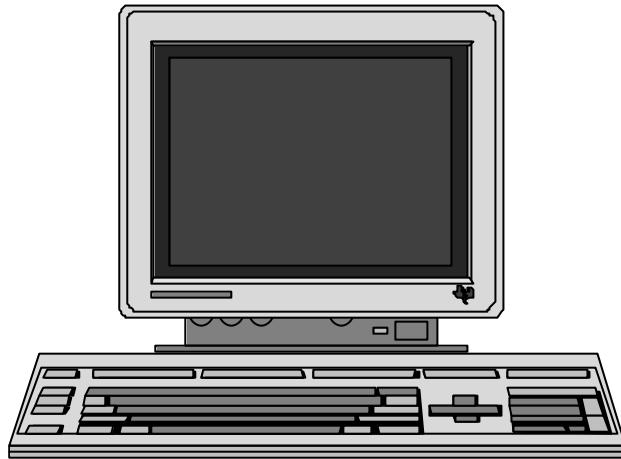


UNIVERSITE de CERGY

Programmation en Turbo C



José GILLES - 1995

IUP GENIE ELECTRIQUE

Mes sincères remerciements à Fabienne PRUVOST

qui a contribué à la préparation de ce polycopié.

Merci d'avance à tous ceux qui voudront bien
me faire part de leurs remarques et suggestions
visant à améliorer cette seconde édition.

D) Introduction

- 1) Les menus principaux de Turbo C

II) Description du langage C

III) Les ingrédients de base d'un programme C

- 1) Les directives de compilation de base
 - a) #include
 - b) #define
- 2) Les variables et types de base
 - a) Systèmes de numération
 - b) Les types entiers
 - c) Les types flottants
 - d) Les types caractères
 - e) Les chaînes de caractères
 - f) Les différentes classes de variables
 - g) Les pointeurs
 - h) Définition d'un nouveau type
- 3) Les tableaux de variables
- 4) Les structures
- 5) Les boucles et sauts
 - a) Les boucles
 - b) Les tests et sauts conditionnels
 - c) Les choix multiples
- 6) Les fonctions
 - a) Description
 - b) Passage de paramètres
 - c) Retour de valeurs

IV) Les fonctions d'entrée / sortie

- 1) printf() et scanf()
- 2) Fonctions de saisie et affichage de caractères et de chaînes
 - a) putchar() et getchar()
 - b) puts() et gets()

V) Fonctions de manipulation de chaînes de caractères

- 1) Recopier une chaîne dans une autre
- 2) Concaténer deux chaînes
- 3) Comparaison de deux chaînes
- 4) Longueur d'une chaîne
- 5) Recherche de caractères dans une chaîne
- 6) Conversion des caractères
- 7) Conversion de nombres en chaînes et vice versa

VI) Le mode texte

- 1) Le fenêtrage et la gestion du curseur
- 2) Saisie et affichage
- 3) Gestion des couleurs et attributs
- 4) Sauvegarde et restitution de fenêtre

VII) Le mode graphique

- 1) Le principe du mode graphique
- 2) L'initialisation du mode graphique
- 3) Fenêtrage et gestion du curseur
- 4) Gestion des couleurs
- 5) Les tracés
 - a) Tracés de lignes
 - b) Tracés d'ellipses et cercles
 - c) Tracés de rectangles et polygones
- 6) Le remplissage
 - a) Remplissage de cercles et ellipses
 - b) Remplissage de polygones
 - c) Remplissage d'un contour
 - d) Statistiques
- 7) Affichage de texte en mode graphique
- 8) Sauvegarde et restauration d'une partie de l'écran

VIII) Fonctions de manipulation de fichiers et répertoires

- 1) Opérations sur les fichiers
 - a) Les fonctions de bas niveau
 - b) Les fonctions de haut niveau
- 2) Les fonctions de manipulation des fichiers
- 3) Les fonctions de manipulation des répertoires
 - a) Opérations sur les répertoires et unités
 - b) Recherche d'un fichier dans un répertoire

IX) Variables dynamiques

- 1) Les principes
- 2) La programmation

X) Gestion des processus

- 1) Notion de processus
- 2) La programmation

XI) Gestion de projets en Turbo C

- 1) Création d'un fichier de projet
- 2) Contenu d'un projet
- 3) Compilation et linkage d'un projet

PROGRAMMATION EN C

IUP GE 1ERE ET 2EME ANNEE

Matériel requis: Un PC fonctionnant sous DOS
Environnement: TURBO C ou BORLAND C

D) Introduction:

On suppose que le lecteur est familier du DOS et des PC et que le logiciel TURBO C ou BORLAND C a été correctement installé sur le PC.

Cet environnement de développement comporte des modules fonctionnant en ligne de commande et des modules intégrés dans un environnement piloté par menus à la souris. Nous n'étudierons que ce second aspect, l'autre étant réservé à certains usages particuliers.

1) Les menus principaux du TURBO C:

Le lancement du TURBO C (respectivement BORLAND C) se fait par la commande TC ↵ (respectivement BC ↵) sous DOS.

On arrive alors dans un menu principal comme ci-contre:

Voyons dans l'immédiat les options indispensables.
D'autres seront abordées plus loin dans ce poly.

Menu file: Accessible d'au moins trois façons:
- cliquer sur FILE avec la souris.
- faire ALT_F au clavier.
- se déplacer dessus avec les flèches et faire ↵ au clavier.

Les options utiles dans l'immédiat sont:

NEW: Permet d'ouvrir un nouveau fichier dans lequel nous ne
tarderons pas à taper notre premier programme en C.

SAVE: Permet de sauvegarder le programme tapé dans la fenêtre
fichier.

SAVE AS: Idem en donnant au fichier un nouveau nom.

OPEN: Permet de rouvrir un fichier précédemment sauvegardé.

CHANGE DIR: Permet de changer le répertoire par défaut dans lequel sont
sauvegardés les fichiers et dans lequel sont recherchés
les fichiers à ouvrir.

DOS SHELL: Permet de revenir temporairement sous dos pour effectuer
telle ou telle manipulation. On revient dans le TURBO C
en faisant EXIT ↵ dans la session DOS.

QUIT: Pour quitter totalement le logiciel.

Menu compile: Permet comme nous le verrons bientôt la compilation du
programme tapé dans la fenêtre fichier.

Les options vitales sont dans l'immédiat:

COMPILE: (ALT_F9)
Compile le programme source C en un équivalent exécutable
dans l'environnement TURBO C.

(Voir menu RUN ci-après).

MAKE: (F9)
Compile le programme C en un programme EXE qui sera
exécutable directement sous DOS sans nécessiter le
TURBO C.

Menu run: Permet l'exécution d'un programme compilé.
(Voir menu COMPILE ci-dessus)

La commande utile dans l'immédiat est:

RUN: (CTRL_F9)
Exécute immédiatement le programme de la fenêtre fichier.

Menu Help: Procure une aide très utile sur les mots clés, les fonctions,
les éléments pré-définis du TURBO C.

Deux options sont particulièrement utiles:

INDEX: Accès à l'index des mots clés reconnus par l'aide.

TOPIC SEARCH: S'utilise en enluminant préalablement à la souris
(bouton appuyé) ou au clavier (SHIFT + FLECHE)
un mot clé dont l'aide associée apparaîtra directement.

II) Description du langage C:

Chers lecteurs , je vous fais grâce de la sempiternelle introduction sur la naissance du langage C que vous trouverez dans tout bon ouvrage sur le sujet !

Dans ce qui suit les programmes en C seront présentés encadrés afin de bien les distinguer du texte.

☛ Attention: Le langage C est pointilleux sur les minuscules et majuscules.

Le but d'un programme classique en C est en général de saisir des données (nombres , chaînes de caractères , fichier) , de les soumettre à des fonctions dont le rôle est de modifier ces données , d'effectuer des calculs , et enfin de retourner des résultats (via l'écran , l'imprimante ...).

Pour accomplir cela le programmeur dispose de variables de différents types qui seront détaillés plus loin (entiers, nombres en virgule flottante, caractères, chaînes de caractères ...) , de constantes à définir si nécessaire, comme $PI = 3.1416$, de fonctions pré-définies comme `printf()` qui permet d'afficher un résultat à l'écran, de fonctions à définir qui utiliseront les variables, constantes, fonctions pré-définies, d'opérateurs de calcul pour réaliser les opérations souhaitées (si tout va bien !), de boucles, d'instructions de saut conditionnel ou inconditionnel.

Les fonctions pré-définies — il y en a des centaines — sont réparties en de nombreuses catégories telles que :

- fonctions de saisie / affichage
- fonctions de manipulation de chaînes de caractères
- fonctions de manipulation de fichiers
- fonctions graphiques
- ...

La description formelle de chaque catégorie de fonctions est regroupée dans un ou plusieurs headers: il s'agit de fichiers (obscurs pour les débutants !) d'extension `.h` qui doivent être inclus dans le source C si l'on veut appeler les fonctions de cette catégorie. (voir § Fonctions).

Les fonctions définies par l'utilisateur doivent faire l'objet en début de programme d'une déclaration personnalisée.

Toutes les variables du programme doivent être déclarées également.

Un programme C commence toujours son exécution au début de la fonction pré-définie `main()` et il se termine en général à la fin du `main()`

Les commentaires qui sont toujours bienvenus doivent être encadrés par `/*` et `*/` .

Ceci nous permet de mettre en évidence la structure générale d'une source C élémentaire:

```
#include <stdio.h>          /* directive d'inclusion pour printf() et scanf()*/
#define PI 3.14             /* déclaration de constante */

float surface(float);       /* déclaration du prototype de la fonction surface()
                             qui reçoit un paramètre float -- le rayon --
                             et retourne un résultat float -- la surface -- */

float rayon, surf;         /* déclaration de variables globales */

float surface(float r)     /* définition de la fonction surface() */
{
    float s;               /* variable locale -- temporaire -- */

    s= PI*r*r;            /* calcul de la surface du cercle */
    return(s);            /* retourne à la fin le résultat calculé */
}

main()
{
    printf( "Donner le rayon du cercle:" );      /* affiche message */
    scanf( "%f", &rayon);                       /* saisie du rayon */

    surf= surface(rayon); /* appel de la fonction de calcul de surface */

    printf( "La surface du cercle est: %f", surf ); /* affiche résultat */
}
```

Pour saisir ce programme, ouvrir une fenêtre avec file → new

faire F5 pour agrandir la fenêtre et taper le texte scrupuleusement.

Sauver alors le texte sous le nom ESSAI.C avec file → save as

Pour compiler ce programme, faire ensuite compile → compile

En cas de message d'erreur, noter la ligne incriminée et vérifier ce qui a été tapé.
Reprendre alors la compilation.

Enfin, pour exécuter le programme faire run → run

Saisir une valeur et guetter le résultat.

Attention: Une fois le programme achevé on revient dans le menu.
Aussi pour visualiser les résultats du programme faut-il faire ALT_F5.

Au delà du résultat sans surprise, le code source de ce premier programme appelle quelques commentaires:

Le début du programme est constitué de déclarations,
la suite étant constituée d'instructions qui seules engendrent des actions et produisent des résultats.

Remarquons qu'un bloc d'instructions est délimité par des accolades { }.
Elles délimitent ici les fonctions surface() et main().
Chaque instruction est terminée par un ; qui est indispensable en C.
Par contre les déclaration en # ne nécessitent pas de ; final.

L'exécution du programme commence au main() et correspond à l'organigramme suivant:
(Prière de faire les organigrammes d'abord à l'avenir !).

Remarque: Respectez dans les sources C les décalages (tabulations) qui assureront plus tard la lisibilité de vos sources.

III) Les ingrédients de base d'un programme C:

1) Les directives de compilation de base:

a) #include

Dans l'immédiat il sera nécessaire d'inclure systématiquement le fichier `<stdio.h>` qui contient la déclaration des fonctions de saisie / affichage telles que `scanf()` et `printf()`

D'autres s'y ajouteront par la suite.

Remarque: Dans l'aide associée à chaque fonction pré-définie — essayer avec `printf()` — sont indiqués les headers qui doivent être inclus.

b) #define

Servira surtout à définir des constantes entières, flottantes, caractères ou chaînes, par exemple:

```
#define MAX 65535
#define PI 3.14 /* ne pas mettre = ni ; */
#define NUL 0 /* caractère NUL de code ascii 0 */
#define PROGRAMMEUR "J. GILLES".
```

Bien que cela ne soit pas imposé, il est un usage que l'on respectera qui consiste à toujours déclarer les constantes avec des noms en MAJUSCULES, afin de les distinguer des variables.

2) Les variables — types de base:

a) Systèmes de numération

La numération standard se fait en base 10, mais il est possible de donner des valeurs directement en hexadécimal — base 16 — en utilisant les chiffres 0, ..., 9, A, B, C, D, E, F. Une valeur hexa commence par 0x en langage C.

Exemples: 0x1B correspond à 27 en décimal
0x1B3F correspond à 6975 en décimal

Un autre système de numération, plutôt obsolète, est encore disponible: la numération en base 8 avec les chiffres 0 à 7.

Une valeur octale commence par 0 en langage C.

Exemple: 0215 correspond à 141 en décimal.

b) Les types entiers

L'entier de base est de type `int`. On le déclare par `int i;`

On peut en déclarer plusieurs d'un coup: `int i,j;`

On peut aussi initialiser une ou plusieurs variables lors de la déclaration: `int i,j=0;`

(Ces remarques restent valables pour tous les types qui suivent).

Une variable `int` est un entier signé codé sur 16 bits; en hexadécimal les valeurs codées sur

16 bits comprennent 4 digits et vont de `0x0000` à `0xFFFF`.

La plage `0x0000` à `0x7FFF` — dont le premier bit est à 0, bit de signe — correspond

aux entiers positifs de 0 à + 32767.

La plage `0x8000` à `0xFFFF` — dont le premier bit est à 1 — correspond aux entiers négatifs de -32768 à -1.

Cette plage étant un peu réduite, il existe un type `long int` (ou `long` tout court)

Exemples: `long l;`
`long int k;`

Les entiers longs sont codés sur 32 bits soit 8 digits hexadécimaux avec un bit de signe en tête ce qui autorise des valeurs de -2 147 483 648 à +2 147 483 647.

Les entiers nécessaires n'étant pas toujours signés, il existe deux types d'entiers positifs, les `unsigned int` (ou `unsigned`) et `unsigned long`.

Exemples: `unsigned u;`
`unsigned int v;`
`unsigned long l;`

Dans ce cas les plages de valeurs commencent à 0 et vont respectivement jusqu'à 65535

(`0xFFFF`) et 4.294.967.295 (`0xFFFFFFFF`)

résumé: On choisit le type entier voulu en fonction de la plage de valeurs dans laquelle la variable est censée évoluer:

type entier	valeur minimale	valeur maximale
<code>int</code>	-32 768	+32 767
<code>long (int)</code>	-2 147 483 648	+2 147 483 647
<code>unsigned (int)</code>	0	+65 535
<code>unsigned long</code>	0	+4 294 967 295

Les opérateurs sur entiers


```

if ( condition )
    {
    instructions
    }
else
    {
    instructions
    }

```

Exemples:

```

int i;
...
if ( i == 0 ) printf ( "c'est nul" );
...

```

Si $i = 0$ le programme affiche le message et continue à la ligne suivante.
Sinon, il passe directement à la ligne suivante.

```

int i;
...
if ( i == 0 )
    {
    printf( "c'est nul" );
    ...
    }
else {
    printf( "c'est bien" );
    ...
    }

```

Dans ce cas un bloc d'instruction est exécuté si $i = 0$ et un autre bloc est exécuté si $i \neq 0$

☛ Attention: Une erreur très courante consiste à écrire `if (i = 0) ...`
Ce qui ne fonctionnera pas, bien que la compilation ne dise rien parfois.

On respectera les décalages (tabulations) lors de test associés à des blocs pour améliorer la lisibilité du programme.

Opérateurs binaires

Ces opérateurs jouent sur les bits des données comme leur nom l'indique.

&	Et logique bit à bit (and)
	Ou logique bit à bit (or)
^	Ou exclusif bit à bit (xor)

>> Décalage à droite (shift)
 Décale les bits de plusieurs positions vers la droite

<< Décalage à gauche (shift)
 Idem à gauche.

~ Négation binaire.
 Les bits à 0 passent à 1 et vice versa.

Exemples:

```
unsigned int i, x, y, z;
...
x= i & 0x 00FF; /* isole dans x les bits de poids faible de i */
y= y | 0x 8000; /* force le 1er bit de y à 1 */
z= i >> 8; /* décale i de 8 bits vers la droite dans z */
```

Remarques: On peut aussi écrire au lieu de $a = a \ll 5$; l'équivalent $a \ll= 5$;
 et même chose pour $\gg=$, $\&=$, $|=$, $\wedge=$.
 Les parenthèses permettent de forcer l'évaluation des expressions dans l'ordre voulu.

Exemples:

```
int i, j, a, b;
...
a= ( i+ j ) * 2;
b= ( i << 8 ) | ( j & 0x 00FF );
```

c) Les types flottants

Un nombre en virgule flottante est utilisé pour représenter un nombre à virgule
 comme 3.14 ou un nombre en notation scientifique tel que $6.023 \cdot 10^{-23}$
 Le type de base associé est le float.

Exemple:

```
float f, g;
...
f= 2.71;
g= 1.5e-12; /* notation scientifique sans espace */
```

Un float est stocké sur 32 bits en interne et permet des valeurs de $\pm 3.4 \cdot 10^{-38}$ à $\pm 3.4 \cdot 10^{+38}$.

Si cette plage ne suffit pas on peut envisager le type double float (déclaré double) stocké

sur 64 bits et qui autorise des valeurs de $\pm 1.7 \cdot 10^{-308}$ à $\pm 1.7 \cdot 10^{+308}$

Enfin le type long double sur 80 bits étend la plage de $\pm 3.4 \cdot 10^{-4932}$ à $\pm 1.1 \cdot 10^{+4932}$

Opérateurs sur les flottants

De nombreux opérateurs définis sur les entiers n'auraient pas de sens appliqués à des nombres en virgule flottante.

Il reste les opérateurs suivants:

Opérateurs mathématiques

+ , - , * , / Addition, soustraction, multiplication, division exacte.

Les opérateurs += , -= , *= , /= sont encore disponibles.

Opérateurs de comparaison

= , != Egalité, non égalité
< , <= , > , >= Inférieur (ou égal), supérieur (ou égal)

Ils sont utilisés dans des tests similaires à ceux décrits pour les entiers.

d) Les types caractères

Il est souvent utile de déclarer des variables de type caractère nommées char.

Exemple:

```
char c;  
...  
c= 'A';
```

Un char occupe 1 octet (8 bits) et prend donc des valeurs de -128 à +127.
Le type dérivé unsigned char correspond à la plage 0 à 255.

En pratique, on ne calcule guère avec les caractères, aussi il est assez indifférent de déclarer char ou unsigned char. La représentation interne étant la même de 0x00 à 0xFF .

On peut affecter une variable caractère de diverses manières:

```
char c;
...
c= 'A';      /* affectation de la lettre A */
c= 27;       /* code ascii décimal de ESC */
c= 0x 1B;    /* code ascii hexa de ESC */
```

Toutefois certains caractères ne sont pas accessibles au clavier et sont pourtant très utiles;

ils disposent alors d'une séquence d'échappement \ suivi d'une lettre:

\n	ascii 10	(newline)	retour à la ligne
\r	ascii 13	(return)	retour en début de ligne
\b	ascii 8	(backspace)	retour d'un caractère en arrière
\t	ascii 9	(tab)	avance d'une tabulation
\a	ascii 7	(alarm)	émet un beep sonore
\f	ascii 12	(formfeed)	saut de page
\0	ascii 0	(nul)	fin de chaîne

Utilisés dans des printf(), ces caractères permettent d'améliorer l'affichage des résultats.

Exemples:

```
printf ( "\aVeuillez entrer une valeur:" );
/* affiche le message précédé d'un beep sonore */

printf ( "Fin du programme\n" );
/* affiche le message et saute au début de la ligne suivante
*/
```

Remarque: Tous les opérateurs vus pour les entiers peuvent s'appliquer aux caractères.

Les opérations sur les char, int, float peuvent conduire à des dépassement de capacité qui ne sont pas décelables à l'exécution. Il convient donc de choisir un type adéquat et de rester dans les limites prévues.

Exemple:

```
unsigned int a= 650000;
...
a+= 100000;
```

Contrairement à ce que l'on pourrait croire ça ne vaut pas 75000 car les unsigned int sont limités à 65535. En fait l'opération réalisée sera la suivante en hexadécimal au niveau machine:

$$\begin{array}{rcl}
 65000 & \rightarrow & \text{(1)} \\
 + 10000 & \rightarrow & \text{0xFDE8} \quad \text{hexa} \\
 \hline
 75000 & \rightarrow & \text{0x(1)24F8} \quad \text{hexa}
 \end{array}$$

La retenue 1 en hexadécimal est perdue. La valeur obtenue sur 16 bits sera donc 0x24F8 c'est à dire 9464 qui n'est autre que 75000 modulo 65536.

e) Les chaînes de caractères

Les chaînes de caractères sont essentiellement, comme on le verra plus loin (et si vous persévérez dans l'apprentissage du C !), des tableaux de caractères.

