CHAPITRE 8

PROGRAMMATION PAR OBJETS C++

La programmation par objets est née vers 1972 avec le langage Smalltalk lui-même hérité de Simula. Nous avons choisi le langage à objets le plus répandu actuellement C++, cependant nous ne présentons ici que les spécificités essentielles qui le distingue des langages impératifs classiques : les notions de hiérarchie de classes, de généricité et de modularité, mais aussi l'usage généralisé des pointeurs. Le C++, créé par B. Stroustrup en 1986, est surtout utile pour la réalisation de grands logiciels et peu adapté à l'expérimentation de petits problèmes comme ceux qui figurent dans ce livre, par ailleurs, voulant tout faire, le C++ n'est pas d'une grande cohérence pour la syntaxe.

Syntaxe des déclarations et instructions de base du C et du C++

Les fonctions Il n'y a en C que des fonctions retournant un type quelconque, définies par :
< type du résultat > < nom de fonction > (<type de chaque paramètre > liste de paramètres)
// commentaires, qui en C sont aussi encadrés par /* ... */
{ type des variables locales ; (leur nom et une éventuelle initialisation)
suite d'instructions suivies par des ;
return < expression du type attendu > ; }
Par exemple, la fonction "x puissance n" se programmera par :

```
int puissance (int x, int n)              // par défaut en C le résultat est entier, en C++, il faut le spécifier
        {int p = 1; for (int i = 1; i <= n; ++i) p *= x; return p ; }
```

On remarquera que déclaration et initialisation peuvent être conjointes. Une fonction peut être déclarée "inline" si elle est non récursive, le compilateur la recopie alors à la place de chacun de ses appels.

Arguments par défaut : des valeurs par défauts peuvent être mise dans l'en tête d'une fonction, ainsi f(int x = 0, int y = 2, int z = 6, int t = 8) pourra être appelée par f(5, 2), en ce cas les deux derniers arguments seront quand même pris comme 6 et 8. "Return" peut ne pas avoir d'arguments, en ce cas il est facultatif et la fonction n'aura simplement qu'un effet de bord, "void" signifiant "sans type défini".
Autre possibilité intéressante, plusieurs fonctions portant le même nom comme f(int), f(int, int), f(float, char), ... peuvent être utilisée simultanément dans un même programme, le C++ reconnaissant bien d'après les paramètres fournis, laquelle est la bonne.

Opérateurs et instructions

==	égalité	
!=	différent	
<, >, <=, >=	symboles d'ordres usuels pour tous les types définis par énumération	
!	négation	exemple !0 = 1 et !1 = 0 (il n'y a pas de type booléen)
‖	disjonction	exemple 0 ‖ 1 = 1
&&	conjonction	exemple 0 && 1 = 0
%	modulo	exemple 23 % 7 = 2
?	opérateur ternaire qu'on utilise par : test ? expr1 : expr2 (un test est 0 ou 1)	
=	Affectation	exemple p = 1, l'affectation est une fonction, qui renvoie la valeur.

++ Incrémentation ++N au lieu de N = N + 1 (n += 2 sera "n augmenté de 2", p *= 10 sera "p multiplié par 10", plus généralement p *= x pour p := p*x). On peut lier deux affectations, ainsi si n = 5, x = ++n aura pour action x = n = 6 car n est d'abord incrémenté, alors que x = n++ aura pour action x = 5 et en second lieu n = 6.

Sélection
if <test> <instruction> ; ou bien if <test> <instr1> ; else <instr2>; exemple if (c == 'A') ++ n; switch (n) {case 0 : inst; case 1 ; inst; ... default : instr;} est l'équivalent du "case" pascalien.
Exemple d'une fonction inscrivant voyelle ou consomne pour une lettre :

```
void voyelle (char c) {
switch (c) case 'a': case 'e' : case 'i' : case 'o' : case 'u' : case 'y' :  printf ("voyelle") default : printf ("consomne");}
```

Entrées : En faisant appel à la bibliothèque <stdio.h>, on a c = getchar(); pour un caractère, scanf ("%d%c", n, c) ; pour l'entrée d'un entier et d'un caractère,
% est ici un opérateur de conversion, d pour entier, f pour réel, c pour caractère et s pour une chaîne. Scanf est une fonction, c'est pourquoi, on peut parfaitement écrire :
"while (scanf ("%d", &x) > 0) {...}" pour un traitement à faire tant que x est un entier positif, ou encore "while (p--)" qui permet de tester p différent de 0, puis de le décrémenter.
En faisant appel à la bibliothèque <iostream.h> on utilisera plus simplement cin >> n >> c

Sorties : putchar (c); pour un caractère, printf ("...") pour une chaîne et à la fin de ce message \b pour un "backspace", \t pour une tabulation, \n pour un retour à la ligne.
On utilisera aussi l'opérateur << avec par exemple cout << "le résultat est " << n;

Boucles : Trois possibilités : for (<initialisation>; <condition>; incrémentation>) <instruction>
ou while <test> {instr1; instr2; ..... } ou do <instruction> while <test>, exemple :

```
for (int x = 0; x < 20; x = x + 2) // Noter la déclaration du compteur x au sein de la boucle
        printf ("%6.1f\n", x) // affiche x converti en réél sur 6 espaces au moins avec 1 décimale
```

Débranchement

Break;	sort d'un niveau d'une instruction et passe la main à l'instruction suivante.
Continue;	permet le passage à l'itération suivante dans une boucle : for (n = 1; n < 9; n++) {if <test> continue; <instruction (si non test)};
Goto	si un label :... est placé dans le programme, "goto label" est le débranchement.

