La Programmation en C

U.F.R. d'informatique

Juliette Dibie

Plan

I. Un premier programme C

```
#include <stdio.h>                    //bibliothèque d'entrées/sorties
int main()
{
      int i=0, total=0, n=5;
      for(i=1;i<=n;i++)
            total=total+i;
      printf("La somme des entiers de 1 a %d est %d\n",n,total);
}
```

- La fonction main est une fonction particulière qui représente le code à exécuter.
- printf correspond à une fonction d'affichage dans la fenêtre console.
- Dans un programme C, les **commentaires** sont placés soit après le délimiteur //, soit entre les délimiteurs /* et */.

Le résultat de l'exécution de ce programme est :

```
La somme des entiers de 1 a 5 est 15
```

Le format général de la fonction printf est le suivant :

printf(format, val_1, val_2, …, val_n)

- val_1, val_2, …, val_n représentent les valeurs à afficher ;

- format est une chaîne de caractères contenant les codes de mise en forme associés, dans l'ordre, aux arguments val_i de la fonction printf.

Un **code de mise en forme** pour une valeur donnée précise le format d'affichage de la valeur. Il est précédé du caractère % :

- %d permet d'afficher une valeur entière,
- %f permet d'afficher une valeur réelle,
- %c permet d'afficher un caractère,
- %s permet d'afficher une chaîne de caractères,
- %% permet d'afficher le caractère « % »…

II. Les Types primitifs

II.1.Les types entier

II.1.1. Les différents types entier

On distingue les types entier signé des types entier non signé.

a. Les types entier signé

type	Taille (n bits)	domaine de valeurs (-2^{n-1} à $2^{n-1} – 1$)
short int	16 bits	-32 768 à 32 767
int	16 ou 32 bits	
long int	32 bits	-2 147 483 648 à 2 147 483 647

Le choix d'un des trois types entier short int, int ou long int peut être guidé par les trois considérations suivantes :

- l'intervalle de valeurs entières nécessaires,
- la quantité d'espace mémoire disponible,
- la vitesse du programme.

Le type int est traditionnellement le type entier « standard » du langage C et les opérations sur ce type sont généralement les plus efficaces. Cependant, la représentation en mémoire des entiers de type int dépend du compilateur C utilisé, ce qui pose des problèmes de portabilité du programme C.

Le type **int** est représenté sur 32 bits avec le compilateur Dev-C++.

b. Les types entier non signé

type	Taille (n bits)	domaine de valeurs (0 à $2^n – 1$)
unsigned short int	16 bits	0 à 65 535
unsigned int	16 ou 32 bits	
unsigned long int	32 bits	0 à 4 294 967 295

II.1.2. La représentation en mémoire des types entier signé

Un bit est réservé au signe (0 pour positif et 1 pour négatif).

Les autres bits servent à représenter :

- la valeur absolue du nombre pour les positifs,
  Exemple d'un entier positif représenté sur 8 bits (un octet)

0	0	0	0	0	0	0	1
Signe 2^6	2^5	2^4	2^3	2^2	2^1	2^0	

$1*2^0 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6 = 1$

0	0	0	1	0	1	1	0

$2^1 + 2^2 + 2^4 = 22$

0	1	1	1	1	1	1	1

$2^7 - 1 = 127$ *(Valeur maximale)*

- ce que l'on nomme le « complément à deux » pour les négatifs (tous les bits sont inversés sauf celui réservé au signe et une unité est ajoutée au résultat).
  Exemple d'un entier négatif représenté sur 8 bits (un octet)

1	1	1	1	1	1	1	1
↓							
1	0	0	0	0	0	0	0

$-1*((0*2^0 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6) + 1)$
$= -1 * (0 + 1) = -1$

1	0	0	1	0	1	1	0
↓							
1	1	1	0	1	0	0	1

$-1 * ((2^0 + 2^3 + 2^5 + 2^6) + 1) = -106$

1	0	0	0	0	0	0	0
↓							
1	1	1	1	1	1	1	1

$-1 * ((2^7 - 1) + 1) = -128$ *(Valeur minimale)*

Le nombre 0 est codé par :

0	0	0	0	0	0	0	0

II.2.Les types réel

Il existe deux types réel standards : float et double. Certains compilateurs proposent aussi le type réel long double.

type	**taille**	**domaine de valeurs** *(pour les valeurs positives)*
float	32 bits	$3.4E^{-38}$ à $3.4E^{38}$
double	64 bits	$1.7E^{-308}$ à $1.7E^{308}$
long double	64 ou 96 bits	

II.3.Le type caractère

Les caractères sont de type char.

Les caractères sont représentés en mémoire sur 8 bits :

- domaine de valeurs du type char de -128 à 127 ;
- domaine de valeurs du type unsigned char de 0 à 255.

Les valeurs de type char sont traitées comme des entiers et peuvent être utilisées dans des expressions entières.

TD1. Prise en main de l'environnement de développement

En suivant les indications données par l'animateur de session, compilez et executez le programme suivant :

```
#include <stdio.h>                    //bibliothèque d'entrées/sorties
#include <stdlib.h>
int main()
{
      int i=0, total=0, n=5;
      for(i=1;i<=n;i++)
            total=total+i;
      printf("La somme des entiers de 1 a %d est %d\n",n,total);
      return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int somme = 0;
	int deux = 2;
	int sept = '7';
	somme = deux + sept;
	printf("la somme est %d\n", somme);
	return 0;
}
```

Ce programme devrait afficher à l'écran la somme 2+7, soit 9.

Créez, toujours sous le répertoire TD1, le projet td1_c, avec le programme essai_debug.c.

Quand on compile avec succès un programme XXX.c, on obtient du code-objet, dans le fichier XXX.o. Le code objet n'est pas encore exécutable : il faut lui adjoindre le code des fonctions externes appelées (ici printf et system) et établir les liens entre tous ces morceaux de code. Cette opération s'appelle l'**édition de liens**. Elle produit un programme exécutable dans un fichier de suffixe exe (XXX.exe ou PPP.exe, si PPP est le nom du projet).

Exécutez le programme précédent à l'aide du débogueur : Menu Debug – Debugger (F8). Commencez par poser un point d'arrêt sur l'instruction : int deux = 2; L'exécution est déclenchée jusqu'au point d'arrêt. Obtenez l'affichage des variables que vous souhaitez examiner et exécutez pas à pas votre programme pour voir ce qu'il faut corriger. Corrigez l'erreur et re-exécutez le programme.

The GNU C library reference manual :
http://www.gnu.org/software/libc/manual/html_mono/libc.html

Support de cours intéressant:
http://www.lri.fr/~bastoul/teaching/systeme/docs/Introduction_ANSI_C.pdf

Liens utiles :
https://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/LI205-2005oct/public/fiches-cini.pdf

III. Les constantes

III.1. Les constantes de type entier

Une constante entière est de l'un des types int ou long int suivant sa valeur. On peut forcer une constante entière à être du type long int en faisant suivre sa valeur de la lettre **l** (ou **L**).
Exemple : 25**L**

La valeur d'une constante entière est toujours positive en l'absence de dépassement de capacité. S'il existe un signe moins devant la constante, celui-ci est considéré comme un opérateur unaire appliqué à la constante.

Dans le cas d'un dépassement de capacité d'une constante entière, le compilateur n'avertit pas le programmeur et remplace la valeur de la constante par une valeur quelconque représentable.

Notations

- **décimale**
  Exemple : 125 ($1*10^2 + 2*10^1 + 5*10^0$)

- **octale**
  La valeur exprimée en base 8 (0->7) est précédée de **0**.
  Exemple : **0**175 ($1*8^2 + 7*8^1 + 5*8^0 = 64 + 56 + 5 = 125$)

- **hexadécimale**
  La valeur exprimée en base 16 (0->9, a, b, c, d, e, f) est précédée de **0x** ou **0X**.
  Exemple : **0x**7d ($7*16^1 + 13*16^0 = 112 + 13 = 125$)

Exemple

```
#include <stdio.h>
int main()
{
	printf("%d\t %d\t %d\n",125,0175,0x7d);
	printf("%o\t %#o\n",125,125);
	printf("%x\t %X\t %#X\n",125,125,125);
	system("pause");
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
125	125	125	//affichage décimal
175	0175		//affichage octal
7d	7D	0X7D	//affichage hexadécimal
```

III.2. Les constantes réelles

Une constante réelle est de type double.

La valeur d'une constante réelle est toujours positive en l'absence de dépassement de capacité. S'il existe un signe moins devant la constante, celui-ci est considéré comme un opérateur unaire appliqué à la constante.

Dans le cas d'un dépassement de capacité d'une constante réelle, le compilateur n'avertit pas le programmeur et remplace la valeur de la constante par une valeur quelconque représentable.

Notations

- **décimale**
  La présence d'un point est obligatoire.
  Exemple : 1.25 4. .7

- **exponentielle**
  Utilisation de la lettre e ou E pour introduire un exposant entier (puissance de 10).
  Exemple : -3.2E6

Exemple

```
#include <stdio.h>
int main()
{
      printf("%f\n",-3.2E6);
      printf("%.2f\n",3.26941);
      system("pause");
      return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
-3200000.000000      //précision par défaut de 6 chiffres après la virgule
3.27                 //résultat arrondi à deux chiffres après la virgule
```

III.3. Les constantes de type caractère

Une constante de type caractère est de type int. Sa valeur correspond à une valeur entière dans le code ASCII.

Notations

- **classique entre apostrophes**
  Exemple :
  'a' (code ASCII : 97)
  'A' (code ASCII : 65)
  Exemple de caractères particuliers :
  '\0' (caractère spécial de fin de chaîne de caractères, code ASCII : 0)
  '\a' (signal sonore)
  '\r' (retour chariot)
  '\n' (nouvelle ligne)
  '\t' (tabulation horizontale)
  '\\' (barre inverse)
  '\" (apostrophe)
  '\"' (guillemet)
  '\?' (point d'interrogation)

- **octale** (pour les caractères ayant un code compris entre 0 et 255)
  Exemple : '\101' (représente le caractère 'A' de code ASCII 65)

- **hexadécimale**
  Exemple : '\x41' (représente le caractère 'A' de code ASCII 65)

Exemple

```
#include <stdio.h>
int main()
{
      printf("%c\t %d\t %c\t %d\n",'A','A','\101','\x41');
      printf("%o\t %#o\n",'A','A');
      printf("%x\t %#X\n",'A','A');
      return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
A       65      A     65      //affichage décimal
101     0101                  //affichage octal
41      0X41                  //affichage hexadécimal
```

Des relations particulières entre certaines constantes caractères

On a les relations suivantes entre certaines constantes caractères :

'0'<'1'<…<'9'<'A'<'B'<…<'Z'<'a'<'b'…<'z'

Les valeurs '0', '1', … '9' forment une suite croissante d'entiers positifs et consécutifs. Dans l'environnement de développement Dev-C++, on a :

code ASCII de '0'= 48
code ASCII de '1'= 49
...
code ASCII de '9'= 57

Les valeurs 'A', 'B', ... 'Z' forment une suite croissante d'entiers positifs et consécutifs. Dans l'environnement de développement Dev-C++, on a :

code ASCII de 'A'= 65
code ASCII de 'B'= 66
...
code ASCII de 'Z'= 90

Les valeurs 'a', 'b', ... 'z' forment une suite croissante d'entiers positifs et consécutifs. Dans l'environnement de développement Dev-C++, on a :

code ASCII de 'a'= 97
code ASCII de 'b'= 98
...
code ASCII de 'z'= 122

III.4. Les constantes de type primitif

Le mot clé **const** permet de déclarer que la valeur d'une variable ne doit pas être modifiée pendant l'exécution d'un programme.

Une variable déclarée avec le mot clé **const** doit être initialisée dès sa définition.

Exemple

```
#include <stdio.h>
int main()
{
	const int n=10;
	n=5;                    //erreur de compilation : n a été déclaré const
	int p=2*n;
	return 0;
}
```

III.5. Un programme-exemple

Le programme suivant permet de convertir une chaîne de caractères en entier.

```
#include<stdio.h>

int main()
{
        char ch[20];
        int i=0,n=0;
        printf("Donnez un entier positif: ");
        scanf("%s",ch);
        for(i=0;ch[i]!='\0';i++)
        {
                n = 10 * n + ch[i] - '0';
        }
        printf("L'entier donne en entree est %d\n",n);
        return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif : 9573
L'entier donne en entree est 9573
```

L'expression ch[i] – '0' donne la valeur numérique du caractère contenu dans ch[i]. En effet, les valeurs '0', '1', ... forment une suite croissante d'entiers positifs et consécutifs.

'\0' est le caractère spécial de fin de chaîne de caractères.

Le format général de la fonction scanf est le suivant :

scanf(format, av_1, av_2, …, av_n)

- av_1, av_2, …, av_n représentent les adresses des valeurs à lire ;

- format est une chaîne de caractères contenant les codes de conversion qui permettent d'indiquer la manière dont les valeurs doivent être lues et associées, dans l'ordre, aux arguments av_i de la fonction scanf.

Un **code de conversion** pour une valeur donnée précise le format de lecture de la valeur. Il est précédé du caractère % :
- %d permet de lire une valeur entière,
- %f permet de lire une valeur réelle,
- %c permet de lire un caractère,
- %s permet de lire une chaîne de caractères…

TD2. Un programme pour convertir une chaîne de caractères en entier

Améliorez le programme précédent, qui permet de convertir une chaîne de caractères en entier, selon les deux axes suivants :

1. testez que la chaîne saisie est bien un entier positif : pour ce faire, il faudra s'assurer que tous les caractères de la chaîne saisie sont compris entre les caractères '0' et '9' ;

2. tester que l'entier saisi est bien inférieur à un nombre donné, par exemple 500000.

Pour ce faire, ouvrez le projet td2 qui se trouve dans le répertoire TD2.

IV. Les entrées/sorties élémentaires

Les opérations d'entrées/sorties étudiées jusqu'à présent sont des lectures/écritures (à partir du clavier et vers l'écran), qui sont effectuées via les fonctions standard printf et scanf. On dit que ces fonctions travaillent sur des flux standard de caractères. Ce sont :

- le flux stdin, qui désigne en général le clavier,
- le flux stdout, qui désigne en général l'écran.

L'utilisation de la fonction scanf n'est pas toujours très commode, car cette fonction attend, pour terminer son exécution, d'avoir rencontré une fin de ligne (obtenue quand, en saisie, on frappe la touche Entrée).
Si, par exemple, on exécute :

```
#include <stdio.h>
#include <ctype.h>
int main()
{
	char carLu='\0';
	//essais de scanf : test de lecture de caractères, arrêt quand on tape F
	do {
		printf("tape un caractere (la lettre f pour arreter) : ");
		scanf("%c", &carLu);
		printf("Caractere lu : %c, code ascii : %d\n", carLu, carLu);
	} while (toupper(carLu) != 'F');
	system("pause") ; return 0 ;
}
//toupper(c) renvoie la majuscule correspondant à c, si c est une lettre de
//l'alphabet anglo-saxon. Sinon toupper(c) renvoie c. Le header est <ctype.h>
```

Et si on tape le caractère A, validé immédiatement par la touche Entrée, on obtient l'affichage suivant :

```
tape un caractere (la lettre f pour arreter) : A
Caractere lu : A, code ascii : 65
tape un caractere (la lettre f pour arreter) : Caractere lu :
, code ascii : 10
tape un caractere (la lettre f pour arreter) : f
```

Cet affichage montre que la touche Entree (code ascii 10, qui correspond à \n) est considérée comme un caractère. Ce caractère, dont la présence termine la

première exécution de scanf, est celui qui sera lu à la seconde exécution ! Par exemple si on veut se servir de la séquence précédente, pour effectuer deux saisies validées, des lettres C et D par exemple, la lettre D ne sera saisie qu'avec le troisième appel de scanf.

Pour lire un seul caractère, on peut aussi utiliser la fonction standard **getchar**, qui renvoie le prochain caractère du flux stdin :

```
#include <stdio.h>
#include <ctype.h>
int main()
{
	char carLu='\0';
	//essais de getchar : test de lecture de caractères, arrêt quand on tape F
	printf("tape un caractere (lettre f pour arreter) : ");
	while (toupper((carLu = getchar())) != 'F') {
		printf("Caractere lu : %c, code ascii : %d\n", carLu, carLu);
		printf("tape un caractere (lettre f pour arreter) : ");
	};
	system("pause") ; return 0 ;
}
```

Si on exécute cette séquence en tapant C (validé avec la touche Entree), puis D (également validé avec la touche Entree), on obtient l'affichage suivant :

```
tape un caractere (la lettre f pour arreter) : C
Caractere lu : C, code ascii : 67
tape un caractere (la lettre f pour arreter) : Caractere lu :
, code ascii : 10
tape un caractere (la lettre f pour arreter) : D
Caractere lu : D, code ascii : 68
tape un caractere (la lettre f pour arreter) : Caractere lu :
, code ascii : 10
tape un caractere (la lettre f pour arreter) : f
```

Ce qui montre que chaque lecture d'un caractère validé nécessite deux exécutions de getchar. Cela n'est pas très commode si l'on veut tester le caractère validé. Par exemple, la séquence d'instructions suivante :

```
char reponse;
while(1) {
	reponse = toupper(getchar());
	if (reponse == 'O' || reponse == 'N') break;
	printf("il faut repondre O ou N, recommence ! ");
}
```

ne fonctionne pas correctement car, chaque fois que l'utilisateur tape un caractère erroné, il a deux fois le message il faut répondre O ou N, recommence.

Pour améliorer la saisie des valeurs d'un seul caractère, on peut, sous Windows, utiliser la fonction **getche**, qui effectue une saisie immédiate, sans validation. Malheureusement, ce n'est pas une fonction de la librairie standard : on ne peut pas l'utiliser, par exemple, sous Linux.

TD3. Un programme pour tester les entrées/sorties élémentaires

A partir du programme lecture_caracteres.c qui se trouve dans le projet td3 du répertoire TD3, testez les fonctions d'entrées/sorties scanf, getchar et getche.

Pour bien comprendre le fonctionnement de getche, lisez le fichier infos_sur_conio.txt, qui fait partie du projet. Pour voir quels codes sont renvoyés par les touches du clavier (touches fonctions et de déplacement du curseur en particulier), ouvrez le projet td3_b, étudiez le code source du programme codes_ascii.c et exécutez-le.

V. Les variables

V.1. La définition d'une variable

Une **définition de variable** précise le type, l'identificateur et éventuellement une ou plusieurs valeurs d'initialisation :

type identificateur [= valeur(s) d'initialisation]

Un **identificateur** est une suite de caractères formée avec des lettres, des chiffres et le caractère souligné _. Le premier caractère est obligatoirement une lettre. Les minuscules sont distinguées des majuscules.

La définition d'une variable correspond à la réservation de l'emplacement mémoire nécessaire à la représentation de la variable. La définition d'une variable ne doit pas nécessairement apparaître au début du bloc d'instructions dans lequel elle est définie, mais cela est fortement recommandé.

Une variable ne doit pas nécessairement être initialisée lors de sa définition ; cependant, **il est fortement recommandé de lui affecter une valeur avant de l'utiliser**.