Elles méritent toutefois une attention particulière:

Une chaîne de caractère se déclare ainsi:

```
char nom[20];
```

char est le type de base, nom est le nom de la variable, 20 est le nombre de caractères maximal de la chaîne

Dans l'immédiat la chaîne nom n'est pas initialisée et ne contient donc rien de significatif.

On peut l'initialiser dans la déclaration:

```
char nom[20]="Gilles";
```

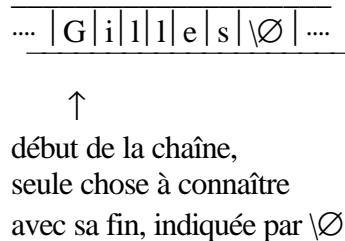
Dans ce cas la valeur 20 peut être omise et on peut se contenter de déclarer:

```
char moi[ ]="Gilles";
```

Auquel cas le compilateur allouera une chaîne de sept caractères.

Sept ? Vous avez bien dit sept ? Comme c'est bizarre !

En effet, il faut savoir qu'une chaîne de caractères en C est terminée par un caractère spécial, nommé NUL, de code ascii \emptyset (c'est à dire $\backslash\emptyset$).
 La chaîne "Gilles" sera donc stockée en mémoire sous la forme



Aussi la chaîne nom ci-dessus ne devra pas se voir affecter une chaîne de plus de 19 caractères.
 Si on essaye d'affecter à nom la chaîne "Anticonstitutionnellement" de la façon suivante, deux problèmes vont se poser:

```

char nom[20];
...
nom= "Anticonstitutionnellement";    /* erreur */
  
```

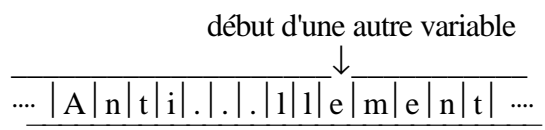
Tout d'abord l'affectation d'une chaîne de caractères—en dehors d'une initialisation lors de la déclaration— ne peut se faire avec =. Il faut utiliser la fonction de copie de chaîne strcpy()
 (voir § Chaînes de caractères).

Le compilateur refusera le programme précédent mais acceptera le suivant:

```

char nom[20];
...
strcpy( nom,
"Anticonstitutionnellement");
  
```

Toutefois un autre problème nous guette à l'exécution: la chaîne nom ne peut recevoir plus de 19 lettres et le mot en fait 25 !
 On aura alors écrasement en mémoire des variables qui suivent nom, avec des conséquences imprévisibles sur le déroulement du programme:



↑ ↑
 début de nom fin
 normale
 de nom

Remarque: De même la comparaison de deux chaînes de caractères ne peut se faire avec == ou !=. Il faut utiliser la fonction de comparaison de chaînes strcmp(). (voir § Manipulations de chaînes).

Au delà des fonctions de manipulation globales de chaînes (voir § Manipulations de chaînes)

il est utile d'accéder individuellement aux caractères d'une chaîne:

Exemple:

```

char nom[20]= "Gilles";
char c;
unsigned k;
...
c= nom[k];
...
nom[0]
```

c=nom[k]; permet de lire la k-ieme lettre de nom.

nom[0] = 'M'; modifie l'initiale du nom "Gilles" et en fait "Milles".

☛ Attention: La numérotation des caractères d'une chaîne commence à 0 !

Remarque: La chaîne vide "" est parfaitement licite.

f) Les différentes classes de variables

Maintenant que nous savons déclarer et manipuler les variables de base, demandons-nous où les déclarer.

Il existe en effet quatre principales classes de variables:
 les variables globales, locales, statiques, et externes.

Les variables globales:

Elles sont déclarées au début du programme et sont accessibles dans toutes les fonctions du programme.

Les variables locales:

Elles sont déclarées au début d'une fonction et ne sont utilisables, temporairement donc, qu'au sein de cette fonction.

On utilisera au maximum des variables locales, en ne déclarant globales que celles qui doivent être partagées par plusieurs fonctions.

Le main() lui même possède ses propres variables locales.

Les variables statiques:

Utilisées dans des cas très précis, dans une fonction, en faisant précéder la déclaration du mot static

Exemple:

```
int fonction_truc( )
{
    static int compteur=0;
    ...
    compteur+ +;
    return (compteur);
}
```

Cette fonction possède une variable statique, qui est intermédiaire entre globale et locale:

- Elle conserve sa dernière valeur appel après appel, alors qu'une variable locale est perdue d'un appel à l'autre de la fonction.
- Elle n'est pas accessible en dehors de la fonction.

Ici la variable compteur est initialisée à 0 mais seulement pour le premier appel de la fonction.

Ensuite elle est incrémentée à chaque appel et la fonction retourne ainsi le nombre de fois où elle a été utilisée au sein du programme.

Les variables externes:

Elles sont définies dans un autre source C qui doit être lié à celui-ci pour obtenir le programme complet (voir § Projets).

Exemple général:

```

#include <stdio.h>

int x;                /* variable globale */

int fonction_truc( )
{
    char c;
    char nom[25];    /* variables locales */
    static int x= Ø; /* variable statique initialisée */
    ...
}

int fonction_bidule( )
{
    char c;
    float f;
    ...
}

main( )
{
    int a;           /* variable locale au main( ) */
    ...
}

```

Remarques:
variables

- Dans des fonctions différentes on peut donner le même nom à des variables différentes sans qu'il y ait de confusion—cf variable c.

ailleurs

- Une variable locale peut porter le même nom qu'une variable globale. Dans la fonction truc(), x fera référence à la variable locale, partout ailleurs il s'agira de la variable x globale. A éviter car prête à confusion.

g) Les pointeurs:

La programmation en C serait finalement assez simple (mais oh combien limitée !) si les pointeurs n'existaient pas...
Ils font la puissance du C et leur emploi très fréquent distingue le C de nombreux autres langages évolués.

Un pointeur est une variable d'un type nouveau, destinée à recevoir l'adresse mémoire d'une variable de type int, float, char...

La déclaration d'un pointeur a la syntaxe suivante:

```
type *var_ptr;
```

Exemple:

```
char *chr_ptr;  
int *i_ptr;
```

L'étoile * signifie qu'il s'agit d'un pointeur sur le type indiqué.
C'est à dire que i_ptr contiendra l'adresse d'un entier.

☛ Attention: L'entier en question n'est pas précisé dans cette déclaration, et l'adresse contenue dans le pointeur i_ptr est actuellement indéterminée. C'est une des erreurs les plus courantes que de définir un pointeur sans qu'il n'y ait rien derrière!

On peut déclarer simultanément plusieurs pointeurs sur un même type à condition de faire précéder chacun d'entre eux d'une *.

Exemple: float *f_ptr, *g_ptr;

Le suffixe ptr attribué à chaque pointeur n'est pas impératif, il s'agit simplement d'une habitude personnelle qui permet de se rappeler qu'il s'agit d'un pointeur.

Toute variable a une adresse mémoire (qui est l'adresse de son premier octet) et qui peut être récupérée dans un pointeur. (Afin par exemple de passer un paramètre par adresse à une fonction—cf § Fonctions à ce sujet).

On récupère cette adresse via l'opérateur &:

Exemple:

```
int i;  
int *i_ptr;  
...  
i_ptr=&i;          /* i_ptr reçoit l'adresse de la variable i  
*/
```

La fonction scanf() utilise l'adresse des variables à saisir, car il faut lui signifier où mettre chaque valeur saisie.

Exemple: scanf("%d", &i); (cf § Fonctions d'entrée/sortie)

Inversement, étant donné un pointeur ayant été initialisé—c'est à dire qui pointe effectivement sur une variable connue—il est nécessaire de pouvoir récupérer le contenu de la variable pointée, via l'opérateur * d'indirection.

Exemple:

```
int i, j;
int *i_ptr;    /* pointeur non initialisé */

i_ptr=&i;      /* i_ptr pointe maintenant sur la variable i */

j= *i_ptr;    /* équivaut en fait à écrire j=i */
```

Les opérateurs * et & s'annulent: $j = *i_ptr = *(&i) = i$

Un pointeur déclaré peut être dans trois états dont deux ont été vus:

- Le pointeur n'est pas initialisé et pointe sur un emplacement mémoire très aléatoire.
- Le pointeur pointe sur une variable connue.

Le troisième état fait intervenir le pointeur symbolique pré-défini NULL:

- Le pointeur pointe sur le néant, il a pour contenu l'adresse NULL.

Exemple:

```
char *chr_ptr;
...
chr_ptr= NULL;
```

En général un pointeur ayant la valeur NULL signifie quelque chose de particulier, par exemple une fonction retournant un pointeur peut retourner NULL si elle n'est pas en mesure d'accomplir l'action demandée (voir plus loin pour des exemples).

☛ Attention: On ne peut appliquer l'opérateur * à un pointeur de valeur NULL puisqu'il n'y a pas de variable pointée. On rejoint le cas où le pointeur n'est pas initialisé, à cela près qu'à l'exécution le programme risque de générer le message désagréable "NULL pointer assignment".

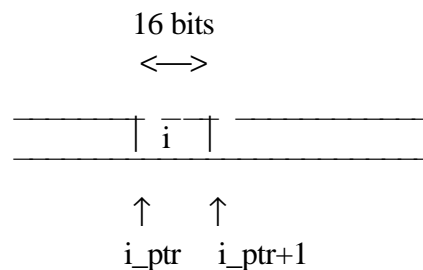
Arithmétique des pointeurs

Les pointeurs sont soumis à un nombre restreint d'opérations très particulières.

Incrémentation d'un pointeur:

```
int *i_ptr;
int i;
...
i_ptr=&i;          /* i_ptr pointe sur i */
i_ptr ++;
```

Que peut bien signifier l'opération `i_ptr ++` ?
On a en mémoire le schéma suivant:



L'opération `i_ptr++` c'est à dire `i_ptr= i_ptr+1` ajoute en fait à `i_ptr` la taille en octets du type pointé; ici un `int` est sur 16 bits soit 2 octets.

Donc `i_ptr+1` est un pointeur sur la variable située deux octets plus loin que `i`.

De même l'opération `i_ptr= i_ptr+k` ou encore `i_ptr+=k` ajoute à `i_ptr` `k` fois la taille du type pointé.

Ceci permet comme on en reparlera (cf § Tableaux) de se déplacer en avant dans un tableau en mémoire.

Décrémentation d'un pointeur

De même `i_ptr--` ou `i_ptr-=k`

permet de se déplacer en arrière en mémoire, même si cela est moins fréquent.

Reste à voir un point que vous pouvez passer en première lecture:
il s'agit des différents modèles de pointeurs.

L'existence de ces modèles de pointeurs est liée à l'architecture matérielle des PC qui pour des raisons historiques de compatibilité ascendante ont un adressage mémoire très particulier.

La plupart des processeurs ont un adressage linéaire de la mémoire, les adresses des octets en mémoire variant progressivement de 0 à une valeur maxi. Si par exemple le processeur

a des registres 32 bits les adresses vont au maximum de 0x00000000 à 0xFFFFFFFF

soit 4 GO de mémoire au maximum.

S'il dispose effectivement de 4 MO de mémoire linéaire, les adresses iront de 0x00000000

à 0x40000000.

Par contre sur un PC les registres de base sont sur 16 bits et la mémoire est segmentée.

En effet 16 bits permettent des adresses de 0x0000 à 0xFFFF soit 64 KO de mémoire,

ce qui est insuffisant pour adresser 1 MO de mémoire conventionnelle.

(La mémoire étendue est encore une autre paire de manches !).

Une adresse mémoire est stockée sur 2 mots de 16 bits, un registre de segment et un offset.

L'adresse réelle correspondant linéairement au couple segment:offset s'obtient de la façon suivante:

$$\text{adresse linéaire} = 16 * \text{segment} + \text{offset}$$

Comme multiplier par 16 revient en base 16 à décaler d'un digit on peut traiter facilement

un exemple:

L'adresse linéaire correspondant à l'adresse segmentée 1B3E : 0143 est:

$$\begin{array}{r} 1B3E \\ + \quad 0143 \\ \hline 1B523 \end{array}$$

Ce mode d'adressage est très inférieur à un véritable adressage sur 32 bits car les adresses linéaires vont de 0x00000000 à 0xFFFFFFFF et permettent de gérer 1 MO de mémoire.

Cela s'explique par le fait qu'une même adresse linéaire peut être décrite par de nombreux couples segment :offset. Il suffit de diminuer le segment de 1 et d'augmenter

l'offset de 16 pour rester au même endroit vu que:

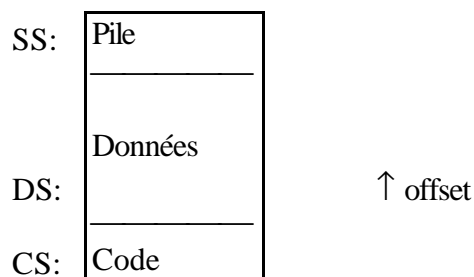
$$16 * (\text{segment} - 1) + (\text{offset} + 16) = 16 * \text{segment} + \text{offset}$$

Un programme exécutable sur PC comporte au moins 3 parties:

- une partie code
- une partie données
- une partie pile

Chaque partie est associée à un ou plusieurs segments selon la taille requise.

Le segment de code est donné par le registre 16 bits CS, les données sont référencées par DS ou ES et la pile par SS.



Les variables globales et statiques sont définies dans les données, par contre les variables locales sont allouées sur la pile.

☛ Attention: Il y a un seul segment de pile, de 64 KO maximum, aussi ne doit on pas déclarer des variables locales trop volumineuses (sinon gare au dépassement de pile)
telles que de gros tableaux...

S'il y a un seul segment de données—dans le cas d'un petit programme—chose qui est déterminée par le compilateur et par le modèle mémoire adopté, d'adresse de toute variable

est entièrement déterminée par son offset à l'intérieur du segment DS.

On parle alors de pointeur near ou pointeur court, dont la taille est de 16 bits.

S'il y a plusieurs segments de données—dans le cas d'un gros programme—l'adresse d'une variable comporte un segment et un offset.

On parle alors de pointeur far ou pointeur long dont la taille est de 32 bits.

Par défaut le format nécessaire aux pointeurs est déterminé par le compilateur lors de la déclaration:

```
type *var_ptr;
```

On peut en cas de besoin forcer un pointeur à être near ou far de la façon suivante:

```
type far *var_ptr;  
type near *var_ptr;
```

Exemple:

```
char far *chr_ptr;
```

h) définition d'un nouveau type

La déclaration typedef permet en début de programme (en général) de définir un nouveau type conçu à partir des types de base:

Exemple: Si on utilise souvent dans un programme des chaînes de 64 caractères maximum on peut déclarer:

```
typedef char chemin[64];  
  
/* le type chemin <= > char[64] */  
  
chemin ch1, ch2;  
  
/* ch1 et ch2 sont deux variables du type chemin */
```

3) Les tableaux de variables:

Un tableau est une liste ordonnée contenant un certain nombre de données du même type.

Le type en question peut être simple ou structuré (cf § Structures).

Tableaux unidimensionnels:

La déclaration est de la forme

```
type nom_tab [N];
```

où type est le type des éléments du tableau, nom_tab le nom de cette variable tableau et N une constante donnée qui est le nombre d'éléments du tableau.

Exemples:

```
int tab [10];  
float data [25];  
char chaîne [20];
```

On retrouve ici le fait qu'une chaîne de caractères n'est autre qu'un tableau de caractères.

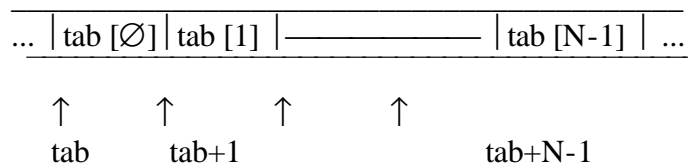
L'accès aux éléments du tableau se fait en utilisant leur indice, c'est à dire leur position dans le tableau.

☛ Attention: La numérotation des indices commence à \emptyset et se termine donc à N-1.
Elle se terminerait même plutôt à N-2 pour une chaîne en raison du caractère NUL final.

Exemple:

```
int i;
int tab [10];          /* tableau de 10 entiers */
...
i= tab [0];           /* lecture premier élément du tableau */
tab[9]=325;          /* écriture dans dernier élément du tableau */
```

Remarque: Pour tout entier non signé k, tab[k] représente le k-ième élément du tableau:



Les éléments sont rangés en mémoire de façon consécutive et dans l'ordre croissant des indices.

Le symbole tab, nom générique du tableau représente pour le compilateur un pointeur du type déclaré sur le début de ce tableau.

Selon l'arithmétique des pointeurs, les adresses des éléments suivants du tableau sont données par $\text{tab} + k * (\text{taille du type pointé})$.

Ainsi le k-ième élément tab[k] peut-il être obtenu également par $*(\text{tab} + k) \Leftrightarrow \text{tab}[k]$

Tableaux multidimensionnels

On peut définir des tableaux ayant 2 ou 3 dimensions, voire plus:

Exemple:

```
float matrice [3] [3];
char motx [8] [10];
int tab [2] [5] [10];
```

La variable matrice représenterait ici une matrices 3x3 à coefficients flottants.

La variable motx pourrait être un mot croisé de 8 lignes et 10 colonnes.

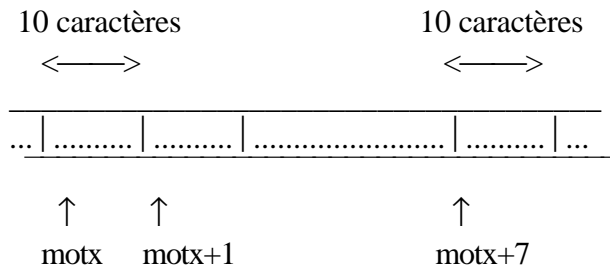
Tab serait enfin un tableau tridimensionnel d'entiers.

L'accès aux éléments d'un tel tableau se fait avec 2 ou 3 indices, commençant tous à \emptyset .

Exemples:

```
float matrice [3] [3];
float f;
...
f= matrice [Ø] [2];
```

Remarque: Un tableau bidimensionnel tel que motx ci-dessus est stocké en mémoire de la façon suivante:



Il est construit comme un tableau de 8 éléments, dont chaque élément est un tableau de 10 caractères.

Cette disposition montre que les 8 lignes du mot croisé sont facilement lisibles (elles sont aux adresses motx+10*k car motx est un pointeur sur caractère). Mais il est plus difficile d'en extraire les colonnes !

Si on voulait privilégier les colonnes on déclarerait char motx [10] [8];

☛ Attention: Lors de l'accès en lecture ou écriture à un élément de tableau par son indice, il n'est fait aucune vérification de la validité de l'indice: si celui-ci sort des limites prévues il y a risque d'écrasement de données en mémoire donc danger !

Initialisation d'un tableau

Un tableau peut être initialisé dans sa déclaration avec une liste de valeurs entre accolades. Quelques exemples valant mieux qu'un long discours:

Exemples:

```
int tab [5] = { 1, 3, 5, 7, 9 };
/* le nombre d'élément ( entre crochets ) peut être omis */
/* le compilateur le calculera */

float matrice [3] [3] = {{1, -2, 3 },
                        { 5, Ø , 7 },
                        { Ø, -1, 4 }};

char motx [3] [5] = {{ 'C', 'A', 'R', 'N', 'E'},
                    { ' ', 'M', 'A', 'I', 'N'},
                    { 'B', 'E', 'T', 'E', ' '}};
```

Le dernier pourrait être vu comme tableau de 3 chaînes de caractères et initialisé ainsi, en tenant compte du NUL final de chaque chaîne:

```
char motx [3] [6] = {"CARNE",  
                    " MAIN",  
                    "BETE "};
```

4) Les structures:

La manipulation de données nécessite souvent l'emploi de structures définies par le programmeur. Une structure est constitué de plusieurs éléments nommés champs qui peuvent être de types variés.

Alors que dans un tableau tous les éléments sont du même type.

La déclaration d'une structure est de la forme suivante:

```
struct nom_struct  
{  
    type1 nom_champ1;  
    ...  
    typeN nom_champN;  
};
```

Une variable ayant cette structure sera déclarée par

```
struct nom_struct nom_var;
```

Exemples: (Il me semble que cela s'impose !)

On veut définir une structure date servant à mémoriser

- un jour de la semaine (Lundi → Dimanche)
- un jour du mois (1 → 31)
- un mois (1 → 12)
- une année (4 chiffres)

On déclarera alors:

```

struct date
{
    char js[10];          /* 10 caractères pour le jour de la semaine */
    unsigned char jm;    /* jour du mois 8 bits suffisent */
    unsigned char ms;    /* le mois */
    unsigned an;         /* l'année */
};

struct date today;

```

Accès aux champs d'une structure:

Pour remplir la structure today, de type date on accède aux champs de la façon suivante:

Exemple:

```

today.an= 1995;
today.ms= 9;
today.jm= 1;
strcpy( today.js, "V.endredi" );

```

On fait suivre le nom de la variable structurée d'un point et du nom du champ.

La lecture d'un champ se fait de même:

Exemple:

```

unsigned a;
...
a= today.an;

```

Remarque: Le C utilise lui même des structures pré-définies comme la structure FILE.

(cf § Manipulations de fichiers).

Initialisation d'une structure:

On peut initialiser une structure tout comme un tableau:

Exemple: struct date revol= { "", 14, 7, 1789 };
 (Ignorant le jour de la semaine j'y ai mis la chaîne vide "");

Pointeur sur une structure:

Il est courant d'introduire des pointeurs sur des structure, en particulier pour accéder à une structure passée en paramètre à une fonction.