Types de données : On a les "int" (codés sur 16 bits), les "float" codés sur 32 bits, les "char" (1 octet), unsigned int (1 octet), long int (4 octets), double (8 octets), et range ($\pm 10^{\pm 38}$)

Pointeurs et tableaux

L'originalité du C est de pouvoir accéder directement à la mémoire avec les pointeurs qui sont de vraies adresses, et au fait que si p est un pointeur, p+1 est l'adresse suivante. A partir de là toutes les structures de données peuvent être construites à commencer par les tableaux qui ne sont que des pointeurs.
& opérateur adresse, donc p = &x affecte p de l'adresse où est x (p pointe sur x)
* opérateur contenu, donc y = *p fera l'indirection y = x, à noter que les déclarations int* x, et int *x sont équivalentes, exemple :

```
void echange (int *x, int *y); {int t; t = *x; *x = *y; *y = t;}
```

Le passage par référence se marque par & (équivalent du "var" de Pascal), on a donc plutôt :

```
void echange (int & x, int & y);        // échange les valeurs de x et y qui sont passés par référence
        {int t; t = x; x = y; y = t;}
```

Cette transmission est la plus efficace lorsqu'il s'agit d'un objet structuré de taille importante. Cela est vrai dans tous les langages, mais ici elle est plus spécialement utilisée. Afin de protéger l'argument, s'il ne doit pas être modifié, on le fait précéder de "const", pour simuler un passage par valeur.
Le point est l'opérateur d'accès à une donnée ou une méthode sur un objet, par contre, la flèche -> est un opérateur pour obtenir un champ particulier du même objet structuré s'il est accessible au moyen d'un pointeur. Ainsi, si p est un pointeur sur un article de type "fiche" dont des champs particuliers s'appellent "nom", "tel", ... alors, (*p).nom est équivalent à p->nom.
Par exemple, pour une liste chaînée générique, d'éléments de type T, définie par :

```
struct liste {T valeur; liste *suivant; };
```

Si L est une liste, l'expression "L.suivant->suivant->suivant->valeur" permet d'accéder à la valeur du toisième élément de la liste L.
void *p est la déclaration d'un pointeur p sans type défini, c'est une pure adresse machine.
Il n'y a pas de mot réservé "null" ou "nil", c'est l'adresse 0 qui en tient lieu.

Spécificité de la structuration des tableaux en C et C++

Si on déclare int *p = &tab[0]; alors p est un pointeur dont le contenu est l'adresse du premier élément de tab, en ce cas p+2 est une expression valide, c'est un pointeur dont le contenu est l'adresse de tab[2]. Un élément p[i] est équivalent à *(p + i), plus généralement, pour tout tableau appelé p, p est la même chose que &p[0] et p[i] est la même chose que *(p + i)
Remarques, les tableaux sont toujours indicés par des entiers à partir de 0, la déclaration int t[8] voudra dire que t est un tableau d'entiers de 9 éléments indicés de 0 à 8, les tableaux étant des pointeurs, l'opérateur & est sous entendu, il en est de même des fonctions (voir plus loin le passage de fonctions en paramètres).
Les déclarations <int* tab> et <int *tab> sont identiques, elles déclarent "tab" comme un pointeur d'entiers.

```
char s[9]; // est une déclaration de chaîne de caractères d'au maximum 10 symboles
int tab[9] = {3, 5, 9, 2, 4}; // déclaration et initialisation, tab [0] vaut 3, les 5 derniers valant 0
int tab[3][5] = {{1}, {2}, {3}}; // initialise le premier élément de chaque ligne, les autres étant nuls
int tab [2] [5] = {{0,1,2,3,4,5} {0,2,4,6,8,10}}; // déclaration et affectation, tab [1] [3] vaut 3.
char s[5] = "abcde";          // initialise s[0] à 'a' jusqu'à s[4] à 'e'
```

Remarques : 'x' délimite un caractère, "x" délimite une chaîne. Les chaînes de caractères sont des tableaux, qui eux-mêmes sont toujours des pointeurs.

Une chaîne est déclarée par char *s, car ce n'est qu'un tableau de caractères, par exemple :

```
int longueur (char *s) // identique à la déclaration <char s[]>, ou encore <char s[9]>
        {int i = 1; while (s[i] != '\o') ++i; return (i); }          // le caractère "fin de chaîne" est '\o'
```

Type défini par énumération, exemple :

```
enum couleur {violet, rouge, orange, jaune, vert, bleu, noir };
```

Les enregistrements sont déclarés par "struct", exemple :

```
struct date {int j; int m; int a; char nom[];};
date d = {4, 7, 1789, juillet}; // déclaration de d et son affectation
```

Union de plusieurs types : un objet de type union possède un seul type, celui d'un des types contenus dans l'union, par exemple :

```
union nombre {int entier; double reel; };
nombre x, y ;      // est la déclaration de deux "nombres"
x.entier = 4;      // affecte x à 4 et décide que x sera définitivement un "int"
```

Définition d'un nouveau type, par exemple le type "arbre" et son pointeur "arbptr"

```
typedef struct noeud {char : *mot; int num; struct noeud *gauche; struct noeud *droite;} arbre, *arbptr;
typedef enum { vrai = 1, faux = 0 } bool; // autre exemple, définirait le type booléen
```

Structure d'un programme

Un programme C++ est tout à fait analogue à un programme Pascal, il sera nommé sous "Unix" ou sous "Dos" par nom.c ou nom.cpp, le compilateur produira nom.obj, puis l'édition des liens produira un nom.exe. Un programme est une collection de fonctions, il est lui-même terminé par une fonction nommée main (int x, char y) ; par exemple ou bien le plus souvent sans arguments main (). Exemple :

```
# include <nom.ext>          // appel à des unités précompilées on y mettra des classes cohérentes
# include <stdio.h>          // librairie habituelle des entrées-sorties contenant "getchar", "scanf", "printf"...
# include <iostream.h>       // est la librairie permettant d'utiliser les flots "cin" et "cout"
# include <math.h>           // pour les fonctions mathématiques, notamment "sin", "atan" ...
# define pi 3.14;            // déclaration de constante, en C, on utilise aussi "const x = 0"

void main()                  // programme qui compte les caractères entrés
        {int nc; for (nc = 0; getchar() != eof; ++nc); printf ("%d\n", nc);} // ou bien while (cin.get(c)) {++nc}
```