Exemple

```
#include <stdio.h>
int main()
{
        int n=10;
        int l=30;
        int p=2*n;
        int m;
        printf("n=%d\t p=%d\t l=%d\n", n, p, l);
        printf("m=%d\n", m);
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
n=10        p=20        l=30
m=25468          //initialisation de la variable avec une valeur quelconque
```

V.2. Les instructions et la portée d'une variable

Une **instruction** en C est :

- une **instruction simple** toujours terminée par un point-virgule et pouvant être librement écrite sur plusieurs lignes ;

- une **instruction composée**, encore appelée **bloc d'instructions**, qui est une suite d'instructions placées entre accolades.

La portée d'une variable dépend de l'endroit où la variable est définie.

V.2.1. Les variables automatiques

Une variable définie dans un bloc d'instructions est une **variable locale temporaire**. En C, une variable locale temporaire a l'attribut auto (pour automatique) et est appelée **variable automatique**.

La portée d'une variable automatique va de la ligne de sa définition jusqu'à la fin du bloc d'instructions dans lequel elle est définie. Plus précisément, lors de l'exécution, une variable automatique est créée à l'entrée du bloc d'instructions dans lequel elle est définie et est détruite à la sortie de ce bloc.

Une variable automatique n'est jamais initialisée par défaut.

Exemple

```
#include <stdio.h>
int main()
{
        int n=10,m=20;
        int n=5;                        //erreur de compilation : redéfinition de n
        {
            float n=3.5; int m=3;       //redéfinition de n et de m : A EVITER
            int p=20;
            printf("n=%f", n) ;         //affiche 3.500000
            printf("m=%d", m) ;         //affiche 3
            printf("p=%d", p) ;         //affiche 20
       }
       printf("n=%f", n) ;              //affiche 10
       printf("m=%d", m) ;              //affiche 20
       printf("p=%d", p);               //erreur de compilation : p inconnu
}
```

V.2.2. Les variables externes

Une variable définie à l'extérieur de tout bloc d'instructions (y compris la fonction main) est une **variable globale**.

La portée d'une variable globale définie dans un fichier s'étend de l'endroit où elle est définie jusqu'à la fin du fichier. Plus précisément, une variable globale est créée au début de l'exécution du programme et est détruite à la fin du programme.

Une variable globale est initialisée par défaut à zéro.

La portée d'une variable globale peut être étendue à tout fichier dans lequel la variable est déclarée avec l'attribut extern. De ce fait, une variable globale en C est appelée **variable externe**.

Il est important de **distinguer la déclaration d'une variable de sa définition**. Comme nous l'avons déjà dit, la définition d'une variable correspond à la réservation de l'emplacement mémoire nécessaire à la représentation de la variable. La déclaration externe de cette variable ne la redéfinit pas : elle indique seulement les caractéristiques de la variable nécessaires à son utilisation. Une variable ne peut être déclarée avec l'attribut extern que s'il existe, par ailleurs, une définition de cette variable.

Exemple

Fichier A

```
int n=0;
float x=0;

void Somme()
{
    ...
    /*n et x sont accessibles*/
}
```

Fichier B

```
extern int n;

int main()
{
    ...
    /*n est accessible*/
}
```

V.2.3. Les variables statiques

Une **variable statique** est une variable déclarée avec l'attribut static.

Une variable statique est créée une seule fois au début de l'exécution du programme et est détruite à la fin du programme. S'il existe une ou des valeur(s) d'initialisation pour la variable, celle-ci n'est initialisée qu'une seule fois au début de l'exécution du programme et la variable conserve sa valeur. La mémorisation d'une variable statique au cours de l'exécution du programme est permanente.

Une variable statique peut être interne ou externe. La portée d'une variable statique interne est limitée au bloc d'instructions dans lequel elle est définie. La portée d'une variable statique externe est limitée au fichier dans lequel elle est définie.

Une variable statique est initialisée par défaut à zéro.

Exemple

```
#include <stdio.h>
int main() {
        int n=5, total=0;
        while(1) {
                static int i=1;          //variable statique initialisée une seule fois
                total=total+i;
                if(i==n) break;
                i=i+1;}
        printf("La somme des entiers de 1 a %d est %d\n",n,total);
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
La somme des entiers de 1 a 5 est 15
```

Exemple

Fichier A

```
int n=0;
static float x=0;
```

Fichier B

```
extern int n;
```

La portée de la variable statique x, définie dans le fichier A, est limitée au fichier A.

VI. Les opérateurs et les conversions

VI.1. Les opérateurs arithmétiques

Les opérateurs de calcul binaires sont **+ - * / %**.

Le quotient de deux entiers est un entier (5 / 2 = 2), alors que le quotient de deux nombres dont un réel est un réel (5.0 / 2 = 2.5).

L'opérateur **modulo**, noté **%**, ne peut porter que sur des entiers. Il calcule le reste de la division entière de son premier opérande par son second.

Exemple

11 % 4 = 3 ;

Il existe également un opérateur unaire **-**.

Les opérateurs binaires **+** et **-** ont la même priorité, celle-ci est inférieure à la priorité des opérateurs binaires *****, **/** et **%**, qui est elle même inférieure à celle de l'opérateur unaire **-**.

Exemple :

-2 * 4 + 3 = -5 ;

VI.2. Les conversions implicites de types

Il existe deux conversions implicites de types : les promotions numériques et les conversions d'ajustement de type.

Les **promotions numériques** convertissent automatiquement :
1. toutes valeurs de type short int ou char apparaissant dans une opération arithmétique en int ;
2. toutes valeurs de type float apparaissant dans une opération arithmétique en double.

Exemple

```
short int n=10,p=2;
char c='A';
n * p + c;          //promotions numériques de n et de p en int et multiplication
                    //promotion numérique de c en int et addition
                    //le résultat final est de type int et vaut 85
```

Les **conversions d'ajustement de type** (int → long int → float → double) sont automatiquement mises en œuvre par le compilateur, à condition qu'il n'y ait pas rétrécissement du domaine.

Exemple

```
int n=2;
long int p=2000;
float x=3.5;
n * p + x;      //conversion d'ajustement de n en long int, multiplication par p
                //promotion numérique de x en double
                /*le résultat de n*p est de type long int, conversion
                d'ajustement en double pour être additionné à x*/
                //le résultat final est de type double et vaut 4003.5
```

De plus, si, dans une opération arithmétique entre entiers, un des opérandes est du type unsigned, les autres opérandes sont convertis en unsigned et le résultat est de type unsigned.

VI.3. L'opérateur d'affectation

La forme générale de l'affectation est la suivante :

variable = expression

L'affectation renvoie la valeur de la variable affectée.

L'opérateur d'affectation possède une associativité de droite à gauche.

Exemple

```
int i=5;
int j=10;
i = j;                  //attribue la valeur de j à i : i=10
```

Des **conversions implicites** ont également lieu lors des affectations. La valeur du côté droit de l'affectation est convertie dans le type du côté gauche, qui sera le type du résultat.

Exemple

```
#include <stdio.h>
int main()
{
        int n=0;
        char c='A';
        float x=4.25;
        short int p=0;
        n = c;                  //n vaut 65
        n = x;                  //n vaut 4
        n = 3 * x;              //n vaut 12
        c = 20 * x + n;         //c est le caractère de code 97 : 'a'
        p = 30000 * n;          /*dépassement de capacité qui conduit à un
                                résultat erroné*/
}
```

VI.4. L'opérateur d'affectation élargie

L'affectation peut être combinée avec un opérateur arithmétique pour former un **opérateur combiné**. D'une manière générale, les affectations de la forme :

variable = variable opérateur expression

peuvent être combinées en :

variable opérateur= expression

Exemple

L'instruction i = i + 2 ; peut être remplacée par i += 2 ;

Le type d'une expression d'affectation correspond au type de l'opérande de gauche.

Exemple

```
#include <stdio.h>
int main()
{
        short int n=0;
        char c='A';
        float x=4.25;
        n + = c;                //n vaut 65, le résultat est de type short int
        x *= 3;                 //x vaut 12.75, le résultat est de type float
        c += 3;                 //c est le caractère de code 68 : 'D'
}
```

VI.5. Les autres opérateurs

VI.5.1. Les opérateurs relationnels

Les constantes VRAI et FAUX sont représentées par des entiers. Le type entier permet de représenter des valeurs booléennes pour lesquelles 0 est considéré comme « faux » et toutes les valeurs non nulles sont considérées comme « vraies », l'entier 1 représentant la valeur canonique « vrai ».

Les opérateurs relationnels sont **< <= > >= == !=**.

Les opérateurs **== !=** sont les opérateurs d'égalité « égal à » et « différent de ». Leur priorité est inférieure à celle des opérateurs de comparaison **< <= > >=**.

VI.5.2. Les opérateurs logiques

Les opérateurs logiques sont les opérateurs suivants classés par ordre de priorité décroissante :

Opérateurs	**Signification**	**Valeur de vérité**	**Remarques**
!	non	!cond : vrai si la condition est fausse	
&&	et conditionnel	cond1 && cond2 : vrai si les deux conditions sont vraies	*l'évaluation à faux de la condition est finie dès qu'elle devient fausse.*
||	ou conditionnel	cond1 || cond2 : vrai si au moins une des deux conditions est vraie	*l'évaluation à vrai de la condition est finie dès qu'elle devient vraie.*

VI.5.3. Les opérateurs d'incrémentation et de décrémentation

L'opérateur **++** permet d'ajouter 1 à son opérande. L'opérateur **--** permet de retrancher 1 à son opérande.

Les opérateurs **++** et **--** peuvent être postfixés ou préfixés.

Exemple

```
#include <stdio.h>
int main()
{
        int i = 3 ;
        int j = i++ ;          //j vaut 3 et i vaut 4
        int k = ++i ;          //k et i valent 5
}
```

VI.5.4. L'opérateur ternaire

Soient les instructions suivantes avec a, b et max trois variables numériques :

```
if (a > b)
    max = a;
else
    max = b;
```

Ces instructions calculent le maximum entre a et b. Le résultat est mis dans la variable max.
L'opérateur ternaire **? :** permet d'effectuer le même calcul et d'obtenir le même résultat :

```
max = (a > b) ? a : b;
```

L'opérateur ternaire **? :** permet d'écrire une **expression conditionnelle** qui fait intervenir trois opérandes et a la forme générale suivante :

condition ? expression_1 : expression_2

L'exécution de cette expression conditionnelle évalue d'abord condition. Si le résultat de cette évaluation est vraie, alors expression_1 est évalué et la valeur ainsi calculée est retournée comme résultat de l'expression conditionnelle. Sinon, expression_2 est évalué et sa valeur est retournée comme résultat de l'expression conditionnelle.

Le type du résultat d'une expression conditionnelle suit les lois de conversions implicites de types présentées ci-dessus.

Exemple

```
short int x=2;
float y=0;
(x > 0) ? x+y : y;        //promotions numériques de x en int et de y en double
                          /*conversion d'ajustement de x en double pour être
                          additionné à y*/
```

Exemple

```
#include <stdio.h>
int main()
{
        char reponse='\0';
        printf("Voulez-vous continuer (O/N): ");
        reponse = toupper(getche());
        if((reponse == 'O' || reponse == 'N' )
              printf("\nVous avez repondu %s",(reponse=='O')? "Oui":"Non");
        return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Voulez-vous continuer (O/N): N
Vous avez repondu Non
```

VI.6. Les conversions explicites de types : l'opérateur cast

Le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide de l'opérateur **cast**.

L'opérateur **cast** a la forme générale suivante :

(nom_type) expression

L'expression est convertie dans le type précisé entre parenthèses selon les lois de conversions déjà présentées.

L'opérateur **cast** a une grande priorité.

Exemple

```
#include <stdio.h>
#include <math.h>
int main()
{
	float x=4.25;
	int n=81;
	printf("x=%f\n",x);
	printf("x=%d\n",x);
	printf("x=%d\n",(int)x);
	printf("x=%d\n",(int)x*x);
	printf("x=%d\n",(int)(x*x));
	printf("x=%d\n",(short int)(x*10000));
	printf("La racine carree de %d est %f",n,sqrt((double)n));
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
x=4.250000
x=0
x=4
x=0
x=18
x=-23036                    //dépassement de capacité (42 500 > 32 767)
La racine carree de 81 est 9
```

Pour rappel, le format général de la fonction printf est le suivant :

printf(format, val_1, val_2, …, val_n)

- val_1, val_2, …, val_n représentent les valeurs à afficher ;
- format est une chaîne de caractères contenant les codes de mise en forme associés, dans l'ordre, aux arguments val_i de la fonction printf.

Il est important de noter que la correspondance entre les codes de mise en forme et les valeurs à afficher doit être exacte, aussi bien en ce qui concerne le type qu'en ce qui concerne le nombre.

La fonction sqrt de la bibliothèque math.h prend en argument un nombre de type double et retourne la racine carrée de ce nombre, le résultat étant de type double.

VI.7. Les opérateurs de manipulation de bits

Les opérateurs de manipulation de bits permettent de travailler directement sur le motif binaire d'une valeur. Ils ne peuvent porter que sur des types entiers.

Opérateurs	Signification
&	et (bit à bit)
|	ou inclusif (bit à bit)
^	ou exclusif (bit à bit)
<<	décalage à gauche (en gardant le signe)
>>	décalage à droite (en gardant le signe)
~(unaire)	complément à un (bit à bit)

VI.7.1. Les opérateurs bit à bit

Table de vérité des opérateurs « bit à bit »

Opérande 1 **Opérande 2**	**0** **0**	**0** **1**	**1** **0**	**1** **1**
et (&)	0	0	0	1
ou inclusif (|)	0	1	1	1
ou exclusif (^)	0	1	1	0

L'opérateur « bit à bit » unaire ~ se contente d'inverser chacun des bits de son unique opérande (0 donne 1 et 1 donne 0).

Exemple

n p	00000101 00000011
n & p	00000001
n | p	00000111
n ^ p	00000110
~ n	11111010

VI.7.2. Les opérateurs de décalage

Les opérateurs de décalage permettent de réaliser des décalages à droite ou à gauche sur le motif binaire correspondant à leur premier opérande. L'amplitude du décalage, exprimée en nombre de bits, est fournie par le second opérande.

- `n << 2` décalage du motif binaire de `n` de 2 bits vers la gauche ; les bits de gauche sont perdus et des bits à zéro apparaissent à droite.
- `n >> 3` décalage du motif binaire de `n` de 3 bits vers la droite ; les bits de droite sont perdus et des bits apparaissent à gauche. Ces derniers sont identiques au bit de signe du motif d'origine : on dit qu'il y a propagation du bit de signe.

Exemple

n	10000101
n << 2	00010100
n >> 3	11110000

VI.8. Un récapitulatif sur les priorités des opérateurs

La liste des opérateurs classés par ordre de priorité décroissante et accompagnés de leur mode d'associativité est la suivante :

Opérateurs	**Associativité**
( ) [ ] . ->	de gauche à droite
! ~ -(unaire) ++ -- (cast) sizeof * &	de droite à gauche
* / %	de gauche à droite
+ -	de gauche à droite
<< >>	de gauche à droite
< <= > >=	de gauche à droite
== !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
|	de gauche à droite
&&	de gauche à droite
||	de gauche à droite
? :	de droite à gauche
= += -= *= /= %= <<= >>= &= |= ^=	de droite à gauche

L'opérateur ( ) permet d'appeler une méthode.
L'opérateur [ ] permet de faire référence à un élément d'un tableau.
Les opérateurs . et -> permettent de faire référence à un membre d'un objet.
L'opérateur sizeof permet de connaître la taille d'un objet en bits.
Les opérateurs * et & sont utilisés avec les pointeurs.

TD4. Un programme pour convertir un caractère en minuscule ou en majuscule

Ecrivez un programme qui demande à l'utilisateur de saisir un caractère et le convertit soit en minuscule si le caractère donné est une majuscule, soit en majuscule si le caractère donné est une minuscule.

Le programme bouclera tant que l'utilisateur souhaitera continuer.

Pour convertir un caractère CarLu en minuscule, vous pourrez utiliser l'instruction suivante : CarLu = CarLu + 'a' - 'A' ;

VII. Les instructions de contrôle

VII.1. Les énoncés conditionnels

VII.1.1. L'instruction if else

```
if (condition)
        instruction_1
[ else                          //partie facultative
        instruction_2 ]
```

Les instructions instruction_1 et instruction_2 sont soit des instructions simples, soit des blocs d'instructions.

Le programme évalue condition. Si le résultat de cette évaluation est vraie, alors il exécute l'instruction instruction_1, sinon il exécute l'instruction instruction_2, lorsqu'elle existe.

Les instructions **if** peuvent être imbriquées.

Exemple

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char c='\0';
    do {
       printf("Entrez un caractere (ENTREE pour sortir): ");
       c=getche();
       if(c>='0' && c<='9')
          printf("\nVous avez tape l'entier %c\n",c);
       else if(c>='A' && c<='Z')
                printf("\nVous avez tape la lettre majuscule %c\n",c);
            else if(c>='a' && c<='z')
                     printf("\nVous avez tape la lettre minuscule %c\n",c);
                 else if(c=='\r')
                          printf("\nAu revoir. A bientot.\n");
                      else
                          printf("\nVous avez tape le caractere %c\n",c);
    }
    while(c!='\r');
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Entrez un caractere (ENTREE pour sortir): Y
Vous avez tape la lettre majuscule Y
Entrez un caractere (ENTREE pour sortir): r
Vous avez tape la lettre minuscule r
Entrez un caractere (ENTREE pour sortir): 7
Vous avez tape l'entier 7
Entrez un caractere (ENTREE pour sortir):
Au revoir. A bientot.
```

VII.1.2. L'instruction switch

Lorsqu'une expression conditionnelle ne se réduit pas à une alternative, on peut utiliser l'instruction switch.

```
switch (expression)
    { case expressionconstante _1 : [suite_instructions_1]
      case expressionconstante _2 : [suite_instructions_2]
      …
      case expressionconstante _n : [suite_instructions_n]
      [ default : suite_d'instructions ]                    //cas facultatif
    }
```

suite_instructions_i, 1≤i≤n, est soit une instruction simple, soit une suite d'instructions séparées par des « ; ».

L'instruction switch évalue expression, qui doit être de type entier ou caractère, et compare sa valeur avec celle des différents cas.

Chaque cas est étiqueté par une expression constante expressionconstante_i, qui doit être de type entier ou caractère.

Si l'étiquette d'un cas, par exemple expressionconstante_i, correspond à la valeur de l'expression testée expression, alors les instructions correspondantes suite_instructions_i sont exécutées. L'exécution continue en séquence jusqu'à la fin de l'instruction switch ou jusqu'à ce qu'une instruction break soit rencontrée.

Si aucun cas ne correspond, les instructions du cas default, si elles existent, sont exécutées. Si aucun cas ne correspond et si le cas default n'existe pas, alors aucune instruction n'est exécutée.

Les différents cas ainsi que le cas default peuvent être écrits dans n'importe quel ordre. Ils doivent tous être différents.

Exemple

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char c='\0';
    do {
       printf("Entrez un entier positif (ENTREE pour sortir): ");
       switch(c=getche())
       {
          case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case (char)9+'0':printf("\nVous avez tape l'entier %c\n",c);break;
          case '\r': printf("\nAu revoir. A bientot.\n");break;
          default:printf("\n%c n'est pas un entier positif !\n",c);
       }
    }
    while(c!='\r');
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Entrez un entier positif (ENTREE pour sortir): 2
Vous avez tape l'entier 2
Entrez un entier positif (ENTREE pour sortir): 9
Vous avez tape l'entier 9
Entrez un entier positif (ENTREE pour sortir): u
u n'est pas un entier positif !
Entrez un entier positif (ENTREE pour sortir):
Au revoir. A bientot.
```

VII.2. Les énoncés itératifs

VII.2.1. La boucle while

while (condition) instruction

L'instruction instruction est soit une instruction simple, soit un bloc d'instructions. Elle est couramment appelée **corps** de la boucle.