Exemple:

```
struct test
{
    char nom[10];
    int num;
};

struct test *var_ptr;
```

(Désolé, cette structure ne sert à rien !)

On accède au champ num de la variable pointée — qui n'est pas déclarée ici — a priori de la façon suivante:

```
int n;
...
n= (*var_ptr ).num;
```

Cette notation étant un peu lourde un raccourci a été introduit:

n= var_ptr->num; qui équivaut à n= (*var_ptr).num;

Dans cette écriture, la flèche s'obtient avec - et > .

Définition d'un nouveau type structuré:

Vous aurez remarqué que d'on doit répéter struct devant le nom de la structure à chaque variable structurée déclarée, ce qui est fastidieux si l'on en fait grand usage. Pour éviter cela on peut faire un typedef:

Exemple:

```
typedef struct
{
    char *str_ptr;
    int num;
}
test;

test var;
```

Dans ce cas le nom de ladite structure est placé à la fin.

Le mot réservé struct n'est plus nécessaire devant la déclaration de la variable var.

Remarque: Il peut être utile d'avoir une idée de la taille d'une structure (pour copier une structure dans un fichier par exemple) .
L'opérateur sizeof () est là pour cela:

Exemple: sizeof(struct date)
 sizeof(test)

Il renvoie un unsigned qui est la taille en octets de la structure.
Cet opérateur s'applique ainsi—mais c'est moins intéressant—aux types de base.
Par exemple sizeof(long double) devrait renvoyer 10 (octets).

Remarque: On peut manipuler des pointeurs sur structures comme les autres.
(Comme dans les listes chaînées).

5) Les boucles et sauts:

Le langage C est un langage structuré dans lequel la notion de saut inconditionnel—goto—existe mais est déconseillée.
On préférera utiliser des boucles et tests judicieusement agencés.

a) Les boucles:

Une boucle dans un programme est un bloc d'instructions devant être répété un certain nombre de fois, ou jusqu'à ce qu'arrive un événement particulier...
Il existe en C trois types de boucles, les boucles for, les boucles do...while et les boucles while.

Leur syntaxe est la suivante:

boucles for: for (initialisation ; condition d'itération ; incrémentation)
 {
 ...
 instructions à répéter
 ...
 }

L'initialisation consiste en général à positionner une variable à une valeur initiale, l'incrémentacion indique comment évolue ladite variable après chaque itération, la condition d'itération stipule à quelles conditions—sur cette variable en général—les itérations doivent continuer.

Exemple:

```

int tab[10];
unsigned i;
...
for ( i = 0 ; i < 10 ; i ++ )
    {
        printf( "Entrer une valeur entière:\n" );
        scanf( "%d", &tab[i] );
    }

```

Cette boucle permet la saisie des éléments d'un tableau.

Remarque: Si le bloc comporte une seule instruction les { } peuvent être omises.

☛ Attention: Ne pas mettre de ; derrière le for sinon le compilateur considère la boucle terminée et il n'y a pas d'itération !

Boucles do...while:

```

do
    {
        ...
        instruction à répéter
        ...
    }
while ( condition d'itération );

```

La boucle est exécutée tant que la condition d'itération est satisfaite. Le test étant effectué à la fin, une telle boucle sera toujours exécutée au moins une fois.

Exemple:

```

char c;
...
do
    {
        printf( "choisissez oui ou non ( O/N ): " );
        scanf( "%c", &c );
    }
while (( c != `O` ) && ( c != `N` ));
...

```

Cette boucle opère la saisie d'une touche au clavier jusqu'à ce que la réponse soit convenable (O ou N).

Remarque: Si le bloc comporte une seule instruction, les { } peuvent être omises.

Le while se termine ici par un ;

Boucles while:

```
while ( condition d'itération )
{
...
instruction à répéter
...
}
```

La boucle est exécutée tant que la condition d'itération est vraie. Cette fois ci le test est effectué avant la boucle, contrairement au cas précédent do...while.

Exemple:

```
#define NUL 0

unsigned k= 0;
char chaine[ ] = "what's up doc?";
...
while (( chaine[k] != `d` ) && ( chaine[k] != NUL ))
{
    k++;
}
...
```

Cette boucle parcourt la chaîne jusqu'à rencontrer la lettre `d`.
En sortie l'indice k donne la position de cette lettre dans la chaîne.
On teste aussi la fin de la chaîne avec le NUL final pour ne pas aller trop loin.

☛ Attention: Gare aux boucles infinies avec while et do...while !

Signalons que: while (1)
 {
 ...
 }

est une boucle infinie, en cas de besoin, car la valeur 1 non nulle, est considérée comme TRUE.

Le choix d'un type de boucle:

La diversité de ces boucles pose un problème au programmeur débutant: laquelle choisir ?

Il n'y a pas de réponse unique, et disons même que l'on doit pouvoir quasiment tout faire avec n'importe quel type de boucle, mais c'est une question d'élégance.

L'organigramme suivant peut dans le doute, vous guider vers un choix judicieux:

Les exemples précédemment cités illustrent cette stratégie.
Ajoutons en un quatrième pour la situation manquante:

Exemple:

```
char passwd[ ]= "JBØØ7";
char saisie[1Ø];

do
    {
    printf( "taper le mot de passe:" );
    scanf( "%s", saisie );      /* saisie d'une chaîne */
    }
while ( ! strcmp ( saisie, passwd ) )
```

On saisit un mot de passe jusqu'à ce qu'il soit correct.
La décision dépend du résultat de strcmp() donc le test doit être placé à la fin.

Instructions de sortie de boucle:

Pour rompre la monotonie d'une boucle for, while ou do...while, on peut utiliser deux instructions:

Break: Met fin à la boucle et saute à l'instruction qui la suit immédiatement dans le programme.

Continue: Met fin à l'itération en cours et recommence la prochaine itération si la condition le permet.
Cela permet d'ignorer une partie des instructions de la boucle qui peuvent être inutiles dans certains cas.

Instruction de saut inconditionnel:

Il existe une seule instruction de saut, il s'agit de goto.
Comme indiqué au début de ce paragraphe, cette instruction doit être évitée autant que possible.

Un goto permet de faire un saut à un label au sein d'une même fonction.
Un label est une marque qui désigne une instruction du programme.

Exemple:

```
int i;
...
if ( i == 0 ) goto fin;
...
fin: /* label suivi de : */
printf( "erreur\n" );
...
```

b) Les tests et les sauts conditionnels:

Un programme serait bien limité en l'absence de tests permettant d'effectuer des choix.

On a déjà parlé des tests plus haut dans les opérateurs de comparaison.

Syntaxe:

```
if ( condition )
{
...
instructions
...
}
else
{
...
}
```

```
instructions
...
}
```

Le premier bloc est exécuté si la condition est remplie et le second l'est dans le cas contraire.

Rappelons que le else est facultatif. En son absence, et lorsque la condition n'est pas satisfaite, on passe directement à l'instruction suivante.

Lorsqu'un bloc est réduit à une instruction, les { } peuvent être omises.

Le plus intéressant est la forme de la condition, ce qui vaut également pour les boucles

vues plus haut.

Une condition élémentaire prend la forme d'une comparaison de deux variables avec des opérateurs =, !=, <, >, <=, >=.

Les deux premiers permettent de comparer des nombres, des structures, mais pas des tableaux pour lesquels on doit opérer une comparaison élément par élément.

Ils ne s'appliquent pas non plus aux chaînes de caractères pour lesquelles on fera appel à strcmp() (cf § Manipulations de chaînes).

Les autres opérateurs s'appliquent uniquement à des nombres.

Exemple:

```
if ( i != Ø )
{
...
}
else
{
...
}
```

On peut également opérer un test directement sur le résultat d'une fonction lorsque celui-ci est un entier:

Ainsi la fonction strcmp(chaîne1, chaîne2) renvoie Ø si chaîne1 et chaîne2 coïncident

et un résultat non nul dans le cas contraire.

Le test if (strcmp (chaîne1, chaîne2) == Ø)

qui détermine l'égalité de ces deux chaînes peut être abrégé en

```
if ( ! strcmp ( chaîne1, chaîne2))
```

dans la mesure où la négation logique ! transforme FALSE (la valeur Ø) en TRUE, et TRUE (toute valeur non nulle) en FALSE.

Plusieurs conditions élémentaires peuvent être combinées en une condition complexe à l'aide des opérateurs logiques:

&& et (and)
|| ou (or)
! négation (not)

La table de vérité de ces opérateurs est:

C1	C2	!C2	C1&&C2	C1 C2
TRUE	TRUE	F	T	T
TRUE	FALSE	T	F	T
FALSE	TRUE	F	F	T
FALSE	FALSE	T	F	F

L'usage de parenthèses permet d'écrire toutes les conditions imaginables:

Exemples: if (a == b)
 if ((a > 1) && (a < 10))
 if ((a == 0) || ((a != 0) && (b == 1)))

c) les choix multiples:

Lorsque de nombreux choix sont possibles il est fastidieux d'imbriquer de multiples if...else qui sont difficiles à relire.

Aussi on pourra utiliser les instructions switch...case dont la syntaxe est:

```
switch ( variable )  
{  
case v1:        instructions  
                  ...  
                  break;  
...  
case vN:        instructions  
                  ...  
                  break;  
default:        instructions
```

```
...
break;
}
```

Variable est une variable de type entier ou caractère.

v1, ..., vN sont des valeurs particulières de cette variable auxquelles sont associées des actions. Chaque cas se termine par un break qui fait sortir du switch...case.

☛ Attention: Si le break est omis les instructions du cas suivant seront aussi exécutées.

La rubrique facultative default associe un traitement par défaut à tous les cas autres que ceux explicitement mentionnés.

Exemple typique:

```
char c;
...
printf( "choisissez (R)ecommencer (C)ontinuer (A)bandonner\n"
);
scanf( "%c", &c );
switch (c)
{
    case 'R':    ...
                break;
    case 'C':    ...
                break;
    case 'A':    ...
                break;
    default:    printf( "choix erroné\n" );
                break;
}
```

6) Les fonctions:

a) Description:

On distingue deux types de fonctions,

- Les fonctions pré-définies dans les bibliothèques de langage C.
- Les fonctions définies par l'utilisateur.

Elles fonctionnent de la même façon, et si nous nous concentrons sur les secondes, tout ce que nous dirons s'appliquera aussi aux premières, sauf mention contraire.

Le but d'une fonction est, à partir de certaines données, d'effectuer des calculs ou des manipulations et si nécessaire d'en rendre compte par des résultats.

En somme une fonction est une portion de programme chargée de réaliser une tâche particulière.

Tout programme C comporte une fonction `main()`, qui fait appel en général à d'autres fonctions, qui appellent elles mêmes des sous fonctions...

D'où l'architecture d'un programme c:

Une saine programmation consiste à fractionner le travail du programme en tâches aussi indépendantes que possible qui seront elles mêmes fractionnées en tâches élémentaires.

Les opérations les plus élémentaires, dont certaines ne sauraient être réinventées comme la saisie ou l'affichage, les manipulations de fichiers, etc. sont réalisées par les fonctions pré-définies du langage C.

Une fonction porte un nom que l'on choisira aussi explicite que possible quant au rôle de la fonction; le nom peut comporter des soulignés '_'.

Exemple: `find_first_maj()`;
 pour une fonction qui rechercherait (why not ?)
 la première majuscule d'une chaîne de caractères.

Hormis les fonctions pré-définies qui sont déclarées dans des headers.h, toutes les fonctions doivent être déclarées au début du programme.

Syntaxe déclaration: `type_retourné nom_ft (type_param_1, . . . , type_param_N);`

Exemple déclaration: `float cube(float);`

Pour une fonction nommée `cube` qui calcule le cube d'un nombre flottant.

La déclaration se termine par un ;

Plus loin dans le programme on pourra définir la fonction, c'est à dire préciser son action et les résultats qu'elle renvoie:

Exemple définition:

```
float cube( float f )
{
    float f3;

    f3= f*f*f;
    return ( f3 );
}
```

Il n'y a rien de surprenant, mais on notera que cette fois le paramètre porte un nom, f, qui est utilisé dans la fonction, et que la définition ne se termine pas par un ;
Return (f3) indique la valeur float que renvoie la fonction.

Remarque: Une fonction ne peut être définie à l'intérieur d'une autre fonction.

Plus loin dans le programme on pourra faire appel à la fonction cube:

Exemple d'appel de fonction:

```
main( )
{
    float x, y;

    printf( "donner un nombre flottant:" );
    scanf( "%f", &x );
    y= cube( x );
    printf( "le cube de ce nombre est:" );
    printf( "%f", y );
}
```

Le programme calcule et affiche bien sûr le cube de ce qu'on lui soumet.
Le paramètre passé à la fonction cube peut porter n'importe quel nom de variable float définie dans main(). La valeur passée se substituera à f lors du calcul du cube dans la fonction.

b) Passage des paramètres:

On peut passer à une fonction des paramètres de nombreux types:
char, int, unsigned, long, float, double float, long double, pointeur, structure.

Une fonction sans paramètre est déclarée sous la forme

type_retourné nom_ft(); ou type_retourné nom_ft(void).

Le passage de paramètre se fait en C par valeurs, c'est à dire que la fonction reçoit la valeur du paramètre, par opposition à d'autres langages où le passage de paramètre se fait par adresse, lorsque la fonction reçoit en fait l'adresse du paramètre.

En fait on peut faire du passage de paramètre par adresse si nécessaire en utilisant un pointeur.

C'est d'ailleurs le cas lorsqu'on passe un tableau ou une chaîne de caractères à une fonction:

en effet il est trop long de transmettre tout le tableau ou toute la chaîne et on transmet en fait son adresse.

Exemple: La fonction pré-définie `strlen()` calcule la longueur d'une chaîne de caractères. Son prototype est en fait `unsigned strlen(char *)`.

Donc elle reçoit un pointeur sur caractère et retourne un entier long.

Un pointeur sur caractère car on lui passe en fait l'adresse du premier caractère de la chaîne dont elle doit calculer la longueur.

A l'utilisation on aura par exemple:

```
char toto[ ]= "To be or not to be";
unsigned l;
...
l= strlen( toto );
```

On ne doit pas mettre de `&` devant `toto` car, rappelons le, le nom d'une chaîne de caractères

(ou d'un tableau) est en fait un pointeur sur son premier élément;

donc le paramètre transmis est bien du type attendu `char * !`

c) Valeur de retour:

Le type de valeur en retour d'une fonction est hélas bien plus limité:

`void`, `char`, `int`, `unsigned`, `long`, `float`, `double float`, `long double`, `pointeur`.

En particulier on ne peut retourner une chaîne de caractères ou un tableau,

mais on peut tout de même retourner une structure !

Le retour de la valeur se fait par `return` (variable du type attendu).

Il peut y avoir plusieurs `return` (variable) dans une même fonction.

Le premier rencontré met fin à la fonction et retourne la valeur indiquée.

Une fonction a un type de retour `void` si elle ne renvoie aucun résultat.

Une fonction `void` se termine par `return ()` ou tout simplement à la fin du bloc qui la définit.

La fonction main() est en général déclarée void main() ou void main(void) sauf cas particuliers (voir § Exécution de processus).

La principale limitation est qu'une fonction ne peut renvoyer qu'un seul résultat.

Que faire si on a besoin en retour de deux informations — ou plus ?

On prévoit dans la fonction appelante deux variables — ou plus —

et on passe leurs adresses à la fonction appelée, via des pointeurs, afin que la fonction puisse mettre ses résultats dans lesdites variables !

On procède de même si la fonction doit retourner une chaîne de caractères ou un tableau.

Exemple:

```
#include <stdio.h>
#define NUL  
#define SPC 32

void first_word( char *, char * );

char phrase[2 ]= "langage c";
char debut[2 ];

void first_word( char *chaîne, char *mot )
{
    unsigned k= ;

    while (( chaîne[k] != SPC ) && ( chaîne[k] != NUL ))
        {
            mot[k]= chaîne[k];    /* recopie une lettre à chaque fois */
        }
}

main()
{
    printf( "%s", phrase );
    first_word( phrase, debut );
    printf( "%s", debut );    /* devrait afficher langage */
}
```

La fonction first_word() isole dans une chaîne de caractères le premier mot —délimité par un espace ou une marque de fin de chaîne.

Remarques: chaîne est effectivement un paramètre transmis, tandis que mot sert en fait au retour d'information.

Une autre technique consiste à utiliser une variable globale par laquelle la fonction renvoie son résultat, mais cela est moins esthétique et peut engendrer des conflits si plusieurs fonctions prétendent utiliser la même variable globale.

Remarque: De nombreuses fonctions pré-définies dans les bibliothèques du C renvoient un int, qui n'est pas vraiment un résultat, mais plutôt un compte rendu qui permet de savoir si la fonction s'est bien déroulée ou pas:

retour= 0 signifie que oui
retour= -1 (ou autre valeur convenue) signifie qu'il y a eu un problème.

On n'hésitera pas à conserver ce principe pour les fonctions que l'on définit soi-même.

Remarque: Le casting de type:

Une fonction qui reçoit théoriquement un paramètre de type long peut très bien se voir passer un paramètre de type int: le compilateur assure dans ce cas le casting (conversion de type) automatiquement.

De même un int peut être remplacé par un char sans problème.
Il en est de même pour les valeurs de retour des fonctions où un int peut recevoir un résultat char, et un long peut recevoir un résultat int.

Un casting en sens inverse est possible mais la valeur est alors tronquée et il peut y avoir perte d'information.

Le casting est encore plus souple entre un int et un unsigned dans la mesure où ces deux types sont tous les deux représentés sur 16 bits.

Au delà des castings automatiques réalisés par le compilateur, le programmeur peut imposer un casting en faisant précéder une variable ou une fonction d'un type entre parenthèses.

exemple:

```
int i;  
float fl;  
...  
fl= (float) i ; /* on convertit i en float avant de le diviser par 2 */  
fl/=2; /* afin d'obtenir un résultat exact */
```

IV) Les fonctions d'entrée / sortie.

On appelle fonctions d'entrée/sortie les fonctions de saisie au clavier et d'affichage à l'écran.

Les fonctions ci-dessous nécessitent l'inclusion du header <stdio.h>.

1) Printf() et scanf():

Ces deux fonctions déjà rencontrées dans les exemples permettent un dialogue (rudimentaire) entre le programme et l'utilisateur.

Elles permettent de saisir—pour scanf()—et d'afficher—pour printf()—des données de types char, int, unsigned, long, float, double float, long double, et des chaînes de caractères.

Exemple: `printf("hello, brave new world");` `/* affiche la chaîne indiquée */`

Les chaînes de caractères peuvent contenir des **caractères de contrôle** sous forme de séquences d'échappement:

<code>\a</code>	(alarm)	Beep sonore
<code>\b</code>	(backspace)	Retour un caractère en arrière
<code>\f</code>	(form feed)	Saut de page
<code>\n</code>	(newline)	Saut de ligne
<code>\r</code>	(return)	Retour chariot
<code>\t</code>	(tab)	Tabulation

Du fait du rôle particulier de l' \ , celui-ci ne peut être utilisé tel quel.

De même le délimiteur de chaîne " ne peut être utilisé tel quel:

<code>\'</code>	Donnera une apostrophe
<code>\"</code>	Donnera un guillemet
<code>\\</code>	Donnera un antislash

Pour afficher des variables, on doit indiquer à `printf()` le type de variable à afficher dans une **chaîne de format** qui précède la variable elle-même:

Exemple:

```
int k;
...
printf( "%d", k );      /* affiche la valeur de l'entier k */
```

Les formats principaux sont:

<code>%d</code> ou <code>%i</code>	affichage d'un int
<code>%u</code>	affichage d'un unsigned
<code>%c</code>	affichage d'un char
<code>%s</code>	affichage d'une chaîne de caractères

%e ou %E	affichage d'un float en notation scientifique
%f	affichage d'un float en notation décimale
%o	affichage d'un unsigned au format octal
%x ou %X	affichage d'un unsigned au format hexadécimal
%Fp	affichage d'un pointeur far
%Np	affichage d'un pointeur near

Il existe des variations avec le préfixe l ou L.

%li ou %ld	affichage d'un long int
%lu	affichage d'un unsigned long
%lo	affichage d'un unsigned long en octal
%lx ou %lX	affichage d'un unsigned long en hexadécimal
%le , %lE ou %lf	affichage d'un double float
%Le , %LE ou %Lf	affichage d'un long double

Exemples:

```
int a= 25;
char chr= '@';
char chaine[ ]= "Bonjour";
...
printf( "%d", a );
printf( "%c", chr );
printf( "%s", chaine );
```

Les choses commencent à devenir intéressantes lorsque l'on sait que l'on peut combiner formateurs et textes dans une même chaîne de format avec plusieurs variables:

les formateurs seront à l'affichage remplacés par les variables dans l'ordre correspondant:

Exemple:

```
int j= 25;
char mois[ ]= "Septembre";
...
printf( "Nous sommes le %d %s 1995\n", j, mois );

/* affiche Nous sommes le 25 Septembre 1995 */
/* avec un saut de ligne à la fin */
```

Pour terminer signalons encore quelques options utiles dans les formateurs:

+ force l'affichage de + ou - pour une variable numérique.

