



Langage de Communication Informatique

Cours semestriel N° 4402

Cours de programmation de base en C

Arjan Egges
Stéphane Garchery

Version: Octobre 2004

Table des matières

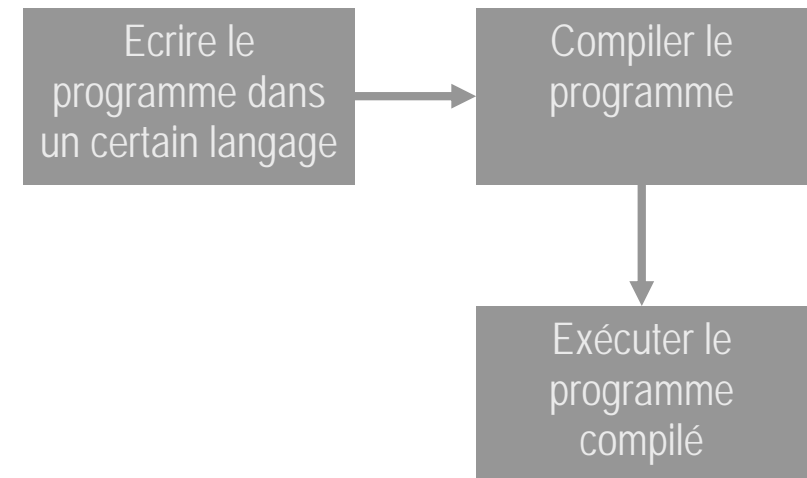
Cours semestriel N° 4402	1
Table des matières	2
Introduction au Langage C – Eléments de base.....	4
Qu'est-ce que la programmation ?.....	4
Exécution d'un programme en C	5
Eléments de base du C.....	7
La directive <i>#define</i>	9
La notion de variable	9
Affectation	11
Un type de variable spécial : <i>string</i>	12
Les expressions arithmétiques	13
Principales règles d'évaluation	15
Introduction au Langage C – Opérateurs, lecture de données ..	17
Les opérateurs logiques	17
Les opérateurs relationnels	19
Les types définis par l'utilisatrice	20
Les pointeurs.....	21
Introduction au Langage C – Les instructions de contrôle	25
Le bloc de contrôle	25
L'instruction <i>if else</i>	25
L'instruction <i>while</i>	26
L'instruction <i>for</i>	28
L'instruction <i>switch</i>	30
Introduction au Langage C – Fonctions et décomposition d'un programme.....	34
La décomposition d'un programme.....	34
Déclarations globales et locales	37
Introduction au Langage C – Paramètres de fonction/procédure	40
Passage de paramètres	40
Introduction au Langage C – Portée et persistance des variables	45

Portée d'une variable	45
Variables temporaires et statiques	47
Autres mots clefs	48
Fonctions et variables externes	49
Pointeurs comme arguments de fonctions	51
Introduction au Langage C – Les tableaux	53
Les tableaux unidimensionnels	53
Pointeurs et tableaux	55
Les tableaux multidimensionnels	57
Tableaux comme arguments de fonctions	58
Tableaux de fonctions	61
Manipulation de texte	63
Affectation de chaîne de caractères	64

Introduction au Langage C – Eléments de base

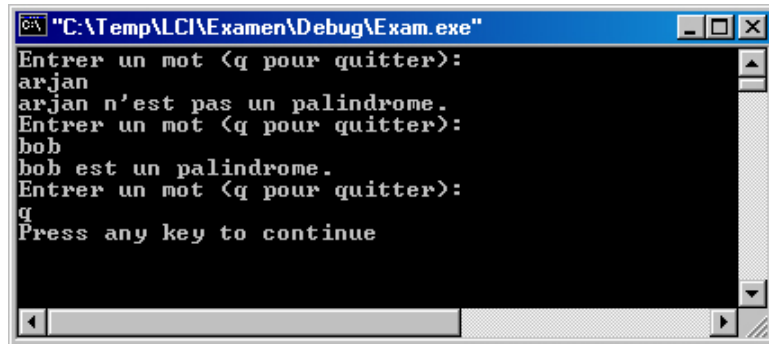
Qu'est-ce que la programmation ?

Un programme est une liste d'instructions pour un ordinateur. Un programme est écrit dans un langage (par exemple C ou Java). Les programmeurs comprennent ce langage, mais normalement l'ordinateur ne le comprend pas du tout. Pour cela, l'ordinateur doit transformer ce programme dans un langage qu'il comprend. Le processus de traduction entre ce que nous comprenons et ce que l'ordinateur comprend s'appelle **compilation**.



C est un langage **structuré** : les programmes sont organisés en blocs. C est aussi un langage **déclaratif** : normalement, tout objet doit être déclaré avant d'être utilisé. Les programmes écrits en C ont un **format libre** : la mise en page des divers

composants d'un programme est totalement libre. Finalement, C est un langage **modulaire** : une application pourra être découpée en modules qui pourront être compilés séparément. Le moyen d'interaction entre l'utilisateur et l'ordinateur se fait par la **console**.



Exécution d'un programme en C

Chaque programme écrit en C a une fonction *main()*. Cette fonction contient les instructions du programme (par exemple, une instruction pour écrire quelque chose sur la console). Parfois, il est nécessaire de mettre dans le programme des références aux autres bibliothèques pour que le programme puisse utiliser des fonctions définies dans la bibliothèque. Par exemple, l'instruction pour écrire des caractères sur la console, *cout*, est dans la bibliothèque *iostream*¹. Inclure une bibliothèque dans un programme se fait avec *#include*.

Il est aussi possible de mettre des commentaires dans un programme. L'ordinateur ne regarde pas les commentaires. Les commentaires sont **toujours nécessaires** dans un programme

¹ *iostream* et *cout* sont en fait des extensions de C++, en pur C il faut utiliser *stdio.h* et la fonction *fprintf(stdout, " Hello, world!\n")*.

pour clarification ! Il y a deux façons de mettre des commentaires dans un programme² :

```
// Voici des commentaires sur une ligne
```

ou

```
/* Voici des commentaires sur plusieurs lignes. On met
normalement des commentaires dans tous les programmes écrits en
C. */
```

Les programmes en C sont structurés par **blocs**. Un bloc est limité par les accolades { et }. Un bloc peut contenir des **instructions** ou d'autres blocs. Une instruction élémentaire est une expression terminée par le caractère ; . Un exemple d'une instruction simple qui écrit 'Hello, world !' sur la console :

```
cout << "Hello, world!" << endl;
```

Un exemple d'un bloc avec deux instructions :

```
{
    cout << "Hello, world!" << endl;
    cout << "Goodbye, world!" << endl;
}
```

Et on peut ajouter des commentaires aussi :

² // est aussi une extension de C++

```

/* Et voila un bloc qui écrit deux phrases sur la console. Il
est toujours préférable de mettre qu'une instruction par ligne.
*/
{
    cout << "Hello, world!" << endl; // première instruction
    cout << "Goodbye, world!" << endl;
    // fin du programme
}

```

Avec les blocs, les librairies et la fonction main(), on peut maintenant écrire un petit programme qui écrit 'Hello, world !' sur la console :

```

#include <iostream> // inclure la librairie Entrée/Sortie (i/o)
using namespace std; // nécessaire pour toutes les librairies STL

/* Programme qui affiche "Hello, world !" sur la console. */
int main()
{
    cout << "Hello, world !" << endl;
}

```

Eléments de base du C

a) Les nombres

Nombres entiers : suite de chiffres, peuvent être précédés d'un signe + ou d'un signe - :

Par exemple : 42 +327 -13678

Nombres réels : comportent un point, peuvent être précédés d'un signe + ou d'un signe - :

Par exemple : +3.14 6.28 -0.3456

Notation scientifique :

Par exemple : $6.23 \cdot 10^7$ s'écrit 6.23E+7
 $4.24 \cdot 10^{-15}$ s'écrit 4.24E-15

b) Les caractères

En C, les caractères sont toujours encadrés d'apostrophes.

Par exemple : 'a' 'T' '+' '1' 'f'

Le jeu de caractères utilisables normalement se met dans 3 catégories :

1. les lettres : 'A' - 'Z' et 'a' - 'z'
2. les chiffres : '0' - '9'
3. les caractères spéciaux : '+', '=', ';', '.', '%' etc.