Le programme évalue condition. Si le résultat de cette évaluation est vrai, alors il exécute le corps de la boucle, puis évalue à nouveau condition. Le corps de la boucle est exécuté jusqu'à ce que la valeur de condition devienne fausse.

Exemple

```
#include <stdio.h>
#include <ctype.h>
int main()
{
        char reponse='\0';
        printf("\nVoulez-vous continuer (O/N): ");
        while ((reponse = toupper(getche()))!='O' && reponse != 'N' )
                printf("\nVoulez-vous continuer (O/N): ");
        printf("\nVous avez repondu %s\n",(reponse == 'O')?"Oui":"Non");
        system("pause"); return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Voulez-vous continuer (O/N) : u
Voulez-vous continuer (O/N) : n
Vous avez repondu Non
```

VII.2.2. La boucle do while

do instruction **while** (condition);

L'instruction instruction est soit une instruction simple, soit un bloc d'instructions. Elle est couramment appelée **corps** de la boucle.

Le programme exécute le corps de la boucle, puis évalue condition. Si le résultat de cette évaluation est vrai, alors il exécute à nouveau le corps de la boucle et ainsi de suite. La boucle s'arrête lorsque la valeur de condition devient fausse.

Contrairement à la boucle while, le corps de la boucle do while est toujours exécuté au moins une fois.

Exemple

```
#include <stdio.h>
#include <ctype.h>
int main()
{
	char reponse='\0';
	do {
		printf("\nVoulez-vous continuer (O/N): ");
	} while ((reponse = toupper(getche()))!='O' && reponse != 'N' );
	printf("\nVous avez repondu %s\n",(reponse == 'O')?"Oui":"Non");
	system("pause"); return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Voulez-vous continuer (O/N) : u
Voulez-vous continuer (O/N) : n
Vous avez repondu Non
```

VII.2.3. La boucle for

La boucle for contient trois parties séparées par des point-virgules, chaque partie étant facultative.

for ([initialisation] ; [condition] ; [incrémentation])
    instruction

L'instruction instruction est soit une instruction simple, soit un bloc d'instructions. Elle est couramment appelée **corps** de la boucle.

La première partie initialisation est une expression ou une suite d'expressions séparées par des virgules. Elle permet d'initialiser les variables utilisées dans le corps de la boucle. Les variables doivent être déclarées en dehors de la boucle.

La troisième partie incrémentation est une expression ou une suite d'expressions séparées par des virgules. Elle permet d'incrémenter ou de décrémenter les variables utilisées dans le corps de la boucle.

Le programme initialise les variables utilisées dans le corps de la boucle et évalue condition. Si le résultat de cette évaluation est vrai, alors il exécute le corps de la boucle, incrémente ou décrémente les variables utilisées dans le corps de la boucle puis évalue à nouveau condition. Le corps de la boucle est exécuté jusqu'à ce que la valeur de condition devienne fausse.

Exemple

```
for(int i=1,total=0,float x=0;i<=n;i++)          //erreur à la compilation

int i=0,total=0;
for(i=1,total=0, x=0;i<=n;i++,x++)               //ok, mais rarement utilisé
```

L'instruction for est équivalente à :

```
initialisation
while(condition)
{
    instruction
    incrémentation
}
```

La boucle for(; ;) est une boucle infinie.

Exemple

```
#include<stdio.h>
int main()
{
	int i=0;
	int n=0;
	int total=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&n);
	for(i=1;i<=n;i++)
		total = total + i;
	printf("La somme des entiers de 1 a %d est %d\n",n,total);
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif: 5
La somme des entiers de 1 a 5 est 15
```

VII.3. Les instructions de branchement inconditionnel

VII.3.1. L'instruction break

L'instruction break permet de sortir d'un switch ou d'un énoncé itératif.

Exemple

```
#include<stdio.h>
const int EntierMax = 2147483647;
int main()
{
	int n=0,i=0,total=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&n);
	for(i=1; i<=n;i++)
	{
		total = total+i;
		if(total >= EntierMax)
		{
			total = 0;
			printf("Depassement de capacite !\n");
			break;
		}
	}
	if (total!=0)
		printf("La somme des entiers de 1 a %d est %d\n",n,total);
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif: 30000
Depassement de capacite !
```

VII.3.2. L'instruction continue

L'instruction continue permet de passer prématurément à l'itération suivante.

Exemple

```
#include<stdio.h>
int main()
{
	int n=0,i=0,total_impair=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&n);
	for(i=1;i<=n;i++)
	{
		if (i%2 == 0) continue ;
		total_impair = total_impair + i;
	}
	printf("La somme des entiers impairs de 1 a %d est %d\n",n,total);
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif: 5
La somme des entiers impairs de 1 a 5 est 9
```

TD5. Un programme pour compter le nombre de caractères, mots et lignes

Ecrivez un programme qui permet de compter le nombre de caractères, mots et lignes frappés au clavier.

Le programme s'arrêtera lorsque l'utilisateur tapera, en début de ligne, un caractère spécial de votre choix.

Vous utiliserez la fonction getchar() de la bibliothèque stdio.h pour lire les caractère frappés au clavier. Cette fonction retourne le caractère lu.

Contrairement à la fonction getche() de la bibliothèque conio.h, la fonction getchar() permet de lire le caractère spécial '\n', qui est une combinaison de deux caractères : le caractère « retour chariot » (caractère '\r') qui permet d'aller au début de la ligne et le caractère « passage à la ligne » qui permet de passer à la ligne suivante.

Voici un exemple d'exécution du programme.

```
Programme qui compte le nombre de caracteres, mots et lignes frappes au clavier
Tapez 0 pour arreter le programme
aaa   bbb ccc     ddd
        eee
  fff
0
nombre de caracteres : 18
nombre de mots : 6
nombre de lignes : 3
```

VIII. Les fonctions

VIII.1. Qu'est-ce qu'une fonction ?

Une fonction est une sorte de « boîte noire » qui permet d'effectuer une ou plusieurs opérations et que l'on peut utiliser sans se préoccuper de la façon dont sont effectuées ses opérations.

Exemple

```
#include<stdio.h>

int Somme(int n)
{
	int i=0,total=0;
	for(i=1;i<=n;i++)
		total = total + i;
	return (total);
}

int main()
{
	int x=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&x);
	printf("La somme des entiers de 1 a %d est %d\n",x,Somme(x));
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif: 5
La somme des entiers de 1 a 5 est 15
```

L'instruction return permet de renvoyer le résultat de la fonction Somme, c'est à dire la somme des n premiers entiers, à la fonction main.

VIII.2. La définition d'une fonction

Une **définition de fonction** comprend deux parties :

- un **en-tête** qui précise le type renvoyé par la fonction, le nom (identificateur) de la fonction et ses arguments (paramètres) formels. L'en-tête d'une fonction correspond à son **interface**, c'est à dire à ce qui sera visible de l'extérieur ;

- un **corps** qui contient l'**implémentation** de la fonction, c'est à dire les instructions à exécuter à chaque appel de la fonction.

```
TypeRenvoyé NomFonction (Type_1 arg_1, ..., Type_n arg_n)
{
        instruction
}
```

Une fonction peut renvoyer ou ne pas renvoyer de résultat, dans ce cas le type renvoyé par la fonction est void.

L'instruction return permet de renvoyer le résultat de la fonction au programme appelant : return (expression).
Si aucune expression ne suit l'instruction return, alors aucune valeur n'est transmise au programme appelant, mais ce dernier reprend la main.
Il faut qu'il y ait correspondance (selon les règles de conversions implicites de types) entre le type du résultat renvoyé par l'instruction return et le type renvoyé par la fonction, qui est déclaré dans son en-tête.

Une fonction peut avoir de 0 à n arguments formels.

Une fonction se définit à l'extérieur de toute autre fonction : on ne peut pas emboîter des définitions de fonctions.

Exemple

```
#include<stdio.h>

void AfficherSomme()
{
	int n=0,i=0,total=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&n);
	for(i=1;i<=n;i++)
		total = total + i;
	printf("La somme des entiers de 1 a %d est %d\n",n,total);
	system("pause");
}

int main()
{
	AfficherSomme();
	return 0;
}
```

VIII.3. La portée d'une fonction et sa déclaration

Une fonction est nécessairement définie à l'extérieur de tout bloc d'instructions. Elle est un **objet global**.

La portée d'une fonction définie dans un fichier s'étend de l'endroit où elle est définie à la fin du fichier.

La portée d'une fonction peut être étendue à tout fichier dans lequel elle est déclarée. De ce fait une fonction en C est un **objet global externe**.

Une **déclaration** de fonction s'effectue en écrivant l'en-tête de la fonction, terminée par un point-virgule.
TypeRenvoyé NomFonction (Type_1 Param_1, ..., Type_n Param_n)**;**

Il est important de **distinguer la déclaration d'une fonction de sa définition**. La définition d'une fonction décrit la manière d'exécuter cette fonction. La déclaration externe de cette fonction reprend uniquement l'en-tête de la fonction pour indiquer son type renvoyé, son nom et ses arguments formels.

Exemple

Fichier A

```
int Somme(int n)
{ ... }

void AfficherSomme()
{
    ...
    /*Somme est accessible*/
}
```

Fichier B

```
int Somme(int n);

int main()
{
    ...
    /*Somme est accessible*/
}
```

La fonction Somme définie dans le fichier A peut être utilisée (appelée) dans le fichier B. Le fichier B n'a pas besoin de connaître l'implémentation de la fonction Somme, mais il doit contenir une déclaration de cette fonction.

VIII.4. La transmission des arguments « par valeur »

En C, la **transmission** d'un argument à une fonction a toujours lieu **par valeur**. Plus précisément, lors d'un appel de fonction par un programme, les arguments sont transmis par recopie de leur valeur.

On appelle **arguments formels** les arguments figurant dans l'en-tête de la définition d'une fonction, et **arguments effectifs** les arguments fournis lors de l'appel de la fonction.

Une fonction ne peut pas modifier la valeur d'un argument effectif. En effet, la fonction reçoit une copie de la valeur de l'argument effectif et toute modification effectuées sur cette copie n'a aucune incidence sur la valeur de l'argument effectif.

Exemple

```
#include<stdio.h>

void Echange(int a, int b)
{
	int temp=0;
	temp = a; a = b; b = temp;
}

int main()
{
	int n=2, p=5;
	Echange(n, p);
	printf("n=%d \t p=%d \n",n,p);
	return 0;
}
```

L'appel de la fonction Echange dans la fonction main laisse les arguments effectifs n et p inchangés.

VIII.5. Les fonctions récursives

Les fonctions peuvent être utilisées de manière récursive, c'est à dire qu'une fonction peut s'appeler elle-même soit directement, soit indirectement.

Exemple

```
#include<stdio.h>

int Factorielle(int n)
{
	int Fact=0;
	if(n>1)
		Fact = n * Factorielle(n-1);
	else
		Fact=1;
	return Fact;
}

int main()
{
	int n=0;
	printf("Donnez un entier positif: ");
	scanf("%d",&n);
	printf("La factorielle de %d est %d\n",n,Factorielle(n));
	return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier positif: 5
La factorielle de 5 est 120
```

TD6. Une fonction pour lire une ligne

Ecrivez une fonction qui permet de lire une ligne de longueur maximale donnée.

int LireLigne(char *ch,int nbmax)

La fonction LireLigne lit les caractères frappés au clavier jusqu'à ce que :

- soit elle rencontre le caractère « retour chariot » (caractère '\r') ;
- soit le nombre de caractères déjà lus est égal à nbmax.

Vous utiliserez la fonction getche pour lire chaque caractère de la ligne.

Le résultat sera stocké dans la chaîne ch passée en argument, à laquelle sera ajouté le caractère de fin de chaîne '\0'.

La fonction LireLigne retournera le nombre de caractères lus.

Vous écrirez également une fonction main pour tester la fonction LireLigne.

IX. Le préprocesseur

Dans la phase de préparation du programme avant son exécution, le travail du compilateur est précédé par le traitement du préprocesseur. Le préprocesseur interprète les directives, qui permettent de modifier le programme avant la traduction proprement dite par le compilateur.

Une **directive** est écrite sur une seule ligne et commence par le caractère #. Elle n'est pas terminée par un point virgule.

Pour écrire une directive sur plusieurs lignes, il faut terminer chaque ligne qui n'est pas la dernière par le caractère '\'.

IX.1. L'inclusion d'un fichier

La directive #include permet d'inclure le contenu d'un fichier dans le programme.
#include "NomFichier"
Le fichier NomFichier est cherché dans le répertoire courant.

Lorsque le fichier à inclure se trouve dans un des répertoires contenant les bibliothèques prédéfinies, les guillemets sont remplacés par les caractères '<' et '>'.
#include <NomBibliothèque>

IX.2. La définition d'une macro

La directive #define permet de définir deux types de macros : des macro substitutions et des macro instructions.

IX.2.1. Les macro substitutions

Les macro substitutions sont des macros sans argument, qui permettent de définir des constantes.
#define NomMacro ValeurDeSubstitution

Le préprocesseur remplace, dans le programme, chaque nom de macro NomMacro par la chaîne de caractères ValeurDeSubstitution qui lui est associée.

Exemple
#define Taille_Maximale 500

IX.2.2. Les macro instructions

Les macro instructions sont des macros avec arguments, qui permettent d'effectuer une ou plusieurs opérations.
#define NomMacro(arg_1, arg_2, …, arg_n) ExpressionEquivalente

Le préprocesseur remplace, dans le programme, chaque appel de la macro NomMacro par la chaîne de caractères obtenue à partir de ExpressionEquivalente, en remplaçant chaque argument formel de la macro par l'argument effectif fourni lors de l'appel de la macro.

Exemple
```
#define Chiffre(Car) (Car-'0')

#define Max(A,B) (((A) > (B)) ? (A) : (B))
```

Exemple

```
#include <stdio.h>
#include <stdlib.h>
#define FinMain() return 0;
#define Ecrire(mes,ch) printf(mes,ch);

int main()
{
        char reponse;
        do {
                printf("\nVoulez-vous continuer (O/N): ");
       } while ((reponse = toupper(getche()))!='O' && reponse != 'N' );
        Ecrire("\nVous avez repondu %s\n",(reponse == 'O')?"Oui":"Non");
        FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Voulez-vous continuer (O/N): y
Voulez-vous continuer (O/N): N
"Vous avez repondu Non"
```

IX.3. D'autres directives

#undef NomMacro	*//annule la définition de la macro* NomMacro
#if condition	*//évalue* condition
#ifdef NomMacro	*//teste si la macro* NomMacro *a une définition* *//est équivalent à* #if defined NomMacro
#ifndef NomMacro	*//teste si la macro* NomMacro *n'a pas de définition* *//est équivalent à* #if !defined NomMacro
#elif condition	*//sinon si d'une directive* #if *: permet d'évaluer une autre condition* condition
#else	*//sinon d'une directive* #if
#endif	*//fin d'une directive* #if, #ifdef *ou* #ifndef

Exemple

```
#ifndef Taille_Minimale
	#define Taille_Minimale 0
#endif
#if Taille_Minimale == 0
	#ifndef Taille_Maximale
		#define Taille_Maximale 50
	#endif
#elif Taille_Minimale == 10
	#ifndef Taille_Maximale
		#define Taille_Maximale 500
	#endif
#else
	#undef Taille_Minimale
	#ifdef Taille_Maximale
		#undef Taille_Maximale
	#endif
#endif
```

TD7. Les macros

1- Ecrivez une macro Maj qui convertit un caractère en majuscule à l'aide de l'opérateur ? : et testez la dans un programme.

2- Quel est le résultat de l'exécution de ce programme ?

```
#include<stdio.h>
#define Max(A,B) (((A) > (B)) ? (A) : (B))
int main()
{
	int x=3,y=5;
	int res=Max(x++,y++);
	printf("x=%d \t y=%d \t res=%d\n",x,y,res);
}
```

3- Quel est le résultat de l'exécution de ce programme ?

```
#include<stdio.h>
#define Carre(n) n*n
int main()
{
	int x=3;
	int res=Carre(x+1);
	printf("x=%d \t res=%d\n",x,res);
}
```

X. La structure d'un programme

Un programme C est composé d'un ou plusieurs fichiers sources, qui peuvent être compilés séparément et exécutés en même temps, tout en utilisant des fonctions appartenant à des bibliothèques pré compilées.

X.1. Les caractéristiques d'un programme C

Un **programme C** est composé d'un ou plusieurs **fichiers sources**. Chaque fichier source contient une partie de l'ensemble du programme C ; c'est à dire des définitions d'**objets** comme des constantes, des variables, des définitions de types et des fonctions.

Un des fichiers sources doit contenir la fonction main, qui constitue le point d'entrée du programme C. Elle représente le code à exécuter en premier.

Les fichiers sources ont souvent des **fichiers d'en-tête** associés comportant des déclarations d'objets (fonctions) et des définitions d'objets (constantes et définitions de types), qui pourront être utilisés dans d'autres fichiers sources.

Rappel

La **définition** d'un objet décrit complètement l'objet. La **déclaration** d'un objet indique seulement les caractéristiques de l'objet, dont on doit trouver la définition ailleurs, dans le même ou un autre fichier source.

Schéma général d'un fichier source

```
#include          /*directive pour inclure des déclarations et des
                   définitions d'objets provenant d'un autre fichier*/

#define           /*directive pour définir des macros*/
…                 /*déclarations d'objets globaux définis dans un autre
                   fichier source et utilisés dans le programme*/
…                 /*définitions d'objets globaux utilisés dans le
                   programme*/

int main()        /*fonction qui doit se trouver dans un des fichiers
{ ... }            sources du programme*/
```

X.2. La programmation modulaire

Chaque fichier qui compose un programme est appelé un **module**.

Dans un projet de programmation modulaire réalisé en C, un des modules contient la fonction main. Pour chacun des autres modules, on a :

- des objets globaux (constantes et définitions de types composés) et des interfaces de fonctions dans un fichier d'en tête (de suffixe h) ;

- des objets globaux (variables) et des implémentations de fonctions dans un fichier source (de suffixe c) ou mieux dans un fichier compilé (de suffixe obj).

X.3. Un programme-exemple

Le programme suivant permet de lire un entier caractère par caractère et retourne l'entier lu. Le signe, s'il est donné, doit être saisi au début. Ensuite, la saisie de tout caractère qui n'est pas un chiffre est refusée avec un bruitage (le caractère n'est pas affiché à l'écran). La saisie s'arrête lorsque le caractère '\r' (retour chariot) est frappé ou lorsque la valeur de l'entier saisi a dépassé la valeur maximale 2 147 483 647.

