Patrick TRAU
http://pat.fr.st

Programmer en C

partie 1 : le langage

Université Louis Pasteur
Institut Professionnel des
Sciences et Technologies
15 rue du Maréchal Lefèbvre
67100 STRASBOURG (F)

Table des matières

Table des matières

1 Langage C

Le Langage C

Aujourd'hui, l'informatique est présente dans tous les domaines de la vie courante, mais à des degrés différents. Il y a pour cela trois grandes raisons :

− les gains (en temps, argent, qualité) que l'informatique peut apporter,

− le prix abordable des matériels,

− la disponibilité de logiciels dans tous les domaines.

Deux domaines sont pleinement exploités :

− les logiciels généraux, vendus en grande série, et donc relativement bon marché,

− les logiciels spécifiques, d'un coût total important et donc limités à des sujets très pointus, pour de très grosses industries.

Le domaine intermédiaire, qui peut encore se développer, concerne les programmes spécifiques, pour des applications de moindre importance. Pour cela, il est nécessaire de disposer de langages de programmation. Les tableurs et bases de données par exemple disposent désormais de véritables langages de programmation (souvent orientés objets) qui vont plus loin que les précédents langages de macro−commandes. Pour les autres cas, le C est souvent le meilleur choix. En effet, c'est un langage structuré, avec toutes les possibilités des autres langages structurés. Mais il permet également (avec son extension C++) de gérer des objets. A l'inverse, il permet également une programmation proche du langage machine, ce qui est nécessaire pour accéder aux interfaces entre l'ordinateur et son extérieur. Mais son principal avantage est que ces trois types de programmation peuvent être combinés dans un même programme, tout en restant portable sur tous les ordinateurs existants. Le langage C a néanmoins deux inconvénients majeurs, c'est d'être un peu plus complexe d'utilisation (mais uniquement du fait de ses nombreuses possibilités), et d'être séquentiel, ce qui ne lui permettra pas d'être le langage optimal pour les machines massivement parallèles (mais aujourd'hui il n'existe pas encore de langage universel pour ce type de machines qui puisse combiner efficacement des calculs procéduraux et du déclaratif).

Ceci est la première partie du livre "Programmation en C, langage et algorithmes", qui est composé de trois parties.

* La première définit le langage C, avec de nombreux exemples directement dans le texte, et certains supplémentaires en annexe.

* La seconde partie traite des structures de données et des algorithmes. Elle n'est pas spécifique, seuls les exemples sont en C, le texte et les algorithmes restent utilisables dans tout langage séquentiel. Les fonctionnalités du C non disponibles dans d'autres langages sont analysées, le méthodes utilisées pour parvenir au même résultat sont précisées (par exemple, gestion d'un liste dynamique à l'aide d'un tableau).

* La troisième partie quand à elle traite des algorithmes d'infographie. Elle est importante du fait de la nécessité du graphisme dans les programmes. Mais elle détaille également, pour ces cas pratiques, les méthodes et moyens utilisés pour optimiser des algorithmes. Une bibliothèque graphique est fournie en annexe.

* Les parties que j'ai prévues, mais pas encore rédigées : utilisation des objets C++, programmation graphique évennementielle (Microsoft Windows, X−Windows)

2 Introduction (première partie)

2.1 Organisation de l'ordinateur

− Multipostes : plusieurs consoles sur un même ordinateur (CPU puissant, tout est partageable)

− Réseau : plusieurs CPU et MC non partageable (sauf réseau de multipostes), MdM et périphériques partageables ou locaux.

2.2 Langages de programmation

Un ordinateur est une machine bête, ne sachant qu'obéir, et à très peu de choses :

* addition, soustraction, multiplication en binaire, uniquement sur des entiers,

* sortir un résultat ou lire une valeur binaire (dans une mémoire par exemple),

* comparer des nombres.

Sa puissance vient du fait qu'il peut être PROGRAMME, c'est à dire que l'on peut lui donner, à l'avance, la séquence (la suite ordonnée) des ordres à effectuer l'un après l'autre. Le grand avantage de l'ordinateur est sa rapidité. Par contre, c'est le programmeur qui doit TOUT faire. L'ordinateur ne comprenant que des ordres codés en binaire (le langage machine), des langages dits "évolués" ont été mis au point pour faciliter la programmation, au début des années 60, en particulier FORTRAN (FORmula TRANslator) pour le calcul scientifique et COBOL pour les applications de gestion. Puis, pour des besoins pédagogiques principalement, ont été créés le BASIC, pour une approche simple de la programmation, et PASCAL au début des années 70. Ce dernier (comme le C) favorise une approche méthodique et disciplinée (on dit "structurée"). Le C a été développé conjointement au système d'exploitation UNIX, dans les Laboratoires BELL, par Brian W Kernigham et Dennis M Ritchie, qui ont défini en 78, dans "The C Language", les règles de base de ce langage. Le but principal était de combiner une approche structurée (et donc une programmation facile) avec des possibilités proches de celles de l'assembleur (donc une efficacité maximale en exécution, quitte à passer plus de temps de programmation), tout en restant standard (c'est à dire pouvoir être implanté sur n'importe quelle machine). Puis ce langage a été normalisé en 88 (norme ANSI), cette norme apportant un nombre non négligeable de modifications au langage.

Le C est un langage compilé, c'est à dire qu'il faut :

* entrer un texte dans l'ordinateur (à l'aide d'un programme appelé EDITEUR),

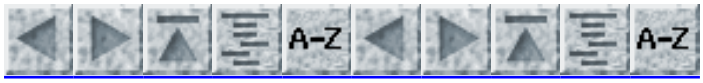
* le traduire en langage machine (c'est à dire en codes binaires compréhensibles par l'ordinateur) : c'est la compilation et, si plusieurs modules ont été compilés séparément, l'édition de liens (LINK ou BIND),

* l'exécuter.

Contrairement à un basic interprété, l'exécution sera beaucoup plus rapide puisqu'il n'y a plus de traduction à effectuer, mais la phase de mise au point sera plus complexe.

Bien que le langage soit normalisé, un certain nombre de points dépendent de la machine et du compilateur utilisé (par exemple comment appeler le compilateur). Ces indications ne seront pas données ici. Si vous avez

le choix, je vous conseille TURBO C, le plus pratique d'emploi (en particulier parce qu'il possède son propre éditeur de texte). Il permet une mise au point aussi simple que si le langage était interprété



3 Connaissances de base

regardons ce petit programme :

```
#include <stdio.h>
#define TVA 18.6
void main(void)
 {
  float HT,TTC;
  puts ("veuillez entrer le prix H.T.");
  scanf("%f",&HT);
  TTC=HT*(1+(TVA/100));
  printf("prix T.T.C. %f\n",TTC);
 }
```

On trouve dans ce programme :

* des directives du pré processeur (commençant par #)

#include : inclure le fichier définissant (on préfère dire déclarant) les fonctions standard d'entrées/sorties (en anglais STanDard In/Out), qui feront le lien entre le programme et la console (clavier/écran). Dans cet exemple il s'agit de puts, scanf et printf.

#define : définit une constante. A chaque fois que le compilateur rencontrera, dans sa traduction de la suite du fichier en langage machine, le mot TVA, ces trois lettres seront remplacées par 18.6. Ces transformation sont faites dans une première passe (appelée pré compilation), où l'on ne fait que du "traitement de texte", c'est à dire des remplacements d'un texte par un autre sans chercher à en comprendre la signification.

* une entête de fonction. Dans ce cas on ne possède qu'une seule fonction, la fonction principale (main function). Cette ligne est obligatoire en C, elle définit le "point d'entrée" du programme, c'est à dire l'endroit où débutera l'exécution du programme.

* un "bloc d'instructions", délimité par des accolades {}, et comportant :

***** des déclarations de variables, sous la forme : type listevariables;

Une variable est un case mémoire de l'ordinateur, que l'on se réserve pour notre programme. On définit le nom que l'on choisit pour chaque variable, ainsi que son type, ici float, c'est à dire réel (type dit à virgule flottante, d'où ce nom). Les trois types scalaires de base du C sont l'entier (int), le réel (float) et le caractère (char). On ne peut jamais utiliser de variable sans l'avoir déclarée auparavant. Une faute de frappe devrait donc être facilement détectée, à condition d'avoir choisi des noms de variables suffisamment différents (et de plus d'une lettre).

***** des instructions, toutes terminées par un ;. Une instruction est un ordre élémentaire que l'on donne à la machine, qui manipulera les données (variables) du programme, ici soit par appel de fonctions (puts, scanf, printf) soit par affectation (=).

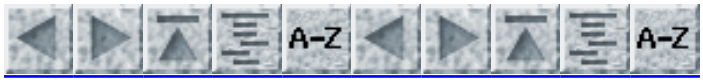
Détaillons les 4 instructions de notre programme :

puts affiche à l'écran le texte qu'on lui donne (entre parenthèses, comme tout ce que l'on donne à une fonction, et entre guillemets, comme toute constante texte en C).

scanf attend que l'on entre une valeur au clavier, puis la met dans la mémoire (on préfère dire variable) HT, sous format réel (%f).

une **affectation** : on commence par diviser TVA par 100 (à cause des parenthèses), puis on y ajoute 1, puis on le multiplie par le contenu de la variable HT. Le résultat de ce calcul est stocké (affecté) dans la variable cible TTC. Une affectation se fait toujours dans le même sens : on détermine (évalue) tout d'abord la valeur à droite du signe =, en faisant tous les calculs nécessaires, puis elle est transférée dans la mémoire dont le nom est indiqué à gauche du =. On peut donc placer une expression complexe à droite du =, mais à sa gauche seul un nom de variable est possible, aucune opération.

printf affichera enfin le résultat stocké dans TTC.



4 Fonctions d'entrées/sorties les plus utilisées

Le langage C se veut totalement indépendant du matériel sur lequel il est implanté. Les entrées sorties, bien que nécessaires à tout programme (il faut lui donner les données de départ et connaître les résultats), ne font donc pas partie intégrante du langage. On a simplement prévu des bibliothèques de fonctions de base, qui sont néanmoins standardisées, c'est à dire que sur chaque compilateur on dispose de ces bibliothèques, contenant les même fonctions (du moins du même nom et faisant apparemment la même chose, mais programmées différemment en fonction du matériel). Ces bibliothèques ont été définies par la norme ANSI, on peut donc les utiliser tout en restant portable. Nous détaillerons ici deux bibliothèques : CONIO.H et STDIO.H.

Dans CONIO.H on trouve les fonctions de base de gestion de la console :

putch(char) : affiche sur l'écran (ou du moins stdout) le caractère fourni en argument (entre parenthèses). stdout est l'écran, ou un fichier si on a redirigé l'écran (en rajoutant >nomfichier derrière l'appel du programme, sous DOS ou UNIX). Si besoin est, cette fonction rend le caractère affiché ou EOF en cas d'erreur.

getch(void) : attend le prochain appui sur le clavier, et rend le caractère qui a été saisi. L'appui sur une touche se fait sans écho, c'est à dire que rien n'est affiché à l'écran. En cas de redirection du clavier, on prend le prochain caractère dans le fichier d'entrée.

getche(void) : idem getch mais avec écho. On pourrait réécrire cette fonction en fonction des deux précédentes :

```
char getche(void);
 {
  char caractere;
  caractere=getch();
  putch(caractere);
  return(caractere);
 }
```

ou même `{return(putch(getch))}`

Dans la pratique, ce n'est pas ainsi que getche est réellement défini, mais en assembleur pour un résultat plus rapide.

Dans STDIO.H, on trouve des fonctions plus évoluées, pouvant traiter plusieurs caractères à la suite (par les fonctions de conio.h), et les transformer pour en faire une chaîne de caractères ou une valeur numérique, entière ou réelle par exemple. Les entrées sont dites "bufférisées", c'est à dire que le texte n'est pas transmis, et peut donc encore être modifié, avant le retour chariot.

puts(chaîne) affiche, sur stdout, la chaîne de caractères puis positionne le curseur en début de ligne suivante. puts retourne EOF en cas d'erreur.

gets(chaîne) lecture d'une chaîne sur stdin. Tous les caractères peuvent être entrés, y compris les blancs. La saisie est terminée par un retour chariot. Gets retourne un pointeur sur le premier caractère entré (donc égal à son paramètre d'entrée, ou le pointeur NULL en cas d'erreur (fin du fichier stdin par exemple).

printf(format,listevaleurs) affiche la liste de valeurs (variables ou expressions) dans le format choisi. Le format est une chaîne de caractères entre guillemets (doubles quote "), dans laquelle se trouve un texte qui sera écrit tel quel, des spécifications de format (débutant par %) qui seront remplacées par la valeur effective des variables, dans l'ordre donné dans listevaleurs, et de caractères spéciaux (\). Printf retourne le nombre de

caractères écrits, ou EOF en cas de problème.

Une spécification de format est de la forme :

% [flag][largeur][.précision][modificateur]type (entre [] facultatifs)

Le **flag** peut valoir : − (cadrage à gauche, rajout de blancs si nécessaire à droite), + (impression du signe, même pour les positifs), blanc (impression d'un blanc devant un nombre positif, à la place du signe), 0 (la justification d'un nombre se ferra par rajout de 0 à gauche au lieu de blancs). D'autres Flags sont possibles mais moins utiles, pour plus de détails voir l'aide en ligne de Turbo C.

La **largeur** est le nombre minimal de caractères à écrire (des blancs sont rajoutés si nécessaire). Si le texte à écrire est plus long, il est néanmoins écrit en totalité. En donnant le signe * comme largeur, le prochain argument de la liste de valeurs donnera la largeur (ex printf("%*f",largeur,réel)).

La **précision** définit, pour les réels, le nombre de chiffres après la virgule (doit être inférieur à la largeur). Dans la cas d'entiers, indique le nombre minimal de chiffes désiré (ajout de 0 sinon), alors que pour une chaîne (%s), elle indique la longueur maximale imprimée (tronqué si trop long). La précision peut, comme la largeur, être variable en donnant .*

Le **modificateur** peut être : h (short, pour les entiers), l (long pour les entiers, double pour les réels), L (long double pour les réels).

Le **type** est : c (char), s (chaîne de caractères, jusqu'au \0), d (int), u (entier non signé), x ou X (entier affiché en hexadécimal), o (entier affiché en octal), f (réel en virgule fixe), e ou E (réel en notation exponentielle), g ou G (réel en f si possible, e sinon), p (pointeur), % (pour afficher le signe %).

Les caractères spéciaux utilisables dans le format sont \t (tabulation), \n (retour à la ligne), \\ (signe \), \nb tout code ASCII, en décimal, hexa ou octal (\32=\040=\0x20=' ').

scanf(format,listeadresse) lecture au clavier de valeurs, dans le format spécifié. Les arguments sont des pointeurs sur les variables résultats (dans le cas de variables scalaires, les précéder par l'opérateur &). Scanf retourne le nombre de valeurs effectivement lues et mémorisées (pas les %*).

Le format peut contenir (entre ") : du texte (il devra être tapé exactement ainsi par l'utilisateur, et ne sera pas stocké), des séparateurs blancs (l'utilisateur devra taper un ou plusieurs blancs, tabulations, retours à la ligne), et des spécifications de format, sous la forme %[*][largeur][modificateur] type.

* signifie que la valeur sera lue mais ne sera pas stockée (ex scanf("%d%*c%d",&i,&j) : lecture de deux entiers séparés par n'importe quel caractère)

la largeur est la largeur maximale lue, scanf s'arrête avant s'il trouve un séparateur (blanc, tab, CR).

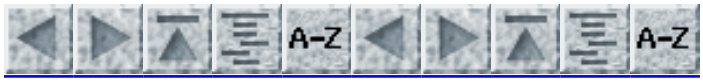
les modificateurs et types sont les mêmes que pour printf (excepté d,o,x : pour entiers short, D,O,X pour long). Dans le cas des chaînes (%s), le blanc est également un séparateur. On ne pourra donc pas entrer une chaîne avec des blancs par scanf, il faut utiliser gets.

Scanf lit dans stdin en considérant les retours chariot (CR) comme des blancs. on peut donc séparer par des CR plusieurs valeurs demandées dans le même scanf. Mais de même, si scanf a pu lire tous ses arguments sans arriver à la fin de la ligne, la suite servira au prochain scanf. Utilisez gets(bidon) avant (pour être sur de commencer sur une nouvelle ligne) ou après scanf (pour ignorer la fin de la ligne) si nécessaire.

getchar(void) fonctionne comme getche, mais utilise le même tampon que scanf.