Autres caractères spéciaux :

'\n' nouvelle ligne
 '\t' tabulateur horizontal
 '\\' backslash
 '\'' apostrophe
 '\"' guillemet

c) Les chaînes des caractères (constantes)

En C, une chaîne de caractère est encadrée de symboles "

Par exemple :

```
cout << "Il va à l'école" << endl;
```

Affiche : "Il va à l'école"

La directive `#define`

Il est possible en C de donner un nom à des valeurs en les déclarant avant leur utilisation.

Par exemple :

```
#include <iostream>
using namespace std;

#define PI 3.14159
#define RADIUS 12
#define TEXT "Contenu du cercle : "

int main()
{
    cout << TEXT << PI*RADIUS*RADIUS << endl;
}
```

La notion de variable

Sert à donner un nom à une valeur numérique de manière à la retrouver ultérieurement et la modifier si nécessaire. Le nom d'une variable doit nécessairement commencer par une lettre, les chiffres ne sont pas acceptés comme premier caractère du label.

Toutes les variables doivent être déclarées en C avant leur utilisation. Une telle déclaration s'effectue en faisant précéder les noms de variable par l'identificateur de leur type.

Il y a quatre identificateurs standards :

1. **int** pour les variables entières
2. **float** pour les variables réelles (ou flottantes) en simple précision
3. **double** pour les variables réelles (ou flottantes) en double précision, nécessaire pour des calculs très précis

4. **char** pour les variables de type caractère qui peuvent prendre comme valeur n'importe quel caractère du jeu disponible

Un exemple d'une déclaration d'une variable de type entier avec nom « x2 » et une variable de type char, nommé « c », par contre « 2x » n'est pas un nom de variable valide :

```
int x2;
char c;
int 2x;
```

On peut aussi initialiser plusieurs déclarations en même temps, par exemple : déclarer trois variables entières x, y et z

```
int x,y,z;
```

est équivalent à :

```
int x;
int y;
int z;
```

Encore quelques déclarations :

```
char variableOne, x, test2;
float abcABC;
double thisIsADoubleVariable, coursLCI;
```

De plus, préfixes short, long, signed et unsigned peuvent être placés devant un identificateur de type pour en modifier le sens.

int	16 bits	-32767 à 32768
unsigned int	16 bits	0 à 65535
short int	8 bits	-128 à 127
unsigned short int	8 bits	0 à 255
long int	32 bits	-2147483648 à 2147483649

unsigned long int	32 bits	0 à 4294967296
float	32 bits	env. 6 digits précision
double	64 bits	env. 12 digits précision
long double	128 bits	env. 24 digits précision

Par exemple :

```
short int a, b;
unsigned int somme;
long double c;
```

Notations abrégées suivantes :

short pour *short int*
long pour *long int*
unsigned pour *unsigned int*

Par exemple :

```
short a, b;
unsigned somme;
long double c;
```

Affectation

Pour donner une valeur à une variable, on utilise généralement une affectation du type

variable = expression ;

qui consiste à donner à une variable la valeur d'une expression.
 Par exemple :

```
int x;    // déclaration de la variable
x = 3;    // donner une valeur 3 à la variable x
```

La déclaration et l'affectation peuvent aussi être faites avec une seule expression :

```
// déclaration de la variable x et initialisation avec la valeur 3
int x = 3;
```

Quelques déclarations et affectations :

```
char variableOne = 'a';
float abcABC;
abcABC = -3.5675f;
double thisIsADoubleVariable = 5.602, coursLCI = 5.0;
```

Maintenant, on peut aussi afficher les valeurs des variables sur la console :

```
int i;
i = 45;
cout << "La valeur de la variable i est " << i << endl;
i = 2;
cout << "La valeur de la variable i est " << i << endl;
```

Ces instructions affichent à l'exécution :

La valeur de la variable i est 45
 La valeur de la variable i est 2

Un type de variable spécial : *string*

Une variable de type *string* est une chaîne de caractères. On peut déclarer des variables de type *string*, mais comme *string* ne

fait pas partie du langage standard C, il faut inclure une librairie (C++), *string*.

Déclarer une *string* se fait exactement comme avec les types de base (*int*, *char*, *float*, *double*) :

```
#include <string>
using namespace std;

int main() {

    // déclaration de la string s
    string s;
}
```

L'affectation d'une *string* avec une chaîne de caractères :

```
// affectation de la string s
s = "The quick brown fox jumps over the lazy dog";
```

On peut aussi accéder aux caractères dans la *string* :

```
// déclaration et affectation de la string s
string s = "Hello";
char c = s[0]; // s[0] donne le caractère 'H'
char d = s[4]; // s[4] donne le caractère 'o'
cout << s << endl; // afficher la string s sur la console
```

H	e	l	l	o
0	1	2	3	4

Les expressions arithmétiques

Combinaison de valeurs numériques (constantes, invocation de fonctions...), de signes opératoires et de parenthèses.

Pour les nombres entiers, 5 opérations sont disponibles : l'addition (+), la soustraction (-), la multiplication (*), la division (/) et le reste de la division (%).

Pour les nombres réels, 4 opérations sont disponibles : l'addition (+), la soustraction (-), la multiplication (*) et la division (/). Le reste de la division n'a pas de sens pour des nombres réels, il est donc impossible d'employer l'opérateur % entre 2 nombres réels.

```
int i=20, j=40, k=12; // déclaration et affectation des variables
i = j + k; // maintenant i a la valeur 52
j = j - 20; // j a la valeur 20
i = i*2 + j; // i a la valeur 124
int d = 40/k; // d a la valeur 3
int r = 40%k; // r a la valeur 4
```

Pour abréger l'écriture, on peut compresser l'expression $j = j + 4$ en $j += 4$.

Cette contraction de l'écriture peut se faire chaque fois que la variable à gauche de l'affectation se retrouve à droite du signe = et ceci est valable pour les opérateurs :

+ - * / %

Il faut noter que l'expression de droite est entièrement évaluée avant de faire l'opération avec la variable indiquée à gauche.

$x *= y + 2 \rightarrow x = x * (y + 2)$ et pas
 $x = x*y + 2$

Pour incrémenter et décrémenter les variables, C offre deux opérateurs :

++ additionne 1 à l'opérande et -- soustrait 1 à l'opérande. Ce sont donc des opérateurs unaires.

$j = j + 1 \rightarrow j++$

Ces opérateurs peuvent être utilisés comme préfixes ++j ou comme suffixes j-- .

Dans ++j on incrémente j **avant** d'utiliser sa valeur tandis que dans j++, on utilise la valeur de j et **ensuite** on incrémente j.

Si $j = 5$ $x = j++$ donne à x la valeur 5 et à j la valeur 6
 $x = ++j$ donne à x et à j la valeur 6

Un certain nombre d'opérations sont disponibles en utilisant les fonctions arithmétiques de la bibliothèque mathématique (math). Pour cela, il faut inclure la librairie math.h.

Par exemple :

```
#include <math.h>           // inclure la librairie mathématique

int main()
{
    double x = sqrt(100) ; // racine carrée du nombre 100
    double y = x*sin(0.4) + cos(-0.4) ; // des autres fonctions
    math
}
```

Principales règles d'évaluation

1. En l'absence de parenthèses, les multiplications et les divisions (*, /, %) sont traitées avant les additions et les soustractions.
2. En cas d'égalité de priorité, il faut savoir que l'évaluation se fait de gauche à droite.

3. S'il y a des parenthèses, elles sont effectuées en premier. Cette règle est aussi valable pour les arguments de fonctions qui sont évalués en premier.

Par exemple :

$3 + 4 * 7 / (2 - 5) \rightarrow 3 + 4 * 7 / (-3)$
 $\rightarrow 3 + 28 / -3$
 $\rightarrow 3 + -9$
 $\rightarrow -6$

Introduction au Langage C – Opérateurs, lecture de données

Les opérateurs logiques

Il n'y a pas de variables logiques en C. Cependant, toute valeur égale à 0 est interprétée logiquement comme FAUX et toute valeur différente de 0 est interprétée logiquement comme VRAI. Le résultat d'une opération logique vaut 0 ou 1.

Trois opérateurs logiques peuvent être définis pour toutes les variables : le "et" **&&**, le "ou" **||**, et le "non" **!**. En admettant la déclaration :

```
int p,q,q_et_q,p_ou_q,non_q;
[...] // initialisation
p_et_q = p && q;
p_ou_q = p || q;
non_q = !q;
```

p_et_q vaut 1 si et seulement si *p* et *q* sont différents de 0,
p_ou_q vaut 0 si et seulement si *p* et *q* sont égaux à 0,
non_p vaut 1 si *q* vaut 0 et inversement.