Le projet projProgModulaire.dev (suffixe propre à l'environnement de développement Dev-C++) est composé de deux modules :

- un module MainLitEntier.c qui contient la fonction main ;
- un module LitEntier.c auquel est associé un fichier d'en tête LitEntier.h.

Le fichier d'en tête LitEntier.h est inclus dans chacun des deux modules MainLitEntier.c et LitEntier.c.

Fichier LitEntier.h

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

#define EntierMax 2147483647
#define Bruit '\a'
#define Entree '\r'
#define Chiffre(Car) (Car -'0')
#define FinMain() return 0;

int LireEntier();
```

Fichier LitEntier.c

```
#include "LitEntier.h"

int LireEntier()
{
        //fonction qui lit un entier chiffre par chiffre et rend la valeur de l'entier lu
}
```

Fichier MainLitEntier.c

```
#include "LitEntier.h"

int main()
{
        printf("L'entier donne en entree est %d\n",LireEntier());
        FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez un entier inferieur a 32767: +56
L'entier donne en entree est 56
```

X.4. La compilation d'un programme C

Chaque fichier source est traité indépendamment par un **compilateur C**, qui traduit le code du programme C en un **code objet** propre à l'ordinateur, qui n'est, en principe, pas lisible.

Dans la phase de compilation, le **pré-processeur** précède le compilateur pour exécuter les directives, ce qui permet de modifier le programme source avant la traduction proprement dite.

Certaines erreurs du programme C peuvent être détectées par le compilateur qui fournit alors un message d'erreur et interrompt sa traduction.

Exemple

Dans le projet projProgModulaire.dev on peut compiler :

- d'une part le module MainLitEntier.c, ce qui permet d'obtenir le fichier MainLitEntier.obj ;
- d'autre part le module LitEntier.c, ce qui permet d'obtenir le fichier LitEntier.obj.

Une fois que la mise au point des modules est satisfaisante, on peut fournir à l'utilisateur le fichier LitEntier.h et le fichier LitEntier.obj qui représente la partie cachée et inviolable du programme.

X.5. L'édition de liens

Lorsque tous les fichiers sources sont compilés, les codes objets produits sont présentés à un programme appelé **éditeur de liens**. L'éditeur de liens permet de résoudre les références entre les codes objets et ajoute les fonctions standards de bibliothèques pré-compilées nécessaires pour effectuer des opérations spéciales telles que les entrées/sorties.

L'éditeur de lien produit un **programme exécutable** unique.

Certaines erreurs de programmation, telles que la non définition d'une fonction utilisée, sont détectées par l'éditeur de liens et provoquent des messages d'erreurs.

Exemple

L'exécution du projet projProgModulaire.dev permet d'obtenir le programme exécutable projProgModulaire.exe.

TD8. Une fonction pour lire un entier chiffre par chiffre

Ecrivez la fonction LireEntier() qui permet de lire un entier non nécessairement positif (gestion du signe -) chiffre par chiffre et qui rend la valeur de l'entier lu.

Vous utiliserez les deux fonctions suivantes :

- la fonction getch(), de la bibliothèque conio.h, qui lit le caractère frappé au clavier sans l'afficher à l'écran et retourne le caractère lu ;

- la fonction putch(int c), de la bibliothèque conio.h, qui affiche à l'écran le caractère c passé en argument et retourne le caractère affiché.

La fonction LireEntier() lit un entier chiffre par chiffre à l'aide de la fonction getch() comme suit :
- soit le caractère lu est un entier et il est affiché à l'écran à l'aide de la fonction putch(),
- soit le caractère lu n'est pas un entier, il est alors refusé avec un bruitage à l'aide de l'instruction putch(Bruit).

Le signe s'il est donné doit être saisi au début.

La saisie de l'entier s'arrête lorsque :
- soit l'utilisateur a frappé le caractère '\r',
- soit l'entier saisi est supérieur à 2147483647.

XI. Les pointeurs

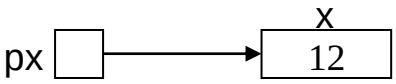
XI.1. Qu'est-ce qu'un pointeur ?

Un **pointeur** est une variable qui contient l'adresse d'une autre variable.

Exemple

```
int x=12;
int *px;
px = &x;
```

px est un pointeur qui contient l'adresse de la variable entière x. On dit que px « pointe » sur x.



L'opérateur unaire & affecte l'adresse de x à la variable px.

XI.2. La définition d'un pointeur

Une définition de pointeur précise le type du pointeur.

typeP *NomPointeur;

*NomPointeur est une variable du type typeP.

Les pointeurs sont typés. Un pointeur sur un type typeP ne pourra pointer que sur des variables de ce type.

Exemple

```
int x=12;
int *px;
float *pf;
px = &x;
pf=&x;              //erreur à la compilation
```

XI.2.1. L'opérateur unaire &

L'opérateur unaire & donne l'adresse d'un objet. Il ne peut être employé qu'avec des variables et des éléments de tableaux.

&NomVariable renvoie l'adresse de la variable NomVariable.

Exemple

```
int x=12;
int *px,*py;
px=&x;          //affecte l'adresse de x à la variable px
py=&12;         //erreur à la compilation
```

XI.2.2. L'opérateur unaire *

L'opérateur unaire * donne le contenu de l'objet dont son opérande donne l'adresse.

*NomPointeur renvoie le contenu de la variable pointée par NomPointeur.

Exemple

```
int x=12, y=0, *px;
px = &x;
y=*px;          //met dans y le contenu de l'adresse indiquée par px, soit 12
```

*px peut remplacer x partout où on pourrait trouver x, et ce tant que la valeur de px n'est pas modifiée.

Exemple

```
int x=12, y=0, *px;
px = &x;
y=*px + 2;      /*met dans y le contenu de l'adresse indiquée par px auquel
                est ajouté 2, soit 14 ; instruction équivalente à y=x+2;*/
*px = *px + 1;  //incrémente x de 1 ; instruction équivalente à x=x+1;
```

XI.2.3. La constante NULL

Un pointeur dont la valeur est NULL ne pointe sur rien.

Exemple

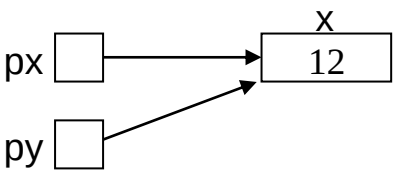
```
int *px;
if(px==NULL)
        printf("Le pointeur ne pointe sur rien\n");
else
        printf("Le pointeur pointe sur %d\n",*px);
```

XI.3. L’affectation entre deux pointeurs

Une affectation entre deux pointeurs porte sur les variables pointées.

Exemple

```
int x=12;
int *px,*py;
px = &x;
py=px;
```



XI.4. Le type de pointeur générique et les conversions de types entre les pointeurs

Le langage C propose un **type de pointeur générique**, le type void*, qui permet à un pointeur de pointer sur n'importe quel type de variable.

Exemple

```
int x=12,y=0,*py;
float z=2.5,w=0,*pf,*pfbis;
void *pgenInt,*pgenFloat;

conversion void * → int * : OK
pgenInt=&x;
y=*pgenInt;                 //erreur à la compilation
y=(int)(*pgenInt);          //erreur à la compilation
y=*(int*)pgenInt;           //met dans y le contenu de pgenInt, soit 12

conversion int * → (void * → int *) : OK
py=pgenInt;
y=*py;                      //met dans y le contenu de py, soit 12

conversion void * → float * : OK
pf=&z;
pgenFloat=pf;
w=*(float*)pgenFloat;       //met dans w le contenu de pgenFloat, soit 2.5

conversion float * → (void* → int*) : NON
pf=pgenInt;                 //aucune erreur détectée, mais dangereux
w=*pf;                      //w vaut 0.000000

conversion int * → (void* → float*) : NON
py=pgenFloat;               //aucune erreur détectée, mais dangereux
y=*py;                      //met dans y une valeur quelconque
```

XI.5. Les pointeurs utilisés comme arguments des fonctions

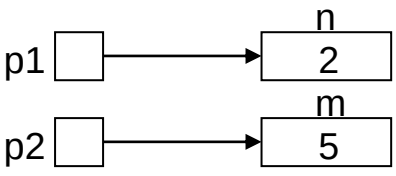
Nous avons vu précédemment que la transmission d'un argument à une fonction a lieu par valeur et qu'une fonction ne peut pas modifier la valeur de cet argument dans le programme appelant.

Les pointeurs permettent de transmettre un argument à une fonction **par référence**.
Pour transmettre un argument par référence à une fonction, on fournit explicitement un pointeur sur cet argument et la fonction peut alors changer l'argument que le pointeur désigne dans le programme appelant.

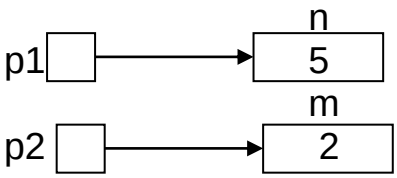
Exemple

```
#include<stdio.h>
#define FinMain() return 0;
void Echange(int *p1, int *p2) {
        int temp = *p1; *p1 = *p2; *p2 = temp; }
int main() {
        int n=2, m=5;
        Echange(&n, &m);
        printf("n=%d \t m=%d \n",n,m);
        FinMain();
}
```

En début d'exécution



A la fin de l'exécution

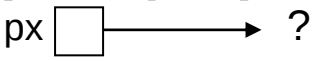


L'unique manière d'avoir des arguments résultats est donc de les passer par référence. Ces arguments résultats sont généralement utilisés lorsque la fonction doit renvoyer plus d'un résultat.

```
void Fonction(Type1 *Arg_Resultat_1, Type2 *Arg_Resultat_2) {...}
int main() {
        Type1 Var1; Type2 Var2;...
        NomFonction(&Var1, &Var2); }
```

XI.6. La création et la suppression de variables dynamiques

Lorsque l'on définit un pointeur par TypeP *NomPointeur;, on obtient un pointeur qui ne pointe sur rien.



On peut ensuite affecter à ce pointeur l'adresse d'une variable existante de type TypeP.

Une autre façon d'initialiser un pointeur est de lui allouer dynamiquement un espace mémoire.

XI.6.1. L'allocation dynamique d'un emplacement mémoire

La fonction malloc permet d'allouer dynamiquement un espace mémoire à un pointeur :
TypeP *NomPointeur = (TypeP*) malloc (sizeof(TypeP));

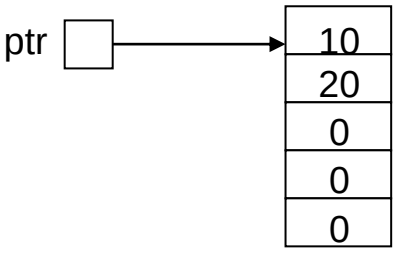
Cette instruction effectue les trois opérations suivantes :

1. un appel de la fonction malloc, qui alloue un emplacement mémoire de la taille d'un objet de type TypeP et renvoie l'adresse de cet emplacement, qui est du type void* ;
2. une conversion de l'adresse renvoyée par la fonction malloc dans le type du pointeur TypeP* ;
3. une affectation de l'adresse obtenue au pointeur NomPointeur.

Si, lors de l'appel de la fonction malloc, la mémoire disponible n'est pas suffisante, alors l'allocation n'est pas effectuée et la fonction malloc renvoie NULL.

Exemple

```
int *ptr;
ptr=(int*) malloc(5*sizeof(int));       //ptr pointe sur une zone de 5 entiers
if (ptr==NULL)
        exit(0);
memset(ptr,0,5*sizeof(int));  //la fonction memset permet d'initialiser
                                        //chaque entier de la zone de 5 entiers à 0
ptr[0]=10;
ptr[1]=20;
```



La fonction malloc est déclarée dans les bibliothèques stdlib.h et malloc.h comme suit : (void*) malloc (size_t taille); où size_t est le type du résultat retourné par l'opérateur sizeof, opérateur qui retourne la taille d'un objet (variable ou type) en nombre de bits.

XI.6.2. La libération d'un emplacement mémoire alloué dynamiquement

La fonction free permet de libérer un emplacement mémoire alloué par la fonction malloc.
free(NomPointeur);

Un emplacement mémoire alloué par la fonction malloc doit toujours être libéré à l'aide de la fonction free.

Attention, il ne faut jamais utiliser le contenu d'un pointeur après sa libération.

Exemple

```
ptr=(int*) malloc(5*sizeof(int));
if (ptr==NULL)
        exit(0);
memset(ptr,0,5*sizeof(int));
ptr[0]=10;
ptr[1]=20;
free(ptr);          //supprime le tableau d'entiers pointé par ptr
x=*ptr;             //met dans x une valeur quelconque
```

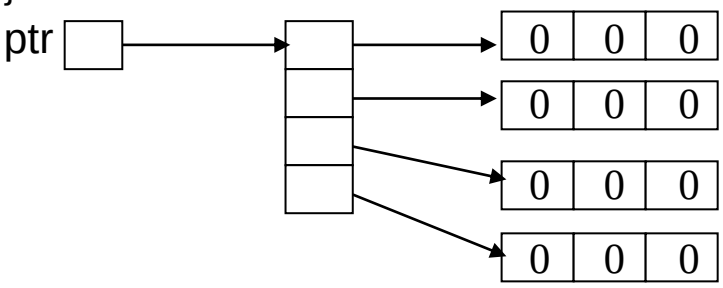
La fonction free est déclarée dans les bibliothèques stdlib.h et malloc.h.

XI.6.3. Les pointeurs de pointeurs

On peut réserver un espace mémoire à un pointeur tel que chaque élément de la zone est lui-même un pointeur.

Exemple

```
int **ptr;
int i=0,nblignes=4,nbcolonnes=3;
ptr=(int**) malloc(nblignes*sizeof(int*));
if (ptr==NULL)
        exit(0);
for(i=0;i<nblignes;i++)
{
        ptr[i]=(int*) malloc(nbcolonnes*sizeof(int));
        if (ptr[i]==NULL)
                exit(0);
        memset(ptr[i],0,nbcolonnes*sizeof(int));
}
```




```
for(i=0;i<nblignes;i++)
        free(ptr[i]);
free(ptr);
```

XII. Les tableaux

XII.1. La déclaration d'un tableau

Une **déclaration** de tableau précise le nombre d'éléments du tableau et leur type :

type NomTableau[dimension]

On appelle **taille** d'un tableau, le nombre d'éléments du tableau.

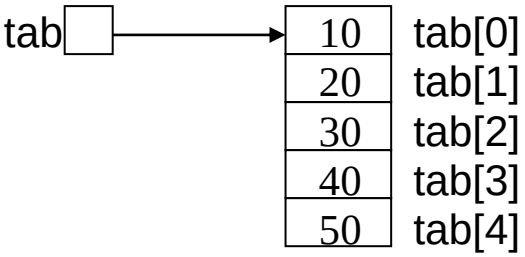
Les éléments d'un tableau sont indexés de 0 à dimension–1.

L'opérateur [] permet de désigner un élément du tableau. NomTableau[i] désigne le $i^{ème}$ élément de NomTableau. Aucune vérification n'est faite par le compilateur sur la valeur de i, qui doit être compris entre 0 et dimension–1.

Exemple

```
int tab[5];
tab[0]=10;
tab[1]=20;
tab[2]=30;
tab[3]=40;
tab[4]=50;
```

tab désigne un tableau de 5 entiers.



XII.2. L'initialisation d'un tableau d'entiers ou de réels

Un tableau peut être initialisé en faisant suivre sa déclaration d'une liste de valeurs entre accolades :

type NomTableau[n] = {valeur_init_1, ..., valeur_init_m}

Les règles suivantes sont valables dans l'environnement de développement Dev-C++.

S'il y a plus de valeurs d'initialisation que d'éléments dans le tableau (m>n), alors il y a une erreur à la compilation.

S'il y a moins de valeurs d'initialisation que d'éléments dans le tableau (m<n), alors les éléments non initialisés du tableau (qui sont du type entier ou réel) sont initialisés à 0.

S'il n'y a aucune valeur d'initialisation, les éléments d'un tableau (d'entiers ou de réels) ne sont pas initialisés.

Si la taille d'un tableau n'est pas précisée lors de son initialisation, alors le compilateur lui affecte la taille correspondant au nombre de valeurs d'initialisation (m).

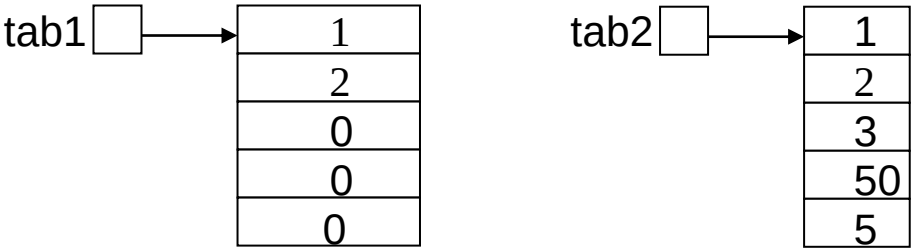
Exemple

```
float tabFloat[]={1.5,2.2,3.1,4,5.8};        //tableau de réels de taille 5
```

Les tableaux de caractères sont des tableaux particuliers qui permettent de représenter des chaînes de caractères. Nous les étudierons plus loin.

Exemple

```
#include<stdio.h>
#define FinMain() return 0;

int main()
{
    int i=0;
    int tab1[5];
    memset(tab1,0,5*sizeof(int));
    tab1[0]=1; tab1[1]=2;

    int tab2[5]={1,2,3,4,5};
    tab2[10/3]=50;          //équivalent à tab2[3]=50
    tab2[-10]=40;           /*indexation incorrecte mais aucune erreur n'est
                            détectée*/

    int tab3[5]={1,2,3,4,5,6};      /*plus de valeurs d'initialisation
                                    erreur à la compilation*/
    FinMain();
}
```

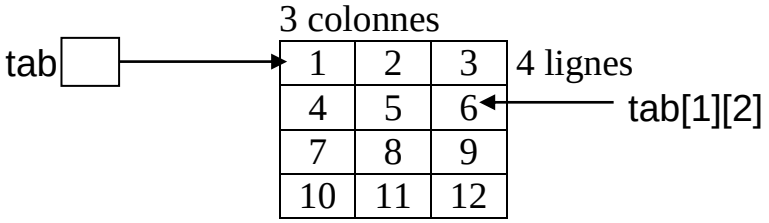


XII.3. Les tableaux à plusieurs dimensions

Le langage C permet de créer des tableaux de tableaux, c'est à dire des tableaux dont les éléments sont eux-mêmes des tableaux.

Exemple

int tab [4] [3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} };
tab est un tableau à 4 éléments, où chaque élément est un tableau de 3 entiers. On dit que tab est une matrice composée de 4 lignes et de 3 colonnes.



tab[i][j] désigne l'élément de la ligne i+1 et de la colonne j+1.

Exemple

