Introduction au
Langage C

I. Généralités :

A l'origine inventé pour écrire UNIX (années 70).

- ➔ langage impératif (donne des ordres, différent de déclaratif).
- ➔ langage permitif (pas de typage fort par rapport au C++).
- ➔ langage compilé (différent des langage interprétés comme Basic).
  - Le code source (if, do, …)
  - Le code objet (exécutable) = traduction du langage source en langage machine.
  - La compilation permet d'effectuer cette traduction source/objet. (différent d'interprétation comme en Java).
- ➔ Typé et déclaratif (Variables).
- ➔ Structuré pour compléter l'exécution séquentielle (Alternatives, boucles, appels de fonctions).

II. Premier programme :

```
#include <stdio.h>

int main(void) {
	printf(“ Hello World ! “);
	return 0;
}
```

Librairies :

stdio.h : librairie standard entrée sortie.
math.h : cos, pow …

III.Type :

1. Entiers :

Type	Description	Taille Mémoire
Int	Entier standard signé	4 octets : -2^{31} à $2^{31}-1$
unsigned int	Entier standard positif	4 octets : 0 à 2^{32}
Short	Entier court signé	2 octets : -2^{15} à $2^{15}-1$
unsigned short	Entier court positif	2 octets : 0 à 2^{16}
Char	Caractère signé	1 octet : -2^{7} à $2^{7}-1$
unsigned char	Caractère positif	1 octet : 0 à 2^{8}

Rq : le type *char* est un cas particulier du type entier (Entier sur 8bits).

Attention : Les tailles données dans ce tableau correspondent à un environnement 32bits (largeur du bus du microprocesseur), elles sont susceptibles de changer et ne sont données ici qu'à titre d'exemple. Le seul moyen sûr de connaître la taille d'un type est d'utiliser ***sizeof*** qui renvoie un entier (%d) qui correspond au nombre d'octets qu'occupe le type passé en argument:

Ainsi sur certaines architectures on peut avoir
short = 2 octets, *int* = 2 octets, *long* = 4 octets
et sur d'autres
short = 2 octets, *int* = 4 octets, *long* = 4 octets

2. Réels :

En informatique, les rationnels sont souvent appelés des 'flottants'. Ce terme vient de *'en virgule flottante'* et trouve sa racine dans la notation traditionnelle des rationnels:

<+|-> <*mantisse*> * 10$^{<exposant>}$
<+|-> est le signe positif ou négatif du nombre
<*mantisse*> est un décimal positif avec un seul chiffre devant la virgule
<*exposant*> est un entier relatif

Exemples :
$3.14159*10^{0}$ $1.25003*10^{-12}$
$4.3001*10^{321}$ $-1.5*10^{3}$

En C, nous avons le choix entre trois types de rationnels: ***float***, ***double*** et ***long double***. Dans le tableau ci-dessous, vous trouverez leurs caractéristiques:

Type	Précision	Mantisse	Domaine min	Domaine max	Taille Mém.
Float	Simple	6	$3.4*10^{-38}$	$3.4*10^{38}$	4

Double	Double	15	$1.7*10^{-308}$	$1.7*10^{308}$	8
Long double	suppl.	19	$3.4*10^{-4932}$	$3.4*10^{4932}$	10

3. Initialisation :

Lors de la déclaration d'une variable, son contenue n'est pas initialisé et possède une valeur imprévisible. L'affectation d'une valeur à une variable s'effectue grâce à l'emploi de l'opérateur d'affectation « = ».

Attention : cette opérateur est différent de l'opérateur égalité « == ».

```
char c ;
c = 'A' ;  /* équivalent à char c = 'A' ;

int i ;
i = 50 ;   /* équivalent à int i = 50 ;
```

4. Déclaration de constantes :

➔ dans le main : const float PI = 3.14159 ;
➔ au dessus : #define PI 3.14159 ;

5. Conversions de type :

Variable de type 2 = (type 2)Variable de type 1

Cette conversion permet de passer d'un type à l'autre et parfois d'effectuer des opération, comme celle de supprimer les chiffre après la virgule d'un réel :

```
int res;
int i = 2 ;
float r=3.14159;
res = i*(int)r;                /* res=2*3=6 */
```

IV. Console :

Les instruction les plus importantes pour faire un programme qui marche en mode console :

1. printf :

a. affichage d'un texte :

printf(" BONJOUR ") ; / pas de retour à la ligne */*
printf(" BONJOUR \n") ; / affichage avec retour à la ligne du curseur */*

b. affichage d'une variable :

printf("%format ",nom_de_la_variable) ;

format : %c (char),%d (int), %4d (int sur 4 caractères), %e (float notation exponentielle), %le (double notation exponentielle), %f (float notation décimale), %lf …, %8f …, %5.4f …, %s (chaîne de caractères), … Cf Help %.

Exemple :

```
#include <stdio.h>

int main(void) {
	char c = 66 ;		/* c est le caractère alphanumérique A */
	printf("%d\n",c) ;	/* affichage du code ASCII en décimal */
	printf("%o\n",c) ;	/*                          en octal */
	return 0;
}
```

2. puts :

Exemple :

```
#include <stdio.h>

int main(void) {
	puts("BONJOUR ") ;
	return 0;
}
```

Rq: voir aussi *putchar(c)* ; (équivalent à *printf(« %c\n »,c)* ;)

3. scanf :

Exemple :

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

```
int main(void) {
	int x;
	printf("entrez un entier : " ) ;
	scanf(" %d",&x) ;
	printf("%d\n",x);/*affichage de l'entier tapé au clavier avant la touche"entrée"*/
	return 0;
}
```

Rq : Voir aussi getch(), getchar() … Cf Help. *scanf(« %c »,&c) ;* equivaut à *c=getchar() ;*.

4. Format de sortie :

a. Entiers :
%d : décimal
%x : hexadécimal
%u : décimal unsigned

char : %d : code ASCII en décimal
%c : caractère
%o : ASCII en base huit
%x : ASCII en hexadecimal

b. **Réels** :
%f : float
%lf : double

V. Opérateurs :

1. Arithmétiques :

+, -, *, /, %.

```
#include <stdio.h>

int main(void) {
	char c;
	char d;
	c='G' ;
	d=c+'a'-'A' ;		/*convertion de majuscule G en minuscule g*/
	return 0;
}
```

2. Opérateurs Logiques :

&ET, |OU, ^XOR, ~Complément à 1, <<Décalage à gauche, >> à droite, &&ET globale, ||OU globale, !NON …

3. Incrémentation, décrémentation :

```
i=i+1 ; /*est équivalent à i++ ; */
i=i-1 ; /*est équivalent à i-- ; */
```

Exemple :

```
int res ;
int i = 3 ;
res = 2 * i--;    /* res = 6 */
res = 2 * i;      /* res = 4 */
```

4. Opération combinés :

```
a = a+b; /*est équivalent à a+=b;*/
a = a-b; /*est équivalent à a-=b;*/
a = a&b; /*est équivalent à a&=b;*/
```

VI. Exercices :

1. Taille des variables (sizeof) :

Faire un programme (.c), qui affiche la taille des différents types du langage C :

a. Les entiers :

Exemple :

```
#include <stdio.h>          /*Librairie d'entrée/sortie*/

int main (void) {           /*Fonction principale*/

    printf("int = %d bytes\n",sizeof(int));
    printf("char = %d bytes\n",sizeof(char));
    …

    return 0;
}
```

affichage : int = 2 bytes
char = 1 byte

Effectuez cette affichage pour *les short int, long int, unsigned int, signed int* …et relevez les tailles obtenues.

Estimez l'intervalle de validité des différents types en fonction des résultats précédents (nb de bytes). Observé à l'aide d'un petit programme l'évolution des valeurs prises par les variables à la frontière des zones de validité.

Rq : Si le nombre d'octet ne change pas pour une variable *signed* ou *unsigned*, il n'en est évidemment pas de même pour l'intervalle de validité. Imaginez une situation qui justifie l'emploi d'un type *unsigned* et expliquer son intérêt.

Exemple :

```
#include <stdio.h>          /*Librairie d'entrée/sortie*/
#include <math.h>           /*Librairie pour pow*/

int main (void) {           /*Fonction principale*/
    int x;
    x=pow(2,8*sizeof(int)-1);     /*2^15 si int=2octets*/

    printf ("%d\n",x-1);
    printf ("%d\n",x);
    printf ("%d\n",x+1);
    printf ("%d\n",-x-1);
```

```
        printf ("%d\n",-x);
        printf ("%d\n",-x+1);

        return 0;
}
```

Vérifier les valeurs prisent : par exemple –32768<int< 32767.
=>$-1*2^{15}$<int<$2^{15}-1$.

Réalisez un tableau regroupant toutes ces valeurs.

b. Les réels :

De la même manière que pour les entiers, effectuer l'affichage du nombre d'octet (byte) des différent types de réels : float (%f), double (%lf), long double (%dl).

Exemple :

```
#include <stdio.h>      /*Librairie d'entrée/sortie*/

int main (void) {       /*Fonction principale*/

        printf("%d\n",sizeof(long double));

        return 0;
}
```

2. Conversions de types (cast) :

Ecrire un programme (.c) qui lira au clavier un nombre réel x et un nombre entier ndec et qui calculera la valeur du nombre réel avec ndec décimals. Par exemple si l'utilisateur entre : 1,2345 et 2 le résultat sera 1,23. Pour réaliser cette troncature, on utilisera la propriété de conversion de type explicite : *float x = 3.2, y; y=int(x) ;* y vaut 3.

3. Expressions booléennes :

a. Calcul :

On considère un quadrillage régulier de pas unité. Ecrire un programme qui lira au clavier les coordonnées entières de deux points sur ce quadrillage et qui calculera et affichera la distance entre ces deux points (utiliser sqrt() de math.h).

b. Localisation :

Les deux points précédents forment un rectangle. On ajoute maintenant au programme la lecture des coordonnées d'un troisième point. On cherche enfin à

déterminer si ce nouveau point se situe dans le rectangle. Cette information sera indiquée par une valeur Booléenne qui sera affiché sous forme numérique. Pour cela, on effectuera des divisions entre les différentes coordonnées des points dont le résultat sera soit un nombre supérieur 1 (Vrai), soit 0 (Faux) par conversion de type en entier (On ne prendra pas en compte le cas ou le point se trouve sur une frontière du rectangle).

La variable Booléenne sera définie avant le *main*, comme le montre le début de programme suivant :

```
#include <stdio.h>   /*Librairie d'entrée/sortie*/
#include <math.h>          /*Librairie pour pow et sqrt*/

typedef unsigned char BOOLEEN;

int main (void){     /*Fonction main*/

    int x1;/* declaration des variable */
    int y1;
    int x2;
    int y2;
    double dist;
    int xp;
    int yp;
    BOOLEEN xin;
    BOOLEEN yin;
    BOOLEEN in; …
```

4. Caractère et code ASCII :

Ecrire un programme qui lira un caractère au clavier (Utiliser *getche()*). A celui-ci sera ajouté un nombre entier (également lu au clavier) afin de passer à un autre caractère de la table ASCII. Afficher le caractère de départ et d'arrivée sous forme de caractère (%c) et de son code ASCII (%d).