Classes et objets

Un objet est quelquechose de concret formé de plusieurs champs, ainsi un entier n'en a qu'un, un complexe en a deux, un point peut en avoir trois (abscisse, ordonnée, couleur) un article peut être (nom, date de fabrication, prix ...). Un objet appartient toujours à une classe, c'est à dire une description abstraite de ces champs et des opérations qu'on peut lui appliquer (des messages qu'il peut recevoir), une "encapsulation" dans le langage d'Ada : les champs "private" ne sont accessibles que par les méthodes définies dans la même classe.
Instancier un objet signifie dire à quelle classe il appartient et initialiser les contenus qui le caractérise. L'opération qui réalise ceci est le "constructeur", c'est toujours en C++ l'opération portant le même nom que la classe, ainsi on a une cohérence avec une déclaration ordinaire telle que "int n" consistant à construire un objet de nom n et de type int.. L'opération portant le même nom précédé du symbole ~ sera le "destructeur" libérant de la place en mémoire.
Les classes sont organisées suivant une hiérarchie, chacune hérite des structures et opérations "public" définies dans sa classe mère (héritage simple). Ces opérations sont appelées "méthodes". En revanche, la classe fille peut définir des opérations qui lui sont privilégiées. Lorsqu'une fonction doit être appliquée, c'est naturellement la première définition rencontrée en remontant dans l'arbre des classes, qui est appliquée.

On a donc la même cohérence que celle observée avec les variables locales dans la hiérarchie des sous-programme en Pascal.
Remarque : tout est "private" par défaut dans une "class" et "public" par défaut dans une "struct", afin d'être le plus clair possible nous définierons les classes avec leurs parties "private" puis "public". Exemple :

```
typedef int sesterce ;                  // déclaration d'un nouveau type de monnaie

class legionnaire {
private :       int solde;          // déclare qu'un seul champ peut caractériser un légionnaire, sa solde
public :        legionnaire (sesterce n = 0) {solde = n;}       // est un "constructeur d'objet"
                void fixesolde (sesterce n) { solde = n;}       // opération d'attribution d'une solde n
                sesterce sasolde () { return solde ; } };       // fonction donnant la solde d'un légionnaire

class centurion : public legionnaire {
// indique que les objets de cette classe héritent de tous ce qui est public dans la classe légionnaire
private :       legionnaire *groupe [100] ; };                  // l'unique champ d'un centurion en plus de "solde"
// l'unique champ caractérisant un centurion est le groupe, limité à 100, des légionnaires qu'il commande
public :        centurion () {legionnaire ();}
                legionnaire subord (int n) { return *groupe[n] ; }      // fonction donnant le subordonné n° n
                void disposer (legionnaire lg, int n) {*groupe[n] = lg;}};
                        // opération  plaçant le légionnaire lg sous les ordres d'un centurion au numéro n
# include <stdio.h>
```

```
void main () {              // petit programme d'essai
centurion Olibrius;
Olibrius.fixesolde (3000); printf ("%d\n", Olibrius.sasolde ());
legionnaire Mordicus (1000);
legionnaire Aredbus, Omnibus;
int S = 2000;
Aredbus.fixesolde (S); printf ("%d", Aredbus.sasolde ());
Mordicus.fixesolde (Aredbus.sasolde ()); printf ("%d", Mordicus.sasolde ());
centurion Autobus; Autobus.fixesolde (5000);
Autobus.disposer (Olibrius, 0); Autobus.disposer (Aredbus, 1);
printf ("%s\n", Autobus.subord (1));}
```

Remarques : l'écriture x.f(a, b, ...) qui n'est pas équivalente à f(x, a, b ...), s'énonce par "on envoie le message f à l'objet x, message se réalisant grâce aux paramètres a, b ...
L'opérateur :: de qualification de classe, permet de définir une fonction f se rapportant à la classe A (membre de A) en dehors de cette classe, on le fera alors par

A :: <type de f> f (<liste des paramètres de f>) {définition de f}

Ceci peut être utile si on veut séparer l'interface d'un module, du corps des définitions mais ce n'est recommandé que pour des logiciels où l'utilisateur veut les spécifications sans connaître les algorithmes. Nous ne le ferons pas ici.

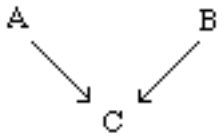
Héritage multiple

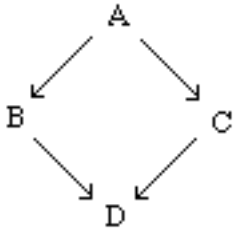
Une classe C peut hériter à la fois de plusieurs classes, donc de toutes leurs méthodes. La première question qui se pose est l'accès aux méthodes homonymes, on doit toujours préciser en ce cas la classe où est définie la méthode que l'on veut utiliser. Exemple :

```
class A {private : int valA;                    // les objets de type A ne sont caractérisés que par un entier
public :    A (int x = 0) {valA = x;}           // constructeur d'objet de type A
            int valeur () {return valA + 5;} }; // seule opération dans A calculant val + 5
class B {private : int valB;
public :    B (int x = 0) {valB = x;}
            int valeur () {return valB - 5;} };
class C : public A, public B {private : int valC;
public :    C (int x = 0) {valC = x;}
            int valeur () {return valC;} };

#include <iostream.h>
void main () {C z(3);          // :: est l'opérateur de "résolution de portée"
cout << "Valeurs : " << z.A::valeur() << z.B::valeur() << z.valeur() ; ) // donne bien sûr 5, -5 et 3
```

La seconde question est, si B et C héritent de A et que les objets de D héritent à la fois de B et de C, un mot réservé "virtual" (qui devrait être sous-entendu) permet aux objets de D de n'hériter qu'une seule fois des méthodes de A, il force le compilateur à ne compiler la fonction qu'en cas de nécessité. Les appels à ces méthodes ne sont résolus qu'à l'éxecution.





