoncé.

Exemple

```
int i, somme = 0;
 for (i = 0; ; )
      if (i++ >= 10)
            break;
   else
        somme += i;
```

Ce dernier exemple est certainement la plus mauvaise implantation de notre boucle qui additionne les nombres 1 à 10.

L'énoncé break permet de sortir de n'importe quelle boucle for, do - while ou while, et aussi de l'énoncé switch.

Énoncé switch

Cet énoncé permet de construire une table de branchement selon une expression retournant une valeur entière.
Il constitue une alternative aux énoncés if - else imbriqués.

syntaxe

```
switch ( <expression entière> )
  {
       [ <clause> ]*
       [ default: <énoncé>+]
  }
```

avec

```
<clause> ::= [ case <expression constante> :]+ <énoncé>*
```

- L'expression constante d'une clause doit toujours être entière et évaluable à la compilation.
- Sa valeur doit être différente de toutes les autres.
- Les énoncés d'une clause comprennent ceux de la clause et tous ceux qui le suivent jusqu'à la fin du switch.
- Pour interrompre le flot, il faut utiliser l'énoncé break.

Enoncé switch (Exemples)

```
char c;

switch (c) {
     case 'A':case 'E':
     case 'O':case 'U':
        voyelle++;
        caracteres++;
        break;
    default:
      caracteres++;
}
```

case 'I':
case 'Y':

On aurait pu écrire cet exemple différemment.

```
switch (c) {
        case 'A':case 'E':case 'I':
        case 'O':case 'U':case 'Y':
            voyelle++;/* tombe dans default */
        default:
        caracteres++;
}
```

Autres énoncés

Il s'agit des énoncés continue et goto. Ce sont des vestiges des ancêtres du langage C, et leur utilisation n'est pas recommandée.

Brièvement, l'énoncé continue s'utilise dans une boucle pour interrompre l'itération courante et reprendre son exécution au test de la boucle.

Exemple

```
while (expression) {
        ...
        if (condition != 0)
            continue;
        ...
}
```

Quant à l'énoncé goto, ...

Types de données composés

Tableaux

Un tableau est un regroupement de données, toutes ayant le même type, et qui porte un nom.

syntaxe: <type> <identificateur> [[<expression constante>]]+;

Un tableau se déclare comme une variable ordinaire, mais son nom est suivi par autant de paires de crochets qu'il a de dimensions. L'expression contenue entre crochets est évaluée à la compilation et elle représente le nombre d'éléments pour sa dimension.

Le premier élément porte obligatoirement l'indice zéro. Ainsi une dimension déclarée comme ayant `N` éléments comprend les indices allant de 0 à `N-1`.

Exemple

```
char chaine[20];          /* tableau à une dimension */
int tab[10][2+2];         /* tableau de 2 dimensions */
```

Pour accéder à un élément d'un tableau, on spécifie son nom suivi par son indice :

```
tab[0][1] = .../* l'abréviation tab[0,1] n'existe pas */
```

- Il n'existe aucun opérateur pour manipuler un tableau comme un tout : il faut manipuler chacun de ses éléments individuellement.
- Aucun contrôle n'est fait sur le débordement des indices (ni à la compilation, ni à l'exécution).
- L'emmagasinage d'un tableau à plusieurs dimensions se fait en lignes - colonnes avec l'indice le plus à droite représentant la colonne.
  Exemple: int `t[2][3];`
  `t[0][0] t[0][1] t[0][2] t[1][0] t[1][1] t[1][2]`

Pointeurs

Malgré que les pointeurs fassent parties des types de données élémentaires, leur usage les place dans une classe à part. C'est pourquoi nous les traitons avec les types composés.

Un pointeur est une variable qui contient l'adresse d'une autre donnée.
Les pointeurs occupent une place très importante en C :

- il faut les utiliser pour modifier le contenu d'un paramètre passé à une fonction;
- le passage de tableau en paramètre à une fonction ne peut se faire qu'en passant un pointeur sur le tableau (c.-à-d. le nom du tableau).
- Il est possible d'avoir un pointeur de n'importe quel type de donnée (élémentaire ou composé).

La déclaration d'un pointeur se fait comme pour une variable du type qu'on désire pointer, mais on précède le nom de la variable par le caractère `*`. (Le caractère `*` ne fait pas partie du nom de la variable.)

Exemple

```
int i, *ip, *tp1[10], *tp2[10][10], (*pt)[10];
```

`i`	est un entier;
`*ip`	est un pointeur sur un entier;
`*tp1[10]`	est un tableau de 10 pointeurs d'entiers;
`*tp2[10][10]`	est un tableau à 2 dimensions de pointeurs d'entiers;
`(*pt)[10]`	est un pointeur à un tableau de 10 entiers;

Opérateur `&` ("adresse de")

Cet opérateur permet d'extraire l'adresse d'une variable (élémentaire ou composée).

Sa priorité est identique aux opérateurs `++, --, !` et `~,` et il est associatif de droite à gauche.

Exemple

```
int *x, *y,                 /* 2 pointeurs d'entiers    */
    tab[10];                /* tableau de 10 entiers    */
 x = &tab[0];               /* x = adresse de tab[0]    */
 y = &tab[4];               /* y = adresse de tab[4]    */
```

Opérateur `*` ("valeur de")

Lorsqu'un pointeur contient l'adresse d'une donnée, on peut accéder à cette donnée par l'opérateur `*`.

Si nous reprenons l'exemple précédent,

```
*x = 4;/* équivalent à tab[0] = 4*/
 *y = *x;/* équivalent à tab[4] = tab[0] */
```

- L'opérateur `*` a la même priorité que l'opérateur `&` et il est aussi associatif de droite à gauche. Ainsi

  `y = *x++` est équivalent à `y=*x; x++;`

  ce qui est différent de `x++;` suivi de `y=*x.`

Opérateur * ("valeur de") (suite)

- Si nous voulons qu'une fonction puisse modifier le contenu d'une variable, il faut passer l'adresse de la variable en paramètre.

Exemple

```
void IncrementInt(int *);


void main(void)
{
      int i;
      printf("la valeur de i est %d,",i);
      IncrementInt(&i);
      printf(" et maintenant elle vaut %d\n",i);
}




void IncrementInt(int *entier)
{
      *entier += 1;
}
```

Arithmétique sur pointeurs

Lorsque nous incrémentons un pointeur d'un type donné (par exemple dans un tableau), le pointeur est déplacé de la taille de ce type. Plus généralement, si `ip` pointe sur un élément d'un tableau, alors `ip+m` pointe `m` éléments plus loin que `ip` et `ip-n` pointe `n` éléments moins loin que `ip`.

Exemple

```
int *x, *y, tab[10];
 tab[4] = 4;
 x = &tab[0];
 y = x++;/* y=adresse de tab[0] et x=adresse de tab[1]
*y = *(x+3); /* tab[0]=tab[4]                                    */
*y = *x++;/* tab[0]=tab[1] et x=adresse de tab[2]                */
                                                                 */
```

Ainsi, lorsqu'on incrémente un pointeur d'entiers, son adresse est incrémentée de 2 si les entiers ont 16 bits et de 4 si les entiers ont 32 bits.

Autre exemple
(Rappelons que les tableaux sont emmagasinés en lignes-colonnes avec l'indice le plus à droite variant en premier).

```
int *x, tab[10][5];      /* tab contient 10 lignes de 5 colonnes */
int (*it)[5];            /* un pointeur sur un tableau de 5 int */
x = &tab[0][0];
x++;
x += 5;                  /* x = adresse de tab[0][1]            */
it = &tab[0][4];         /* x = adresse de tab[1][1]            */
it++;
it += 2;
                         /* it = adresse de tab[1][4]           */
                         /* it = adresse de tab[3][4]           */
```

Arithmétique sur pointeurs (suite)

- Lorsque nous disposons de 2 pointeurs sur des données d'un même type, on peut les comparer pour savoir s'ils sont égaux ou non. De plus, s'ils pointent dans un même tableau, on peut aussi utiliser les relations <, <=, > et >= pour déterminer les positions relatives de ces pointeurs.

Usage des pointeurs combinés avec les tableaux

•Contrairement aux variables de type élémentaire, on ne peut pas passer une copie des tableaux en paramètre à une fonction. Il faut toujours passer leur adresse (en spécifiant simplement le nom du tableau) et les récupérer comme des pointeurs ou des tableaux.