Exemple:

```
int a = + 30;  
...  
printf( "a= %+d", a ); /* affichage a=+30 */
```

Nombre_entier nb

Indique le nombre minimal de chiffres à afficher.

S'il y a plus de chiffres que prévu ils seront tous affichés.

S'il y en a moins l'affichage sera complété par:

- des espaces après si nb est positif
- des espaces avant si nb est négatif
- des zéros au lieu d'espaces si nb commence par un zéro

Exemples:

```
int a= 253;  
...  
printf( "a= <%5i>", a );  
printf( "a= <%-5i>", a );  
printf( "a= <%-05i>", a );
```

Ces instructions affichent respectivement

```
a=<253 >  
a=< 253>  
a=<00253>
```

.Nombre_entier nb

Indique le nombre maximal de décimales à afficher pour un nombre flottant.

Pour une chaîne .nb définit le nombre maximal de caractères à afficher, quitte à tronquer la chaîne.

Exemples:

```
float fl= 12, 457;  
char toto[ ]= "Trucider";  
...  
printf( "fl= %.2f", fl ); /* affiche fl=12, 45 */  
printf( "%.4s", toto ); /* affiche truc */
```

Passons enfin à scanf():

On peut saisir avec scanf() des char, int, unsigned, long, float en tous genres, ainsi que des chaînes de caractères.

Le principe est le même que pour printf() ! Une chaîne de format indique le type de ce qui doit être saisi, mais suivie de **l'adresse** de la variable à saisir.

Exemples:

```
int a;
char chr;
char toto[10];
...
scanf( "%d", &a );
scanf( "%c", &chr );
scanf( "%s", toto );
/* Attention ⚠ : */
/* toto est déjà un pointeur donc une adresse, ne pas mettre & */
```

La saisie de plusieurs données en une fois est possible mais plutôt scabreuse... Pour la saisie des chaînes de caractères on utilisera de préférences les fonctions dédiées vues ci-après.

La chaîne toto est remplie dans cet exemple avec tous les caractères saisis jusqu'au ↵. Si on dépasse les 9 caractères (+NUL) autorisés on risque un écrasement mémoire. Afin de l'éviter on peut utiliser un formateur nombre qui limite le nombre de caractères saisis (les autres seront ignorés).

Exemple: scanf("%9s", toto); /* saisie d'au plus 9 caractères dans toto */

2) Fonctions de saisie/affichage de caractères et de chaînes:

Ces fonctions nécessitent également le header <stdio.h>

a) putchar() et getchar():

Syntaxe: putchar(int chr);
 int getchar();

getchar() lit un caractère suivi de ↵ au clavier.

Elle retourne soi-disant un int pour des raisons techniques: en cas d'erreur (je ne vois pas trop quoi) elle retourne -1;

Ce -1 est codé sur 16 bits c'est à dire 0xFFFF qui le distingue alors de la valeur -1

sur 8 bits 0xFF qui est aussi le caractère de code ascii 255.

En fait on l'utilise avec des char dans la mesure où le compilateur assure la conversion (casting) de façon automatique.

De même on passera en fait un char à la fonction putchar(), pour affichage.

Exemple:

```
char chr;
...
printf( "Taper un caractère:" );
chr= getchar( );
printf( "Ce caractère est:" );
putchar( chr );
```

b) puts() et gets():

Syntaxe: puts(char *chaîne);
 char* gets(char *chaîne);

puts() affiche une chaîne de caractère à l'écran avec passage à la ligne automatique à la fin.

gets() saisit une chaîne de caractères au clavier, la saisie étant terminée par ↵.

gets() renvoie accessoirement un pointeur sur la chaîne saisie, le même qui lui a été transmis. On a un exemple de passage de paramètre par adresse pour récupérer un résultat dans chaîne !

☛ Attention: Il n'y a pas de vérification de la longueur de la chaîne saisie par rapport à la longueur déclarée. Il vaut mieux prévoir large !

Exemple:

```
char nom[256];
...
puts( "donnez votre nom:" );
gets( nom );                    /* nom est bien un pointeur sur char */
printf( "bonjour %s !", nom );
```

V) Fonctions de manipulation de chaînes de caractères:

Ces fonctions nécessitent l'inclusion du header <string.h>

Elles permettent toutes sortes d'opérations telles que le calcul de la longueur d'une chaîne

de caractères, la concaténation de deux chaînes, la copie d'une chaîne ou d'une portion de chaîne, la recherche de caractères dans une chaîne, la comparaison de chaînes...

Nous ne citerons que celles qui nous paraissent vraiment utiles.

Rappelons qu'une chaîne de caractère se déclare sous la forme char string[N]; où string est en fait un pointeur char * sur le premier caractère de la chaîne.

Elle se termine par un NUL (ascii \emptyset) et peut comporter au plus N-1 caractères utiles, numérotés à partir de \emptyset .

1) Recopier une chaîne dans une autre:

Syntaxe: strcpy(char *dest, char *srce);
 strncpy(char *dest, char *srce, unsigned nb);

La fonction strcpy() recopie le contenu de la chaîne srce dans la chaîne dest. Rappelons qu'il n'est pas possible d'utiliser le symbole = à cette fin.

La fonction strncpy() recopie les nb premiers caractères de srce dans dest. Le paramètre nb peut très bien être un int, le casting de type étant automatique.

☛ Attention: La fonction strncpy() présente un défaut majeur: si nb est strictement supérieur à la longueur de la chaîne srce, dest est complétée par un NUL final mais si nb est inférieur ou égal à la longueur de la chaîne srce, la chaîne dest n'est pas terminée par un NUL.

Autant dire que dans ce cas dest est inutilisable telle quelle: il faut y ajouter soi-même le NUL final.

Exemples:

```
#define NUL  $\emptyset$ 

char nom[ ] = "Jules César";
char copie[12], prenom[6];
...
strcpy( copie, nom );            /* Recopie nom dans copie */
...
strncpy( prenom, nom, 5 );      /* Recopie Jules dans prenom */
                                 /* mais ne termine pas la chaîne ! */
prenom[5]= NUL;                /* termine la chaîne */
```

2) Concaténer deux chaînes:

La concaténation (du latin catena: chaîne) consiste à réunir deux chaînes en une seule.

Il faut prévoir pour cela une taille suffisante de la chaîne où se fera la concaténation.

Syntaxe: strcat(char *chaîne1, char *chaîne2);
 strncat(char *chaîne1, char *chaîne2, unsigned nb);

Avec strcat(), chaîne1 est complétée par chaîne2 qui est concaténée à la suite.

Avec strncat(), chaîne1 est complétée par les nb premiers caractères de chaîne2 qui sont concaténés à la suite.

Exemples:

```
char prenom[ ]= "Jules";
char nom[ ]= "César";
char empereur[12];
...
strcpy( empereur, prenom );
strcat( empereur, " " );
strcat( empereur, nom );       /* empereur = "Jules César" */
```

3) Comparaison de deux chaînes:

La comparaison de deux chaînes ne peut pas s'effectuer avec == ou !=

Il faut utiliser la fonction strcmp(). Il y a à cela une raison profonde:

Si on écrit:

```
char toto[ ]= "oui";
char truc[ ]= "oui";
...
if ( toto == truc )       /* erreur ! */
{
    ...
}
```

Certes les contenus de toto et truc sont identiques, mais toto et truc sont en fait des **pointeurs** sur les premiers caractères de ces deux chaînes. Or les deux chaînes ne sont pas stockées

à la même adresse, donc le test échouera !

Syntaxe: `int strcmp(char *chaîne1, char *chaîne2);`

Compare chaîne1 et chaîne2 caractère par caractère et retourne \emptyset si chaîne1 et chaîne2 coïncident. Elle renvoie sinon une valeur négative ou positive, selon que chaîne1 précède ou suit chaîne2 dans l'ordre lexicographique des codes ascii.

Des variantes sont `strncmpi()` ou `stricmp()` et `strncmp()`:

Syntaxe: `int strncmpi(char *chaîne1, char *chaîne2);`
`int stricmp(char *chaîne1, char *chaîne2);`

Même chose que `strcmp()` mais ne tiennent pas compte des différences majuscules/minuscules.

Syntaxe: `int strncmp (char *chaîne1, char *chaîne2, unsigned nb);`

Compare les nb premiers caractères des deux chaînes et retourne \emptyset si ces nb caractères coïncident et une valeur non nulle dans le cas contraire.

Exemples:

```
char jc[ ]= "Jules César";
char jv[ ]= "Jules Verne";
int i;
...
i= strcmp( jc, jv );    /* renvoie i < 0 */
...
if ( ! strcmp( jc, jv, 5 ))
    {
    ...
    /* le test réussit car strcmp( ) renvoie 0 */
    ...
    }
```

4) Longueur d'une chaîne:

La fonction `strlen()` calcule la longueur d'une chaîne de caractères:

Syntaxe: `unsigned strlen(char *chaîne);`

Cette fonction renvoie la longueur de la chaîne sans compter le NUL final.

Exemple:

```
unsigned l;  
char mot[ ]= "anticonstitutionnellement";  
...  
l= strlen( mot );      /* devrait donner 26 */
```

5) Recherche de caractères dans une chaîne:

On mettra pour cela à contribution strchr() et strrchr():

Syntaxe: char* strchr(char *chaîne, char c);
 char* strrchr(char *chaîne, char c);

La fonction strchr() recherche le caractère c dans la chaîne et renvoie un pointeur sur la première occurrence trouvée.

Si le caractère c ne figure pas dans la chaîne la fonction renvoie le pointeur symbolique NULL.

Idem pour strrchr() mais en commençant par la fin de la chaîne.

Exemples: Il est plus utile souvent d'avoir l'indice du caractère recherché dans la chaîne que d'avoir un pointeur dessus:

```
char toto[ ]= "Abracadabra";  
char *c_ptr;  
unsigned p=0;  
...  
c_ptr= strchr( toto, 'a' );        /* c_ptr pointe sur le premier a minuscule */  
while ( toto+p != c_ptr ) p++;     /* recalcule l'indice */  
printf( "p= %u", p );            /* affiche la position voulue c'est à dire 3 */
```

⚠ Attention: Se méfier de la portabilité de ces instructions, qui sont fondées sur l'arithmétique des pointeurs.

6) Conversions de caractères:

Syntaxe: char* strlwr(char *chaîne);
 char*strupr(char *chaîne);

strupr() et strlwr() transforment toute une chaîne en majuscules ou en minuscules.

Les caractères non alphabétiques restent inchangés.
Elles renvoient un pointeur sur la chaîne elle même.

Signalons au passage qu'il existe des fonctions transformant un caractère minuscule en majuscule ou vice versa, nommées tolower() et toupper():

Syntaxe: int tolower(int c);
 int toupper(int c);

Les int seront remplacés sans problème par des char dans ces deux fonctions.

Exemples:

```
char annee[ ]= "mdccccxcv";
...
strupr( annee );
printf( annee );                   /* affichage MDCCCCXCV */
...
printf( strupr( annee ));         /* idem en plus court */
/* on profite du fait que strupr( ) renvoie un pointeur sur annee */
```

7) Conversions de nombres en chaînes et vice versa:

Il est parfois nécessaire de convertir une chaîne de caractères contenant un nombre en un nombre effectif (int, long, float) sur lequel on puisse calculer.
Diverses fonctions réalisent ces tâches; elle nécessitent le header <stdlib.h>

Syntaxe: int atoi(char *chaîne)
 long atol(char *chaîne)
 float atof(char *chaîne)

Les fonctions atoi(), atol(), atof() convertissent une chaîne en donnée de type int, long ou float.

La conversion s'arrête dès qu'un caractère inacceptable est rencontré.

Si la chaîne ne peut être convertie, la fonction retourne \emptyset .

Exemples:

```
char nbre [ ]= "3.141592";
char annee[ ]= "1995";
int an;
float pi;
...
pi= atof( nbre );
an= atoi( annee );
```

Inversement on utilisera itoa(), ltoa(), ultoa():

Syntaxe: char* itoa(int val, char *chaîne, int base);
 char* ltoa(long val, char *chaîne, int base);
 char* ultoa(unsigned long, char *chaîne, int base);

Convertit un int ou un long ou un unsigned long en chaîne, selon une base de numération donnée. La base peut aller de 2 à 36. On utilisera en pratique les bases 2, 8, 10 ou 16.

La variable val contient la valeur à convertir, chaîne est la chaîne qui doit contenir le résultat.

Les fonctions retournent un pointeur sur cette même chaîne.

Exemples:

```
char result[256];
long l;
...
l= 126248;
printf( "0x %s", itoa( l, result, 16 ));       /* affichera 0x1ed28 */
```

On pourrait passer le résultat en majuscules avec strupr().

Signalons enfin la fonction sprintf() qui est similaire à la fonction printf()—cf § Fonctions d'entrée/sortie—sauf qu'elle renvoie ses résultats non pas à l'écran, mais dans la chaîne de caractères string.

On obtient dans string exactement ce que printf() aurait affiché à l'écran.

Syntaxe: sprintf(char *string, char *format, var1, ..., varN);

La chaîne de format et la liste de variables fonctionnent comme pour printf().

Cette fonction offre une alternative aux fonctions itoa(), ltoa(), ultoa():

Ainsi itoa(l, result, 16) dans l'exemple précédent peut être remplacé par sprintf(result, "%lx", l)

De plus elle comble une lacune concernant la conversion de floats en chaînes de caractères.

Exemple:

```
float f= 1.23e5;
char toto[128];
...
sprintf( toto, "%f", f);       /* toto= "123000" */
```

Il faut naturellement prévoir dans la chaîne string suffisamment de place pour recevoir

le résultat.

VI) Le mode texte:

Les fonctions de saisie/affichage ne permettent qu'une mise en forme rudimentaire de l'écran.

Il existe une bibliothèque de fonctions plus sophistiquées qui permettent d'y mettre de la couleur, des fenêtres, des cadres, des menus ...

On peut avec cette bibliothèque réaliser l'équivalent de l'éditeur du Turbo C (avec certes pas mal de travail !).

Cette bibliothèque de fonctions ne fait pas partie du standard C défini par l'ANSI (American National Standard Institute) mais est en fait un héritage du TURBO PASCAL.

Elle nécessite l'inclusion du header <conio.h>.

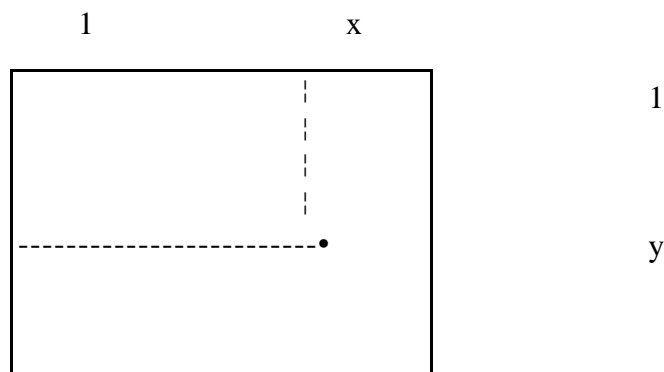
1) Le fenêtrage et la gestion du curseur:

Cette bibliothèque de fonctions repose sur le fenêtrage qui permet de travailler sur une partie de l'écran déclarée fenêtre active.

Avant d'entrer dans les détails voyons ce qu'il en est de l'écran lui même:

l'écran est—selon le mode vidéo actif—composé de 25 lignes de texte de 4Ø ou 8Ø caractères.

Des coordonnées (x,y) désignent chaque caractère à l'écran à partir du coin supérieur gauche:



Chaque caractère à l'écran dispose d'un attribut qui définit la couleur du caractère et du fond sur lequel il est écrit.

La fonction textmode() permet de définir le mode vidéo actif.

Syntaxe: `void textmode(int mode);`

Les modes utilisables sont associés à des constantes pré-définies dans <conio.h> qui sont essentiellement:

BW4Ø	Mode 4Ø colonnes 25 lignes, noir et blanc	(CGA, EGA, VGA)
C4Ø	Mode 4Ø colonnes 25 lignes, couleur	(CGA, EGA, VGA)

MONO	Mode monochrome 80 colonnes 25 lignes.	(Hercules)
BW80	Mode 80 colonnes 25 lignes, noir et blanc	(CGA, EGA, VGA)
C80	Mode 80 colonnes 25 lignes, couleur	(CGA, EGA, VGA)
C4350	Mode 80 colonnes 43 ou 50 lignes, couleur	(EGA, VGA)

Le mode par défaut du PC correspond à C80, ce que nous utiliserons dans tout ce qui suit.

Exemple:

```
textmode( C80 );
/* mode standard par défaut */
```

Un programme gérant l'affichage texte devrait comporter au début une instruction de ce genre.

Le fenêtrage:

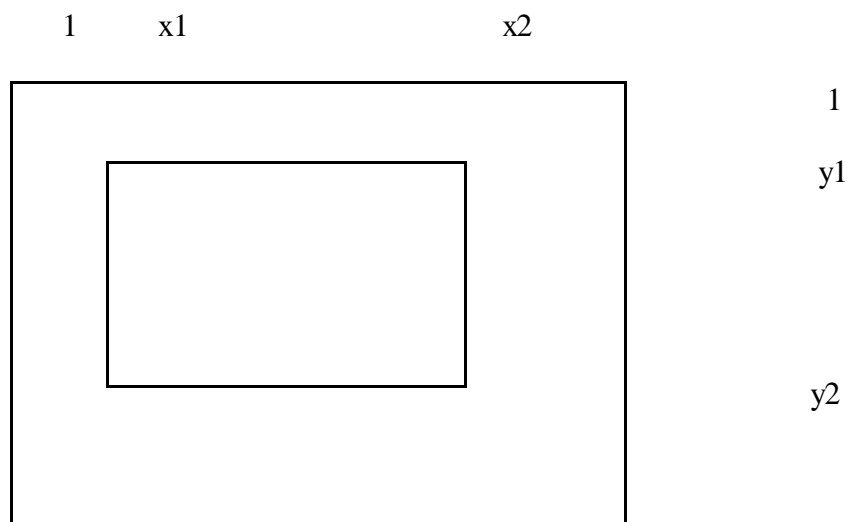
Par défaut la fenêtre active est l'écran tout entier.

On peut restreindre la fenêtre active: dès lors toute les opérations de saisie, affichage, déplacements ultérieurs auront lieu dans cette fenêtre active—on est certain ainsi de ne pas affecter le reste de l'écran.

La fonction clé est window():

Syntaxe: void window(int x1, int y1, int x2, int y2);

Elle déclare active la fenêtre comprenant les lignes y1 à y2 et les colonnes x1 à x2:



Exemple: window(10, 5, 70, 20);

☛ Attention: La fenêtre ainsi définie est purement virtuelle et ne se voit pas sur l'écran !

Remarques: Les coordonnées de la fenêtre sont données par rapport au bord supérieur gauche de l'écran.

Il n'y a à tout instant qu'une seule fenêtre active.
Tout nouvel appel à `window()` remplace donc l'ancienne fenêtre par la nouvelle.

Exemple: `window(1, 1, 80, 25);`
`/* restaure l'écran tout entier en tant que fenêtre active */`

On dispose (dans la fenêtre active) d'un curseur visible à l'écran qui permet d'indiquer où se fera le prochain affichage.
On peut déplacer ce curseur à volonté à l'intérieur de la fenêtre active avec `gotoxy()`:

Syntaxe: `gotoxy(int x, int y);`

Exemple:

```
gotoxy( 5, 2 );
/* place le curseur aux coordonnées 5,2 de la fenêtre active */
```

Remarque: Les coordonnées dans `gotoxy()` sont exprimées par rapport au coin supérieur gauche de la fenêtre active et non de l'écran. (Vous verrez que c'est préférable à l'usage !).

Au cas où à la suite d'un affichage on ne sait plus trop où est le curseur, on peut récupérer sa position dans la fenêtre active avec `wherex()` et `wherey()`:

Syntaxe: `int wherex(void);`
`int wherey(void);`

Exemple:

```
int cur_x, cur_y;
...
cur_x= wherex( );
cur_y= wherey( );    /* récupère position du curseur */
```

Remarque: Chaque instruction `window()` repositionne le curseur dans le coin supérieur gauche de la fenêtre activée, dont les coordonnées sont (1, 1).

On peut à tout moment effacer la fenêtre active avec `clrscr()`:

Syntaxe: `void clrscr(void);`

Exemple:

```
window( 1, 1, 80, 25 );  
clrscr();      /* efface tout l'écran */
```

Remarque: `clrscr()` ramène également le curseur dans le coin supérieur gauche de la fenêtre active.

2) Saisie et affichage:

On va retrouver des fonctions comparables à celles rencontrées auparavant, mais avec quelques spécificités:

- Ces fonctions n'écrivent que dans la fenêtre active:
Donc lorsqu'on arrive au bord droit de la fenêtre il y a passage automatique au bord gauche de la ligne inférieure.
Si on arrive en bas à droite de la fenêtre, la contenu de la fenêtre sera scrollé d'une ligne vers le haut pour libérer une nouvelle ligne vierge.
- Ces fonctions tiennent compte des couleurs et attributs courants, tels que décrits ci-après.
- Ces fonctions écrivent à la position du curseur et déplacent le curseur à chaque écriture.