On dispose aussi de **sprintf(chaîne, format, listevaleurs)** qui permet d'écrire dans une chaîne plutôt que sur

l'écran, et donc faire des conversions numériques −>ASCII; et de **sscanf(chaîne, format,l isteadresse)** qui lit dans une chaîne plutôt qu'au clavier. On possède encore d'autres fonctions dans STDIO, en particulier pour gérer les fichiers.



5 La syntaxe du C

5.1 Second exemple, définitions

Considérons le programme ci−dessous :

```
#include <stdio.h>
void affiche_calcul(float,float); /* prototype */
float produit(float,float);
int varglob;

void main(void)
 {
  float a,b; /* déclaration locale */
  varglob=0;
  puts("veuillez entrer 2 valeurs");
  scanf("%f %f",&a,&b);
  affiche_calcul(a,b);
  printf("nombre d'appels à produit : %d\n",varglob);
 }

float produit(float r, float s)
 {
  varglob++;
  return(r*s);
 }

void affiche_calcul(float x,float y)
 {
  float varloc;
  varloc=produit(x,y);
  varloc=produit(varloc,varloc);
  printf("le carré du produit est %f\n",varloc);
 }
```

Un programme C est composé :

* de directivess du pré−processeur, commençant par #, terminées par un retour à la ligne (pas de ;)

* de déclarations globales (terminées par un ;)

* d'une suite de fonctions, écrites les unes après les autres (sans imbrication comme on le ferait en Pascal)

Les fonctions sont écrites sous la forme : entête { corps }

L'entête est de la forme : type_résultat nom (arguments) . Le type_résultat n'était obligatoire (avant la norme

ANSI) que s'il était différent de int (entier). Il doit désormais être void (rien) si la fonction ne renvoie rien (dans un autre langage on l'aurait alors appelé sous−programme, procédure, ou sous−routine). Les arguments, s'ils existent, sont passés par valeur. Si la fonction ne nécessite aucun argument, il faut indiquer (void) d'après la norme ANSI, ou du moins ().

Le corps est composé de déclarations de variables locales, et d'instructions, toutes terminées par un ;

5.2 Variables / identificateurs / adresse / pointeurs

On appelle variable une mémoire de l'ordinateur (ou plusieurs), à laquelle on a donné un nom, ainsi qu'un type (on a précisé ce qu'on mettra dans cette variable (entier, réel, caractère,...), pour que le compilateur puisse lui réserver la quantité de mémoire nécessaire. Dans cette variable, on pourra y stocker une valeur, et la modifier au cours du programme.

Exemple : int a; a est une variable entière, le compilateur va lui réserver en mémoire la place nécessaire à un entier (2 octets en Turbo C). Le nom de cette variable est choisi par le programmeur. On préfère utiliser le terme identificateur plutôt que nom, car il permet d'identifier tout objet que l'on voudra utiliser (pas seulement les variables). Les identificateurs doivent suivre quelques règles de base : il peut être formé de lettres (A à Z), de chiffres et du caractère _ (souligné). Le premier caractère doit être une lettre (ou _ mais il vaut mieux le réserver au compilateur). Par exemple valeur1 ou prem_valeur sont possibles, mais pas 1ere_valeur. En C, les minuscules sont différentes des majuscules (SURFace et surFACE désignent deux objets différents). Le blanc est donc interdit dans un identificateur (utilisez _). Les lettres accentuées sont également interdites. La plupart des compilateurs acceptent n'importe quelle longueur d'identificateurs (tout en restant sur la même ligne) mais seuls les 32 premiers caractères sont significatifs.

On considère comme blanc : soit un blanc (espace), soit un retour à la ligne, soit une tabulation, soit un commentaire, soit plusieurs de ceux−ci. Les commentaires sont une portion de texte commençant par /* et finissant par le premier */ rencontré. les commentaires ne peuvent donc pas être imbriqués. mais un commentaire peut comporter n'importe quel autre texte, y compris sur plusieurs lignes.

Un identificateur se termine soit par un blanc, soit par un signe non autorisé dans les identificateurs (parenthèse, opérateur, ; ...). Le blanc est alors autorisé mais non obligatoire.

L'endroit où le compilateur a choisi de mettre la variable est appelé adresse de la variable (c'est en général un nombre, chaque mémoire d'un ordinateur étant numérotée de 0 à ?). Cette adresse ne nous intéresse que rarement de manière explicite, mais souvent de manière indirecte. Par exemple, dans un tableau, composé d'éléments consécutifs en mémoire, en connaissant son adresse (son début), on retrouve facilement l'adresse des différentes composantes par une simple addition. On appelle pointeur une variable dans laquelle on place (mémorise) une adresse de variable (où elle est) plutôt qu'une valeur (ce qu'elle vaut).

Les types de variables scalaires simples que l'on utilise le plus couramment sont le char (un caractère), l'int (entier) et le float (réel). Le char est en fait un cas particulier des int, chaque caractère étant représenté par son numéro de code ASCII.

5.3 Expressions / opérateurs

Une expression est un calcul qui donne une valeur résultat (exemple : 8+5). Une expression comporte des variables, des appels de fonction et des constantes combinés entre eux par des opérateurs (ex : MaVariable*sin(VarAngle*PI/180)).

Une expression de base peut donc être un appel à une fonction (exemple sin(3.1416). Une fonction est un bout de programme (que vous avez écrit ou faisant partie d'une bibliothèque) auquel on "donne" des valeurs (arguments), entre parenthèses et séparés par des virgules. La fonction fait un calcul sur ces arguments pour "retourner" un résultat. Ce résultat pourra servir, si nécessaire, dans une autre expression, voire comme

argument d'une fonction exemple atan(tan(x)). Les arguments donnés à l'appel de la fonction (dits paramètres réels ou effectifs) sont recopiés dans le même ordre dans des copies (paramètres formels), qui elles ne pourront que modifier les copies (et pas les paramètres réels). Dans le cas de fonctions devant modifier une variable, il faut fournir en argument l'adresse (par l'opérateur &, voir plus bas), comme par exemple pour scanf.

Pour former une expression, les opérateurs possibles sont assez nombreux, nous allons les détailler suivant les types de variables qu'ils gèrent.

5.3.1 Arithmétiques

Ces opérateurs s'appliquent à des valeurs entières ou réelles.

5.3.1.1 unaires

Ce sont les opérateurs à un seul argument : − et + (ce dernier a été rajouté par la norme ANSI). Le résultat est du même type que l'argument.

5.3.1.2 deuxaires

Le terme "**deuxaire**" n'est pas standard, je l'utilise parce que binaire est pour moi associé à la base 2.

Ces opérateurs nécessitent deux arguments, placés de part et d'autre de l'opérateur. Ce sont **+** (addition), **−** (soustraction), ***** (produit), / (division), **%** (reste de la division). % nécessite obligatoirement deux arguments entiers, les autres utilisent soit des entiers, soit des réels. Les opérandes doivent être du même type, le résultat sera toujours du type des opérandes. Lorsque les deux opérandes sont de type différent (mais numérique évidement), le compilateur prévoit une conversion implicite (vous ne l'avez pas demandée mais il la fait néanmoins) suivant l'ordre : { char −> int −> long −> float −> double } et { signed −> unsigned }. On remarque qu'il considère les char comme des entiers, les opérations sont en fait faites sur les numéros de code (ASCII). Les calculs arithmétiques sont faits uniquement soit en long soit en double, pour éviter des dépassements de capacité.

exemples :

```
int a=1,b=2,c=32000;
float x=1,y=2;
a=(c*2)/1000; /* que des int, le résultat est 64, bien que l'on soit
passé par un résultat intermédiaire (64000) qui dépassait la capacité
des entiers (mais pas celle des long) */
b=7/b; /* signe = donc en premier calcul de l'argument à droite : 7
(entier) / 2 (entier) donne 3 (entier, reste 1, que l'on obtient par
5%2). donc b=3 */
x=7/b; /* 7 et b entiers => passage en réel inutile, calcul de 7/3 donne
2 (entier, reste 1) puis opérateur = (transformation du 2 en 2.0 puis
transfert dans X qui vaut donc 2.0) */
x=7/y; /* un int et un float autour de / : transformation implicite de 7
en réel (7.0), division des deux réel (3.5), puis transfert dans x */
x=((float)(a+1))/b; /* calcul (entier) de a+1, puis transformation
explicite en float, et donc implicite de b en float, division 65.0/3.0
−> 21.666... */
```

5.3.2 Relationnels

5.3.2.1 comparaisons

Ces opérateurs sont deuxaires : = = (égalité), != (différent), <, >, <=, >=. Des deux côtés du signe opératoire, il faut deux opérandes de même type (sinon, transformation implicite) mais numérique (les caractères sont classés suivant leur numéro de code ASCII). Le résultat de l'opération est 0 si faux, 1 si vrai (le résultat est de type int). Exemple : (5<7)+3*((1+1)= =2) donne 4. Attention, le compilateur ne vous prévient pas si vous avez mis = au lieu de = = (= est aussi un opérateur, voir plus loin), mais le résultat sera différent de celui prévu.

5.3.2.2 logique booléenne

Le résultat est toujours 0 (faux) ou 1 (vrai), les opérandes devant être de type entier (si char conversion implicite), 0 symbolisant faux, toute autre valeur étant considérée vraie.

Opérateur unaire : ! (non). !arg vaut 1 si arg vaut 0, et 0 sinon.

Opérateurs deuxaires : && (ET, vaut 1 si les 2 opérandes sont non nuls, 0 sinon) et || (OU, vaut 0 si les deux opérandes sont nuls, 1 sinon). Le deuxième opérande n'est évalué que si le premier n'a pas suffi pour conclure au résultat (ex (a= =0)&&(x++<0) incrémente x si a est nul, le laisse intact sinon).

5.3.2.3 binaires

Ces opérateurs ne fonctionnent qu'avec des entiers. Ils effectuent des opérations binaires bit à bit. On peut utiliser ~ (complément, unaire), & (et), | (ou inclusif), ^ (ou exclusif), >> (décalage à droite, le 2ème opérande est le nombre de décalages), << (décalage à gauche). Contrairement aux opérateurs relationnels, les résultats ne se limitent pas à 0 et 1.

exemples : 7&12 donne 4 (car 0111&1100 donne 0100); ~0 donne −1 (tous les bits à 1, y compris celui de signe); 8>>2 donne 32.

5.3.3 Affectation

5.3.3.1 affectation simple =

En C, l'affectation (signe =) est une opération comme une autre. Elle nécessite deux opérantes, un à droite, appelé Rvalue, qui doit être une expression donnant un résultat d'un type donné, et un à gauche (Lvalue) qui doit désigner l'endroit en mémoire où l'on veut stocker la Rvalue. Les deux opérandes doivent être de même type, dans le cas d'opérandes numériques si ce n'est pas le cas le compilateur effectuera une conversion implicite (la Lvalue doit être de type "plus fort" que la Rvalue). L'opération d'affectation rend une valeur, celle qui a été transférée, et peut donc servir de Rvalue.

Exemples : a=5 (met la valeur 5 dans la variable a. Si a est float, il y a conversion implicite en float); b=(a*5)/2 (calcule d'abord la Rvalue, puis met le résultat dans b); a=5+(b=2) (Le compilateur lit l'expression de gauche à droite. la première affectation nécessite le calcul d'une Rvalue : 5+(b=2). Celle ci comporte une addition, dont il évalue le premier opérande (5) puis le second (b=2). Il met donc 2 dans b, le résultat de l'opération est 2, qui sera donc ajouté à 5 pour être mis dans a. A vaut donc 7 et b, 2. Le résultat de l'expression est 7 (si l'on veut s'en servir).

Remarque : il ne faut pas confondre = et = =. Le compilateur ne peut pas remarquer une erreur (contrairement au Pascal ou Fortran) car les deux sont possibles. Exemple : if (a=0) est toujours faux car quelle que soit la valeur initiale de a, on l'écrase par la valeur 0, le résultat de l'opération vaut 0 et est donc interprété par IF comme faux.

5.3.3.2 incrémentation / décrémentation

++a : ajoute 1 à la variable a. Le résultat de l'expression est la valeur finale de a (c'est à dire après incrémentation). On l'appelle incrémentation préfixée.

a++ : ajoute 1 à la variable a. Le résultat de l'expression est la valeur initiale de a (c'est à dire avant incrémentation). C'est l'incrémentation postfixée.

de même, la décrémentation −−a et a−− soustrait 1 à a.

exemple : j=++i est équivalent à j=(i=i+1)

5.3.3.3 affectation élargie

+= , −= , *= , /= , %= , <<= , >>= , &= , ^= , |=

a+=5 est équivalent à a=(a+5). Il faut encore ici une Rvalue à droite et une Lvalue à gauche.

5.3.4 Opérateurs d'adresses

Ces opérateurs sont utilisées avec des pointeurs. On utilise

- &variable : donne l'adresse d'une variable
- *pointeur : réfère à la variable pointée (opérateur d'indirection)
- . : champ d'une structure
- −> : champ pointé

exemple : supposons déclarer : int i1=1,i2=2; int *p1,*p2; i1 et i2 sont deux mémoires contenant un entier, alors que p1 et p2 sont des pointeurs, puisqu'ils contiennent une adresse d'entier. p1=&i1; met dans p1 l'adresse de i1. p2=p1; met la même adresse (celle de i1) dans p2. printf("%d\n",*p1) affiche ce qui est désigné (pointé) par p1 donc i1 donc 1. p2=&i2;*p2=*p1; à l'adresse pointée par p2 mettre ce qui est pointé par p1, donc copier la valeur de i1 dans i2. `printf("%d\n",i2)` affiche donc 1.

5.3.5 Autres

5.3.5.1 conditionnel ? :

C'est un (le seul) opérateur ternaire. L'expression a?b:c vaut la valeur de b si a est vrai (entier, différent de 0), et c si a est faux. Exemple : max=a>b?a:b

5.3.5.2 séquentiel ,

Cet opérateur permet de regrouper deux sous expressions en une seule. On effectue le premier opérande puis le second, la valeur finale de l'expression étant celle du second opérande. On l'utilise pour évaluer deux (ou plus) expressions là où la syntaxe du C ne nous permettait que d'en mettre une, exemple : for(i=j=0;i>10;i++,j++). Dans le cas d'une utilisation de cet opérateur dans une liste, utilisez les parenthèses pour distinguer les signes , : exemple (inutile) : printf("%d %d",(i++,j++),k) i est modifié mais sa valeur n'est pas affichée.

5.3.6 Ordre de priorité et associativité

opérateurs	associativité	description
() [] −> .	−>	

! ~ ++ −− − + & * (cast)	<−	unaires (* pointeurs)
* / %	−>	multiplicatifs
+ −	−>	addition
>> <<	−>	décalages
< <= > >=	−>	relations d'ordre
= = !=	−>	égalité
&	−>	binaire
^	−>	binaire
|	−>	binaire
&&	−>	logique
| |	−>	logique
? :	−>	conditionnel (ternaire)
= += −= *= etc.	<−	affectation
,	<−	séquentiel

Dans ce tableau, les opérateurs sont classés par priorité décroissante (même priorité pour les opérateurs d'une même ligne). Les opérateurs les plus prioritaires se verront évaluer en premier. L'associativité définit l'ordre d'évaluation des opérandes. La plupart se font de gauche à droite (4/2/2 donne (4/2)/2 donc 1 (et pas 4/(2/2))).

Les seules exceptions sont :

- les opérateurs unaires, écrits à gauche de l'opérateur. L'opérande est évalué puis l'opération est effectuée, le résultat est celui de l'opération; sauf dans le cas de l'incrémentation / décrémentation postfixée, où le résultat de l'expression est la valeur de l'argument avant l'opération.

- L'affectation : on calcule l'opérande de droite, puis on l'affecte à celui de gauche. Le résultat est la valeur transférée.

- La virgule : la valeur à droite est calculée avant celle à gauche (en particulier lors d'un appel de fonction)

- Les opérateurs logiques et conditionnel évaluent toujours leur premier argument. Le second par contre n'est évalué que si c'est nécessaire.