Exemple

```
/* Cette fonction permet de concaténer une seconde chaîne (ch2)
   à la fin de la première (ch1).
   Elle cherche d'abord la fin de la première chaîne, puis elle
   copie la seconde chaîne à la suite.
   * Usage: void Concat(char *ch1, char *ch2)
   Les 2 chaînes doivent se terminer par le caractère '\0' (nul).
   La première chaîne (ch1) doit être assez grande pour contenir
   le résultat.
   * Exemple d'utilisation :
           char chaine[50];
           chaine[0] = '\0';
           Concat(chaine,"Hello>");
           Concat(chaine," Comment allez-vous?");
*/
void Concat(char *ch1, char *ch2)
{
   while (*ch1++ != '\0') /* trouver la fin de ch1*/
          ;
   ch1--;/* reculer pour pointer sur le nul*/
   while ((*ch1++ = *ch2++) != '\0') /* concaténe la 2e chaine */
          ;
} /* fin de Concat */
```

Similitude entre tableaux et pointeurs

Il existe une similitude entre les tableaux et les pointeurs. Nous venons de voir que la fonction `Concat` reçoit les adresses des premiers indices des tableaux passés en paramètre :

```
chaine = &chaine[0] = ch1
```

On peut alors écrire

```
chaine[i] = *(chaine + i) = *(ch1 + i) = ch1[i]
```

Toutefois, il existe une petite différence entre pointeurs et tableaux :

un pointeur est une variable qui peut être modifiée, mais
le nom d'un tableau est une constante qui pointe vers le premier élément du tableau.

Types de données composés (suite)

Structures

Comme les tableaux, une structure est une façon de regrouper différentes données ayant un sens commun. Contrairement aux tableaux, les données d'une structure n'ont pas l'obligation d'être du même type. (Les structures C sont l'équivalent des articles Ada et des enregistrements Pascal.)

syntaxe

/* déclaration */struct <identificateur> { <définition>+ } ;
struct <identificateur> <liste d'identificateurs>; /* définition */

- La déclaration ne fait que dicter le squelette de la structure et ne définit pas d'occurrence. C'est lors de la définition de variables qu'il y aura des occurrences de la structure.

Exemple

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

- Il existe une forme abrégée permettant de rassembler la déclaration et la définition de ses occurrences en un seul énoncé :
  struct [< identificateur>] { <définition>+ } <liste d'identificateurs>;

- Sous cette forme, le nom de la structure n'est pas requis. Il doit cependant être présent si d'autres occurrences de la structure doivent être définies. Selon la norme ANSI, une structure doit obligatoirement porter un nom.

Structures (suite)

- Les noms des champs d'une structure doivent être uniques.
- Une structure peut aussi contenir d'autres structures. Toutefois elle ne peut pas contenir une occurrence à elle-même. Elle peut, par contre, contenir un pointeur vers une structure du même type qu'elle-même, ce qui permet d'implémenter des listes chaînées ainsi que d'autres structures nécessitantes des pointeurs.

Exemples

```
struct Personne {
        char nom[20];
        struct Date dateEmbauche;
};
```

```
struct Employes {
        int numero;
        struct Personne employe;
        struct Employes *prochain;
};
```

- L'initialisation des champs d'une structure peut se faire simultanément.
- Il est possible d'affecter une structure à une autre d'un seul coup.
- L'accès aux champs d'une structure se fait en qualifiant le champs par le nom de la structure suivi de l'opérateur . (point).

Exemple

```
struct Personne jean, jules, *sosie;
jean.dateEmbauche.jour = 25;
jules = {"Jules",{20, 1, 97}};
jean = jules;
```

Structures (suite)

Lorsque des pointeurs de structures sont utilisés, l'opérateur . (point) est plus prioritaire que l'opérateur * (valeur de).

Exemple

```
sosie = &jean;
 (*sosie).dateEmbauche.jour = 3;
```

- Les pointeurs de structures sont utilisés tellement souvent qu'un opérateur supplémentaire existe pour éviter de mettre les parenthèses. Il s'agit de l'opérateur `->`.

  ```
  sosie->dateEmbauche.jour = 3;
  ```

- En ANSI C, il est possible de passer une structure en paramètre et de retourner une structure comme résultat. Il n'est pas nécessaire, comme en C, de passer par le biais de pointeurs.

Enoncé typedef

L'énoncé typedef permet de créer des synonymes à partir de types de données.

Exemple

```
typedef int *PINT;
 PINT p;
                          /* équivalent à int *p */
```

Exemple

Parcours d'un arbre en pré-ordre (c.-à-d. selon le sens: sous-arbre de gauche, noeud, sous-arbre de droite)

```
typedef struct Noeud {
      int valeur;
      struct Noeud *gauche, *droite;
} NOEUD;
```

```
int ParcourtEnOrdre(NOEUD *arbre)
{
      if (arbre->gauche != NULL)
         ParcourtEnOrdre(arbre->gauche);
      printf("%d ",arbre->valeur);
      if (arbre->doite != NULL)
         ParcourtEnOrdre(arbre->doite);
} /* fin de ParcourtEnOrdre */
```

Type énuméré

Ils permettent de nommer certaines valeurs constantes que peut prendre une variable entière. Le but étant d'écrire des programmes plus clairs.

syntaxe de la définition

<type_énuméré> ::= enum [<identificateur>] [<énumération>] ;
<énumération> ::= { <identificateur_d' énumération>
[,<identificateur_d' énumération>]* }
<identificateur_d' énumération > ::= <identificateur > [= <expression>]

Exemple

```
enum carte {coeur, carreau, trefle, pique};
enum carte sorte;
sorte = carreau;
```

Les variables définies à l'aide d'un type énuméré sont traitées exactement comme des variables de type entier. Il n'y a aucun moyen de demander à `printf()` d'écrire l'identificateur associé à une valeur d'une variable énumérée.

La définition de `carte` est traitée comme une variable entière. Les valeurs associées forment une suite arithmétique commençant par 0 et ayant un pas de 1. Ainsi, `coeur` vaut 0, `carreau` 1, etc.

Type énuméré (suite)

Pour briser la séquence, nous pouvons inclure une expression entière dans l'énumération des valeurs.

Exemple

```
enum carte {coeur, carreau = 5, trefle, pique};
enum carte sorte = carreau;
```

Ces 2 lignes sont équivalentes à

```
#define coeur      0
#define carreau    5
#define trefle     6
#define pique      7
int sorte = carreau;
```

Historiquement, les types énumérés sont une des premières extensions proposées au langage C décrite par Kernighan et Ritchie. Ils font partie de la norme ANSI. Cependant, ils ne sont pas implémentés dans les anciens compilateurs.

Type union

Le type `union` correspond à la partie variante des enregistrements Pascal ou des articles Ada. Cependant, en C, elles sont moins sécures puisqu'il n'y a aucun discriminant.

La syntaxe de définition est semblable à celle d'une structure:

syntaxe

union <identificateur> { <définition>+ }; /* déclaration */
union <identificateur> <liste d'identificateurs>; /* définition */

Exemple

```
struct fleurs {
    char *nom;
    enum {rouge, bleu, blanc} couleur;
    char *senteur;
};
struct fruits {
    char *nom;
    int calories;
};
union plante {
    struct fleurs fleur;
    struct fruits fruit;
};
```

Remarques

Pour une utilisation sûre, il vous appartient d'implémenter vous-même un discriminant.

Contrairement à Pascal et à Ada, la partie variante peut se placer n'importe où dans une structure.

Attribution dynamique de mémoire

Il n'est pas rare que la quantité de mémoire requise pour un tableau ne soit connue que lors de l'exécution. Plutôt que
•de déclarer une quantité excessivement grande (gaspillage),
•ou d'imposer une limite supérieure à la quantité de mémoire,
on peut allouer dynamiquement la mémoire nécessaire.

Quatre fonctions sont définies dans `stdlib.h` et dans `alloc.h`:

```
void *malloc(size_t nbOctets);
void *calloc(size_t nbCellules, size_t nbOctetCellule);
void *realloc(void *ptrMemoire, size_t nbOctets);
void free(void *ptrMemoire);
```

Le fait que l'allocation retourne un pointeur sur rien, oblige une conversion explicite de type.

`malloc(), calloc()` et `realloc()` retourne `NULL` s'il ne reste pas suffisamment de mémoire à disposition.

`calloc()` est équivalent à `malloc(nbElements * nbOctetCellule)` et initialise, contrairement à `malloc()`, la mémoire allouée à 0.

`realloc()` modifie la taille déjà affectée.

`free()` libère une zone mémoire acquise via `malloc()`.