Saisie et affichage de caractères:

La fonction `putch()` permet d'afficher un caractère, les fonctions `getch()` et `getche()` permettent de lire une touche au clavier; `getch()` n'affiche pas la touche saisie tandis que `getche()` en fait l'écho à l'écran:

Syntaxe: `putch(int c);`
`int getch(void);`
`int getche(void);`

Remarque: On utilisera ces fonctions avec des `char` au lieu des `int` sans problème. `getch()` et `getche()` restent en attente jusqu'à ce qu'une touche soit frappée au clavier. Il n'est pas nécessaire de faire ↵ après avoir frappé cette touche.

On peut tester la frappe d'une touche avec la fonction `kbhit()`. Cela peut éviter de rester en attente d'une touche indéfiniment:

Syntaxe: `int kbhit(void);`

La fonction renvoie \emptyset si aucun caractère n'est disponible et une valeur non nulle s'il y en a un.

Exemples:

```
char chr;
...
if ( kbhit( ) )
    {
        chr= getch( );
        putchar( chr );
    }
```

On pourrait remplacer les deux lignes du bloc par `chr= getch();`

Saisie et affichage de chaînes de caractères:

Les fonctions `cputs()` et `cgets()` assurent l'affichage et la saisie des chaînes dans la fenêtre active et avec les attributs voulus:

Syntaxe: `cputs(char *string);`
 `char *cgets(char *string);`

Le fonctionnement de `cputs()` est limpide mais celui de `cgets()` est un peu particulier: on lui passe l'adresse d'une chaîne de caractères où seront stockés les caractères saisis au clavier, mais ce stockage ne commence en fait qu'au troisième caractère de la chaîne car

`string[\emptyset]` et `string[1]` sont réservés:

le premier indique avant l'appel combien de caractères doivent être saisis au maximum

—en cas de dépassement la saisie s'arrête aussitôt;

le second indique au retour combien de caractères ont été réellement saisis.

De plus la fonction retourne un pointeur très utile sur le début de la chaîne saisie, c'est à dire `&string[2]` ou encore `string+2`.

Remarque: Ces garanties permettent d'éviter que la saisie ne déborde du champ qui lui est réservé à l'écran.

Exemple:

```
char nom[23];
char *nom_ptr;
...
cputs( "donnez votre nom:" );
nom[ $\emptyset$ ] = 21;
nom_ptr = cgets( nom );
```

On demande au maximum 21 caractères sur 23 car 2 sont réservés au début et le `\n` final sera remplacé par le NUL de fin de chaîne.

Saisies et affichages variés:

Les fonctions `cprintf()` et `cscanf()` permettent de réaliser, avec la même syntaxe, les mêmes opérations que `printf()` et `scanf()`, en tenant compte du fenêtrage.

Syntaxe: `cprintf(char *format, var1, ..., varN);`
 `cscanf(char *format, &var);`

Remarque: Pour les saisies, on préférera `cgets()` pour les garanties qu'elle apporte, quitte à traduire la saisie dans le type voulu avec les fonctions `atoi()`, `atol()`, `atof()`... (cf § Manipulations de chaînes).

3) Gestion des couleurs et attributs:

La mémoire vidéo contient pour chaque caractère affiché à l'écran 2 octets contigus: l'un pour le code ascii du caractère, l'autre pour son attribut.

On a donc 8 bits d'attributs, 4 pour le caractère lui-même et 4 pour le fond sur lequel il s'affiche.

Les 4 bits de couleur du texte permettent 16 couleurs, qui sont associées à des constantes pré-définies:

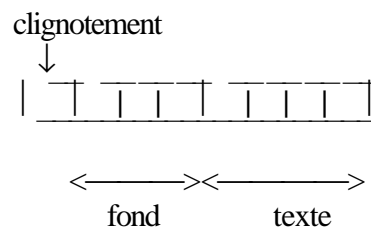
8 couleurs normales: `BLACK, BLUE, GREEN, CYAN,`
 `RED, MAGENTA, BROWN, LIGHTGRAY.`

8 couleurs claires: `DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN,`
 `LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE.`

Les 4 bits de fond se partagent en 3 bits de couleur qui autorisent les 8 couleurs normales

et 1 bits de clignotement correspondant à la constante pré-définie `BLINK`.

Les bits se répartissent comme suit dans l'octet d'attribut:



Les fonctions associées sont `textcolor()`, `textbackground()` et `textattr()`.

Syntaxe: `void textcolor(int color);`
 `void textbackground(int color);`
 `void textattr(int attrib);`

Il suffit de passer à `textcolor()` une des 16 couleurs disponibles pour le texte. On peut ajouter à cette couleur la constante `BLINK` si nécessaire.

et à `textbackground()` une des 8 couleurs utilisables pour le fond pour que les prochains affichages aient lieu dans les couleurs indiquées.

Une variante consiste à définir les deux d'un coup à l'aide de `textattr()`, au prix d'une petite gymnastique: Il faut passer à `textattr()` un octet d'attribut complet qu'il faut fabriquer:

`attribut = COULEUR_TEXTE + 16 * COULEUR_FOND (+ BLINK)`

ou

`attribut = COULEUR_TEXTE + (COULEUR_FOND << 4) (+ BLINK)`

Il faut en effet shifter la couleur de fond de 4 bits vers la gauche.

Exemples:

```
textcolor( YELLOW );
textbackground( BLUE );
cputs( "Jaune sur cyan" );
...
textattr( BLUE + ( CYAN << 4 ));
...
cputs( "Bleu sur cyan" );...
textattr( WHITE + ( RED << 4 ) + BLINK );
cputs( "Blanc sur rouge clignotant" );
```

Remarque: Les couleurs de l'écran par défaut, sous DOS, correspondent à: LIGHTGRAY sur fond BLACK.

On restaurera toujours ces couleurs à la fin du programme.

Astuce: Pour visualiser la fenêtre active dans une couleur particulière, il suffit de l'effacer après avoir défini la couleur de fond.

Exemple:

```
window( 1Ø, 5, 7Ø, 2Ø );
textbackground( BLUE );
clrscr( ); /* la fenêtre apparaît en bleu */
```

4) Sauvegarde et restitution de fenêtre:

On a vu qu'il n'y avait à tout moment qu'une fenêtre active. Toutefois il peut être nécessaire de surcharger temporairement une partie d'une fenêtre par une nouvelle fenêtre,

sans perdre le contenu de l'ancienne (cf exemple ci-après qui simule des messages d'erreur).

Les fonctions `gettextinfo()`, `gettext()` et `puttext()` permettent respectivement de sauvegarder le contexte d'une fenêtre, de sauvegarder le contenu d'une fenêtre, de restaurer le contenu d'une fenêtre précédemment sauvegardée.

Syntaxe: `void gettextinfo(struct text_info *ti_ptr);`
 `gettext(int x1, int y1, int x2, int y2, void *buffer);`
 `puttext(int x1, int y1, int x2, int y2, void *buffer);`

La fonction `gettextinfo()` s'appuie sur la structure pré-définie `text_info` dont voici quelques champs:

```
struct text_info
{
    unsigned char winleft      /* bord gauche */
    unsigned char wintop      /* bord supérieur */
    unsigned char winright    /* bord droit */
    unsigned char winbottom   /* bord inférieur */
    unsigned char curmode     /* mode video en cours */
    unsigned char screenheight /* nb de lignes de l'écran */
    unsigned char screenwidth /* nb de colonnes de l'écran */
    unsigned char cur_x       /* coordonnées du curseur */
    unsigned char cur_y
}
```

Des `unsigned char` ont été déclarés au lieu de `int` pour économiser la place mais on utilisera des `int` sans problème en raison du casting automatique.

Exemple:

```
struct text_info ti;
...
gettextinfo( &ti );
```

La fonction `gettextinfo()` complète alors la structure `ti` avec les renseignements concernant la fenêtre active.

☛ Attention: Une erreur classique et redoutable serait d'écrire

```
struct text_info *ti_ptr;
...
gettextinfo( ti_ptr ); /* incorrect */
```

Car le pointeur ti_ptr ne pointe sur rien de concret et l'écriture des informations se fera à un endroit aléatoire, d'où risque de plantage ultérieur très difficile à diagnostiquer !

La fonction gettext() mémorise quant à elle le contenu de la fenêtre indiquée (en général la fenêtre active) dans un tableau de caractères; la taille du tableau doit être suffisante pour accueillir les caractères et attributs de la fenêtre.

On appliquera la recette de cuisine suivante:

$$\text{taille buffer} \geq 2 * (y2 - y1 + 1) * (x2 - x1 + 1)$$

L'écran tout entier correspond à une taille de 4000 octets.

Exemple:

```
char buffer[2000];
...
gettext( 10, 5, 70, 20, buffer );
```

On peut dès lors que l'on a sauvegardé le contenu de la fenêtre active et son contexte en déclarer une nouvelle en surcharge (cf Exemple général ci-après).

La récupération ultérieure se fait en deux temps:

la restauration du contexte à l'aide des fonctions usuelles et des champs de la structure ti,

puis la restauration du contenu se fait à l'aide de puttext().

Exemple:

```
window( ti.winleft, ti.wintop, ti.winght, ti.winboth );
        /* restaure l'ancienne fenêtre */
gotoxy( ti.curx, ti.cury );
        /* replace le curseur où il était dans cette fenêtre */
textattr( ti.attribut );
        /* restaure attribut courant */
puttext( 10, 5, 70, 20, buffer );
```

Exemple général: Voici un exemple qui illustre toutes ces manipulations en affichant un pseudo message d'erreur en surcharge à l'écran, avec sauvegarde de la zone surchargée.

```

#include <stdio.h>
#include <conio.h>

#define BEEP  0x07          /* code ascii Beep sonore */

typedef char stg[80];      /* définit nouveau type */
typedef char encart[4000]; /* pour sauvegarder une partie de
l'écran */

void erreur(int);          /* déclarations de fonctions */
void cadre_simple(int, int, int, int);

void cadre_simple(int X1,int Y1,int X2,int Y2) /* trace un cadre */
{
    int I;

    gotoxy(X1,Y1);
    putchar('┌');
    for (I= X1+1 ; I<X2 ; I++) putchar('-');
    putchar('┐');
    for (I=Y1+1 ; I<Y2 ; I++)
    {
        gotoxy(X1,I); putchar('├');
        gotoxy(X2,I); putchar('┤');
    }
    gotoxy(X1,Y2);
    putchar('└');
    for (I=X1+1 ; I<X2 ; I++) putchar('-');
    putchar('┘');
}

```

```

void erreur(int num) /* affiche message d'erreur dans une fenetre */
{
    struct text_info tr; /* structure pré-définie */
    encart alerte;
    stg message[]= {" Erreur matérielle critique. ",
                    " Erreur d'opération sur un fichier. ",
                    " Répertoire non valide. ",
                    " Paramètres incorrects. ",
                    "Erreur dans le fichier configuration."};

    gettextinfo(&tr); /* mémorise contexte fenetre */
    gettext(20,11,61,14,alerte); /* sauvegarde contenu fenetre */

    window(21,12,61,14);
    textbackground(BLACK);
    clrscr();
    window(20,11,60,13);
    textbackground(RED);
    clrscr();
    window(20,11,61,14);
    textattr(0x10*RED+YELLOW);
    cadre_simple(1,1,41,3);

    putch(BEEP);
    gotoxy(18,1); cprintf("Erreur");
    gotoxy(3,2); cprintf(message[num]);
    gotoxy(12,3); cprintf("<Presser une Touche>");
    putch(BEEP);
    getch();

    puttext(20,11,61,14,alerte); /* restaure contenu fenetre */
    window(tr.winleft,tr.wintop,tr.winright,tr.winbottom);
    gotoxy(tr.curx, tr.cury);
    textattr(tr.attribute); /* restaure contexte fenetre */
}

void main()
{
    int n;

    textmode(C80);
    window(1,1,80,25);
    clrscr();
    window(1,1,80,24);
    textcolor(WHITE);
    textbackground(BLUE);
    clrscr();
    window(1,1,80,25);
    cadre_simple(1,1,80,24);

    gotoxy(5,5);
    cprintf("Entrer un numéro de 0 ... 5: ");
    cscanf("%d", &n);
    if ((n > 0) && (n <= 5)) erreur(n-1);

    window(1,1,80,25); /* laissons les lieux propres en partant */
    textcolor(LIGHTGRAY);
    textbackground(BLACK);
    clrscr();
}

```


Remarque générale: Un problème que l'on rencontre est de ne pas pouvoir écrire dans le coin inférieur droit de la fenêtre active car le passage du curseur à la ligne suivante fait scroller la fenêtre vers le haut. Pour y remédier on déclarera en général une fenêtre qui fait une ligne de plus qu'il n'est nécessaire et on évitera d'écrire sur cette dernière ligne (cf Exemple général ci-dessus).

VII) Le mode graphique:

Le passage en mode graphique permet de gérer l'écran point par point, donc de faire toutes sortes de constructions graphiques en couleur (tout en gardant la possibilité d'afficher des chaînes de texte).

Une bibliothèque séparée, `graphics.lib`, est dédiée aux fonctions graphiques.

Elle nécessite l'inclusion du header `<graphics.h>`.

Les fonctions de cette bibliothèque ne font pas partie du standard C ANSI, mais sont plus ou moins héritées du TURBO PASCAL.

1) Le principe du mode graphique:

La gestion du mode graphique est un peu compliquée par l'évolution des moniteurs des PC:

on a connu successivement des moniteurs CGA, HERCULES, EGA, VGA .

(pour ne citer que les principaux).

Or ces moniteurs ont des possibilités qui n'ont fait que s'améliorer, tant au niveau de la définition (nombre de points par ligne et par colonne) que du nombre de couleurs disponibles.

Les meilleurs modes de ces écrans sont :

Type	Mode BIOS	définition	couleurs
CGA:	ØxØØ	32Øx2ØØ	4
	ØxØ6	64Øx2ØØ	2
HERCULES:	ØxØ7	72Øx348	2
EGA:	Øx1Ø	64Øx35Ø	16
VGA:	Øx12	64Øx48Ø	16
	Øx13	32Øx2ØØ	256

Le nombre de couleurs augmentant au détriment de la définition car la taille de la mémoire vidéo qui stocke les données affichées à l'écran est fixe !

La bibliothèque de fonctions disponible ne gère que les modes ayant 16 couleurs au plus.

Elle s'appuie sur des drivers livrés avec le compilateur qui assurent l'interface entre la bibliothèque de fonctions et le BIOS de la carte graphique installée dans le PC.

Ceci afin que la programmation reste assez indépendante de la carte utilisée (mis à part la définition et le nombre de couleurs disponibles).

Les drivers, situés dans la répertoire BGI sont: CGA.BGI, HERC.BGI, EGAVGA.BGI

Les drivers BGI (BIOS Graphic Interface) nécessaires doivent accompagner le programme .EXE, une fois le programme C compilé et linké.

☛ Attention: Nous allons dans tout ce qui suit nous intéresser au mode VGA

(640x480) qui dispose de 640x480 points en 16 couleurs.

2) Initialisation du mode graphique:

● Attention: Pour exploiter le mode graphique il est nécessaire de cocher la case <graphics lib> dans la rubrique linker→libraries du menu options.

L'écran passe en mode graphique lors de l'appel à initgraph():

Syntaxe: void initgraph(int far *drv_ptr, int far *mod_ptr, char far *chemin);

- drv_ptr est l'adresse d'un entier représentant le driver vidéo voulu —correspondant à la carte vidéo installée.

Les valeurs sont représentées par des constantes pré-définies:
CGA, HERCMONO, EGA, VGA.

La constante DETECT permet de demander la détermination automatique de la carte vidéo installée. Dans ce cas les variables pointées par drv_ptr et mod_ptr sont renseignées par la fonction au retour.

- mod_ptr est l'adresse d'un entier contenant le mode vidéo souhaité sur la carte.

Les constantes suivantes sont disponibles:

CGAHI	640x200	2 couleurs
HERCMONOH	720x348	2 couleurs
EGAHI	640x350	16 couleurs
VGAHI	640x480	16 couleurs

- chemin est une chaîne de caractères désignant le répertoire où se trouvent les drivers BGI.

Si chemin= "" les drivers sont recherchés dans le répertoire du programme.

Exemple:

```
/* ce que nous utiliserons dans la suite */
#include <graphics.h>.
int video_drv= VGA;
int video_mod= VGAHI;
...
initgraph( &video_drv, &video_mod, "" );
```

Remarque: En cas d'erreur les fonctions graphiques ne renvoient pas de résultat particulier mais positionnent une variable globale à une valeur indiquant l'erreur.

On peut diagnostiquer une erreur en faisant appel à graphresult() et grapherrormsg()

Syntaxe: int graphresult();
 char far * grapherrormsg(int errnum);

graphresult() retourne un numéro d'erreur, qui, transmis à grapherrormsg() permet d'afficher un message d'erreur en clair.

La fonction closegraph() met fin au mode graphique et ramène l'écran en mode texte. Cela est impératif à la fin du programme.

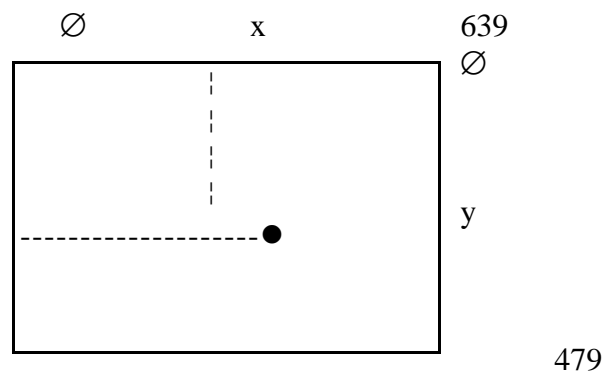
Syntaxe: closegraph();

3) Fenêtrage et gestion du curseur:

La mode graphique présente quelques analogies avec le mode texte dans la mesure où l'on travaille dans une fenêtre graphique déterminée par ses coordonnées à l'écran, et que l'on dispose d'un curseur graphique—invisible à l'écran—qui détermine l'endroit où se fera le prochain tracé.

Les coordonnées:

L'écran est composé de pixels (points de couleur) agencés en lignes et colonnes, numérotées à partir du coin supérieur gauche de l'écran, en commençant à Ø.



Les dimensions de l'écran dans le mode vidéo choisi seront accessibles via getmaxx() et getmaxy().

Syntaxe: int getmaxx();
 int getmaxy();

Ces fonctions renvoient les valeurs maximales en x et y , c'est à dire les dimensions de l'écran en pixels moins un puisqu'on commence à Ø !

Exemple:

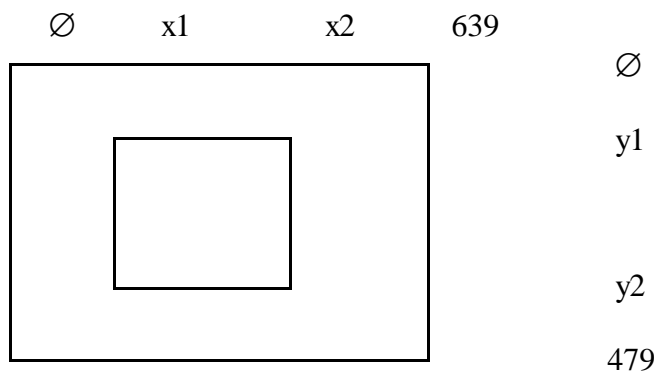
```
int L, H,  
...  
L = getmaxx() +1;  
H = getmaxy() +1;  
printf( "la dimension de l'écran est %d x %d", L, H );
```

On déclare la
fenêtre

active—dans laquelle se feront les tracés à venir—avec setviewport().

Syntaxe: setviewport(int x1, int y1, int x2, int y2, int clip);

Les coordonnées de la fenêtre seront données par (x1, y1) et (x2, y2):



Le paramètre clip sert à gérer les effets de bord.

Un tracé dépassant la fenêtre active sera ou ne sera pas prolongé en dehors de celle-ci.

Les valeurs suivantes déterminent le comportement:

- clip = \emptyset → tracés prolongés
- clip = 1 → traces limités à la fenêtre

La fonction clearviewport() efface la fenêtre active et y replace le curseur en haut à gauche.

Syntaxe: clearviewport();

Exemple: setviewport(2 $\emptyset\emptyset$, 1 $\emptyset\emptyset$, 44 \emptyset , 38 \emptyset , 1);
 clearviewport();

Remarque: cleardevice() efface l'écran tout entier.

Le curseur graphique se déplace dans la fenêtre active et ses coordonnées sont relatives

au coin supérieur gauche de cette fenêtre.

Il se déplace au fil des tracés, mais on peut aussi le positionner à un endroit précis pour le prochain tracé avec moveto() et moverel()

Syntaxe: `moveto(int x, int y);`
 `moverel(int dx, int dy);`

`moveto()` positionne le curseur en (x, y) dans la fenêtre active tandis que `moverel()` effectue un déplacement relatif en ajoutant dx et dy aux coordonnées actuelles du curseur.

Les coordonnées curseur commencent à (\emptyset , \emptyset).

On peut à tout moment récupérer la position du curseur à l'aide de `getx()` et `gety()`

Syntaxe: `int getx();`
 `int gety();`

4) Gestion des couleurs:

Chaque pixel à l'écran a une couleur parmi 16 (en mode VGAHI) qui peut être définie individuellement ou au cours d'un tracé.

Le fond de l'écran a une couleur (parmi 16) qui est la même partout.

Le fait de changer la couleur de fond affecte donc l'écran tout entier.