```
int tab2 [4] [3] [2] = {    { {1,2}, {3,4}, {5,6} },
                            { {7,8}, {9,10}, {11,12} },
                            { {13,14}, {15,16}, {17,18} },
                            { {19,20}, {21,22}, {23,24} }     }
tab2[1][2][0] = 50;         //l'ancienne valeur de tab2[1][2][0] était 11
```

XII.4. Les pointeurs et les tableaux

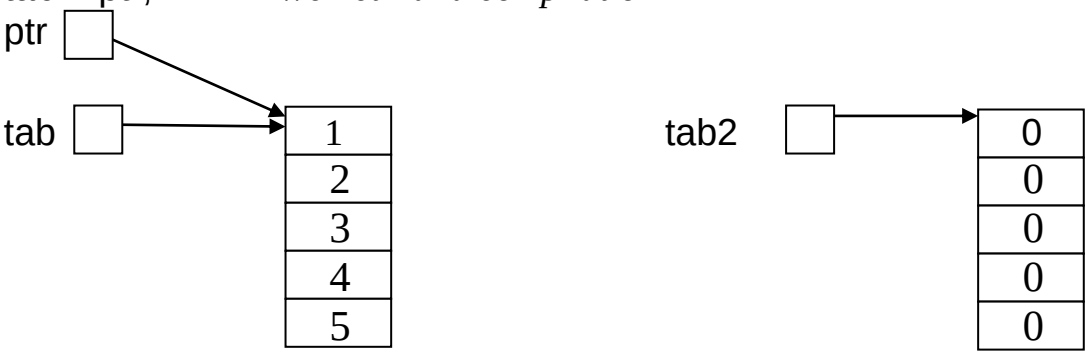
XII.4.1. Les tableaux, des pointeurs constants

Le nom d'un tableau est un pointeur constant qui pointe sur son premier élément.

Un pointeur peut pointer sur un tableau, l'inverse est faux.

Exemple

```
int tab[5]={1,2,3,4,5};
int tab2[5];
memset(tab2,0,5*sizeof(int));
int x=0,*ptr;
ptr=tab;
x=*ptr;          //met dans x le contenu de l'adresse indiquée par ptr, soit 1
tab2=tab;        //erreur à la compilation
tab2=ptr;        //erreur à la compilation
```

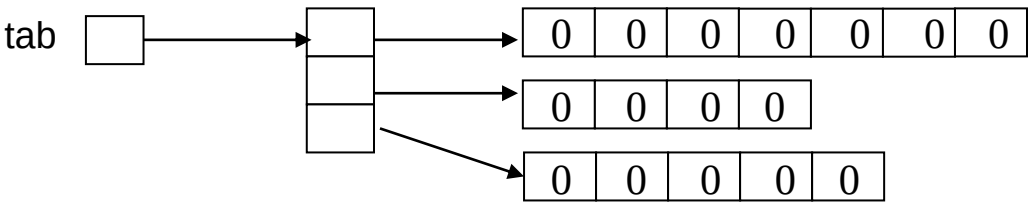


On remarquera que l'affectation ptr=tab; est équivalente à ptr=&tab[0];.

Exemple

```
int *tab[3];
tab[0]=(int*) malloc(7*sizeof(int)); memset(tab[0],0,7*sizeof(int));
tab[1]=(int*) malloc(4*sizeof(int)); memset(tab[1],0,4*sizeof(int));
tab[2]=(int*) malloc(5*sizeof(int)); memset(tab[1],0,5*sizeof(int));
```

tab est un tableau dont chaque élément est un tableau d'entiers de tailles différentes.



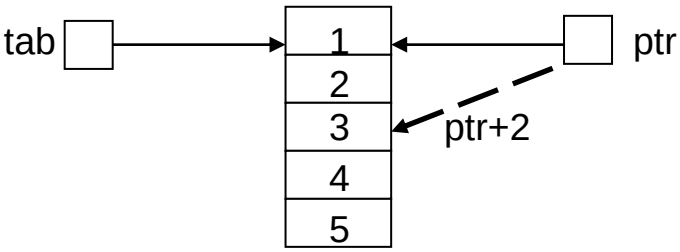
XII.4.2. Les opérations sur les pointeurs

a. Addition ou soustraction entre un pointeur et un entier

Lorsqu'un pointeur pointe sur un élément particulier d'un tableau, on peut lui ajouter ou retrancher un entier.

Exemple

```
int tab[5]={1,2,3,4,5},x=0,*ptr;
ptr=tab;
x=*(ptr+2);          //instruction équivalente à x=tab[2]; ou encore x=ptr[2];
```



Si un pointeur ptr pointe sur un élément particulier d'un tableau, alors ptr+i (ptr-i) pointe sur le $i^{ème}$ élément suivant (précédent) celui sur lequel pointe ptr. Plus précisément ptr+i (ptr-i) augmente (diminue) la valeur du pointeur de i multiplié par la taille des objets sur lesquels pointe ptr.

Exemple

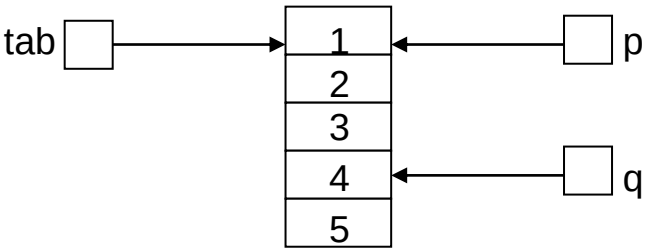
```
int tab[5]={10,20,30,40,50},x=0,y=0,z=0,*ptr;
void* pgen;
ptr=tab+2;           //instruction équivalente à ptr=&tab[2];
ptr++;
x=*ptr;              //x vaut 40
y=*(tab+1);          //y vaut 20
y=*tab+1;            //y vaut 11
y=*(tab++);          //erreur de compilation: tab est un pointeur constant
ptr=&tab[1];         //instruction équivalente à ptr=tab+1;
z=*(ptr+1);          //z vaut 30
z=ptr[3];            //z vaut 50
pgen=tab;
z=*(int*)(pgen+2);   /*erreur de compilation: la taille de l'objet pointé par
                       void* est inconnue*/
```

b. Soustraction entre deux pointeurs

Lorsque deux pointeurs pointent sur des éléments d'un même tableau, on peut faire une soustraction entre ces deux pointeurs. Cette soustraction calcule le nombre d'éléments séparant les deux pointeurs.

Exemple

```
int tab[5]={1,2,3,4,5};
int x=0,*p,*q;
p=tab;
q=&tab[3];
x=p-q;              //représente le nombre d'éléments entre p et q, soit -3
```



XII.4.3. Un programme-exemple

Le programme suivant permet de calculer la longueur d'une chaîne de caractères.

```
#include <stdio.h>
int LongueurChaine(char *s, int taillemax)
{
	char *p=s;
	int i=0;
	while (*p!='\0' && i<taillemax)
	{
		p++;
		i++;
	}
	return(p-s);
}
int main()
{
	char ch[50];
	memset(ch,'\0',50*sizeof(char));
	printf("Donnez une chaine de caracteres (de taille < 50): ");
	scanf("%s",ch);
	printf("La longueur de la chaine %s est %d.\n",ch,
	LongueurChaine(ch,50));
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez une chaine de caracteres : Bonjour
La longueur de la chaine Bonjour est 7.
```

XII.4.4. Les tableaux et la transmission par « référence »

Lorsqu'un nom de tableau est transmis à une fonction, c'est en fait l'adresse du premier élément du tableau qui est transmise. Un tableau est donc toujours transmis à une fonction par référence.

a. Les tableaux à une dimension

Un tableau à une dimension peut être l'argument d'une fonction acceptant en argument un pointeur. Réciproquement, un pointeur peut être l'argument d'une fonction acceptant en argument un tableau à une dimension.

En tant qu'argument formel dans la définition d'une fonction, les expressions `Type *NomVar` et `Type NomVar[]` sont donc équivalentes, la première expression étant la plus couramment utilisée. La fonction peut indifféremment les traiter comme des tableaux ou des pointeurs.

Lorsque l'argument formel d'une fonction est un tableau à une dimension, il est inutile de préciser la taille du tableau, puisque de toute façon le compilateur n'effectue aucune vérification de dépassement de capacité. Par contre, cette taille peut être donnée sous la forme d'un argument formel supplémentaire.

```
TypeRenvoyé NomFonction(Type *NomTableau,int TailleTableau)
```

Une fonction ne peut pas renvoyer un tableau de type `TypeR`, mais elle peut renvoyer un pointeur de type `TypeR`.

```
TypeR* NomFonction (Type_1 arg_1, ..., Type_n arg_n)
```

b. Les tableaux à plusieurs dimensions

Pour les tableaux à plusieurs dimensions, il n'y a pas de conversion possible avec les pointeurs.

Lorsque l'argument formel d'une fonction est un tableau à plusieurs dimensions, il faut préciser les n-1 dimensions les plus à droite.

```
TypeRenvoyé NomFonction(Type NomTableau[][3][2],int TailleTableau)
```

TD9. Un programme modulaire pour gérer des tableaux d'entiers

Ecrivez un programme qui permet de gérer des tableaux d'entiers. Le programme sera composé de deux modules :

- un module GestionTableaux.c, auquel est associé un fichier d'en-tête GestionTableaux.h, qui contient les fonctions suivantes :
  - une fonction void InsererValeur(int *T,int taillemax,int *taille, int elem) qui insère la valeur elem dans le tableau trié T, où taillemax est la taille maximale du tableau saisi et taille sa taille effective ;
  - une fonction void SupprimerValeur(int *T, int *taille, int elem) qui supprime la ou les valeur(s) elem du tableau trié T de taille taille.
  - une fonction void AfficherTableau(int *T,int taille) qui affiche le tableau T de taille taille en mettant 3 valeurs par ligne ;
- un module ProgPrinc.c qui contient la fonction main. La fonction main permettra de saisir un tableau et de tester les fonctions définies ci-dessus à l'aide d'un menu comme défini ci-dessous.

```
#include"GestionTableaux.h"
#define TailleMax 50
int AfficherMenu()
{
        char rep='\0';
        printf("Gestion d'un tableau\n");
        printf("1- Afficher les elements du tableau\n");
        printf("2- Inserer une nouvelle valeur\n");
        printf("3- Supprimer une valeur\n");
        printf("4- Sortir du programme\n");
        while((rep=getche())<'1' || rep > '4') putch('\a');
        printf("\n"); return (rep-'0');
}
```

```
int main()
{
	int choix=0;
	while((choix=AfficherMenu())!=4)
	{
		switch(choix)
		{
			case 1 :		//afficher le tableau
				break;
			case 2 :		//inserer une valeur dans le tableau
				break;
			case 3 :		//supprimer une valeur du tableau
				break;
		}
	}
	return 0;
}
```

XIII. Les chaînes de caractères

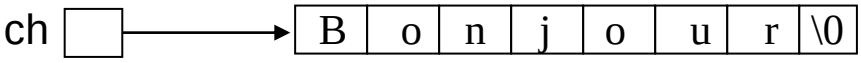
XIII.1. La déclaration et l'initialisation d'une chaîne de caractères

Une constante de type chaîne de caractères, telle que « Bonjour », est un tableau de caractères.

En C, une chaîne de caractères est terminée par le caractère spécial '\0', qui indique la fin de la chaîne. La place qu'occupe en mémoire une chaîne de caractères de taille n est donc n+1.

Exemple

```
char tabch[10];       //chaine pouvant contenir jusqu'à 9 caractères significatifs
char *ch = "Bonjour";   //chaine de taille 8 : à éviter
```



XIII.2. Des fonctions standards de manipulation des chaînes de caractères

La bibliothèque string.h propose de nombreuses fonctions pour manipuler des chaînes de caractères. Ces fonctions commencent toutes par le préfixe « str ».

XIII.2.1. Longueur d'une chaîne de caractères

int strlen(char *ch);

La fonction strlen retourne la longueur de la chaîne ch.

Exemple

```
#include <stdio.h>
#include <string.h>
#define FinMain() return 0;
int main()
{
	char *ch="Bonjour";
	printf("longueur = %d\n",strlen(ch));		//affiche 7
	FinMain();
}
```

XIII.2.2. Copie de chaînes de caractères

char* strcpy(char *dest,char *source);

La fonction strcpy copie le contenu de la chaîne source dans la chaîne dest et retourne un pointeur sur la chaîne dest. La chaîne dest doit être suffisamment grande pour contenir la chaîne source.

Exemple

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
        char ch1[20];
        memset(ch1,'\0',20* sizeof(char));
        char *ch2=(char*) malloc(20*sizeof(char));
        memset(ch2,'\0',20* sizeof(char));
        char *ch3;
        strcpy(ch1,"Bonjour");          //copie "Bonjour" dans ch1
        strcpy(ch2,"Bonjour");          //copie "Bonjour" dans ch2
        strcpy(ch3,"Bonjour");          //erreur à l'exécution
}
```

La fonction suivante permet de copier une chaîne de caractères source dans une chaîne de caractères destination.

```
char* copie_version1(char *source)
{
        int i=0;
        char *dest=(char *) malloc((strlen(source)+1)*sizeof(char));
        while(source[i]!= '\0')
        {
                dest[i]=source[i];
                i=i+1;
        }
        dest[i] = '\0';
        return dest;
}
```

XIII.2.3. Concaténation de chaînes de caractères

char* strcat(char *ch1,char *ch2);

La fonction strcat concatène la chaîne ch2 à la chaîne ch1 et retourne un pointeur sur la chaîne ch1. La chaîne ch1 doit être suffisamment grande pour contenir les deux chaînes.

Exemple

```
#include <stdio.h>
#include <string.h>
int main()
{
        char ch1[30];
        memset(ch1,'\0',30* sizeof(char));
        strcpy(ch1,"Bonjour");
        char ch2[10];
        memset(ch2,'\0',10* sizeof(char));
        strcpy(ch2," Monsieur");
        strcat(ch1,ch2);                    //retourne "Bonjour Monsieur"
}
```

XIII.2.4. Comparaison de chaînes de caractères

int strcmp(char *ch1,char *ch2);

La fonction strcmp retourne 0 si les chaînes ch1 et ch2 sont égales, une valeur positive si la chaîne ch1 est supérieure du point de vue lexicographique à la chaîne ch2 et une valeur négative sinon.

Exemple

```
#include <stdio.h>
#include <string.h>
int main()
{
        char ch1[10];
        memset(ch1,'\0',10* sizeof(char));
        strcpy(ch1,"Bonjour");
        char ch2[10];
        memset(ch2,'\0',10* sizeof(char));
        strcpy(ch2,"Bonsoir");
        int comp=strcmp(ch1,ch2);              //retourne –1 (ch1<ch2)
        comp=strcmp(ch2,ch1);                  //retourne 1
        comp=strcmp(ch1,"Bonjour");            //retourne 0
}
```

XIII.2.5. Recherche d'un caractère dans une chaîne de caractères

char* strchr(char *ch,int carac);

La fonction strchr retourne un pointeur sur la première apparition du caractère carac dans la chaîne ch et NULL si le caractère carac n'apparaît pas dans la chaîne ch.

Exemple

```
#include <stdio.h>
#include <string.h>
int main()
{
        char ch1[10];
        memset(ch1,'\0',10* sizeof(char));
        strcpy(ch1,"Bonjour");
        char *ch2;
        ch2=strchr(ch1,'o');                    //ch2 pointeur sur "onjour"
}
```

XIII.2.6. Un programme-exemple

Le programme suivant permet de compter la fréquence d'apparitions de chaque lettre dans une chaîne de caractères.

```
#include <stdio.h>
#include <stdlib.h>
#define FinMain() return 0;

int main()
{
	char ch[50],c='\0';
	memset(ch,'\0',50* sizeof(char));
	int i=0,j=0,Freq[26];
	memset(Freq,0,26* sizeof(int));
	printf("Donnez une chaine de caracteres : ");
	scanf("%s",ch);
	while(ch[i]!='\0')
	{
		c= toupper(ch[i])-'A';
		if(c>=0 && c<26)
			Freq[c]++;
		i=i+1;
	}
	for(i=0;i<26;i++)
		if(Freq[i]>0)
			printf("La lettre %c a ete rencontree %d fois\n",i+'A',
			Freq[i]);
	FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez une chaine de caracteres : Bon3JOUR
La lettre B a ete rencontree 1 fois
La lettre J a ete rencontree 1 fois
La lettre N a ete rencontree 1 fois
La lettre O a ete rencontree 2 fois
La lettre R a ete rencontree 1 fois
La lettre U a ete rencontree 1 fois
```

La fonction int toupper(int c) de la bibliothèque stdlib.h convertit le caractère c passé en argument en majuscule. Elle return 0 sinon.

XIII.3. Un exemple sur les conversions de type entre les pointeurs

```
char c='\0';
char ch[10];
memset(ch,'\0',10* sizeof(char));
strcpy(ch,"Bonjour");
char *pch;
int tab[5] = {65,66,67,68,69};
int x=0,*px;
void *pgenChar;

c=*(ch+2);                  //met dans c le contenu de ch[2], soit n

conversion void * → char * : OK
pgenChar=ch;
c=*(char*)pgenChar;         //met dans c le contenu de pgenChar, soit B

conversion char * → (void * → char *) : OK avec cast
pch=pgenChar;
c=*(pch+3);                 //met dans c le contenu de pch+3, soit j

conversion char * → int * : NON
pch=(char*)tab;             //aucune erreur détectée, mais dangereux
c=*pch;                     //met dans c le contenu de tab[0], soit A
c=*(pch+3);                 //met dans c une valeur quelconque

conversion int * → char * : NON
px=(int*)ch;                //aucune erreur détectée, mais dangereux
x=*px;                      //met dans x une valeur quelconque
```

TD10. Des fonctions de manipulation des chaînes de caractères

Ecrivez les trois fonctions suivantes :

- une fonction int Palindrome(char *ch) qui permet de tester si une chaîne de caractères est un palindrome. Les mots palindromes sont des mots à symétrie bilatérale (qui s'épellent de la même façon dans les deux sens) ; par exemple RADAR, ROTOR ou encore ÉTÉ.
  Vous utiliserez deux pointeurs pour parcourir la chaîne de caractères ch passée en argument : un pointeur pour parcourir la chaîne du début à la fin et un pointeur pour parcourir la chaîne de la fin au début. La fonction renvoie 1 si la chaîne est un palindrome, 0 sinon.

- une fonction void InverserChaine(char *ch) qui permet d'inverser la chaîne de caractères ch passée en argument. Vous utiliserez la fonction Echange qui permet d'échanger la valeur de deux variables ;
  Exemple
  ```
  char ch[10];
  memset(ch,'\0',10* sizeof(char));
  strcpy(ch,"Bonjour");
  InverserChaine(ch);                    //ch vaut "ruojnoB"
  ```

- une fonction char* ExtraireChaine(char *ch,int lg) qui permet d'extraire de la chaîne ch passée en argument une chaîne correspondant à ses lg premiers caractères.
  Exemple
  ```
  char *ch=ExtraireChaine("Bonjour",3);          //ch vaut "Bon"
  ```

Vous écrirez également une fonction main pour tester ces trois fonctions.

XIII.4. Les arguments de la ligne de commande et la fonction main

XIII.4.1. Les arguments de la ligne de commande

Dans les systèmes d'exploitation utilisant le langage C, il est possible de transmettre directement des arguments dans la ligne de commande qui appelle un programme pour l'exécuter.

Exemple

somme 12 53 tapé sur la ligne de commande permet d'exécuter le programme somme et d'obtenir le résultat 65.

12 et 53 sont appelés les **arguments de la ligne de commande**.

Pour ce faire, il faut paramétrer la fonction main écrite dans un module somme.c.