VII. Structures de contrôle :

1. Les opérateurs logiques :

   == , !=, >, <, >=, <=.

   & (ET bit à bit), | (OU bit à bit), ~ (NON bit à bit), ^ (OU EXCLUSIF bit à bit).

   && (ET logique), || (OU logique), ! (NON).

2. Incrémentation, décrémentation :

   i=i+1 ; est équivalent à *i++ ;*
   i=i-1 ; est équivalent à *i-- ;*

3. Les structures conditionnelles :

   a. Opérateur condition :

   Variable = Condition ? Valeur si condition vrai : Valeur si condition fausse ;

   Exemple :

   VarBool = (choice == 0) ? true : false ;

   b. if :

```
if (condition réalisée)
{
liste d'instructions Vrai
}
else
{
autre série d'instructions Faux
}
```

   Rq : *else* est optionnel et *{}* ne sont indispensables que s'il y a plus d'une instruction dans chaque bloc.

   c. switch case :

```
switch (Variable)
{
case Valeur 1 : Liste d'instructions 1 ; break ;
case Valeur 2 : Liste d'instructions 2 ; break ;
…
```

```
default : Liste d'instructions Default ;
}
```

Rq : Les breaks ne sont pas obligatoires ; leur absence permet de laisser exécuter toutes les instructions des *case*s qui suivent.

4. Les Boucles :

a. while :

```
while (condition réalisée)
{
bloc d'instructions
}
```

b. for :

```
for (initialisation ; condition de continuité ; modification)
{
bloc d'instructions
}
```

c. do while :

```
do
{
bloc d'instructions
}
while (condition réalisée);
```

VII. Exemples :

1. Factoriel :

```
/*****************/
/* N!  Facto           */
/* for, while, do … */
/* DESS EESC       */
/* 14 Octobre 2003 */
/* PRELE Damien  */
/*****************/

/* Calcul de factoriel */

#include <stdio.h>   /*Librairie d'entrée/sortie*/

int main (void){     /*Fonction principale*/

    /* déclaration de variables */
    int N;
    int Fact;
    int temp;

    /* aquisition */
    printf ("entrez le nombre dont vous souhaité connaître le factoriel : ");
    fflush(stdin); /*initialise le buffer standard d'entrée*/
    scanf ("%d",&N);

    /* for */
    printf ("for\n");
    for (temp = N , Fact = 1 ; temp != 0 ; temp--)
            Fact *= temp;
    printf("%d! = %d\n\n",N,Fact);

    /* while */
    printf ("while\n");
    temp = N;
    Fact = 1;
    while (temp != 0)
            Fact *= temp--; /*Fact*=temp; temp--;*/
    printf("%d! = %d\n\n",N,Fact);

    /* do while */
    printf ("do while\n");
    temp = N;
    Fact = 1;
    if (N > 1)
            do Fact *= temp--; /*Fact*=temp; temp--;*/
                    while (temp != 0);
```

```
printf("%d! = %d\n\n",N,Fact);

/* switch */
printf ("switch\n");
temp = N;
Fact = 1;
switch(N) {
        case 7 : Fact *= temp--;
        case 6 : Fact *= temp--;
        case 5 : Fact *= temp--;
        case 4 : Fact *= temp--;
        case 3 : Fact *= temp--;
        case 2 : Fact *= 2; break;
        case 1 :
        case 0 : Fact = 1; break;
        default : printf("valeur non prévue");
}
printf("%d! = %d\n\n",N,Fact);

/* if */
printf ("if\n");
if (N < 2) Fact = 1;
else {
        if (N <= 7) {
                temp = N;
                Fact = 2;
                if (N > 2) {
                        Fact *= temp--;
                        if (N > 3) {
                                Fact *= temp--;
                                if (N > 4) {
                                        Fact *= temp--;
                                        if (N > 5) {
                                                Fact *= temp--;
                                                if (N > 6) Fact *= temp--;
                                        }
                                }
                        }
                }
        }
        else printf("valeur non prévue");
}
printf("%d! = %d\n\n",N,Fact);

/* ?: */
printf ("?:\n");
temp = N;
Fact = (N > 1) ? temp--:1;
Fact *= (N > 2) ? temp--:1;
Fact *= (N > 3) ? temp--:1;
```

```
    Fact *= (N > 3) ? temp--:1;
    Fact *= (N > 5) ? temp--:1;
    Fact *= (N > 6) ? temp:1;
    (N > 7) ? printf("valeur non prévue"):printf("%d! = %d\n\n",N,Fact);

    return 0;
}
```

VIII. Les Fonctions :

1. Déclaration (prototype) :

De la même manière qu'une variable, une fonction doit être au préalable déclarée. Pour cela on précise le **type de la valeur de retour**, le **nom de la fonction** et le **type des variables passées en paramètres**. Cette déclaration s'achève par un point virgule.

Exemple :

double Puissance (float, int) ;

2. Définition :

Type de Retour **Nom** (Type et nom des Paramètres)
{
    Code en C correspondant à la fonction
    r*eturn* Var de retour ;
}

Exemple :

```
double Puissance (float x, int y) {
        int temp;
        double z=1.;

        for (temp=y;temp>0;temp--)
                z*=(double)x;

        return z;
}
```

3. Appels :

L'utilisation de la fonction se fait par un **appel de fonction** dans le programme principale (main) ou dans une autre fonction.

Exemple :

res = Puissance(a,b);

4. paramètres :

Il peut y avoir plusieurs, un ou même zéro (void) paramètres à une fonction. Ces paramètres sont attribués aux variables de la fonction par une copie. La modification des variables de la fonction n'entraîne donc pas de modification des variables du programme qui appelle la fonction.

5. valeur de retour :

Il ne peut y avoir qu'une ou aucune valeur de retour. Celle-ci est retournée par l'instruction *return*.

Exemple :

```
return var ;
```

6. Exemple de fonction :

Fonction Puissance :

```
/**********************/
/* Fonction Puissance     */
/* DESS EESC              */
/* 14 Octobre 2003        */
/* PRELE Damien           */
/**********************/


#include <stdio.h>                          /*Librairie d'entrée/sortie*/

#define true 1                              /* définition des valeurs booléennes*/
#define false 0

float RecFloat (void);                      /*Prototypes des fonctions globale*/
int RecInt (void);
double Puissance (float, int);
void AffRes (float, int, double);
void Attente (void);

typedef unsigned char BOOLEEN;              /*Declaration du type BOOLEEN*/

BOOLEEN bool;                               /*variable global*/

int main (void) {                           /*Fonction Principale*/

        float a;                            /*Déclaration des variables*/
        int b;
        double res;


        do {
                a = RecFloat();
                b = RecInt();
                res = Puissance(a,b);
                AffRes(a,b,res);
```

```
                Attente ();
        }while (bool);

        return 0;
}

float RecFloat (void) {
        float f;
        printf("entrée f : ");
        fflush(stdin);
        scanf("%f",&f);
        return f;
}

int RecInt (void) {
        int i;
        printf("entrée i : ");
        fflush(stdin);
        scanf("%d",&i);
        return i;
}

double Puissance (float x, int y) {
        int temp;
        double z=1.;

        for (temp=y;temp>0;temp--)
                z*=(double)x;
        return z;
}

void AffRes (float x, int y, double z) {
        printf("%f à la puissance %d = %lf\n",x,y,z);
}

void Attente (void) {
        char caract;

        printf("Tapez \"0\" et retour pour sortir.\n");
        fflush(stdin);
        scanf("%c",&caract);
        if (caract=='0')
                bool = false;
        else bool = true;
}
```

IX. Exercices :

Décomposer au maximum vos programmes en sous-fonctions de manière à avoir un programme principal le plus évocateur possible.

Exemple :

```
int main (void) {
int x ;
int res ;

x = AcquisitionVar () ;
res = FonctionCalcule (x) ;
AffichageRes (res) ; …
```

1. Développement limité du sinus au voisinage de zéro :

Sin x = $x - x^3/3! + x^5/5!$ ... pour x petit.

a. factoriel :

Ecrire une fonction *factoriel* qui reçoit en argument un entier et retourne un entier. Prototype : *int factoriel (int) ;* . Vérifier son bon fonctionnement en écrivant un programme principal qui permet, à l'aide d'une boucle *do while* et d'une valeur booléenne, de tester plusieurs fois la fonction *fact* au cours d'une même exécution (Cf. exemple VIII. 5. Fonction Puissance).

0 !=1, 1 !=1 , 2 !=2 , 3 !=6 ; 4 != 24 ; 5 != 120, 6 !=720, 7 !=5040 … attention aux dépassements de capacités du type int.

b. dlsin :

Ecrire une fonction *dlsin* qui utilise la fonction *factorielle* précédente et reçoit en argument le réel x de sin(x), et un entier correspondant à l'ordre du développement. Prototype : *float dlsin (float, int) ;* . *dlsin* renvoie à la fonction principale un réel qui est le résultat de ce développement limité.

Utiliser une boucle *for* pour le calcul du DL : *for(i=1 ;i<=ordre ;i++) {*.

Vérifier le bon fonctionnement de votre programme puis écrire une autre version de la fonction *dlsin* à l'aide d'une boucle *while*.

2. Affichage d'une variable en binaire :

En C il est possible d'afficher un type entier sous forme décimale (*%d*), hexadécimale (*%x*) ou même octale (*%o*), mais ne permet pas d'affichage en binaire.

a. Fonction AffBin :

Ecrire une fonction *AffBin* qui permette d'afficher une variable de type entier sous forme binaire. Prototype : *void AffBin (int) ;*. Exemple :

Pour int a = 3 ; on a un affichage de type 0000000000000011.
Pour int a =-3 ; on a un affichage de type 1111111111111101.

b. Opération Logique :

Utiliser cette fonction pour afficher le résultat des différents opérateurs logiques : &, &&, ||, |, ^, entre deux entier a et b. De la même manière visualiser l'effet des opérateur ~, >>, << et – sur une variable de type entier.

X. Exemples de solutions des Exos VI. :

1. Taille des variables (sizeof) :

Cf premier poly.

