Table des matières

A.C inclus en C++....4
1. Un projet console en C++....4
a.Bibliothèque <iostream>....5
b.L'instruction using namespace std;....5
2.Un programme C compilé en C++....5
B.C augmenté en C++....9
1.Entrée-sortie console : cout et cin....9
a.Utiliser cout et cin....9
b.Instructions de formatage en sortie....10
2.Variables et constantes....12
a.Déclarations plus souples....12
b.Type deux valeurs : bool....12
c.Type caractère unicode : wchar_t....13
d.Typedef inutile pour les structures....13
e.Type référence (pointeur masqué)....14
f.Type pointeur, opérateurs new et delete....17
g.Constantes et pointeurs....17
h.Constantes (const) et énumération (enum) plutôt que #define....18
3.Conversions de types....18
a.static_cast<type>....19
b.const_cast<type>....19
c.reinterpret_cast<type>....19
d.dynamic_cast<type>....20
4.Fonctions....20
a.Fonctions embarquées "inline"....20
b.Retourner une référence....21
c.Surcharge des fonctions....22
d.valeurs par défaut de paramètres....23
e.Fonctions génériques (template)....24
f.Fonctions comme champs de structures....26
5.Gestion des exceptions (base)....27
a.Instruction throw....27

b.Instruction de saut try-catch....28
c.Instruction throw et appels de fonctions....29
d.instruction throw sans valeur de retour....30
e.Préciser les retours d'exceptions des fonctions....31
f.Exception non identifiée....31
g.Bloc catch(...) par défaut....32
6.Espaces de noms(namespace) et raccourcis (using)....33
7.Intégrer d'autres langages dans le code C++....36
C.La dimension objet du C++....37
1.Classe et objet....38
a.Qu'est ce qu'une classe ?....38
b.Qu'est ce qu'un objet ?....38
c.Définir une classe, déclarer un objet, droit d'accès....38
d.Un programme C muté en classe et objet....40
2.Constructeurs....44
a.Paramétrer un objet à sa déclaration....44
b.Le pointeur this....45
c.Constructeur sans paramètre....46
d.Constructeurs avec paramètres....47
e.Constructeur et copie d'objet....48
f.Constructeur avec conversion....51
3.Destructeur....52
4.Classes et membres "static"....54
a.Qualificatif static en C....54
b.Qualificatif static et objets....54
5.Surcharge des opérateurs....55
a.Fonction operator globale hors classe....56
b.Fonction operator localisée dans une classe....57
c.Fonction operator et données dynamiques....59
6.Classes génériques ("template" ou "patron")....63
a.Principe....63
b.Syntaxe de base....63
c.syntaxe constructeurs....64
d.Syntaxe avec plusieurs types génériques....64

e.Exemple d'implémentation d'une pile générique 64
f.Spécialisation de fonction sur un type donné..... 66
7.Héritage..... 67
a.Définir une classe dérivée..... 68
b.Appeler explicitement un constructeur de la classe de base 69
c.Redéfinition de données ou de fonctions..... 70
d.Spécifier un membre de la classe de base..... 71
e.Droits d'accès de la classe héritée..... 71
f.Héritage multiple..... 76
g.Héritage multiple avec une base virtuelle 79
8.Pointeurs polymorphes, virtualité..... 82
a.Accès pointeurs par défaut aux fonctions redéfinies..... 82
b.Accès pointeur aux fonctions virtuelles redéfinies..... 83
c.Intérêt des accès pointeurs aux fonctions virtuelles 85
d.Classes abstraites, fonctions virtuelles pures..... 86
9.Classes et fonctions "amies" (friend)..... 89

A. C inclus en C++

Tout ce que nous avons vu sur le langage C aux chapitres précédents est inclus dans le C++ à savoir :

- Variables simples, char, short,int long, float, double, signed, unsigned
- opérations arithmétiques +, -, *, /, %,
- opérations bit à bit &, |, ^, ~
- opérateur de comparaison <, >, <=, >=, !=, ==
- Sauts conditionnels i, if - else, if - else if - else
- Branchements switch, goto
- Tests multiconditions &&, ||
- Boucles while, do-while, for
- Fonctions retour et paramètre
- Structures struct
- Définition de types typedef
- Tableaux [ ]
- Pointeurs &, , →, [ ], malloc, free, NULL, void*

Les principes et la syntaxe en sont rigoureusement les mêmes. Un programme C peut être compilé en C++.

Pour le vérifier nous allons porter en C++ le programme de l'automate cellulaire donné au chapitre Structuration d'un programme (3.I). Le premier point est de faire un projet C++ selon l'environnement de développement (IDE) et le compilateur avec lesquels vous travaillez.

1. Un projet console en C++

Sous CodesBlocks et Mingw32 sélectionnez un projet console. En standard nous obtenons le main suivant :

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

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Pas de différence fondamentale dans le structure du main(). Juste des variations au niveau des inclusions (#include) et la curieuse fonction d'affichage cout<< <<endl qui remplace le printf() habituel en C (son utilisation est présentée plus loin). L'extension du fichier source passe à point cpp au lieu de point c.

Sous Visual C++ de Microsoft il y a deux possibilités pour faire un projet console en C++. La première consiste à sélectionner un projet console dans la fenêtre "nouveau projet". Cette solution fait apparaître un projet console avec du C++ non standard et spécifique à Microsoft. Pour rester dans le C++ standart qui est notre objet d'étude ici il faut sélectionner un "projet vide", créer une page de code cpp et y copier le code ci-dessus.

Si vous compiler le projet vous obtenez une fenêtre console avec "Hello world !" en haut à gauche.

a. Bibliothèque <iostream>

Le fichier d'entête <iostream> est équivalent à <iostream.h>. Le point h a disparu avec la norme ISO/IEC 14882-1998 mais sur plupart des compilateurs les deux écritures fonctionnent encore. Nous utiliserons ici la version sans point h plus récente.

Iostream est une des nombreuses bibliothèques standart du C++. Elle permet d'utiliser cout<< << et cin >> >> qui en réalité sont des objets.

Une bibliothèque C++ contient en effet non seulement des fonctions et des définitions de types comme en C mais aussi et surtout des classes. Les classes sont spécifiques à la programmation objet. La dimension objet est l'apport important du C++. Nous l'abordons dans la dernière partie de ce chapitre.

b. L'instruction using namespace std;

Un espace de noms (namespace) permet de regrouper sous un seul nom un ensemble de fichiers d'entêtes (librairies) ou d'éléments de code source qui par ailleurs peuvent être répartis sur de nombreux fichiers. Nous expérimentons plus loin comment créer ses propres espaces de nom

Le mot clé using associé à un espace de nom permet un raccourci pour l'accès aux données contenues dans cet ensemble.

L'espace de nom std regroupe 50 fichiers d'entête constitués de 18 fichiers d'entête de la bibliothèque C standard et de 32 autres, appelés en-tête STL (Standard Template Library). Vous pouvez trouver de la documentation à ce sujet en ligne par exemple sur http://www.cplusplus.com.

L'instruction "using namespace std;" permet ainsi d'inclure ces 50 librairies afin de pouvoir les utiliser dans le projet. Toutes ces librairies sont visibles dans le dossier "dépendances externes" d'un projet visual C++.

C'est la norme ISO/IEC 14882-1998 qui spécifie que les fichiers C commencent par la lettre c. Ainsi stdio.h devient cstdio.h, et pour les 32 fichier C++ ils perdent l'extension .h, iostream.h devient iostream.