En C++, il existe un nouveau type pour les variables booléennes : *bool*, ainsi que deux constantes booléennes *true* et *false*. Nous allons les utiliser par la suite par soucis de simplification et de clarté. L'exemple précédent devient alors :

```
bool p,q,q_et_q,p_ou_q,non_q;
[...] // initialisation
p_et_q = p && q;
p_ou_q = p || q;
non_q = !q;
```

p_et_q vaut **true** si et seulement si *p* et *q* sont différents de **false**,

p_ou_q vaut **false** si et seulement si *p* et *q* sont égaux à **false**,
non_p vaut **true** si *q* vaut **false** et inversement.

Il est évidemment possible de construire des expressions booléennes contenant plusieurs opérateurs. Par exemple :

```
bool active, initialisée, finie, prête;
[...]
active = (prête && initialisée) || !finie;
```

Priorité des opérateurs :

- 1- En l'absence de parenthèses, les opérations sont effectuées dans l'ordre suivant : **!**, **&&**, **||**. Ainsi l'expression :

```
bool p=true, q=true, r, s=false;
r = !p || q && s;
```

sera évaluée de la sorte :

$r = !\text{true} \parallel \text{true} \&\& \text{false} \rightarrow \text{false} \parallel \text{false}$
 $\rightarrow \text{false}$

- 2- En cas d'égalité de priorité, les opérations sont évaluées de gauche à droite.
- 3- Les expressions entre parenthèses sont évaluées en premier. Ainsi l'expression :

```
bool p=false, q=false, r, s=true;
r = !((p || q) && s);
```

sera évaluée de la sorte :

$r = !((\text{false} \parallel \text{false}) \&\& \text{true}) \rightarrow !(\text{false} \&\& \text{true})$
 $\rightarrow \text{!false}$
 $\rightarrow \text{true}$

Les opérateurs relationnels

On peut comparer deux expressions simples à l'aide d'un des six opérateurs relationnels suivants :

opérateur en C	symbole usuel	signification
>	>	plus grand que
>=	≥	plus grand ou égal
<	<	plus petit que
<=	≤	plus petit ou égal
==	=	égal
!=	≠	différent

Ces opérateurs booléens ont tous une priorité inférieure aux opérateurs arithmétiques et les 4 premiers opérateurs (>, >=, <, et <=) ont une priorité supérieure à == et !=.

Les expressions booléennes simples doivent être de même type. Par exemple :

```
int quantité = 10;
bool Charbon, Taxable, célibataire=true, Voyelle;
float revenu = 24000.0f;
char lettre = 'G';

Charbon = quantité > 0;
Taxable = (revenu >= 12000.0f) && célibataire;
Voyelle = (lettre == 'A' || lettre == 'E' || lettre == 'I' ||
lettre == 'O' || lettre == 'U' || lettre == 'Y') ;
```

Le non-respect de la correspondance des types dans une expression entraîne généralement des conversions de type implicites par le compilateur (opération nommée « c type casting »). Le résultat pouvant alors s'avérer très différent de celui escompté...

```
int quantité=0;
bool Charbon, Taxable, célibataire=false, Voyelle;
float revenu=4000.0f;
char lettre = 'w';

Charbon = quantité > revenu; // c-type casting implicite
Taxable = (revenu >= lettre) && célibataire; // ... de même
```

Les types définis par l'utilisateur

Définition d'un nouveau type

Syntaxe :

```
typedef type_existant nouveau_type;
```

Chaque nouveau type a un identificateur de type. Par exemple :

```
typedef long int grand_entier_t;
grand_entier_t a,b;
```

Définition d'un type par énumération

Syntaxe :

```
enum { liste_de_définitions } nom_de_type;
```

Type par énumération : défini par une liste de valeurs qui sont nécessairement des identificateurs. Par défaut, le premier identificateur prend la valeur 0, et les identificateurs suivants sont incrémentés de 1 en 1.

Par exemple, un type pour représenter les jours de la semaine :

```
typedef enum {lundi, mardi, mercredi, jeudi, vendredi, samedi,
dimanche} jour_t;
jour_t aujourd'hui = mercredi;
```

Une variable de ce type (*jour_t*) peut prendre n'importe laquelle des 7 valeurs prédéfinies et seulement celles-ci. La définition ci-dessus est équivalente à :

```
typedef enum {lundi=0, mardi=1, mercredi=2, jeudi=3, vendredi=4,
samedi=5, dimanche=6} jour_t;
```

Pour obtenir des indexes de 1 à 7, il suffirait de spécifier explicitement la valeur du premier identificateur :

```
typedef enum {lundi=1, mardi, mercredi, jeudi, vendredi, samedi, dimanche} jour_t;
```

ATTENTION

- 1- Le type de la variable et celui de la valeur qu'on lui affecte doivent être les mêmes.
- 2- Les valeurs contenues dans la liste définissant un type scalaire ne sont pas des chaînes de caractères, mais représentent des entiers. Il est impossible de les lire ou de les afficher en tant que chaînes de caractères.
- 3- Un même label ne peut pas figurer dans deux types *enum* différents. Par exemple, la déclaration suivante :

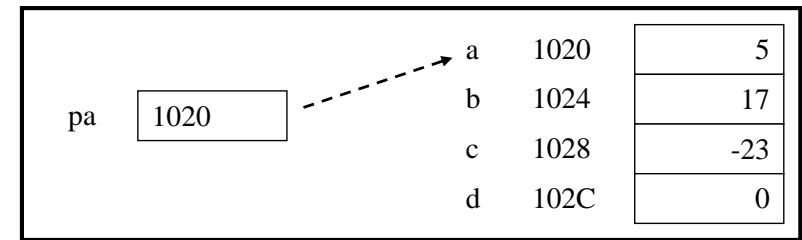
```
typedef enum {vert=60, jaune=70, orange=80, bleu=90} couleur_t;
typedef enum {cerise=60, citron=70, orange=80, fraise=50} fruit_t;
```

est impossible, car orange figure dans les deux types.

Les pointeurs

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. Il est nécessaire de se rappeler que dans un ordinateur, une variable a 2 attributs :

- la valeur qui lui est associée
- et l'endroit de la mémoire de l'ordinateur où elle est rangée (adresse de la variable).



Par convention, on note *a* la valeur de la variable *a* et *&a* son adresse. Si *pa* est un pointeur vers la variable *a*, alors *pa* désigne le contenu (la valeur) de la variable *a* (**pa* = 5 dans notre exemple).

```
int a = 5;
int *pa; // pointeur vers un int
pa = &a;
cout << "La valeur de la variable a vaut " << *pa << endl;
```

Pour déclarer un pointeur, on précède son nom d'un astérisque. Le type du pointeur désigne le type de variable dont il contient l'adresse ; on ne peut pas affecter à un pointeur de type *int* l'adresse d'une variable de type *float*.

```

int a = 5;
int *pa; // pointeur vers un int
float f = 1.23f;
float *pf; // pointeur vers un float

pa = &a; // pa pointe vers la variable a

*pa += 3; // identique à a += 3, donc *pa = a = 8

pa += 3; /* attention pa pointe désormais 3 variables int après
a donc on ne sait PAS du tout vers quoi !!! */

pf = &f; // OK

pa = pf; // ILLEGAL : 2 pointeurs vers des types différents

```

Constante NULL

Il est nécessaire de prévoir une valeur particulière pour un pointeur ne pointant nulle part (sur rien). On utilise pour cela la constante *NULL*.

```

float f = 1.23f;
float *pf = NULL; // pointeur vers un float initialisé à NULL

cout << *pf << endl; // Illégal : le pointeur ne pointe nulle
part
pf = &f; // OK
cout << *pf << endl; // OK, affiche 1.23 sur la console

```

Initialisation de pointeurs

On peut initialiser des pointeurs lors de leurs déclarations à condition de ne pas référencer une variable avant qu'elle n'ait été définie. Exemples :

```

float f = 1.23f;
float *pf = &f; // correct

int *pi = &qi; // illégal
int qi;
*pi = &qi; //correct