Troisième question, si A et B sont deux classes indépendantes, comment définir une fonction f(A x, B y) à valeurs dans N par exemple ? La solution consiste à créer une troisième classe C héritant de A et de B, ou alors à définir f dans B en mentionnant qu'elle est amie de A par :

friend class A f (...) ou encore dans A par friend B:: f (...)

Une fonction amie d'une ou plusieurs classes n'a plus d'argument implicite, il n'est donc plus question d'une écriture x.f(y) mais f(x, y).

Polymorphisme, fonctions amies et redéfinition d'opérateurs

La procedure "printf" est l'exemple type du polymorphisme, c'est à dire qu'elle s'applique à des objets de types différents. Un mot réservé "operator" permet de définir des fonctions de plongement d'un type vers un autre.
D'autre part, une fonction déjà définie dans une classe A peut être redéfinie dans une autre classe B comme amie dans la mesure où le type des objets de B est compatible avec celui des objets de A.
Par ailleurs, il faut parfois toute une gymnastique pour pouvoir nommer l'objet implicite, "this" est un pointeur réservé vers cet objet. Exemple, pour définir l'égalité des matrices (avec le symbole habituel ==) d'éléments de type T :

```
class matrice {private : int lgn, int col, T *elem        // sont les 3 champs privés
int operator == (matrice A) {int drap = 1; if ((lgn != A.lgn) || (col != A.col)) return 0;
        else for (int i = 0; i < lgn; i++) for (int j = 0; j < col; j++) if *this.elem[i][j] != A.elem[i][j] drap = 0;
        return drap; }
```

Il est même possible de redéfinir l'opérateur de parenthèses, toujours pour les matrices, on peut écrire :

```
        T operator () (int i, int j) {return *(elem + (i-1)*col + j - 1);}
        // en ce cas, les indices vont maintenant de 1 à lgn et de 1 à col
```

Généricité

Le mot réservé "template" qui signifie patron ou gabarit, permet de définir des classes et des fonctions abstraites portant sur un type quelconque, voire sur plusieurs, dont l'évaluation ne se fera que dans des sous-classes concrètes. Exemple de la classe abstraite des piles :

```
typedef enum {vrai = 1, faux = 0} bool;       // définition du type booléen utile partout

template <class T> class pile {       // définition de la classe pile d'éléments de type T
private :       int nb;
                T elem [100];                 // on représente une pile par une table donc par un pointeur
public :        int effectif () {return nb;}          // on définit 4 méthodes qui sont des fonctions ...
                bool vide () {return nb <= 0;}
                bool pleine () {return nb >= 100;}
                T sommet () {return elem [nb];}
                T depile () {return [--nb];}          // 2 méthodes qui modifient l'objet ...
                void empile (T x) {elem [nb++] = x;}
                pile () {nb = 0;}}                    // ... et le constructeur de pile

# include <stdio.h>
void main () {    // programme d'essai
pile <int> PE;     // on définierait de même une pile de caractères par "pile <char> PC;"
int i; printf ("Entrez des entiers non nuls, et 0 pour finir ");
do {scanf ("%d", &i); PE.empile (i);} while (i != 0);          // le zéro sera dans la pile
printf ("Le sommet est %d ", PE.sommet ());
printf ("Vous avez entré %d entiers du dernier au premier\n", PE.effectif());
do {printf ("%d ", PE.depile()); } while (! PE.vide ()); printf ("\n");}
```

Exemple de fonction abstraite, le minimum d'un tableau :
Le type T est un type quelconque pour lequel la relation < a un sens, en ce cas on peut définir une fonction "template" :

```
template <class T> T minimum (T* t, int n) // donne l'élément minimal du tableau t d'éléments de type T
        {T m = t[0], for (int i = 1; i < n; i++) if (t[i] < m) m = i; return m;}
```

Modularité

La classe permet un ensemble de descriptions cohérent, c'est une unité autonome pouvant être compilée séparément si les classes dont elle hérite le sont préalablement. Ainsi :

```
# include <unités supérieures> // est l'en-tête du module comportant la classe A
# define A_h
class A {définition des membres de la classe A}
```

Utilisation : `# include "A_h" // en-tête d'un module (une classe ou un programme utilisant A)`

ou encore :

```
# ifndef A_h
# define A_h
        class A {toute sa définition ici }
# endif
```

Passage d'une fonction en argument et fonction renvoyant une fonction

Un nom de fonction est de type pointeur (constant) sur le type du résultat de la fonction, aussi dans l'exemple suivant, on définit "fonc" comme un pointeur sur le type "double", en se servant de la déclaration :

```
# include <stdio.h>
# include <math.h>          // afin de disposer de la fonction "sin" qui est définie sur le type "double"
typedef double (*fonc) (double); // *fonc est alors nécessairement une fonction de R dans R à un seul argument

double parabole (double x) {return x*x;}
double exemple (double x) {return 4 / (1 + x*x); } // est 4 fois la dérivée de arctan
double integrale (fonc f, double a, double b, int n) // fonction à 4 arguments dont le premier est une fonction
        {double s = 0, double h = (b - a) / n ;          // s sera la somme partielle, h le pas de calcul
        for (int i = 0; i < n; i++) s += (*f) (a + i*h); // s est incrémenté de la valeur de *f
        return s * h ; }

void main ()
{printf ("1° vérification %f\n", integrale (parabole, 0, 1, 100)); // donne 1/3
printf ("2° vérification %f\n", integrale (exemple, 0, 1, 1000)); // doit donner approximativement la valeur π
printf ("3° vérification %f\n", integrale (sin, 0, 3.14, 500));} // doit donner approximativement 2
```