2. Un programme C compilé en C++

Pour tester du C compilé en C++ voici le programme de l'automate cellulaire étudié précédemment. Nous avons juste modifié les matrices. De statiques elles sont maintenant dynamiques ce qui permettra de vérifier le fonctionnement de l'allocation

dynamique et des pointeurs. De plus nous avons ajouté à l'automate la possibilité de changer de couleur avec le temps.

Chaque position conserve en mémoire le nombre de fois où elle a été activée (passage à 1) et c'est maintenant ce nombre qui donne la couleur de la position. De ce fait il faut deux variables pour chaque position, une comme précédemment qui indique l'état de la position et l'autre qui indique le nombre de fois où elle est passée à 1 et donne sa couleur. Nous avons regroupé ces deux informations dans une structure qui contient deux entiers et les matrices sont maintenant des matrices de structures.

Pour compiler en C++ ce programme C il suffit de le copier coller ce code dans le projet C++ en concervant les entêtes C++, ce qui donne :

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

#include <windows.h>
#include <conio.h>

typedef struct pos{
    int val;          // 0 ou 1
    int color;
}t_pos;

int TX;
int TY;

t_pos**MAT;           //matrices dynamiques de structure
t_pos**SEV;

void    init_matrice        (int tx, int ty);
void    destroy_mat         (void);
int     compte_voisins      (int y, int x);
void    calcul              (void);
void    copie               (void);
void    affiche             (void);
void    gotoxy              (int x, int y);
void    textcolor           (int color);
/***************************************/
int main(int argc, char *argv[])
{
int fin=0;

  printf("Action  : appuyer sur n'importe quelle touche\n"
         "Quitter : q");
  init_matrice(80,23);
  while(fin!='q'){

     if(kbhit()){
        fin=getch();
        affiche();
        calcul();
        copie();
     }
  }
  destroy_mat();

  gotoxy(1,TY+1);
  textcolor(16);
```

```
  return 0;
}
/**************************************/
void init_matrice(int tx, int ty)
{
int i ;
    TX=tx;
    TY=ty;
    MAT=(t_pos**)malloc(sizeof(t_pos*)*TY);
    SEV=(t_pos**)malloc(sizeof(t_pos*)*TY);
    for (i=0; i<TY;i++){
        MAT[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        SEV[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        memset(MAT[i],0,sizeof(t_pos)*TX);
        memset(SEV[i],0,sizeof(t_pos)*TX);
    }
   MAT[TY/2][TX/2].val=1;
   MAT[TY/2+1][TX/2].val=1;
   MAT[TY/2][TX/2+1].val=1;
   MAT[TY/2+1][TX/2+1].val=1;

   MAT[TY/2][TX/2].color=1;
   MAT[TY/2+1][TX/2].color=1;
   MAT[TY/2][TX/2+1].color=1;
   MAT[TY/2+1][TX/2+1].color=1;
}
/**************************************/
void destroy_mat()
{
int i;
    for (i=0; i<TY; i++){
        free(MAT[i]);
        free(SEV[i]);
    }
    free(MAT);
    free(SEV);
}
/**************************************/
void calcul()
{
int x,y,nb_voisins;

   for (y=1; y<TY-1; y++){
      for (x=1 ;x<TX-1 ;x++){
         nb_voisins=compte_voisins(y,x);
         if (nb_voisins <2 || nb_voisins>3){
            SEV[y][x].val=0;
            SEV[y][x].color = MAT[y][x].color;
         }
         else{
            SEV[y][x].val=1;
            if (MAT[y][x].color<15)
                SEV[y][x].color = MAT[y][x].color+1;
         }
      }
   }
}
/**************************************/
int compte_voisins(int y, int x)
{
```

```
int nb=0;

   if (MAT[y][x+1].val==1)
     nb++;
   if (MAT[y-1][x+1].val==1)
     nb++;
   if (MAT[y-1][x].val==1)
     nb++;
   if (MAT[y-1][x-1].val==1)
     nb++;
   if (MAT[y][x-1].val==1)
     nb++;
   if (MAT[y+1][x-1].val==1)
     nb++;
   if (MAT[y+1][x].val==1)
     nb++;
   if (MAT[y+1][x+1].val==1)
     nb++;

   return nb;
}
/****************************************/
void copie()
{
int i,j ;
   for (j=0; j<TY; j++){
      for (i=0 ;i<TX ;i++){
         MAT[j][i]= SEV[j][i];
      }
   }
}
/****************************************/
void affiche()
{
int x,y ;

   for (y=0; y<TY; y++){
      for (x=0 ;x<TX ;x++){
         gotoxy(x+1,y+1);
         textcolor(MAT[y][x].color*16);
         putchar(' ');
      }
   }
}
/****************************************/
void gotoxy(int x, int y)
{
  COORD c;

  c.X = x;
  c.Y = y;
  SetConsoleCursorPosition (GetStdHandle(STD_OUTPUT_HANDLE), c);
}
/****************************************/
void textcolor (int color)
{
    SetConsoleTextAttribute (GetStdHandle(STD_OUTPUT_HANDLE),
color);
}
/****************************************/
```

Sous CodeBlocks avec Mingw32 le programme compile sans aucun problème ni warming et il fonctionne très bien.

Sous Visual C++ il y a deux warmings à propos des fonction kbhit() et getch() qui sont considérées comme "deprecated". Pour éviter ces warmings vous devez juste rebaptiser kbhit() en _kbhit et getch() en _getch().

Clairement ce programme permet de constater que le C est inclus dans le C++ et qu'il est toujours possible de faire du C dans un environnement C++. Cependant l'inverse n'est pas vrai, un programme C++ peut ne pas compiler en C parce que le C++ élargit les possibilités du C avec des nouveautés de syntaxe et d'écriture.

Nous distinguons ici deux niveaux pour les apports du C++ : 1) Ce qui facilite l'écriture sans rien toucher de fondamental du point de vue de la conception et 2) la dimension objet qui modifie de façon importante la conception. Passer à l'objet c'est un peu comme passer de la 2D à la 3D dans le domaine graphique.

B. C augmenté en C++

Nous allons explorer ici des apports du C++ qui modifient l'écriture du C mais sans modifier la conception de programme, c'est à dire sans les classes et les objets.

1. Entrée-sortie console : cout et cin

a. Utiliser cout et cin

Les fonctions printf() et scanf(), même si elles sont toujours utilisables, sont avantageusement remplacées par les objets cout et cin.

Avec cout nous pouvons écrire dans la fenêtre console, par exemple :

```
cout<<"Salut !;
```

écrit "Salut !" dans la fenêtre console. L'opérateur << utilisé indique une sortie (il ne s'agit pas dans ce contexte d'un opérateur bit à bit de décalage)

Avec cin il est possible d'affecter à une variable une valeur tapée au clavier :

```
int i;
cin>>i;
```

Dans la fenêtre console, le curseur en écriture clignote et attend que l'utilisateur entre une valeur et tape entrée (touche enter ou return). Si l'utilisateur entre une valeur non conforme au type de la variable, la variable n'est pas touchée. Cette fois c'est l'opérateur >> qui est utilisé (ce n'est pas un opérateur bit à bit). A noter également l'absence de l'opérateur adresse de & pour la variable, à la différence de scanf().

Mais l'intérêt de cout et cin c'est aussi la disparition des formats (%d, %f, %c etc.). Il n'est plus nécessaire de se soucier du type de ses variables ni de la chaine formatée. Par exemple :