5.4 Instructions

Une instruction peut être :

− soit une expression (pouvant comprendre une affectation, un appel de fonction...), terminé par un ; qui en fait signifie "on peut oublier le résultat de l'expression et passer à la suite",
− soit une structure de contrôle (boucle, branchement...),
− soit un bloc d'instructions : ensemble de déclarations et instructions délimités par des accolades {}. Un bloc sera utilisé à chaque fois que l'on désire mettre plusieurs instructions là où on ne peut en mettre qu'une.

Seule la première forme est terminée par un ;. Un cas particulier est l'instruction vide, qui se compose uniquement d'un ; (utilisé là où une instruction est nécessaire d'après la syntaxe).

5.5 Structures de contrôle

Normalement, les instructions s'effectuent séquentiellement, c'est à dire l'une après l'autre. Pour accéder à une autre instruction que la suivante, on a trois solutions : le branchement inconditionnel, le branchement conditionnel et la boucle.

5.5.1 Boucles

Une boucle permet de répéter plusieurs fois un bloc d'instructions.

5.5.1.1 While (tant que)

structure : while (expression) instruction

Tant que l'expression est vraie (!=0), on effectue l'instruction, qui peut être simple (terminée par ;), bloc (entre {}) ou vide (; seul). L'expression est au moins évaluée une fois. Tant qu'elle est vraie, on effectue l'instruction, dès qu'elle est fausse, on passe à l'instruction suivante (si elle est fausse dès le début, l'instruction n'est jamais effectuée).

exemple :

```
#include <stdio.h>
void main(void)
 {
  float nombre,racine=0;
  puts("entrez un nombre réel entre 0 et 10");
  scanf("%f",&nombre);
  while (racine*racine<nombre) racine+=0.01;
  printf("la racine de %f vaut %4.2f à 1%% près\n",
      nombre, racine);
 }
```

Exercice (while_puiss) : faire un programme qui affiche toutes les puissances de 2, jusqu'à une valeur maximale donnée par l'utilisateur. On calculera la puissance par multiplications successives par 2. Cliquez ici pour une solution.

Exercice (while_err) : que fait ce programme ?

```
#include <stdio.h>
#include <math.h>
#define debut 100
#define pas 0.01
void main(void)
 {
  float nombre=debut;
  int compte=0,tous_les;
  puts("afficher les résultats intermédiaires
           tous les ? (333 par exemple) ?");
  scanf("%d",&tous_les);
  while (fabs(nombre−(debut+(compte*pas)))<pas)
   {
    nombre+=pas;
    if (!(++compte%tous_les))
       printf("valeur obtenue %12.8f, au lieu de %6.2f en %d calculs\n",
             nombre,(float)(debut+(compte*pas)), compte);
   }
  printf("erreur de 100%% en %d calculs\n",compte);
 }
```

Cliquez ici pour une solution.

5.5.1.2 Do While (faire tant que)

structure : do instruction while (expression); (attention au ; final)

comme while, mais l'instruction est au moins faite une fois, avant la première évaluation de l'expression.

exemple :

```
#include <stdio.h>
void main(void)
 {
  int a;
  do
   {
    puts("entrez le nombre 482");
    scanf("%d",&a);
   }
  while (a!=482);
  puts("c'est gentil de m'avoir obéi");
 }
```

Exercice (do_while) : écrivez un programme de jeu demandant de deviner un nombre entre 0 et 10 choisi par l'ordinateur. On ne donnera pas d'indications avant la découverte de la solution, où l'on indiquera le nombre d'essais. La solution sera choisie par l'ordinateur par la fonction rand() qui rend un entier aléatoire (déclarée dans stdlib.h). Cliquez ici pour une solution.

5.5.1.3 For (pour)

structure : for (expr_initiale;expr_condition;expr_incrémentation) instruction

Cette boucle est surtout utilisée lorsque l'on connaît à l'avance le nombre d'itération à effectuer. L'expr_initiale est effectuée une fois, en premier. Puis on teste la condition. On effectue l'instruction puis l'incrémentation tant que la condition est vraie. L'instruction et l'incrémentation peuvent ne jamais être effectuées. La boucle est équivalente à :

```
expr_initiale;
while (expr_condition)
 {
  instruction
  expr_incrémentation;
 }
```

Une ou plusieurs des trois expressions peuvent être omises, l'instruction peut être vide. for(;;); est donc une boucle infinie.

exemple :

```
{ char c; for(c='Z';c>='A';c--)putchar(c); }
```

Exercice (for) : faire un programme qui calcule la moyenne de N notes. N et les notes seront saisies par scanf. Le calcul de la moyenne s'effectue en initialisant une variable à 0, puis en y ajoutant progressivement les notes saisies puis division par N. Cliquez ici pour une solution.

5.5.2 Branchements conditionnels

On a souvent besoin de n'effectuer certaines instructions que dans certains cas. On dispose pour cela du IF et du SWITCH.

5.5.2.1 If − Else (Si − Sinon)

structure : if (expression) instruction1

ou : if (expression) instruction1 else instruction2

Si l'expression est vraie (!=0) on effectue l'instruction1, puis on passe à la suite. Sinon, on effectue l'instruction 2 puis on passe à la suite (dans le cas sans else on passe directement à la suite).

Exercice (jeu) : modifier le jeu de l'exercice (do_while) en précisant au joueur à chaque essai si sa proposition est trop grande ou trop petite. Cliquez ici pour une solution.

L'instruction d'un if peut être un autre if (imbriqué)

exemple :

```
if(c1) i1;
else if (c2) i2;
else if (c3) i3;
else i4;
i5;
```

si c1 alors i1 puis i5, sinon mais si c2 alors i2 puis i5, ... Si ni c1 ni c2 ni c3 alors i4 puis i5.

Le else étant facultatif, il peut y avoir une ambiguïté s'il y a moins de else que de if. En fait, un else se rapporte toujours au if non terminé (c'est à dire à qui on n'a pas encore attribué de else) le plus proche. On peut aussi terminer un if sans else en l'entourant de {}.

exemple : `if(c1) if(c2) i1; else i2;` : si c1 et c2 alors i1, si c1 et pas c2 alors i2, si pas c1 alors (quel que soit c2) rien.

`if (c1) {if (c2) i1;} else i2;` : si c1 et c2 alors i1, si c1 et pas c2 alors rien, si pas c1 alors i2.

5.5.2.2 Switch − Case (brancher − dans le cas)

Cette structure de contrôle correspond à un "goto calculé".

structure :

```
switch (expression_entière)
  {
    case cste1:instructions
    case cste2:instructions
        ........
    case csteN:instructions
    default :instructions
  }
```

L'expression ne peut être qu'entière (char, int, long). L'expression est évaluée, puis on passe directement au "case" correspondant à la valeur trouvée. Le cas default est facultatif, mais si il est prévu il doit être le dernier cas.

exemple : fonction vérifiant si son argument c est une voyelle.

```
int voyelle(char c)
 {
  switch(c)
   {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
    case 'y':return(1); /* 1=vrai */
    default :return(0);
   }
 }
```

Remarque : l'instruction break permet de passer directement à la fin d'un switch (au }). Dans le cas de switch imbriqués on ne peut sortir que du switch intérieur.

Exemple :

```
switch (a)
 {
  case 1:inst1;inst2;....;break;
  case 2:....;break;
  default:.....
 }
 /*endroit où l'on arrive après un break*/
```

Exercice (calcul) : faire un programme simulant une calculatrice 4 opérations. Cliquez ici pour une solution.

5.5.3 Branchements inconditionnels

Quand on arrive sur une telle instruction, on se branche obligatoirement sur une autre partie du programme. Ces instructions sont à éviter si possible, car ils rendent le programme plus complexe à maintenir, le fait d'être dans une ligne de programme ne suffisant plus pour connaître immédiatement quelle instruction on a fait auparavant, et donc ne permet plus d'assurer que ce qui est au dessus est correctement terminé. Il ne faut les utiliser que dans certains cas simples.

5.5.3.1 Break (interrompre)

Il provoque la sortie immédiate de la boucle ou switch en cours. Il est limité à un seul niveau d'imbrication.

exemples :
`do {if(i= =0)break;....}while (i!=0);` /* un while aurait été mieux */
`for (i=0;i<10;i++){....;if (erreur) break;}` /* à remplacer par
`for(i=0;(i<10)&&(!erreur);i++){...}` */

5.5.3.2 Continue (continuer)

Cette instruction provoque le passage à la prochaine itération d'une boucle. Dans le cas d'un while ou do while, saut vers l'évaluation du test de sortie de boucle. Dans le cas d'un for on passe à l'expression d'incrémentation puis seulement au test de bouclage. En cas de boucles imbriquées, permet uniquement de continuer la boucle la plus interne.

exemple :
`for (i=0;i<10;i++) {if (i= =j) continue; ..... }`
peut être remplacé par :

```
for (i=0;i<10;i++) if (i!=j) { ..... }
```

5.5.3.3 Goto (aller à)

La pratique des informaticiens a montré que l'utilisation des goto donne souvent des programmes non maintenables (impossibles à corriger ou modifier). Les problèmes qu'ils posent ont amené les programmeurs expérimentés à ne s'en servir qu'exceptionnellement.

structure : goto label;

Label est un identificateur (non déclaré, mais non utilisé pour autre chose), suivi de deux points (:), et indiquant la destination du saut. Un goto permet de sortir d'un bloc depuis n'importe quel endroit. Mais on ne peut entrer dans un bloc que par son { (qui créera proprement les variables locales du bloc).

```
{.....
  {.....
   goto truc;
   .....
  }
 .....
 truc:
 .....
}
```

5.5.3.4 Return (retourner)

Permet de sortir de la fonction actuelle (y compris main), en se branchant à son dernier }. Return permet également (et surtout) de rendre la valeur résultat de la fonction.

structure : return ou return(valeur)

exemple :
```
int max(int a, int b) {if (a>b) return(a); else return(b);}
```

5.5.3.5 Exit (sortir)

Ceci n'est pas un mot clef du C mais une fonction disponible dans la plupart des compilateurs (définie par ANSI, dans STDLIB.H). Elle permet de quitter directement le programme (même depuis une fonction). On peut lui donner comme argument le code de sortie (celui que l'on aurait donné à return dans main). Cette fonction libère la mémoire utilisée par le programme (variables + alloc) et ferme (sur beaucoup de compilateurs) les fichiers ouverts.

structure : exit(); ou exit(valeur);

5.6 Déclaration et stockage des variables

Une variable doit être définie par le programmateur dans une déclaration, où l'on indique le nom que l'on désire lui donner, son type (int, float,...) pour que le compilateur sache combien de mémoire il doit lui réserver et les opérateurs qui peuvent lui être associés, mais aussi comment elle doit être gérée (visibilité, durée de vie,...).

5.6.1 Déclarations locales

Dans tout bloc d'instructions, avant la première instruction, on peut déclarer des variables. Elles seront alors "locales au bloc" : elles n'existent qu'à l'intérieur du bloc. Ces variables sont mises en mémoire dans une zone de type "pile" : quand, à l'exécution, on arrive sur le début du bloc ({), on réserve la mémoire nécessaire aux

variables locales au sommet de la pile (ce qui en augmente la hauteur), et on les retire en quittant le bloc. L'intérêt consiste à n'utiliser, à un instant donné, que la quantité nécessaire de mémoire, qui peut donc resservir par après pour d'autres variables. Le désavantage (rarement gênant et pouvant être contrecarré par la classe STATIC) est qu'en quittant le bloc (par } ou un branchement), et y entrant à nouveau plus tard (par son {), les variables locales ne sont plus nécessairement recréées au même endroit, et n'auront plus le même contenu. De plus la libération/réservation de la pile aura fait perdre un peu de temps. Par contre, lorsque l'on quitte temporairement un bloc (par appel à une fonction), les variables locales restent réservées. La sortie d'un bloc par un branchement gère la libération des variables locales, mais seule l'entrée dans un bloc par son { gère leur création.

Exemple :

```
  #include <stdio.h>
  int doubl(int b)
    {int c; c=2*b; b=0; return(c); }
  void main(void)
   {
    int a=5;
    printf("%d %d\n",doubl(a),a);
   }
```

A l'entrée du bloc main, création de a, à qui l'on donne la valeur 5. Puis appel de doubl : création de b au sommet de la pile, on lui donne la valeur de a. Puis entrée dans le bloc, création sur la pile de c, on lui donne la valeur b*2=10, on annule b (mais pas a), on rend 10 à la fonction appelante, et on libère le sommet de la pile (c et b n'existent plus) mais a reste (avec son ancienne valeur) jusqu'à la sortie de main. On affichera donc : 10 5.

Une variable locale est créée à l'entrée du bloc, et libérée à la sortie. Cette période est appelée sa durée de vie. Mais pendant sa durée de vie, une variable peut être visible ou non. Elle est visible : dans le texte source du bloc d'instruction à partir de sa déclaration jusqu'au }, mais tant qu'une autre variable locale de même nom ne la cache pas. Par contre elle n'est pas visible dans une fonction appelée par le bloc (puisque son code source est hors du bloc).

autre exemple :

```
void main(void);
 {int a=1;             [1]
   {int b=2;           [2]
     {int a=3;         [3]
      fonction(a);     [4]
     }                 [5]
    fonction(a);       [6]
   }                   [7]
 }                     [8]
int fonction (int b)   [a]
 {int c=0;             [b]
  c=b+8;               [c]
 }                     [d]
```

analysons progressivement l'évolution de la pile au cours du temps (en gras : variable visible) :

[1] **a=1**
[2] **a=1** | **b=2**
[3] a=1 | **b=2** | **a=3** : seul le a le plus haut est visible (a=3), l'autre vit encore (valeur 1 gardée) mais n'est plus visible.
[4a] a=1 | b=2 | a=3 | **b=3** : entrée dans la fonction, recopie de l'argument réel (a) dans l'argument formel (b). Mais a n'est plus visible.
[4b] a=1 | b=2 | a=3 | **b=3** | **c=0**

[4c] a=1 | b=2 | a=3 | **b=3** | **c=11** : quand le compilateur cherche la valeur de b, il prend la plus haute de la pile donc 3 (c'est la seule visible), met le résultat dans le c le plus haut. L'autre b n'est pas modifié.
[4d] a=1 | **b=2** | **a=3** : suppression des variables locales b et c du sommet de la pile
[5] **a=1** | **b=2** : sortie de bloc donc libération de la pile
[6a] a=1 | b=2 | **b=1** : l'argument réel (a) n'est plus le même qu'en [4a]
[6b] a=1 | b=2 | **b=1** | **c=0**
[6c] a=1 | b=2 | **b=1** | **c=9**
[6d] **a=1** | **b=2** : suppression b et c
[7] **a=1**
[8] la pile est vide, on quitte le programme

Notez que la réservation et l'initialisation prennent un peu de temps à chaque entrée du bloc. Mais ne présumez jamais retrouver une valeur sur la pile, même si votre programme n'a pas utilisé la pile entre temps (surtout sur système multitâche ou avec mémoire virtuelle).

Une déclaration a toujours la structure suivante :

[classe] type liste_variables [initialisateur]; (entre [] facultatif)

Le **type** peut être simple (char, int, float,...) ou composé (tableaux, structures..., voir plus loin).

La **liste_variables** est la liste des noms des variables désirées, séparées par des virgules s'il y en a plusieurs. Chaque nom de la liste peut être précédés d'une *, ceci spécifiant un pointeur.

L'**initialisateur** est un signe =, suivi de la valeur à donner à la variable lors de sa création (à chaque entrée du bloc par exemple). La valeur peut être une constante, mais aussi une expression avec des constantes voire des variables (visibles, déjà initialisées).

La **classe** peut être :

- **auto** (ou omise, c'est la classe par défaut pour les variables locales) : la variable est créée à l'entrée du bloc (dans la pile) et libérée automatiquement à sa sortie (comme expliqué plus haut).
- **register** : la variable est créée, possède la même durée de vie et visibilité qu'une classe auto, mais sera placée dans un registre du (micro)processeur. Elle sera donc d'un accès très rapide. Si tous les registres sont déjà utilisés, la variable sera de classe auto. Mais le compilateur peut avoir besoin des registres pour ses besoins internes ou pour les fonctions des bibliothèques, s'il n'en reste plus le gain peut se transformer en perte. De plus les compilateurs optimisés choisissent de mettre en registre des variables auto, et souvent de manière plus pertinente que vous. Mais le compilateur ne sait pas à l'avance quelles fonctions seront appelées le plus souvent, dans ce cas une optimisation manuelle peut être utile, par exemples dans le cas des éléments finis où une même instruction peut être répétée des milliards de fois.
- **static** : la variable ne sera pas dans la pile mais dans la même zone que le code machine du programme. Sa durée de vie sera donc celle du programme. Elle ne sera initialisée qu'une fois, au début du programme, et restera toujours réservée. En retournant dans un bloc, elle possédera donc encore la valeur qu'elle avait à la précédente sortie. Sa visibilité reste la même (limitée au bloc). Une variable statique permet en général un gain en temps d'exécution contre une perte en place mémoire.