2. Conversions de types :

```
/*******************/
/* TP 1 EXO 2          */
/* Typage et cast      */
/* DESS EESC           */
/* 9 Octobre 2003      */
/* PRELE Damien        */
/*******************/

#include <stdio.h>              /*Librairie d'I/O*/
#include <math.h>               /*Librairie pour pow*/

int main (void) {               /*Fonction Principale*/
float val;                      /*Déclaration des variables*/
int ndec;

printf("entrée le réel et le nombre de décimale : ");
scanf("%f %d",&val,&ndec);
val = (int)(val * pow(10,ndec))/pow(10,ndec);
printf("\n%f",val);

return 0;
}
```

3. Expressions booléennes :

```
/***************/
/* TP 1 EXO 3     */
/* Typage et cast  */
/* DESS EESC     */
/* 9 Octobre 2003 */
/* PRELE Damien*/
/***************/



#include <stdio.h>     /*Librairie d'entrée/sortie*/
#include <math.h>      /*Librairie pour pow et sqrt*/

typedef unsigned char BOOLEEN;

int main (void){        /*Fonction Principale*/
```

```
int x1;                    /* déclaration des variables */
int y1;
int x2;
int y2;
double dist;
int xp;
int yp;
BOOLEEN xin;
BOOLEEN yin;
BOOLEEN in;

printf("entrez les coordonnées du premier point :");
scanf("%d %d",&x1,&y1);
printf("entrez les coordonnées du deuxieme point :");
scanf("%d %d",&x2,&y2);
dist=sqrt(pow(x2-x1,2)+pow(y2-y1,2));
printf("la distance entre ces deux points est : %lf",dist);

printf("\nentrez les coordonnées d'un point :");
scanf("%d %d",&xp,&yp);

/* test x */
xin=!!(xp/x1)^!!(xp/x2);

/* test y */
yin=!!(yp/y1)^!!(yp/y2);

/* test in */
in = xin && yin;

printf("\n %d",in);

return 0;
}
```

4. Caractère et code ASCII :

```
/****************/
/* TP 1 EXO 4     */
/* Code ASCII     */
/* DESS EESC      */
/* 9 Octobre 2003 */
/* PRELE Damien*/
/***************/


#include <stdio.h>      /*Librairie d'entrée/sortie*/
```

```
int main (void){        /*Fonction Principale*/

char cara;              /* déclaration des variables */
int nb;

printf("entrez un caractère :");
gets(cara);


printf("\nle code ASCII de %c est %d",cara,cara);
printf("\nentrez un nb :");
scanf("%d",&nb);
printf("\n%d cases après on a : %c",nb,cara+nb);

return 0;
}
```

XI. Les Tableaux :

Un tableau en C est une grille de cellules où l'on range des données, les cellules étant repérées par leurs indices. Les données sont toutes du même type. Il est donc caractérisé par sa taille et son type.

1. Les tableaux à une dimension :

a. Déclaration :

type nom[dim] ;

Cette déclaration signifie que le compilateur réserve *dim* places (de la taille du type) en mémoire pour ranger les éléments du tableau.

Rq : *dim* est nécessairement une valeur numérique et ne peut en aucun cas être une variable du programme.

Exemples :

```
int compteur[10] ;
float nombre[20] ;
```

b. Utilisation :

Un élément du tableau est repéré par son indice. En langage C les tableaux commencent à l'indice 0. L'indice maximum est donc *dim-1*.

Appel :

```
nom[indice]
```

Exemple :

```
compteur[2]=5 ;
nombre[i]=6.789 ;
printf("%d",compteur[i]) ;
scanf("%f",&nombre[i]) ;
```

2. Les Tableaux à plusieurs dimensions :

a. Tableau à deux dimensions :

déclaration :

```
type nom[dim1][dim2] ;
```

Exemple :

```
int compteur[4][5] ;
```

float nombre[2][10] ;

b. Utilisation :

Un élément du tableau est repéré par ses indices. En langage C les tableaux commencent aux indices 0. Les indices maximum sont donc *dim1-1*, *dim2-1*.

Appel :

nom[indice1][indice2]

Exemple :

```
compteur[2][4] = 5 ;
nombre[i][j]=6.789 ;
printf("%d",compteur[i][j]) ;
scanf(" %f",&nombre[i][j]) ;
```

c. Tableau à plus de deux dimensions :

On procède de la même façon en ajoutant les éléments de dimensionnement ou les indices nécessaires.

3. Initialisation des tableaux :

a. Au moment de leur déclaration :

```
int liste[10]={1,2,4,8,16,32,64,128,256,512} ;
float nombre[4]={2.67,5.98,-8,0.09} ;
int x[2][3]={{1,5,7},{8,4,3}} ; /*2 lignes et 3 colonnes, équivalent à {1,5,7,8,4,3}*/
```

b. Dans une boucle :

```
int TabNul[100] ;
int i;

for(i=0;i<100;i++)
    TabNul[i]=0;
```

XII. Pointeurs :

1. L'opérateur adresse & :

L'opérateur adresse & retourne l'adresse d'une variable en mémoire :

Exemple :

```
int i = 8 ;
printf(« voici i : %d\n »,i) ;
printf(« voici son adresse en hexa. : %p\n »,&i) ;
```

2. Les pointeurs :

Un pointeur est une adresse mémoire. On dit que le pointeur pointe sur cette adresse.

Déclaration des pointeurs :

Une variable de type pointeur se déclare à l'aide de l'objet pointé précédé du symbole * (opérateur d'indirection).

Exemple :

```
char *pc      /* pc est un pointeur pointant sur un type char */
int *pi       /* pi est un pointeur pointant sur un type int */
float *pf     /* pf est un pointeur pointant sur un type float */
```

Rq :

L'opérateur * désigne en-fait le contenu de l'adresse.

Exemple :

```
char *pc ;
*pc = 34 ;
printf(« contenu de la case mémoire : %c\n »,*pc) ;
printf(« valeur de l'adresse en hexadécimal : %p\n », pc) ;
```

Arithmétique des pointeurs :

On ne peut déplacer un pointeur que d'un nombre de case mémoire multiple du nombre de case réservées pour le type de la variable sur laquelle il pointe.

Exemple :

```
int *pi ;        /* pi pointe sur un type entier */
float *pr ;      /* pr pointe sur un type réel */
```

```
char *pc ;          /* pc pointe sur un type caractère */

*pi = 421 ;         /* 421 est le contenu de la case mémoire p et des
3 suivantes */
*(pi+1)=53 ;        /* on range 53 4 cases mémoire plus loin */
*(pi+2)=0xabcd ;    /* on range 0xabcd 8 cases mémoire plus loin */
*pr = 45.7;  /* 45,7 est rangé dans la case mémoire pr et les 3
suivantes */
pr++; /* incrémente la valeur du pointeur pr (de 4 cases  mémoire) */
printf("L'ADRESSE pr VAUT : %p\n",pr);/* affiche la valeur de
l'adresse pr */

*pc = 'j' ; /* le contenu de la case mémoire c est le code ASCII de 'j' */
pc-- ;  /* décrémente la valeur du pointeur c (d'une case mémoire) */
```

3. Allocation dynamique :

Lorsque l'on déclare une variable char, int, float … un nombre de cases mémoire bien défini est réservé pour cette variable. Il n'en est pas de même avec les pointeurs.

Exemple :

```
char *pc ;
*pc = 'a' ;
/* le code ASCII de a est rangé dans la case mémoire pointée par c */
*(pc+1) = 'b' ;
/* le code ASCII de b est rangé une case mémoire plus loin */
*(pc+2) = 'c' ;
/* le code ASCII de c est rangé encore une case mémoire plus loin */
…
```

Dans cet exemple, le compilateur a attribué une valeur au pointeur c, les adresses suivantes sont donc bien définies ; mais le contenu des cases mémoires pc pc+1 pc+2 …sera perdu.

Il existe en langage C, des fonctions permettant d'allouer de la place en mémoire à un pointeur. Il faut absolument les utiliser dès que l'on travaille avec les pointeurs.

- *void *malloc (int taille)* : réserve une zone mémoire contiguë de taille octets, et retourne un pointeur non typé sur le début du bloc réservé. Retourne le pointeur NULL en cas d'erreur (pas assez de mémoire).
- *void *calloc (int nb, int taille)* : équivalent à *malloc(nb*taille)*.

Exemple : la fonction ***malloc*** :

```
char *pc ;
int *pi, *pj, *pk ;
```

```
float *pr ;
pc = (char*)malloc(10) ;
/* on réserve 10 cases mémoire, soit la place pour 10 caractères */
pi = (int*)malloc(16) ;
/* on réserve 16 cases mémoire, soit la place pour 4 entiers */
pr = (float*)malloc(24) ;
/* on réserve 24 places en mémoire, soit la place pour 6 réels */
pj = (int*)malloc(sizeof(int)) ;
/* on réserve la taille d'un entier en mémoire */
pk = (int*)malloc(3*sizeof(int));
/* on réserve la place en mémoire pour 3 entiers */
```

Exemple : la fonction ***calloc*** :

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

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

- *void free(void *pointeur)* libère la place réservée auparavant par *malloc* ou *calloc*.

Exemple : la fonction ***free*** (cf. : exemple ***malloc***) :

```
free(pi) ;      /* on libère la place précédemment réservée pour pi */
free(pr) ;      /* on libère la place précédemment réservée pour pr */
```

Voir aussi *realloc* pour réajuster la taille d'un bloc de mémoire. Ces fonctions sont définies dans *stdlib.h* ou *alloc.h* (suivant le compilateur).

4. Affectation d'une valeur à un pointeur :

On ne peut pas affecter directement une valeur à un pointeur : l'écriture : *char *pc ; pc = 0xfffe ;* est interdite. On peut cependant être amené à définir par programmation la valeur d'une adresse. On utilise pour cela l'opérateur de « cast » :

Exemples :

```
char *pc ;
pc = (char*)0x1000 ;
/* p est l'adresse 0x1000 et pointe sur un caractère */

int *pi ;
pi = (int*)0xfffa ;
```

```
/* i est l'adresse 0xfffa et pointe sur un entier */
```

5. Petit retour à la fonction scanf :

On a vu au premier cours que pour saisir une variable, on donne en fait son adresse.

Exemple :

```
char c ;
printf("taper une lettre : ") ;
scanf(" %c ",&c) ;
```

On saisit ici le contenu de l'adresse &c c'est à dire le caractère c lui-même.

On peut donc aussi procéder ainsi :

```
char *adr ;
printf("taper une lettre : ") ;
scanf(" %c ",adr) ;
```

On saisit ici le contenu de l'adresse adr.

XIII. Tableaux et Pointeurs :

En déclarant, par exemple, int *TAB[10] ;* l'identificateur *TAB* correspond en fait à l'adresse du début du tableau. Les deux écritures *TAB* et *&TAB[0]* sont équivalentes (ainsi que *TAB[0]* et **TAB*). On définit l'opération d'incrémentation pour les pointeurs par TAB+1 = adresse de l'élément suivant du tableau. L'arithmétique des pointeurs en C a cette particularité que l'opération dépend du type de variable pointée (*TAB+i=&TAB[i]*).

Déclarons : *int TAB[10], i, *ptr ;*

Ceci réserve en mémoire
- la place pour 10 entiers, l'adresse du début de cette zone est *TAB*,
- la place pour l'entier i,
- la place pour un pointeur d'entier (le type pointé est important pour définir l'addition).