```

Introduction au Langage C – Les instructions de contrôle

Le bloc de contrôle

Il est délimité par des accolades `{ }`. Il permet notamment de définir la portée d'une déclaration de variables.

```
float f = 1.23f;
int bloc1 = 1;
{
    int bloc2;

    bloc2 = bloc1 * f; // OK
}
bloc2 = bloc1 * f; // illégal : bloc2 n'est plus une variable
// définie dans le bloc courant
```

L'instruction *if else*

Syntaxe :

```
If (expression booléenne)
    { /* bloc à exécuter 1 */ }
else if (expression booléenne)
    { /* bloc à exécuter 2 */ }
else { /* bloc à exécuter 3 */ }
```

Elle permet de tester une condition (expression booléenne) et d'effectuer sélectivement une ligne (ou un bloc) d'instruction(s) seulement si le résultat de la condition est vrai, et alternativement un autre bloc d'instructions suivant le mot clef **else** (c. à. d. *bloc 3*) lorsque la condition est évaluée à faux.

```
if(couleur == 3)
    radio = 1;

if(couleur == 3)
{
    radio = 1;
}
else if(couleur <= 2)
{
    radio = 2;
}
else
{
    cerr << "radio inconnue" << endl;
}
```

L'instruction *while*

Syntaxe :

```
while (expression booléenne)
{ /* bloc à exécuter */ }
ou
do { /* bloc à exécuter */ }
while (expression booléenne) ;
```

Elle permet d'effectuer un bloc d'instruction tant que la condition (expression booléenne) est vraie.

```
int couleur = 1 ;
while(couleur <= 3)
{
    ++couleur ;
}
// couleur vaut 4
cout << "couleur =" << couleur << endl ;
```

Avec l'instruction `do ... while`, afficher le premier terme de la suite de Fibonacci supérieur à 500. Rappelons que chaque terme de cette suite est la somme des 2 précédents ; les premiers valent 1.

```
/** programme de calcul du premier terme de la
    suite de Fibonacci > 500 */

#include <iostream>
using namespace std ;

int main()
{
    int N = 500;
    int somme = 1;
    int dernier = 1;
    int avantdernier;

    do
    {
        avantdernier = dernier;
        dernier=somme;
        somme = avantdernier + dernier;
    }
    while (somme <= N);

    cout << "Premier terme de la suite de Fibonacci supérieur à
"
        << N << " vaut " << somme << endl ;
}
```

Exécution :
Premier terme de la suite de Fibonacci supérieur à 500 vaut 610

L'instruction *for*

Syntaxe :

```
for(expression initiale;
    expression booléenne;
    expression d'itération)
{ /* bloc à exécuter */ }
```

Elle permet d'effectuer, après avoir exécuté l'expression d'initialisation, un bloc d'instruction et une expression d'itération tant que la condition (expression booléenne) est vraie. En ce sens, elle est équivalente à :

```
expression initiale;
while (expression booléenne)
{
    { //bloc d'instruction du for
    }
    expression d'itération;
}
```

```
typedef enum
{lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche} JOUR_t;

JOUR_t j, journee=mercredi;
for(j = vendredi ; j >= lundi ; --j)
{
    if(journee == j)
        cout << "c'est un jour ouvrable" << endl ;
}
```

```
// programme affichant la somme des carrés des 100 premiers entiers
```

```
#include <iostream>
using namespace std ;
```

```
int main()
{
    int N;
    long int somme;
    int entier;

    for(entier = 1, somme = 0, N = 100; //initialisations
        entier <= N; //condition
        somme += (entier*entier), ++entier) //itérations
    {
        if(entier % 25 == 0)
            cout << "Somme intermédiaire à l'itération " << entier
                << " = " << somme << endl;
    }
    cout << "La somme finale = " << somme << endl;
}
```

Exécution :

```
Somme intermédiaire à l'itération 25 = 4900
Somme intermédiaire à l'itération 50 = 40425
Somme intermédiaire à l'itération 75 = 137825
Somme intermédiaire à l'itération 100 = 328350
La somme finale = 338350
```

L'instruction *switch*

Syntaxe :

```
switch (variable_entière)
{
    case valeur1 :
        //liste d'instructions
        break ;
    case valeur2 :
        //liste d'instructions
        break ;
    default :
        //liste d'instructions
        break ;
}
```

Elle permet de réaliser un choix multiple, avec des actions associées aux différentes valeurs de la variable entière (*int*, *long int*, etc...) considérée. L'instruction **break** peut être omise (ou remplacée par **return()**). Dans le premier cas, l'absence de **break** implique que la ou les action(s) associée(s) aux **case** suivants seront aussi exécutées.

```
float a=0.0f;
int n=1;
switch(n)
{
    case 1:
        a += 1.0f;
    case 2:
        a += 2.0f/3.0f;
    case 3:
        a /= 5.0f;
        break;
    default : // facultatif mais figure nécessairement
               // après tous les 'case'
        cout << "cas non répertorié" << endl ;
        return false ;
}
```

Cet exemple est équivalent (du point de vue logique) à :

```
if (n == 1) { a += 1.0f ; a += 2.0f/3.0f; a /= 5.0f;}
else if (n == 2) { a += 2.0f/3.0f; a /= 5.0f;}
else if (n == 3) { a /= 5.0f;}
else {
    cout << "cas non répertorié" << endl ;
    return false;
}
```

Il faut donc prendre garde que les cas ne sont pas mutuellement exclusifs. Ainsi, si $n=1$, on obtient :

$$a = (a + 1 + 2/3) / 5$$

Ce qui n'est probablement pas ce que l'on désirait.

```
// programme effectuant la somme des achats d'un client

#include <iomanip> // pour setprecision() et setfixed
#include <iostream>
using namespace std ;

int main()
{
    float somme = 0, prix;
    char catégorie;

    cout << "Exécution: " << endl;
    cout << "Tapez la catégorie, puis le prix (f pour terminer)" << endl;
    cin >> catégorie;
    while(catégorie != 'f')
    {
        cin >> prix;
        switch(catégorie)
        {
            case 'a':
            case 'c':
                somme += 1.06f * prix; break;
            case 'd':
                somme += 1.08f * prix; break;
            case 'e':
                somme += 1.11f * prix; break;
            case 'b':
                somme += prix; break;
            default:
                cerr << "mauvaise catégorie '" << catégorie << "' "
                    << endl;
                break;
        }
        cin >> categorie; // fin du switch()
    } // fin de la boucle while()
    cout << endl;
    cout << "Montant total = " << std::fixed
        << std::setprecision(2) << somme << endl;
} // fin du main()
```

```
Exécution:
Tapez la catégorie, puis le prix (f pour terminer)
a 339.95
b 15.17
e 6.25
```



```
a 236
c 1133.5
b 142.15
b 137
d 12.45
f
```

Montant total = 2126.72

Considérons l'exemple complet précédent. Dans un grand magasin, la caissière enregistre sur une console d'ordinateur le prix des achats des clients accompagnés d'une lettre indiquant la catégorie des marchandises :

- 'a' articles ménagers, taxe : 6%
- 'b' vêtements, pas de taxe
- 'c' meubles, taxe : 6%
- 'd' outils, taxe : 8%
- 'e' parfums, taxe : 11%

Introduction au Langage C – Fonctions et décomposition d'un programme

La décomposition d'un programme

Qu'elles soient appelées fonctions, procédures ou sous-routines, les fonctions se retrouvent dans tous les langages de haut niveau. Une fonction C est un ensemble d'opérations qui accomplit une tâche donnée. Un programme consiste en un groupe de fonctions. Une des principales raisons d'être des fonctions est de permettre de décomposer un programme en parties logiques plus simples. Prenons par exemple le problème de la résolution d'une série d'équations du second degré.

$$ax^2 + bx + c = 0$$

On doit lire les coefficients **a**, **b**, **c** ; résoudre les équations et afficher leur solution. On s'arrête quand le coefficient **a** vaut 0. Le problème peut se programmer en C à un niveau très général :

```
int main()
{
    while(il_y_a_une_équation())
    {
        lire_les_coefficients() ;
        calculer_solutions_et_afficher() ;
    }
}
```

Nous avons ramené notre problème à trois sous-problèmes :

1/ *il_y_a_une_équation()* revient à tester si le coefficient de l'équation est non-nul ;

```
a != 0
```

ce qui implique de connaître la valeur du coefficient **a** de la première équation avant le début de la boucle.