5.6.2 Déclarations globales

Une déclaration faite à l'extérieur d'un bloc d'instructions (en général en début du fichier) est dite globale. La variable est stockée en mémoire statique, sa durée de vie est celle du programme. Elle est visible de sa déclaration jusqu'à la fin du fichier. Elle sera initialisée une fois, à l'entrée du programme (initialisée à 0 si pas d'autre précision). Le format d'une déclaration globale est identique à une déclaration locale, seules les classes varient.

Par défaut, la variable est publique, c'est à dire qu'elle pourra même être visible dans des fichiers compilés séparément (et reliés au link).

La classe **static**, par contre, rend la visibilité de la variable limitée au fichier actuel.

La classe **extern** permet de déclarer une variable d'un autre fichier (et donc ne pas lui réserver de mémoire ici, mais la rendre visible). Elle ne doit pas être initialisée ici. Une variable commune à plusieurs fichiers devra donc être déclarée sans classe dans un fichier (et y être initialisée), extern dans les autres.

Toute fonction, pour pouvoir être utilisée, doit également être déclarée. Une déclaration de fonction ne peut être que globale, et connue des autres fonctions. Une déclaration de fonction est appelée "prototype". Le prototype est de la forme :

[classe] type_retourné nom_fonction(liste_arguments);

elle est donc identique à l'entête de la fonction mais :
− est terminée par un ; comme toute déclaration
− les noms des arguments n'ont pas besoin d'être les mêmes, il peuvent même être omis (les types des arguments doivent être identiques).

Sans précision de classe, la fonction est publique. Sinon, la classe peut être **extern** (c'est ce que l'on trouve dans les fichiers .H) ou **static** (visibilité limitée au fichier). Le prototype peut être utilisé pour utiliser une fonction du même fichier, mais avant de l'avoir définie (par exemple si l'on veut main en début du fichier). En général, lorsque l'on crée une bibliothèque (groupe de fonctions et variables regroupées dans un fichier compilé séparément), on prépare un fichier regroupant toutes les déclarations extern, noté .H, qui pourra être inclus dans tout fichier utilisant la bibliothèque.

exemples de déclarations globales :

```
int i,j; /* publiques, initialisées à 0 */
static int k=1; /* privée, initialisée à 1 */
extern int z; /* déclarée (et initialisée) dans un autre fichier */
float produit(float,float); /* prototype d'une fonction définie plus
loin dans ce fichier */
extern void échange(int *a, int *b); /* prototype d'une fonction définie
dans un autre fichier */
```

Avant la norme ANSI, le prototype n'existait pas. Une fonction non définie auparavant était considérée comme rendant un int (il fallait utiliser un cast si ce n'était pas le cas).

5.6.3 Déclaration de type

La norme ANSI permet de définir de nouveaux types de variables par typedef.

structure : typedef type_de_base nouveau_nom;

Ceci permet de donner un nom à un type donné, mais ne crée aucune variable. Une déclaration typedef est normalement globale et publique.

exemple :

```
typedef long int entierlong; /* définition d'un nouveau type */
```

entierlong i; /* création d'une variable i de type entierlong */
typedef entierlong *pointeur; /* nouveau type : pointeur = pointeur d'entierlong */
pointeur p; /* création de p (qui contiendra une adresse), peut être initialisé par =&i */

Remarque : Le premier typedef pouvait être remplacé par un #define mais pas le second.

5.7 Fonctions

5.7.1 Définitions générales

Une fonction est définie par son entête, suivie d'un bloc d'instructions

entête : type_retourné nom_fonction(liste_arguments) (pas de ;)

Avant la norme ANSI, le **type_retourné** pouvait être omis si int. Désormais il est obligatoire, si la fonction ne retourne rien on indique : void.

La **liste_arguments** doit être typée (ANSI), alors qu'auparavant les types étaient précisés entre l'entête et le bloc :
ANSI: float truc(int a, float b) {bloc}
K&R: float truc(a,b) int a;float b; {bloc}

Si la fonction n'utilise pas d'arguments il faut la déclarer (ANSI) nom(void) ou (K&R) nom(). L'appel se fera dans les deux cas par nom() (parenthèses obligatoires).

Les arguments (formels) sont des variables locales à la fonction. Les valeurs fournies à l'appel de la fonction (arguments réels) y sont recopiés à l'entrée dans la fonction. Les instructions de la fonction s'exécutent du début du bloc ({) jusqu'à return(valeur) ou la sortie du bloc (}). La valeur retournée par la fonction est indiquée en argument de return.

exemple :

```
float produit(float a;float b)
 {
  float z;
  z=a*b;
  return(z);
 }
```

5.7.2 Récursivité, gestion de la pile

Une fonction peut s'appeler elle−même :

```
int factorielle(int i)
 {
  if (i>1) return(i*factorielle(i−1));
  else return(1);
 }
```

analysons la pile en appelant factorielle(3) :

		i=1			
	i=2	i=2	i=2		
i=3	i=3	i=3	i=3	i=3	
(a)	**(b)**	**(c)**	**(d)**	**(e)**	

- (a) appel de factorielle(3), création de i, à qui on affecte la valeur 3. comme i>1 on calcule i*factorielle(i−1) : i=3,i−1=2 on appelle factorielle(2)
- (b) création i, affecté de la valeur 2, i>1 donc on appelle factorielle(1)

- (c) création de i, i=1 donc on quitte la fonction, on libère le pile de son sommet, on retourne où la fonction factorielle(1) a été appelée en rendant 1.
- (d) on peut maintenant calculer i*factorielle(1), i (sommet de la pile) vaut 2, factorielle(1) vaut 1, on peut rendre 2, puis on "dépile" i
- (e) on peut calculer i*factorielle(2), i vaut 3 (sommet de la pile), factorielle(2) vaut 2 3*2=6, on retourne 6, la pile est vidée et retrouve sont état initial.

Attention, la récursivité est gourmande en temps et mémoire, il ne faut l'utiliser que si l'on ne sait pas facilement faire autrement :

```
int factorielle(int i)
 {
  int result;
  for(result=1;i>1;i--) result*=i;
  return(result);
 }
```

5.7.3 Arguments passés par adresse

Imaginons la fonction :

```
void échange(int i;int j)
{int k;k=i;i=j;j=k;}
```

Lors d'un appel à cette fonction par échange(x,y), les variables locales i,j,k sont créées sur la pile, i vaut la valeur de x, j celle de y. Les contenus de i et j sont échangés puis la pile est libérée, sans modifier x et y. Pour résoudre ce problème, il faut passer par des pointeurs. On utilisera les opérateurs unaires : & (adresse de) et * (contenu de). Définissons donc la fonction ainsi :

```
void échange(int *i;int *j)
{int k;k=*i;*i=*j;*j=k;}
```

On appelle la fonction par **échange(&x,&y);** les deux arguments formels de la fonction (i et j) sont des pointeurs sur des int, c'est à dire qu'à l'appel, on crée sur la pile des variables i et j pouvant contenir une adresse, dans i on recopie l'argument réel qui et l'adresse de x, et l'adresse de y dans j. en entrant dans le bloc, on crée une variable locale k pouvant contenir un entier. Puis on met dans k non pas la valeur de i mais le contenu pointé par i (donc ce qui se trouve à l'adresse marquée dans i, qui est l'adresse de x), donc le contenu de x. On place la valeur pointée par j (donc y) à l'adresse pointée par j (donc x). Puis on place la valeur de k à l'adresse pointée par j (y). On a donc échangé x et y.

On a tout intérêt à essayer cet exemple en se fixant des adresses et des valeurs, et voir l'évolution des contenus des variables.

En conclusion, pour effectuer un passage d'argument par adresse, il suffit d'ajouter l'opérateur & devant l'argument réel (à l'appel de la fonction), et l'opérateur * devant chaque apparition de l'argument formel, aussi bien dans l'entête que le bloc de la fonction.

5.7.4 La fonction main

Si on le désire, la fonction main peut rendre un entier (non signé) au système d'exploitation (sous MSDOS, on récupère cette valeur par ERRORLEVEL). Une valeur 0 signale en général une fin normale du programme, sinon elle représente un numéro d'erreur. L'arrivée sur le } final retourne la valeur 0, dans le cas où on n'a pas indiqué de return(code).

De même, le système d'exploitation peut transmettre des arguments au programme. La déclaration complète de l'entête de la fonction main est :

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

Le dernier argument est optionnel).

On peut aussi utiliser `char **argv`, mais cela peut paraître moins clair. `argc` indique le nombre de mots de la ligne de commande du système d'exploitation, `argv` est un tableau de pointeurs sur chaque mot de la ligne de commande, env pointe sur les variables de l'environnement (sous MSDOS obtenues par **SET**, mais aussi très utilisées sous UNIX par **env**).

Si votre programme s'appelle COPIER, et que sous MSDOS vous ayez entré la commande COPIER TRUC MACHIN alors argc vaut 3, argv[0] pointe sur "COPIER", argv[1] sur "TRUC" et argv[2] sur "MACHIN". argv[3] vaut le pointeur NULL. env est un tableau de pointeurs sur les variables d'environnement, on n'en connaît pas le nombre mais le dernier vaut le pointeur NULL.

5.7.5 Fonction retournant un pointeur et pointeur de fonction

type*fonc(arguments) est une fonction qui renvoie un pointeur.

exemple :

```
int *max(int tableau[], int taille)
 {
  int i,*grand;
  for(grand=tableau,i=1;i<taille;i++)
            if(tableau[i]>*grand) grand=tableau+i;
  return(grand);
 }
```

Cette fonction rend l'adresse du plus grand entier du tableau.

type (*fonc)(arguments) est un pointeur sur une fonction

exemple :

```
int max(int,int);
int min(int,int);
void main(void);
 {
  int (*calcul)(int,int);
  /* calcul est un pointeur donc une variable qui
                        peut être locale */
  char c;
  puts("utiliser max (A) ou min (I) ?");
  do c=getchar(); while ((c!='A')&&(c!='I'));
  calcul=(c= ='A')?&max:&min;
  printf("%d\n",(*calcul)(10,20));
 }
int max(int a,int b) {return(a>b?a:b);}
int min(int a,int b) {return(a<b?a:b);}
```

Cette fonctionnalité du C est assez peu utilisée, mais est nécessaire dans les langages orientés objets.

5.8 Les types de données du C

Nous allons définir les différents types de variables existantes en C. On verra les types scalaires (entiers,...) et les types agrégés (combinaisons de scalaires, tableaux par exemple).

5.8.1 Variables scalaires

On appelle variable scalaire une variable ne contenant qu'une seule valeur, sur laquelle on pourra faire un calcul arithmétique. On possède trois types de base (char, int, float) que l'on peut modifier par 3 spécificateurs (short, long, unsigned).

5.8.1.1 char : caractère (8 bits)

Une constante caractère est désignée entre apostrophes (simples quotes). 'a' correspond à un octet, alors que "a" à deux ('a' et \0). On peut définir certains caractères spéciaux, par le préfixe \ (antislash) :

- \n nouvelle ligne
- \t tabulation
- \b backspace
- \r retour chariot (même ligne)
- \f form feed (nouvelle page)
- \' apostrophe
- \\ antislash
- \" double quote
- \0 nul
- \nombre en octal sur 3 chiffres (ou moins si non suivi d'un chiffre).
- \xnombre : en hexa

Les char sont considérés dans les calculs comme des int (on considère leur code ASCII). Par défaut en C un char est signé donc peut varier de −128 à +127. Pour utiliser les caractères spéciaux du PC (non standard), il faut utiliser des **unsigned char** (de 0 à 255).

5.8.1.2 int : entier

Si l'on désire une taille précise, utiliser **short int** (16 bits) ou **long int** (32 bits). Sans précision, int donnera les programmes les plus rapides pour une machine donnée (int = short sur PC, mais long sur les stations 32 bits). Par défaut, les int sont signés, mais on peut préciser **unsigned int**.

Désormais certains compilateurs considèrent short comme 16 bits, int comme 32 bits et long comme 64 bits.

5.8.1.3 float : réel

Un flottant est un nombre stocké en deux parties, une mantisse et un exposant. La taille de la mantisse définit le nombre de chiffres significatifs, alors que la taille de l'exposant définit le plus grand nombre acceptable par la machine. Les opérations sur les réels sont plus lents que sur les entiers. Pour une addition par exemple, il faut d'abord décaler la mantisse pour égaliser les exposants puis faire l'addition. Les réels sont toujours signés. On peut par contre utiliser le spécificateur long pour des réels avec une précision accrue. On peut également utiliser le nom **double** au lieu de **long float**. Certains compilateurs acceptent même des **long double** (quadruple précision).

5.8.1.4 Tailles et plages

type	**taille** (en bits)	**plage de valeurs**
char	8	−128 à +127
unsigned char	8	0 à 255
short (short int)	16	−32768 à 32767

unsigned short	16	0 à 65535
long (long int)	32	−2.147.483.648 à 2.147.483.647
unsigned long	32	0 à 4.294.967.295
float	32	−3.4e38 à 3.4e38 (7 chiffres significatifs)
double (long float)	64	−1.7e308 à 1.7e308 (15 chiffres significatifs)
long double (non standard)	80	3.4E−4932 à 1.1E+4932 (19 chiffres signif.)

5.8.1.5 Conversions de type / cast

Dans les calculs, les char sont automatiquement transformés en int. Quand un opérateur possède des arguments de type différent, une transformation de type est effectuée automatiquement, suivant l'ordre :
char −> int −> long −> float −> double
signed −> unsigned

Attention la transformation n'est effectuée que le plus tard possible, si nécessaire. 5/2+3.5 donnera donc 5.5. De plus les opérations arithmétiques sont toujours effectuées sur des long ou double, pour une précision maximale quels que soient les résultats intermédiaires (voir exemples au chapitre expressions arithmétiques).

On peut forcer une transformation en utilisant le **cast**, qui est un opérateur unaire. La syntaxe est :
(type_résultat) valeur_à_transformer

exemple : `{float x;int a=5; x=(float)a;}`

Un cast transformant un réel en entier prendra la partie entière. Cette transformation doit être explicite, elle est impossible implicitement. Pour obtenir l'entier le plus proche , utiliser `(int)(réel_positif+0.5)`.

Il faut bien noter que le cast n'est une opération de transformation que pour les types scalaires, pour tous les autres types, le cast ne permet que de faire croire au compilateur que la variable est d'un autre type que ce qu'il attendait, pour qu'il n'émette pas de message d'erreur (à utiliser avec grande prudence).