```
#include <iostream>    // à ne pas oublier pour avoir cout et cin
using namespace std;

int main(int argc, char *argv[])
```

```
{
int i=5;
float f;
char s[80];
    cout<<"entrer une valeur entiere :"<<endl;
    cin>>i;

    cout<<"entrer une valeur flottante :"<<endl;
    cin>>f;

    cout<<"entrer un nom :"<<endl;
    cin>>s;

    cout<<"vous avez entré :"<<endl;
    cout<<"entier "<<i<<'\n'<<"float "<<f<<endl<<"nom "<<s<<endl;

    return 0;
}
```

L'utilisateur entre une valeur entière, flottante, et une chaine de caractère. Les valeurs entrées par l'utilisateur sont affichées ensuite :

```
cout<<"entier "<<i<<'\n'<<"float "<<f<<endl<<"nom "<<s<<endl;
```

Cette ligne montre comment imbriquer les différents éléments de l'affichage simplement en les séparant chacun avec l'opérateur <<.

endl vaut '\n', et marque une fin de ligne, c'est un retour chariot (passage à la ligne).

b. Instructions de formatage en sortie

La bibliothèque <iomanip> offre des possibilités de formatage du texte en sortie qui le cas échéant peuvent être utiles. Pour y avoir accès ne pas oublier d'inclure <iomanip>

Largeur minimum de l'affichage

Utilisez la fonction void setw(int n);

Cette fonction spécifie la largeur minimum de l'affichage en caractères (w pour width). Le nombre de caractères est donné en paramètre. Le programme suivant affiche une série de nombre aléatoires compris entre 0 et 99 sur 5 lignes et 10 colonnes correctement alignées :

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

int main()
{
    for(int j=0; j<5; j++){
        for (int i=0; i<10; i++)
            cout<<setw(3)<<rand()%100;
        cout<<endl;
    }
    return 0;
}
```

Alignement des sorties

Par défaut l'alignement se fait sur la droite. Pour un alignement des caractères à gauche utilisez l'appel : setiosflags(ios::left);

Pour revenir à un alignement des caractères à droite utilisez l'appel : setiosflags(ios::right);

Exemple : 5 lignes de 10 nombres aléatoires compris entre 0 et 99 affichés alignés à gauche et en colonnes de 4 caractères :

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

int main()
{
    for(int j=0; j<5; j++){
        for (int i=0; i<10; i++){
            cout<<setw(4)<<setiosflags(ios::left);
            cout<<rand()%100;
        }
        cout<<endl;
    }
    return 0;
}
```

Choisir un caractère de remplissage

Eventuellement il peut être utile de remplacer les espaces par un caractères spécifique. Dans ce cas utilisez la fonction void setfill(int c) en spécifiant en c le caractères souhaité, le point dans l'exemple ci-dessous :

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

int main()
{
    for(int j=0; j<5; j++){
        for (int i=0; i<10; i++){
            cout<<setw(4)<<setiosflags(ios::left)<<setfill('.');
            cout<<rand()%100;
        }
        cout<<endl;
    }
    return 0;
}
```

Afficher ou non les zéros après la virgule

Pour ne pas afficher les zéros après la virgule utilisez la fonction setiosflags() avec en paramètre la valeur ios::showbase.

Pour avoir les zéros après la virgule passer à setiosflags() la valeur ios::showpoint

Le programme suivant affiche une suite de 10 float initialisés avec des valeurs entières aléatoires entre 0 et 9 :

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

```
int main()
{
    //cout<<setiosflags(ios::showpoint); // 0 après la virgule
    cout<<setiosflags(ios::showbase);    // pas de 0 après la
                                         // virgule
    for (int i=0; i<10; i++){
        float f=rand()%10;
        cout<<f<<endl;
    }
    return 0;
}
```

Afficher le signe des nombres positifs

Pour afficher le signe d'un nombre positif utilisez la fonction setiosflags() avec en paramètre la valeur ios::showpos.

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

int main()
{
    cout<<setiosflags(ios::showpos);
    for (int i=0; i<10; i++){
        cout<<rand()%100<<endl;
    }
    return 0;
}
```

2. Variables et constantes

a. Déclarations plus souples

Des facilités sont accordées pour les déclarations de variables. Les variables ne sont plus obligatoirement toutes déclarées en début de bloc mais n'importe où dans un bloc. C'est particulièrement utile dans une boucle for pour la variable compteur. La ligne :

```
for (int i=0; i<10; i++){ ... }
```

compile en C++ mais pas en C.

b. Type deux valeurs : bool

En C++ nous avons le type bool. Une variable de ce type n'admet que deux valeurs possibles 0 ou 1, false ou true :

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

int main()
{
bool b;
    for (int i=0; i<10; i++){
        b=rand()%10;
        cout<<b<<endl;
    }
```

```
    return 0;
}
```

Dans le programme ci-dessus rand()%10 retourne une valeur de 0 à 9 mais le programme imprime 0 si b vaut 0 et 1 pour toutes les autres valeurs.

c. Type caractère unicode : wchar_t

Avec l'internationalisation de l'informatique s'est posé le problème du codage des caractères des nombreuses langues écrites de l'humanité. Au départ sur 8 bits il n'y avait que 256 possibilités de lettres alors nous sommes passés au codage unicode en 16 bits voire en 32 bits soit un minimum de 65535 possibilités de lettres. Tous les alphabets et signes connus des écritures de l'humanité sont maintenant codés y compris ceux des langues anciennes comme l'égyptien hiéroglyphique ou le hittite cunéiforme.

En C le type char originel est sur 8 bits. Le type wchar_t pour "wide character" (caractère large) a été ajouté ensuite ainsi qu'une librairie associée, la librairie <wchar.h>. Le type wchar_t est définit dans la librairie <stddef.h>.

En C++ le type wchar_t est intégré. Toutefois dans la pratique, à la différence du C , ce sont des objets string avec l'inclusion de <string> ou des objets wstring avec l'inclusion de <wsring> qui sont utilisés (pas de bidouillage avec des tableaux de wchar_t).

Le programme suivant affiche la taille en octets d'un caractère wchar_t et sa valeur Ensuite une chaine wchar_t* est parcourue en char* c'est à dire octet par octet :

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

int main()
{
wchar_t c = 'a';
    cout<<sizeof(c)<<" : "<<c<<endl;

wchar_t s[] = L"012345678";
char*ptr = (char*)s;

    for (int i=0;i<18; i++) // 9 caractères 16 bits soit 18 octets
        cout<<*(ptr+i);
    cout<<endl;
    return 0;
}
```

Le préfixe L est utilisé pour spécifier qu'il s'agit d'une chaine en 16 bits. Ce préfixe peut aussi être utilisé pour notifier qu'un caractère est en 16 bits. Par exemple l'expression L'A' explicite que 'A' est un caractère 16 bits.

d. Typedef inutile pour les structures

En général en C il convient de supprimer le mot clé struct de la définition d'une structure en utilisant un typedef :

```
typedef struct test{
    …
}t_est;
```

```
int main()
{
t_test t0;
    ...
}
```

En C++ plus besoin de cette opération nous pouvons utiliser directement le nom de la struct comme nom de type :

```
struct test{      // plus de typedef ici
    ...
}t_est;
int main()
{
t_test t0;
    ...
}
```

e. Type référence (pointeur masqué)

Une variable de type référence se déclare avec l'opérateur & adresse de :

```
int &r
```

elle doit contenir l'adresse d'une autre variable et son initialisation est obligatoire à la déclaration :

```
int a;
int &r = a;
```

Une fois que la variable référence contient une adresse mémoire de variable elle peut être utilisée à sa place comme un synonyme :

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

int main()
{
int i;
int &r = i;
    r=50;
    cout<<r<<" : "<<i<<endl;
    return 0;
}
```