Analysons les instructions suivantes :

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

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

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

Exemples :

```
#include <stdio.h>
void annule_tableau(int *t,int max) {
        for(;max>0;max--)*(t++)=0;
```

```
}

void affiche_tableau(int t[], int max) {
	int i;
	for(i=0;i<max;i++) printf("%d : %d\n",i,t[i]);
}

void main(void) {
	int tableau[10];
	annule_tableau(tableau,10);
	affiche_tableau(tableau,10);
}
```

XIV. Chaînes de caractères:

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

Exemples :

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

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

```
char *strptr="bonjour";
```

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

Bibliothèques de fonctions pour tableaux et chaînes :

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

int strlen(chaîne) donne la longueur de la chaîne (\0 non compris)

*char *strcpy(char *destination,char *source)* recopie la source dans la destination, rend un pointeur sur la destination

*char *strncpy(char *destination,char *source,int longmax)* idem strcpy mais s'arrête au \0 ou longmax (qui doit comprendre le \0)

*char *strcat(char *destination,char *source)* recopie la source à la suite de la destination, rend un pointeur sur la destination

*char *strncat(char *destination,char *source,int longmax)* idem

*int strcmp(char *str1,char*str2)* renvoie 0 si str1= =str2, <0 si str1<str2, >0 si str1>str2. Idem strncmp

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

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

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

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

de même atol et atof

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

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

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

XV. Exercices :

1. Echange :

Ecrire une fonction d'échange de valeurs de deux variables entières passées en argument. Cette fonction sera introduite dans un programme principal pour la tester. On prendra soin d'utiliser une boucle pour permettre à l'utilisateur de tester plusieurs exemples sans avoir à ré-exécuter le programme.

2. Moyenne :

a. Tableau :

Ecrire une fonction calculant la moyenne des éléments d'un tableau d'entiers. On initialisera les valeurs à l'aide de la fonction *random()* et on n'utilisera dans un premier temps que des opérations liées aux tableaux. Un effort sera fait pour décomposer le programme principal en un maximum de fonctions.

b. Pointeurs et allocations dynamiques :

Refaire l'exercice précédent en utilisant des pointeurs. On utilisera une allocation dynamique de mémoire qui permettra à l'utilisateur de spécifier le nombre d'éléments qu'il souhaite moyenner. Ces éléments ne seront plus initialisés aléatoirement mais entrés au clavier.

3. Palindrome :

Ecrire une fonction retournant 1 ou 0 (BOOLEEN) selon qu'elle détecte ou non la présence d'un palindrome dans la chaîne de caractères passée en argument.

4. Produit scalaire :

Ecrire une fonction qui calcule le produit scalaire de deux vecteurs.

XVI. Exemples de solutions des Exos IX. :

1. Développement limité du sinus au voisinage de zéro :

```
/*************************/
/* dlsin = x - x^3/3! + x^5/5! ... */
/*************************/

#include <stdio.h>

float dlsin (float,int);                    /* prototype fonction globale dlsin */

int main (void) {
        float sin;
        float x;
        int n;

        printf("saisit du x du sin(x) et de l'ordre n : ");
        fflush(stdin);
        scanf("%f %d",&x,&n);
        sin=dlsin(x,n);
        printf("le dl de sin(%f) à l'ordre %d est %f",x,n,sin);
        return 0;
}

float dlsin (float xx, int nn) {

        int fact (int);                             /* prototype fonction factorielle */

        float res=xx;
        float temp=xx;
        int i;
        int signe=-1;

        for(i=1;i<=nn;i++) {
                temp*=xx*xx;                        /* puissance */
                res=res+signe*temp/fact(2*i+1);     /* dl */
                signe*=-1;                          /* changement de signe */
        }
        return res;
}

int fact (int n) {
        if (n > 1)                                  /* cas général */
                return (fact (n-1) * n);
        else                                        /* fact = 1 si n = 1 ou 0*/
                return 1;
}
```

2. Affichage d'une variable en binaire :

```
/*********************/
/* Operateur Logique     */
/* AffBin                */
/* DESS EESC             */
/* 14 Octobre 2003       */
/* PRELE Damien          */
/*********************/

#include <stdio.h>              /*Librairie d'entrée/sortie*/
#include <math.h>

void AffBin (int);

int main (void) {               /*Fonction Principale*/
        int a;                  /*Declaration des variables*/
        int b;

        printf("entrée les réel a et b : ");
        fflush(stdin);
        scanf("%d %d",&a,&b);
        printf("a : ");
        AffBin (a);
        printf("b : ");
        AffBin (b);
        printf("a && b : ");
        AffBin (a&&b);
        printf("a & b : ");
        AffBin (a&b);
        printf("a || b : ");
        AffBin (a||b);
        printf("a | b : ");
        AffBin (a|b);
        printf("!!a ^ !!b : ");
        AffBin (!!a^!!b);
        printf("a ^ b : ");
        AffBin (a^b);
        printf("a>>1 : ");
        AffBin (a>>1);
        printf("b<<1 : ");
        AffBin (b<<1);
        printf("~a : ");
        AffBin (~a);                                    /*C1*/
        printf("~a+1 : ");
        AffBin (~a+1);                                  /*C2*/
        printf("-a : ");
        AffBin (-a);
        printf("pow(2,8*sizeof(int)-1) : ");
```

```
	AffBin (pow(2,8*sizeof(int)-1));

	return 0;
}

void AffBin (int b) {
	int i=pow(2,8*sizeof(int)-1);
	printf("%d",!!(b&i));				/*signe*/
	i=pow(2,8*sizeof(int)-2);
	do {
		printf("%d",!!(b&i));
		i>>=1; /*i\=2*/
	}while (i>0);
	printf("\n");
}
```

XVII. Les déclarations de types synonymes : ***typedef*** :

On a vu les types de variables utilisés par le langage C : int, char, float … . Le programmeur a la possibilité de créer ses propres types : il suffit de les déclarer en début de programme (après les instructions du préprocesseur : déclarations des bibliothèques et les « defines » : #) avec la syntaxe suivante :

Exemple :

Typedef int entier ; / on définit un type «entier» synonyme de «int» */*

Typedef unsigned char BOOLEEN ; / on définit un type "BOOLEEN" qui n'est rien d'autre qu'un entier non signé sur un octet. Il vaut FAUX pour 0 et VRAI pour toutes autres valeurs différentes de 0 */*

Typedef int vecteur[3] ; / on défini un type «vecteur» synonyme d'un tableau à 3 entiers */*

*Typedef float *fpointeur ; /* on définit un type «fpointeur » synonyme d'un pointeur sur un réel */*

La portée de la déclaration du type dépend de l'endroit où il est déclaré : dans le *main()* ou dans une autre fonction, le type n'est connu que de *main* ou de la fonction dans laquelle est déclaré ; en début de programme, le type est reconnu dans tout le programme.

Exemple :

```
#include <stdio.h>

#define VRAI 1
#define FAUX 0

typedef int entier;
typedef unsigned char BOOLEEN;
typedef float point[2];

int main() {
        entier n = 6;
        BOOLEEN B=VRAI ;
        point xy;
        xy[0] = 8.6 ;
        xy[1] = -9.45 ;
        …
        return 0 ;
}
```

XVIII. Les structures :

Le langage C autorise la déclaration de types particuliers : les structures. Une structure est constituée de plusieurs éléments qui peuvent être de types différents contrairement à un tableau.

1. Déclaration :

Exemple :

```
Typedef struct /* On définit un type struct */
{
        char nom[10] ;
        char prenom[10] ;
        int age ;
        float note ;
}StudE2SC ;
```

Rq : on peut faire des structures sans *typedef*, il suffit de réécrire *struct* .

2. Utilisation :

On déclare des variables :

```
StudE2SC S1, S2 ; /* possible grâce à typedef, sinon struct StudE2SC ... */
```

Puis on les initialisent :

```
strcpy(S1.nom, "DUPONT") ; //accès aux champ d'une structure
strcpy(S1.prenom, "TOTO") ;
S1.age = 24;
S1.note = 12.75;
```

L'affectation globale est possible avec les structures : on peut écrire :
S2 = S1;

XIX. Structures et Tableaux :

On peut définir un tableau de structures.

1. Déclaration :

Exemple :

StudE2SC S[26] ; / On déclare un tableau de 26 fiches StudE2SC*/*

2. Utilisation :

```
strcpy(S[i].nom, "DUPONT") ; /* pour un indice i quelconque */
strcpy(S[i].prenom, "TOTO") ;
S[i].age = 24;
S[i].note = 12.75;
```

XX. Structures et Pointeurs :

On peut déclarer des pointeurs sur des structures. Cette syntaxe est très utilisée en langage C, **elle est notamment nécessaire lorsque la structure est un paramètre modifiable dans la fonction.**

Un symbole spécial a été créé pour les pointeurs de structures, il s'agit du symbole ->

1. Déclaration :

```
StudE2SC *S ; /* on déclare un pointeur de fiche StudE2SC */
```

2. Utilisation :

```
S = (StudE2SC*)malloc(sizeof(StudE2SC)); /* réserve de la place */
strcpy(S->nom, "DUPONT") ;
strcpy(S->prenom, "TOTO") ;
S->age = 24;
S->note = 12.75;
```

XXI. Exercices :

1. Point de couleur RVB :

Définir la structure *PointCoul* représentant un point de couleur. Pour ceci, on créera deux autre sous structures *loc* pour deux entiers x et y, et *rvb* pour trois booleens r v b. Ainsi la structure PointCoul sera défini de la manière suivante :

```
typedef struct{
    struct loc pos;         //position du point
    struct rvb coul;        //couleur du point
}PointCoul;
```

Ecrire les fonctions permettant de saisir InitP et d'afficher en mode texte un point AffP, prenant respectivement un pointeur sur PointCoul et un PointCoul pour paramètre.

Définir un « tableau » qui stocke un ensemble de PointCoul (version statique et dynamique).

2. Graphique :

Reprendre l'exercice précédent et l'adapter à un environnement graphique où l'on pourra voir s'afficher à l'écran les points de couleur. Utilisation des librairies graphics.h de Borland.

XXII. Exemples de solutions des Exos XV. :

1. Echange :

```
/*************/
/* ECHANGE */
/* PRELE D.   */
/*************/

#include <stdio.h>

void Echange(int *, int *);

int main (void) {
     int var1=1, var2=2;

     printf("var1 = %d\nvar2 = %d\n",var1,var2);
     printf("echange\n");
     Echange(&var1, &var2);
     printf("var1 = %d\nvar2 = %d\n",var1,var2);
     return 0;
}

void Echange(int *pvar1, int *pvar2) {
     int temp;
     temp = *pvar1;
     *pvar1 = *pvar2;
     *pvar2 = temp;
}
```

2. Moyenne :

a. Tableau :

```
/************/
/* Moyenne */
/* Exo2 a     */
/* Tableau   */
/* PRELE D */
/************/

#include <stdio.h> /* I/O */
#include <math.h>  /* pow ... */
#include <stdlib.h> /* random ... */
#include <conio.h> /* putch */

#define TAILLE 1000
```

```
void InitTab (int []);
void AffTab (int []);
double MoyTab (int []);

int main (void) {
	int tableau[TAILLE];

	InitTab (tableau);
	AffTab (tableau);
	printf("La moyenne des elements du tableau est
%lf.",MoyTab(tableau));
	return 0;
}

void InitTab (int tab[]) {
	int i;
	int maxInt;
  randomize();

	maxInt = pow(2,8*sizeof(int)-2);
	for(i=0; i<TAILLE; i++)
		tab[i]=random(maxInt);
}

void AffTab (int tab[]) {
	int i;
	for (i=0; i<TAILLE; i++)
		printf("%d\t",tab[i]);  /* \t tabulation */
	putch('\n');
}

double MoyTab (int tab[]) {
	double m;
	int i;
	for (i=0, m=0; i<TAILLE; i++)
		m+=tab[i];
	return m/TAILLE;
}
```

b. Pointeurs et allocations dynamiques :