Attribution dynamique de mémoire (suite)

Exemple

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


void main(void)
{
     char *chaine;
     chaine = (char *)malloc(10*sizeof(char));
     strcpy(chaine,"Hello");
     printf("La chaine vaut %s\net son adresse est %p\n",
                  chaine,chaine);
     chaine = (char *)realloc(chaine,20*sizeof(char));
     printf("La chaine vaut %s\net son adresse est %p\n",
                  chaine,chaine);
     free(chaine);
}
```

Attribution dynamique de mémoire (suite)

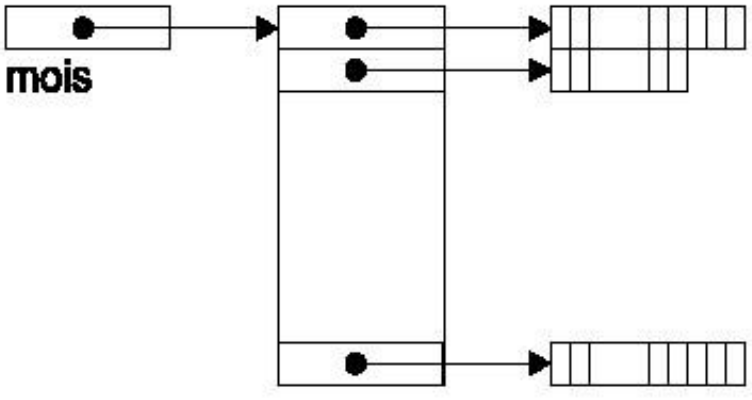
Pointeurs aux tableaux

```
#include <stdlib.h>
void main(void)
{
     int **mois, i;
     int jour[12] = {31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31};



     if ((mois = (int **)malloc(12*sizeof(int *))) != NULL)
         for (i = 0; i < 12; i++)
              mois[i] = (int *)malloc(jour[i]*sizeof(int));

}
```



Attribution dynamique de mémoire (suite)

Allocation dynamique d'un tableau 3D

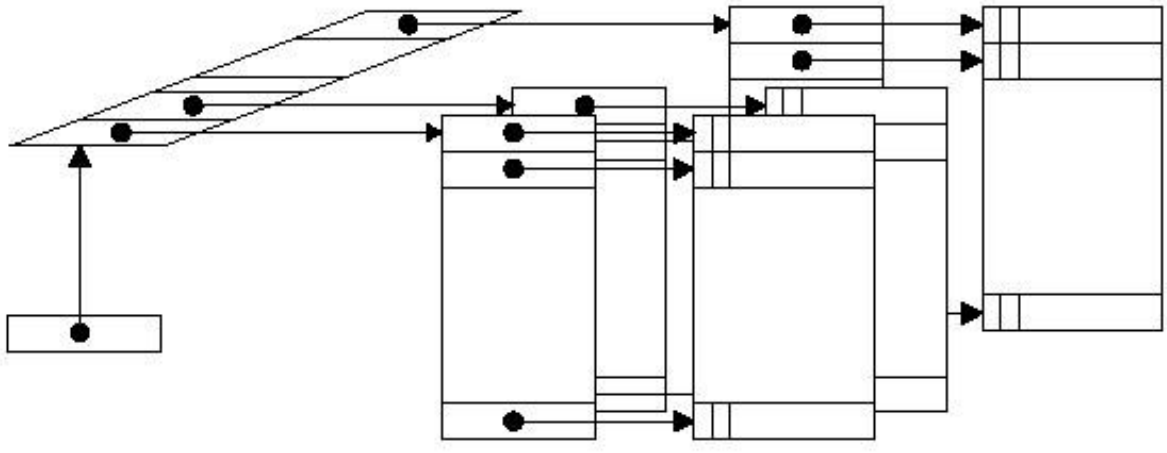
```
#include <stdlib.h>
int ***AllocTableau3D(int dim1, int dim2, int dim3)
{
  int i, j;
  int ***ptr;
  if ((ptr = (int ***)malloc(dim1*sizeof(int **))) == NULL)
      return NULL;
  for (i = 0; i < dim1; i++) {
       if ((ptr[i] = (int **)malloc(dim2*sizeof(int *))) == NULL)
           return NULL;
       for (j = 0; j < dim2; j++)
           if ((ptr[i][j] = (int *)malloc(dim3*sizeof(int))) == NULL)
                return NULL;
  }
  return ptr;
} /* fin de AllocTableau3D */
```

Cette fonction peut s'utiliser comme suit

```
void main(void)
{
     int ***monTableau;
     if ((monTableau = AllocTableau3D(10,14,18)) != NULL)
          monTableau[2][3][4] = 5;
}
```

Attribution dynamique de mémoire (suite)

Allocation dynamique d'un tableau 3D (suite)



Exercice
Écrirez une fonction qui libère le tableau 3D alloué par `AllocTableau3D()`.

Source des erreurs
- La libération et la modification d'une zone de mémoire remettent cette mémoire pour d'autres utilisations, la réutiliser est dangereux.
- A la terminaison de l'exécution d'un programme, la mémoire dynamique est rendue au système. Pour la libérer avant, il faut appeler une seule fois la fonction `free()`.
- Libérer un pointeur `NULL` est fatal.

Structures de données complexes : listes chaînées

Une liste chaînée est un ensemble ordonné de données homogènes. À la différence d'un tableau, les éléments ne sont pas nécessairement contigus en mémoire.
Une liste chaînée permet d'atteindre une plus grande efficacité dans l'exécution des opérations de gestion. Par exemple, l'insertion ou le retrait d'articles dans la liste ne requiert aucun transfert d'information, seulement l'affectation de pointeurs.
Une liste chaînée est dite à lien simple lorsque chaque article n'utilise qu'un seul pointeur pour constituer la liste. Une telle liste ne peut, par conséquent, se parcourir que dans un sens. La fin de la liste s'identifie au moyen du lien du dernier article qui contient un pointeur `NULL`.

À partir d'une liste chaînée à lien simple, nous pouvons définir 2 nouvelles structures :
- Une file est une liste dont l'insertion et le retrait des articles se réalisent à des extrémités différentes.
- Une pile est une liste dont l'insertion et le retrait des articles se réalisent à la même extrémité.

Structures de données complexes : listes chaînées

Exercice 1

Réaliser un ensemble de fonctions pour l'insertion et le retraite d'articles et la gestion d'une liste.

```
typedef struct Element {
  int article;
  struct Element *suivant;
} ELEMENT;

typedef struct Liste {
  ELEMENT *tete, *queue;
} LISTE;

LISTE *InsertFile(LISTE *liste, int article);
LISTE *InsertPile(LISTE *liste, int article);
int RetireFile(LISTE *liste);
int RetirePile(LISTE *liste);
void EffaceListe(LISTE *liste);
int CompteElementsList(LISTE *liste);
```

Exercice 2

Réaliser un ensemble de fonctions permettant la gestion d'une liste triée.

```
typedef struct Element {
      int cle;
      struct Element *suivant;
 } ELEMENT;

typedef struct Liste {
  ELEMENT *tete, *queue;
} LISTE;
```

Déclarations complexes de types

Aussitôt que nous essayons de déclarer des types quelque peu compliqués, nous commençons à avoir des difficultés avec la syntaxe. La seule façon de s'en sortir est de se fier à la priorité des opérateurs.

Exemples (rappel)

```
int   a;     /* a est un entier                                  */
int   *b;    /* b est un pointeur d'entier                       */
int   c();   /* c est une fonction (ceci n'est pas ANSI)         */
int   d[];   /* d est un tableau d'entiers                       */
```

Mais que dire de

int `*e();`

Ceci est équivalent à int `*( e() )`, et c'est donc une fonction qui retourne un pointeur vers un entier.

Pour obtenir un pointeur vers une fonction qui retourne un entier, il faut écrire

int `(*f)();`

car dans ce cas, `f` est un pointeur avant d'être une fonction.

De la même manière et en notant que les crochets carrés sont plus prioritaires que l'étoile, on écrit

int `*g[];`

pour obtenir un tableau de pointeurs d'entiers.

Par contre,

int `(*h)[];`

est un pointeur avant d'être un tableau. Il s'agit donc d'un pointeur vers un tableau d'entiers.

Pour déclarer un tableau de pointeurs de fonctions qui retournent des entiers, il faut écrire

int `(*h[])();`

Pointeurs de fonctions

Ils sont intéressants lorsque nous désirons généraliser un algorithme ou une fonction.

Exemple

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


void TraceFonction(double (*F)(double),            /* fonction a tracer   */
                   double,                         /* limit inferieure    */
                double,                            /* limit superieure    */
                int);                              /* nombre de points    */



void main(void)
{
  TraceFonction(sin,2.0,10.0,20);
  TraceFonction(cos,2.0,10.0,20);
}



typedef struct POINT{
    float x, y;
};
typedef struct POINT POINT;



void TraceFonction(double (*F)(double), double limInf, double limSup, int nbPoints)
{
  POINT *points;
  double inc, x;
  int i;



  points = (POINT *)malloc(nbPoints * sizeof(POINT));
  inc = (limSup - limInf) / (double)(nbPoints - 1);
  for (i = 0, x = limInf; x <= limSup; x += inc, i++) {
     points[i].x = x;
     points[i].y = F(x);
  }
  PolyLine(nbPoints,points);


}
```

Pointeurs de fonctions (suite)

Pour simplifier l'écriture, nous avons les formes équivalentes:

- entêtes de fonctions

```
void TraceFonction(double (*F)(double), double limInf, double limSup,
                   int nbPoints)
void TraceFonction(double F(double), double limInf, double limSup,
                   int nbPoints)
```

- lignes de prototype

```
void TraceFonction(double (*F)(double), double, double, int);
void TraceFonction(double F(double), double, double, int);
void TraceFonction(double (*)(double), double, double, int);
```

Fonctions d'entrées-sorties

En C, les entrées-sorties ne font pas partie du langage. Elles sont fournies par l'intermédiaire d'une programmathèque du système, distribuée avec le compilateur. Les 2 fonctions élémentaires qui permettent de lire et d'écrire des caractères individuels au terminal sont `getchar()` et `putchar()`, et les fonctions qui permettent de faire des entrées-sorties avec format sont `printf()` et `scanf()`.