Dans cet exemple la référence r prend comme valeur l'adresse de l'entier i ensuite une valeur est affectée directement à l'adresse de i c'est à dire dans i, via la référence r, r et i désigne le même espace mémoire. Le programme affiche :

50 : 50

Avec un pointeur classique cela s'écrit :

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

int main()
{
int i;
int *r = &i;
    *r = 50;
    cout<< *r <<" : "<< i <<endl;
    return 0;
```

```
}
```

Nous voyons que le type référence permet de supprimer l'étoile pour l'utilisation d'un pointeur.

Bien entendu cela fonctionne avec n'importe quel type de variable, avec une structure par exemple nous avons :

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

int main()
{
struct test{
int i;
float f;
}t,&tr=t;

    tr.i=rand()%10;
    tr.f=rand()%10000/100.0;
    cout<<tr.i<<" : "<<t.i<<endl;
    cout<<tr.f<<" : "<<t.f<<endl;
    return 0;
}
```

t est une struct test et tr est une référence à t : tr et t peuvent s'employer indifféremment. Notons que l'accès aux champs de la structure t via la référence tr se fait avec l'opérateur point (et non la flèche des pointeurs)

L'intérêt est surtout en paramètre de fonction, le type référence permet de transformer les entrées en sortie sans alourdir le code par des opérateurs dont l'oubli est source de bugs parfois difficiles à retrouver.

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

void modif(int&val)
{
    val*=10;
}
int main()
{
int v=3;
    modif(v);
    cout<<v<<endl;
    return 0;
}
```

Le programme affiche 30. C'est l'adresse mémoire de la variable v qui est passée au paramètre.

Voici un autre exemple avec une structure :

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

struct t_test{
    int i;
    float f;
};
```

```
void init(t_test&t)
{
    t.i=rand()%10;
    t.f=rand()%10000/100.0;
}

int main()
{
t_test tv;
    srand(time(NULL));
    init(tv);
    cout<<tv.i<<endl;
    cout<<tv.f<<endl;
    return 0;
}
```

La structure tv est bien initialisée dans le main après appel de la fonction init().

En dehors des paramètres de fonctions les références sont aussi utilisées comme allias ou raccourcis sur des objets.

Une référence est constante

Une référence reste constante et il n'est pas possible de modifier sa valeur (l'adresse de variable contenue) une fois la première affectation effectuée :

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

int main()
{
int a,b= 15 ;
int&r=a;
    r=50;
    r=b;        // qu'est ce qui se passe ici ?
    cout<<a<<" : "<<b<<endl;
    return 0;
}
```

Ce programme affiche : 15 : 15

L'instruction r=b affecte la valeur de b à la variable a via son synonyme r mais l'adresse contenue par r n'est pas modifiée.

Connaitre la valeur de la référence

Pour avoir l'adresse contenue par le variable référence il faut utiliser l'opérateur & :

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

int main()
{
int i;
int&r=i;
    printf("%p-%p\n",&i,&r);
    return 0;
}
```

Dans ce programme l'expression &r vaut &i et désigne l'adresse de la variable i.

f. Type pointeur, opérateurs new et delete

En C++ la fonction malloc() est remplacé par l'opérateur new et la fonction free() par l'opérateur delete. En effet new et delete ne sont plus des fonctions mais des opérateurs mots-clé du langage. Ils s'utilisent pour l'allocation dynamique de la façon suivante, soit TYPE n'importe quel type, l'instruction :

```
TYPE* var = new TYPE;
```

alloue un objet de type TYPE. Dans l'autre sens pour libérer la mémoire c'est l'instruction :

```
delete var ;
```

Pour obtenir un tableau dynamique, l'instruction :

```
TYPE*tab = new TYPE[nb];
```

avec nb une valeur entière, alloue un tableau de nb objets de type TYPE. L'effacement se fait avec :

```
delete[]tab;
```

Par exemple :

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

int main()
{
float *v;
    // allocation d'une variable simple
    v=new float;
    *v=1.5;
    cout<<*v<<endl;
   // liberation v
   delete v;

    // allocation dynamique d'un tableau de 10 int
   int*tab=new int[10];

    for (int i=0; i<10; i++){
        tab[i]=rand()%100;  // initialisation
        cout<<tab[i]<<endl; // affichage
    }
    // libération tab
    delete tab;

    return 0;
}
```

g. Constantes et pointeurs

En cas d'utilisations de constantes pointeurs il convient de distinguer trois possibilités.

1) Est constant l'objet pointé mais pas le pointeur qui peut prendre une autre valeur. La syntaxe est la suivante, par exemple avec un pointeur chaine de caractères :

```
const char *p="test ptr et constantes";

    p[2]='W';           // ne marche pas
    p = new char[10];   // fonctionne
```

2) Est constant le pointeur mais pas l'objet pointé :
```
// le pointeur doit ici faire l'objet d'une allocation et la
// chaine doit être recopiée (voir remarque plus bas)
char*tmp = "test ptr et constantes";
char *const p = new char[strlen(tmp)+1];
    strcpy(p, tmp);

    p[2] = 'W';         // fonctionne
    p = new char[10];   // ne marche pas
```

3) Sont constants le pointeur et l'objet pointé. Ni l'un ni l'autre ne peuvent changer de valeur :
```
const char *const p = "test ptr et constantes";

    p[2] = 'W';         // ne marche pas
    p = new char[10];   // ne marche pas
```

Remarque :

Une chaine en dur comme "test ptr et constantes" est une constante. L'affectation
```
char*tmp = "test ptr et constantes";
```
ne provoque pas d'erreur à la compilation et tmp n'est pas considérée comme constante mais l'adresse contenue par tmp est bien constante et à l'exécution toute tentative d'accès provoque un "acces violation" et une sortie du programme. C'est pourquoi pour pouvoir tester la cas 2 il faut une allocation du pointeur p et une recopie de la chaine à l'adresse de p.

Par ailleurs il faut faire attention à ces "chaines données en dur" dans un programme. Notamment ne pas oublier que la mémoire correspondante est libérée à l'issue du bloc dans lequel elles sont déclarées.

h. Constantes (const) et énumération (enum) plutôt que #define

Quoique les #define fonctionnent en C++ ils est généralement recommandé de les remplacer lorsque c'est possible par des constantes ou par des enumérations. Par exemple :
```
#define TX 40
#define TY 20
```
est remplacé par :
```
const int TX = 40;
const int TY = 20;
```
La liste
```
#define NORD     0
#define EST      1
#define SUD      2
#define OUEST    3
```
est remplacée par :
```
enum direction{NORD,EST,SUD,OUEST};
```

3. Conversions de types

L'opérateur de conversion (cast) du C est conservé. Mais pour des raisons de sécurité du code la conversion explicite entre types différents est organisée en quatre parties chacune disposant d'un opérateur spécifique. Ce sont les opérateurs static_cast<type>, const_cast<type>, reinterpret_cast<type> et dynamic_cast<type>. Ces opérateurs plus facile a repérer dans le code et indique un type de conversion.

a. static_cast<type>

Cet opérateur de cast est dit static parce que la conversion de type est réalisée à la compilation et non pendant le fonctionnement du programme. Il est très proche de l'opérateur cast du C. Il gère des conversions entre variables de même famille (entre char, short, int,long, float, double par exemple). Quoique la plupart du temps ces conversions sont faites implicitement. Pour l'utiliser il faut remplacer type par le type voulu et mettre l'expression à convertir à sa droite entre parenthèse :