Les couleurs sont les mêmes qu'en mode texte: BLACK, BLUE, ..., WHITE (cf § Mode texte).

La fonction `setbkcolor()` permet de définir la couleur du fond, et `getbkcolor()` permet de la récupérer.

Syntaxe: `setbkcolor(int color);`
 `int getbkcolor();`

La fonction `setcolor()` détermine la couleur des prochains tracés et `getcolor()` sert à récupérer la dernière couleur activée.

Syntaxe: `setcolor(int color);`
 `int getcolor();`

Exemple:

```
setbkcolor( BLUE ); /* fond bleu */
setcolor( YELLOW ); /* tracés jaunes */
```

La fonction `putpixel()` permet d'allumer un pixel dans une couleur donnée , tandis que `getpixel()` permet de retrouver la couleur d'un pixel à l'écran.

Syntaxe: `putpixel(int x, int y, int color);`
 `int getpixel(int x, int y);`

Exemple:

```
int color;
...
putpixel( 100, 50, GREEN );      /* allume pixel ( 100, 50 ) en vert */
color = getpixel(0, 0 );          /* lit couleur du premier pixel de la fenêtre */
```

5) Les tracés:

Diverses fonctions permettent de tracer des lignes, des ellipses, des cercles, des arcs ou secteurs d'ellipses—donc de cercles, des rectangles, et des polygones.

Un tel tracé se fait dans la couleur définie par `setcolor()` et avec une épaisseur et un style de ligne que l'on peut moduler avec `setlinestyle()`:

Syntaxe: `setlinestyle(int style, unsigned motif, int épaisseur);`

Le paramètre `style` définit le style conformément aux constantes pré-définies.

<code>SOLID_LINE</code>	trait plein
<code>DOTTED_LINE</code>	trait pointillé court
<code>CENTER_LINE</code>	trait mixte
<code>DASHED_LINE</code>	trait pointillé long
<code>USERBIT_LINE</code>	trait personnalisé

Le paramètre `motif` ne sert qu'avec l'option `USERBIT_LINE` pour définir un tracé personnalisé: les 16 bits de motif représentent 16 pixels consécutif du tracé, qui seront ou non allumés selon que le bit est à 1 ou à 0.

Enfin `epaisseur` est l'épaisseur de la ligne en pixel.

Remarque: La fonction `getlinesettings()` permet au besoin de récupérer les caractéristiques actuelles de tracé. (cf Aide à ce sujet)

a) tracés de lignes:

Les fonctions `line()`, `linto()` et `linerel()` sont disponibles.

Syntaxe: `line(int x1, int y1, int x2, int y2);`
`linto(int x, int y);`
`linerel(int dx, int dy);`

`line()` trace une ligne de (`x1`, `y1`) à (`x2`, `y2`).
Elle ne modifie pas la position du curseur.

Elle trace naturellement un rectangle dont les coordonnées sont celles indiquées.
On pourra s'en servir pour tracer des cadres (simples, doubles, triples, ...) autour des fenêtres.

Pour ce qui est des polygones cela se complique (Il va falloir réfléchir un peu !).

La fonction drawpoly() se charge de tracer un polygone si on lui donne le nombre de sommets et leurs coordonnées

Syntaxe: drawpoly(int nb, int *poly_ptr);

nb est le nombre de sommets, poly_ptr est l'adresse d'un tableau de nb paires d'entiers représentant les coordonnées des sommets.

Un exemple vaut mieux qu'un laborieux discours:

Exemple:

```
int triangle[ 8 ]= {100, 100, 540, 100, 320, 380, 100, 100 };
...
drawpoly( 4, triangle );
```

Trace un triangle joignant les points (100, 100), (540, 100), (320, 380).

Remarques:

- triangle est l'adresse du début du tableau donc il serait faux de mettre un & devant...

- n'allez pas croire qu'un triangle ait 4 sommets, mais si on veut que le tracé soit complet il faut revenir au point de départ !

6) Le remplissage:

Diverses fonctions permettent de dessiner des secteurs de cercle, d'ellipse, des polygones, cernés d'un trait et remplis d'un motif donné.

Le trait est toujours défini par setlinestyle() et le motif de remplissage est quant à lui décrit par setfillpattern() et setfillstyle().

Syntaxe: setfillpattern(char far *pattern_ptr, int color)
 setfillstyle(int motif, int color)

setfillstyle() reçoit une constante motif pré-définie parmi les suivantes:

EMPTY_FILL	→	motif vide
SOLID_FILL	→	motif plein

LINE_FILL	→	lignes
LTSLASH_FILL	→	hachures
BKSLASH_FILL	→	hachures
HATCH_FILL		
INTERLEAVE_FILL		
WIDEDOT_FILL	→	pointillés
CLOSEDOT_FILL		
USER_FILL	→	motif personnalisé

et une couleur parmi les 16 disponibles.

Si on utilise USER_FILL il faut faire au préalable un appel à setfillpattern():

setfillpattern() reçoit en fait l'adresse d'un tableau de 8 octets correspondant à un motif de 8 pixels sur 8:

	Tableau:	Motif:
octet 0	0 1 0 1 1 0 1 0	● ● ● ●
octet 1	0 0 1 0 1 1 1 1	● ● ● ● ●
octet 2	1 1 0 0 1 0 1 1	● ● ● ● ●
octet 3	1 1 1 1 1 1 1 1	● ● ● ● ● ● ● ●
octet 4	0 0 1 1 0 1 0 1	● ● ● ● ●
octet 5	1 1 0 0 0 0 1 1	● ● ● ● ●
octet 6	1 0 0 0 1 1 1 1	● ● ● ● ● ● ● ●
octet 7	0 1 0 1 0 1 0 1	● ● ● ● ●

a) Remplissage de cercles, ellipses:

Les fonctions utiles sont sector(), pieslice() et fillellipse()

Syntaxe: pieslice(int x, int y, int alpha, int beta, int r);
 fillellipse(int x, int y, int a, int b);
 sector(int x, int y, int alpha, int beta, int a, int b);

Les paramètres sont les mêmes que pour les fonctions de tracé.

b) Remplissage de polygones:

La fonction fillpoly() assure ce rôle.

Syntaxe: fillpoly(int nb, int far *poly_ptr);

poly_ptr est toujours l'adresse d'un tableau de nb paires d'entiers désignant les sommets du polygone.

Exemple:

```

char pattern[ 8 ] = { 0x15, 0x37, 0x22, 0xA0, 0xF5, 0x4D, 0x87, 0x32 };

setcolor(YELLOW);
setlinestyle( SOLID_LINE, 2 );
setfillstyle( SOLID_FILL, RED );
pieslice( 100, 100, 0, 180, 50 );          /* remplit un demi cercle en rouge
                                             bordé de jaune */
...
setfillpattern( pattern, CYAN );
fillellipse( 500, 400, 100, 50 );          /* remplit une ellipse bordée de jaune
                                             d'un motif CYAN très aléatoire */

```

c) Remplissage d'un contour:

Citons enfin une technique de remplissage intéressante, basée sur floodfill(), qui remplit du motif courant toute la partie de l'écran délimitée à partir d'un point donné par une frontière de couleur donnée: c'est à dire que tout autour du point indiqué, on colorie tous les pixels que l'on trouve tant qu'ils n'ont pas la couleur indiquée pour la frontière.

Syntaxe: `floodfill(int x, int y, int couleur);`

(x, y) est le point de départ et couleur est la couleur de la frontière.

- Remarques:
- Si le point (x, y) est dans le contour de couleur on peindra effectivement l'intérieur de ce contour. S'il est en dehors on peindra l'extérieur du contour de couleur.
 - Si le contour de couleur n'est pas fermé on risque fort de peindre presque toute la fenêtre ou presque tout l'écran !
 - On utilisera de préférence `fillpoly()` dans les cas simples où elle peut s'appliquer.

d) Statistiques:

Pour les statistiques on peut faire des camemberts avec pieslice().
Il est possible de faire également des diagrammes en bâtons avec bar() et bar3D().
(Consulter l'aide à ce sujet)

7) Affichage de texte en mode graphique:

On peut afficher des chaînes de caractères en mode graphique, à la position du curseur ou à une position donnée à l'aide de outtext() et outtextxy().

Syntaxe: outtext(char far *string);
 outtextxy(int x, int y, char far *string);

Remarque: Le style et la taille des caractères ainsi que le sens d'écriture (horizontal ou vertical) peuvent être modifiés avec settextstyle() et autres fonctions associées
(Consulter l'aide pour plus de détails).

Exemple:

```
moveto( 100, 10 );  
outtext( "Figure Principale" );        /* ou encore */  
outtextxy( 100, 10, "Figure Principale" );
```

Remarque: Il n'existe pas de fonction de saisie de texte ou de variables en mode graphique.
(Il faudra l'écrire soi-même !) On ne peut pas utiliser scanf().

8) Sauvegarde et restauration d'une partie de l'écran:

Comme en mode texte, il est possible de sauvegarder le contexte et le contenu d'une fenêtre,
de l'écraser par une nouvelle fenêtre et de restaurer l'ancienne ultérieurement.
Les fonctions nécessaires sont getviewsettings(), imagesize(), putimage() et getimage():

Syntaxe: getviewsettings(struct viewporttype far *view_ptr);
 unsigned imagesize(int left, int top, int right, int bottom);
 putimage(int left, int top, void far *data_ptr, int op);
 getimage(int left, int top, void far *data_ptr);

La structure viewporttype est pré-définie et contient les champs suivants:

```
struct viewporttype:  
    {  
        int left;  
        int top;  
        int right;  
        int bottom;  
        int clip;  
    }
```

Elle sert à la fonction `getviewsettings()` à mémoriser les coordonnées de la fenêtre active.

—cf fonction `setviewport()`—

La fonction `imagesize()` calcule pour nous le nombre d'octets nécessaires pour stocker

le contenu d'une fenêtre à l'écran.

La valeur retournée peut servir à une allocation dynamique (cf § Allocation dynamique)

ou, lors de la mise au point, à calculer la taille du tableau à déclarer.

☛ Attention La taille de l'image ne doit pas dépasser 64 KO.

La fonction `putimage()` mémorise dans le tableau pointé par `data_ptr` le contenu de la fenêtre indiquée. Les deux premiers mots contiennent les dimensions de l'image en pixels.

La fonction `getimage()` restaure à partir des coordonnées top et left l'image mémorisée

dans un tableau par `putimage()` .

Exemple général:

```
#include <stdio.h>
#include <conio.h>
#include <graphics.h>

typedef char stg[80];

void erreur(int);
void cadre_double(int, int, int, int);
void fillrectangle(int, int, int, int, int);

int video_drv= VGA, video_mod= VGAHI, H_size, V_size;
char buffer[20000];

void fillrectangle(int X1, int Y1, int X2, int Y2, int color)
{
    int rect[10];

    /* définit un rectangle fermé */
    rect[0]= X1; rect[1]= Y1;
    rect[2]= X2; rect[3]= Y1;
    rect[4]= X2; rect[5]= Y2;
    rect[6]= X1; rect[7]= Y2;
    rect[8]= X1; rect[9]= Y1;

    setfillstyle(SOLID_FILL, color);
    fillpoly(5, rect);
}

void cadre_double(int X1, int Y1, int X2, int Y2)
{
    setlinestyle(SOLID_LINE, 0, 1);

    rectangle(X1, Y1, X2, Y2);
    rectangle(X1+2, Y1+2, X2-2, Y2-2);
}
```

```

void erreur(int num)
{
    struct viewporttype view;
    stg message[]= {" Erreur matérielle critique. ",
                   " Erreur d'opération sur un fichier. ",
                   " Répertoire non valide. ",
                   " Paramètres incorrects. ",
                   "Erreur dans le fichier configuration."};

    getviewsettings(&view);          /* mémorise fenêtre */
    getimage(160,200,480,250, buffer);

    setviewport(160,200,480,250, 1); /* définit nouvelle fenêtre */
    clearviewport();                /* efface cette fenêtre */

    fillrectangle(0,0,319,49, RED);
    setcolor(WHITE);
    cadre_double(0, 0, 319, 49);

    moveto(15,20);
    outtext(message[num]);

    getch();

    setviewport(view.left, view.top, view.right, view.bottom, view.clip);
    putimage(160,200, buffer, COPY_PUT); /* restaure fenêtre */
}

void main()
{
    int n;

    initgraph(&video_drv, &video_mod, "");
    /* initialise mode VGA 640x480 */

    setbkcolor(BLUE);
    cleardevice();                /* efface écran en bleu */

    H_size= getmaxx()+1;          /* dimension horizontale écran */
    V_size= getmaxy()+1;          /* dimension verticale écran */

    setcolor(YELLOW);
    cadre_double(0, 0, H_size-1, V_size-1); /* trace un cadre double */

    setfillstyle(INTERLEAVE_FILL, CYAN);
    fillellipse(320,230, 200,100); /* trace ellipse pleine */

    setcolor(WHITE);
    moveto(64,48);                /* d,place curseur graphique */
    outtext("Donnez un num,ro de 1 ... 5: "); /* affiche message */

    n= getch()-48; /* transforme '0' en 0 etc. */
    if ((n > 0) && (n <= 5)) erreur(n-1);
    getch();

    closegraph(); /* fin du mode graphique */
}

```


VIII) Fonctions de manipulation de fichiers et répertoires

1) Opérations sur les fichiers:

Un fichier est un ensemble de données (octets) stockées sur disque dur, disquette, CD ROM...

Les données peuvent être organisées en lignes de texte (fichier texte) ou purement binaires (fichier binaire).

Exemple: Les fichiers .TXT en .DOC sont en général des fichiers texte, les autres sont en général des fichiers binaires.

Les opérations de base sur un fichier sont:

- création ou ouverture
- déplacement dans le fichier
- lecture, écriture
- fermeture

C'est lors de l'ouverture ou de la création que l'on précise si on travaille sur un fichier texte ou binaire.

Sous MS-DOS un fichier est identifié par un nom de 8 lettres suivi d'une extension de 3 lettres, séparés par un point.

En plus de ce nom importe le nom du répertoire et le lecteur (A:, B:, C:, ...) dans lequel est situé le fichier.

Exemples: TOTO.TXT
C:\USR\DATA.CFG

Sont des noms de fichiers valides.

Il existe en C deux bibliothèques d'accès aux fichiers:

- Une bibliothèque de bas niveau nécessitant les headers <io.h> et <fcntl.h>.
- Une bibliothèque de haut niveau nécessitant <stdio.h>.

Les fonctions de haut niveau permettent des opérations plus sophistiquées que les autres.

Elles s'appuient en fait—sans qu'on le sache—sur les fonctions de bas niveau.

Les fonctions de bas niveau sont appelées ainsi car elles sont plus proches du système d'exploitation, et exploitent directement les données sur disque tandis que les fonctions de haut niveau utilisent des **tampons** par lesquels transitent les données lors de la lecture ou de l'écriture.

L'intérêt de ces tampons est de limiter le nombre d'accès au disque: en cas de lecture

ou écriture de plusieurs petits paquets de données, ceux ci sont regroupés en un seul paquet dans le tampon et c'est le tampon qui fera l'objet d'une seule opération sur disque.

a) Les fonctions de bas niveau:

Les fonctions de bas niveau manipulent un fichier à l'aide d'un **handle** (descripteur) qui est associé au fichier lors de l'ouverture ou de la création et qui servira à tous les accès ultérieurs jusqu'à la fermeture du fichier.

Le nom du fichier ne sert qu'à obtenir ce handle de la part du système d'exploitation.

Ouverture et fermeture d'un fichier:

L'ouverture d'un fichier passe par open() ou creat() et sa fermeture par close().

Syntaxe: int open(char *nom_fichier, int omode, int amode);
 int creat(char *nom_fichier, int amode);
 int close(int handle);

La fonction open() reçoit une chaîne contenant le nom du fichier—avec éventuellement

un nom d'unité et un chemin d'accès.

Elle reçoit aussi le mode d'ouverture voulu par le biais de constantes pré-définies que l'on peut combiner par | dans omode:

O_TEXT ou O_BINARY

Selon le mode Texte ou Binaire souhaité.

O_RDONLY, O_WRONLY, O_RDWR,

Selon que l'accès se fera en lecture, écriture ou les deux.

O_TRUNC Pour vider le fichier (dangereux !)

O_CREAT Pour en imposer la création s'il n'existe pas.

O_APPEND Pour se positionner directement à la fin du fichier après l'ouverture.

Le paramètre amode est important dans la fonction open():

On dispose des constantes.

S_IREAD ou S_IWRITE dans le header <sys/stat.h>

pour indiquer si après sa fermeture les droits d'accès au fichier seront en lecture seule ou en lecture/écriture.

La fonction open() retourne en échange de tout cela un entier baptisé handle (descripteur) délivré par le système d'exploitation et qui servira lors des accès ultérieurs à ce fichier.

creat() est un raccourci parfois utilisé pour la création d'un fichier, avec remise à zéro de celui-ci s'il existe déjà.

```
creat( char *nom_fichier, amode );  
équivalent à  
open(char *nom_fichier, O_CREAT | O_TRUNC, amode );
```

La fonction `creat()` renvoie de même un handle pour manipuler le fichier.

Remarque: En cas d'erreur `open()` et `creat()` renvoient -1. Il est préférable de vérifier

cette valeur de retour afin de ne pas continuer des opérations sur un fichier inexistant !

La fonction `close()` reçoit un handle. Elle ferme le fichier associé à ce handle par

`open()`

ou `creat()`. Elle renvoie -1 en cas d'erreur.

Exemples:

```
int hd1, hd2;  
...  
hd1= creat( "TOTO", S_IWRITE );  
hd2= open( "C:\\WORK\\FILE.DAT", O_CREAT | O_TEXT | O_WRONLY,  
S_IWRITE );  
...  
close( hd1 );  
close( hd2 );
```

☛ **Attention:** Dans les chaînes de caractères contenant les noms de fichiers, les antislashes `\` doivent être redoublés en `\\`, car l'antislash seul est le préfixe des séquences d'échappement (cf § Chaînes de caractères).

Déplacement dans un fichier:

On peut se déplacer dans un fichier ouvert grâce à un curseur (je n'ose pas dire pointeur !)

de lecture / écriture qui indique l'endroit où se fera la prochaine opération, par rapport au début du fichier.

Le curseur est par défaut placé au début du fichier lors de l'ouverture, sauf dans le mode d'ouverture `O_APPEND` où il est placé à la fin. Une opération d'écriture ne peut se faire

qu'à la fin d'un fichier, mais la lecture peut avoir lieu n'importe où dans le fichier, en déplaçant ce curseur.

Les fonctions disponibles pour cela sont: `lseek()`, `tell()`, `eof()`, `filelength()`.

Syntaxe: long `lseek(int handle, long dep, int org);`
 long `tell(int handle);`
 int `eof(int handle);`
 long `filelength(int handle);`

La fonction `lseek()` reçoit bien sûr le handle du fichier concerné, mais surtout un long int

qui détermine de combien d'octets le curseur sera déplacé, et une constante org qui indique à partir de quelle origine se fait le déplacement:

SEEK_SET	déplacement absolu en avant depuis le début du fichier.
SEEK_CUR	déplacement relatif en avant par rapport à la position courante.
SEEK_END	déplacement absolu en arrière par rapport à la fin du fichier.

Inversement, la fonction `tell()` renvoie la position actuelle du curseur, par rapport au début du fichier.

Par ailleurs `eof()` détermine si la fin du fichier a été atteinte, après un déplacement ou une opération de lecture.

La valeur `TRUE (1)` signifie que la fin a été atteinte, la valeur `FALSE (0)` signifie que non.

La fonction `filelength()` retourne la taille en octets du fichier associé au handle indiqué.

Lecture et écriture dans un fichier:

Les incontournables fonctions `read()` et `write()` sont là pour cela:

Syntaxe: `int read(int handle, char *zone, unsigned nb);`
 `write(int handle, char *zone, unsigned nb);`

La lecture avec `read()` a lieu à la position courante du curseur, lequel est déplacé vers l'avant du nombre d'octets lus.

L'écriture ne peut—pour des raisons techniques—avoir lieu qu'à la fin du fichier.

En plus du handle incontournable ces fonctions reçoivent l'adresse d'une zone (un tableau de caractères en général) où les octets à transférer seront lus ou écrits—le nombre d'octets à transférer est donné par l'entier `nb`.

La fonction renvoie le nombre d'octets effectivement lus ou écrits, ou `-1` en cas d'erreur.

Remarque: Si le fichier a été ouvert en mode Texte, la marque de fin de ligne `CR (\n)`

est convertie en `CR/LF (\n \r)` lors de l'écriture et vice versa lors de la lecture.

Dans ce même mode, le caractère spécial `EOF (^Z)` est interprété en lecture comme une marque de fin de fichier—même si la fin physique du fichier n'est pas atteinte.

Lors de la fermeture d'un fichier en mode Texte, une telle marque de fin de fichier `EOF`

sera automatiquement écrite.