Lorsqu'on utilise les routines du système pour effectuer des entrées-sorties, il faut faire un `#include` d'un fichier système au début du programme. Celui-ci s'appelle presque toujours `stdio.h` (pour STanDard Input/Output Header file : fichier d'en-tête pour les entrées-sorties standard).

Fonction getchar()

La fonction `getchar()` retourne le prochain caractère lu au clavier :

```
int getchar(void);
```

Si aucun caractère n'a encore été entré lors de l'appel de cette fonction, le programme attend le caractère. Si la fin du fichier a été détectée (un concept plutôt bizarre sur un terminal), la valeur retournée est la constante `EOF` définie dans le fichier `stdio.h`.

Fonction putchar()

La fonction `putchar()` est utilisée pour écrire au terminal le caractère qu'elle reçoit en paramètre :

```
int putchar(int);
```

Si la fonction réussit, elle retourne la valeur du caractère affiché, sinon elle renvoie la valeur `EOF`.

Fonctions d'entrées-sorties (suite)

Exemple

```
#include <stdio.h>
void main(void)
{
    int c;
    while ((c = getchar()) != EOF) {
        putchar(c);
        /* traiter le caractere */
    }
}
```

Bien que l'on lise des caractères, il faut définir `c` comme un entier. Ceci est nécessaire car la constante `EOF` vaut `-1`, ce qui est différent du caractère `(unsigned char)-1 = 255`.

Affichages formatés

Comme nous l'avons vu, la fonction `printf()` permet de formater un affichage. Il est possible de préciser davantage le format de sortie en faisant suivre le caractère '%' des modificateurs optionnels suivants :

```
%[signe][taille][.prec]type
```

L'indicateur `signe` peut prendre l'un des caractères suivants :

`-` pour ajuster à gauche plutôt qu'à droite;

`+` précède une valeur numérique par son signe (+ ou -);

espace précède une valeur numérique négative par son signe et les positives par un espace.

La taille du champ se fait par `taille` :

`n` nombre de caractères minimaux du champ;

`0n` idem mais affiche des zéros non significatifs (ex: 0005).

La précision se fait par `prec` :

`nnn` nombre de chiffres après le point décimal ou le nombre maximal de caractères de la chaîne qui sont affichés.

Finalement, `type` est le code de conversion proprement dit :

- `c` est pour les char
- `s` est pour les chaînes de caractères
- `d, i, o, x, X` et `u` sont pour des int
- `hd, ho, hx, hX` et `hu` sont pour des short int
- `ld, lo, lx, lX` et `lu` sont pour des long int
- `f, e` et `E` sont pour des float
- `lf, le` et `lE` sont pour des double
- `Lf, Le` et `LE` sont pour des long double
- `p` est pour l'affichage de pointeurs

Affichages formatés (suite)

Exemples:

```
printf("1-|%12s|\n","C'est du C!");
 printf("2-|%-12s|\n","C'est du C!");
 printf("3-|%5s|\n","C'est du C!");
 printf("4-|%-5s|\n","C'est du C!");
 printf("5-|%.5s|\n","C'est du C!");
 printf("6-|%-.5s|\n","C'est du C!");
 printf("7-|%12.5s|\n","C'est du C!");
 printf("8-|%-12.5s|\n","C'est du C!");
```

donnent:

```
1-|  C'est du C!|
 2-|C'est du C!  |
 3-|C'est du C!|
 4-|C'est du C!|
 5-|C'est|
 6-|C'est|
 7-|C'est|
 8-|C'est|
```

Lectures formatées

La fonction `scanf()` permet des lectures avec format et rend beaucoup plus facile la traduction en format machine les entités lues. Elle fonctionne comme `printf()`, mais décortique les entrées pour les mettre dans différentes variables, plutôt que de les afficher.
int `scanf(const char *format [, variable]*);`

Le premier paramètre est le format dont il faut interpréter la lecture, et les paramètres suivants sont les variables qu'il faut modifier. Puisqu'il s'agit d'une fonction modifiant des paramètres, il est impératif de toujours passer l'adresse des variables.

Dans le cas particulier où on passe en paramètre le nom d'un tableau, il ne faudrait pas utiliser l'opérateur `&`, car le nom du tableau représente déjà son adresse :
char `chaine[20];`

```
...
scanf("%s",&chaine[0]);/* pointer sur chaine[0] */
scanf("%s",chaine);/* idem */
```

La fonction `scanf()` retourne un entier qui indique le nombre de conversions effectuées. Ce nombre peut être plus petit que le nombre demandé s'il y a une erreur lors de la lecture ou si l'entrée est incomplète. Si la chaîne à décortiquer se termine avant que toutes les conversions aient été réalisées, la fonction retourne la valeur `EOF`.

Les caractères qui ne sont pas des codes de conversion doivent être lus tels quels. Un espace vide dans le format entraîne le saut de tous les espaces vides apparaissant dans la chaîne lue. Un espace vide inclut toute suite d'espaces, de tabulateurs et de caractères de fin de ligne.

Exemple :

```
char chaine[20];
int entier;
scanf("%dqq%s",&entier,&chaine[0]);
```

Si les caractères `"55qqHello"` sont fournis en entrée, il y affectation de la valeur 55 à `entier` et de `"Hello"` à `chaine`, et la fonction retourne la valeur 2 pour indiquer que deux conversions ont été réalisées. Avec la chaîne `"55Hello"` en entrée, il y a l'affectation de 55 à `entier`, et `scanf()` termine son traitement car elle ne trouve pas les 2 caractères `'q'` requis. La valeur de retour est alors 1.

Lectures formatées (suite)

Spécification du format de conversion d'un champ

```
%[*][taille_maximale]type
```

`*` indique que la conversion doit avoir lieu mais que son résultat n'est pas affecté à une variable. Elle sert à sauter les caractères en entrée;

Lorsque la taille maximale du champ est précisée, il n'y a que ce nombre de caractères lus pour ce champ :

```
scanf("%2s%2d%e",&chaine[0],&entier,&flottant);
```

Avec les données `"12345"`, la valeur `"12"` est affectée à `chaine`, la valeur 34 est affectée à `entier` et 5.0 à `flottant`.

Parmi les types de conversion, il y a 2 formats de conversion en plus de ceux disponibles à `printf()` et qui sont particuliers à `scanf()` :

1. la conversion d'ensemble d'inclusion de caractères

   Seuls les caractères entre les crochets peuvent composer une chaîne de caractères.

```
scanf("%[ 123456789]",&chaine[0]);
```

   avec les données `"23 45 67 0 33"`, la chaîne `"23 45 67 "` est affectée à la variable `chaine`. Les caractères restants sont conservés pour un appel subséquent à `scanf()`. On peut aussi utiliser le trait d'union pour indiquer une suite de caractères:

```
scanf("%[ 1-9]",&chaine[0]);
scanf("%[a-b1-9]",&chaine[0]);
```

2. la conversion d'ensemble d'exclusion de caractères

   Seuls les caractères ne figurant pas entre les crochets peuvent composer une chaîne de caractères. Ainsi

```
scanf("%[^1-9]",&chaine[0]);
```

   avec les données `"a B cD023"`, la chaîne `"a B cD0"` est affectée à la variable `chaine`.

Entrées-sorties sur fichiers

En C il existe 2 types de fichiers : les fichiers textes et les fichiers binaires.
Les fichiers textes présentent les caractéristiques suivantes :

1. Chaque élément du fichier est un caractère ASCII. Ces caractères sont structurés en lignes, c.-à-d. en séquences de longueur variable qui se terminent par les caractères de contrôle <CR> et <LF>.
2. L'accès à l'information est séquentiel : la position physique d'une ligne particulière ne peut être déterminée autrement que par la lecture de toutes les données à partir du début du fichier.
3. Les informations sont conservées sous une forme intelligible et elles peuvent être modifiées à l'aide d'un éditeur de texte. Ceci est intéressant pour la mise au point de programmes.
4. Les fonctions d'entrées-sorties nécessitent des formats de conversion pour transformer une suite de caractères en une représentation compatible avec le type des variables à affecter et inversement. Cette conversion demande un traitement supplémentaire. De plus, les variables numériques représentées par des chaînes de caractères occupent, en moyenne, plus d'octets qu'une représentation binaire.