8-1° Définir la fonction Fibonacci récursivement.

```
int fib (int n) {return (n < 2) ? 1 : fib (n - 1) + fib (n - 2);}
```

8-2° Refaire le programme de calcul du jour de la semaine correspondant à une date donnée en utilisant le symbole de cas "?" et les opérateurs logiques pour traiter les cas avant la réforme du jeudi 4 octobre 1582 dont le lendemain fut le 15 octobre.

```
#include <stdio.h>
main () {char *jour[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vencredi", "samedi"};
        int  j, m, a, aa, ss, avancht, num;
printf ("Entrez une date jour, mois, année ");
scanf ("%d%d%d", &j, &m, &a); printf ("Le %d %d %d est un ", j, m, a);

avancht = (a < 1582) || ((a == 1582) && (m < 10)) || ((a == 1582) && (m == 10) &&(j < 5));
m += (m < 2) ? 10 : -2; a -= (m < 2) ? 1 : 0; ss = a / 100; aa = a % 100;

num = ((j + a - 2*ss + aa/4 + ss/4 + (26*m - 2) / 10) + (avancht ? 3 : 0)) % 7; printf (jour [num]); }
```

8-3° En supposant que le nombre de couples de lapins suit une loi de fibonacci (tout lapin est adulte à 2 mois et chaque couple produit un nouveau couple de lapins chaque mois, écrire un programme itératif donnant le nombre de couples en m mois et le nombre de mois nécessaire pour obtenir n couples.

```
# include <stdio.h>
int nblapins (int m) { // donne le nombre de couples de lapins en m mois
        int u = 1, v = 1, w;
        for (int i = 2; i < m; i++;) {w = u + v; u = v; v = w;}
        return w;}
int nbmois (int n) { // donne le nombre de mois pour avoir n couples
        int u = 1, v = 1; w = 2; i = 1;
        do {w = u + v; u = v; v = w; i++}while (w < n); return i;}
void main () {int nl, mois; // petit programme d'essai
printf ("Pour Fibonacci, donnez le nombre de mois "); scanf ("%d", &mois);
printf (" Cela fait %d couples", nblapins (mois));
printf ("\nCombien de couples de lapins voulez-vous ? "); scanf ("%d", &nl);
printf (" il faut %d mois pour avoir %d couples.", nbmois (nl), nl);}
```

Autre possibilité avec deux variables (attention, outre l'illisibilité, il se trouve que les différents compilateurs ne donnent pas nécessairement les mêmes résultats) :

```
int nbmois (int n) { pour avoir n couples
        int u = 1, v = 1, i = 1;
        do {v = u + (u = v); i++} while (v < n); return i+1;}
# include <iostream.h>
void main () {cout << nbmois (5) << " " << nbmois (22) << " " << nbmois (65);}
```

8-4° Tri bulle par balayage droite-gauche de toutes les sections commençantes d'un tableau.

```
#include <stdio.h>
void echange (int *a, int *b)                   // échange des deux valeurs des pointeurs a et b
        {int x = *a; *a = *b; *b = x;}          // le "return" qui ne retourne rien est ici facultatif
void trier (int* t, int n)                      // t est un pointeur sur entiers, c'est à dire un tableau d'entiers
        { for (int i = 0; i < n; i++)
                {int j = i; while ((j != 0) && (t[j-1] > t[j])) {echange (&t[j-1], &t[j]); j--;}}}
void main ()
        {int N; int T[10];
        printf ("Donnez un entier au plus égal à 10 "); scanf ("%d", &N);
        for (int i = 0; i < N; i++)
                {printf ("\ndonnez l'entier numéro %d ", i); scanf ("%d", &T[i]);}
        trier (T, N);
        printf ("\nAprès tri "); for (i = 0; i < N; i++) printf ("%d", T[i]);}
```

8-5° Créér les fonctions "être une minuscule", "être une majuscule", "être un mot" (formé de lettres) et transformer en majuscule.

```
# include <stdio.h>
int minus (char c) {return ('a' <= c) && (c <= 'z');}
int majus (char c) {return ('A' <= c) && (c <= 'Z');}
char* majuscule (char* s) // transforme les minuscules de la chaîne s en majuscules
        for (int n = 0; s[n] != '\o'; n++) if (minus (s[n])) s[n] += 'A' - 'a'; return s;}
int mot (char* s[]) {int n = 0; int d; char c;
        while (((minus (c = s[n++])) || (majus (c))) && (c != '\o')); return c == '\o';}
void main () {printf ("%d\n", majuscule ("azeRtyuIOp"));
        printf ("%d %d\n", mot ("legrand"), mot ("le grand"));}
```

8-6° Conversion de chaîne en valeur numérique

```
double conversion (char *s) // Conversion d'une chaîne s en un entier long
        {for (int i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++);
        int sgn = 1;
        if (s[i] == '+' || s[i] == '-') sgn = (s[i+1] == '+') ? 1 : -1;
        for (double v = 0; s[i] >= '0' && s[i] <= '9'; i++) v = 10*v + s[i] - '0';
        if (s[i] == '.' || s[i] == ',') i++;        // v sera la valeur, p, la puissance
        for (double p = 1; s[i] >= '0' && s[i] <= '9'; i++) {v = 10*v + s[i] - '0'; p *= 10;}
        return (sgn * v / p); }

# include <iostream.h>
void main () {cout << conversion ("12.456") + conversion ("5.544");} // essai qui donne 18
```

8-7° Définir la classe des rationnels avec comme opérateurs, le constructeur de rationnel (donnant la vraie fraction irréductible à partir de celle qui est donnée),.la somme et le produit définis comme "amis" de la somme et du produit ordinaires.