5.8.1.6 Enumérations

On peut définir un nouveau type (d'entiers) sous la forme :
[classe] enum nomdutype {liste_valeurs} [liste_variables]; ([] facultatifs)

```
exemple : enum jour {lundi, mardi, mercredi, jeudi, vendredi, samedi,
dimanche};
```

On a créé un nouveau type. toute variable de type jour pourra valoir soit lundi, soit mardi, etc... On peut directement mettre la liste des variables à créer (entre le "}" et le ";"), ou indiquer :
`enum nomdutype liste_variables;`

exemple : `enum jour aujourd_hui=mardi;)`

En fait le type jour est un type int, avec lundi=0, mardi=1,... On peut donc faire toutes les opérations entières (aujourd_hui++ par exemple). Il n'y a aucun test de validité (dimanche+1 donne 7). Ceci permet de rendre les programmes plus clairs. On obtiendrait un résultat équivalent avec #define. Attention, printf affichera un entier, mais on peut faire:

```
char *nom[7]={"lundi", "mardi", "mercredi", "jeudi", "vendredi",
"samedi"," dimanche"};
```
puis `printf("%s",nom[aujourd_hui]);`

On peut aussi prévoir une codification non continue :
```
enum truc {a=4,b,c,d=2,e,f} : d=2,e=3,f=a=4,b=5,c=6
```

En utilisant typedef, on n'a pas besoin de répéter enum dans la déclaration de variables :
```
typedef enum {coeur,carreau,pique,trèfle}couleurs;
couleurs i,j,k;
```

5.9 Tableaux

5.9.1 Tableaux unidimensionnels

Un tableau est un regroupement, dans une même variable, de plusieurs variables simples, toutes de même type.

déclaration : [classe] type nom [nombre_d'éléments];

exemple : `int tab[10];`

Ceci réserve en mémoire un espace contigu pouvant contenir 10 entiers. Le premier est tab[0], jusqu'à tab[9]. Attention, en utilisant tab[10] ou plus, aucune erreur ne sera signalée et vous utiliserez une partie de mémoire qui a certainement été réservée pour autre chose. Il est possible de définir un tableau de n'importe quel type de composantes (scalaires, pointeurs, structures et même tableaux). Il est également possible de définir un type tableau par typedef :
```
typedef float vecteur[3];
vecteur x,y,z;
```

On peut aussi initialiser un tableau. Dans ce cas la dimension n'est pas nécessaire. Mais si elle est donnée, et est supérieure au nombre de valeurs données, les suivantes seront initialisées à 0 :
```
vecteur vect0={0,0,0};
int chiffres[]={0,1,2,3,4,5,6,7,8,9};
int tableau[20]={1,2,3}; /* les 17 autres à 0 */
```

On peut également déclarer un tableau sans en donner sa dimension. Dans ce cas là le compilateur ne lui réserve pas de place, elle aura du être réservée autre part (par exemple tableau externe ou argument formel d'une fonction).

Exercice (moyenne) : Ecrire le programme qui lit une liste de Nb nombres, calcule et affiche la moyenne puis l'écart entre chaque note et cette moyenne. Cliquez ici pour une solution.

5.9.2 Tableaux et pointeurs / arithmétique des pointeurs

En déclarant, par exemple, `int TAB[10];` l'identificateur TAB correspond en fait à l'adresse du début du tableau. Les deux écritures TAB et &TAB[0] sont équivalentes (ainsi que TAB[0] et *TAB). On définit l'opération d'incrémentation pour les pointeurs par TAB+1=adresse de l'élément suivant du tableau. L'arithmétique des pointeurs en C a cette particularité que l'opération dépend du type de variable pointée, ajouter 1 consistant à ajouter à l'adresse la taille de l'objet pointé. On définit donc l'addition (pointeur+entier): TAB+i=&TAB[i], la soustraction (pointeur − entier), mais également la soustraction (pointeur − pointeur) qui donne un nombre d'éléments. Les opérations de comparaisons entre pointeurs sont donc également possibles.

Déclarons : `int TAB[10],i,*ptr;`

Ceci réserve en mémoire
− la place pour 10 entiers, l'adresse du début de cette zone est TAB,
− la place pour l'entier i,

− la place pour un pointeur d'entier (le type pointé est important pour définir l'addition).

Analysons les instructions suivantes :

```
ptr=TAB; /*met l'adresse du début du tableau dans ptr*/
for(i=0;i<10;i++)
 {
  printf("entrez la %dième valeur :\n",i+1);
  /* +1 pour commencer à 1*/
  scanf("%d",ptr+i);
  /* ou &TAB[i] puisque scanf veut une adresse*/
 }
puts("affichage du tableau");
for(ptr=TAB;ptr<TAB+10 /* ou &TAB[10] */;ptr++)
  printf("%d ",*ptr);
puts(" ");
/* attention actuellement on pointe derrière le tableau ! */
ptr−=10; /* ou plutôt ptr=TAB qui lui n'a pas changé */
printf("%d",*ptr+1); /* affiche (TAB[0])+1 */
printf("%d",*(ptr+1)); /* affiche TAB[1] */
printf("%d",*ptr++); /* affiche TAB[0] puis pointe sur TAB[1] */
printf("%d",(*ptr)++); /* affiche TAB[1] puis ajoute 1 à TAB[1]*/
```

TAB est une "constante pointeur", alors que ptr est une variable (donc TAB++ est impossible). La déclaration d'un tableau réserve la place qui lui est nécessaire, celle d'un pointeur uniquement la place d'une adresse.

Pour passer un tableau en argument d'une fonction, on ne peut que le passer par adresse (recopier le tableau prendrait de la place et du temps).

exemple utilisant ces deux écritures équivalentes :

```
#include <stdio.h>
void annule_tableau(int *t,int max)
 {
  for(;max>0;max−−)*(t++)=0;
 }
void affiche_tableau(int t[], int max)
 {
  int i;
  for(i=0;i<max;i++) printf("%d : %d\n",i,t[i]);
 }
void main(void)
 {
  int tableau[10];
  annule_tableau(tableau,10);
  affiche_tableau(tableau,10);
 }
```

Exercice (rotation) : Ecrire un programme qui lit une liste de Nb nombres, la décale d'un cran vers le haut (le premier doit se retrouver en dernier), l'affiche puis la décale vers le bas. On pourra décomposer le programme en fonctions. Cliquez ici pour une solution.

Exercice (classer) : Classer automatiquement un tableau de Nb entiers puis l'afficher dans l'ordre croissant puis décroissant. On pourra utiliser des fonctions de l'exercice précédent. On pourra créer un (ou plusieurs) tableau temporaire (donc local). Si vous vous en sentez la force, prévoyez le cas de valeurs égales. Cliquez ici pour une solution.

5.9.3 Chaînes de caractères

En C, comme dans les autres langages, certaines fonctionnalités ont été ajoutées aux tableaux dans le cas des tableaux de caractères. En C, on représente les chaînes par un tableau de caractères, dont le dernier est un caractère de code nul (\0). Une constante caractères est identifiée par ses délimiteurs, les guillemets " (double quote).

exemples :
```
puts("salut");
char mess[]="bonjour"; /* évite de mettre ={'b','o',..,'r',\0} */
puts (mess);
```

mess est un tableau de 8 caractères (\0 compris). On peut au cours du programme modifier le contenu de mess, à condition de ne pas dépasser 8 caractères (mais on peut en mettre moins, le \0 indiquant la fin de la chaîne). Mais on peut également initialiser un pointeur avec une chaîne de caractères :
```
char *strptr="bonjour";
```

Le compilateur crée la chaîne en mémoire de code (constante) et une variable strptr contenant l'adresse de la chaîne. Le programme pourra donc changer le contenu de strptr (et donc pointer sur une autre chaîne), mais pas changer le contenu de la chaîne initialement créée.

Exercice (chaînes) : écrire un programme qui détermine le nombre et la position d'une sous−chaîne dans une chaîne (exemple ON dans FONCTION : en position 1 et 6). Cliquez ici pour une solution.

5.9.4 Bibliothèques de fonctions pour tableaux et chaînes

Toutes les fonctions standard d'entrée / sortie de chaînes considèrent la chaîne terminée par un \0, c'est pourquoi en entrée elles rajoutent automatiquement le \0 (gets, scanf), en sortie elles affichent jusqu'au \0 (puts, printf). La bibliothèque de chaînes (inclure string.h) possède des fonctions utiles à la manipulation de chaînes :

- `int strlen(chaîne)` donne la longueur de la chaîne (\0 non compris)
- `char *strcpy(char *destination,char *source)` recopie la source dans la destination, rend un pointeur sur la destination
- `char *strncpy(char *destination,char *source,int longmax)` idem strcpy mais s'arrête au \0 ou longmax (qui doit comprendre le \0)
- `char *strcat(char *destination,char *source)` recopie la source à la suite de la destination, rend un pointeur sur la destination
- `char *strncat(char *destination,char *source,int longmax)` idem
- `int strcmp(char *str1,char*str2)` rend 0 si str1= =str2, <0 si str1<str2, >0 si str1>str2. Idem strncmp

Des fonctions similaires, mais pour tous tableaux (sans s'arrêter au \0) sont déclarées dans mem.h. La longueur est à donner en octets (on peut utiliser sizeof) :

- int memcmp(void *s1,void *s2,int longueur);
- void *memcpy(void *dest,void *src,int longueur);

On possède également des fonctions de conversions entre scalaires et chaînes, déclarées dans stdlib.h

- int atoi(char *s) traduit la chaîne en entier (s'arrête au premier caractère impossible, 0 si erreur dès le premier caractère)
- de même atol et atof

Dans ctype.h, on trouve des fonctions utiles (limitées au caractères) :

- int isdigit(int c) rend un entier non nul si c'est un chiffre ('0' à '9'), 0 sinon.
- de même : isalpha (A à Z et a à z, mais pas les accents), isalnum (isalpha||isdigit), isascii (0 à 127), iscntrl (0 à 31), islower (minuscule), isupper, isspace (blanc, tab, return...), isxdigit (0 à 9,A à F,a à f)...

- int toupper(int c) rend A à Z si c est a à z, rend c sinon. Egalement tolower

5.9.5 Allocation dynamique de mémoire

La taille déclarée d'un tableau est définie à la compilation. Dans le cas d'une taille inconnue à l'avance, il faut donc surdimensionner le tableau (et donc réserver des mémoires dont on ne servira que rarement, au dépends d'autres variables ou tableaux. En C, le lien entre les tableaux et pointeurs permet de réserver, lors de l'exécution, une zone de mémoire contiguë, de la taille désirée (mais pas plus). Il faut d'abord déclarer une variable pointeur qui contiendra l'adresse du début du tableau. A l'exécution, lorsque l'on connaît la taille désiré, on peut réserver une zone mémoire (dans la zone appelée "tas" ou "heap") par les fonctions :

− void ***malloc**(int taille) : réserve une zone mémoire contiguë de taille octets, et retourne un pointeur sur le début du bloc réservé. Retourne le pointeur NULL en cas d'erreur (en général car pas assez de mémoire).

− void ***calloc**(int nb, int taille) : équivalent à malloc(nb*taille).

exemple :

```
float *tab;
int nb;
puts("taille désirée ?");
scanf("%d",&nb);
tab=(float*)calloc(nb,sizeof(float));
```

malloc et calloc nécessitent un cast pour que le compilateur ne signale pas d'erreur.

− void **free**(void *pointeur) libère la place réservée auparavant par malloc ou calloc. Pointeur est l'adresse retournée lors de l'allocation. En quittant proprement le programme, la mémoire allouée est automatiquement restituée même si on omet d'appeler free.

− void ***realloc**(void *pointeur,int taille) essaie, si possible, de réajuster la taille d'un bloc de mémoire déjà alloué (augmentation ou diminution de taille). Si nécessaire, le bloc est déplacé et son contenu recopié. En retour, l'adresse du bloc modifié (pas nécessairement la même qu'avant) ou le pointeur NULL en cas d'erreur.

Ces fonctions sont définies dans stdlib.h ou alloc.h (suivant votre compilateur).

Une erreur fréquente consiste à "perdre" l'adresse du début de la zone allouée (par tab++ par exemple) et donc il est alors impossible d'accéder au début de la zone, ni de la libérer.

5.9.6 Tableaux multidimensionnels

On peut déclarer par exemple int tab[2][3] : matrice de 2 lignes de 3 éléments. Un tableau peut être initialisé : int t[2][3]={{1,2,3},{4,5,6}} mais cette écriture est équivalente à {1,2,3,4,5,6}, car il place dans l'ordre t[0][0],t[0][1],t[0][2],t[1][0],t[1][1],t[1][2], c'est à dire ligne après ligne. Dans un tableau multidimensionnel initialisé, seule la dimension la plus à gauche peut être omise (ici int t[][3]={...} était possible).

t correspond à l'adresse &t[0][0], mais t[1] est aussi un tableau (une ligne), donc désigne l'adresse &t[1][0]. En fait, une matrice est un tableau de lignes. On peut expliciter cela par typedef :

```
typedef int ligne[3];
typedef ligne matrice[2];
```

En utilisant pointeurs et allocation dynamique, pour gérer un tableau de NBLIG lignes de NBCOL éléments, , on peut :

- soit créer une matrice complète : allocation par t=malloc(NBLIG* NBCOL* sizeof(élément)), accès

à l'élément l,c par *(t+l*NBCOL+c).

- soit créer un tableau de NBLIG pointeurs de lignes, puis chaque ligne séparément. Ceci permet une optimisation si les lignes n'ont pas toutes la même longueur (traitement de textes par exemple) mais aussi de manipuler facilement les lignes (exemple : échanger deux lignes sans recopier leur contenu). Cette méthode est plus rapide que la précédente, car les adresses de chaque début de ligne sont immédiatement connues, sans calcul.
- soit utiliser des pointeurs de pointeurs (même principe que le cas précédent, mais remplacement du tableau de pointeurs (dimension prévue à l'avance) par une allocation dynamique.

Exercice (matrices) : faire le calcul de multiplication d'une matrice (M lignes, L colonnes) par une matrice (L,N) : résultat (M,N).
Cliquez ici pour une solution.

Exercice (déterminant) : écrire un programme qui calcule le déterminant d'une matrice carrée (N,N), sachant qu'il vaut la somme (sur chaque ligne) de l'élément de la ligne en 1ère colonne par le déterminant de la sous−matrice obtenue en enlevant la ligne et la 1ère colonne (en changeant le signe à chaque fois). Le déterminant d'une matrice (1,1) est sont seul élément. On utilisera bien évidement la récursivité. Il existe (heureusement) d'autres méthodes plus rapides. Cliquez ici pour une solution.

5.10 Structures et unions

Dans un tableau, tous les constituants doivent être du même type. Ce n'est pas le cas des structures, qui sont des variables composées de plusieurs variables (ou CHAMPS) de types différents. Chaque champ n'est plus désigné par un numéro comme dans un tableau, mais par un identificateur.

5.10.1 Déclaration

déclaration : struct nom_type {déclaration des champs} liste_variables ;

exemple :

```
struct identite { char nom[30];
                  char prenom[20];
                  int age;
                }jean,jacques,groupe[20];
```

Toute variable de type identité (jean, jacques, groupe[i]) comporte trois champs : nom, prenom et age. sizeof(jacques) retournera 52. Les champs peuvent être de n'importe quel type valable (scalaires, tableaux, pointeurs...), y compris une structure (à condition d'être déclaré plus haut). nom_type et liste_variables sont optionnels mais au moins l'un des deux doit être présent. Les noms de champs ont une portée limitée à la structure (c'est à dire qu'un autre objet peut avoir le même nom, s'il n'est pas cité dans cette structure). Nom_type (ici identite) est le nom d'un nouveau type, il peut être utilisé plus loin pour déclarer d'autres variables, voire d'autres types:

```
 struct identite lui;
struct prisonnier {long numero; struct identite id;} moi;
```

Il est également possible d'utiliser typedef, et (pas sur tous les compilateurs) d'initialiser une structure :

```
typedef struct {int jour;int mois;int année;}date;
date aujourdhui={24,12,1992}; /*évite de répéter struct*/
```

5.10.2 Utilisation

On accède à une composante par NOM_VARIABLE . NOM_CHAMP , par l'opérateur unaire "."

```
gets(jean.nom);
printf("initiales : %c %c\n",lui.nom[0],lui.prenom[0]);
```

```
printf("nom %s \n",groupe[10].nom);
scanf("%d",&moi.id.age);
```

Une composante d'enregistrement s'utilise comme une variable du même type (avec les mêmes possibilités mais aussi les mêmes limitations). Depuis la norme ANSI, on peut utiliser l'affectation pour des structures (recopie de tous les champs), ainsi que le passage des structures en arguments de fonction passés par valeur. Sur les compilateurs non ANSI, il faut utiliser des pointeurs.

On utilise des pointeurs de structures comme des pointeurs sur n'importe quel autre type. L'opérateur −> permet une simplification d'écriture (il signifie champ pointé) :

```
date *ptr;
ptr=(struct date *)malloc(sizeof(date));
*ptr.jour=14;ptr−>mois=7;
```

5.10.3 Champs de bits

En ne définissant que des champs entiers (signés ou non), on peut définir la taille (en bits) de chaque champ. Il suffit pour cela de préciser, derrière chaque champ, sa taille après un ":".

```
struct état{unsigned ReadOnly:1;int Crc:6;}
```

Les champs sont créés à partir des bits de poids faible. Le nom du champ est optionnel (dans le cas de champs réservés, non utilisés par le programme). Les champs n'ont alors pas d'adresse (impossible d'utiliser & sur un champ). On utilise ces structures comme les autres.