Peu importe le type du fichier qu'on désire traiter, il y a toujours 3 étapes :

- l'ouverture du fichier;
- les opérations d'entrées-sorties;
- la fermeture du fichier.

Entrées-sorties sur fichiers (suite)

Ouverture de fichiers

```
FILE *fopen(const char *nomFichier, const char *mode);
```

La fonction `fopen()` retourne un pointeur de fichier qu'on devra utiliser dans les appels subséquents aux fonctions de lecture ou d'écriture pour identifier un fichier parmi les autres qui pourraient être ouverts au même moment. Si une erreur est détectée lors de l'ouverture du fichier, `fopen()` retourne la valeur `NULL`.

Le premier paramètre passé à `fopen()` est le nom du fichier à ouvrir et se donne sous la forme d'une chaîne de caractères. Ce nom peut être absolu ou relatif :

fichier dans le répertoire `repa:\rep\fichier.dat`

même fichier si le programme s'exécute dans `repfichier.dat`

Le second paramètre représente le type d'accès. Ce paramètre s'introduit également sous la forme d'une chaîne de caractères :

mode	Ouvre le fichier
`"r"`	en lecture seulement; le fichier doit exister sinon il y a erreur;
`"w"`	en écriture; le contenu du fichier est détruit si le fichier existe ou le fichier est créé s'il n'existe pas;
`"a"`	en écriture à partir de la fin du fichier; le fichier est créé s'il n'existe pas.
`"r+"`	en lecture et en écriture; le fichier doit exister.
`"w+"`	en lecture et en écriture; le contenu du fichier est détruit si le fichier existe, sinon le fichier est créé.
`"a+"`	en lecture et en écriture; si le fichier n'existe pas, il est créé. Tout le fichier peut être lu, mais l'écriture ne peut se faire qu'à partir de la fin.

Fermeture de fichier

Lorsque le traitement du fichier est terminé, on le ferme par

```
int fclose(FILE *fp);
```

où `fp` est le pointeur du fichier retourné par `fopen()`. Le résultat de la fonction est 0 s'il n'y a pas d'erreur, et il est `EOF` autrement.

Entrées-sorties sur fichiers (suite)

Écriture et lecture d'un fichier

Les énoncés d'écriture et de lecture sont équivalents à ceux pour l'affichage à écran, sauf qu'ils prennent les formes suivantes :

int `fprintf(FILE *fp, const char *format [, variable]*);`

int `fscanf(FILE *fp, const char *format [, variable]*);`

Lors d'une lecture par `fscanf()`, les caractères contenus dans le fichier sont convertis en un type correspondant au format de conversion. Un pointeur indique là où se réalisent les lectures. Après une lecture, ce pointeur se déplace au caractère qui suit le format, préparant ainsi la prochaine entrée. Lors d'une écriture, les données à écrire dans le fichier sont converties en caractères avant d'être insérées à l'endroit indiqué par le pointeur d'écriture. Ce pointeur est ensuite déplacé au caractère qui suit le dernier caractère inséré.

Il existe d'autres fonctions d'entrées-sorties qui sont applicables aux fichiers :

int `fgetc(FILE *fp);`

int `fputc(int c, FILE *fp);`

agissent de la même manière que `getchar()` et `putchar(c)`.

Entrées-sorties sur fichiers (suite)

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


void main(void)
{
  FILE *source;
  int c;



  if ((source = fopen("exemple.c","r")) == NULL) {
     printf("Le fichier source n'existe pas\n");
     exit(1);
  }
  while ((c = fgetc(source)) != EOF)
     printf("%c",c);
  fclose(source);



}
```

Pour la manipulation de chaînes de caractères, les fonctions vues ne sont pas commodes :

`getc()` ne lit qu'un caractère à la fois, et

`scanf()` filtre les caractères d'espacement.

Entrées-sorties sur fichiers (suite)

Fonctions de la famille gets() et puts()

Ces fonctions agissent sur des lignes d'information : elles lisent et écrivent des lignes complètes.

char `*fgets(char *tampon,` int `tailleTampon, FILE *fp);`
char `*gets(char * tampon);`

`fgets()` s'assure que le tampon fourni est assez grand pour contenir les caractères lus. La fonction arrête de lire quand `tailleTampon-1` caractères sont lus ou quand une fin de ligne est lue. Par contre, get() lit les caractères en entrée et les emmagasine dans le tampon jusqu'à ce qu'une fin de ligne soit lue. `fgets()` laisse le caractère `'\n'` dans le tampon, alors que `gets()` l'enlève.

Les 2 fonctions `fgets()` et `gets()retournent` l'adresse du tampon si tout se passe correctement, ou encore la valeur `NULL` s'il y a une erreur avant la lecture du premier caractère, ce qui indique généralement une fin de fichier.

int `fputs(const` char `*tampon, FILE *fp);`
int `puts(const` char `* tampon);`

Ces 2 fonctions agissent d'une manière cohérente : `puts()` ajoute un caractère `'\n'` à la fin du contenu du tampon alors que `fputs()` ne le fait pas. Ces 2 fonctions retournent la valeur EOF si une erreur se produit et elles donnent une valeur non négative autrement.

Entrées-sorties sur fichiers (suite)

Exemple

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


#define TAILLETAMPON 256


void main(void)
{
   char tampon[TAILLETAMPON];
   FILE *source;



   if ((source = fopen("exemple.c","r")) == NULL) {
      printf("Le fichier source n'existe pas\n");
      exit(1);
   }
   while (fgets(tampon,TAILLETAMPON,source) != NULL)
      printf("%s",tampon);
   fclose(source);



}
```

Fichiers binaires

Les fichiers binaires sont caractérisés par les propriétés suivantes.

1. Les éléments contenus dans le fichier sont représentés de la même façon que les données en mémoire. Si un entier occupe 4 octets en mémoire, une variable entière occupe aussi 4 octets dans un binaire quelle que soit sa valeur.
2. L'accès à l'information peut être direct ou séquentiel.
3. L'information est sous une forme codée et n'est généralement pas intelligible lorsqu'elle est affichée par un éditeur de texte.
4. Les fonctions d'entrées et de sorties ne réalisent aucune conversion. Le transfert de données numériques est beaucoup plus rapide que pour les fichiers textes.

L'ouverture de fichiers binaires utilise la même fonction d'ouverture de fichiers que nous avons vue précédemment sauf que le type d'accès diffère.

```
FILE *fopen(const char *nomFichier, const char *mode);
```

Le type d'accès peut prendre les chaînes de caractères suivantes :

mode	Ouvre le fichier
`"rb"`	en lecture seulement; le fichier doit exister sinon il y a erreur;
`"wb"`	en écriture; le contenu du fichier est détruit si le fichier existe ou le fichier est créé s'il n'existe pas;
`"ab"`	pour écriture à partir de la fin du fichier; le fichier est créé s'il n'existe pas.
`"rb+"`	en lecture et en écriture; le fichier doit exister.
`"wb+"`	en lecture et en écriture; le contenu du fichier est détruit si le fichier existe, sinon le fichier est créé.
`"ab+"`	en lecture et en écriture; si le fichier n'existe pas, il est créé. Tout le fichier peut être lu, mais l'écriture ne peut se faire qu'à partir de la fin.

Les 3 derniers modes peuvent aussi s'écrire `"r+b"`, `"w+b"` et `"a+b"`.

Fichiers binaires (suite)

Lecture et écriture

```
size_t fread(void *tampon, size_t tailleElement,
             size_t nbElements, FILE *fp);
size_t fwrite(const void *tampon, size_t tailleElement,
             size_t nbElements, FILE *fp);
```

Ces fonctions nécessitent 4 paramètres :

- `tampon` est l'adresse de l'élément à lire ou à écrire;
- `tailleElement` est le nombre d'octets d'un élément;
- `nbElements` est le nombre d'éléments à lire ou à écrire;
- `fp` est le pointeur vers le fichier préalablement ouvert.

Ces 2 fonctions retournent le nombre d'éléments traités. En cas d'erreur, la valeur de retour est plus petite que `nbElements`.

Exemple

```
int tab[2];
...
fread(tab,sizeof(int),2,fp);
```

ou

```
fread(tab,2*sizeof(int),1,fp);
```

Fichiers binaires (suite)

Positionnement des pointeurs de lecture et d'écriture

Il existe plusieurs fonctions particulièrement conçues pour la gestion du pointeur de lecture et d'écriture des fichiers binaires. (Ces fonctions sont aussi disponibles pour les fichiers textes.)

int `fseek(FILE *fp,` long `decalage,` int `methode);`

Cette fonction permet de positionner le pointeur de lecture ou d'écriture à un endroit correspondant au nombre d'octets précisé dans le décalage. Cette valeur peut être positive ou négative. La position de départ est le 3e paramètre et elle peut être définie depuis

le début du fichier;`SEEK_SET`
la position courante du pointeur;`SEEK_CUR`
la fin du fichier.`SEEK_END`

Si la fonction réussit, la valeur retournée est 0 et, en cas d'erreur, elle est différente de zéro.

long int `ftell(FILE *fp);`

Cette fonction retourne la position courante du pointeur de lecture ou d'écriture en octets. En cas d'erreur, la valeur -1L est retournée.