Cet exemple montre d'abord l'intérêt du constructeur d'objets permettant d'interpréter ses données.

```
#include <stdio.h>;
int pgcd (int a, int b) {return (a == b) ? a : (a < b) ? pgcd (a, b - a) : pgcd (b, a - b) ; }
        // est une petite fonction récursive qui servira à la réduction des fractions
int abs (int a) {return (a < 0) ? -a : a;}                      // valeur absolue d'un entier
int sgn (int a) {return (a < 0) ? -1 : (a == 0) ? 0 : 1;}       // signe d'un entier

class fraction {
private :       int num, denom; // sont les deux champs d'une fraction
public :        int numerateur () {return num ;}
                int denominateur () {return denom ;}
                fraction (int a = 0, int b = 1)  // par défaut de données, une fraction sera nulle
                // la fonction qui porte le nom de la classe est le constructeur d'objets de cette classe
                        {int p = pgcd (abs (a), abs (b));
                        num = (a == 0) ? sgn (a*b)*abs (a) / p : 0;
                        denom = (a == 0) ? abs (b) / p : 1; }
                operator float () // est l'opérateur de conversion vers un "float" c'est à dire un réel
                        {return (float) num / (float) denom;}
                friend fraction operator * (fraction a, fraction b)
                // définit le produit noté * comme ami du * des réels, * est surdéfini
                        {return fraction (a.num * b.num, a.denom * b.denom); }
                friend fraction operator + (fraction a, fraction b)
                // définit le + des fractions compatible avec le + usuel
                        {return fraction (a.num * b.denom + a.denom * b.num, a.denom * b.denom); }
                void affiche () // affichage spécifique aux fractions, "printf" n'étant pas redéfinissable
                        { printf ("%d / %d", num, denom) ;}
};
void main () {          // programme d'essai
fraction r0;            // définit la fraction r0 et l'initialise à 0
fraction r1 (4, 10);    // on définit r1 comme le rationnel 2/5
fraction r2 (3);        // r2 sera 3 = 3/1, l'unique argument entier est donc converti en fraction
(r1*r2 + r0).affiche (); // affiche 6/5
(fraction (5, 12) + fraction (3, 16)).affiche ();}      // affiche 29 / 48
```

Nous redéfinissons dans cet exemple les opérateurs + et *, mais pas "printf" qui n'est pas redéfinissable, on pourra par contre redéfinir l'opérateur << de sortie avec :

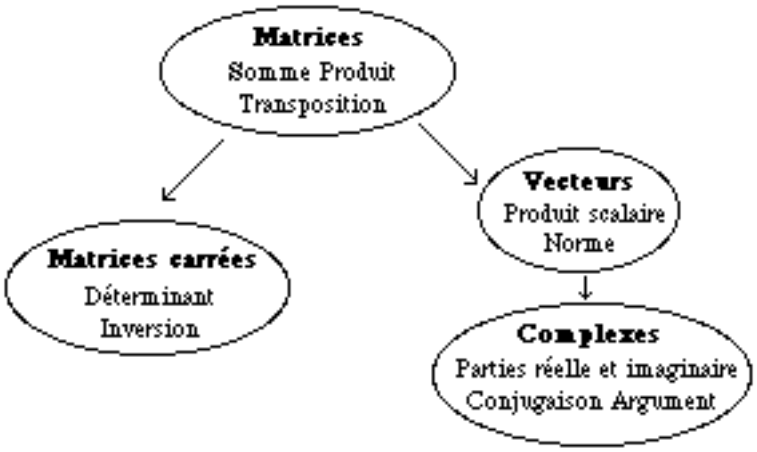
```
friend ostream & operator << (ostream & s, fraction r) {  return s << r.num << " / " << r.denom; }
```

8-8° En appelant "centurie", le groupe d'un centurion et intercalant la classe "decurion", concevoir la classe cohorte avec ses 6 centuries, et la légion avec ses 10 cohortes

8-9° Définir la classe des matrices réelles avec les opérateurs somme, produit et transposition.

```
# include <iostream.h>
# define m 20
class matrice {
private :       int nl, nc;
               float elem [m][m];
               // les 3 champs d'une matrice sont le nombre de lignes, de colonnes et le tableau des valeurs
public :       matrice (int p, int n = 1, float x = 0) // constructeur (colonne par défaut) d'éléments nuls
                    {nl = p; nc = n; elem = new float [m][m];
                    for (int i = 0; i < p; i++) for (int j = 0; j < n; j++) elem[i][j] = x; }
               ~matrice () { delete elem; }          // destructeur d'un objet matrice
               friend matrice operator + (matrice A, matrice B)
                    { matrice C (A.nl, A.nc);
                    for (int i = 0; i < A.nl; i++) for (int j = 0; j < A.nc; j++)
                         C.elem [i][j] = A.elem[i][j] + B.elem [i][j];
                    return C; }
               void vue () {for (int i = 0; i < nl; i++)
                         { for (int j = 0; j < nc; j++) cout << elem [i][j] << '\t'; cout << '\n';};}
               matrice transpo () {matrice C (nc, nl);
                    for (int i = 0; i < nc; i++) for (int j = 0; j < nl; j++) C.elem[i][j] = elem[j][i];}
               void creation () { cout << "Nombre de lignes "; cin >> nl;
                    cout << "\nNombre de colonnes "; cin >> nc;
                    for (int i = 0; i < nl; i++) { for (int j = 0; j < nc; j++)
                         { cout " élément " << i << j << '\t'; cin >> elem[i][j];} cout << '\n';}}
};        // fin provisoire de la définition de classe
void main () { matrice A (3, 2); matrice B (3, 2); A.creation (); B.creation ();
        (A + B).vue (); (A.transpo ()).vue ();}
```

8-10° A partir de la classe des matrices réelles, construire la classe des matrices carrées ayant les opérations supplémentaires déterminant et inversion, ainsi que la classe des vecteurs (colonnes) ayant les opérateurs produit scalaire et norme ami de "abs".