Exemple:

```
int hd, len;
char texte[80], copie[80];
...
hd= open( "ESSAI.TXT", O_CREAT | O_TEXT, S_IWRITE );
        /* création en mode Texte */
if( hd == -1 ) printf( "erreur d'ouverture\n" );
else
    {
        strcpy( texte, "ceci est un essai \n" );
        len= strlen( texte );
        write( hd, texte, len ); /* écriture texte avec conversion de \n en CR/LF */
        lseek( hd, 0, SEEK_SET ); /* retour au début du fichier */
        read( hd, copie, len ), /* relit le fichier */
        printf( copie ); /* devrait afficher le message texte */
        close( hd ), /* fermeture indispensable */
    }
```

b) Les fonctions de haut niveau:

Les fonctions de haut niveau manipulent cette fois un fichier à l'aide d'un **pointeur** sur une structure de type pré-défini FILE. Ce pointeur servira à tous les accès ultérieurs jusqu'à la fermeture du fichier.

Le nom du fichier ne sert qu'à compléter la structure FILE associée et à obtenir le pointeur sur cette structure.

Il n'est pas nécessaire de connaître le contenu de la structure FILE pour utiliser ces fonctions ! Elles nécessitent les headers <stdio.h> et parfois <sys/stat.h>

Ouverture et fermeture d'un fichier:

Deux fonctions remplissent ces rôles, fopen() et fclose().

Syntaxe: FILE * fopen(char *nom_fichier, char *fmode);
 int fclose(FILE *f_ptr);

La fonction d'ouverture fopen() reçoit une chaîne de caractères représentant le nom du fichier, avec si nécessaire une unité et un chemin d'accès.

On précise de plus le mode d'ouverture du fichier à l'aide d'une chaîne de caractères fmode:

"r" ou "r+"	accès en lecture
"w"	accès en écriture
"a"	accès en ajout c'est à dire écriture à la fin du fichier
"w+"	création d'un nouveau fichier en lecture/écriture
"a+"	accès en ajout avec création si nécessaire
"b"	ouverture en mode Binaire

"t" ouverture en mode Texte

En fait fmode est une concaténation de la forme "rt" ou "w+b"...

fopen() retourne un pointeur sur une structure de type FILE, qui est gérée par fopen().

On ne déclarera pas cette structure sans le programme mais seulement un pointeur dessus,

qui recevra le résultat de fopen().

Si le pointeur est NULL, l'ouverture ou la création a échoué. On vérifiera cette valeur de retour pour éviter des opérations ultérieures sur un fichier fantôme.

La fonction fclose() ferme le fichier associé au pointeur f_ptr transmis par fopen(). Elle assure avant la fermeture la synchronisation des données, c'est à dire l'écriture des données encore en attente dans le tampon d'écriture.

Exemple:

```
FILE *f_ptr;
...
f_ptr = fopen( "C:\\TOTO.TXT", "rt" );
...
fclose( f_ptr );
```

☛ Attention: Dans les noms de fichiers ou doit redoubler les \ en \\ afin d'éviter l'interprétation d'un \ comme le début d'une séquence d'échappement. (cf § Chaînes de caractères).

Déplacement dans un fichier:

Comme pour les fonctions de bas niveau, un fichier ouvert dispose—dans la structure FILE associée—d'un curseur (j'hésite toujours à dire un pointeur) qui indique l'endroit

dans le fichier où auront lieu les prochaines opérations de lecture/écriture.

A l'ouverture le curseur est au début du fichier, sauf en mode ajout.

L'écriture ne peut avoir lieu qu'à la fin du fichier, mais la lecture peut avoir lieu en tout point

du fichier au gré des déplacements du curseur.

Les fonctions associées sont fseek(), ftell(), feof() et rewind().

Syntaxe: long fseek(FILE *f_ptr, long dep, int org);
 long ftell(FILE *f_ptr);
 int feof(FILE *f_ptr);
 void rewind(FILE *f_ptr);

La fonction fseek() sert à déplacer le curseur dans le fichier du nombre d'octets donné par dep, par rapport à l'origine org conformément aux constantes:

SEEK_SET	déplacement par rapport au début du fichier
SEEK_CUR	déplacement par rapport à la position actuelle
SEEK_END	déplacement en arrière par rapport à la fin du fichier

Elle renvoie -1 en cas d'erreur.

Inversement la fonction `ftell()` renvoie la position actuelle du curseur par rapport au début du fichier associé à `f_ptr`.

La fonction `rewind()` ramène le curseur au début du fichier associé à `f_ptr`; `rewind(f_ptr)` équivaut à `fseek(f_ptr, 0, SEEK_SET)`;

La fonction `feof()` indique si la fin du fichier a été atteinte lors du déplacements du curseur ou après une opération de lecture. Elle retourne `TRUE (1)` ou `FALSE(0)` selon que la fin du fichier a été atteinte ou non.

Lecture et écriture dans un fichier:

La bibliothèque de haut niveau offre davantage de possibilités au niveau des fonctions de lecture / écriture que la bibliothèque de bas niveau: on peut lire ou écrire des caractères, chaînes, variables, blocs de données avec diverses fonctions:

Syntaxe: `fputc(int chr, FILE *f_ptr);`
 `int fgetc(FILE *f_ptr);`

`fputs(char *chaine, FILE *f_ptr);`
 `char * fgets(char *chaine, int nb, FILE *f_ptr);`

`fprintf(FILE *f_ptr, char *format, var1, ..., varN);`
 `fscanf(FILE *f_ptr, char *format, &var1, ..., &varN);`

`fwrite(void *zone, unsigned taille, unsigned nb, FILE *f_ptr);`
 `unsigned fread(void *zone, unsigned taille, unsigned nb, FILE *f_ptr);`

Les fonctions `fputc()` et `fgetc()` écrivent et lisent caractère par caractère dans un fichier.

Les int de la syntaxe seront comme d'habitude copieusement remplacés par des char.

Les fonctions `fputs()` et `fgets()` permettent d'écrire et lire une chaîne de caractères dans un fichier.

La lecture par `fgets()` s'arrête lorsqu'un caractère newline (`\n`) est rencontré, lorsque la fin

du fichier est rencontrée, ou lorsque le nombre maximal nb de caractères a été lu.

Dans ce premier cas le caractère `\n` est écrit avec la chaîne, qui est de plus terminée par un caractère NUL. `fgets()` retourne le pointeur NULL la fin du fichier est atteinte.

Les fonctions `fprintf()` et `fscanf()` fonctionnent comme `printf()` et `scanf()`, mais elle renvoient leurs résultats ou prennent leurs données dans le fichier indiqué. En général un fichier de données écrit de façon structurée avec des `fprintf()` sera relu avec des `fscanf()`.

Enfin les fonctions `fwrite()` et `fread()` permettent d'écrire des blocs de données—tableaux, structures—dans un fichier.

En plus de l'adresse de la zone mémoire contenant les données à écrire ou devant recevoir

les données lues, on indique la taille de chaque bloc de données, et le nombre de blocs à écrire ou lire.

Le volume des données manipulées est donc égal à `taille * nb octets`.

Exemple:

```
FILE *f_ptr;
int len;
char texte[ 80 ], copie[ 80 ];
...
f_ptr = fopen( "TOTO.TXT", "w + t" );
if( f_ptr == NULL ) printf( "erreur d'ouverture de fichier\n" );
else
    {
        strcpy( texte, "second essai\n" );
        len= strlen( texte );
        fputs( texte, f_ptr );
        fseek( f_ptr, 0, SEEK_SET );
        fgets( copie, len, f_ptr );
        printf( copie );
        fclose( f_ptr );
    }
```

Synchronisation des données:

La fonction `fflush()` force la synchronisation des données d'un fichier, c'est à dire qu'elle force l'écriture dans le fichier du contenu du tampon d'écriture, et qu'elle vide le tampon de lecture.

L'appel à cette fonction n'est pas nécessaire car la synchronisation est effectuée périodiquement. On ne l'utilisera que dans des cas particuliers, pour réinitialiser les tampons.

Syntaxe: `fflush(FILE *f_ptr);`

Exemple: `fflush(stdin);`

Il faut savoir que `stdin` et `stdout` sont des pointeurs pré-définis sur les drivers de périphériques associés à l'entrée et à la sortie standard, c'est à dire le clavier et l'écran.

L'opération `fflush(stdin)` a pour but de vider le tampon du clavier, par où transitent les caractères frappés encore en attente de traitement

2) Les fonctions de manipulation de fichiers:

En plus des fonctions précédentes permettant des opérations sur le contenu des fichiers,

il existe des fonctions manipulant les fichiers eux même, afin de les renommer, supprimer,

d'en changer les droits d'accès.

Elles nécessitent l'inclusion des headers <io.h> et parfois <sys\stat.h>.

Syntaxe: int rename(char *old_name, char *new_name);
 int remove(char *nom_fichier);
 int chmod(char *nom_fichier, int amode);

La fonction rename() renomme le fichier indiqué par old_name et lui donne le nom new_name.

Ces deux noms peuvent contenir une unité et un chemin.

Les deux unités doivent être identiques.

Si les deux chemins diffèrent, le fichier est déplacé dans le nouveau répertoire indiqué.

La fonction remove() supprime le fichier indiqué

La fonction chmod() modifie les droits d'accès au fichier.

Le nouveau mode d'accès en lecture / écriture est déterminé par les constantes S_IREAD et S_IWRITE.

Sachant que si l'écriture est autorisée, la lecture le sera automatiquement.

Les trois fonctions retournent -1 en cas d'erreur, notamment si le fichier n'existe pas.

3) Les fonctions de manipulation de répertoires:

Lorsqu'on manipule des fichiers on peut être amené à se déplacer parmi les différentes unités

(A:, C:, ...) ou dans l'arborescence des répertoires.

On peut aussi créer ou supprimer un répertoire.

De nombreuses fonctions sont disponibles pour réaliser ces opérations.

Elles nécessitent l'inclusion du header < dir.h >

Certaines sont issues du standard ANSI C, d'autres du TURBO PASCAL.

a) Opérations sur les répertoires et unités:

Les fonctions getdisk() et setdisk() traitent des unités de disques logiques (A:, B:, C:, ...)

Syntaxe: int getdisk();
 int setdisk(int drive);

La fonction getdisk() retourne l'unité active sous forme numérique:

Ø pour A:
1 pour B:
2 pour C: etc.

Inversement `setdisk()` permet de changer l'unité active avec les mêmes conventions. `setdisk()` renvoie -1 en cas d'erreur.

Les répertoires sont gérés par les fonctions `getcurdir()`, `chdir()`, `mkdir()`, `rmdir()` principalement.

Syntaxe: `getcurdir(int drive, char *chemin);`
`char * getcwd(char *chemin, int len);`
`int chdir(char *chemin);`
`int mkdir(char *chemin);`
`int rmdir(char *chemin);`
`int rename(char *old_name, char *new_name);`

La fonction `getcurdir()` permet d'obtenir dans la chaîne chemin le répertoire courant sur l'unité désignée par drive:

Ø pour l'unité courante
1 pour A
2 pour B
etc.

Cette fonction n'est pas au standard ANSI, contrairement à `getcwd()`:
`getcwd()` récupère dans la chaîne chemin le répertoire courant sur l'unité courante.
Le paramètre `len` est un nombre maximum de caractères que la fonction pourra stocker dans chemin.

Sous DOS on optera pour `len = 65` car un chemin ne dépasse jamais 64 caractères. Elle retourne un pointeur sur la chaîne chemin.

`chdir()` permet de changer de répertoire sur l'unité courante. On peut utiliser un déplacement absolu par rapport à la racine `\`, ou un déplacement relatif au répertoire actuel, y compris à l'aide de `.` (répertoire actuel) et `..` (répertoire au dessus).

En cas d'erreur `chdir()` retourne -1, sinon elle retourne Ø.

☛ **Attention:** Il faut toujours redoubler les `\` en `\\` dans les noms de répertoires.

Ensuite `mkdir()` et `rmdir()` créent ou suppriment le répertoire indiqué.

Elles retournent aussi -1 en cas d'impossibilité et sinon Ø.

La fonction `rename()` vue pour les fichiers peut également servir à renommer un répertoire.

Le répertoire ne peut toutefois pas être déplacé.

Exemples:

```

char rep[ 65 ];

getcurdir( Ø, rep );           /* équivaut à getcwd( rep, 65 ); */
mkdir( "C: \\ TOTO" ).        /* crée C: \ TOTO */
chdir( "C: \\ TOTO" );        /* passe dans C: \ TOTO */
mkdir( "TRUC" );              /* crée C: \ TOTO\TRUC */
chdir( rep );                  /* retour dans le répertoire d'origine */
rmdir( "C: \\ TOTO \\ TRUC" );
rmdir( "C: \\TOTO" );         /* supprime ces deux répertoires */

```

b) Recherche d'un fichier dans un répertoire:

Signalons la fonction searchpath(), qui assure la recherche d'un fichier donné dans les répertoires déclarés dans la variable d'environnement PATH du DOS.

Syntaxe: char * searchpath(char *nom_fichier);

En cas de succès, cette fonction retourne un pointeur sur une chaîne de caractères statique—donc écrasée à chaque appel—qui contient le nom complet du fichier avec le chemin d'accès.

En cas d'échec elle retourne le pointeur NULL.

Mis à part le cas particulier de searchpath() la recherche d'un fichier ou d'un groupe de fichiers correspondants à un format générique avec * et ? passe par les fonctions findfirst() et findnext(), fonctions qui sont héritées du TURBO PASCAL et ne sont pas au standard ANSI.

Syntaxe: int findfirst(char * chemin, struct fblk *ff_ptr, int attrib);
 int findnext(struct fblk *ff_ptr);

Ces fonctions reposent sur la structure pré-définie fblk dont quelques champs sont:

```

struct fblk
{
...
char ff_attrib;
int ff_fime;
int ff_fdate;
long ff_fsize;
char ff_name[13 ];
}

```

On doit appeler une fois et une seule la fonction `findfirst()` en lui passant:

- Le nom générique des fichiers recherchés dans la variable chemin;
Exemples: "C:\BC\SOURCE.C", "*.DOC", "C:MENU?.EXE", "*.*"
etc.
- L'attribut des fichiers recherchés → voir ci après
- L'adresse d'une structure `ffblk` déclarée dans le programme, qui recevra les renseignements concernant le premier fichier trouvé conforme au format générique indiqué.

Au retour, si aucun fichier convenable n'est trouvé la fonction `findfirst()` renvoie -1, sinon elle retourne \emptyset et la structure est complétée ainsi:

`ff_attrib` est l'attribut du fichier trouvé (Voir ci-après).
`ff_fime` l'heure de dernière modification.
`ff_date` la date de dernière modification, à un format détaillé dans l'aide.
`ff_fsize` est la taille du fichier en octets.
`ff_name` est le nom exact du fichier trouvé.

L'attribut est une combinaison des constantes pré-définies dans `<dos.h>`:

<code>FA_NORMAL</code>	fichier ordinaire
<code>FA_ARCH</code>	fichier archivé
<code>FA_DIREC</code>	sous répertoire
<code>FA_LABEL</code>	label de volume
<code>FA_SYSTEM</code>	fichier système
<code>FA_HIDDEN</code>	fichier caché
<code>FA_RDONLY</code>	fichier en lecture seule

Les fichiers usuels ont l'attribut `FA_ARCH` ou à défaut `FA_NORMAL`.

Ensuite—et en cas de succès de `findfirst()`—on doit appeler `findnext()` jusqu'à épuisement

des fichiers recherchés.

On passe seulement à `findnext()` l'adresse d'une structure `ffblk`.

Si plus aucun fichier convenable n'est trouvé `findnext()` retourne -1, sinon elle retourne \emptyset

et complète la structure comme ci-dessus.

Exemple:

```
int i;
struct fblk ff;
char nom [80];

strcpy( nom, "C:\\DOS\\*.SYS" );
i= findfirst( nom, &ff, FA_ARCH );
if ( i != Ø ) printf( "Aucun fichier ne correspond\n" );
else
    {
    printf( "Fichier trouvé: %s\n", ff.fname );
    do
        {
        i= findnext( &ff );
        if ( i != Ø ) printf( "Plus de fichier\n" );
        else printf( "Fichier trouvé: %s\n", ff.fname );
        }
    while ( i != Ø );
    }
```

IX Variables dynamiques.

1) Les principes:

Les variables utilisées jusqu'à présent étaient des variables globales ou locales (éventuellement statiques), qui étaient logées dans la zone de données ou sur la pile du programme.

L'inconvénient majeur est que ces données doivent être déclarées avant toute utilisation

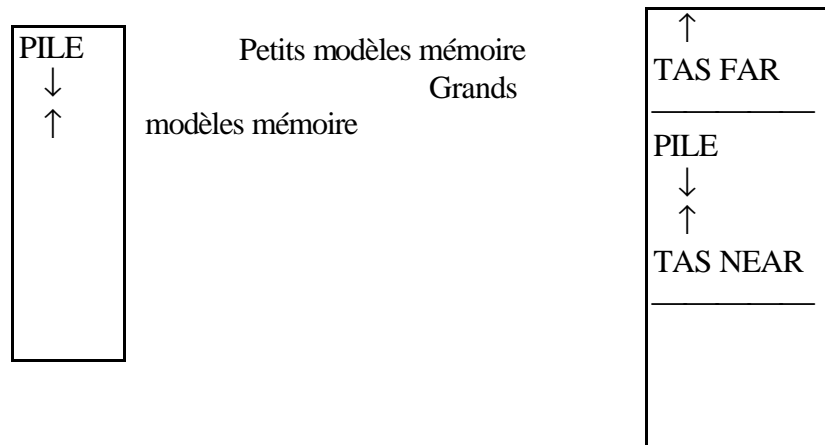
et ont une taille—pour les tableaux en particulier—qui est figée lors de la compilation du programme.

Pour remédier à ce problème on peut utiliser (seulement en cas de besoin absolu !) des variables dynamiques, qui permettent notamment de créer des tableaux de taille variable

et de stocker des structures en nombre variable aussi.

Les variables dynamiques sont allouées sur le tas (heap) qui se situe soit entre des données

et la pile dans les modèles mémoires tiny, small et medium, soit au delà de la pile et jusqu'à la fin de la mémoire dans les modèles compact, large et huge:



On peut ainsi calculer la taille nécessaire à un tableau et se faire allouer un emplacement du nombre d'octets voulu (inférieur toutefois à 64 KO).
Les variables dynamiques sont aussi utiles pour manipuler des données volumineuses de façon temporaire. Pourquoi encombrer le segment de données d'un tableau de 30 KO qui ne servirait qu'une fois et brièvement ? Autant utiliser l'allocation dynamique !

2) La programmation:

La bibliothèque de fonctions disponible comprend malloc(), calloc(), realloc() et free().
Elles nécessitent le header <alloc.h>.

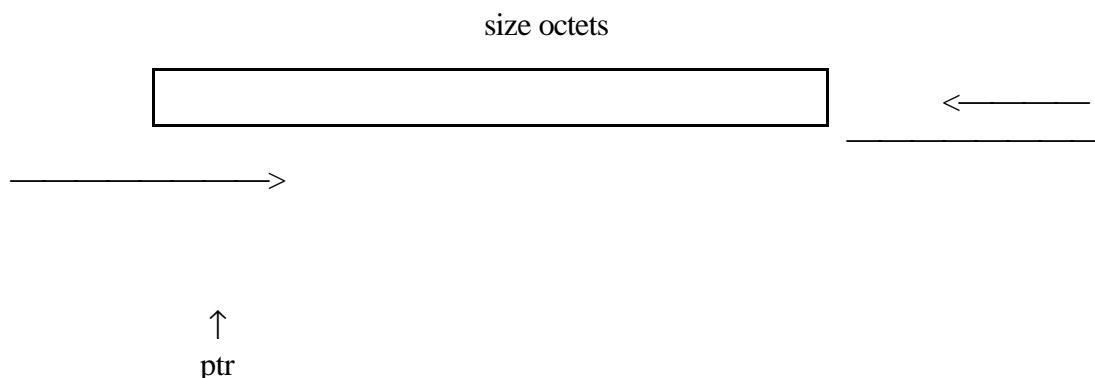
Syntaxe: void * malloc(unsigned size);
 void * calloc(unsigned nb, unsigned size);
 void * realloc(void * old_ptr, unsigned size);
 free(void * ptr);

On voit immédiatement que les variables dynamiques sont manipulées par des pointeurs !
L'allocation d'une zone mémoire de taille donnée se fait essentiellement à l'aide de malloc()
ou calloc().

La fonction malloc() reçoit la taille du bloc mémoire requis en octets (inférieure à 64 KO).

Elle retourne alors un pointeur sur le bloc mémoire alloué—par défaut de type void car on ne sait pas trop sur quoi cela va pointer—l'utilisateur peut alors écrire et relire des données dans le bloc mémoire commençant à l'adresse définie par ce pointeur et de la taille demandée.

Le bloc mémoire est toujours alloué d'un seul bloc.



L'arithmétique des pointeurs permet alors d'écrire et lire partout dans ce bloc mémoire.

calloc() fait de même, mais elle réclame comme paramètres la taille des données et leur nombre. Elle alloue donc un bloc de taille size * nb. On l'utilisera par exemple

pour allouer une zone mémoire à un certain nombre de structures avec `size = sizeof(structure)`.

De plus `calloc()` remplit de NULS ($\backslash\emptyset$) la zone allouée.

En cas d'erreur et notamment d'impossibilité de satisfaire la requête (taille trop grande par rapport au tas disponible) ces fonctions retournent le pointeur NULL. On doit toujours envisager cette éventualité dans le programme.

La fonction `realloc()` permet de réaménager un bloc mémoire déjà alloué par `malloc()` ou `calloc()` et d'en modifier la taille.