Le premier élément d'un fichier binaire se trouve toujours à la position 0 octet à partir du début. Ainsi, pour un fichier de n octets, le dernier élément se trouve à la position n−1.

Fichiers binaires : Exemple

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


void main(void)
{
  FILE *donnees;
  int i;



  if ((donnees = fopen("exemple.tmp","wb+")) == NULL) {
     printf("Probleme de creation de exemple.tmp\n");
     exit(1);
  }
  /* Ecriture des nombres 10, 20, 30, 40 et 50 */
  for (i = 10; i < 60; i += 10)
     if (fwrite(&i,sizeof(int),1,donnees) != 1) {
        printf("Probleme d'ecriture\n");
        fclose(donnees);
        exit(1);
     }
  printf("Ecrit %d nombres.\n",ftell(donnees)/sizeof(int));
  /* Affichage de chaque valeur ainsi que sa position */
  printf("PositionValeur\n"
         "------------------\n");
  fseek(donnees,0,SEEK_SET);
  while (fread(&i,sizeof(int),1,donnees) > 0)
     printf("%4ld %10d\n",ftell(donnees)/sizeof(int)-1,i);
  /* Lecture de la 3e valeur */
  fseek(donnees,2*sizeof(int),SEEK_SET);
  if (fread(&i,sizeof(int),1,donnees) > 0)
     printf("La 3e valeur est %d.\n",i);
  /* Remplacer la 3e valeur par 60 */
  fseek(donnees,2*sizeof(int),SEEK_SET);
  i = 60;
  fwrite(&i,sizeof(int),1,donnees);
  /* Affichage de chaque valeur ainsi que sa position */
  printf("PositionValeur\n------------------\n");
  fseek(donnees,0,SEEK_SET);
  while (fread(&i,sizeof(int),1,donnees) > 0)
     printf("%4ld %10d\n",ftell(donnees)/sizeof(int)-1,i);
  fclose(donnees);









}
```

Fichiers binaires : Exemple (suite)

L'affichage est

```
Ecrit 5 nombres.
PositionValeur
------------------
   010
   120
   230
   340
   450
La 3e valeur est 30.
PositionValeur
------------------
   010
   120
   260
   340
   450
```

Fonctions de manipulation de chaînes de caractères

Les chaînes de caractères sont souvent utilisées dans des programmes, et un ensemble de fonctions est fourni par la programmathèque C : `string.h`.

Rappel : Une chaîne de caractères est représentée dans un tableau de caractères qui se termine par le caractère nul ('\0').

Fonction `strlen()`

```
size_t strlen(const char *chaine);
```

retourne la longueur de la chaîne de caractères sans compter le caractère nul final. Celle-ci pourrait être implémentée comme suit :

```
size_t strlen(const char *chaine)
  {
     size_t i = 0;
     while (*chaine++ != '\0')
           i++;
     return i;
  }
```

Au lieu d'ajouter 1 à une variable à chaque passage dans la boucle, on peut se servir de la soustraction de pointeurs :

```
size_t strlen(const char *chaine)
  {
      char *sauveChaine = chaine;
      while (*chaine++ != '\0')
            ;
      return chaine - sauveChaine - 1;
  }
```

Fonctions de manipulation de chaînes de caractères (suite)

Fonction `strcat()`

`char *strcat(char *chaine1, const char *chaine2);`

ajoute une copie des caractères composant la chaîne `chaine2` à la fin de la chaîne `chaine1`. La fonction retourne un pointeur vers la chaîne résultante, soit `chaine1`.

Fonction `strncat()`

`char *strncat(char *ch1, const char *ch2, size_t maxChars);`

fait la même chose que `strcat()`, mais elle copie au plus `maxChars` caractères. Par exemple, si `ch1` contient `"abc"` et `ch2` contient `"def"`, l'appel `strncat(ch1,ch2,2)` ajoute `"de"` à la fin de `ch1` et retourne un pointeur vers `"abcde"`. Dans notre exemple, nous avons ajouté 2 caractères à la chaîne résultante.

Fonction `strcmp()`

`int strcmp(const char *chaine1, const char *chaine2);`

retourne un entier négatif, nul ou positif selon que la chaîne `chaine1` est respectivement inférieure, égale ou supérieure à la chaîne `chaine2`. La comparaison se fait caractère par caractère, de gauche à droite, jusqu'à ce qu'on trouve une différence ou la fin d'une des deux chaînes.

Exemples :`"abcd" < "abz"` car

```
'a' == 'a', 'b' == 'b', 'c' < 'z'
```

`"abc" > "ab"` car

```
'a' == 'a', 'b' == 'b', 'c' > '\0'
```

Fonctions de manipulation de chaînes de caractères (suite)

Fonction `strncmp()`
int `strncmp(const` char `*ch1,` const char `*ch2, size_t maxChars);`
réalise la même chose que `strcmp(),` mais en comparant au plus `maxChars` caractères.

Fonction `strcpy()`
char `*strcpy(char *dest,` const char `*source);`
copie les caractères qui composent `source` dans `dest` en écrasant le contenu précédent de `dest.` Lors de la copie, le caractère nul est aussi copié. Le résultat retourné est un pointeur vers `dest.`

Fonction `strncpy()`
char `*strncpy(char *dest,` const char `*source, size_t maxChars);`
copie au plus `maxChars` caractères de `source` vers `dest.` Si la longueur de `source` est supérieure ou égale à `maxChars,` les `maxChars` premiers caractères sont copiés et le résultat ne se terminera pas par un caractère nul. Si la longueur de `source` est inférieure à `maxChars,` la chaîne `dest` comportera un caractère nul à la fin.

Fonction `strchr()`
char `*strchr(const` char `*chaine,` int `c);`
renvoie un pointeur vers la première occurrence du caractère `c` dans `chaine,` ou un pointeur nul (NULL) si `c` n'est pas contenu dans `chaine.`

Fonction `strrchr()`
char `*strrchr(const` char `*chaine,` int `c);`
retourne un pointeur vers la dernière occurrence de `c` dans `chaine.` Si `c` n'apparaît pas dans `chaine,` la valeur retournée est `NULL.`

Fonctions de manipulation de chaînes de caractères (suite)

Fonction `strstr()`

```
char *strstr(const char *chaine, const char *cible);
```

retourne un pointeur vers le début de la première apparition de `cible` dans `chaine`, ou `NULL` si `cible` n'apparaît pas dans `chaine`. Par exemple, l'appel `strstr("Le monde est bon.","on")` retourne un pointeur vers le caractère `'o'` de `"monde"`.

Exercices

Réalisez une implémentation des fonctions ci-dessous.

```
char *strcat(char *, const char *);
char *strncat(char *, const char *, size_t);
int strcmp(const char *, const char *);
int strncmp(const char *, const char *, size_t);
char *strcpy(char *, const char *);
char *strncpy(char *, const char *, size_t);
char *strchr(const char *, int);
char *strrchr(const char *, int);
char *strstr(const char *, const char *);
```

Fonctions de manipulation de blocs de mémoire

Il existe une série de fonctions permettant de manipuler des blocs de mémoire.
Elles se distinguent des fonctions `strxxxx()` par le fait qu'il faut spécifier la taille des blocs intervenants.
La déclaration de ces fonctions peut être soit dans `mem.h` ou dans `string.h`.

Fonction `memchr()`
void `*memchr(const` void `*source,` int `c, size_t nbOctets);`
retourne un pointeur vers la première occurrence du caractère `c` dans le bloc `source`, ou un pointeur nul (NULL) si `c` n'est pas contenu dans `source`. Si `nbOctets=0`, certaines librairies retournent `NULL`.

Fonction `memcmp()`
int `memcmp(const` void `*bloc1,` const void `*bloc2, size_t nbOctets);`

réalise la comparaison de deux blocs de mémoire. La comparaison se fait en supposant que les blocs sont des caractères non-signés (unsigned char), et le résultat suit la même convention que `strcmp()` : le résultat est négatif, 0 ou positif selon que `bloc1` est respectivement inférieur, égal ou supérieur à `bloc2`.

Fonction `memcpy()`
void `*memcpy(void *dest,` const void `*source, size_t nbOctets);`
copie les `nbOctets` premiers octets composant `source` dans `dest` en écrasant le contenu précédent de `dest` et retourne un pointeur vers `dest`. Si les blocs `dest` et `source` se chevauchent, le comportement de `memcpy()` est imprévisible.

Fonction `memmove()`
void `*memmove(void *dest,` const void `*source, size_t nbOctets);`
est identique à `memcpy()` mais les blocs peuvent se chevaucher.

Fonction `memset()`
void `*memset(void *dest,` int `c, size_t nbOctets);`
positionne tous les octets de `dest` à la valeur de `c` et retourne un pointeur vers `dest`.

Classes de données

En C, les fonctions et les variables ont deux attributs :
- leur type
- leur emplacement mémoire.

L'emplacement mémoire régit la durée de vie et la visibilité des variables.

Pour comprendre ce qui suit, il faut expliquer quelque peu l'utilisation de la mémoire d'un programme C. Le compilateur organise tout programme en au moins 3 segments :

- Le segment des instructions contient toutes les instructions du programme. Donc toutes les fonctions qui y sont définies.
- Le segment des données permanentes contient toutes les données qui existent pendant toute la durée du programme. C'est l'emplacement désigné pour les globales.
- Le segment des données temporaires contient les données dont l'existence varie au cours de l'exécution du programme. Cette portion de la mémoire est gérée comme une pile. Les variables y sont affectées à chaque fois que le programme entre dans un bloc d'instructions.

Ces trois segments sont offerts par tout compilateur C. Certains compilateurs incluent aussi un segment pour les données à lecture unique.

La notion de classe de données définit le segment de mémoire où résident les fonctions et les variables. Elle définit donc la durée de vie et en quelque sorte la visibilité de ces entités.

Classe automatique

En C, toute variable définie à l'intérieur d'un bloc appartient par défaut à la classe automatique. Ces variables se retrouvent dans le segment de données temporaires. Les paramètres des fonctions sont aussi affectés dans ce segment, car ils cessent d'exister lorsque la fonction se termine.

durée de vie: temporaire (égale à celle de la fonction où la donnée est définie)
locale à la fonction ou au blocvisibilité:

Exemple