```
int a;
    static_cast<float>(a); // vaut float
```

avec ce cast la valeur de a est considérée comme float.

Le programme suivant permet d'obtenir un nombre aléatoire compris entre 0 et 1 :

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

int main()
{
float f;
    f= static_cast<float>(rand())/RAND_MAX;
    cout<<f<<endl;
    return 0;
}
```

Rappelons que rand() retourne un entier et RAND_MAX, la valeur maximum est également entière. Sans cast nous aurions pratiquement toujours 0.

b. const_cast<type>

Cet opérateur de conversion gère les conversions lorsque dans une situation il est nécessaire de faire disparaître la qualification const d'un membre. Par exemple :

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

int main()
{
const char*msg="chaine constante";
char*ptr = const_cast<char*>(msg);
    cout<<ptr<<endl;
    return 0;
}
```

c. reinterpret_cast<type>

Cet opérateur de conversion permet des conversion entre variables de famille différentes par exemple de int vers pointeur ou entre pointeurs quelconques, souvent des opérations risquées. Par exemple

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

int main()
{
int*i= reinterpret_cast<int*>(0xFF);
    cout<<i<<endl;
    return 0;
}
```

Ce programme déclare un pointeur d'entier et lui affecte une adresse mémoire sans vérification de sa validité.

Toutefois cet opérateur peut être utilisé dans des opérations de bas niveau. Examinons par exemple les octets d'une variable de type double en la transformant en tableau d'octets :

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

int main()
{
int f=0xDDCCBBAA;
unsigned char*ptr=reinterpret_cast<unsigned char*>(&f);
    for (int i=0; i<sizeof(f);i++){
        printf("%d-",ptr[i]);
    }
    return 0;
}
```

Le programme imprime 170-187-204-221-

d. dynamic_cast<type>

A la différence des trois autres opérateurs de conversion dynamic_cast utilise les informations de type d'un objet pendant le fonctionnement du programme et non avec la compilation. Son utilisation intéresse les classes et les objets. Elle est requise en particulier pour convertir un pointeur ou une référence de classe en un pointeur ou une référence de classe dérivée. Ou encore pour convertir un objet d'héritage multiple en un objet d'une de ses bases.

4. Fonctions

Quelques différences par rapport au C pour les fonctions en C++.

a. Fonctions embarquées "inline"

Ce sont des petites fonctions optimisées pour la vitesse par le compilateur lors de la compilation. C'est équivalent aux fonctions obtenues avec #define en C. En général cette appellation est réservée à de petites fonctions qui doivent s'exécuter

particulièrement promptement. Pour écrire une telle fonction utiliser le mot clé inline :

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

inline int cmpt()
{
static int c=0;
    return c++;
}

int main()
{
    for (int i=0; i<50;i++)
    cout<<cmpt()<<endl;
    return 0;
}
```

b. Retourner une référence

Une fonction peut retourner une référence ce qui donne un piège à éviter et la possibilité par exemple de masquer un tableau par une fonction.

Piège à éviter

Une fonction peut retourner une référence mais il faut faire attention à ce que l'adresse de la référence reste valide après l'exécution de la fonction. C'est très facile de faire une erreur, qui plus est, ne sera pas forcément détectée rapidement y compris à l'exécution. C'est le genre d'erreur sinistre qui provoque un plantage de temps en temps sans qu'il soit facile de la repèrer. Par exemple :

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

int& ref()
{
    int i =0;
    printf("%p\n",&i);
    return i;          // ERREUR TRES DANGEREUSE
}

int main()
{
int&r=ref();
    printf("%p\n",&r);
    r=100;             // &i de ref() n'est plus réservée
    cout<<r<<endl;

    return 0;
}
```

Ce programme fonctionne sur ma machine sous codeBlocks et sous Visual C++. La seule indication que donne le compilateur est un "warming : reference to local variable i returned". C'est à prendre très au sérieux.

En effet un espace mémoire est réservé pour la variable i uniquement pour la durée du bloc dans lequel elle est déclarée. Ensuite cet espace mémoire n'est plus

réservé. Tant que cette adresse mémoire n'est pas réallouée. A priori rien ne se passe et le programme fonctionne. Mais si cette adresse est réattribuée alors le programme va planter lorsque la référence y accédera.

Masquer un tableau par une fonction

Par ailleurs le retour d'une référence peut permettre de masquer un tableau par une fonction :

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

float tab[]={1.1,2.2,3.3,4.4,5.5};

void affiche()
{
    for (int i=0; i<sizeof(tab)/sizeof(tab[0]); i++)
        cout<<setw(4)<<tab[i];
    cout<<endl;
}

float& ref(int i)
{
    if (i>=0 && i < sizeof(tab)/sizeof(tab[0]))
        return tab[i];
    else
        return tab[0];
}

int main()
{
    affiche();
    ref(1)=10;          // affecte 10 à tab[1]
    ref(2)=20;          // affecte 20 à tab[2]
    ref(3)=30;          // affecte 30 à tab[3]
    affiche();

    ref(0)= ref(4);     // affecte tab[4] à tab[0]
    affiche();
    return 0;
}
```

Dans ce programme le tableau de float est global. La fonction affiche permet d'en afficher le contenu. La fonction ref retourne la référence de l'élément dont l'indice est spécifié en paramètre. Ce n'est pas la valeur de tab[i] qui est retournée mais la référence de retour de la fonction est initialisée avec l'adresse de l'élément tab[i] et de ce fait vaut pour tab[i]. C'est pourquoi il est possible d'écrire ensuite : ref(0)=10; par exemple.

De même une instruction comme ref(2)=ref(3); signifie que la valeur de tab[3] via sa référence en retour de l'appel ref(3) donne sa valeur à tab[2] via sa référence retournée par l'appel ref(2).

c. Surcharge des fonctions

Un point important du C++ par rapport au C est la possibilité de différencier les fonctions par le type et le nombre des paramètres et non pas par leurs noms. Des

fonctions avec des paramètres différents peuvent avoir le même nom. La valeur de retour ne compte pas dans la différenciation. C'est le principe de la surcharge : la fonction est reconnue au moment de l'appel selon les valeurs qui lui sont communiquées.

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

int test(int a)
{
    cout<<"test(int a)"<<endl;
    return a++;
}
/* Provoque une erreur à la compilation
void test(int a)
{
    cout<<"test(int a)"<<endl;
}
*/
int test (double a)
{
    cout<<"test(double a)"<<endl;
    return static_cast<int>(++a);
}
void test(char c, int i)
{
    cout<<"test(char c, int i)"<<endl;
}

int main()
{
    test(10);
    test(15.15);
    test('A',20);
    return 0;
}
```

Ce programme imprime :

```
test(int a)
test(double a)
test(char c, int i)
```

d. valeurs par défaut de paramètres

Le C++ offre la possibilité de donner des valeurs par défaut à des paramètres de fonction. Les valeurs par défaut sont affectées aux paramètres concernés avec la déclaration de la fonction :

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

// déclaration de la fonction
float prixTTC(float m,float tva=18.6);   // 18.6 par defaut

int main()
{
float p,tva;
   cout<<"entrez un prix HT"<<endl;
```

```
   cin>>p;

   // 1) avec tva par défaut :
   cout<<"prix TTC: "<<prixTTC(p)<<" (tva : 18.6)"<<endl;

   // 2) avec une nouvelle tva :
   cout<<"entrer un prix et le montant de la tva :"<<endl;
   cin>>p>>tva;
   cout<<"prix TTC: "<<prixTTC(p,tva)<<" (tva : "<<tva<<")"<<endl;

   system("PAUSE");
   return 0;
}

// définition de la fonction
float prixTTC(float prixHT,float tva)
{
   if(tva>0)
          prixHT += (prixHT*tva)/100;
   return prixHT;

}
```

Le montant par défaut du paramètre tva est spécifié à la déclaration de la fonction. Si la fonction est placée avant le main() il est alors spécifié directement à la définition de la fonction (qui fait alors office de déclaration)

L'utilisation est simple : 1) sans précision de la tva c'est la valeur par défaut qui est prise, 2) si vous précisez une valeur elle écrase la valeur par défaut.