5.10.4 Unions

déclaration : union nom_type {déclaration des champs} liste_variables ;

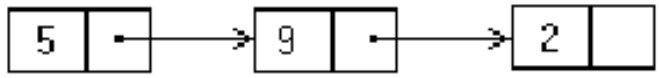
Les différents champs commenceront tous à la même adresse (permet d'utiliser des variables pouvant avoir des types différents au cours du temps, mais un seul à un instant donné). Les champs peuvent être de tout type, y compris structures. On les utilise comme les structures, avec les opérateurs "." et "−>".

Exercice (tel) A l'aide d'un tableau de personnes (nom, prénom, numéro dans la rue, rue, code postal, ville, numéro de téléphone), faire un programme de recherche automatique de toutes les informations sur les personnes répondant à une valeur d'une rubrique donnée (tous les PATRICK , tous ceux d'Obernai, etc...). On suppose que le tableau est déjà initialisé. Cliquez ici pour une solution.

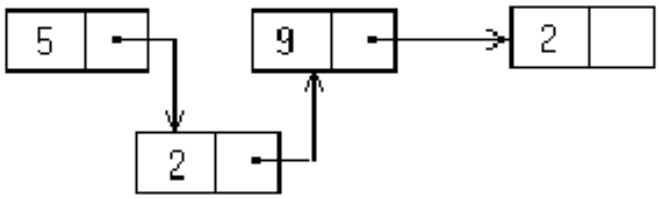
5.10.5 Structures chaînées

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

Appliquons cela , de manière informatique, à une liste d'entiers, avec pour chaque valeur l'adresse (numéro de mémoire) du suivant :



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



En C on définira un type structure regroupant une valeur entière et un pointeur :
`struct page {int val; struct page *suivant; };`

Un pointeur (souvent global) nous indiquera toujours le début de la liste:
`struct page *premier;`

Au fur et à mesure des besoins on se crée une nouvelle page :
`nouveau=(struct page *)malloc(sizeof(struct page));`

En n'oubliant pas de préciser le lien avec le précédent :
`precedent->suivant=nouveau;`

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

exemple :

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <alloc.h> /*ou stdlib.h*/
struct page {int val; struct page *suivant; };
struct page *premier;


int encore(void) /* demande si on en veut encore*/
 {
  printf("encore (O/N) ? ");
  return(toupper(getche())= ='O');
 }
void lecture(void)
 {
  struct page *precedent,*nouveau;
  premier=(struct page *)malloc(sizeof(struct page));
  puts("entrez votre premier entier");
  scanf("%d",&premier->val);
  precedent=premier;
  while (encore())
   {
    nouveau=(struct page *)malloc(sizeof(struct page));
    precedent->suivant=nouveau;
    precedent=nouveau;
    puts("\nentrez votre entier");
    scanf("%d",&nouveau->val);
   }
  precedent->suivant=NULL;
 }
```

```
void affiche(struct page *debut)
 {
  printf("\nliste : ");
  while(debut!=NULL)
   {
    printf("%d ",debut−>val);
    debut=debut−>suivant;
   }
  printf("\n");
 }
void main(void)
 {
  lecture();
  affiche(premier);
 }
```

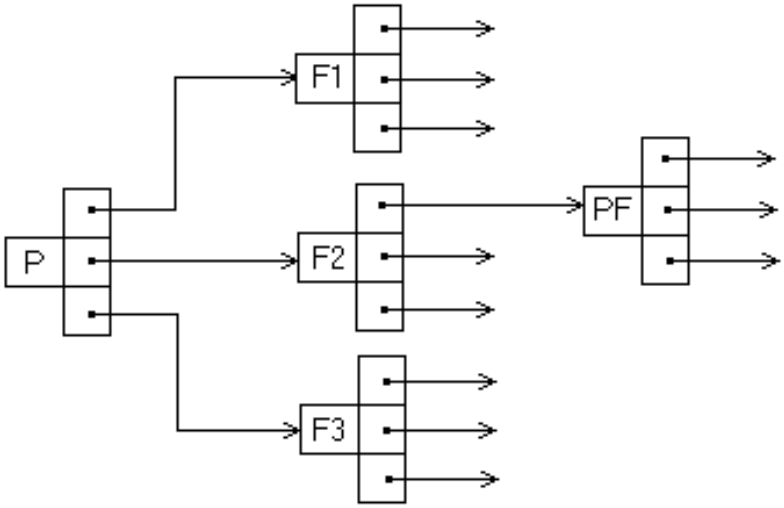
Exercice (liste) : modifier la fonction lecture ci−dessus pour que la liste soit stockée dans l'ordre inverse de son introduction (chaque nouvel élément est placé devant la liste déjà existante).
Cliquez ici pour une solution de cet exercice (et du suivant).

Les modifications sont aisées, une fois que l'on a repéré l'endroit de la modification. Exemple : suppression d'un élément :

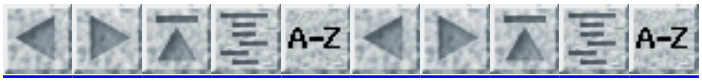
```
void suppression(void)
 {
  struct page *actu,*prec;
  actu=premier;
  while (actu!=NULL)
   {
    printf("\nvaleur : %d − supprimer celui_ci (O/N) ? ",
            actu−>val);
    if (toupper(getche())= ='O')
     {
      if(actu= =premier)premier=actu−>suivant;
      else prec−>suivant=actu−>suivant;
      free(actu);
      break;
     }
    else
     {
      prec=actu;
      actu=prec−>suivant;
     }
   }
 }
```

Exercice (insertion) : ajouter au programme précédent une procédure d'insertion d'une valeur dans la liste.
La solution de l'exercice précédent contient également cette insertion

Ce type de données (structure pointant sur un même type) est utilisé dans d'autres cas. Par exemple, pour représenter un arbre, il suffit pour chaque élément de connaître l'adresse de chaque fils :



Remarque : si le nombre de fils n'est pas constant, on a intérêt à stocker uniquement le fils aîné, ainsi que le frère suivant(voir partie algorithmique et structures de données).



6 Les fichiers de données

Les données stockées en mémoire sont perdues dès la sortie du programme. Les fichiers sur support magnétique (bande, disquette, disque) sont par contre conservables, mais au prix d'un temps d'accès aux données très supérieur. On peut distinguer les fichiers séquentiels (on accède au contenu dans l'ordre du stockage) ou à accès direct (on peut directement accéder à n'importe quel endroit du fichier). Les fichiers sont soit binaires (un float sera stocké comme il est codé en mémoire , d'où gain de place mais incompatibilité entre logiciels), soit formaté ASCII (un float binaire sera transformé en décimal puis on écrira le caractère correspondant à chaque chiffre). Les fichiers étant dépendants du matériel, ils ne sont pas prévus dans la syntaxe du C mais par l'intermédiaire de fonctions spécifiques.

6.1 Fichiers bruts

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

La première opération à effectuer est d'ouvrir le fichier. Ceci consiste à définir le nom du fichier (comment il s'appelle sous le système) et comment on veut l'utiliser. On appelle pour cela la fonction :

```
int open(char *nomfic, int mode);
```

nomfic pointe sur le nom du fichier (pouvant contenir un chemin d'accès). Mode permet de définir comment on utilisera le fichier. On utilise pour cela des constantes définies dans fcntl.h :

O_RDONLY lecture seule, O_WRONLY écriture seule, O_RDWR lecture et écriture. On peut combiner cet accès avec d'autres spécifications, par une opération OU (|) :

O_APPEND positionnement en fin de fichier (permet d'augmenter le fichier), O_CREAT crée le fichier s'il n'existe pas, au lieu de donner une erreur, sans effet s'il existe (rajouter en 3ème argument S_IREAD | S_IWRITE | S_IEXEC déclarés dans stat.h pour être compatible UNIX et créer un fichier lecture/ écriture/ exécution autorisée, seul S_IWRITE utile sur PC), O_TRUNC vide le fichier s'il existait, O_EXCL renvoie une erreur si fichier existant (utilisé avec O_CREAT).

Deux modes spécifiques au PC sont disponibles : O_TEXT change tous les \n en paire CR/LF et inversement, O_BINARY n'effectue aucune transformation.

La fonction rend un entier positif dont on se servira par la suite pour accéder au fichier (HANDLE), ou −1 en cas d'erreur. Dans ce cas, le type d'erreur est donné dans la variable errno, détaillée dans errno.h.

On peut ensuite, suivant le mode d'ouverture, soit lire soit écrire un bloc (l'opération est alors directement effectuée sur disque) :

```
int write(int handle, void *bloc, unsigned taille);
```

On désigne le fichier destination par son handle (celui rendu par open), l'adresse du bloc à écrire et la taille

(en octets) de ce bloc. Le nombre d'octets écrits est retourné, −1 si erreur.

```
int read(int handle, void *bloc, unsigned taille);
```

lit dans le fichier désigné par son handle, et le met dans le bloc dont on donne l'adresse et la taille. La fonction retourne le nombre d'octets lus (<=taille, <si fin du fichier en cours de lecture, 0 si on était déjà sur la fin du fichier, −1 si erreur).

```
int eof(int handle)
```

dit si on se trouve (1) ou non (0) sur la fin du fichier.

Lorsque l'on ne se sert plus du fichier, il faut le fermer (obligatoire pour que le fichier soit utilisable par le système d'exploitation, entre autre mise à jour de sa taille :

```
int close(int handle)
```

fermeture, rend 0 si ok, −1 si erreur.

Le fichier peut être utilisé séquentiellement (le "pointeur de fichier" est toujours placé derrière le bloc que l'on vient de traiter, pour pouvoir traiter le suivant). Pour déplacer le pointeur de fichier en n'importe que autre endroit, on appelle la fonction :

```
long lseek(int handle, long combien, int code);
```

déplace le pointeur de fichier de combien octets, à partir de : début du fichier si code=0, position actuelle si 0, fin du fichier si 2. La fonction retourne la position atteinte (en nb d'octets), −1L si erreur.

```
long filelength(int handle);
```

rend la taille d'un fichier (sans déplacer le pointeur de fichier).

Exemple : copie de fichier (les noms de fichiers sont donnés en argument du programme)

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>
#define taillebloc 1024
int main(int argc,char *argv[])
 {
  int source, destination;
  char buffer[taillebloc];
  int nb_lus,nb_ecrits;
  if (argc!=3) {puts("erreur arguments");return(1);}
  if((source=open(argv[1],O_RDONLY|O_BINARY))<0)
   {puts("erreur ouverture");return(2);}
  if((destination=open(argv[2], O_WRONLY| O_CREAT| O_TRUNC| O_BINARY,
    S_IREAD| S_IWRITE| S_IEXEC))<0)
         {puts("erreur ouverture");return(2);}
  do
   {
    nb_lus=read(source,(char *)buffer,taillebloc);
    if (nb_lus>0) nb_ecrits= write(destination,(char*)buffer, nb_lus);
   }
  while ((nb_lus==taillebloc)&&(nb_ecrits>0));
  close(source);
  close(destination);
  return(0);
```

```
}
```

6.2 Fichiers bufférisés

Les opérations d'entrée / sortie sur ces fichiers se font par l'intermédiaire d'un "buffer" (bloc en mémoire) géré automatiquement. Ceci signifie qu'une instruction d'écriture n'impliquera pas une écriture physique sur le disque mais dans le buffer, avec écriture sur disque uniquement quand le buffer est plein.

Les fichiers sont identifiés non par un entier mais par un pointeur sur une structure FILE (définie par un typedef dans stdio.h). Les fonctions disponibles (prototypées dans stdio.h) sont :

FILE *fopen(char *nomfic, char *mode) : ouvre le fichier, suivant le mode : r (lecture seule), w (écriture, si le fichier existe il est d'abord vidé), a (append : écriture à la suite du contenu actuel, création si inexistant), r+ (lecture et écriture, le fichier doit exister), w+ (lecture et écriture mais effacement au départ du fichier si existant), a+ (lecture et écriture, positionnement en fin de fichier si existant, création sinon). Sur PC, on peut rajouter t ou b au mode pour des fichiers texte (gestion des CR/LF, option par défaut) ou binaires, ou le définir par défaut en donnant à la variable _fmode la valeur O_TEXT ou O_BINARY. fopen rend un identificateur (ID) qui nous servira pour accéder au fichier. En cas d'erreur, le pointeur NULL est retourné, le type d'erreur est donné dans une variable errno, détaillée dans errno.h. La fonction void perror(char *s mess) affichera le message correspondant à l'erreur, en général on lui donne le nom du fichier.

int fread(void *bloc, int taille, int nb, FILE *id) : lit nb éléments dont on donne la taille unitaire en octets, dans le fichier désigné par id, le résultat étant stocké à l'adresse bloc. La fonction rend le nombre d'éléments lus (<nb si fin de fichier), 0 si erreur.

int fwrite(void *bloc, int taille, int nb, FILE *id) : écriture du bloc sur fichier, si le nombre rendu est différent de nb, il y a eu erreur (tester ferror ou errno).

int fclose(FILE *id) : ferme le fichier, en y recopiant le reste du buffer si nécessaire. Cette fonction est obligatoire pour être sur d'avoir l'intégralité des données effectivement transférées sur le disque. Retourne 0 si tout s'est bien passé.

int fflush(FILE *id) : transfère effectivement le reste du buffer sur disque, sans fermer le fichier (à appeler par exemple avant une instruction qui risque de créer un "plantage").

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

int feof(FILE *id) dit si on est en fin de fichier ou non (0).

Les fichiers bufférisés permettent aussi des sorties formatées :

au niveau caractère : **char fgetc(FILE *id)**, **char fputc(char c, FILE *id)**, et même **char ungetc(char c, FILE *id)** qui permet de "reculer" d'un caractère. Cette fonction correspond donc à {fseek(id,−1,1);c=fgetc(id)}.

au niveau chaîne de caractères :

char *fgets(char *s, int max, FILE *id) : lit une chaîne en s'arrêtant au \n ou à max−1 caractères lus, résultat dans la zone pointée par s, et retour du pointeur s ou NULL si erreur.

char fputs(char *s, FILE *id) : écrit la chaîne dans le fichier sans ajouter de \n, rend le dernier caractère écrit ou EOF si erreur.

int fprintf(FILE *id, char *format, listearguments) : rend le nb d'octets écrits, ou EOF si erreur. Les \n sont transformés en CR/LF si fichier en mode texte (spécifique PC).

int fscanf(FILE *id, char *format, listeadresses) : rend le nombre de variables lues et stockées, 0 si erreur).

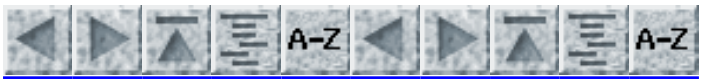
En général, on utilise les fichiers bufférisés :

− Soit en accès direct, en lecture et écriture, avec tous les éléments de même type et même taille (souvent une structure, en format binaire), ceci permettant d'accéder directement à un élément donné (le 48ème, le précédent, l'avant dernier...).

− Soit en accès séquentiel, avec des éléments de type différent, tous formatés sous forme ASCII, en lecture seule ou écriture seule (il est peu probable que le remplacement d'un élément se fasse avec le même nombre d'octets et nécessiterait un décalage dans le fichier), ces fichiers seront compréhensibles par n'importe quel autre programme (éditeur de texte, imprimante...). Un tel fichier s'utilise comme l'écran et le clavier, par des fonctions similaires.

Exercice (agenda) : modifier l'agenda de l'exercice tel en permettant de sauver les données sur disque. On utilisera un fichier binaire à accès direct. On pourra ensuite apporter les améliorations suivantes : recherche rapide par dichotomie sans lire tout le fichier (en le supposant classé par ordre alphabétique), création de fichiers index classés alphabétiquement sur les noms, département et ville pour accès rapide par dichotomie, les autres se faisant par recherche séquentielle, avec possibilité d'ajout, suppression, édition du fichier.

Exercice (fic_formaté) : modifier le programme de produit de matrices en lui permettant de donner sur la ligne de commande trois noms de fichiers : les deux premiers contiendront la description des matrices à multiplier, le dernier sera créé et contiendra le résultat. Les fichiers seront formatés, contiendront en première ligne le nombre de lignes puis le nombre de colonnes, puis chaque ligne du fichier contiendra une ligne de la matrice.