```
void Exemple(void)
 {
       auto int i = 5;
       ...
 }
```

Le mot réservé `auto` indique qu'une variable est automatique, mais il est rare que l'on le rencontre.

L'initialisation d'une variable automatique, lors de sa définition, n'est en fait qu'une abréviation d'écriture.

Classe registre

Une variable automatique utilisée très souvent peut être définie avec la classe registre. Celle-ci indique au compilateur que l'usage prévu de cette variable sera intense. Dans ce cas, le compilateur peut décider d'affecter cette variable dans un registre plutôt que sur la pile.

durée de vie:	temporaire (égale à celle de la fonction où la donnée est définie)
visibilité:	locale à la fonction ou au bloc

Exemple

```
void Exemple(void)
 {
       register int i = 5;
       ...
 }
```

Cette classe ne convient qu'aux types élémentaires pouvant entrer dans un registre.

Le fait d'affecter une variable à la classe registre n'est qu'une indication au compilateur. On peut désigner plus de variables qu'il n'y a de registres dans la machine!

Comme une variable de classe registre ne réside pas en mémoire, il est illégal de demander l'adresse d'une telle variable par l'opérateur `&`.

Classe statique

Les variables de cette classe sont affectées dans le segment de données permanentes. Quand la fonction termine, ses variables statiques conservent leur contenu.

durée de vie:	permanent (égal à celle du programme)
visibilité:	locale à la fonction

Exemple

```
void Exemple(void)
{
        static int i = 5;
        ...
}
```

L'initialisation d'une variable statique se fait lors du chargement du programme. Ce qui signifie que le membre de droite de l'initialisation doit être une expression constante.

Les variables statiques qui ne sont pas initialisées prennent la valeur 0.

Pour initialiser des variables statiques composées, on place les valeurs des différents champs dans des expressions séparées par des virgules, et on entoure le tout par une paire d'accolades :

```
static struct date {
            int jour, mois, annee;
  } aujourdhui = {10, 3, 97};
static char message[8] = {'b','o','n','j','o','u','r'};
static char message[8] = "bonjour";
static char message[] = "bonjour";
```

Lorsque le nombre d'éléments d'un tableau est spécifié et que le nombre d'expressions d'initialisation est inférieur au nombre d'éléments, les derniers éléments sont initialisés à 0.

```
static int tableau[50] = {1};
```

Il n'est pas bon de définir de gros tableaux automatiques dans des fonctions récursives car leur création requiert beaucoup d'espace sur la pile. Il vaut mieux les définir comme statiques. Ils seront alors initialisés qu'une seule fois, lors du chargement. Cependant, cette façon de faire a le désavantage que ces tableaux doivent être en mémoire pendant toute la durée du programme.

Classe volatile

La plupart des compilateurs C optimisent les instructions qu'ils génèrent. Il existe cependant des situations particulières où ces optimisations ne sont pas souhaitables.

Exemple

```
void Exemple(void)
  {
      volatile int *clavier = 0177560;
      ...

       /* attendre que le bit 7 du registre du clavier soit à 0 */
       while (*clavier & 0200)
              ;
       ...

   }
```

Sans le mot réservé volatile, la condition de l'énoncé while pourrait être générée incorrectement par un compilateur optimisant. En effet, un seul accès à la mémoire est suffisant si nous optimisons ce code. Le résultat serait un programme avec une boucle infinie.

Le standard ANSI a introduit une nouvelle classe de données, nommée volatile, dont tous les accès à des variables de cette classe doivent résulter en un accès immédiat à la mémoire et les optimisations, les concernant, sont interdites au compilateur.

durée de vie:	temporaire
visibilité:	locale à la fonction mais peut être modifiée par le système d'exploitation

Une variable ne peut pas simultanément appartenir aux classes statique, registre et volatile.

Classe constante

La norme ANSI a ajouté au langage la classe de données const. Celle-ci indique qu'une constante identifiée (de n'importe quel type) possède une valeur qui ne peut être modifiée.

Toute tentative de modifier une variable appartenant à la classe constante donne une erreur.

Exemples

```
const double pi = 3.1416;
/* pointeur vers une constante entière */
const int *ptrInt;
/* pointeur constant vers un double */
double *const ptrDob = &pi;
/* pointeur constant vers une constante réelle */
const double *const ptrPi = &pi;
```

Classe globale publique

Une variable définie à l'extérieur de toute fonction est dite globale. Comme pour les variables statiques, les globales sont affectées au segment de données permanentes. Les globales peuvent être initialisées lors de leur définition.
Il existe 2 types de variables globales : les publiques et les privées.
Les variables globales publiques sont connues à l'extérieur du fichier dans lequel elles sont définies (la relation se fait durant l'édition des liens).

Lorsque nous parlons de variables globales publiques, il nous faut rappeler de la différence subtile entre la définition et la déclaration. La définition désigne l'endroit où le type d'une variable est défini et où l'espace de mémoire est réservé pour celle-ci.

La déclaration d'une variable ne fait que spécifier son type. Elle est nécessaire lorsque la variable est utilisée avant d'être définie. Il est interdit d'initialiser une variable lors de sa déclaration, puisque celle-ci ne comporte pas d'espace de mémoire.
Les lignes de prototype sont des déclarations de fonctions.

La déclaration d'une variable globale publique est une déclaration normale précédée du mot clé extern. De telles déclarations peuvent apparaître dans un bloc ou au niveau global (c.-à-d. à l'extérieur de toute fonction). Dans ce dernier cas, la déclaration est valide jusqu'à la fin du fichier.

Classe globale publique: exemple

```
/************ fichier 1 ************/
int i = 10, j = 20;/* définition de 2 globales publiques */


void main(void)
{
   extern int i;/* déclaration optionnelle car i est visible */
   extern int fi(int);/* déclaration d'une fonction */
   int fint(int);/* déclaration d'une fonction */



   if (i < 50)
       j = fi(i++);
   else
      i = fint(i + j++);


}


/************ fichier 2 ************/
/* extern int i, j; /* déclarations pour l'ensemble du fichier */


int fi(int j)
{
   extern int i;
                          /* déclaration de i pour la durée de fi() */
   ...
}
int fint(int k)
{
   extern int j;
   int i;
}
```

En résumé

durée de vie:	permanente
visibilité:	globale à toutes les fonctions définies dans le fichier et à celles qui l'importe

Classe globale privée

Une variable globale privée est obtenue en précédant la définition d'une variable globale du mot réservé static. Dans ce cas, la variable est inaccessible à l'extérieur du fichier source où elle est définie (c.-à-d. son identificateur n'est pas connu de l'éditeur de liens).

En résumé, le mot clé static met la variable (automatique ou globale) dans le segment de données permanentes.

durée de vie:	permanente
visibilité:	globale uniquement aux fonctions définies dans le fichier où la variable est définie

Les variables globales privées permettent d'implémenter le concept de module.

Classe globale privée: exemple