2/ *lire_les_coefficients()* se résume donc à :

```
cin >> a >> b >> c ;
```

3/ *calculer_solutions_et_afficher()* est un sous-problème plus complexe ; il faut en effet calculer le discriminant delta puis suivant sa valeur, on a une, deux ou pas de solutions réelles. On a donc la décomposition suivante :

```
delta = calculer_delta() ;
if(delta < 0)
    écrire_message() ;
else if (delta == 0)
    solution_unique() ;
else
    deux_solutions() ;
```

On peut donc remplacer *calcul_delta()* par :

```
delta = b * b - 4 * a * c ;
```

Et *écrire_message()* simplement par :

```
cout << "pas de solutions réelles"
<< endl;
```

Nous pouvons donc à ce stade écrire la fonction C correspondante :

```
void calculer_solutions_et_afficher()
{
    delta = b * b - 4.0 * a * c ;
    if(delta < 0)
        cout << "pas de solutions réelles" << endl;
    else if (delta == 0)
        solution_unique() ;
    else
        deux_solutions() ;
}
```

On remarque le mot clef **void** qui signifie que la fonction ne retourne aucun résultat.

Maintenant nous pouvons donner les deux fonctions C qui impriment les solutions pour les deux autres cas :

$$\text{delta} = 0 : x = \frac{-b}{2a}$$

$$\text{delta} > 0 : x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, x_2 = \frac{-b}{a} - x_1$$

```
void solution_unique()
{
    x = -b / (2.0 * a) ;
    cout << "la solution est " << x << endl;
}

void deux_solutions()
{
    x1 = (-b - sqrt(b * b - 4.0 * a * c)) / (2.0 * a) ;
    x2 = -b / a - x1 ;
    cout << "les 2 solutions sont " << x1 << ", " << x2 << endl;
}
```

Avant d'écrire le programme complet, il faut noter qu'un certain nombre de variables doivent être déclarées : *a*, *b*, *c*, *x*, *x1*, *x2*, *delta*. Mais où ?

Déclarations globales et locales

Pour répondre à la question de la section précédente, il faut tenir compte de l'usage des variables. En effet, x n'est utilisée que par la fonction `solution_unique()`, $x1$ et $x2$ que par la fonction `deux_solutions()`, tandis que a, b, c et Δ sont utilisés par toutes les fonctions. On déclarera donc x seulement dans `solution_unique()` et $x1$ et $x2$ dans `deux_solutions()`. Ce sont des variables **locales**. Par contre a, b, c et Δ seront déclarées au début du programme. Ce sont des variables **globales**. Les variables globales sont connues dans toutes les fonctions tandis que les variables locales ne sont connues que dans la fonction où elles sont déclarées.

```
// programme calculant les solutions d'une équation du second
degré
// Le programme a été développé selon décomposition modulaire

#include <iostream>
#include <math.h>
using namespace std ;

// déclarations des variables globales
double a, b, c, delta ;

void solution_unique()
{
    double x = -b / (2.0 * a) ; // variable locale
    cout << "la solution est " << x << endl;
}

void deux_solutions()
{
    // variables locales
    double x1 = (-b - sqrt(b * b - 4.0 * a * c)) / (2.0 * a) ;
    double x2 = -b / a - x1 ;
    cout << "les 2 solutions sont " << x1 << ", " << x2 << endl;
}

void calculer_solutions_et_afficher()
{
    delta = b * b - 4.0 * a * c ;
    if(delta < 0)
        cout << "pas de solutions réelles" << endl;
    else if (delta == 0)
        solution_unique() ;
    else
        deux_solutions() ;
}

int main()
{
    cout << "coefficients : ";
    cin >> a ;
    while(a != 0.0)
    {
        cin >> b >> c ;
        calculer_solutions_et_afficher() ;
        cout << "coefficients : ";
        cin >> a ;
    }
}
```

```

Exécution:
coefficients : 1 5 6
les 2 solutions sont -3, -2
coefficients : 1 -2 1
la solution est 1
coefficients : 3.45 4.5 150
pas de solutions réelles
coefficients : 2.4 4.5 0.25
les 2 solutions sont -1.81769, -0.0573071
coefficients : 0.5 0.25 .5
pas de solutions réelles
coefficients : 1 1 -1
les 2 solutions sont -1.61803, 0.618034
coefficients : 0

```

Introduction au Langage C – Paramètres de fonction/procédure

Passage de paramètres

Considérons le problème suivant : on doit lire un caractère et l'afficher 10 fois, puis afficher 5 fois le caractère + et enfin de nouveau 6 fois le caractère lu au début.

Essentiellement, le problème se ramène à afficher 3 séries de caractères. Il nous faut donc une procédure qui affiche une série de n caractères. Cette procédure peut s'écrire :

```

void afficher()
{
    int i ;
    for(i=0 ; i < n ; ++i)
        cout << c;
}

```

Pour résoudre notre problème, il est judicieux d'employer des paramètres. C'est-à-dire de définir la fonction *afficher(c, n)* qui affiche n fois le caractère c . Voici la déclaration C d'une telle procédure :

```

void afficher(char c, int n)
{
    int i ;
    for(i=0 ; i < n ; ++i)
        cout << c;
}

```

L'appel de cette procédure se fait dès lors très simplement depuis le programme :

```

int main()
{
    char c ;
    cin >> c ;
    afficher(c, 10);    // afficher 10 fois le caractère c
    afficher('+', 5);   // afficher 5 fois '+'
    afficher(c, 6);     // afficher 6 fois le caractère c
}

```

Exécution:

```

$
$$$$$$$$$++++$$$$$$$

```

Les fonctions que nous avons définies jusqu'à maintenant ne retournaient pas de résultat (ce sont donc des procédures), ce qu'on indiquait par le mot clef *void*. Pour retourner une (et une seule) valeur, une fonction doit en indiquer le type lors de la déclaration de la fonction. C'est l'instruction **return(expression)** qui doit figurer dans la fonction et qui retourne la valeur de l'expression. Le type de la valeur retournée après l'évaluation de l'expression doit correspondre à celui spécifié lors de la déclaration de la fonction. L'instruction *return* peut figurer à plusieurs endroits d'une fonction et elle redonne le contrôle au programme appelant.

Il faut aussi remarquer que les arguments passés à une fonction C sont passés par valeur. Donc, même si la fonction modifie les arguments, cela ne modifiera en rien les variables dans le programme appelant.

```

float puissance_entiere(float x, int n)
// calcule x à la puissance n
{
    float puis = 1.0f ;
    for( ; n > 0 ; --n)
        puis *= x ;
    return puis ;
}

int main()
{
    int i=4 ;
    float a = 4.532, b;
    // i ne sera pas modifié lors de l'appel
    b = puissance_entiere(a, i) ; // appel de fonction
    cout << a << " à la puissance " << i << " vaut " << b << endl ;
}

```

Exemple : On désire afficher le binôme de Newton $(x+y)^n$ sous la forme :

```

x+y
x2+2xy+y2
x3+3x2y+3xy2+y3
...

```

On calcule les coefficients par la formule des combinaisons :

$$(x + y)^n = \sum_{i=0}^n C_n^i x^{n-i} y^i$$

$$C_p^q = \frac{p!}{q!(p-q)!}$$

On définira donc deux fonctions :

1/ *int fact(short f)* ; qui calcule f!

2/ *short coeff(short p, short q)* ; qui calcule C_p^q .

```
// programme affiche le binôme de Newton (x+y) à la puissance n

#include <iostream>
#include <math.h>
using namespace std ;

// déclaration
void aff(short n, short i, short c, short nmi);

int fact(short f) // calcule f!
{
    short nombre =1;
    int factorielle =1;
    do
    { //possibilité d'optimiser, éviter 1*1
        factorielle = factorielle * nombre; ++nombre;
    } while(nombre <= f);
    return factorielle;
}

short coeff(short p, short q) // calcule les combinaisons C(p,q)
{
    return(fact(p)/(fact(q)*fact(p-q))) ;
}

int main()
{ short n, i, c; //n = degré
  for(n=1; n <=6; ++n) { // (x+y)n pour n=1..6
    for(i=0; i <=n; ++i) { //Σ pour i=0..n
      c = coeff(n, i); //C(n,i)
      aff(n, i, c, n-i);
    }
    cout << endl;
  }
}
```

```
void aff(short n, short i, short c, short nmi)
{
    if(c!=1) //évite d'afficher "1*"
        cout << c << "*";
    if(nmi!=0)
    { //il y a un terme xn-i
        if(nmi == 1) cout << "x"; //x1
        else cout << "x^"<< nmi; //xn-i
        if(i!=0) cout << "*"; //il y a un terme y qui suit...
    }
    if(i!=0) { //il y a un terme yi
        if(i == 1) cout << "y"; //y1
        else cout << "y^"<< i; //yi
    }
    if(i!=n) //reste t'il un terme ?
        cout << "+"; // ... oui
}
}
```

Exécution:

```
x+y
x^2+2*x*y+y^2
x^3+3*x^2*y+3*x*y^2+y^3
x^4+4*x^3*y+6*x^2*y^2+4*x*y^3+y^4
x^5+5*x^4*y+10*x^3*y^2+10*x^2*y^3+5*x*y^4+y^5
x^6+6*x^5*y+15*x^4*y^2+20*x^3*y^3+15*x^2*y^4+6*x*y^5+y^6
```

Introduction au Langage C – Portée et persistance des variables

On distingue la région d'un programme où une variable peut être référencée : sa portée, du stockage temporaire (variable temporaire) ou permanent (variable statique) de sa valeur en mémoire.