7 Directives du pré−compilateur

Attention, les directives se terminent par le retour à la ligne et pas un ";".

#include <nomfic> permet d'insérer à cet endroit un fichier, qui serra cherché dans le répertoire correspondant à la bibliothèque du C (sur PC dans \TC\INCLUDE)

#include "nomfic" permet d'insérer à cet endroit un fichier, comme s'il était écrit ici, en le cherchant dans le répertoire actuel (le votre)

#define nom valeur : remplace chaque occurrence du nom par la valeur. On ne peut prendre en compte qu'une ligne (le retour à la ligne terminant la définition). En C on a l'habitude de noter les constantes numériques en majuscules.

exemple :

```
#define PI 3.1415926
#define begin {
#define end }
```

#define macro(parametres) définition : permet la définition d'une macro, les paramètres seront substitués lors de la réécriture. Il ne faut pas de blanc entre le nom de la macro et la "(" pour différencier d'un define simple.

`#define double(a) a*2`
remplacera double (i+5) par i+5*2. Il faut donc utiliser des parenthèses, même si elles semblent inutiles :
`#define double(a) (a)*2`

autre exemple:
`#define max(x,y) x>y?x:y`
replacera max(a,max(b,c)) par a>b>c?b:c?a:b>c?b:c

`#define max(x,y) (((x)>(y)?(x):(y)`
replacera max(a,max(b,c)) par (((a)>((((b)>(c))?(b):(c)))) ? (a) : ((((b)>(c)) ? (b) : (c))) ce qui donne des parenthèses superflues mais le résultat escompté.

Une macro ressemble à une fonction, mais sera d'exécution plus rapide : le texte est directement inséré à l'endroit voulu, mais pas de gestion de pile.

#undef nom : annule le précédent #define nom ...

#if (expression) : les lignes qui suivent ne seront lues (et compilées que si l'expression est vraie

#endif : fin de portée du #if précédent

#else : possible mais optionnel, faire attention en cas de #if imbriqués

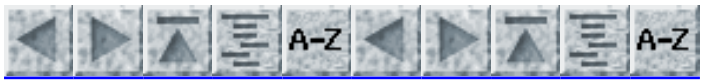
#ifdef nom : comme #if mais vrai s'il y a eu auparavant un #define nom

#ifndef nom : vrai si variable non définie

exemples:

```
#ifndef entier
  #define entier int
```

```
  /* si une autre bibliothèque incluse plus haut l'a
  déjà défini, on ne le redéfinit plus */
#endif
#ifdef biblio_graphique
  initialise_ecran();
  efface_ecran();
  trace(dessin);
#else
  puts("si on avait eu un écran graphique j'aurai fait un dessin");
#endif
/* biblio_graphique peut être défini dans un "header"
   (fichier.h) inclus ou non plus haut */
```



8 Utiliser Turbo C (3.5 par exemple)

Turbo C est d'après moi le compilateur le plus intéressant sur PC en phase d'apprentissage : éditeur intégré, lié aux messages d'erreur, éditeur multifenêtres, aide en ligne, débogueur puissant. Placez vous dans votre répertoire (pas celui de Turbo C pour ne pas mélanger avec vos fichier) et appelez TC.

Les commandes sont accessibles par menus, il vaut mieux posséder une souris. Sinon, on accède à une commande d'un menu par ALT et lettre en surbrillance, on change d'option par TAB, on valide par ESPACE. Au menu principal, on peut :

FILE : gérer les fichiers : en créer un nouveau (new), reprendre un fichier déjà existant (open, puis choisir le fichier dans la liste), sauver le fichier actuel (save ou save as), quitter TC (quit ou ALT X).

EDIT : couper − coller, et surtout remettre une ligne de texte dans son état initial (restore line), surtout après l'avoir effacée par erreur par CTRL Y.

SEARCH : recherche ou remplacement d'un texte dans tout le fichier (CTRL L pour rechercher le suivant).

RUN : exécuter le programme. On peut exécuter le programme pas à pas : Aller jusqu'à la ligne dans laquelle est le curseur, avancer d'une ligne (trace into ou step over). Ceci permet de voir dans quel ordre s'exécutent les instructions.

COMPILE : compiler pour voir les erreurs, il faut faire un EXE pour pouvoir utiliser le programme hors de TC.

DEBUG : en mode pas à pas on peut voir ou modifier l'état d'une variable (evaluate, inspect), voir l'état de la pile (imbrication des fonctions et arguments réels : call stack) voir dans une fenêtre à tout moment l'état d'une variable (add watch).

PROJECT : un projet contient la liste de fichiers nécessaires au fonctionnement d'un programme (fichiers séparés).

OPTION : on peut tout choisir : répertoires, niveau de messages d'erreur de compilation, mode 43 lignes (environnement preferences) et même couleurs.

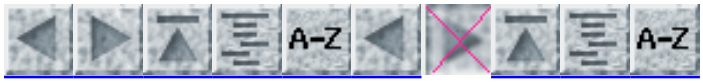
WINDOW : choix des fenêtres ouvertes, déplacement, changement de taille... Entre autres la fenêtre OUTPUT correspond à l'écran tel que le gère votre programme, utile pour voir à la fois vos résultats et votre programme.

HELP : aide

En vous plaçant sur une commande d'un menu, l'appui sur F1 vous détaille ce que permet cette commande. Dans une fenêtre de texte , l'appui sur CTRL F1 vous donne une aide sur le mot sous le curseur, uniquement si c'est un mot clef du C (mais connaît toutes les fonctions, sait dans quel header elle est déclarée). par exemple, tapez printf puis CTRL F1, vous aurez une indication sur tous les formats possibles ainsi que des exemples.

En choisissant l'option menus longs, les touches de raccourci sont indiquées à côté des fonctions des menus.

Dernier conseil, sauvez votre source avant de l'exécuter, au cas ou votre programme planterait la machine (automatisation possible sous options)

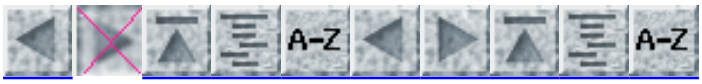

A-Z
A-Z

9 Liens vers d'autres sites sur le C

Je ne suis pas le seul à publier mon cours sur Internet. Voici d'autres liens, pour compléter (ou remplacer) ce que je propose.

- Le cours d'Irène Charon (ENST Paris) : le language est présenté progressivement à partir d'exemples. C'est un très bon complément à qui n'a pas trop aimé ma méthode de présentation (mes étudiants par exemple)
- Le cours de Alain Dancel : Très bon cours, très complet.
- cours ANSI−C de Fred Faber (Lycée Technique des Arts et Métiers, Luxembourg)
- le cours d'Yves Papegay (Inria) (sous Unix)

vous cherchez un compilateur (freeware, évidement) ? DJGPP pour DOS, ou GNU−Win32 pour Win 95/NT, ou GCC pour Unix (c'est le seul que j'ai essayé). De nouveaux arrivants : Pacific : sous DOS (freeware), avec un environnement sympa (un peu comme Turbo C), et une doc en pdf (la version Windows est payante). Désormais, Inprise (Borland) vous propose ses compilateurs **gratuitement !** (pour certaines versions Turbo C, Turbo Pascal, Borland C, Delphi..., évidement pas les plus récentes).



---

10 Correction des exercices

langage C / Patrick TRAU

10.1 1. while_puiss

```
#include <stdio.h>
void main(void)
 {
  int puissance=1,max;
  puts("nombre maximal désiré (ne pas dépasser 16000) ?");
  scanf("%d",&max);
  while (puissance<max) printf("%d\n",puissance*=2);
 }
```

retour au sujet de cet exercice

10.2 2. while_err

Ce programme démontre les erreurs de calcul toujours effectuées sur des nombres réels. On additione successivement 0.01 (qui n'a pas de représentation finie en binaire) à un réel initialement nul. On compte le nombre de calculs jusqu'à obtenir une erreur de 100%. Dans ce cas il faut 16246 calculs. On peut essayer d'autres pas et d'autres débuts.

retour au sujet de cet exercice

10.3 3. do_while

```
#include <stdio.h>
#include <stdlib.h> /* pour rand() */
#include <time.h>   /* pour trouver l'heure pour srand */
void main(void)
 {
  int solution,reponse,nb_essais=0;
```

```
  {time_t t;srand((unsigned) time(&t)); } /* initialiser le générateur
       à partir du compteur de temps, pour qu'il soit plus aléatoire */
  solution=rand()%11;    /* reste sera toujours entre 0 et 10 */
  do
   {
    nb_essais++;
    puts("proposez votre nombre entre 0 et 10");
    scanf("%d",&reponse);
   }
  while (reponse!=solution);
  printf("trouvé en %d essais\n",nb_essais);
 }
```

retour au sujet de cet exercice

10.4 4. for

```
#include <stdio.h>
void main(void)
 {
  int i,N;
  float note,somme=0,moyenne;
  puts("nombre de notes ? ");
  scanf("%d",&N);
  for(i=0;i<N;i++)
   {
    printf("entrez votre %dième note",i+1);
    scanf("%f",&note);
    somme+=note;
   }
  moyenne=somme/N;
  printf("moyenne calculée :%5.2f\n",moyenne);
 }
```

retour au sujet de cet exercice

10.5 5. jeu

```
#include <stdio.h>
#include <stdlib.h> /* pour rand() */
#include <time.h>   /* pour trouver l'heure pour srand */
void main(void)
 {
  int solution,reponse,nb_essais=0;
  { time_t t;srand((unsigned) time(&t)); } /* initialiser le générateur*/
  solution=rand()%11;     /* reste sera toujours entre 0 et 10 */
  do
   {
    nb_essais++;
    puts("proposez votre nombre entre 0 et 10");
    scanf("%d",&reponse);
    if (reponse>solution) puts("trop grand");
    else if (reponse!=solution) puts("trop petit");
   }
  while (reponse!=solution);
  printf("trouvé en %d essais\n",nb_essais);
  if (nb_essais==1) puts("vous avez eu un peu de chance");
  else if (nb_essais<4) puts("bravo");
  else if (nb_essais>6) puts("ce score me semble bien minable");
 }
```

retour au sujet de cet exercice

10.6 6. calcul

```
#include <stdio.h>
void main(void)
 {
  float val1,val2,res;
  char op;
  int fin=0;
  do
   {
    puts("calcul à effectuer (par ex 5*2), ou 1=1 pour finir ? ");
    scanf("%f%c%f",&val1,&op,&val2);
    switch (op)
     {
      case '*':res=val1*val2;break;
      case '/':res=val1/val2;break;
      case '+':res=val1+val2;break;
      case '−':res=val1−val2;break;
      case '=':fin++; /* pas besoin de break, je suis déjà au } */
     }
    if (!fin) printf("%f%c%f=%f\n",val1,op,val2,res);
   }
  while (!fin);
 }
```

retour au sujet de cet exercice

10.7 7. moyenne

```
#include <stdio.h>
#define max 100
typedef float tableau[max];
tableau tab;
int Nb;
void lecture(void)
 {
  int i;
  puts("entrez le nombre de notes à traiter :");
  do scanf("%d",&Nb); while ((Nb<=0)||(Nb>max));
  for(i=0;i<Nb;i++)
   {
    printf("valeur n° %d ? ",i+1);
    scanf("%f",&tab[i]);
   }
 }
float moyenne(void)
 {
  int i;
  float somme=0;
  for (i=0;i<Nb;i++) somme+=tab[i];
  return(somme/Nb);
 }
void affichage(float moy)
 {
  int i;
  printf("la moyenne des %d notes est %f\n",Nb,moy);
  for(i=0;i<Nb;i++) printf("%dième note : écart : %f\n",i+1,tab[i]−moy);
 }
```

```
void main(void)
 {
  lecture();
  affichage(moyenne());
 }
```

retour au sujet de cet exercice

10.8 8. rotation

```
#include <stdio.h>
typedef int composante;
void lecture(composante *t,int *nb)
 {
  int i;
  puts("nombre de valeurs à entrer ? ");
  scanf("%d",nb);
  for(i=1;i<=*nb;i++)
   {
    printf("%dième valeur : ",i);
    scanf("%d",t++);
   }
 }
void affiche(composante *t, int max)
 {
  int i;
  for(i=0;i<max;i++) printf("%d ",*t++);
  puts(" ");
 }
void decale_bas(composante *deb, int max)
 {
  composante c,*t;
  t=deb+(max−1);
  c=*t;
  while (t>deb) {*t=*(t−1);t−−;}
  *t=c;
 }
void decale_haut(composante *t, int nb)
 {
  composante c;int i;
  c=*t;
  for (i=1;i<nb;i++) {*t=*(t+1);t++;}
  *t=c;
 }
void main(void)
 {
  composante tableau[100];
  int nombre;
  lecture(tableau,&nombre);
  puts("tableau initial :");
  affiche(tableau,nombre);
  decale_haut(tableau,nombre);
  puts("décalage vers le haut :");
  affiche(tableau,nombre);
  decale_bas(tableau,nombre);
  puts("décalage vers le bas :");
  affiche(tableau,nombre);
 }
```

retour au sujet de cet exercice

10.9 9. classer

```
#include <stdio.h>
#define dim 100
typedef int composante;
void lecture(composante *t,int *nb)
 {
  int i;
  puts("nombre de valeurs à entrer ? ");
  scanf("%d",nb);
  for(i=1;i<=*nb;i++)
   {
    printf("%dième valeur : ",i);
    scanf("%d",t++);
   }
 }
void affiche(composante *t, int max)
 {
  int i;
  for(i=0;i<max;i++) printf("%d ",*t++);
  puts(" ");
 }
int indice_min(composante t[],int indice_dep, int nb_indices)
/* cherche l'indice de la valeur du tableau :
 *  −soit égale à t[indice_dep], mais d'indice > à indice_dep;
 *  −soit la plus petite mais >t[indice_deb]
 */
 {
  int i,indice_resultat=−1;
  for(i=indice_dep+1;i<nb_indices;i++) if (t[i]==t[indice_dep]) return(i);
/* si on est encore là c'est qu'il n'y en pas d'égal */
  for(i=0;i<nb_indices;i++)
    if ((t[i]>t[indice_dep]) && ((indice_resultat<0) || (t[i]<t[indice_resultat]))) indice_resu
  return(indice_resultat);
}
void copier(composante *source, composante *dest, int nb)
/* copie le tableau source dans le tableau dest */
 {
  int i;
  for(i=0;i<nb;i++) *(dest++)=*(source++);
 }
void classe(composante tab[], int nb)
 {
  composante tempo[dim]; /* un malloc(sizeof(composante)*nb) aurait été
                mieux mais on n'en a pas encore parlé en cours */
  int i,ind_faits,indice;
/* 1er : recherche du plus petit, le 1er si ex aequo */
  indice=0;
  for(i=1;i<nb;i++)
   if(tab[i]<tab[indice]) indice=i;
  tempo[ind_faits=0]=tab[indice];
/* les suivants : recherche le + petit mais > au précédent */
  for(ind_faits=1;ind_faits<nb;ind_faits++)
   {
    indice=indice_min(tab,indice,nb);
    tempo[ind_faits]=tab[indice];
   }
  copier(tempo,tab,nb);
 }
void main(void)
 {
  composante tableau[dim];
  int nombre;
  lecture(tableau,&nombre);
  puts("tableau initial :");
```

```
  affiche(tableau,nombre);
  classe(tableau,nombre);
  puts("tableau classé :");
  affiche(tableau,nombre);
 }
```

retour au sujet de cet exercice

10.10 10. chaînes

```
#include <stdio.h>
#define taille 255
int debut_egal(char *ch,char *sch)
/* répond si sch est exactement le début de ch */
 {
  while (*sch && *ch)
   {
    if (*sch!=*ch) return(0); else {ch++;sch++;}
   }
  return(!(*sch));
 }
void recherche(char *ch,char *sch)
 {
  int pos=0;
  while (*ch)
   {
    if (debut_egal(ch,sch)) printf("trouvé en position %d\n",pos);
    ch++;pos++;
   }
 }
void main(void)
 {
  char ch[taille],sch[taille];
  puts("chaîne à tester ? ");
  gets(ch);
  puts("sous−chaîne à trouver ?");
  gets(sch);
  recherche(ch,sch);
 }
```