Remarque :

Attention, les paramètres par défaut doivent être placés après les paramètres qui n'ont pas de valeur par défaut sinon des ambiguïtés se produisent, indécidables pour le compilateur :

```
void fonct(int a=1, int b, int c=3);  // produit une erreur
```

Comment résoudre en effet l'appel :

```
fonct (12,15)  // ?
```

Est ce a ou b qui prend la valeur 12 ? b ou c qui prend la valeur 15 ?

Pour éviter ces erreurs ces ambiguïtés doivent être levées et soit par exemple la fonction :

```
void fonct (int a, int b=2, int c=3);
```

nous pouvons avoir des appels comme :

```
fonct(10);       // a:10, b:2, c:3
fonct (10,20);          // a:10, b:20, c:3
fonct(10,20,30);        // a:10, b:20, c:30
```

e. Fonctions génériques (template)

Le C++ offre aux fonctions de disposer d'un type générique pour des paramètre. C'est à dire offre la possibilité d'avoir une fonction qui marche avec des char ou des int ou des float ou des structures etc. La fonction s'adapte au type générique pour un ou plusieurs paramètres.

La définition d'une telle fonction se fait en précédent la fonction de l'expression : template <class nomType> ou nomType est le nom que vous donnez à votre type générique. Par exemple :

```
template <class T> T fonct(T a, T b)
{
    return (a<b) ? a : b;
}
```

Le mot clé class peut être remplacé par typename et c'est me semble t-il plus clair dans ce contexte, ce qui donne :

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

template <typename T> T fonct(T a, T b)
{
    return (a<b) ? a : b;
}
```

C'est la notation que nous utiliserons maintenant.

Cette fonction retourne la valeur minimum sur les deux passées en paramètre. T correspond au type générique. Le type effectif est celui spécifié au moment de l'appel avec les valeurs données aux paramètres :

```
int main()
{
    // avec des float
    cout<<fonct(1.5, 6.2)<<endl;

    // avec des int
    cout<<fonct(200, 100)<<endl;

    // avec des char
    cout<<fonct('z', 'a')<<endl;

    return 0;
}
```

Tous les paramètres et la valeur de retour ne sont pas obligatoirement génériques Des mélanges peuvent être fait entre variables génériques et variables typées. Par exemple une fonction :

```
template<typename TYPE> void maFonct(int a, float b, TYPE c)
{
    ...
}
```

est une fonction qui ne retourne rien et prend en paramètre un int, un float et un générique. Une fonction déclarée comme :

```
template < typename TYPE> TYPE maFonct(char*s, TYPE b);
```

spécifie une fonction qui retourne une variable du type générique et prend en paramètre une chaine de caractères et une variable du type générique.

Une fonction peut utiliser plusieurs types génériques. Dans ce cas elle les déclare chacun entre les chevrons de la façon suivante :

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

```
template <typename T,typename TT,typename TTT>
void fonct(T a, TT b,TTT c)
{
    if (a<b && a<c)
        cout<<a<<endl;
    else if (b<a && b<c)
        cout<<b<<endl;
    else
        cout<<c<<endl;
}
int main()
{
    fonct(65.666, 6,'a');   // affiche 6
    fonct('Z', 0.6,4);      // affiche 0.6
    fonct(80, 'A',78.8);    // aficche A
    fonct(10,20,30);        // affiche 10
    fonct('a',6.9,'z');     // afiche  6.9

    return 0;
}
```

Dans cet exemple la fonction fonct( )dispose de trois types génériques. Elle affiche la plus petite des trois valeurs.

Attention :

Si une fonction utilise un ou plusieurs types génériques elle doit avoir au moins un paramètre pour chaque type générique. La valeur de retour ne permet pas d'identifier un type. Le type ne peut être identifié qu'au moment de l'appel lorsque des valeurs sont spécifiées aux paramètres.

f. Fonctions comme champs de structures

A titre indicatif en C++ des fonctions peuvent être intégrées dans des structures :

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

struct t_test{
    int val;
    void affiche()
    {
        cout<<val<<endl;
    }

};

int main()
{
t_test v;
    v.val=10;
    v.affiche();
    return 0;
}
```

Dans cette exemple la struct t_test a deux champs. La variable val et une fonction pour afficher la valeur de la variable val. La fonction est définie directement avec la définition de la structure t_test.

La fonction peut également être définie en dehors, seule sa déclaration figurant alors dans la structure :

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

struct t_test{
    int val;
    void affiche();

};
// attention !
// la structure de référence est identifiée avec t_test::
void t_test::affiche()
{
   cout<<val<<endl;
}

int main()
{
t_test v;
    v.val=10;
    v.affiche();


    return 0;
}
```

La définition de la fonction nécessite dans ce cas d'identifier la structure de référence parce que des structures différentes peuvent comporter des fonctions de même nom. Cette identification est faite avec le nom de la structure suivi de deux points le tout intercalé entre le type de la valeur de retour et le nom de la fonction.

En C++ les possibilités des structures sont considérablement élargies et la structure est pour ainsi dire synonyme de classe. Mais à l'usage c'est la notion de classe qui est retenue et l'utilisation des structures, la plupart du temps, reste dans le cadre de ce qui est possible en C. La formulation de classe et le concepts associé dominent (voir chapitre classes et objets) .

5. Gestion des exceptions (base)

Le C++ propose une solution pour le traitement des erreurs d'exécution dans un programme. C'est le mécanisme des "exceptions" (exception et erreur sont synonymes dans ce contexte). Ce mécanisme est géré avec trois instructions intégrées dans le langage : throw et try-catch.

a. Instruction throw

L'exception est une valeur spécifique, de n'importe quel type, qui est utilisée pour caractériser une erreur dans une fonction ou un bloc d'instructions. Si l'erreur est détectée grâce à des tests adéquats, l'exécution du bloc est stoppée et cette valeur

est retournée. Mais à la différence du mécanisme de retour ordinaire la récupération de cette valeur nécessite l'utilisation d'une instruction de saut try-catch.

La fonction ci-dessous met en oeuvre le mécanisme de retour spécifique throw.