XIII.4.2. Le paramétrage de la fonction main

La fonction main a l'en-tête suivant:
int main(int argc,char *argv[])

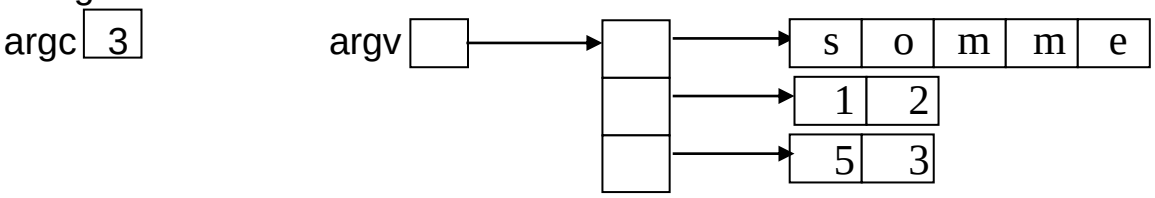
Sachant qu'une **ligne de commande** est une suite de chaînes de caractères séparées par des blancs, on a:

- argc est le nombre d'arguments de la ligne de commande, y compris le nom du programme appelé;

- argv est un tableau de chaînes de caractères contenant les différents arguments de la ligne de commande.

Exemple

Si l'on tape sur la ligne de commande : somme 12 53
alors, à l'appel de la fonction main du programme somme, les arguments argc et argv ont les valeurs suivantes :



Ainsi, si un programme est appelé dans la ligne de commande avec n arguments, alors :

- argc a la valeur n+1 ;

- argv est un tableau de n+1 chaînes de caractères où argv[0] pointe sur le nom du programme et argv[i], $1 \leq i \leq n$, pointe sur le $i^{ème}$ argument de la ligne de commande.

XIII.4.3. Le programme somme

Le programme suivant permet de calculer la somme d'au moins deux entiers.

Fichier somme.c

```
#include <stdio.h>
#include <stdlib.h>
#define FinMain() return 0;

int main(int argc,char *argv[])
{
	int somme=0,x=0;
	if(argc<3)
	{
		printf("Il faut donner au moins deux entiers pour calculer leur
		somme !\n");
		return 0;
	}
	else
		while(argc-- > 1)
		{
			if(!(x=atoi(argv[argc])))
			{
				printf("Un des arguments n'est pas un entier !\n");
				return 0;
			}
			somme +=x;
		}
	printf("La somme est %d\n",somme);
	FinMain();
}
```

Si l'utilisateur tape sur la ligne de commande somme 6 5 7, alors le résultat de l'exécution est :

```
La somme est 18
```

Dans l'environnement Dev-C++, les paramètres sont transmis à l'aide de l'option Parameters… du menu Execute.

La fonction int atoi(char *ch) de la bibliothèque stdlib.h permet de convertir la chaîne de caractères ch passée en argument en entier et retourne l'entier obtenu. Cette fonction retourne 0 si la conversion échoue.

XIV. Les types composés

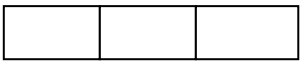
XIV.1. Qu'est ce qu'un type composé ?

Un type composé permet d'organiser des données complexes en offrant la possibilité de traiter un groupe de variables en un seul bloc.

Exemple

```
struct date {
        int jour;
        int mois;
        int annee;
        } ;
```

Type composé struct date :



XIV.2. La définition d'un type composé

Une **définition de type composé** précise les variables, appelées **membres**, qui le composent.

```
struct NomTypeComposé {
        Type_Membre1 NomMembre1;
        ...
        Type_Membre_n NomMembre_n;
};
```

Le nom du type composé est struct NomTypeComposé.

La définition d'un type composé peut directement être suivie d'une liste de variables qui seront du type struct NomTypeComposé.

```
struct NomTypeComposé {...} NomVar_1,..., NomVar_m;
```

Une variable de type composé peut être initialisée en faisant suivre sa définition d'une liste de valeurs entre accolades, qui initialise chaque membre du type composé.

```
struct NomTypeComposé NomVar = {val_1, ..., val_n}
```

L'opérateur « . » permet de désigner un membre d'une variable de type composé. NomVar.NomMembre_i désigne le $i^{ème}$ membre de la variable. L'opérateur « . » a la plus haute priorité.

Les types composés peuvent être imbriqués.

Exemple

```
struct Personne {
        char nom[20];
        char prenom[20];
        char adresse[100];
        struct date date_naissance;
} ;
```

On peut affecter une variable composée à une autre variable composée.

Exemple

```
struct Personne p1,p2={"Dupond","Andre","29 rue des Gobelins 75013",
{12,2,1995}};
p1=p2;
```

XIV.3. Un programme-exemple

```
#include <stdio.h>
#define jourToday 22
#define moisToday 5
#define anneeToday 2015
#define VerifierJour(jour) (((jour>=1) && (jour<=31))?1:0)
#define VerifierMois(mois) (((mois>=1) && (mois<=12))?1:0)
#define VerifierAnnee(annee) (((annee>=1900) && (annee<=3000))?1:0)
#define FinMain() return 0;

int main() {
	struct date {
		int jour;
		int mois;
		int annee;
	} d1;
	struct date d2={jourToday,moisToday,anneeToday};
	printf("La date d'aujourd'hui est le %d/%d/%d\n", d2.jour, d2.mois,
	d2.annee);
	printf("Donnez une date (jour, mois, annee separes par des blancs)
	: ");
	scanf("%d %d %d",&d1.jour,&d1.mois,&d1.annee);
	if((VerifierJour(d1.jour)==0) || (VerifierMois(d1.mois)==0) ||
	(VerifierAnnee(d1.annee)==0))
		printf("Erreur dans la date!\n");
	else
		printf("La date entree est le %d/%d/%d\n",d1.jour,d1.mois,
		d1.annee);
	FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
La date d'aujourd'hui est le 22/5/2006
Donnez une date (jour, mois, annee separes par des blancs) : 4 12 2005
La date entree est le 4/12/2005
```

XIV.4. Les tableaux de types composés

On peut définir des tableaux de types composés.

Exemple

```
struct Personne {
        char nom[20];
        char prenom[20];
        char adresse[100];
        struct date date_naissance;
} tab[20];
tab[0]={"Dupond","Andre","29 rue des Gobelins 75013", {12,2,1995}};
```

XIV.5. Les types composés et les pointeurs

Un pointeur peut pointer sur un type composé.

```
struct NomTypeComposé {
        Type_Membre1 NomMembre1;
        ...
        Type_Membre_n NomMembre_n;
} *NomPointeur;
```

L'opérateur -> (signe moins suivi de >) permet d'accéder aux membres d'un pointeur de type composé. NomPointeur->NomMembre_i désigne le $i^{ème}$ membre du pointeur. Cette expression est équivalente à (*NomPointeur).NomMembre_i.

Exemple

```
struct date {
        int jour;
        int mois;
        int annee;
        } d1={4,11,1988};
struct date *ptrDate,*ptrD;
void *pgen;
ptrDate=&d1;
ptrDate->jour=23;              //équivalent à (*ptrDate).jour=23
```

*conversion void * → struct date * : OK*

```
pgen=ptrDate;
int j=((struct date*)pgen)->jour;      //met dans j le jour de la date, soit 23
```

*conversion struct date * → (void * → struct date *) : OK*

```
ptrD=pgen;
int m=ptrD->mois;                      //met dans m le mois de la date, soit 11
```

L'opérateur -> a la plus haute priorité avec les opérateurs . , ( ) et [ ].

Exemple

```
++ptrDate->jour;               //incrémente jour et non pas ptrDate
```

XIV.6. Utiliser un type composé comme type d'un de ses membres

Un type composé peut contenir un membre qui est un pointeur sur lui-même.

Exemple

```
struct Personne {
        char nom[20];
        char prenom[20];
        char adresse[100];
        struct date date_naissance;
        struct Personne *pere;
        struct Personne *mere;
};
struct Personne p1={"Dupond","Lucien","29 rue des Gobelins 75013",
{30,6,1965},NULL,NULL};
struct Personne p2={"Dupond","Jeanne","29 rue des Gobelins 75013",
{15,12,1969},NULL,NULL};
struct Personne p3={"Dupond","Andre","29 rue des Gobelins 75013",
{12,2,1995},&p1,&p2};
```

XIV.7. Les unions

Une union est une variable qui peut contenir des objets de types et de tailles différentes, mais un seul à la fois.

Exemple

```
#include <stdio.h>
#include <string.h>
const char *tab_mois[] = {"janvier","fevrier","mars","avril","mai","juin",
"juillet","aout","septembre","octobre","novembre","decembre"};
#define nb_max_mois 12
#define FinMain() return 0;

void ChaineMinuscule(char s[]) {
        int i=0;
        for(i=0;i<=strlen(s);i++)
                if(s[i] >= 'A' && s[i] <='Z')
                        s[i] = s[i] + 'a' - 'A';
}

int ValiderNomMois(char s[]) {
        int i=0;
        ChaineMinuscule(s);
        for(i=0;i<nb_max_mois;i++)
                if(strcmp(s,tab_mois[i])==0)
                        return 1;
        return 0;
}

int ValiderNumeroMois(char s[]) {
        int n=0;
        if(s[0]<'0' || s[0]>'9') return 0;
        if(strlen(s)==1)
                n = s[0] - '0';
        else if((strlen(s)==2) && (s[1]>='0') && (s[1]<='9'))
                n = 10 * (s[0] - '0') + (s[1] - '0');
                else return 0;
        if(n<1 || n>12) return 0;
        return n;
}
```

```
int main()
{
	char s[10];
	int num=0;
	struct date {
		int jour;
		int type_mois;
		union {
			int num_mois;
			char *nom_mois;
		} mois;
		int annee;
	} d1;
	printf("Donnez une date -jour, mois (nom ou numero), annee-
	separes par des blancs) : ");
	scanf("%d %s %d",&d1.jour,s,&d1.annee);
	if(ValiderNomMois(s)==1)
	{
		d1.mois.nom_mois=s;
		d1.type_mois=1;
	}
	else if((num=ValiderNumeroMois(s))!=0)
	{
		d1.mois.num_mois=num;
		d1.type_mois=0;
	}
	else
	{
		printf("Le mois saisi est incorrect !\n");
		return 0;
	}
	printf("La date entree est le %d",d1.jour);
	printf((d1.type_mois)?" %s %d\n":"/%d/%d\n",d1.mois,d1.annee);
	FinMain()
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Donnez une date -jour, mois (nom ou numero), annee- separes par des blancs) : 4 10 2002
La date entree est le 4/10/2002
```

XIV.8. Les équivalences de types

Le langage C fournit une fonction typedef qui permet d'associer un nouveau nom à un type de donnée.

```
typedef TypeExistant TypeEquivalent;
```

Le type TypeEquivalent est un synonyme du type TypeExistant.

La définition d'une variable de type TypeExistant peut se faire à l'aide d'une des deux instructions suivantes, qui sont équivalentes :

```
TypeExistant NomVariable;
```

ou

```
TypeEquivalent NomVariable;
```

Exemple

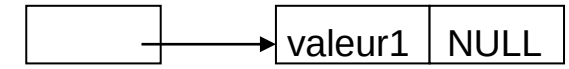
```
typedef char Chaine[20];
Chaine nom="Dupond";

struct Personne {
        Chaine nom;
        Chaine prenom;
        char adresse[100];
        struct date date_naissance;
};
typedef struct Personne Pers,TabPers[30];
Pers p1={"Dupond","Andre","29 rue des Gobelins 75013", {12,2,1995}};
TabPers classe;
classe[0]=p1;

typedef struct {
        int jour;
        int mois;
        int annee;
        } Date,*PtrDate;
Date d1={26,10,2002};
PtrDate pd1 = &d1;
```

TD11. Les piles

Une pile vide

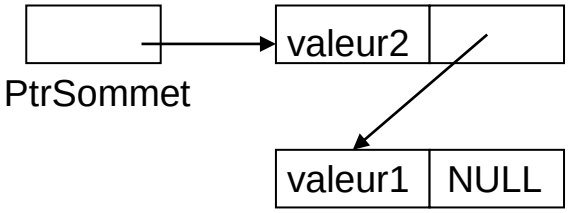
PtrSommet

On empile une valeur



PtrSommet

On empile une deuxième valeur



PtrSommet

Un élément d'une pile est composée d'une valeur et d'un pointeur sur l'élément précédent.

Soit TypeValeur le type des valeurs d'une pile, le type d'un élément d'une pile est le suivant :

```
struct Element {
        TypeValeur valeur;
        struct Element *precedent;
};
typedef struct Element TypeElementPile;
TypeElementPile *PtrSommet;
```

On suppose que les valeurs de la pile sont des entiers.

```
typedef int TypeValeur;
```

Question 1

Ecrivez un programme qui permet :

- d'afficher les éléments d'une pile ;
- empiler une nouvelle valeur donnée par l'utilisateur dans la pile ;
- dépiler une valeur de la pile ;
- vider la pile.

Vous n'utiliserez pas de fonction sauf pour l'affichage des éléments d'une pile :

```
void AfficherPile(TypeElementPile *ptrS)
{
        Si ptrS==NULL
                Afficher Pile vide
        Sinon
                Tant que ptrS !=NULL
                        Afficher ptrS->valeur
                        ptrS=ptrS->precedent
}
```

Le pointeur ptrS est-il modifié lors de l'appel de la fonction AfficherPile ?

Question 2

Reprennez le programme précédent en écrivant des fonctions pour empiler une valeur dans la pile, dépiler une valeur de la pile et vider la pile.

XV. Les flux et les fichiers

XV.1. Qu’est-ce qu’un flux ?

Un flux désigne un « canal » qui peut être connecté à différentes sources ou à différentes cibles : périphérique de communication, fichier, site distant…

Un **flux de sortie** désigne n'importe quel « canal » susceptible de recevoir de l'information sous la forme d'une suite d'octets (périphérique d'affichage comme l'écran, fichier, …). Il offre un service d'écriture.

Un **flux d'entrée** désigne n'importe quel « canal » susceptible de délivrer de l'information sous la forme d'une suite d'octets (périphérique de saisie comme le clavier, fichier, …). Il offre un service de lecture.

XV.2. Les flux standards

Quelque soit le système d'exploitation utilisé, le langage C dispose de trois flux standards :

- **le flux standard d'entrée stdin** : ce flux est généralement associé au clavier, sauf si le lancement du programme a indiqué une redirection ;

  Exemple (sous UNIX)

```
prog                 //le flux d'entrée du programme prog est le clavier
prog < entree        //le flux d'entrée du programme prog est le fichier entree
progavant | prog     /*le flux d'entrée du programme prog est associé au
                       flux de sortie du programme progavant*/
```

- **le flux standard de sortie stdout** : ce flux est généralement associé à l'écran, sauf si lors du lancement du programme, une redirection a été indiquée ;

  Exemple (sous UNIX)

```
prog                 //le flux de sortie du programme prog est l'écran
prog > sortie        //le flux de sortie du programme prog est le fichier sortie
```

- **le flux standard d'indication des erreurs stderr** : ce flux est généralement associé à l'écran, sauf si le lancement du programme a indiqué une redirection.

  Exemple (sous UNIX)

```
prog           //le flux d'indication des erreurs du programme prog est l'écran
prog 2> erreur /*le flux d'indication des erreurs du programme prog est
                 le fichier erreur*/
```

XV.3. L'ouverture et la fermeture d'un fichier

Un fichier est caractérisé par son identification, son emplacement, sa taille, sa date de création et éventuellement sa date de mise à jour.

XV.3.1. L'ouverture d'un fichier

Pour pouvoir utiliser un fichier, c'est à dire lire ou écrire dedans, il faut d'abord l'ouvrir. C'est le système d'exploitation qui se charge de l'ouverture d'un fichier et de sa gestion pendant son utilisation.

La fonction fopen de la bibliothèque stdio.h permet d'ouvrir un fichier et retourne un pointeur, de type composé FILE, sur le fichier ouvert. La fonction retourne NULL en cas d'échec lors de l'ouverture.

```
FILE* fopen(const char *NomFichier,const char *mode)
```

La chaîne de caractères constante mode permet de définir :

- le **type d'accès** au fichier en lecture et/ou écriture ;
- le **mode d'ouverture** du fichier qui peut être le mode texte ou le mode binaire.

XV.3.2. Les différents types d'accès à un fichier

Les différents types d'accès à un fichier sont les suivants :

"r" ouverture d'un fichier en lecture : le fichier doit exister, autrement la fonction fopen return NULL ;

"w" création et ouverture d'un fichier en écriture : si le fichier existe, son contenu est détruit ;

"a" ouverture d'un fichier en écriture à la fin du fichier : si le fichier n'existe pas, il est créé ;

"r+" ouverture d'un fichier en lecture et écriture : le fichier doit exister, autrement la fonction fopen return NULL ;

"w+" création et ouverture d'un fichier en lecture et écriture : si le fichier existe, son contenu est détruit ;

"a+" ouverture d'un fichier en lecture et en écriture à la fin du fichier : si le fichier n'existe pas, il est créé.

XV.3.3. Les fichiers binaires et les fichiers textes

Le langage C sous Windows propose deux modes d'ouverture d'un fichier :

- le **mode texte** ("t") où le fichier est considéré comme une suite de caractères ;
- le **mode binaire** ("b") où le fichier est considéré comme une suite d'octets.

Lors de l'ouverture d'un fichier en mode texte, deux mécanismes sont mis en oeuvre:

- en entrée, la combinaison des deux caractères « retour chariot » (caractère '\r', qui permet d'aller au début de la ligne) et « passage à la ligne » (qui permet de passer à la ligne suivante) est transformée en « passage à la ligne » (caractère '\n') ;
- en sortie le caractère « passage à la ligne » ('\n') est transformée en « retour chariot-passage à la ligne ».

Le mode d'ouverture par défaut des fichiers est donné dans la variable globale _fmode de la bibliothèque stdio.h. Par défaut, cette variable vaut O_TEXT, c'est à dire que les fichiers sont ouverts en mode texte. Pour que les fichiers soient ouverts par défaut en mode binaire, il faut attribuer la valeur O_BINARY à la variable globale _fmode.

XV.3.4. La constante EOF

La constante EOF, définie dans la bibliothèque stdio.h, caractérise la fin d'un fichier.

```
#define EOF -1
```

La fonction feof de la bibliothèque stdio.h retourne une valeur non nulle lorsque la fin d'un fichier est atteinte.

```
int feof(FILE *PtrFichier)
```

XV.3.5. La fermeture d'un fichier

Une fois que les opérations d'entrées/sorties vers un fichier sont terminées, il faut fermer le pointeur vers le fichier, à l'aide de la fonction fclose de la bibliothèque stdio.h.

```
int fclose(FILE *PtrFichier)
```

La fonction fclose retourne 0 si la fermeture s'est bien déroulée et EOF sinon.

XV.4. La manipulation d’un fichier texte

Un fichier texte est considéré comme une suite de caractères, éventuellement structurée en lignes de longueurs quelconques. Un fichier texte est en général manipulable à l’aide d’un éditeur de texte.

L’accès à un fichier texte en lecture et écriture peut se faire de deux façons :

- caractère par caractère,
- ligne par ligne.