Portée d'une variable

Nous avons déjà étudié les variables globales et locales. La portée d'une variable en C correspond aux blocs de contrôle dans lesquels réside la définition de la variable, sachant qu'une variable ne peut être référencée avant sa déclaration.

```
#include <iostream>
using namespace std ;

// variable globale
int N = 500 ; // début de portée de N

int premier_terme(int somme) // début de portée de somme
{
    int dernier = 1 ; // début de portée de dernier

    do
    {
        int adernier = dernier ; // début de portée de adernier
        dernier=somme ;
        somme = adernier + dernier ;
    } // fin de portée de adernier
    while(somme <= N) ;
} // fin de portée de dernier, somme (paramètre)

int main()
{
    int somme = 1 ; // début de portée de somme

    somme = premier_terme(somme) ;
    cout << "Premier terme de la suite de Fibonacci supérieur
à "
        << N << " vaut " << somme << endl ;
} // fin de portée de somme

// fin de portée de N, et de la fonction premier_terme()
```

Dans la fonction *premier_terme()*, la variable *dernier* est une variable locale, sa portée consiste en le corps de la fonction. En dehors de cette fonction, cette variable est inconnue. La portée de la variable globale '*N*' s'étend à tout le module, elle peut donc être référencée dans les fonctions *main()* et *premier_terme()*. Par contre, si sa définition était contenue dans le corps de la fonction *main()*, comme variable locale, il faudrait la passer par argument à la fonction *premier_terme()* pour que celle-ci puisse l'utiliser.

Variables temporaires et statiques

Une variable statique est une variable conservant sa valeur hors de sa portée. On utilise le mot clef *static* lors de la déclaration de la variable pour indiquer ce type de variable. Les variables non-statiques, dites temporaires, conservent leurs valeurs seulement dans la portée de la variable. En fait, la mémoire associée à une variable temporaire est dynamiquement allouée en début de portée puis libérée en fin de portée par le compilateur. Une telle variable n'a donc qu'une durée de vie limitée.

Dans l'exemple suivant, la variable *puis* est une variable temporaire de la fonction *puissance_entiere()*. Si nous appelons cette fonction avec les paramètres 3.0 et 4, nous obtenons comme résultat la valeur 81.0. Que se passe-t-il maintenant si nous répétons encore appelons à nouveau cette fonction avec ces mêmes paramètres ?

Pour répondre à cette question, il faut suivre l'exécution de la fonction. Lors du premier appel, *puis* est initialisée à la valeur 1.0 et vaut 81.0 à la sortie. Lors du second appel, est-ce que la variable est réinitialisée à 1.0 ou continue-t-elle à valoir 81.0 ? Ceci dépend de la persistance de la variable, une variable temporaire, comme *puis*, est réinitialisée à chaque entrée dans la portée de la variable, et sa valeur est perdue dès que l'on sort de sa portée. Donc *puis* vaut 1.0 à l'entrée de *puissance_entiere()*, 81 à sa sortie, est indéfini hors de la portée, puis vaut de nouveau 1.0 à l'entrée et 81 à la sortie. Si par contre, nous utilisons une variable statique, telle que *stat_puis*, grâce au mot clef *static*, la valeur 81 est retrouvée lors du deuxième appel de la fonction *stat_puissance_entiere()*. Ainsi, le résultat du second appel à cette fonction est carrément faux, car le calcul commence avec la valeur 81. Une variable statique n'est initialisée qu'une seule fois et permet de conserver une valeur d'un appel de fonction à l'autre.

```
float puissance_entiere(float x, int n)
// calcule x à la puissance n
{
    float puis = 1.0f ;// variable temporaire
    for( ; n > 0 ; --n)
        puis *= x ;
    return puis ;
}

float stat_puissance_entiere(float x, int n)
// calcule x à la puissance n
{
    static float stat_puis = 1.0f ;// variable statique
    for( ; n > 0 ; --n)
        stat_puis *= x ;
    return stat_puis ;
}

int main()
{
    int i=4 ;
    float a = 3.0f;

    cout << puissance_entiere(a, i)
          << " " << stat_puissance_entiere(a, i) << endl ;
    cout << puissance_entiere(a, i)
          << " " << stat_puissance_entiere(a, i) << endl ;
}
```

Exécution:
81.0000 81.0000
81.0000 6561.0000

Autres mots clefs

Il existe trois autres types de modificateurs d'accès pour des variables : *register*, *const* et *volatile*. Une variable dont la déclaration est précédée du mot *register* indique qu'elle est manipulée très souvent et que le compilateur doit essayer de la conserver de préférence dans un registre pour augmenter l'efficacité du programme.

Le modificateur *const* permet de déclarer une constante, c'est-à-dire que la valeur initiale ne pourra pas être modifiée au cours du programme.

Enfin, une variable est qualifiée de *volatile* est une variable qui peut changer de valeur au cours de l'exécution du programme sans que celui-ci en soit responsable ; Par exemple, l'adresse d'une variable peut être passée à une fonction temps du système d'exploitation et utilisée pour mémoriser l'heure.

```
register int n;  
const float pi = 3.14159f;  
volatile float temps;
```

Signalons encore que les modificateurs *volatile* et *const* sont définis dans les langage C ANSI, mais pas toujours implantés dans les compilateurs.

Fonctions et variables externes

Nous avons déjà vu la notion de variables globales et locales. Nous pouvons donner comme nouvelle définition d'une variable locale : une variable dont la portée est limitée à une fonction ou un bloc de contrôle. Une variable globale est une variable définie à l'extérieur de toute fonction et de tout bloc. Mais alors quelle est la portée d'une variable globale ? Si nous considérons un programme contenu entièrement dans un seul fichier, la portée d'une variable globale va de la définition de la variable à la fin du fichier. Mais un programme peut aussi faire appel à des fonctions contenues dans d'autres fichiers. Dans ce cas, la portée d'une variable globale s'étend aux différents fichiers. En effet, on peut définir une variable (ou fonction) dans un fichier et l'utiliser dans un autre. La définition se fait comme habituellement et correspond à une allocation de mémoire pour cette variable. Les fichiers qui référencent cette variable (ou

fonction) doivent contenir une allusion à celle-ci, cette allusion ne correspond à aucune allocation mémoire et se présente comme une déclaration mais précédée du mot clef réservé *extern*.

```
Fichier func.c :  
  
extern int n ; //définition d'une variable externe  
  
float puissance_entiere(float x)  
{  
    float puis = 1.0f ;// variable temporaire  
    for( ; n > 0 ; --n)  
        puis *= x ;  
    return puis ;  
}  
  
Fichier main.c :  
  
//définition d'une fonction externe :  
extern float puissance_entiere(float x) ;  
  
int n = 4 ; // définition d'une variable globale  
  
int main()  
{  
    cout << puissance_entiere(3.0f) << endl ;  
}
```

Si on avait déclaré la variable *n* dans le fichier func.c sans le mot clef *extern*, on aurait eu un problème au niveau de l'édition de liens (link), car la variable aurait été considérée comme définie deux fois dans le même programme. Il est possible de déclarée des variables globales de même nom dans deux fichiers différents à la condition de les déclarer statiques, en précédant leur déclaration du mot clef *static*. Dans ce cas, chaque variable est considérée comme globale, mais seulement au niveau du fichier concerné. Les déclarations de variables globales statiques sont surtout utiles pour définir des variables partagées par les fonctions d'un module. Il est vivement conseillé d'utiliser des

variables globales statiques lorsque leur usage reste confiné dans un module source afin d'éviter :

- Les problèmes d'édition de liens provenant de multiples définitions de variables utilisant le même nom.
- La possibilité de modification indésirable/incontrôlable de la valeur d'une variable par un autre module.