```
#define NBELEMENTS 100
static int pile[NBELEMENTS];
static int sommet;


static int PilePleine(void);            /* Fonction uniquement visible
                                           dans ce fichier*/


int PileVide(void)                      /* Retourne 1 si la pile est */
{                                       /* vide, et 0 autrement.*/
     ...
}



int Empile(int element)                 /* Retourne 1 si OK et 0 en      */
{                                       /* cas d'erreur                  */
     ...
}



int Desempile(void)                     /* retourne -1 si erreur         */
{
     ...
}



static int PilePleine(void)             /* retourne 1 si plein et 0      */
{                                       /* autrement                     */
}
```

Quelques remarques

Les données `pile` et `sommet` doivent être globales puisqu'elles sont partagées par les 4 fonctions. Toutefois elles ont un accès contrôlé.

Les fonctions sont nécessairement globales, mais rien nous empêche de limiter leur visibilité.

Préprocesseur

Le précompilateur est un interpreteur qui lit les programmes sources avant le compilateur afin d'y apporter certaines modifications. Il permet

- de substituer une chaîne de caractères par une autre;
- d'insérer le contenu d'un fichier dans le fichier courant;
- de définir des constantes symboliques.

Remarquons que d'autres précompilateurs peuvent être ajoutés à C, e.g. C++.

Toutes les directives du précompilateur débutent par le caractère `#` (dièse). Sur les compilateurs C pré-ANSI, ce caractère doit être écrit en première colonne du fichier source. La norme ANSI a enlevé cette contrainte, mais les seuls caractères qui le précèdent sur la ligne doivent être des espaces ou des tabulations.

Constantes symboliques

`#define` <constante symbolique> [<chaîne de remplacement>]

Cette directive remplace une chaîne de caractères par une autre dans la suite du fichier.

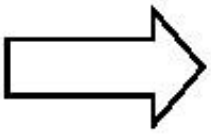
La chaîne de remplacement inclut tout le texte de la ligne qui suit le nom de la constante, jusqu'à la fin de la ligne. Il faut donc faire attention aux commentaires insérés sur ces lignes: ils doivent toujours se terminer sur cette même ligne. Il faut aussi faire attention aux commentaires introduits en AINSI C par les caractères `//`.

Constantes symboliques (suite)

Exemples

```
#define EOF             (-1)      /* fin de fichier */


#define MAX115
#define MAX2(MAX1 + 4)
                  int#define BOOL
#define TRUE1
#define FALSE0
void main(void)
{
  BOOL fini = TRUE;
  int x;
  while (fini) {
     x = MAX2;
     fini = FALSE;
  }
}
```



```
void main(void)
{
   int fini = 1;
   int x;
   while (fini) {
      x = (15 + 4);
      fini = 0;
   }
}
```

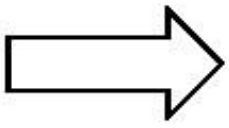
Remarquons qu'aucun calcul n'a été effectué: le précompilateur ne fait que la substitution, et il ne se substitue pas au compilateur! Voici un autre exemple qui illustre ce fait:

```
#define SOMME 200 + 4
...
int total, pourCent;
...
pourCent = SOMME * 2 / 100;
```

Constantes symboliques (suite)

```
#define  EQ ==
#define  do /* rien */
#define  begin {
#define  end }
```

```
while (i EQ 1) do
  begin
    ...
   end
```



```
while (i == 1)
   {
     ...
   }
```

Remarquons que l'énoncé do `{...}` while `(condition);` se compiler sans erreur!

Il faut éviter d'utiliser des constantes symboliques pour convertir le code C en un pseudo-code quelconque. Ceux qui auront à le lire et à le modifier seront deroutés.

Inclusion de fichiers

Lorsque les constantes symboliques se retrouvent dans plusieurs fichiers, il est préférable de demander au précompilateur d'inclure un fichier où ces constantes sont définies.

`#include <` <ficher système> `>`

`#include "` `<ficher` usager> `"`

La norme ANSI permet jusqu'à 8 niveaux d'imbrications de fichiers différents.

La notation par des guillemets indique au précompilateur que le fichier indiqué se trouve dans le répertoire où la compilation est lancée. Il est aussi possible d'indiquer le parcours pour atteindre ce fichier :

```
#include "../mesdefs.h"
```

Compilation conditionnelle

Elle permet d'inclure des portions de code différentes lors de la compilation, selon la valeur d'une constante symbolique du préprocesseur.
Il est important de noter que, lors d'une compilation conditionnelle, la partie rejetée ne figure pas dans le code objet résultant.
Cette construction sert à rendre les programmes plus facilement portables, et surtout à aider la mise au point de ceux-ci.

Les directives

`#if <expression>`

- permet de vérifier si la valeur d'une expression est vraie (différente de zéro).
- Les identificateurs de l'expression doivent être des constantes symboliques.

`#endif`

- marque la fin d'une compilation conditionnelle.

`#else` et `#elif <expression>`

- permettent d'introduire une clause alternative.

Exemple

```
#define IDENT 1
 #if IDENT == 1
      printf("C'est un.\n");
 #else
      printf("Ce n'est pas un.\n");
 #endif
```

Compilation conditionnelle (suite)

`#ifdef` <identificateur>

- permet de vérifier si l'identificateur définit une constante symbolique (c.-à-d. a fait l'objet d'un `#define)`.

`#ifndef` <identificateur>

- permet de vérifier si l'identificateur n'est pas défini.

La norme ANSI comporte une nouvelle notation plus naturelle pour vérifier l'existence d'une définition pour une constante symbolique. Il s'agit de la fonction

`defined(` <identificateur> `)`

qui retourne un booléen. Cette notation trouve son intérêt lorsque plusieurs constantes se retrouvent dans une expression booléenne :

```
#ifdef IDENT1
#ifdef IDENT2
...
#endif
#endif
```

```
#if defined(IDENT1) && defined(IDENT2)
...
#endif
```

`#undef` <identificateur>

- permet d'annuler la définition d'une constante symbolique.

Macros

Outre les constantes symboliques, l'énoncé `#define` permet de définir des macros. Il s'agit de la substitution d'une constante par une chaîne de caractères comportant des arguments.

`#define` <identificateur> `(`<arg> [, <arg>]* `)` <chaîne de remplacement>

Exemples

```
#define max(a,b) (a > b ? a : b)
#define CARRE(a) ((a) * (a))
```

Le fait que le précompilateur n'effectue que des remplacements de chaînes de caractères a des répercussions importantes. Soit le fragment de code

```
#define max(a,b)(a > b ? a : b)
 #define abs(a)a > 0 ? a : -a
 int x, y, z;
 ...
 z = max(abs(x+10),abs(y-5));
```

L'expansion des macros donne

```
z = (abs(x+10) > abs(y-5) ? abs(x+10) : abs(y-5));
z = (x+10 > 0 ? x+10 : - x+10 > y-5 > 0 ? y-5 : - y-5 ?
     x+10 > 0 ? x+10 : - x+10 y-5 > 0 ? y-5 : - y-5);
```

Règles d'usage des macros

1. Ne jamais mettre d'espace entre le nom de la macro et le début de la liste de ses arguments :

```
#define CARRE (a) a * a
  CARRE(2)
```

sera développé en

```
(a)a * a(2)
```

2. Ne jamais mettre le marqueur de fin d'instruction (;) dans la définition du corps :

```
#define CARRE(a) a * a;
  int a = CARRE(2) + CARRE(2);
```

sera étendue en

```
int a = 2 * 2; + 2 * 2;; /* pas d'erreur de compilation */
```

3. Entourer tous les arguments et toutes les instructions où apparaissent les arguments dans le corps de la macro par des parenthèses :

```
#define CARRE(a) a * a
 int z = CARRE(2+4);
```

sera étendue en

```
int z = 2+4 * 2+4;
```

```
#define CARRE(a) (a) * (a)
int z = 8 / CARRE(4);
```

sera étendue en

```
int z = 8 / 4 * 4;
```

On constate que seule `#define CARRE(a) ((a)*(a))` donne toujours le même résultat.

4. Ne jamais utiliser les opérateurs d'incrément et de décrément lors de l'appel ni dans le corps de la macro.

Utilisation de macros

1. En développement

```
#ifdef DEBUG
  #define ASSERT(x)                                                    \
        {if ( ! (x) ) {                                                \
            printf("ligne %d: %s\n",__LINE__,"x");                     \
            exit(1);                                                   \
          }                                                            \
        }
#else
  #define ASSERT(x)
#endif
```

L'objectif de cette macro est de vérifier certaines conditions, dites invariantes, avant ou après une opération. Si cette condition n'est pas réalisée, un message est affiché, et le programme se termine. Tous les compilateurs disposent d´un certain nombre de constantes prédéfinies, telles que la ligne du code source en cours de compilation, (__LINE__). Le caractère de continuation (\) indique que le corps de la macro se poursuit sur la prochaine ligne. Pour éviter de rééditer le fichier source et d'enlever ou remettre ces lignes d'assertion, il suffit de définir ou non la constante symbolique `DEBUG`.

2. En ajustage

Certaines fonctions critiques peuvent bénéficier d'un gain de performance si l'appel de la fonction est remplacé par le corps de cette fonction.