XV.4.1. La manipulation d'un fichier texte caractère par caractère

a. Les fonctions fgetc et fputc

Les deux principales fonctions de la bibliothèque stdio.h pour lire et écrire dans un fichier texte caractère par caractère sont :

int fgetc(FILE *PtrFichier)
La fonction fgetc retourne le caractère lu dans le fichier pointé par PtrFichier sous la forme d'un entier. Cette fonction retourne EOF s'il s'est produit une erreur ou si la fin du fichier a été atteinte.

int fputc(int c,FILE *PtrFichier)
La fonction fputc écrit le caractère c passé en argument dans le fichier pointé par PtrFichier et retourne le caractère écrit. Cette fonction retourne EOF s'il s'est produit une erreur.

b. Un programme-exemple

Le programme-suivant permet de lire dans un fichier texte caractère par caractère.

```
#include<stdio.h>
#include<stdlib.h>
#define TailleLigneMax 80
#define FinMain() return 0;

int main()
{
	FILE *ptrf;
	int c=0,nbcarac=0,nblignes=1;
	char lignelue[TailleLigneMax];
	memset(lignelue,'\0',TailleLigneMax*sizeof(char));
	if((ptrf=fopen("FichierTexteTest.txt","r"))==NULL) {
		fprintf(stderr,"Erreur dans l'ouverture du fichier\n"); exit(1);
	}
	while((c=fgetc(ptrf)) != EOF) {
		if(c=='\n') {
			lignelue[nbcarac]='\0';
			printf("%d %s\n",nblignes++,lignelue);
			nbcarac=0;
		}
		else
			if(nbcarac>=TailleLigneMax) {
				printf("Ligne %d trop longue!\n",nblignes++);
				nbcarac=0;
				while(((c=fgetc(ptrf)) != '\n') && c!=EOF);
                                        //pour aller à la fin de la ligne
			}
			else
				lignelue[nbcarac++]=c;
	}
	if(c==EOF && nbcarac!=0) {
		lignelue[nbcarac]='\0'; printf("%d %s\n",nblignes,lignelue);
	}
	if(feof(ptrf)) printf("Fin de fichier\n");
	else	printf("Erreur lors de la lecture du fichier\n");
	fclose(ptrf); FinMain();
}
```

Soit le fichier FichierTexteTest.txt suivant :

```
Bonjour mes amis,

Comment allez-vous aujourd'hui?
Bien, j'espère.

A tres bientot.
Louis
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
1 Bonjour mes amis,
2
3 Comment allez-vous aujourd'hui?
4 J'espere bien.
5
6 A tres bientot.
7 Louis
Fin de fichier
```

La fonction exit(int n) de la bibliothèque stdlib met fin à l'exécution du programme et ferme tous les fichiers ouverts.

La fonction fprintf de la bibliothèque stdio.h est comparable à la fonction printf, mais elle permet en plus de préciser le flux de sortie.

Il existe également dans la bibliothèque stdio.h une fonction fscanf comparable à la fonction scanf, qui permet en plus de préciser le flux d'entrée.

Exemple

```
FILE *ptrf;
char lignelue[TailleLigneMax];
memset(lignelue,'\0',TailleLigneMax*sizeof(char));
ptrf=fopen("FichierTexteTest.txt","r")
fscanf(ptrf,"%s",lignelue);
```

XV.4.2. La manipulation d'un fichier texte ligne par ligne

a. Les fonctions fgets et fputs

Les deux principales fonctions de la bibliothèque stdio.h pour lire et écrire dans un fichier texte ligne par ligne sont :

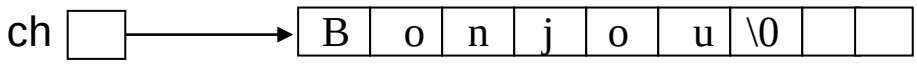
char* fgets(char *ch,int nbcarac,FILE *PtrFichier)
La fonction fgets lit des caractères dans le fichier pointé par PtrFichier jusqu'à ce que :

- soit elle rencontre le caractère « passage à la ligne » (caractère '\n') ;
- soit le nombre de caractères déjà lu est égal à nbcarac - 1.

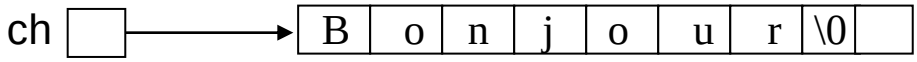
Le résultat est stocké dans la chaîne ch passée en argument, à laquelle est ajoutée le caractère de fin de chaîne '\0'.
La fonction fgets retourne NULL s'il s'est produit une erreur ou si la fin du fichier a été atteinte.

Exemple

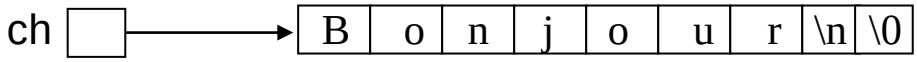
Soit ch un tableau de 9 caractères (char ch[9];). Si on tape "Bonjour\n",
avec l'instruction fgets(ch,7,stdin); on a :



avec l'instruction fgets(ch,8,stdin); on a :



avec l'instruction fgets(ch,9,stdin); on a :



int fputs(const char *ch,FILE *PtrFichier)
La fonction fputs écrit la chaîne de caractère ch passée en argument dans le fichier pointé par PtrFichier et retourne une valeur non nulle s'il n'y a pas eu d'erreur lors de l'écriture ou EOF sinon.

b. Un programme-exemple

Le programme-suivant permet de copier le contenu d'un fichier source dans un fichier destination ligne par ligne.

```
#include<stdio.h>
#include<stdlib.h>
#define TailleLigneMax 80
#define FinMain() return 0;

int main()
{
	FILE *ptrfsource=NULL,*ptrfdest=NULL;
	char lignelue[TailleLigneMax];
	memset(lignelue,'\0',TailleLigneMax*sizeof(char));
	if((ptrfsource=fopen("FichierTexteTest.txt","r"))==NULL)
	{
		fprintf(stderr,"Erreur dans l'ouverture du fichier\n");
		exit(1);
	}
	if((ptrfdest=fopen("FichierTexteSortie.txt","w"))==NULL)
	{
		fprintf(stderr,"Erreur dans l'ouverture du fichier\n");
		exit(1);
	}
	while(fgets(lignelue,TailleLigneMax,ptrfsource) != NULL)
		fputs(lignelue,ptrfdest);
	if(!feof(ptrfsource))
		printf("Erreur lors de la lecture du fichier\n");
	fclose(ptrfsource);
	fclose(ptrfdest);
	FinMain();
}
```

XV.5. La manipulation d'un fichier binaire

Un fichier binaire est considéré comme une suite d'octets. Un fichier binaire peut être traité caractère par caractère, mais il n'est en général pas manipulable à l'aide d'un éditeur de texte. Un programme exécutable est un exemple de fichier binaire.

Un fichier binaire qui rassemble des données structurées (comme un fichier de clients ou un fichier de produits) est en général structuré en enregistrements de longueur fixe, chaque enregistrement décrivant un élément du fichier (un client, un produit,...). Un enregistrement est appelé un **article**.

L'accès à un fichier binaire, structuré en articles, en lecture et écriture peut se faire :

- séquentiellement article par article ou par paquets d'articles ;

- directement en accédant à un article donné.

Exemple

```
struct Personne {
        char nom[20];
        int age;
};
typedef struct Personne Article;
```

XV.5.1. L'accès séquentiel à un fichier binaire

a. Les fonctions fread et fwrite

Les deux principales fonctions de la bibliothèque stdio.h pour lire et écrire dans un fichier binaire structuré en articles sont :

size_t fread(void *Tampon, size_t Nb, size_t TailleArticle, FILE *PtrFichier)
La fonction fread permet de lire Nb articles de taille TailleArticle dans le fichier pointé par PtrFichier. Les articles lus sont stockés dans la variable Tampon passée en argument. Cette fonction retourne le nombre d'articles effectivement lus qui peut être inférieur à Nb si une erreur s'est produite ou si la fin du fichier a été atteinte.

size_t fwrite(const void *Tampon, size_t Nb, size_t TailleArticle, FILE *PtrFichier)
La fonction fwrite permet d'écrire Nb articles de taille TailleArticle de la variable Tampon passée en argument dans le fichier pointé par PtrFichier. Cette fonction retourne le nombre d'articles écrits qui peut être inférieur à Nb si une erreur s'est produite.

b. Un programme exemple

Le programme suivant permet de créer un fichier binaire de personnes et de lire ce fichier article par article.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define FinMain() return 0;
int main()
{
	FILE *ptrf=NULL;
	struct Personne {
		char nom[20];
		int age;
	};
	typedef struct Personne Article;
	Article art; int nb=0; char rep='\0';
	if((ptrf=fopen("FichierBinaireTest","w+b"))==NULL) {
		fprintf(stderr,"Erreur dans l'ouverture du fichier\n"); exit(1);
	}
	printf("Saisie des articles\n");
	do {
		printf("Saisie de la personne %d\n",nb++);
		printf("Donnez son nom : "); scanf("%s",art.nom);
		printf("Donnez son age : "); scanf("%d",&(art.age));
		fwrite(&art,1,sizeof(Article),ptrf);
		printf("Voulez-vous saisir une autre personne (O/N) ? ");
		rep=getche(); printf("\n");
	} while(rep=='O' || rep=='o');
	printf("Vous avez saisi %d ",nb);
	printf((nb>1)?"personnes\n":"personne\n");

	printf("\nLecture du fichier\n");
	fseek(ptrf,(long)0,SEEK_SET);	//pour aller au début du fichier
	nb=1;
	while(fread(&art,1,sizeof(Article),ptrf) != 0)
		printf("La personne %d s'appelle %s et a %d\n", nb++,
		art.nom,art.age);
	fclose(ptrf);
	FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Saisie des articles
Saisie de la personne 1
Donnez son nom : Dupond
Donnez son age : 35
Voulez-vous saisir une autre personne (O/N) ? o
Saisie de la personne 2
Donnez son nom : Durand
Donnez son age : 42
Voulez-vous saisir une autre personne (O/N) ? n
Vous avez saisi 2 personnes

Lecture du fichier
La personne 1 s'appelle Dupond et a 35 ans
La personne 2 s'appelle Durand et a 42 ans
```

XV.5.2. L'accès direct à un fichier binaire

Lorsqu'un fichier binaire est structuré en articles de même longueur, on peut accéder directement à un article dont on connaît le rang.

Ainsi pour accéder au n ème article d'un fichier, où la taille de chaque article est TailleArticle, il faudra se déplacer de (n-1) * TailleArticle octets à partir du début du fichier.

Exemple

Pour accéder à la 5ème personne du fichier FichierBinaireTest, il faut se déplacer à partir du début du fichier de 4*sizeof(Personne).

a. La fonction fseek

size_t fseek(FILE *PtrFichier,long nb,int Origine)

La fonction fseek déplace le pointeur de nb octets à partir de la position donnée par l'argument Origine, qui peut prendre une des valeurs suivantes :

- SEEK_SET représente le début du fichier ;
- SEEK_CUR représente la position courante ;
- SEEK_END représente la fin du fichier.

b. Un programme exemple

Le programme suivant permet de modifier une personne donnée par accès direct à un fichier binaire de personnes.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<io.h>
#define TestErreur(mes) {                    \
                fprintf(stderr,#mes "\n");       \
                exit(1); }
#define FinMain() return 0;

int main()
{
        FILE *ptrf=NULL;
        struct Personne {
                char nom[20];
                int age;
        };
        typedef struct Personne Article;
        Article art;
        int nbarticles=0,idfichier=0,taillefichier=0,num=0,nb=1,testrep=0;
        char rep='\0';
        if((ptrf=fopen("FichierBinaireTest","r+b"))==NULL)
                TestErreur(Erreur dans l'ouverture du fichier);
        idfichier=fileno(ptrf); taillefichier=filelength(idfichier);
        nbarticles=taillefichier/sizeof(Article);
        printf((nbarticles > 1)?"Il y a %d personnes dans le fichier : \n ":"Il y
        a %d personne dans le fichier : \n ", nbarticles);
        while(fread(&art,1,sizeof(Article),ptrf) != 0)
                printf("La personne %d s'appelle %s et a %d ans\n", nb++,
                art.nom,art.age);
        printf("\nDonnez le numero de la personne que vous voulez
        modifier : ");
        scanf("%d",&num);
        if(num<1 || num>nbarticles)
                TestErreur(Le numero donne est incorrect);
        fseek(ptrf,(long)(num-1)*sizeof(Article),SEEK_SET);
        if(fread(&art,1,sizeof(Article),ptrf) == 0)
                TestErreur(Erreur dans la lecture de la personne);
```

```
        printf("Voulez-vous changer le nom de %s ? (O/N) ",art.nom);
        rep=getche(); printf("\n");
        if(rep=='O' || rep=='o') {
                printf("Nouveau nom : ");
                scanf("%s",art.nom);
                testrep=1;
        }
        printf("Voulez-vous changer son age (%d ans) ? (O/N) ",art.age);
        rep=getche(); printf("\n");
        if(rep=='O' || rep=='o') {
                printf("Nouvel age : ");
                scanf("%d",&(art.age));
                testrep=1;
        }
        if(testrep) {
                fseek(ptrf,(long)(num-1)*sizeof(Article),SEEK_SET);
                fwrite(&art,1,sizeof(Article),ptrf);
                printf("La personne %d a bien ete modifiee !\n",num);
        }
        else
                printf("La personne %d n'a pas ete modifiee !\n",num);
        fclose(ptrf);
        FinMain();
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :

```
Il y a deux personnes dans le fichier :
La personne 1 s'appelle Dupond et a 35 ans
La personne 2 s'appelle Durand et a 42 ans

Donnez le numero de la personne que vous voulez modifier : 1
Voulez-vous changer le nom de Dupond ? (O/N) o
Nouveau nom : Dupont
Voulez-vous changer son age (35 ans) ? (O/N) n
La personne 1 a bien ete modifiee !
```

La fonction int fileno(FILE *PtrFichier) de la bibliothèque stdio.h retourne l'identifiant du fichier pointé par PtrFichier.

La fonction long filelength(int IdFichier) de la bibliothèque io.h retourne la taille en octets du fichier d'identifiant IdFichier.

TD12. Un programme pour manipuler des fichiers

Ecrivez un programme qui permet de créer un fichier binaire de personnes, une personne étant décrite par son nom et son prénom, à partir du fichier texte Donnees.txt suivant :

```
Pablo Picasso
Vasily Kandinsky
Amedeo Modigliani
Joan Miro
Rene Magritte
Henri Matisse
Gustave Klimt
Francis Bacon
```

Le programme sera notamment composé des trois fonctions suivantes :

- une fonction CreerFichier() qui permettra de créer le fichier binaire. Vous utiliserez la fonction fgets pour lire les personnes du fichier texte Donnees.txt, l'espace étant le caractère séparateur entre le prénom et le nom d'une personne ;

- une fonction LireFichier() qui permettra de lire le fichier binaire créé ;

XVI. Découvrir l'API Windows

XVI.1. L'API Windows : l'application minimale

Suivez les indications de l'enseignant pour créer et lancer votre première application Windows.

```
#include <windows.h>
/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);
/*  Make the class name into a global variable  */
char szClassName[ ] = "WindowsApp";

/* La fonction WinMain() est le point d'entrée du programme. Elle est exécutée
automatiquement par Windows lorsque le programme est démarré.*/
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszArgument, int nFunsterStil)
{
    HWND hwnd;               /* This is the handle(identifiant) for our window */
    MSG messages;            /* Here messages to the application are saved */

/* Création d'une nouvelle classe de fenêtre */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */
    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);
    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                /* No menu */
    wincl.cbClsExtra = 0;                     /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;                     /* structure or the window instance */
   /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;



/* Une fois la classe définie, un appel à RegisterClassEx() demande à Windows de
mémoriser la classe.*/
```

```
    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;
/* Création de la fenêtre grâce à la fonction CreateWindowEx(). */
    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           "Windows App",       /* Title Text */
           WS_OVERLAPPEDWINDOW, /* default window */
           CW_USEDEFAULT,       /* Windows decides the position */
           CW_USEDEFAULT,       /* where the window ends up on the screen */
           544,                 /* The programs width */
           375,                 /* and height in pixels */
           HWND_DESKTOP,        /* The window is a child-window to desktop */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );

/* La fonction CreateWindowEx() retourne une variable de type HWND qui identifie la
fenêtre. La fenêtre créée est ensuite affichée grâce à la fonction ShowWindow(). Toute
fenêtre créée est invisible par défaut. */
    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

/* La réception des messages est prise en charge par la fonction GetMessage(). Cette
fonction retourne FALSE dès que le message WM_QUIT est reçu, ce qui indique que
l'application doit être fermée. La procédure DispatchMessage() renvoie le message à la
procédure WindowProcedure(). */
    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
       /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }
    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    return messages.wParam;
}

/* La procédure WindowProcedure() décrit le comportement de la fenêtre face aux
messages reçus par la fonction DispatchMessage(). Les messages non traités sont passés
à DefWindowProc(). Le message traité ici est le message WM_DESTROY qui indique
que la fenêtre est en train d'être détruite (probablement suite au clic sur la 'croix'). La
fermeture de la fenêtre ne signifie pas forcément l'arrêt de l'application. Mais comme
dans ce cas l'application n'est formée que d'une fenêtre, l'application se termine et le
message WM_QUIT est envoyé. */
/*  This function is called by the Windows function DispatchMessage() */
```

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM
wParam, LPARAM lParam)
{
   switch (message)                  /* handle the messages */
   {
      case WM_DESTROY:
         PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
         break;
      default:                      /* for messages that we don't deal with */
         return DefWindowProc (hwnd, message, wParam, lParam);
   }
   return 0;
}
```

Exercice 1

Sachant que le message correspondant au clic gauche dans la fenêtre est identifié par la constante WM_LBUTTONDOWN, modifiez la fonction WindowProcedure pour que chaque clic gauche dans la fenêtre affiche un message d'information (MessageBox(hwnd,"Message affiché","Titre de la boîte",MB_OK)).

Référence API Windows :
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp

XVI.2. Les messages reçus par une fenêtre

La procédure WindowProcedure gère tous les messages reçus par la fenêtre.
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
HWND hwnd : identifiant de la fenêtre qui reçoit le message ;
UINT message : identifiant du message ;
WPARAM wParam : paramètre sur 16 bits, dépendant du message ;
LPARAM lParam : paramètre sur 32 bits, dépendant du message.

Comme les messages envoyés à une fenêtre sont nombreux, il serait très pénible de devoir tous les traiter, en particulier pour des fenêtres simples. L'API Windows fournit la fonction DefWindowProc() qui propose un traitement par défaut de tous les messages envoyés à une fenêtre. Pour imposer des comportements personnalisés à une fenêtre, il faut effectuer un traitement personnalisé de certains messages (par exemple, pour masquer la fenêtre lors d'une pression sur la 'croix'). Voici quelques messages :

- WM_CREATE est envoyé à une fenêtre au moment de sa création. Il est généré par l'appel de la fonction CreateWindow().
- WM_PAINT est envoyé lorsqu'une partie de la zone client doit être redessinée.
- WM_SIZE est envoyé lorsque la taille de la fenêtre à été modifiée. wParam : type de changement, lParam : nouvelle taille de la fenêtre (LOWORD(lParam) représente la largeur de la fenêtre et HIWORD(lParam) sa hauteur).
- WM_CLOSE indique que l'utilisateur demande la fermeture de l'application (en cliquant sur la 'croix' ou en pressant ALT+F4).
- WM_DESTROY est envoyé lorsque la fenêtre est détruite.