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

int* InitP (void);
void AffP(int *);
double MoyP(int *);

int main (void) {
```

```
	int *pointeur = NULL;

	pointeur = InitP();
	AffP(pointeur);
	printf("\nMoyenne : %lf",MoyP(pointeur));
	free(pointeur);
	return 0;
}

int* InitP (void) {
	int *p=NULL;
	int t;
	int i;

	printf("taille désirée ");
	fflush(stdin);
	scanf("%d",&t);

	p=(int *)calloc(t + 1, sizeof(int)); /* on ajoute taille */
	*p = t;

	for(i=1;i<(t+1);i++) {
		printf("\nentrez l'entier %d : ",i);
		fflush(stdin);
		scanf("%d",p+i);
	}

	return p; /* la taille est placée une case avant */
}

void AffP (int *pointe) {
	int i = 0;

	while(i<*pointe) {
		i++;
		printf("%d\t",*(pointe+i));
	}
}

double MoyP (int *pointe) {
	double m=0;
	int i;

	for(i=1;i<=*pointe;i++)
		m+=*(pointe+i);
	return m/(*pointe);
}
```

3. Palindrome :

```
/***********/
/* palindrom */
/***********/

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

typedef unsigned char BOOLEEN;
typedef char MOT[26];

BOOLEEN palin(char []);

int main(void) {
    MOT mot;
    printf("Entrez un mot : ");
    fflush(stdin);
    gets(mot);
    if(palin(mot))
        puts(" est un palindrom");
    else puts(" n'est pas un palindrom");
    return 0;
}

BOOLEEN palin(char mot[]) {
    MOT tom;
    int taille;
    int i;

    taille = 0;

    while(mot[taille]!='\0')
        taille++;
    for(i=0;i<taille;i++)
        tom[i]=mot[taille-1-i];
    tom[taille]='\0';

    return !strcmp(tom,mot);
}
```

La programmation orientée objet

Le C++

XXIII. : Introduction au C++ :

1. Historique :

Toutes les machines qui nous entourent fonctionnent grâce à des suites d'impulsions (des bits en base 2: "0" ou "1"). Les premiers programmeurs codaient directement leurs instructions en binaire. Afin de faciliter la communication entre le développeur et la machine, il fut nécessaire de développer une interface, un langage.

Dans un premier temps vinrent des langages **bas niveau** : les assembleurs. Ces langages sont spécifiques à chaque machine et s'adressent directement aux composants constituant celle-ci. C'est en fait une traduction directe du train binaire en instructions simples. (ex: charger en mémoire une valeur, lire une valeur, faire une addition,...)

Ensuite, dans un souci de simplicité, les langages devinrent de plus en plus proches de l'anglais écrit. Se succédèrent alors: BASIC, COBOL, FORTRAN. Notons que plus un langage est dit **évolué** plus il devient facile d'utilisation (nous parlons ici uniquement de la syntaxe).

En 1972, Dennis Ritchie créa le C ancêtre du C++. Ce langage peut être qualifié à la fois de bas niveau car il permet de faire appel à toutes les ressources de la machine mais aussi de langage évolué. En effet, il introduit une syntaxe assez complexe par rapport aux langages précités ainsi que de nombreuses fonctions évoluées. Ces fonctions sont regroupées en librairies.

La mise à jour la plus marquante du C fut apportée par Bjarde Stroustup en 1982. Il y intégra la programmation objet. Le C++ est en fait une surcouche du C (C++ signifie une incrémentation du C). Il hérite donc de tous les outils du C.

2. Principe de la programmation objet :

Trois générations de langage se sont succédés et coexistent encore aujourd'hui: les langages dits **linéaires**, **modulaires** puis **objets**.

Tous les premiers langages s'exécutaient de façon **linéaire**. Chaque ligne du programme était lue puis exécutée jusqu'à la dernière. Il était possible de faire des sauts ou des boucles mais le principe restait le même. Cette approche simpliste ne pouvait pas s'appliquer à des programmes complexes. De plus, les développeurs ne pouvaient pas réutiliser des outils déjà écrits. Le langage assembleur est un exemple de langage linéaire.

Afin, de réutiliser le code et d'éviter les redondances, les langages dits **modulaires** virent le jour. Le principe est de regrouper un ensemble d'instructions dans des fonctions ou procédures. En effet, chaque tâche exécutée par un programme représente un nombre variable d'instructions. Ces instructions sont réunies afin de pouvoir segmenter le code et de favoriser la réutilisation de celles-ci. Le C fait partie de ces langages.

Enfin, la méthode **objet** apparue. Elle est en fait une évolution de l'approche modulaire. Elle lui apporte principalement trois aspects primordiaux:

-> L'encapsulation : cette technique permet de réunir des variables et des fonctions au sein d'une même entité nommée classe. Les variables sont appelées les données membres, les fonctions sont appelées les méthodes. L'accès aux données et méthodes peut-être aussi réglementé.

-> L'héritage : cette technique permet de définir une hiérarchie de classe. Chaque classe fille hérite des méthodes et des données de ces "pères". En pratique, la classe de base est une classe générique, ainsi plus on descend dans la hiérarchie, plus on spécialise cette classe.

-> **Le polymorphisme :** les objets sont définis par leur classe. Deux objets, héritant une même méthode d'une classe parente, peuvent réagir de façon différente à l'appel de cette méthode.

XXIV. UN MEILLEUR C :

1. LES COMMENTAIRES

Le langage C++ offre une nouvelle façon d'ajouter des commentaires. En plus des symboles /* et */ utilisés en C, le langage C++ offre les symboles // qui permettent d'ignorer tout jusqu'à la fin de la ligne.

Exemple :

```
/* commentaire traditionnel
sur plusieurs lignes
valide en C et C++
*/
void main() { // commentaire de fin de ligne valide en C++
...
}
```

Il est préférable d'utiliser les symboles // pour la plupart des commentaires et de n'utiliser les commentaires C (/* */) que pour isoler des blocs importants d'instructions.

2. ENTREES/SORTIES AVEC *cin*, *cout* ET *cerr* :

Les entrées/sorties en langage C s'effectuent par les fonctions *scanf* et *printf* de la librairie standard du langage C. Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes, mais cependant les programmeurs C++ préfèrent les entrées/sorties par flux (ou flot ou stream). Trois flots sont prédéfinis lorsque vous avez inclus le fichier d'en-tête ***iostream.h*** :

cout	qui correspond à la sortie standard
cin	qui correspond à l'entrée standard
cerr	qui correspond à la sortie standard d'erreur.

L'opérateur (surchargé) << permet d'envoyer des valeurs dans un flot de sortie, tandis que >> permet d'extraire des valeurs d'un flot d'entrée.

Exemple :

```
#include <iostream.h>

void main() {
        int i=123;
        float f=1234.567;
        char ch[80]="Bonjour\n", rep;

        cout << "i=" << i << " f=" << f << " ch=" << ch;
        cout << "i = ? ";
        cin >> i;         // lecture d'un entier
        cout << "f = ? ";
```

```
        cin >> f;       // lecture d'un réel
        cout << "rep = ? ";
        cin >> rep;     // lecture d'un caractère
        cout << "ch = ? ";
        cin >> ch;      // lecture du premier mot d'une chaîne
        cout << "ch= " << ch;          // c'est bien le premier mot ...
}

/*-- résultat de l'exécution ----------
i=123 f=1234.57 ch=Bonjour
i = ? 12
f = ? 34.5
rep = ? y
ch = ? c++ is easy
ch= c++
-----------------------------------------------*/
```

Tout comme pour la fonction *scanf*, les espaces sont considérés comme des séparateurs entre les données par le flux *cin*. Notez l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

3. LES MANIPULATEURS :

Les manipulateurs sont des éléments qui modifient la façon dont est lu ou écrit le flot. Les principaux manipulateurs sont :

dec :	lecture/écriture d'un entier en décimal
oct :	lecture/écriture d'un entier en octal
hex :	lecture/écriture d'un entier en hexadécimal
endl :	insère un saut de ligne et vide les tampons
setw(int n) :	affichage de n caractères
setprecision(int n) :	affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur.
setfill(char) :	définit le caractère de remplissage
flush :	vide les tampons après écriture

Exemple :

```
#include <iostream.h>
#include <iomanip.h>

void main() {
        int i=1234;
        float p=12.3456;

        cout << "|" << setw(8) << setfill('*')
        << hex << i << "|\n" << "|"
        << setw(6) << setprecision(4)
        << p << "|" << endl;
}
```

```
/*-- résultat de l'exécution ----------
|*****4d2|
|*12.35|
-----------------------------------------------*/
```

4. LES CONVERSIONS EXPLICITES :

En C++, comme en langage C, il est possible de faire des conversions explicites de type, bien que le langage soit plus fortement typé :

```
double d;
int i;

i = (int) d;
```

Le C++ offre aussi une notation fonctionnelle pour faire une conversion explicite :

```
double d;
int i;

i = int(d);
```

Cette façon de faire ne marche que pour les types simples et les types utilisateurs. Pour les types pointeurs ou tableaux le problème peut être résolu en définissant un nouveau type :

```
double d;
int *i;
typedef int *ptr_int;

i = ptr_int(&d);
```

La conversion explicite de type est surtout utile lorsqu'on travaille avec des pointeurs du type *void * .*

5. DEFINITION DE VARIABLES :

En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code. La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant. Ceci permet de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité. C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.

Exemple :

```
#include <stdio.h>

void main() {
        int i=0;          // définition d'une variable
```

```
        i++;            // instruction

        int j=1;        // définition d'une autre variable
        j++;            // instruction

        int somme(int n1, int n2);      // déclaration d'une fonction
        printf("%d+%d=%d\n", i, j, somme(i, j));     // instruction
}
```

6. VARIABLE DE BOUCLE :

On peut déclarer une variable de boucle directement dans l'instruction *for*. Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.

Exemple :

```
#include <iostream.h>

void main() {
        for(int i=0; i<10; i++)
        cout << i << ' ';
        // i n'est pas utilisable à l'extérieur du bloc for
}

/*-- résultat de l'exécution --------
0 1 2 3 4 5 6 7 8 9
-------------------------------------*/
```

7. VISIBILITE DES VARIABLES :

L'opérateur de résolution de portée *::* permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>

int i = 11;
void main() {
        int i = 34;
        {
             int i = 23;
            ::i = ::i + 1;
            cout << ::i << " " << i << endl;
        }
cout << ::i << " " << i << endl;
}

/*-- résultat de l'exécution ------------------
12 23
```

```
12 34
-------------------------------------------------*/
```

L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms. En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres (Cf. p12).

8. LES CONSTANTES :

Les habitués du C utilisent la directive du préprocesseur *#define* pour définir des constantes. Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter. En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs (inclusion de fichiers ou de librairies).

Le mot réservé *const* permet de définir une constante. L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.

Exemple :

```
const int N = 10;        // N est un entier constant.
const int MOIS=12, AN=1995;        // 2 constantes entières
int tab[2 * N];          // autorisé en C++ (interdit en C)
```

9. CONSTANTES ET POINTEURS :

Il faut distinguer ce qui est pointé du pointeur lui-même.