Le bloc mémoire est susceptible d'être déplacé au sein du tas (notamment en cas d'augmentation de taille du bloc) et la fonction retourne donc un nouveau pointeur qui remplace l'ancien.

L'adresse définie par l'ancien pointeur `old_ptr` qui avait été obtenu des fonctions `malloc()` ou `calloc()` ne doit plus être utilisée !

Enfin la fonction `free()` libère un bloc mémoire alloué par `malloc()`, `calloc()` ou `realloc()`.

On lui transmet le pointeur obtenu d'une de ces fonctions et elle se charge de libérer la place qui pourra être réallouée ultérieurement.

Toute zone allouée dynamiquement par `malloc()`, `calloc()`, `realloc()` doit être libérée par `free()` dès qu'elle ne sert plus et au plus tard avant de terminer le programme.

Exemple:

```
char *tab_ptr;
```

Remarque: Dans cet exemple on fait un casting de type pour transformer le pointeur en retour, de type (`void *`) en un pointeur de type (`char *`) afin d'être en conformité avec le type déclaré pour `tab_ptr` où l'on stockera probablement des chaînes ou tableaux de caractères.

☛ Attention: Le type de pointeur déclaré est très important car c'est lui qui détermine comment se fera l'arithmétique sur ce pointeur (cf § Pointeurs):
Le pointeur `tab_ptr + n` représente en fait l'adresse `tab_ptr + n * sizeof(type pointé)`.

Traitons un exemple complet, dans lequel on va sauvegarder sur le tas une partie donnée de l'écran en mode graphique; l'utilisateur détermine les coordonnées de la zone à sauvegarder.

La taille nécessaire est calculée et un bloc mémoire alloué:

Exemple général:

```

#include <stdio.h>
#include <conio.h>
#include <graphics.h>
#include <alloc.h>

void cadre_double(int, int, int, int);
void save_win(int, int, int, int);

int video_drv= VGA, video_mod= VGAHI, H_size, V_size;

void cadre_double(int X1, int Y1, int X2, int Y2)
    {
        setlinestyle(SOLID_LINE, 0, 1);

        rectangle(X1, Y1, X2, Y2);
        rectangle(X1+2, Y1+2, X2-2, Y2-2);
    }

void save_win(int X1, int Y1, int X2, int Y2)
    {
        struct viewporttype view;
        unsigned size;
        void * sav_ptr;

        size= imagesize(X1, Y1, X2, Y2);    /* calcule taille nécessaire */
        sav_ptr= (void *) malloc(size);      /* alloue bloc mémoire */

        if (sav_ptr == NULL) outtext("Allocation m,moire impossible !");
        else
            {
                getviewsettings(&view);      /* mémorise fenêtre */
                getimage(X1, Y1, X2, Y2, sav_ptr);

                setviewport(X1, Y1, X2, Y2, 1);
                clearviewport();             /* efface cette fenêtre */

                cadre_double(0, 0, X2-X1, Y2-Y1);

                getch();

                setviewport(view.left, view.top, view.right, view.bottom, view.clip);
                putimage(X1, Y1, sav_ptr, COPY_PUT);    /* restaure fenêtre */

                free(sav_ptr);              /* libère bloc mémoire */
            }
    }

```

```

void main()
{
    int x1, y1, x2, y2;

    clrscr();
    printf("Donnez les coordonnées de la fenêtre à sauvegarder :\n");
    printf("Coin supérieur gauche, X= ");
    scanf("%d", &x1);
    printf("Coin supérieur gauche, Y= ");
    scanf("%d", &y1);
    printf("Coin inférieur droit, X= ");
    scanf("%d", &x2);
    printf("Coin inférieur droit, Y= ");
    scanf("%d", &y2);

    initgraph(&video_drv, &video_mod, "");
    /* initialise mode VGA 640x480 */

    setbkcolor(BLUE);
    cleardevice();          /* efface écran en bleu */

    H_size= getmaxx()+1;    /* dimension horizontale écran */
    V_size= getmaxy()+1;   /* dimension verticale écran */

    setcolor(YELLOW);
    cadre_double(0, 0, H_size-1, V_size-1);    /* trace un cadre double */

    save_win(x1, y1, x2, y2);

    getch();

    closegraph(); /* fin du mode graphique */
}

```

Remarque: Au delà de certaines limites on constatera que l'allocation est impossible.

En effet l'écran tout entier en mode VGAHI fait $640 \times 480 = 307200$ pixels.

En 16 couleurs il faut 4 bits c'est à dire 1/2 octet pour chaque pixel donc 153600 octets.

Cela rentre dans la mémoire vidéo qui fait au moins 256 KO, mais ne rentre pas dans les 64 KO que peut allouer malloc().

Au delà d'une certaine taille; il faudrait sauvegarder la fenêtre en plusieurs parties !

X) Gestion des processus.

1) Notion de processus:

Un processus est un programme en cours d'exécution sur un système informatique. L'environnement MS-DOS étant (tristement !) monoutilisateur et (définitivement ?) monotâche, il n'y a en fait qu'un seul processus actif à un moment donné.

Dans l'attente de commandes, c'est le shell COMMAND.COM qui est le processus actif;

il saisit et interprète les commandes DOS qui lui sont soumises et lance l'exécution des programmes demandés, qui sont sous DOS de type .COM ou .EXE.

Il est possible à un processus en cours d'exécution de demander l'exécution d'un autre processus, de différentes manières:

- Un processus peut demander l'exécution d'une commande système (dir, copy, ...) après quoi il reprendra son exécution.
- Un processus peut suspendre son exécution, appeler un autre programme exécutable, et reprendre son exécution à la fin de ce dernier.
- Un processus peut se terminer en étant remplacé par un autre programme exécutable de son choix.

De façon à rendre ces mécanismes plus intéressants, le processus appelant (nommé processus père) peut passer des informations au processus appelé (nommé processus fils), sous forme de paramètres analogues à ceux que l'on peut transmettre à un programme sur la ligne de commande.

Exemple: edit toto.c
 "toto.c" est un paramètre transmis au programme EDIT.COM qui l'interprète comme le nom du fichier à éditer.

Le mécanisme de passage des paramètres sera détaillé plus loin. Inversement un processus qui se termine peut renvoyer à son père un code de retour pour l'informer des conditions dans lesquelles il s'est terminé. En général \emptyset signifie que tout s'est bien passé.

2) Programmation:

Les fonctions disponibles sont exec...(), spawn...(), system() et exit(). Ces fonctions nécessitent les headers <process.h> et parfois <stdlib.h>

Syntaxe: int execl(char *progrname, char *arg \emptyset , ..., char *argN, NULL);
 int execl(char *progrname, char *arg \emptyset , ..., char *argN, NULL, char **env);
 int execlp(char *progrname, char *arg \emptyset , ..., char *argN, NULL);

```

int execlpe( char *progname, char *argØ, ..., char *argN, NULL, char
**env );
int execv( char *progname, char **argv );
int execve( char *progname, char **argv, char **env );

int execvp( char *progname, char **argv );
int execvpe( char *progname, char **argv, char **env );
int spawnl( int mode, char *progname, char *argØ, ..., char *argN,
NULL );
int spawnle( int mode, char *progname, char *argØ, ..., char *argN,
NULL,
char **env );
int spawnlp( int mode, char *progname, char *argØ, ..., char *argN,
NULL );
int spawnlpe( int mode, char *progname, char *argØ, ..., char *argN,
NULL,
char **env );
int spawnv( int mode, char *progname, char **argv );
int spawnve( int mode, char *progname, char **argv, char **env );
int spawnvp( int mode, char *progname, char **argv );
int spawnvpe( int mode, char *progname, char **argv, char **env );

int system( char *commande );
exit( int val );

```

Les fonctions exec...():

Ne nous laissons pas abuser par la multitude de variantes d'exec...() et spawn...() !

Le principe des fonction exec...() est le suivant:

Le programme désigné par progname, avec éventuellement un chemin d'accès et une unité, devient le processus actif, à la place du processus père qui se termine avec l'appel à exec...().

Il est donc vital de comprendre qu'il n'y a pas de retour d'un exec...() réussi puisque l'appelant se termine immédiatement.

Il n'y a retour qu'en cas d'erreur (programme inexistant ...) avec la valeur -1.

On devra envisager cette éventualité.

Les fonctions execl...() reçoivent en plus un nombre déterminé d'arguments argØ, ..., argN

qui sont les chaînes de caractères à passer en paramètre au processus fils.

La liste de ces arguments se termine par un NULL. Par convention l'argument argØ est toujours présent et est le nom du processus fils. (cela permettra au processus fils de savoir comment il s'appelle).

Exemple:

```
int e;
...
e = execl( "C:\\DOS\\DISKCOPY.EXE", "DISKCOPY.EXE", "a:", "b:", NULL );
        /* lance discopy a: b: */
if ( e == -1 ) printf( "Erreur d'exécution\n" );
```

Les fonctions `execv...()` reçoivent les paramètres d'une façon différente, sous la forme

d'un tableau de pointeurs sur des chaînes de caractères `char *argv[]`.

Le dernier élément de ce tableau est un `NULL` marquant la fin de la liste.

On utilise les variantes `execv...()` lorsque le nombre de paramètres transmis n'est pas connu d'avance.

Remarque: L'ensemble des paramètres transmis ne peut pour des raisons techniques excéder 128 caractères.

Exemple:

```
int e;
char *argv[ ] = { "DISKCOPY.EXE", "a:", "b:", NULL };
...
e = execv( "C:\\DOS\\DISKCOPY.EXE", argv );
        /* même résultat que ci-dessus */
if ( e == -1 ) printf( "Erreur d'exécution\n" );
```

Les variantes `execlp()` et `execvp()` opèrent comme `execl()` et `execv()` sauf qu'elles recherchent le programme programme dans le `PATH` du DOS si nécessaire.

Les variantes `execle()` et `execve()` reçoivent un paramètre d'environnement supplémentaire:

il s'agit d'un tableau de pointeurs sur des chaînes de caractères, terminé par un `NULL`, représentant des variables d'environnement à ajouter à l'environnement du processus fils.

Par défaut le processus fils reçoit une copie de l'environnement de son père.

Enfin `execlpe()` et `execvele()` combinent les avantages de deux variantes précédentes.

Exemple:

```
int e;
char *env[ ] = { "INSTALL = C:\\WORK", NULL };
...
e = execle( "C:\\TOTO.EXE", "TOTO.EXE", NULL, env );
        /* le père transmet une variable d'environnement à son fils */
if ( e == -1 ) printf( "Erreur d'exécution\n" );
```

Remarque: Le père du processus fils se terminant, le fils est adopté par COMMAND.COM qui devient son père adoptif, et qui récupérera un éventuel code de retour.

Les fonctions spawn...():

Le principe des fonctions spawn...() est différent dans la mesure où le processus père reste présent en mémoire et est seulement suspendu lors de l'exécution du processus fils.

Le père reprend son exécution à la fin du fils et reçoit un code de retour.

Les paramètres de spawn...() sont identiques en tous points pour les diverses variantes

à ceux d'exec...(). S'y ajoute le paramètre mode qui peut prendre l'une des valeurs pré-définies suivantes:

P_WAIT	→	Le père attend la fin de son fils.
P_OVERLAY	→	Le fils remplace le père, ce qui équivaut alors à faire un exec...().

Les fonctions spawn...() retournent -1 en cas d'erreur, sinon elles retournent le code de retour du processus fils tel qu'il est positionné par la fonction exit() ci-dessous.

La fonction exit():

Le code de retour qu'un processus renvoie à son père (le plus souvent COMMAND.COM) est transmis par la fonction exit():

exit(val) termine le programme immédiatement et retourne à son père le code de retour val.

- val = 0 signifie souvent que tout s'est passé normalement
- val = +1 ou toute autre valeur convenue signifie une terminaison anormale.

Une variante consiste à faire un return(val) dans la fonction main().

En cas d'appel à return(val) on doit déclarer int main() et non void main() dans la mesure où le programme renvoie un code de retour explicite.

Les appels systèmes:

Les fonctions exec...() et spawn...() permettent d'appeler un programme .COM ou .EXE.

Pour appeler une commande interne du DOS on peut utiliser system(), qui reçoit en paramètre la commande à exécuter.

La fonction `system()` renvoie \emptyset si tout va bien et -1 en cas d'impossibilité.
Le processus appelant reprend la main à la fin de la commande invoquée.

Exemple:

```
int e;
...
e = system( "DIR C:\ " );    /* exécute un dir C:\ */
if ( e == -1 ) printf( "Erreur d'appel système\n " );
```

Remarques: `system()` relance en fait `COMMAND.COM` qui se charge d'exécuter la commande qui lui est transmise.
`COMMAND.COM` est localisé si nécessaire à l'aide de la variable d'environnement `COMSPEC`.

L'appel à `system()` ci-dessus pourrait donc être remplacé par un appel à `spawnlp()` avec comme programme "`COMMAND.COM`" et comme paramètres "`DIR`" et "`C:\`".

La récupération des paramètres:

On a vu comment passer des paramètres à un programme, mais reste à savoir comment le programme récupère ces paramètres en C (qu'ils viennent d'un `spawn...()`, d'un `exec...()` ou directement de la ligne de commande).

On déclare en général dans le programme `main()`.
Si on veut récupérer d'éventuels paramètres on doit déclarer `main(int argc, char **argv)`
où `argc` sera le nombre de paramètres transmis et `argv` un tableau de pointeurs sur ces paramètres sous forme de chaînes de caractères accessibles par :
`argv[0]` , ..., `argv[argc-1]`.

Exemple:

```
int main( int argc, char **argv );
{
char nom[128];
...
strcpy( nom, argv[0] );
printf( "je suis le programme %s ", nom );
/* argv[0] est conventionnellement le nom du programme */
...
exit( 0 );      /* fin normale du programme */
}
```

Remarque: char **argv est souvent remplacé par char *argv[], ce qui revient au même, désignant un tableau de pointeurs sur caractères.

XI) Gestion des projets en TURBO C ou Borland C.

1) Création d'un fichier de projet:

Tant qu'un programme reste assez simple, un seul fichier source C suffit, que l'on compile

et linke comme on l'a vu précédemment.

Mais lorsque le programme devient conséquent, il est nécessaire de le rendre plus modulaire

en constituant plusieurs sources C, pour diverses raisons:

- On rassemble ainsi des parties du programme de façon cohérente.
- On évite d'avoir à recompiler tout le programme lors de modifications mineures.
- On facilite la maintenance ultérieure des sources.

On constitue alors un projet, regroupant les divers sources C, en créant un fichier .PRJ dans le menu **project** → **open project**.

Il suffit de taper un nom de fichier pour le projet (*.PRJ).

L'exécutable (*.EXE) résultant de la compilation et du linkage portera le nom de ce projet.

On peut ouvrir de même un projet existant.

Apparaît alors une fenêtre projet et un menu associé en dessous, qui permet d'ajouter / enlever des modules au projet:

- Add pour ajouter un module dans la liste.
- Delete pour supprimer le module enluminé dans la liste.

En fin de session on ferme le projet avec l'option **project** → **close project**.

2) Contenu d'un projet:

Un projet ne gère pas seulement les sources C multiples; on peut également y inclure d'autres types de fichiers:

- Sources C (*.C)
- Sources assembleur (*.ASM)
- Fichiers objets précompilés (*.OBJ)
- Bibliothèques de fonctions précompilées (*.LIB)

Les fichiers seront traités différemment selon leur nature:

- Les sources *.C sont compilés par TURBO C en *.OBJ

- Les sources *.ASM sont assemblés en *.OBJ par appel à l'assembleur intégré.
- Les fichiers objets résultants sont liés avec les fichiers objets précompilés et les bibliothèques pour former l'exécutable .EXE.

3) Compilation et linkage d'un projet:

Une fois définis le fichier projet et les modules qui le composent, on peut compiler ou linker le tout de différentes façons dans le menu **compile** dont les diverses options prennent ici tout leur sens:

Compile → **Compile** (ALT_F9)
compile le source C de la fenêtre active.

Compile → **Make** (F9)
Compile et assemble tous les sources ayant été modifiés et linke le programme en un exécutable .EXE.

Compile → **Link**
Relinke l'exécutable à partir des modules *.OBJ et *.LIB existants.

Compile → **Build all**
Recompile tous les modules du projet et linke le tout en un exécutable.

a) headers personnalisés:

Un projet est souvent l'occasion d'écrire ses propres headers *.h. En effet des déclarations identiques doivent parfois figurer dans plusieurs sources et afin de ne pas avoir à modifier plusieurs occurrences d'une même déclaration ou préférera inclure un header commun.

Un header peut contenir:

- Des déclarations de CONSTANTES
- Des déclarations de fonctions
- Des déclarations de MACRO_FONCTIONS
- Des déclarations de variables globales

Un header présent dans le répertoire du projet sera accessible sous la forme

include "header.h" avec des guillemets.

A	adresse (&)	19
	arc()	70
	arithmétique des pointeurs	20
	atof()	51
	atoi()	51
	atol()	51
	attributs	58
B	bar()	73
	bar3D()	73
	BGI	64
	break	33
C	calloc()	90
	caractères	12
	casting de type	41
	cercle()	70
	cgets()	57
	chaînes de caractères	14
	champs	26
	char	12
	chdir()	85
	chmod()	84
	classes de variables	16
	cleardevice()	67
	clearviewport()	67
	close()	78
	closegraph()	66
	clrscr()	55
	commentaires	4
	continue	33
	contour	73
	coordonnées graphiques	66
	coordonnées texte	55
	couleurs	58
	cprintf()	57
	cputs()	57
	creat()	78
cscanf()	57	
D	déclaration de fonction	38
	define (#)	7
	définition de fonction	38

E	directives	7
	do...while	30
	drawpoly()	70
	ellipse()	70
	entiers	7
	eof()	79
	exec...()	94
exit()	95	

F	FALSE	35
	fclose()	81
	fenêtres texte	54
	fflush()	84
	fgetc()	82
	fgets()	83
	FILE	81
	filelength()	79
	fillellipse()	72
	fillpoly()	72
	findfirst()	86
	findnext()	86
	float	11
	floodfill()	73
	flottant	11
	fonctions	37
	fopen()	81
	for	30
	format (%)	42
	fprintf()	83
	fputc()	82
	fputs()	83
	fread()	83
	free()	90
	fscanf()	83
	fseek()	82
	ftell()	82
feof()	82	
fwrite()	83	

G	getaspectratio()	70
	getbkcolor()	68
	getch()	56
	getchar()	45
	getche()	56
	getcolor()	68
	getcurdir()	85
	getcwd()	85
	getdisk()	85

getimage()	74
getlinesettings()	69
getmaxx()	66
getmaxy()	66
getpixel()	68
gets()	46
gettext()	59
gettextinfo()	59
getviewsettings()	74
getx()	68
gety()	68
goto	33
gotoxy()	55
grapherrormsg()	65
graphresult()	65

H	handle	78
	hexadécimal	7

I	if...else	34
	imagesize()	74
	include (#)	7
	indices	24
	indirection (*)	19
	initgraph()	65
	initialisation de structure	28
	initialisation de tableau	26
	int	8
	itoa()	51

K	kbhit()	56
----------	----------	----

L	line()	69
	linerel()	69
	lineto()	69
	long	8
	lseek()	79
	ltoa()	51

M	malloc()	90
	mkdir()	85
	modèles de pointeurs	21
	modèles mémoire	89
	modes vidéo graphiques	64

	modes vidéo textes	53
	moverel()	67
	moveto()	67
N	NUL	14
	NULL	20
	numération	7
O	octal	7
	open()	78
	opérateurs	9, 10, 12
	organigramme	6
	outtext()	73
	outtextxy()	73
P	passage de paramètres	39, 94, 95
	PATH	86, 96
	pieslice()	72
	pixel	68
	pointeurs	18
	printf()	42
	processus	94
	projet	99
	putch()	56
	putchar()	45
	putimage()	74
	putpixel()	68
	puts()	46
	puttext()	59
R	rapport d'aspect	70
	read()	80
	realloc()	90
	rectangle()	70
	remove()	84
	rename()	84, 85
	rewind()	82
	rmdir()	85
S	scanf()	45
	searchpath()	86
	sector()	72
	séquence d'échappement	13
	setaspectratio()	70
	setbkcolor()	68
	setcolor()	68

setdisk()	85	
setfillpattern()		71
setfillstyle()	71	
setlinestyle()	69	
settextstyle()	73	
setviewport()	67	
sizeof()	29	
spawn...()	95	
sprintf()	52	
stdin	84	
stdout	84	
strcat()	48	
strchr()	50	
strcmp()	48	
strcmpi()	49	
strcpy()	47	
stricmp()	49	
strlen()	49	
strlwr()	50	
strncat()	48	
strncmp()	48	
strncpy()	47	
strrchr()	50	
structures	26	
strupr()	50	
switch...case	36	
synchronisation	84	
system()	95	

T		
tableaux	24	
tell()	79	
text_info	60	
textattr()	58	
textbackground()	58	
textcolor()	58	
textmode()	53	
tolower()	50	
toupper()	50	
TRUE	35	
typedef	23	

U		
ultoa()	51	
unsigned		8

V		
valeur de retour	39	
variables dynamiques	89	
variables externes	17	

variables globales	16
variables locales	16
variables statiques	16
viewporttype	74

W

wherex()	55
wherey()	55
while	31
window()	54
write()	80