XVI.3. WM_PAINT : peindre et repeindre la zone-client

Le contenu de la zone client peut être, à chaque instant, altéré par les événements suivants :

- modification de la taille de la fenêtre,
- recouvrement total ou partiel de la fenêtre par une autre fenêtre.

En général, Windows ne reconstitue pas le contenu de la zone-client mais il demande à la fenêtre de « se repeindre », en lui envoyant le message WM_PAINT.

Pour afficher une information dans la zone-client, il faut créer l'information à afficher et générer un WM_PAINT pour la fenêtre.

Windows repeint la zone-client sans faire appel à l'application, dans les cas suivants :

- déplacement de la souris,
- déplacement d'une icône,
- déplacement de la fenêtre lorsqu'elle est au premier plan,
- fermeture d'une boite de dialogue bloquante.

Il n'est pas toujours utile de repeindre entièrement la fenêtre. Souvent, seule une partie de la zone client est altérée : on l'appelle le **rectangle invalide**. Il suffit alors de repeindre ce rectangle.

Windows envoie le message WM_PAINT à une fenêtre tant que le rectangle invalide n'a pas été revalidé.

XVI.4. L'environnement d'affichage (GDI) et les rectangles

Microsoft : *The graphical component of the Microsoft Windows graphical environment is the graphics device interface (GDI). It communicates between the application and the device driver, which performs the hardware-specific functions that generate output. By acting as a buffer between applications and output devices, GDI presents a device-independent view of the world for the application while interacting in a device-dependent format with the device.*

Le type composé RECT est une zone rectangulaire de l'écran définie par les coordonnées des coins supérieur gauche (left et top) et inférieur droit (right et bottom).

Windows associe en permanence deux rectangles particuliers à chaque fenêtre :
- le **rectangle actif** : région de l'écran dans laquelle les fonctions graphiques sont actives, c'est en général toute la zone-client.
- le **rectangle invalide** : partie de la zone client qui doit être repeinte.

Les coordonnées du rectangle invalide sont fixées par Windows lorsque l'utilisateur manipule des fenêtres.
Cependant, le programmeur qui désire afficher une information dans la zone-client peut modifier le rectangle invalide, en appelant la fonction InvalidateRect, qui modifie le rectangle invalide, puis génère un WM_PAINT :

```
BOOL InvalidateRect(HWND hWnd, LPRECT lpRect, BOOL bErase);
```

Si lpRect est NULL, toute la zone client devient invalide.

Si bErase est à TRUE, le fond de la fenêtre (qui n'est pas dans le rectangle invalide) est repaint, sinon non.

XVI.5. Le contexte d'affichage (*device context*)

Le **contexte d'affichage**, aussi appelé **device context**, est une structure de données interne à Windows qui permet d'accéder aux informations suivantes :

- la zone de l'écran où il est possible de dessiner : le rectangle actif,
- le système de coordonnées utilisé,
- la police de caractères utilisée,
- les outils graphiques disponibles (brosse, pinceau, ...).

Le contexte d'affichage est le lien entre l'application Windows et le périphérique de sortie. C'est une boîte noire qui permet d'être indépendant du périphérique de sortie.

Pour « obtenir une poignée » sur un device context, un premier moyen est d'utiliser les fonctions BeginPaint et EndPaint.

```
PAINTSTRUCT ps;
HDC hDC;
. . .
hDC = BeginPaint(hwnd, &ps);
. . .                          // ici on dessine dans la fenêtre
Endpaint(hwnd, &ps);           // libère hDC
```

ATTENTION, il ne faut pas conserver de poignées sur le device context entre le traitement de deux messages.

Pour connaître le rectangle invalide, on peut utiliser deux des champs de la structure PAINTSTRUCT :

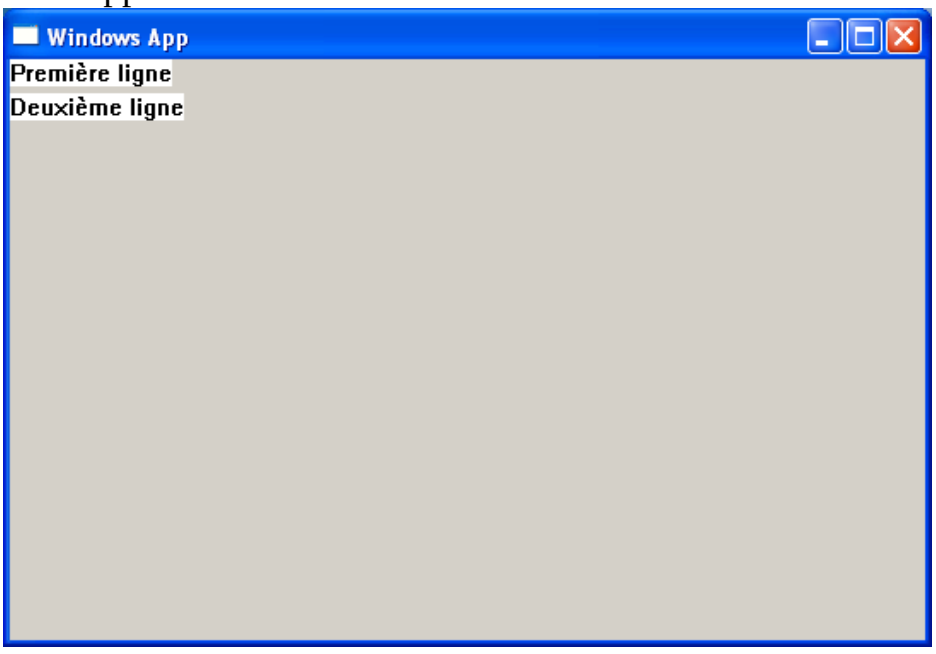
- rcPaint, le rectangle invalide,
- fErase, qui prend comme valeur zéro si le fond de la fenêtre n'a pas été repeint.

Le traitement du message WM_PAINT doit être effectué de préférence à l'aide des fonctions BeginPaint et Endpaint. Lorsque la poignée est obtenue avec BeginPaint, la zone active du device context est initialisée avec les valeurs du rectangle invalide. Endpaint valide automatiquement la région du rectangle invalide.

Exemple

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        switch (message)                /*handle the messages*/
        {
        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);
                TextOut(hdc, 0, 0, "Première ligne", strlen("Première ligne"));
                TextOut(hdc,0,20,"Deuxième ligne",strlen("Deuxième ligne"));
                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);  /*send a WM_QUIT to the message queue*/
                break;
        default:                        /*for messages that we don't deal with*/
                return DefWindowProc(hwnd, message, wParam, lParam);
        }
        return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :



La fonction TextOut permet d'afficher une chaîne de caractères :
BOOL TextOut(HDC hDC,int X,int Y,LPSTR lpStr,int NbCar)

Un deuxième moyen pour « obtenir une poignée » sur un device context est d'utiliser la fonction GetDC.

```
PAINTSTRUCT ps;
HDC hdc;
. . .
hdc = GetDC(hwnd);   /* ATTENTION, le rectangle invalide est la zone client
                        entière*/
. . .                        //ici on dessine dans la fenêtre
ReleaseDC(hwnd, hdc);        // libère hDC
```

La fonction suivante permet de connaître les caractéristiques de la police :
BOOL GetTextMetrics(HDC hdc, TEXTMETRIC *lpTextMetrics)
La structure TEXTMETRIC comporte plusieurs champs dont:

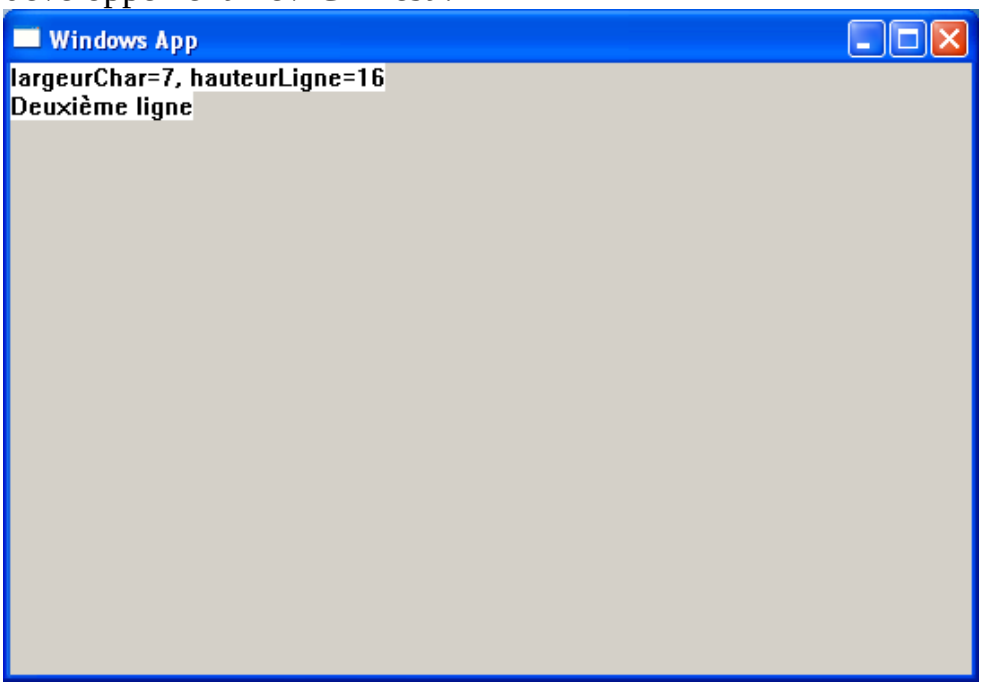
- tmAveCharWidth : la largeur des caractères,
- tmHeight : la hauteur des caractères,
- tmExternalLeading : l'interligne.

Exemple

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
        static short largeurChar, hauteurLigne;  // mesures de la police
        static char Information[80];
        HDC hdc;
        PAINTSTRUCT ps;
        TEXTMETRIC tm;
        switch (message) {                       /* handle the messages */
        case WM_CREATE:
                hdc = GetDC(hwnd);
                GetTextMetrics(hdc, &tm);
                largeurChar = tm.tmAveCharWidth;
                hauteurLigne = tm.tmHeight + tm.tmExternalLeading;
                sprintf(Information, "largeurChar=%d, hauteurLigne=%d",
                largeurChar, hauteurLigne);
                ReleaseDC(hwnd, hdc);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);
                TextOut(hdc, 0,0, Information, strlen(Information));
                TextOut(hdc, 0,hauteurLigne, "Deuxième ligne",
                strlen("Deuxième ligne"));
                EndPaint(hwnd,&ps); break;
```

```
        case WM_DESTROY:
                PostQuitMessage (0);  /*send a WM_QUIT to the message queue*/
                break;
        default:                      /* for messages that we don't deal with */
                return DefWindowProc (hwnd, message, wParam, lParam);
        }
        return 0;
}
```

Le résultat de l'exécution de ce programme dans l'environnement de développement Dev-C++ est :



Exercice 2

Modifiez le programme précédent pour qu'il affiche une troisième ligne qui compte le nombre de WM_PAINT traités. Vous pourrez utiliser le projet qui se trouve dans le répertoire Exercice_API_2.

XVI.6. Dessiner avec le pinceau

Le pinceau est un élément du contexte d'affichage caractérisé par une couleur, un style de ligne et une largeur en pixels.

La fonction GetStockObject permet d'obtenir une poignée sur un pinceau prédéfini (WHITE_PEN, BLACK_PEN) :
HGDIOBJ GetStockObject(WHITE_PEN)

La fonction CreatePen permet de créer son propre pinceau :
HPEN CreatePen(int penStyle, int largeur, COLORREF couleur)
Si penStyle vaut PS_DASHDOT, le pinceau est en pointillé (ligne continue : PS_SOLID).
Pour représenter une couleur, Windows utilise le mode RGB (Red, Green, Blue). Ainsi, le type COLORREF codé sur 32 bits représente une couleur par sa décomposition en rouge, vert et bleu (0 ... 255) :
La macro COLORREF RGB(rouge, vert, bleu) convertit les trois composantes d'une couleur en une couleur de type COLORREF.

La fonction SelectObject permet de changer le pinceau spécifié dans le device context :
HGDIOBJ SelectObject(HDC hDC, HANDLE hObjet)

Avant de quitter l'application, il faut libérer le pinceau avec la fonction suivante : BOOL DeleteObject (HGDIOBJ handle)

Quelques fonctions utiles

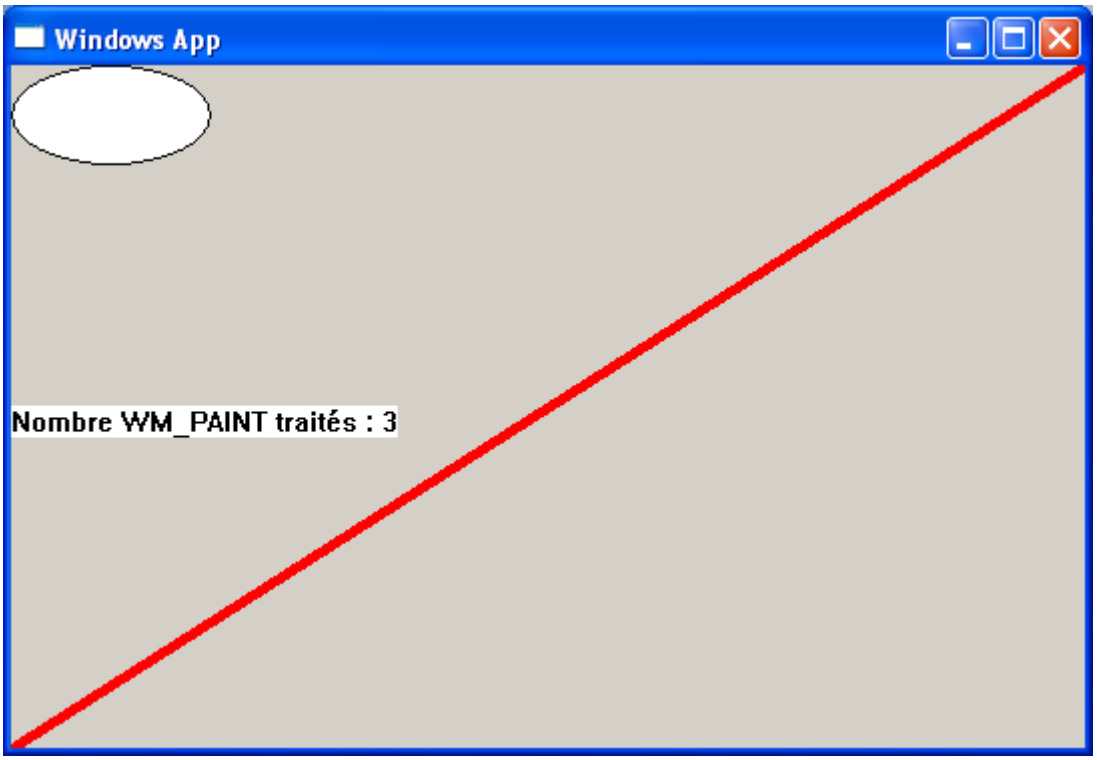
Dessin d'une ligne en positionnant le pinceau,
On va au point de départ :
MoveToEx (HDC hDC, int x, int y, NULL)
puis on trace :
LineTo (HDC hDC, int x, int y)

Affichage de figures
BOOL Rectangle (HDC hDC, int x1, int y1, int x2, int y2)

BOOL Ellipse (HDC hDC, int x1, int y1, int x2, int y2)

TD13. Un programme pour dessiner avec un pinceau

Ecrivez un programme qui permet d'obtenir l'affichage suivant :



L'ellipse a une taille et une position fixes.

La ligne de texte est toujours au milieu de la fenêtre.

La diagonale est rouge.

TD14. Une fenêtre avec un menu

Le menu est défini dans le fichier de ressources Mon_Menu.rc ajouté au projet :

```
Mon_Menu MENU
{  POPUP "Dessin"
   { MENUITEM "Forme", 11
     MENUITEM "Texte", 12
     MENUITEM "Ligne", 13
   }
   POPUP "Forme"
   { MENUITEM "Ellipse", 21
     MENUITEM "Rectangle", 22
   }
}
```

Le fichier Mon_Menu.h est un header pour le programme C :

```
#define MENU_Dessin_Forme 11
#define MENU_Dessin_Texte 12
#define MENU_Dessin_Ligne 13
#define MENU_Forme_Ellipse 21
#define MENU_Forme_Rectangle 22
```

Le menu est associé à la fenêtre dans la classe de fenêtre :

```
wincl.lpszMenuName = "Mon_Menu";  /* Maintenant, il y a un menu */
```

Dans la procédure de gestion de la fenêtre, on obtient une poignée sur le menu :

```
HMENU hmenu = GetMenu(hwnd);
```

Le clic sur une option de menu génère un message WM_COMMAND, dans lequel wParam identifie l'option sélectionnée.

La fonction suivante permet de cocher / décocher une option :
BOOL CheckMenuItem(HMENU hMenu, UINT idItem, UINT operation)
avec operation = MF_CHECKED ou MF_UNCHECKED

Complétez le fichier ci-dessous pour que toutes les options du menu fonctionnent.

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
        . . .
        static char Choix_Menu;       //option Forme, Texte ou Ligne du menu Dessin
        static char Ellipse_Rect;     //Ellipse ou Rectangle
        static HMENU hmenu;
        switch (message) {            /* handle the messages */
        case WM_CREATE:
                hmenu = GetMenu(hwnd);
                CheckMenuItem(hmenu, MENU_Forme_Ellipse, MF_CHECKED);
                Ellipse_Rect = 'E';
                break;
        case WM_SIZE: . . .
        case WM_PAINT:
                . . .
                if (Ellipse_Rect == 'E') Ellipse(hdc, 5, 5, 100, 50);
                else Rectangle(hdc, 5, 5, 100, 50);
                . . .
        case WM_COMMAND:
                switch(wParam) {              //A compléter
                case MENU_Forme_Ellipse:
                  CheckMenuItem(hmenu, MENU_Forme_Ellipse, MF_CHECKED);
                  CheckMenuItem(hmenu,MENU_Forme_Rectangle, MF_UNCHECKED);
                 Ellipse_Rect = 'E';
                  break;
                case MENU_Forme_Rectangle:
                  CheckMenuItem(hmenu, MENU_Forme_Ellipse, MF_UNCHECKED);
                  CheckMenuItem(hmenu, MENU_Forme_Rectangle,MF_CHECKED);
                  Ellipse_Rect = 'R';
                  break;
                default:
                  break;
                } // fin switch pour WM_COMMAND
                InvalidateRect(hwnd, NULL, TRUE);
                break;
        case WM_DESTROY:
                . . .
        }
        return 0;
}
```

XVII. Bibliographie

Patrice Buche, Gilles Clavel, Olivier Clavel, Isabelle Trillaud. *Programmer en C*. Support de cours pour la Spécialisation informatique, UER d'informatique, INA P-G, 2001.

Brian W. Kernighan et Dennis M. Ritchie. *Le langage C*. Masson (France), 1989.