La donnée pointée est constante :

```
const char *ptr1 = "QWERTY";
ptr1++; // autorisé
*ptr1 = 'A'; // ERROR: assignment to const type
```

Le pointeur est constant :

```
char * const ptr2 = "QWERTY";
ptr2++; // ERROR: increment of const type
*ptr2 = 'A'; // autorisé
```

Le pointeur et la donnée sont constants :

```
const char * const ptr3 = "QWERTY";
ptr3++; // ERROR: increment of const type
*ptr3 = 'A'; // ERROR: assignment to const type
```

10. LES TYPES COMPOSES :

En C++, comme en langage C, le programmeur peut définir des nouveaux types en définissant des structures (*struct*). Mais contrairement au langage C, l'utilisation de t*ypedef* n'est plus obligatoire pour renommer un type.

Exemple :

```
struct FICHE {          // définition du type FICHE
        char *nom, *prenom;
        int age;
};
// en C, il faut ajouter la ligne :
// typedef struct FICHE FICHE;

FICHE adherent, *liste;

enum BOOLEEN { FAUX, VRAI};
// en C, il faut ajouter la ligne :
// typedef enum BOOLEEN BOOLEEN;

BOOLEEN trouve;

trouve = FAUX;
trouve = 0; // ERREUR en C++ : vérification stricte des types
trouve = (BOOLEEN) 0;       // OK
```

11.VARIABLES REFERENCES :

En plus des variables normales et des pointeurs, le C++ offre les variables références. Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Dès lors, une modification de l'une affectera le contenu de l'autre.

```
int i;
int & ir = i;           // ir est une référence à i : i et ir ont la même adresse.
int *ptr;

i=1;
cout << "i= " << i << " ir= " << ir << endl;       // affichage de : i= 1 ir= 1

ir=2;
cout << "i= " << i << " ir= " << ir << endl;       // affichage de : i= 2 ir= 2

ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;       // affichage de : i= 3 ir= 3
```

Une variable référence doit être initialisée et le type de l'objet initial doit être le même que l'objet référence.

12. ALLOCATION MEMOIRE :

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour remplacer respectivement les fonctions *malloc* et *free* (bien qu'il soit toujours possible de les utiliser).

L'opérateur *new :*

L'opérateur *new* réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne soit l'adresse de début de la zone mémoire allouée, soit 0 si l'opération à échouée.

```
int *ptr1, *ptr2, *ptr3;

ptr1 = new int;                 // allocation dynamique d'un entier
ptr2 = new int [10];            // allocation d'un tableau de 10 entiers
ptr3 = new int(10);             // allocation d'un entier avec initialisation

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

ptr4 = new date;                // allocation dynamique d'une structure
ptr5 = new date[10];            // allocation dynamique d'un tableau de structure
ptr6 = new date(d);             // allocation dynamique d'une structure avec initialisation
```

L'opérateur *delete :*

L'opérateur *delete* libère l'espace mémoire alloué par *new* à un seul objet, tandis que l'opérateur *delete[]* libère l'espace mémoire alloué à un tableau d'objets.

```
delete ptr1;     // libération d'un entier
delete[] ptr2;   // libération d'un tableau d'entier
```

L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraine aucune conséquence facheuse (l'opération est tout simplement ignorée). A chaque instruction new doit correspondre une instruction *delete*. Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire. La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.

XXV. LES FONCTIONS

1. DECLARATION DES FONCTIONS :

Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction. Ces déclarations sont identiques aux prototypes de fonctions de la norme C-ANSI. Cette déclaration est par ailleurs obligatoire avant utilisation (contrairement à la norme C-ANSI). La déclaration suivante *int f1();* où *f1* est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration *int f1(void);* Alors que la norme C-ANSI considère que *f1* est une fonction qui peut recevoir un nombre quelconque d'arguments, eux-mêmes de type quelconques, comme si elle était déclarée *int f1(...);* .

2. PASSAGE PAR REFERENCE :

En plus du passage par valeur, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel. Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple :

```
void echange( int & n1, int & n2) {int temp = n1; n1 = n2; n2 = temp;}

void main() {
        int i=2, j=3;

        cout << "i= " << i << " j= " << j << endl;          // affichage de : i= 2 j= 3
        echange(i, j);
        cout << "i= " << i << " j= " << j << endl;          // affichage de : i= 3 j= 2
}
```

3. VALEUR PAR DEFAUT DES PARAMETRES :

Certains arguments d'une fonction peuvent prendre souvent la même valeur. Pour ne pas avoir à spécifier ces valeurs à chaque appel de la fonction, le C++ permet de déclarer des valeurs par défaut dans le prototype de la fonction.

Exemple :

```
void print(long valeur, int base = 10);

void main() {
        print(16); // affiche 16 (16 en base 10)
        print(16, 2); // affiche 10000 (16 en base 2)
}

void print(long valeur, int base){
        cout << ltostr(valeur, base) << endl;
```

```
}
```

Les paramètres par défaut sont obligatoirement les derniers de la liste. Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

4. SURCHARGE DE FONCTIONS :

Une fonction se définit par :

son nom,
sa liste typée de paramètres formels,
le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la **signature** de la fonction. On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents :

```
int somme( int n1, int n2) { return n1 + n2; }

int somme( int n1, int n2, int n3) { return n1 + n2 + n3; }

double somme( double n1, double n2) { return n1 + n2; }

void main() {
	cout << "1 + 2 = " << somme(1, 2) << endl;
	cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
	cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}
```

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction. Ce choix se faisant à la compilation, fait que l'appel d'une fonction surchargée procure des performances identiques à un appel de fonction "classique".

XXVI. LES CLASSES

1. DEFINITION D'UNE CLASSE :

Rappel :

La classe décrit le modèle structurel d'un objet :

- ➔ ensemble des **attributs** (ou champs ou données membres) décrivant sa structure
- ➔ ensemble des **opérations** (ou méthodes ou fonctions membres) qui lui sont applicables.

Une classe en C++ est une structure qui contient :

des fonctions membres
des données membres

Les mots réservés *public* et *private* délimitent les sections visibles par l'application.

Exemple :

```
class Avion {
        public :          // fonctions membres publiques
            void init(char [], char *, float);
            void affiche();
        private :         // membres privées
            char immatriculation[6], *type;   // données membres privées
            float poids;
            void erreur(char *message);       // fonction membre privée
}; // n'oubliez pas ce ; après l'accolade
```

2. DROITS D'ACCES :

L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe. Elle est réalisée à l'aide des mots clés :

-> ***private*** : les membres privés ne sont accessibles que par les fonctions membres de la classe.

-> ***protected*** : les membres protégés sont comme les membres privés. Mais ils sont aussi accessibles par les fonctions membres des classes dérivées (voir l'héritage page 18).

-> ***public*** : les membres publics sont accessibles par tous.

Les mots réservés *private, protected* et *public* peuvent figurer plusieurs fois dans la déclaration de la classe. Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

3. TYPES DE CLASSES :

struct Classe1 { /* ... */ };

Tous les membres sont par défaut d'accès public. Le contrôle d'accès est modifiable (on peut les rendre *private*). Cette structure est conservée pour pouvoir compiler des programmes écrits en C.

Exemple :

```
struct Date {
        // méthodes publiques (par défaut)
            void set_date(int, int, int);
            void next_date();
            // autres méthodes ....
        private :  // données privées
            int _jour, _mois, _an;
};
```

class Classe2 { /* ... */ };

Tous les membres sont d'accès *private* (par défaut). Le contrôle d'accès est modifiable. C'est cette forme qui est utilisée en programmation objet C++ pour définir des classes.

4. DEFINITION DES FONCTIONS MEMBRES :

En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe.

```
class Avion {
        public :
            void init(char [], char *, float);
            void affiche();
        private :
            char _immatriculation[6], *_type;
            float _poids;
            void _erreur(char *message);
};
```

Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source.

Syntaxe :

```
type_valeur_retournée Classe::nom_fonction( paramètres_formels ) {
        // corps de la fonction
}
```

Rq : « ***::*** » est appelé l'*opérateur de résolution de portée.*

Exemple de définition de méthode de la classe Avion :

```
void Avion::init(char m[], char *t, float p) {
        strcpy(_immatriculation, m);
        _type = new char [strlen(t)+1];
        strcpy(_type, t);
        _poids = p;
}

void Avion::affiche() {
        cout << _immatriculation << " " << _type;
        cout << " " << _poids << endl;
}
```

5. INSTANCIATION D'UNE CLASSE :

De façon similaire à une *struct*, le nom de la classe représente un nouveau type de donnée. On peut donc définir des variables de ce nouveau type ; on dit alors que vous créez des **objets** ou des **instances** de cette classe.

Exemple :

```
Avion av1;              // une instance simple (statique)
Avion *av2;             // un pointeur (non initialisé)
Avion compagnie[10];    // un tableau d'instances
av2 = new Avion;        // création (dynamique) d'une instance
```

6. UTILISATION DES OBJETS :

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux attributs et méthodes de la classe. Cet accès se fait comme pour les structures à l'aide de l'opérateur **.** (point) ou -> (tiret supérieur).

Exemple :

```
av1.init("FGBCD", "TB20", 1.47);
av2->init("FGDEF", "ATR 42", 80.0);
compagnie[0].init("FEFGH","A320", 150.0);
av1.affiche();
av2->affiche();
compagnie[0].affiche();
av1.poids = 0;              // erreur, poids est un membre privé
```

7. EXEMPLE COMPLET : PILE D'ENTIERS (1) :

```
// IntStack.C : pile d'entiers ------------------------------
#include <iostream.h>
#include <assert.h>
#include <stdlib.h> // rand()
```

```
class IntStack {
        public:
          void init(int taille = 10); // création d'une pile
          void push(int n); // empile un entier au sommet de la pile
          int pop(); // retourne l'entier au sommet de la pile
          int vide() const; // vrai, si la pile est vide
          //Rq : const pour être sur qu'il ne modifie pas les champs de la classe.
          int pleine() const; // vrai, si la pile est pleine
          int getsize() const { return _taille; }  //comme inline
        private:
          int _taille; // taille de la pile
          int _sommet; // position de l'entier à empiler
          int *_addr; // adresse de la pile
};
```

```
void IntStack::init(int taille ) {
          _addr = new int [ _taille = taille ];
          assert( _addr != 0 );
          _sommet = 0;
}

void IntStack::push(int n) {
          if ( ! pleine() )
          _addr[ _sommet++ ] = n;
}

int IntStack::pop()               { return ( ! vide() ) ? _addr[ --_sommet ] : 0;}

int IntStack::vide() const        { return ( _sommet == 0 );}

int IntStack::pleine() const      { return ( _sommet == _taille );}

void main() {
          IntStack pile1;
          pile1.init(15);                   // pile de 15 entiers
          while ( ! pile1.pleine() )        // remplissage de la pile
               pile1.push( rand() % 100 );
          while ( ! pile1.vide() )          // Affichage de la pile
               cout << pile1.pop() << " ";
          cout << endl;
}
```

8. CONSTRUCTEURS ET DESTRUCTEURS :

Les données membres d'une classe ne peuvent pas être initialisées; il faut donc prévoir une méthode d'initialisation de celles-ci (voir la méthode *init* de l'exemple précédent). Si l'on oublie d'appeler cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des surprises fâcheuses dans la suite de l'exécution. De même, après avoir fini

d'utiliser l'objet, il est bon de prévoir une méthode permettant de détruire l'objet (libération de la mémoire dynamique ...).

Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet, assurant ainsi une initialisation correcte de l'objet. Ce constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un *void*).

Exemple :

```
class Nombre {
        public :
            Nombre(); // constructeur par défaut
            // ...
        private :
            int _i;
};

Nombre::Nombre()    {_i = 0;}
```

On appelle **constructeur par défaut** un constructeur n'ayant pas de paramètre ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre {
        public :
            Nombre(int i=0); // constructeur par défaut
            // ...
        private :
            int _i;
};

Nombre::Nombre(int i)        {_i = i;}
```

Si le concepteur de la classe ne spécifie pas de constructeur, le compilateur génèrera un constructeur par défaut. Comme les autres fonctions, les constructeurs peuvent être surchargés.

```
class Nombre {
        public :
            Nombre(); // constructeur par défaut
            Nombre(int i); // constructeur à 1 paramètre
        private :
            int _i;
};
```

Le constructeur est appelé à l'instanciation de l'objet. Il n'est donc pas appelé quand on définit un pointeur sur un objet ...

Exemple :

```
Nombre n1;              // correct, appel du constructeur par défaut
Nombre n2(10);          // correct, appel du constructeur à 1 paramètre
Nombre *ptr1, *ptr2;    // correct, pas d'appel aux constructeurs

ptr1 = new Nombre;      // appel au constructeur par défaut
ptr2 = new Nombre(12);  // appel du constructeur à 1 paramètre

Nombre tab1[10];        /* chaque objet du tableau est initiallisé par un appel au
constructeur par défaut */
Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) }; /* initialisation des 3 objets du
tableau par les nombres 10 20 et 30 */
```

De la même façon que pour les constructeurs, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

Ce destructeur est une fonction :

qui porte comme nom, le nom de la classe précédé du caractère (tilda)
qui ne retourne pas de valeur (pas même un void)
qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

Exemple :

```
class Exemple {
    public :
        // ...
        ~Exemple();
    private :
        // ...
};

Exemple::~Exemple() {
    // par exemple delete ...
}
```

Comme pour le constructeur, le compilateur génèrera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.

9. PILE D'*int* AVEC CONSTRUCTEURS ET DESTRUCTEUR :

```
// IntStack.C : pile d'entiers ------------------------------
#include <iostream.h>
#include <assert.h>
#include <stdlib.h> // rand()

class IntStack {
    public:
        IntStack(int taille = 10);          // constructeur par défaut
        ~IntStack() { delete[] _addr; }     // destructeur
        void push(int n);                   // empile un entier au sommet de la pile
```

```
        int pop();                      // retourne l'entier au sommet de la pile
        int vide() const;               // vrai, si la pile est vide
        int pleine() const;             // vrai, si la pile est pleine
        int getsize() const { return _taille; }
    private:
        int _sommet;
        int _taille;
        int *_addr;                     // adresse de la pile
};

IntStack::IntStack(int taille) {        // remplace init
    _addr = new int [ _taille = taille ];
    assert( _addr != 0 );
    _sommet = 0;
}
// ...
// le reste des méthodes est sans changement
//

void main() {
    IntStack pile1(15);                 // pile de 15 entiers
    while ( ! pile1.pleine() )          // remplissage de la pile
        pile1.push( rand() % 100 );
    while ( ! pile1.vide() )            // Affichage de la pile
        cout << pile1.pop() << " ";
    cout << endl;
}
```

10. CONSTRUCTEUR COPIE :

Pour passer un objet en paramètre d'une fonction, il faut faire une copie de ce dernier ; d'où la nécessiter du **constructeur de copie**. Celui-ci est invoqué à la construction d'un objet à partir d'un objet existant de la même classe.

```
Nombre n1(10);          // appel du constructeur à 1 paramètre
Nombre n2(n1);          // appel du constructeur de copie
Nombre n3=n1;           // appel du constructeur de copie
```

Le constructeur de copie est appelé aussi pour le passage d'arguments par valeur et le retour de valeur.

```
Nombre traitement(Nombre n) {
    static Nombre nbre;
    // ...
    return nbre;        // appel du constructeur de copie
}

void main() {
    Nombre n1, n2;
    n2 = traitement( n1 );      // appel du constructeur de copie
```

```
}
```

Le compilateur C++ génère par défaut un constructeur de copie.

Constructeur de copie de la classe *IntStack* :

```
class IntStack {
        public:
            IntStack(int taille = 10); // constructeur par défaut
            IntStack(const IntStack & s); // constructeur de copie
            ~IntStack() { delete[] _addr; } // destructeur
            // ...
        private:
            int _taille; // taille de la pile
            int _sommet; // position de l'entier à empiler
            int *_addr; // adresse de la pile
};

// ...

IntStack::IntStack(const IntStack & s) { // constructeur de copie
        _addr = new int [ _taille = s._taille ];
        _sommet = s._sommet;
        for (int i=0; i< _sommet; i++) // recopie des éléments
        _addr[i] = s._addr[i];
}
```

11. CLASSES IMBRIQUEES :

Il est possible de créer une classe par une relation d'appartenance : relation **a un** ou **est composée de**.

Exemple : une voiture a un moteur, a des roues ...

```
class Moteur { /* ... */ };
class Roue { /* ... */ };

class Voiture {
        public:
            // ....
        private:
            Moteur _moteur;
            Roue _roue[4];
            // ....
};
```

XXVII. L'HERITAGE :

1. L'HERITAGE SIMPLE :

L'héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d'une classe déjà existante, **la classe de base** (ou super classe). La nouvelle classe (ou **classe dérivée** ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et ainsi peut réutiliser le code déjà écrit pour la classe de base. On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

2. MODE DE DERIVATION :

Si la classe B hérite de la classe A, la syntaxe est :

Pour un héritage public :

```
class B : public A {
        //Membres de la classe B
} ;
```

Protégé :

```
class B : protected A {
        //Membres de la classe B
} ;
```

Privé :

```
class B : private A {
        //Membres de la classe B
} ;
```

		Statut dans la classe de base	**Statut dans la classe dérivée**
Mode de dérivation	**public**	**public**	**public**
		protected	**protected**
		private	**private**
	protected	**public**	**protected**
		protected	
		private	**private**
	private	**public**	**private**
		protected	
		private	

3. REDEFINITION DE METHODES DANS LA CLASSE DERIVEE :

On peut redéfinir une fonction dans une classe dérivée si on lui donne le même nom que dans la classe de base. Il y aura ainsi, comme dans l'exemple ci-après, deux fonctions *f2()*, mais il sera possible de les différencier avec l'opérateur ***::*** de résolution de portée.

Exemple :

```
class X {
        public:
            void f1();
            void f2();
        protected:
            int xxx;
};

class Y : public X {
        public:
            void f2();
            void f3();
};

void Y::f3() {
        X::f2();            // f2 de la classe X
        X::xxx = 12;        // accès au membre xxx de la classe X
        f1();               // appel de f1 de la classe X
        f2();               // appel de f2 de la classe Y
}
```

XXVIII. Exercices :

Reprendre l'exercice de la dernière séance (Point de couleur RVB) en utilisant les notions de classe et méthode. On aura ainsi une première classe ***Point*** à partir de laquelle on créera une nouvelle classe ***PointCoul*** qui héritera de ***Point***.

Il sera demander d'employer au maximum les nouvelles notions propres aux C++ vues dans ce cours. Ainsi vous n'oublierez pas de définir un constructeur et un destructeur (si allocation dynamique) pour chacune de vos classes.

XXIX. Exemples de solutions succincts des Exos XXI. :

4. Point de couleur RVB :

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

struct loc{
	int x;
	int y;
};

struct rvb{
	char R;
	char V;
	char B;
};

typedef struct{
	struct loc pos;
	struct rvb coul;
}PointCoul;

void InitPoint(PointCoul *);
void AffPoint(PointCoul);

int main (void) {
	PointCoul p1;

	InitPoint(&p1);
	AffPoint(p1);
	getch();
	return 0;
}

void InitPoint(PointCoul *p1) {
	printf("entrez les coordonnées du point x y : ");
	scanf("%d %d",&((*p1).pos.x),&(p1->pos.y)); /*deux possibilité */
	printf("entrez la couleur r v b : ");
	scanf("%d %d %d",&(p1->coul.R),&(p1->coul.V),&(p1->coul.B));
}

void AffPoint(PointCoul p1) {
	int x = p1.pos.x;
	int y = p1.pos.y;

	char r = p1.coul.R;
	char v = p1.coul.V;
	char b = p1.coul.B;
```

```
        printf("\nUn point de couleur ");
        if(r==1) printf("rouge ");
        if(v==1) printf("vert ");
        if(b==1) printf("bleu ");
        printf("aux coordonnées (%d,%d)",x,y);
}
```

5. Graphique :

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

void INITGRAPH (void);

struct loc{
        int x;
        int y;
};

struct rvb{
        char R;
        char V;
        char B;
};

typedef struct{
        struct loc pos;
        struct rvb coul;
}PointCoul;

void InitPoint(PointCoul *);
void AffPoint(PointCoul);

int main (void) {
        PointCoul p1;

        InitPoint(&p1);
        INITGRAPH();
        AffPoint(p1);
        getch();
        closegraph();
        return 0;
}

void InitPoint(PointCoul *p1) {
```

```
        printf("entrez les coordonnées du point x y : ");
        scanf("%d %d",&((*p1).pos.x),&(p1->pos.y)); /*deux possibilité */
        printf("entrez la couleur r v b : ");
        scanf("%d %d %d",&(p1->coul.R),&(p1->coul.V),&(p1->coul.B));
}

void AffPoint(PointCoul p1) {
        char r = p1.coul.R;
        char v = p1.coul.V;
        char b = p1.coul.B;
        int couleur;
        int rvb = (r<<2)|(v<<1)|b;

        switch(rvb) {
                case 0 : couleur = 0; /* BLACK */
                case 1 : couleur = 1; /* BLUE */
                case 2 : couleur = 2; /* GREEN */
                case 3 : couleur = 3; /* CYAN */
                case 4 : couleur = 4; /* RED */
                case 5 : couleur = 5; /* MAGENTA */
                case 6 : couleur = 14; /* YELLOW */
                case 7 : couleur = 15; /* WHITE */
        }
        putpixel(p1.pos.x, p1.pos.y, couleur);
}