Pointeurs comme arguments de fonctions

Lorsque l'on désire permettre à une fonction de changer les valeurs de certains arguments de la procédure appelante, on va lui passer les adresses de ces variables au moyen de pointeurs.

```
void puissance_entiere(float x, int n, float * y)
{
    float puis = 1.0f ;
    for( ; n > 0 ; --n)
        puis *= x ;
    // affecte la valeur de puis dans la cible mémoire
    // du pointeur y
    *y = puis ;
}

int main()
{
    float puis ; // sera modifiée dans la fonction suivante :
    puissance_entiere(3.0f, 4, &puis) ;
    cout << puis << endl ;
}
```

Le symbole « * » figurant dans la déclaration d'une variable désigne un pointeur sur le type précédant le *, dans notre exemple *float* * y. Ce même symbole précédant le nom de la variable lors d'une référence indique que nous voulons accéder à la variable et non pas au pointeur lui-même, *y = *puis*. Lors de l'appel de la fonction, nous devons passer le pointeur sur notre variable de type *float*, pour cela on utilise le symbole « & » devant le nom de la variable *puis* : *puissance_entiere(3.0f, 4, &puis)* ;

Introduction au Langage C – Les tableaux

Les tableaux unidimensionnels

Un important problème est le suivant : comment mémoriser un grand nombre de valeurs, sans utiliser un grand nombre d'identificateurs de variables, et tout en laissant accessible n'importe laquelle de ces valeurs ?

Le meilleur moyen consiste à donner un nom unique à toutes ces valeurs et à les référencer au moyen d'un numéro ou d'une adresse. Une telle variable, permettant de mémoriser plusieurs valeurs, est appelée variable indexée ou tableau. Le numéro qui permet d'accéder à une valeur particulière du tableau est appelé indice ou index. Les valeurs elles-mêmes constituent des composantes du tableau.

La déclaration d'un tableau unidimensionnel se fait en précisant le nombre d'éléments contenus dans le tableau, c'est-à-dire sa dimension.

```
int    nb[200];
char   tab[10];
float  prix[30];
```

Attention : les indices en C vont de 0 à $n-1$ et NON de 1 à n !

```
#define NOMBRE 20

int main()
{
    int i ;
    int nb[NOMBRE] ;
    for(i = 0 ; i < NOMBRE ; ++i)
        cin >> nb[i] ;
    for(i = NOMBRE-1 ; i >= 0; --i)
        cout << nb[i] << endl ;
}
```

Exemple : On désire calculer la fréquence des lettres d'un texte. On utilise un tableau dont les composantes sont des entiers et l'indice constitue la lettre dont on compte l'occurrence.

```
#define MAXLETTRE 26

int main()
{
    char c ;
    int freq[MAXLETTRE] ;
    // initialisation du tableau
    for(c = 'a' ; c <= 'z' ; ++c)
        freq[c-'a'] = 0 ;
    do {
        cin >> c ; // lit un caractère
        if(c >= 'a' && c <= 'z')
            ++freq[c-'a'] ;
    } while(c != '.') ;
    for(c = 'a' ; c <= 'z' ; ++c)
        cout << c << " " << freq[c-'a'] << endl ;
}
```

Exécution:

un chasseur sachant chasser sans son chat.

a 6
b 0
c 4
d 0
e 2
f 0
g 0
h 4

```

i 0
j 0
k 0
l 0
m 0
n 4
o 1
p 0
q 0
r 2
s 8
t 2
u 2
v 0
w 0
x 0
y 0
z 0

```

Pointeurs et tableaux

Il existe une relation étroite entre les tableaux et les pointeurs : en effet, toute opération sur un tableau qui peut être faite en utilisant les indices peut aussi bien se faire à l'aide d'un pointeur.

Un tableau déclaré avec :

```
int tab[4] ;
```

définit un bloc mémoire de 5 entiers rangés consécutivement et nommés :

```
tab[0], tab[1], tab[2], tab[3]
```

« tab[i] » désigne l'objet placé à la $i^{\text{ème}}$ position à partir du début.

Soit un pointeur *pta* déclaré par :

```
int * pta;
```

et initialisé à

```
pta = tab ; // équivaut à pta = &(tab[0]) ;
```

Alors *pta* pointe vers le premier élément du tableau et

```
z = *pta ; équivaut à z = tab[0] ;
```

En général *pta + i* pointe vers l'élément en $i^{\text{ème}}$ position dans le tableau. **(pta + i)* correspond au contenu de *tab[i]* et *pta + i* à l'adresse de *tab[i]*.

Voici trois exemples d'initialisation d'un tableau fonctionnellement identiques. La version 1 est réalisée avec un pointeur, la seconde avec un indexage du tableau, et la troisième en utilisant le pointeur associé au tableau (*tab*) en prenant l'index *i* comme offset d'adresse mémoire :

```
int * pta, tab[50], i;
for(pta = tab, i = 0; i < 50; ++i)
    *pta++ = i; //version 1
for(i = 0; i < 50; ++i)
    tab[i] = i; //version 2
for(i = 0; i < 50; ++i)
    *(tab+i) = i; //version 3
```

L'incrémement du pointeur (*pta++* ou *pta += 1*) tient compte de la taille de la donnée en mémoire, et le pointeur sera en fait incrémenté de 1, 2, 4 ou 8 etc. selon le type de données vers lequel il pointe. C'est pourquoi il est nécessaire de déclarer le type du pointeur et il n'est pas possible de substituer un pointeur vers un entier par un pointeur vers un réel.

Le compilateur considère en fait le nom d'un tableau comme un pointeur vers l'élément initial du tableau.

`pta = &tab[0]` ; peut s'abrégier par `pta = tab` ;
et `z = *(tab +4)` ; est équivalent à `z = tab[4]` ;

Il faut faire attention que le nom du tableau (*tab*) est une constante, tandis que *pta* est une variable. Il est donc illégal de changer la valeur de *tab*. Du coup, les opérations suivantes ne sont pas admises :

```
tab++ ; --tab ; tab += 5 ;
```

alors qu'on peut parfaitement modifier *pta*. Ainsi les opérations suivantes sont parfaitement légales :

```
pta++ ; --pta ; pta += 5 ;
```

Les tableaux multidimensionnels

Un tableau à deux dimensions ou plus se déclare de façon similaire à un tableau unidimensionnel, en précisant les dimensions de chaque coordonnée entre crochets.

```
int vect[80][20];
char tab[10][10];
float prix[30][2][50];
```

En mémoire, les éléments sont rangés par ligne ; si on suit l'ordre de la mémoire, c'est donc le 2^{ième} indice qui varie en premier.

Il faut noter que la référence `vect[62][2]` est interprétée comme `*(vect[62]+2)`, donc comme `*(*(vect+62)+2)`.

D'autre part, un tableau multidimensionnel peut être initialisé lors de la déclaration de la façon suivante :

```
int vect1[2][3] = {{0, 1, 5}, {7, 3, 2}};
int vect2[3][2][1] = { {{0},{1}}, {{5},{7}}, {{3},{2}} };
```

Les tableaux peuvent avoir autant d'indices qu'on le souhaite, pourtant au-delà de 3, la clarté des programmes diminue considérablement.

Tableaux comme arguments de fonctions

Un tableau peut être passé comme argument d'une fonction. Mais comme nous avons vu qu'un tableau peut être accédé par indice ou par pointeur, il est aussi possible d'en tenir compte et deux déclarations différentes sont possibles. Par exemple :

```
void traiter(int *t) ;
void traiter(int t[]) ;
```

On remarque que dans le second cas, on n'a pas indiqué la taille du tableau. On peut l'indiquer, mais cela est inutile car une fonction ne réserve aucune place pour le tableau. C'est le programme appelant qui a fait la réservation de mémoire.

Du point de vue traitement, il n'y a pas de différence entre les types de déclaration. Cependant dans le second cas, on a l'avantage de pouvoir utiliser la fonction prédéfinie `sizeof(t)` pour connaître la taille du tableau *t*.