```
void test()
{
    switch(rand()%5){
        case 0  : throw ('a');      break;
        case 1  : throw (1);        break;
        case 2  : throw (2);        break;
        case 3  : throw (2.2);      break;
        case 4  : throw (4.4);      break;
    }
}
```

L'appel de la fonction rand() %5 donne une valeur aléatoire entre 0 et 4 compris.

Pour 0 l'instruction throw retourne un char, la valeur ascii 'a'
Pour 1 l'instruction throw retourne un int, la valeur 1
Pour 2 l'instruction throw retourne un int la valeur 2
Pour 3 l'instruction throw retourne un double, la valeur 2.2
Pour 4 l'instruction throw retourne un double, la valeur 4.4

b. Instruction de saut try-catch

Pour pouvoir récupérer et identifier l'exception nous avons besoin de l'instruction de saut try-catch. Le code à tester est placé dans le bloc try et si une exception est interceptée l'exécution saute directement dans un bloc catch correspondant, prévu pour le traitement de ce type d'erreur et les instructions suivantes ne seront pas exécutées. Chaque bloc catch est identifié par une variable du type de l'exception qu'il traite.

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

void test()
{
    switch(rand()%5){
        case 0  : throw ('a');      break;
        case 1  : throw (1);        break;
        case 2  : throw (2);        break;
        case 3  : throw (2.2);      break;
        case 4  : throw (4.4);      break;
    }
}

int main()
{
    srand(time(NULL));
    // ici code susceptible de produire des erreurs
    try
    {
        test() ;
        cout<<"ligne de code jamais executee"<<endl;
    }
    // ensuite les erreurs interceptées selon leurs types :
    catch(char err)
```

```
    {
        cout<<"char : "<<err<<endl;
    }
    catch(int err)
    {
        cout<<"int : "<<err<<endl;
    }
    catch(double err)
    {
      cout<<"double : "<<err<<endl;
    }

    return 0;
}
```

A chaque lancement du programme la fonction test() déclenche, via l'instruction throw, un type d'erreur (char, int, double) et les erreurs peuvent être traitées dans le bloc catch correspondant. Dans l'exemple le type et le numéro de l'erreur sont affichés.

c. Instruction throw et appels de fonctions

Le retour provoqué par l'instruction throw peut être le fait d'un appel de fonction cette fonction retournant une valeur. Par exemple :

```
int  err_int()
{
    cout<<"err_int appelee : ";
    return rand()%100;
}

void test()
{
    switch(rand()%6){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 2  : throw (2);         break;
        case 3  : throw (2.2);       break;
        case 4  : throw (4.4);       break;
        case 5  : throw err_int();   break; // appel
     }
}
```

La fonction err_int appelée avec throw retourne un int et est récupérée dans le catch de type int.

La fonction appelée peut très bien retourner une structure :

```
struct s_err{
    int v;
    float f;
};

s_err err_struct()
{
s_err s;
    s.v=rand()%10+100;
    s.f= (rand()%10000)/100.0;
    return s;
```

```
}
```

```
void test()
{
    switch(rand()%7){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 2  : throw (2);         break;
        case 3  : throw (2.2);       break;
        case 4  : throw (4.4);       break;
        case 5  : throw err_int();   break;
        case 6  : throw err_struct();break; // appel
    }
}
```

Dans ce cas, pour récupérer une exception de ce type il faut ajouter un catch du même type :

```
    try
    {
        test() ;
        cout<<"ligne de code jamais executee"<<endl;
    }
    catch(char err)
    {
        cout<<"char : "<<err<<endl;
    }
    catch(int err)
    {
        cout<<"int : "<<err<<endl;
    }
    catch(double err)
    {
      cout<<"double : "<<err<<endl;
    }
    catch(s_err s)   // nouveau catch pour le nouveau type
    {
        cout <<"struct s_err : "<<s.v<<" et "<<s.f<<endl;
    }
```

d. instruction throw sans valeur de retour

L'appel de throw sans paramètre provoque l'appel de la fonction terminate() qui met fin au programme par un appel à la fonction abort(). Nous avons placé cette instruction dans le case défaut de notre fonction test(), pour toute valeur d'exception inconnue le programme quitte :

```
void test()
{
    switch(rand()%8){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 2  : throw (2);         break;
        case 3  : throw (2.2);       break;
        case 4  : throw (4.4);       break;
        case 5  : throw err_int();   break;
        case 6  : throw err_struct();break;
```

```
        default : throw;            break;  // met fin au prg
    }
}
```

e. Préciser les retours d'exceptions des fonctions

Les fonctions qui retournent des exceptions peuvent préciser les types d'exceptions qu'elles retournent à la déclaration. C'est utile pour le lecteur du programme.

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

void test() throw(char,int,double, s_err); // à la déclaration


int main()
{
   ...

   return 0;
}

void test() throw(char,int,double, s_err)  // et à la définition
{
    switch(rand()%8){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 2  : throw (2);         break;
        case 3  : throw (2.2);       break;
        case 4  : throw (4.4);       break;
        case 5  : throw err_int();   break;
        case 6  : throw err_struct();break;
        //default : throw;             break;
    }
}
```

Attention

si un type d'exception est oublié dans la déclaration, elle ne peut plus être identifiée même si le bloc catch correspondant existe et le programme prend fin.

f. Exception non identifiée

Lorsqu'une exception déclenchée par throw n'est pas identifiée par un bloc catch la fonction terminate est appelée. Elle met fin au programme en appelant la fonction abort(). Par exemple :

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

void test()
{
    switch(rand()%5){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 3  : throw (2.2);       break;
    }
```

```
}

int main()
{
    srand(time(NULL));

    for (int i=0; i<50; i++) // 50 fois le test
        try
        {
            test() ;
            cout<<"ligne de code jamais executee"<<endl;
        }
        catch(char err)
        {
            cout<<"char : "<<err<<endl;
        }
        catch(int err)
        {
            cout<<"int : "<<err<<endl;
        }
        // le double reste non identifié

    return 0;
}
```

Le programme quitte dés qu'une exception de type double est lancée car alors elle n'est plus identifiée.

g. Bloc catch(...) par défaut

Le bloc catch suivi de (…) est le bloc catch par défaut qui intercepte toutes les exceptions non identifiées par les blocs précédents. Par exemple :

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

void test()
{
    switch(rand()%5){
        case 0  : throw ('a');       break;
        case 1  : throw (1);         break;
        case 3  : throw (2.2);       break;
    }
}

int main()
{
    srand(time(NULL));

    for (int i=0; i<50; i++) // 50 fois le test
        try
        {
            test() ;
            cout<<"ligne de code jamais executee"<<endl;
        }
        catch(char err)
        {
            cout<<"char : "<<err<<endl;
        }
```

```
        catch(int err)
        {
            cout<<"int : "<<err<<endl;
        }
        catch(...) // récupère le double
        {
            cout<<"erreur inconnue"<<endl;
        }
    return 0;
}
```

6. Espaces de noms(namespace) et raccourcis (using)

Les espaces de nom donnent la possibilité d'exprimer dans le programme des regroupements logiques, des unités d'action. Le code va pouvoir être cartographié selon des secteurs d'activité de façon claire. Plus un programme sera long et plus il aura besoin d'être divisé avec différents espaces nommés. Cette répartition du code peut aussi correspondre à un travail d'équipe chacun apporte alors au programme l'espace de nom qu'il a élaborer sur tel ou tel aspect.

Un espace de nom contient du code et le code contenu n'est visible que dans cet espace. L'instruction namespace permet d'écrire ses propres espaces de nom. La syntaxe est la suivante :

```
namespace monEspace{        // le nom de l'espace de nom

     // ici les déclarations et les définitions de variables,
     // fonctions, objets...