retour au sujet de cet exercice

10.11 11. matrices

```
#include <stdio.h>
#define DIM 10
typedef float ligne[DIM];
typedef ligne matrice[DIM];
typedef float *pointeur;

void lecture(matrice t,int *lig,int *col)
/* lit à l'écran une matrice */
 {
  int l,c;
  float f;
  puts("nombre de lignes de la matrice ?");
  scanf("%d",lig);
  puts("nombre de colonnes de la matrice ?");
  scanf("%d",col);
  for (l=0;l<*lig;l++) for(c=0;c<*col;c++)
```

```
   {
    printf("élément [%d,%d] ? ",l,c);
    scanf("%f",&f);
    t[l][c]=f;
   }
 }
void zero(float *t)
/* met toute la matrice à 0 */
 {
  int i;
  for(i=0;i<DIM*DIM;i++) *t++=0;
 }
void unit(matrice t)
/* remplit la matrice unité (1 diagonale, 0 autre) */
 {
  int i;
  zero(t);
  for(i=0;i<DIM;i++) t[i][i]=1;
 }
void init(matrice t)
/* initialisation de matrice à numlig.numcol */
 {
  int i,j;
  for(i=0;i<DIM;i++) for(j=0;j<DIM;j++) t[i][j]=i+j/10.0;
 }
void affiche(matrice t,int l,int c)
/*  puts("affichage du tableau ligne par ligne :"); */
 {
  int i,j;
  for(i=0;i<l;i++)
   {
    printf("ligne %d : ",i);
    for(j=0;j<c;j++) printf("%3.1f ",t[i][j]);
    printf("\n");
   }
 }
void produit(matrice a,matrice b,matrice c,int m,int l,int n)
/* calcul du produit */
 {
  int im,il,in;
  zero(c);
  for(im=0;im<m;im++)
    for(in=0;in<n;in++)
      for(il=0;il<l;il++)
        c[im][in]+=a[im][il]*b[il][in];
 }
void main(void)
 {
  int m,l,n,i;
  matrice a,b,c;
  lecture(a,&m,&l);
  lecture(b,&i,&n);
  if(i!=l) puts("calcul impossible : dimensions incompatibles");
  affiche(a,m,l);
  puts("--- FOIS ---");
  affiche(b,l,n);
  puts("--- FAIT ---");
  produit(a,b,c,m,l,n);
  affiche(c,m,n);
 }
```

retour au sujet de cet exercice

10.12 12. determinant

```
#include <stdio.h>
#include <stdlib.h>
#define DIM 10
typedef float ligne[DIM];
typedef ligne matrice[DIM];
typedef float *pointeur;
long nb_appels;

void lecture(matrice t,int *lig)
/* lit une matrice (au clavier) */
 {
  int l,c;
  float f;
  puts("dimension de la matrice ?");
  scanf("%d",lig);
  for (l=0;l<*lig;l++) for(c=0;c<*lig;c++)
   {
    printf("élément [%d,%d] ? ",l,c);
    scanf("%f",&f);
    t[l][c]=f;
   }
 }
void zero(matrice t,int dim)
/* met toute la matrice à 0 */
 {
  int i,j;
  for(i=0;i<dim;i++)for(j=0;j<dim;j++) t[i][j]=0;
 }
void unit(matrice t, int dim)
/* remplit la matrice unité (1 diagonale, 0 autre) */
 {
  int i;
  zero(t,dim);
  for(i=0;i<dim;i++) t[i][i]=1;
 }
void affiche(matrice t,int l)
/*  puts("affichage du tableau ligne par ligne :"); */
 {
  int i,j;
  for(i=0;i<l;i++)
   {
    printf("ligne %d : ",i);
    for(j=0;j<l;j++) printf("%3.1f ",t[i][j]);
    printf("\n");
   }
 }
void copiesauflc(matrice source,matrice dest,int dim,int ligavirer)
 {
  int l,c,ld=0;
  for (l=0;l<dim;l++) if (l!=ligavirer)
   {
    for (c=1;c<dim;c++) dest[ld][c-1]=source[l][c];
    ld++;
   }
 }
float determinant(matrice m,int dim)
 {
  matrice sous_m;
  int l,signe=1;
  float det=0;
  nb_appels++;
  if (dim==1) return(m[0][0]);
  for(l=0;l<dim;l++)
```

```
   {
    copiesauflc(m,sous_m,dim,l);
    det+=signe*m[l][0]*determinant(sous_m,dim−1);
    signe=−signe;
   }
  return(det);
 }
void produit(matrice a,matrice b,matrice c,int dim)
/* calcul du produit */
 {
  int im,il,in;
  zero(c,dim);
  for(im=0;im<dim;im++)
    for(in=0;in<dim;in++)
      for(il=0;il<dim;il++)
        c[im][in]+=a[im][il]*b[il][in];
 }
void main(int argc,char *argv[])
 {
  int taille;
  matrice mat;
/*  lecture(mat,&taille);  */
      taille=atoi(argv[1]); /* test avec matrice unité, */
      unit(mat,taille);     /* au moins je connais le résultat*/
  affiche(mat,taille);
  printf("déterminant : %20.17f ",determinant(mat,taille));
  printf(" en %ld appels\n",nb_appels);
 }
```

retour au sujet de cet exercice

10.13 13. tel

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define DIM 100
enum champs {nom,prenom,num,rue,cp,ville,tel};
char *nomchamp[7]={"Nom", "Prénom", "Numéro", "Rue",
                   "Code Postal", "Ville", "Tel"};
typedef struct
 {
  char    nom[15];
  char    prenom[20];
  int     num;
  char    rue[60];
  long    codepostal;
  char    ville[20];
  char    tel[15];
 } fiche;

fiche *rech_nom(fiche *,char *);
fiche *rech_prenom(fiche *,char *);
fiche *rech_num(fiche *,char *);
fiche *rech_rue(fiche *,char *);
fiche *rech_cp(fiche *,char *);
fiche *rech_ville(fiche *,char *);
fiche *rech_tel(fiche *,char *);
fiche *rech_nom(fiche *,char *);

typedef fiche *ptrfiche;
typedef ptrfiche (*ptrfonction)(ptrfiche,char*);
ptrfonction tabfonction[7]={rech_nom, rech_prenom, rech_num,
```

```
                  rech_rue, rech_cp, rech_ville, rech_tel};

void affiche(fiche *f)
 {
  if(f->nom[0])
    printf("%s %s\n%d, %s\n%ld %s\nTel : %s\n", f->nom, f->prenom,
           f->num, f->rue, f->codepostal, f->ville, f->tel);
  else
    printf("fiche inconnue\n");
 }
int idem(char *s1,char *s2)
/* compare deux chaines, dit si elles sont égales (1), ou non (0). On
   pourrait supposer égalité quand la chaine s2 est incluse dans s1 */
 {
  return(strcmp(s1,s2)?0:1);
 }

fiche *rech_nom(fiche *pf,char *n)
 {while ((pf->nom[0])&&(!idem(pf->nom,n)))pf++;
  return(pf);}
fiche *rech_prenom(fiche *pf,char *n)
 {while ((pf->nom[0])&&(!idem(pf->prenom,n)))pf++;
  return(pf);}
fiche *rech_num(fiche *pf,char *n)
 {while ((pf->nom[0])&&(pf->num!=atoi(n)))pf++;
  return(pf);}
fiche *rech_rue(fiche *pf,char *n)
 {while ((pf->nom[0])&&(!idem(pf->rue,n)))pf++;
  return(pf);}
fiche *rech_cp(fiche *pf,char *n)
 {while ((pf->nom[0])&&(pf->codepostal!=atoi(n)))pf++;
  return(pf);}
fiche *rech_ville(fiche *pf,char *n)
 {while ((pf->nom[0])&&(!idem(pf->ville,n)))pf++;
  return(pf);}
fiche *rech_tel(fiche *pf,char *n)
 {while ((pf->nom[0])&&(!idem(pf->tel,n)))pf++;
  return(pf);}

int choix(void)
 {
  char lig[40];
  enum champs i,rep;
  for (i=nom;i<=tel;i++) printf("%d:%s ",i,nomchamp[i]);
  printf("\nou −1 pour quitter. Type de recherche désirée ? ");
  gets(lig);
  sscanf(lig,"%d",&rep);
  return(rep);
 }

void lecture(fiche *tab)
 {
  char lig[40];
  do
   {
    printf("nom (rien pour finir) ?");
    gets(tab->nom);
    if(tab->nom[0])
     {
      printf("         prénom ? ");
      gets(tab->prenom);
      printf("             N° ? ");
      gets(lig);
      sscanf(lig,"%d",&(tab->num));
      printf("            rue ? ");
```

```
      gets(tab->rue);
      printf("    code postal ? ");
      gets(lig);
      sscanf(lig,"%ld",&(tab->codepostal));
      printf("          ville ? ");
      gets(tab->ville);
      printf("n° de téléphone ? ");
      gets(tab->tel);
     }
   }
  while ((tab++)->nom[0]);
 }

void main(void)
 {
  enum champs c;
  char clef[40];
  fiche tab[DIM];
  lecture(tab);
  do
   {
    if (((c=choix())<0)||(c>6)) break;
    printf("quel(le) %s recherche-t'on ? ",nomchamp[c]);
    gets(clef);
    affiche(tabfonction[c](tab,clef));
   }
  while (c>=0);
 }
```

retour au sujet de cet exercice

10.14 14. liste et insertion

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <alloc.h>

struct page {int val; struct page *suivant; };
struct page *premier;

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

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

```
 }

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

void suppression(void)
 {
  struct page *actu,*prec;
  actu=premier;
  while (actu!=NULL)
   {
    printf("\nvaleur : %d − supprimer celui−ci (O/N) ? ",actu−>val);
    if (toupper(getche())=='O')
     {
      if(actu==premier)premier=actu−>suivant;
      else prec−>suivant=actu−>suivant;
      free(actu);
      break;
     }
    else
     {
      prec=actu;
      actu=prec−>suivant;
     }
   }
 }
void ajouter(void)
 {
  struct page *nouveau,*prec;
  printf("\najouter en premier (O/N) ? ");
  if (toupper(getche())=='O')
   {
    nouveau=(struct page *)malloc(sizeof(struct page));
    nouveau−>suivant=premier;
    premier=nouveau;
    printf("\nnouvelle valeur ? ");
    scanf("%d",&(nouveau−>val));
   }
  else
   {
    prec=premier;
    while(prec!=NULL)
     {
      printf("\nvaleur : %d − insérer après celui_ci (O/N) ? ", prec−>val);
      if (toupper(getche())=='O')
       {
        nouveau=(struct page *)malloc(sizeof(struct page));
        nouveau−>suivant=prec−>suivant;
        prec−>suivant=nouveau;
        printf("\nnouvelle valeur ? ");
        scanf("%d",&(nouveau−>val));
        break;
       }
      else prec=prec−>suivant;
     }
   }
 }
```

```
void main(void)
 {
  lecture();
  affiche(premier);
  do
   {
    suppression();
    affiche(premier);
   }
  while(encore());
  do
   {
    ajouter();
    affiche(premier);
   }
  while(encore());
 }
```

retour au sujet de cet exercice

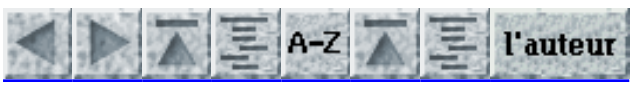
10.15 15. agenda

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

/* définitions des types et variables associées */
enum champs {nom,prenom,num,rue,cp,ville,tel};
char *nomchamp[7]={"Nom","Prénom","Numéro","Rue",
                   "Code Postal","Ville","Tel"};
typedef struct
 {
  char    nom[15];
  char    prenom[20];
  int     num;
  char    rue[60];
  long    codepostal;
  char    ville[20];
  char    tel[15];
 } fiche;
#define taille sizeof(fiche)
typedef fiche *ptrfiche;

/* définitions des fonctions de recherche, regroupées dans un tableau */
fiche *rech_nom(fiche *,char *);
fiche *rech_prenom(fiche *,char *);
fiche *rech_num(fiche *,char *);
fiche *rech_rue(fiche *,char *);
fiche *rech_cp(fiche *,char *);
fiche *rech_ville(fiche *,char *);
fiche *rech_tel(fiche *,char *);
fiche *rech_nom(fiche *,char *);

typedef ptrfiche (*ptrfonction)(ptrfiche,char*);
ptrfonction tabfonction[7]= {rech_nom, rech_prenom, rech_num,
                     rech_rue, rech_cp, rech_ville, rech_tel};

/* variables globales */
FILE *fic;  /* fichier de données */
char *nomfic="agenda.dat";
```

```
int nb;    /* nb de fiches dans le fichier */

void init(void)
/* ouvre le fichier, détermine le nb de fiches de fic */
 {
  if ((fic=fopen(nomfic,"a+b"))==NULL)
   {
    puts("ouverture impossible du fichier de données");
    exit(1);
   }
  fseek(fic,0,2);
  nb=(int)ftell(fic)/taille;
  printf("%d fiches présentes dans l'agenda\n",nb);
 }

void ajouter(void)
 {
  char lig[40];
  fiche f;
  printf("nom             ? ");
  gets(f.nom);
  printf("         prénom ? ");
  gets(f.prenom);
  printf("         Numero ? ");
  gets(lig);
  sscanf(lig,"%d",&(f.num));
  printf("            rue ? ");
  gets(f.rue);
  printf("    code postal ? ");
  gets(lig);
  sscanf(lig,"%ld",&(f.codepostal));
  printf("          ville ? ");
  gets(f.ville);
  printf("n° de téléphone ? ");
  gets(f.tel);
  fseek(fic,0L,2);
  if(fwrite(&f,taille,1,fic)!=1)
   {
    puts("impossible d'ajouter cette fiche au fichier ");
    exit(0);
   }
  nb++;
 }

void affiche(fiche *f)
 {
  if((f!=NULL)&&(f->nom[0]))
    printf("%s %s\n%d, %s\n%ld %s\nTel : %s\n",f->nom, f-> prenom,
              f->num, f->rue,f->codepostal,f->ville,f->tel);
  else
    printf("fiche inconnue\n");
 }

int idem(char *s1,char *s2)
/* compare deux chaines, dit si elles sont égales (1), ou non (0).
   On considère égales majuscules et minuscules.
   Une des chaines peut se terminer par *, on supposera identique
   si tout ce qui précède l'* était identique */
 {
  for(;;)
   {
    if (((!*s1)&&(!*s2))||(*s1=='*')||(*s2=='*')) return(1);
    if ((toupper(*s1)!=toupper(*s2))||(!*s1)||(!*s2)) return(0);
    s1++;s2++;
   }
```

```
 }

fiche *rech_nom(fiche *pf,char *n)
 {
  int nblu;
  fseek(fic,0L,0);
  do
    nblu=fread(pf,taille,1,fic);
  while ((nblu==1)&&(!idem(pf->nom,n)));
  if (nblu==1) return(pf); else return(NULL);
 }

/* les autres recherches sont à continuer */

int choix(void)
 {
  char lig[40];
  enum champs i,rep;
  for (i=nom;i<=tel;i++) printf("%d:%s ",i,nomchamp[i]);
  printf("\nou −1 pour quitter. Type de recherche désirée ? ");
  gets(lig);
  sscanf(lig,"%d",&rep);
  return(rep);
 }

void recherche(void)
 {
  enum champs c;
  char clef[40];
  fiche f;
  do
   {
    if (((c=choix())<0)||(c>6)) break;
    printf("quel(le) %s recherche−t'on ? ",nomchamp[c]);
    gets(clef);
    affiche(tabfonction[c](&f,clef));
   }
  while (c>=0);
 }

void main(void)
 {
  char rep;
  init();
  do
   {
    puts("Ajouter une fiche, Recherche d'une fiche, Quitter le prog ? ");
    switch (rep=toupper(getch()))
     {
      case 'A':ajouter();break;
      case 'R':recherche();break;
      case 'Q':fclose(fic);puts("Au revoir");break;
      default :puts("option non prévue");
     }
   }
  while (rep!='Q');
 }
```

retour au sujet de cet exercice



11 Langage C − Index

11.1 A

11.2 B

11.3 C

11.4 D

11.5 E

11.6 F

11.7 G

11.8 H

11.9 I

11.10 K

11.11 L

11.12 M

11.13 N

11.14 O

11.15 P

11.16 Q

11.17 R

11.18 S

11.19 T

11.20 U

11.21 V

11.22 W

- while [1]
- write [1]