void INITGRAPH (void) {
        int gdriver = DETECT, gmode, errorcode;
        initgraph(&gdriver, &gmode, "c:\\TC\\BGI");
        errorcode = graphresult();
        if (errorcode != grOk)
        {
          printf("Graphics error : %s", grapherrormsg(errorcode));
          printf("Press any key to halt :");
          getch();
          exit(1);
        }
}
```

XXX. : Exemple de solutions de l'Exo XXVIII. :

Classes Point et PointCoul :

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

//typedef unsigned char bool;

class Point {
	public :
		Point(int posx = 0, int posy = 0);
		void InitPoint (void);
		void InitPoint (int,int);
		void AffPoint (void);
	private :
		int x;
		int y;
};

class PointCoul : private Point {
	public :

		PointCoul(bool r=0, bool v=0, bool b=0);
		void InitPointCoul (void);
		void InitPointCoul (int,int);
		void AffPointCoul (void);
	private :
		bool r;
		bool v;
		bool b;

};

class NuagePoint {
	public:
		NuagePoint(int i=10);
		void InitNuage(void);
		~NuagePoint();
		void AffNuage(void);
	private:
		int taille;
		PointCoul *PremierPoint;

};

Point::Point(int posx, int posy) {
	x = posx;
```

```
        y = posy;
}



void Point::InitPoint (void) {
        cout<<"entrez la coordonnee x : ";
        cin>>x;
        cout<<endl<<"entrez la coordonnee y : ";
        cin>>y;
        cout<<endl;
}

void Point::InitPoint(int posx,int posy) {
        x=posx;
        y=posy;
}

void Point::AffPoint (void) {
        cout<<"de coordonnee x : "<<x << endl<<"de coordonnee y : "<<y<<endl;
}

PointCoul::PointCoul(bool rouge, bool vert, bool bleu) {
        r = rouge;
        v = vert;
        b = bleu;
}

void PointCoul::InitPointCoul(void) {
        char carra;

        InitPoint();
        cout<<"rouge, o/n ?"<<endl;
        cin>>carra;
        r=(carra=='o');
        cout<<"vert, o/n ?"<<endl;
        cin>>carra;
        v=(carra=='o');
        cout<<"bleu, o/n ?"<<endl;
        cin>>carra;
        b=(carra=='o');
        cout<<endl;
}

void PointCoul::InitPointCoul(int posx, int posy) {
        InitPoint(posx,posy);
}

void PointCoul::AffPointCoul(void) {
        AffPoint();
```

```
        cout<<"de couleur ";
        if (r) cout << "rouge ";
        if (v) cout << "vert ";
        if (b) cout << "bleu ";
        cout<<endl;
}

NuagePoint::NuagePoint(int i) {
        taille =i;
        PremierPoint = new PointCoul[i];
}

void NuagePoint::InitNuage(void) {

        cout<<"Quel est le nbr de pts ds le nuage"<< endl;
        cin >> taille;
        PremierPoint = new PointCoul[taille];
        srand( (unsigned)time( NULL ) );
        for (int i=0;i<taille;i++)
                (PremierPoint+i)->InitPointCoul(rand()%100,rand()%100);

}

NuagePoint::~NuagePoint() {
        delete[] PremierPoint;
}

void NuagePoint::AffNuage(void) {
        for (int i=0;i<taille;i++)
                (*(PremierPoint+i)).AffPointCoul();
}


int main(void) {

        PointCoul p;
        PointCoul *t=new PointCoul;
        PointCoul tab[100];
        PointCoul *popo;
        NuagePoint truc;
        popo = new PointCoul[100];

        //for(int i=0;i<100;i++)
                //tab[i].AffPointCoul();
        //for(int i=0;i<100;i++)
                //(popo+i)->AffPointCoul();

        truc.InitNuage();
        truc.AffNuage();
```

```
        p.InitPointCoul();
        p.AffPointCoul();

        t->AffPointCoul();

        return 0;
}
```

Examen de Langage C et C++

10 Novembre 2003 PM
Salle 308
DESS Electronique Embarquée et Systèmes de Communication

Consignes :

Aucun document n'est autorisé, pas même la copie du voisin. Toutes tentatives de fraudes démasquées pendant ou après l'examen portera directement atteinte à la note du (des) candidat(s) concerné(s).

Un grand soin devra être apporté à la rédaction des réponses qui seront à la fois **courtes, claires et précises**. Toute phrase apportant effectivement une information pertinente en lien avec la question sera la bien venue.

Pour les quelques questions où il est demandé d'écrire un code, **il ne sera pas tenu compte d'éventuels oublis, dans la mesure où cela reste occasionnel et n'introduit pas d'ambiguïté**.

En règle générale, vous vous efforcerez de montrer que vous avez compris le fonctionnement et les mécanismes essentiels du langage C et C++ tels que nous les avons vus au cours des cinq dernières séances.

Les questions sont indépendantes les unes des autres. N'hésitez donc pas à *sauter* les questions qui vous posent problème.

1. Quelle valeur est affichée à la suite des instructions suivantes ? On se place dans le cas d'un environnement où les entiers sont codés sur 32bits.

a)

```
#include <stdio.h>

int main (void) {
    printf("%d",sizeof(int));
    return 0;
}
```

b)

```
#include <stdio.h>

int main (void) {
    printf("%d",sizeof(unsigned int));
    return 0;
}
```

c)

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

```
int main (void) {
    printf("%d",sizeof(char));
    return 0;
}
```

2. Un entier codé sur quatre octets peut prendre des valeurs allant de -2^{31} à $2^{31}-1$. D'où vient cette dissymétrie ?

3. Quel est l'affichage généré par les instructions suivantes ?

Rq : le code ASCII du 'a' est 97.

a)

```
#include <stdio.h>

int main (void) {
    int i ;
    char c ;

    i = 'a'-'A'; /* Rq : après cette affectation, i est positif */
    c = 'c' - i;
    printf("%c",c);
    return 0;
}
```

b)

```
#include <stdio.h>

int main (void) {
    char c ;

    c = 'c';
    printf("%c \n",c);
    printf("%d",c);
    return 0;
}
```

4. Qu'effectue cette suite d'instruction ?

```
#include <stdio.h>

int main (void) {

    float f;
```

```
    f=3.14159;
    printf("%f",((int)(f*100.))/100.);
    return 0;
}
```

5. Quel est l'affichage des instructions suivantes ?

Rq : une condition vraie renvoie 1 (0 si elle est fausse).

a)

```
#include <stdio.h>

int main (void) {

    int a;
    int b;

    a = 1;
    a+=1;
    printf("0 : %d \n",a);
    b = a;
    printf("1 : %d \n",b++);
    printf("2 : %d \n",(a >> 1));
    printf("3 : %d \n",(b >> 1));
    printf("4 : %d \n",(a == b));
    printf("5 : %d \n",(a != b));
    return 0;
}
```

b)

```
#include <stdio.h>

int main (void) {

    int a;
    int b;

    a = 2;
    b = a + 1;
    printf("6 : %d \n",(a && b));
    printf("7 : %d \n",(a & b));
    return 0;
}
```

c)

```c
#include <stdio.h>

int main (void) {

    int a;
    int b;

    a = 1;
    b = a;
    printf("8 : %d \n",(~a+1));
    printf("9 : %d \n",(a == -b));
    return 0;
}
```

d)

```c
#include <stdio.h>

int main (void) {

    int a;
    int b;

    a = 5;
    b = 2;

    printf("10 : %d \n",(5/2));
    printf("10 : %d",!!(5/2));
    return 0;
}
```

6. Qu'effectuent les instructions suivantes ? L'un des quatre programmes suivants entraîne une erreur à l'exécution. Lequel et pourquoi ?

a)

```c
#include <stdio.h>

int main (void) {
  int x;

    printf("x : ");
    scanf("%d",&x);
    printf("x = %d",x);
    return 0;
```

```
}
```

b)

```
#include <stdio.h>

int main (void) {
   int x;

     printf("x : ");
     scanf("%d",x);
     printf("x = %d",x);
     return 0;
}
```

c)

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

int main (void) {
     int *x;

     x=(int*)malloc(sizeof(int));
     printf("x : ");
     scanf("%d",x);
     printf("x = %d",*x);
     return 0;
}
```

d)

```
#include <stdio.h>

int main (void) {
     int x[1];

     printf("x : ");
     scanf("%d",x);
     printf("x = %d",x[0]);
     return 0;
}
```

7. Décrivez l'exécution des lignes de codes suivantes :

a)

```
#include <stdio.h>

int main (void) {
	int a;
	int b;
	int c;
	int d;
	a=1;
	b=2;
	c=(a==b)?3:4;
	d=(a!=b)?3:4;
	printf("%d %d %d %d",a,b,c,d);
  return 0;
}
```

b) L'affichage des lignes de codes suivantes est "a != 0". Pourquoi, à la vue du code, on peut finalement s'attendre à cet affichage ? Quelle petite modification (ajout d'un élément seulement) permet d'obtenir un comportement plus *conventionnel* ?

```
#include <stdio.h>

int main (void) {
	int a;

	a=0;

	if(a=0)
		printf("a = 0");
	else
		printf("a != 0");
	return 0;
}
```

c) Décrivez également le fonctionnement du programme lorsque l'on ne met pas le *break* à la fin du *case 1*.

```
#include <stdio.h>

int main (void) {
	int a;
	a=0;

	switch (a) {
		case 0 : printf("0");
		case 1 : printf("1");break;
		case 2 : printf("2");
		default : printf("toto");
```

```
        }
        return 0;
}
```

d) Donnez un nom à la fonction réalisée après avoir expliqué son fonctionnement :

```
        #include <stdio.h>

int main (void) {
        int a;
        int b;
        int c;

        b=1;
        printf("a : ");
        scanf("%d",&a);
        c=a;
        while(c>1)
                b*=c--;
        printf("b = %d",b);
        return 0;
}
```

e) Le code suivant génère une erreur à l'exécution. Que doit-on modifier pour que celui-ci fonctionne ? Pourquoi l'exécution est-elle problématique sans-cela ?

```
#include <stdio.h>
#define taille 3

int main (void) {
        int tab[taille];
        int i;

        for(i=1;i<=3;i++) {
                tab[i]=i;
                printf("%d \n",tab[i]);
        }
        return 0;
}
```

8. Ecrire une fonction qui échange la valeur de deux entiers à partir de leur adresse passée en paramètre :

Rq : on donne le programme et le prototype de la fonction qui permet l'exécution.

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

```
void Ech(int *, int *);

int main (void) {
        int v1=1;
        int v2=2;
        Ech(&v1, &v2);
        printf("v1 = %d et v2 = %d",v1,v2);
        return 0;
}
```

9. Soit la structure NOTE.

a) Il est demandé d'écrire dans le *main* la boucle qui permet d'initialiser toutes les variables *note* des 26 cases du tableau *notes*.

```
#include <stdio.h>

typedef struct{
        char nom[30];
        char prenom[30];
        float note;
}NOTE;

int main (void) {
        NOTE notes[26];

          /*  initialisation à 0 de la variable note des 26 cases du tableau */

        return 0;
}
```

b) Ecrire un programme similaire qui utilise un pointeur en lieu et place du tableau.

10. Langage C++ :

a) Citez au moins trois nouvelles fonctions offertes par le C++ (hors des fonctions propres à la programmation objet). Quel est leur intérêt par rapport à leurs équivalents en C ?

b) Quel est la différence entre une structure en C et une classe en C++ ? Quel intérêt présente l'encapsulation ?

Expliquez ce qu'est l'héritage en C++. Des exemples pourront illustrer vos explications ?