Remarque :
#include <assert.h> permet d'utiliser une instruction "assert" qui signale les préconditions non satisfaites, on placera :
assert ((A.nl == B.nl) && (A.nc == B.nc)); au début de la somme de deux matrices A et B, et
assert (A.nc == B.nl); au début de la définition du produit.
Indication : float argument () {float a = re ? atan (im / re) : 0;
if (im < 0 if (0 <= re) return a + pi; else return a - pi;}

8-11° Intercaler la classe des quaternions avec la conjugaison et le produit.

8-12° Définir l'affectation pour les matrices quelconques en surchargeant l'opérateur =.

8-13° Définir le tri par extraction, générique de tableaux d'éléments de type T.

Les trois fonctions ci-dessous seront valables pour tout type T où un ordre total existe (entiers, réels, caractères ...)

```
template <class T> int minindex (T* t, int n) // donne l'indice de l'élément minimal
        {int k = 0, for (int i = 1; i < n; i++) if (t[i] < t[k]) k = i; return k;}
template <class T> void echange (T & a, T & b) {T x = a; a = b; b = x;}
template <class T> void tri (T* t, int n)
        {int k; for (int i = 0; i < n; i++) {k = minindex (t + i, n - i); echange (t[i], t[k]); }}

# include <stdio.h>          // on essaie sur des entiers ou sur des caractères
void main () {char* tab[10];
        printf ("Donnez dix lettres\n");
        for (int i = 0; i < 10; i++) scanf ("%c", tab[i]);
                tri (tab, 10);
        for (i = 0; i < 10, i++) printf ("%c ", tab[i]); }
```

8-14° Définir la classe générique des ensembles avec les opérations union, intersection, différence ensembliste, appartenance, égalité ensembliste ....

```
#include <iostream.h>
template <class T> class ensemble {
private :       int card;
               T* elem;          // on choisit une représentation d'ensemble par des tableaux sans répétition
public : ensemble (int n = 0, T* tab = new T) {
        // par défaut on construira l'ensemble vide (new et delete pour créer et détruire des pointeurs
               T* aux; card = 0; elem = new T; for (int i = 0; i < n; i++)
                       {int drap; int j = i; do {drap = (tab[i] == tab[++j];} while ((!drap) && (j < n));
                       if (j == n) elem [card++] = tab [i];} }
        void voir () {cout << card << " éléments ";
               for (int i = 0; i < card; i++) cout << elem [i] << " "; cout <<"\n";}
        ensemble creer () {T *tab; int n;
               cout <<"Nombre d'éléments "; cin >> n;
               cout << "\n Donnez " << n << " éléments \n";
               for (int i = 0; i < n; i++) cin >> tab[i];
               return ensemble <class T> (n, tab);}
        int vide () {return (card == 0);}
        int app (T x) {int drap = 0; for (int i = 0; (i < card) && !drap; i++) drap = (x == elem[i]);
               return drap;}
        int inclu (ensemble A)    // dit si l'ensemble nommé par "this" est inclu dans A
               {int drap = 1; for (int i = 0; (i < card) && drap; i++) drap = (A.app(elem[i]));
               return drap;}
        int egal (ensemble A)     // B.egal(A) et A.egal(B) seront identiques
               {return inclu (A) && A.inclu (*this);}
        ensemble reunion (ensemble A) {T *tab; tab = elem; (tab + card) = elem;
               return ensemble <class T> (card + A.card, tab);}
        ensemble intersection (ensemble A)
               {int k = 0; T *tab; for (int i = 0; i < A.card; i++)
               if (app (A.elem[i])) tab[k++] = elem[i];
               return ensemble <class T> (k, tab);}
};
void main ()      // programme d'essai
{int tab[8] = {2, 3, 1, 2, 4, 2, 3, 1}; int autre[5] = {1, 2, 3, 4, 5};
ensemble <int> A (8, tab); A.voir ();
ensemble <int> B (5, autre); B.voir(); cout << A.inclu(B);}
```

8-15° Grâce à la syntaxe "template < class T, class I > class table", définir des tables d'éléments de type T indexés par des éléments de type I. Un tel objet devra avoir un indice de début et un indice de fin de type I. Tester avec I entiers ou caractères.