Pour un tableau multidimensionnel, il est indispensable de déclarer toutes les dimensions sauf la première.

```
void traiter(int m[][3][4][3]) ;
```

Il faut encore noter qu'à cause de la représentation des tableaux et de leur relation étroite avec les pointeurs, on peut modifier le contenu des tableaux passés en arguments dans la fonction. Ces tableaux ne sont pas passés par valeur.

Nous allons maintenant présenter un exemple : le tri de nombres.

Nous désirons écrire une fonction qui classe une série de 11 nombres entiers. La méthode utilisée est la suivante :

On compare chaque nombre avec les suivants.

Si l'ordre n'est pas correct, on échange les nombres, sinon on les laisse tels quels.

Par exemple, avec 4 nombres, on a :

```

5   3   2   4
  ←→
3   5   2   4
  ←→→
2   5   3   4
  ←-----→
2   5   3   4
      ←→
2   3   5   4
      ←-----→
2   3   5   4
          ←→
2   3   4   5

```

Les nombres sont placés dans un tableau déclaré comme :

```
int nombres[N] ;
```

La procédure de tri est alors déclarée comme :

```
void tri(int t[]) ;
```

Elle a donc un paramètre qui est à la fois la série de nombres avant le tri et après le tri. On perd donc les valeurs de la série originale après le tri. Voici le programme de tri :

```

#define N 10

void tri(int t[])
{
    int i, j, temp;

    for(i=0; i < N; ++i)
        for(j=i+1; j <= N; ++j)
            if(t[i] > t[j]) { temp = t[i];
                           t[i] = t[j];
                           t[j] = temp;
                        }
}

int main()
{
    int nombre[N+1], i;
    for(i = 0; i <= N; ++i)
        cin >> nombre[i];
    tri(nombre);
    cout << endl;
    for(i = 0 ; i <= N; ++i)
        cout << " " << nombre[i];
    cout << endl;
}

```

Exécution:

```
23 35 56 344 -12 345 34 123 0 -56 84
```

```
-56 -12 0 23 34 35 56 84 123 344 345
```

Dans le programme ci-dessus, nous remarquons que la fonction de tri fait référence à la constante N, qui est le nombre

d'éléments dans le tableau. Ceci rend la fonction dépendante du contexte. Pour éviter cela, on peut passer la dimension en paramètre, l'entête et l'appel de la fonction deviennent respectivement :

```
void tri(int t[], int dim) ;

tri(nombre, N) ;
```

Tableaux de fonctions

Le langage C, comme d'ailleurs la plupart des langages de programmation, ne permet pas de définir des tableaux de fonctions, ce qui est très utile. Par contre, il permet de définir des pointeurs vers des fonctions. Comme il est aussi possible de définir des tableaux de pointeurs, on arrive finalement à créer des tableaux de pointeurs vers des fonctions. Par exemple, on peut déclarer :

```
double (*f[6])() ;
```

qui correspond à un tableau de 6 pointeurs vers des fonctions retournant un résultat de type *double*. A chaque pointeur *f[i]*, on peut donc affecter un nom de fonction retournant un résultat de type double, comme *sin()* ou *sqrt()* par exemple. Le programme suivant nous montre un exemple typique d'utilisation de tableaux de pointeurs vers des fonctions. Il suffit de taper un numéro de fonction *num*, qui est interprété comme indice dans le tableau de pointeurs et on accède à la bonne fonction *f[num]*.

```
#include <iostream>
#include <math.h>
using namespace std;

#define N 6

int main()
{
    double (*f[N])() = {sin, cos, tan, exp, log, sqrt}, x ;
    int num;
    cout << "fonctions à disposition :" << endl;
    cout << "0=sin(x), 1=cos(x), 2=tan(x), 3=exp(x), "
        << "4=log(x), 5=sqrt(x), 6=fin "<< endl << endl;
    cout << "numéro de la fonction f ? ";
    cin >> num;
    while(num >=0 && num < 6) {
        cout << "valeur de la variable x ? ";
        cin >> x ;
        cout << "f["<<num<<"](" << x<< ") = "<< f[num](x)
<<endl << endl ;

        cout << "numéro de la fonction f ? ";
        cin >> num;
    }
}
```

Exécution:

```
fonctions à disposition :
0=sin(x), 1=cos(x), 2=tan(x), 3=exp(x), 4=log(x),
5=sqrt(x), 6=fin
numéro de la fonction f ? 3
valeur de la variable x ? 1
f[3](1)= 2.718282

numéro de la fonction f ? 4
valeur de la variable x ? 10
f[4](10)= 2.302585

numéro de la fonction f ? 1
valeur de la variable x ? 5
f[1](5)= 0.283662

numéro de la fonction f ? 5
valeur de la variable x ? 3
f[5](3)= 1.732051

numéro de la fonction f ? 6
```

Manipulation de texte

Contrairement à certaines croyances, l'ordinateur n'est pas seulement un outil permettant d'effectuer de nombreux calculs. Si ces applications sont, en effet, très nombreuses dans les sciences telles que la physique, la chimie, la biologie, l'astronomie ou les sciences économiques, le traitement des chaînes de caractères (traitement de texte) : éditeurs de texte, programmes de mise en page, compilateurs, traducteurs de langages naturels forment une autre classe d'applications.

Nous avons déjà vu comment on définit une chaîne de caractères comme "*Bonjour*" en utilisant le type C++ *string*. En C, une chaîne de caractères est en fait un tableau de caractères termine par le caractère spécial '\0' (caractère de code zéro, appelé caractère nul).

Pour déclarer une variable contenant au maximum 10 caractères, on peut écrire :

```
char texte[10] = "impossible";
```

En C ANSI, la chaîne est contenue dans les 10 composantes *texte[0]* à *texte[9]*, un caractère nul est stocké en plus dans *texte[10]*. Avec certains anciens compilateurs C, le caractère nul doit être prévu par la programmeuse. Dans ce cas, il est nécessaire de déclarer :

```
char texte[11] = "impossible";
```

Il faut noter qu'il est aussi possible de ne pas spécifier la taille de la chaîne :

```
char texte[] = "impossible";
```

Dans ce cas, nous sommes certain d'avoir une chaîne de caractères comprenant tous les caractères et terminée par un caractère nul.

Puisqu'une chaîne de caractères est un tableau, elle peut être aussi définie à l'aide d'un pointeur :

```
char *texte = "impossible";
```

Affectation de chaîne de caractères

Les affectations pour des variables chaînes de caractères sont assez complexes et sont souvent sources d'erreurs. Pour expliquer les effets de ces affectations, nous allons utiliser des exemples. Pour cela, nous allons considérer les déclarations suivantes :

```
char texte[10] = "impossible";  
char *texte2 = "impossible";
```

Supposons maintenant l'affectation :

```
texte = "possible";
```

Ceci est impossible, car *texte* est une adresse constante. Par contre, nous pouvons écrire :

```
texte2 = "possible";
```

car dans ce cas, cela revient à changer la valeur du pointeur *texte2*. Les deux affectations suivantes sont toutes les deux possibles :

```
texte[0] = '*';  
texte2[0] = '*';
```


Attention toutefois qu'une affectation comme

```
texte2[20] = '*';
```

va au delà de la taille de la chaîne initiale et l'on accède à une zone de mémoire invalide pour y placer le caractère '*', ce qui peut entraîner une erreur fatale lors de l'exécution du programme.

Il faut aussi noter que l'affectation suivante est impossible :

```
*texte2 = "impossible";
```

alors que l'initialisation

```
char *texte2 = "impossible";
```

est tout à fait correcte comme nous l'avons déjà vu. En effet, l'affectation correspond à affecter une adresse à une variable de type *char*, tandis que l'initialisation revient à créer un tableau de caractère dont *texte2* va contenir l'adresse du premier.

Il est aussi important de distinguer un caractère d'une chaîne de 1 caractère. Supposons, par exemple, les déclarations suivantes :

```
char c = 'x';      // variable caractère
char *tx = "x";    // pointeur sur une chaîne de
                  // caractères de dimension 1
```

Considérons les 6 affectations suivantes dont seules 3 sont permises :

```
c = 'y'; // OK, redéfinit le contenu de la variable
c = "y"; // illégal, ce n'est pas un pointeur
tx = 'y'; // illégal, tx n'est pas un char
tx = "yz"; // OK, redéfinit le contenu du
           // pointeur
*tx = 'y'; // OK, redéfinit le contenu du
           // premier caractère de la chaîne
```

```
*tx = "y"; // illégal, on ne peut pas affecter
           // une adresse à une variable char
```