8-16° Définir la classe générique des listes avec les opérations liste vide, tête, queue, rajouter (en tête) et la classe dérivée des couples.

```
# include <iostream.h>
template < class T> class liste {
struct doublet {T elem; doublet *suiv;};
private : doublet *tete;
public :        liste () {tete = 0;}          // équivaut au "null" du C et au "nil" du pascal
                ~liste {doublet *p = tete;
                        while (p)             // remarquer le test "tant que p n'est pas 0"
                                {doublet *m = p; p = p->suiv; delete m;}
                        tete = 0;}            // pour rendre en totalité l'espace mémoire
                int vide () {return (tete == 0) ? 1 : 0;}
                int app (T x) {if (tete == 0) return 0;
                        else if (tete->elem == x) return 1; else return tete.suiv.app (x);}
                liste queue () {return suiv;}
                int longueur () {return vide ? 0 : 1 + queue.longueur();}
                void rajoute (T x)            // modifie l'objet en lui plaçant x en tête
                        {doublet *p = new doublet;
                        p->elem = x; p->suiv = tete; tete = p;}
                void voir () {for (doublet *p = tete; p; p = p->suiv) // remarquer le test de la boucle
                                cout << p->elem << " ";}
                void creation () { cout << "Entrer les éléments séparés par return, et return à la fin ";
                        T x; while (cin >> x) {rajoute (x);};}
};
void main {liste < char > L; L.creation(); cout << '\n' << L.vide(); L.voir();}
```

8-17° Définir la classe générique des files où les opérations possibles sont défiler (enlever le premier placé), enfiler (ajouter un nouvel élément en queue), les fonctions sont premier, suivant, précédent et file vide. (On pourra définir des triplets avec un élément de type T et deux pointeurs suivant et précédent sur des triplets, les éléments privés d'une file étant deux pointeurs de triplets, premier et dernier). On aura par exemple :
file () {premier = dernier = 0;}

8-18° Définir la classe générique des "sacs" ou multi-ensembles. Un sac d'éléments de type T doit comporter des objets de ce type avec répétition, par exemple {a, a, a, b, b, c, d, e, e, e, e} est un sac différent de {a, a, b, b, c, d, e, e, e, e} mais égal à {e, a, b, c, d, e, a, b, e, a, e, e}. Représenter par des fonctions T -> int, c'est à dire l'application "nombre d'occurrences".
Remarque, bien sûr toutes ces classes sont prédéfinies (collection, bag, set ...)

8-19° **Réseaux de Pétri,** soient P un ensemble de places, et T un ensemble de transitions, un réseau de Pétri est défini avec deux matrices Pré et Post telles que Pré (i, j) vaut 1 si la place i précède la transition j (0 sinon) et Post (i, j) vaut 1 si la place i suit la transition j (0 sinon). C = Post - Pré est la matrice d'incidence.
Une "marque" est une matrice colonne M où M(i) est un nombre entier mesurant l'occupation de la place i. La règle de franchissement d'une transition t, est que pour toute place i, M(i) ≥ Pré (i, t), en ce cas le nouveau marquage est M' = M - Pré (., t) + Post (., t).
Faire un schéma pour les transitions a : introduire la voiture, b : laver les vitres, c : assez d'huile ?, d : sortir la voiture, e : ajouter de l'huile. Faire fonctionner le réseau avec la marque initale M = (1 0 0 0 0).
Organiser des classes pour programmer le fonctionnement d'un tel réseau.

8-20° Simulation d'une fourmillière [Bonabeau 94], on considère un quadrillage représenté à l'écran et tel que les points soient disposés en quinconce. On peut prendre un carré avec une pointe en bas qui sera la fourmillière et une pointe en haut où sera plutôt concentrée la nourriture. La grille initiale comporte une distribution de nourriture qui peut être uniforme (proba 0.5 d'avoir une unité de nourriture en chaque point), soit rare (proba 0.1 d'avoir 10 unités en chaque point), soit concentrée vers le haut, soit dans les coins. Les fourmis sont concentrées vers le bas initialement. Une fourmi à la recherche de nourriture ne peut que monter (ou bien hésiter) avec une probabilité 0.5 + 0.5 th ((g + d) / 100 - 1) où g et d sont les quantités de "phéromone" respectives sur les points du haut voisins gauche et droite (nulles au départ). La simulation d'un évenement de proba p se fait par l'appartenance à [0, p] d'un tirage uniforme dans [0, 1]. Si elle monte, elle va à gauche avec une probabilité $(5 + g)^2 / [(5 + g)^2 + (5 + d)^2]$. On permet jusqu'à 20 fourmis par noeud. Toute fourmi avançant dépose une unité de "phéromone" sur le noeud qu'elle quitte (cette quantité est limitée à 300). Quand une fourmi tombe sur un point où se trouve de la nourriture, elle en prélève une unité, retourne avec les mêmes règles qu'en montant, mais en déposant 10 unités de phéromone sur chaque point visité tant que celle-ci est inférieure à 1000. On peut par la suite introduire un taux d'évaporation de la phéromone (par exemple diminution de 1/30 à chaque unité de temps).
Random se trouve dans <stdib.h>, initgrapf, lineto, moveto, setcolor, getpixel, circle dans <graphics.h>, clrscr et gotoxy dans <conio.h>

8-21° Exemple des prédateurs [Bouron 92], voir aussi [Drogoul 93]: on définit un damier par un tableau de n*n de cases où sont disposés une proie et 4 prédateurs N, S, O, E, à chaque unité de temps, chacun de ces agents mobiles fait un déplacement aléatoires sur une des 8 cases voisines, sauf s'il est informé. Concevoir une hiérarchie de classes où on aurait la classe grille, la classe position avec ses champs x, y, bordure ?, voisins : un tableau des 12 cases voisines et la méthode "déplacement aléatoire". De cette classe peut hériter la classe agent avec les méthodes E, NE, N, NO, O, SO, S, SE, puis de celle-ci, à la fois les classes proie, et predateur avec un état particulier "informe" ou "demande" : un prédateur ayant la proie dans ses 12 voisins va informer ce même voisinage, un prédateur scrutant son voisinage, peut donc être informé de la position (a, b) de la proie et va faire le déplacement possible qui minimise |x - a| + |y - b + 1| dans le cas du prédateur N par exemple. La capture est faite si les 4 prédateurs arrivent ainsi à encadrer la proie.