}
```

L'accès du dehors aux éléments de l'espace de nom se fait ensuite de la façon suivante :

```
    <nomEspace>::<nom de l'élément de l'espace de nom>
```

Par exemple :

```
#include <iostream>          // rand()
using namespace std;         // cout, cin et endl;

namespace perso
{

    int val;
    int test(int nb)
    {
        return rand()%nb;
    }
}

int main()
{
    perso::val=10;
    cout<<perso::val<<endl;
    cout<<perso::test(10)<<endl;

    return 0;
}
```

L'espace de nom "perso" contient une déclaration de variable et une fonction. Dans le main la variable est accessible avec perso::val et la fonction de même avec perso::test().

La fonction est à la fois déclarée et définie dans l'espace de nom mais il peut en être autrement. L'espace de nom peut ne contenir que la déclaration de la fonction et la fonction peut être définie en dehors. Sur le même fichier ça donne par exemple :

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

namespace perso
{
    int val;
    int test(int nb);
}

int main()
{
    perso::val=10;
    cout<<perso::val<<endl;
    cout<<perso::test(10)<<endl;

    return 0;
}

int perso::test(int nb)
{
    return rand()%nb;
}
```

Mais il peut être plus approprié de répartir le code sur plusieurs fichiers. Par exemple nous allons définir la fonction sur un autre fichier source. Le fichier fichier main devient :

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

namespace perso
{
    int val;
    int test(int nb);
}

int main()
{
    perso::val=10;
    cout<<perso::val<<endl;
    cout<<perso::test(10)<<endl;

    return 0;
}
```

et sur un autre fichier source intégré dans le programme nous avons la définition de la fonction de la façon suivante :

```
/* fichier 2*/
#include <iostream>
```

```
namespace perso
{
    int test(int nb)
    {
        return rand()%nb;
    }
}
```

En fait c'est toujours le même espace de nom qui est complété avec les définitions de fonction sur une autre page de code.

Pour utiliser un espace de nom dans plusieurs fichiers source il est intéressant d'intégrer l'espace de nom dans une librairie (un point h). L'espace de nom est alors défini dans la librairie. Il peut contenir toutes ses définitions de fonctions mais elles peuvent également être mises sur des fichiers à part. Ensuite il reste juste faire un include de la librairie qui contient un ou plusieurs espaces de noms partout où ils sont nécessaires.

Les espaces de nom peuvent s'imbriquer.

```
namespace perso
{
    int test(int nb);

    namespace perso2
    {
        int test2(int nb)
        {
            // actions
        }

    }
}
```

L'accès spécifie alors le chemin complet :

```
perso::perso2::test2();
```

Il peut y avoir des espaces de noms anonymes :

```
namespace perso
{
    namespace
    {
          void fonct()
          {

          }
    }
    fonct();  // est alors visible au niveau de namespace perso
}
```

Ils sont utilisés pour éviter d'éventuels conflits de noms, cela permet de garantir la localisation du code.

La directive using

L'accès aux éléments d'un espace de nom, dans l'exemple ci-dessus le préfixe perso:: peut s'avérer fastidieux s'il doit être répéter et pire s'il y a des namespaces imbriqués. La directive using permet d'éviter de répéter sans cesse le chemin d'accès à des éléments choisis ou à tous les éléments d'un espace de nom.

Par exemple using pour un élément :

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

namespace perso
{
    int val;
    void fonct()
    {
        cout<<"testfonct()"<<endl;
    }
}
using perso::val;

int main()
{
    val=20;
    cout<<val<<endl;
    perso::fonct();

    return 0;
}
```

Pour avoir using pour tous :

```
(...)
using namespace perso;

int main()
{
    val=20;
    cout<<val<<endl;
    fonct();

    return 0;
}
```

Pour un espace de nom anonyme il a en réalité a un nom caché attribué par par la machine par exemple "$$$" et un using implicite est implémenté à partir de ce nom :

```
namespace perso
{
    namespace
    {
           void fonct()
           {

           }
    }
    //using namespace $$$;  using implicite sur nom caché

    fonct();  // est alors visible au niveau de namespace perso
}
```

7. Intégrer d'autres langages dans le code C++

En C++ il y a moyen d'intégrer du code écrit dans d'autres langages au code C++, assembleur, C, pascal, fortran et probablement d'autres. L'instruction extern "C" permet de spécifier une convention de liaison (le "C" ne signifie pas langage C mais convention de liaison), c'est à dire un type d'édition de liens relatif au langage à

intégrer afin de pouvoir le traiter si l'environnement le permet (il faut consulter la documentation du compilateur pour connaître ses possibilités).

L'expression extern "C" peut être utilisée pour une seule instruction, ou pour un bloc d'instructions ou pour une librairie. Par exemple, pour intégrer du C dans du C++ nous pouvons avoir, pour une seule instruction :

```
extern "C" char* strcpy(char*, const char*);
```

Cette instruction spécifie que la fonction strcpy() de la librairie standard C devra être liée selon les conventions d'édition des liens du C.

Pour un bloc d'instruction :

```
extern "C"{
    char* strcpy(char*, const char*);
    int strlen(const char*);
    int strcmp(const char*, const char*);
}
```

Toutes ces instructions de la librairie standard C devront être liées selon les conventions d'édition des liens du C.

Pour une librairie :

```
extern "C"{
    #include <string.h>
}
```

La librairie entière et le code correspondant devront être liés selon les conventions d'édition des liens du C.

La relation entre le C et le C++ est étroite. Néanmoins ce sont deux langages qui peuvent être traités distinctement par le compilateur sur certains aspects. Cette question se pose par exemple si l'on crée des librairies en C avec l'intention de les utiliser en C et en C++. Dans cette optique nous pouvons trouver dans des librairies un entête comme :

```
#ifdef __cplusplus
extern "C" {
#endif

   // contenu de la lbrairie en question...
   char* strcpy(char*, const char*);
   int strlen(const char*);
   int strcmp(const char*, const char*);


#ifdef __cplusplus
}
#endif
```

Si le nom __plusplus de la macro vaut 1 alors la compilation est faite en C++ et l'instruction extern "C" est prise en compte. Sinon La compilation est faite en C et cette instruction n'est pas prise en compte.

C. La dimension objet du C++

Nous arrivons maintenant au coeur de la proposition C++, la notion d'objet, considérée comme révolutionnaire à son arrivée. Au départ elle répond à deux

objectifs essentiels par rapport à l'utilisation du C : sécuriser le code et faciliter sa réutilisation. Mais dans la pratique la portée de l'objet s'est avérée plus grande. Nous pouvons en effet considérer qu'un objet est finalement un petit programme autonome au sein d'un programme plus vaste constitué de nombreux objets en relation les uns avec les autres. De ce fait la conceptualisation par objets devient un enjeux plutôt stimulant d'autant que l'objet apparaît en même temps que se développe les réseaux. C'est un modèle de données qui permet à des programmes de dialoguer entre eux assez naturellement.

1. Classe et objet

Pour vérifier qu'un objet est un programme nous allons transformer un programme C, l'automate cellulaire avec lequel nous avons commencé ce chapitre, en un objet. Cette mutation mettra en évidence quelques aspects importants de cette nouvelle dimension qu'est la programmation objet.

a. Qu'est ce qu'une classe ?

Nous pouvons lire dans le livre "Le langage C++" de Bjarne Stroustrup créateur du langage que "L'objectif du concept de classe en C++ est de fournir aux programmeurs un outil de création de nouveaux types, aussi facile d'utilisation que les types intégrés [float, int etc.]" (p.248) .

La classe comme la structure est un type défini par l'utilisateur. Comme la structure elle permet de regrouper des variables de n'importe quel type. Mais avec les variables, elle offre aussi la possibilité de regrouper les fonctions associées aux traitements sur ces variables. Les éléments d'une classe sont appelés les membres de la classe. Les "données membres" pour les variables et les "fonctions membres" ou également "méthodes" pour les fonctions.

b. Qu'est ce qu'un objet ?

Un objet est tout simplement une variable du type de la classe dont il dépend. Comme avec une structure l'objet permet d'accéder aux données et fonctions membres qu'il regroupe. Avant d'avoir un objet il faut d'abord définir une classe.

c. Définir une classe, déclarer un objet, droit d'accès

La syntaxe est simple l'instruction class, un nom, un bloc d'instructions et un point virgule. Dans le bloc d'instructions placer les données membres et les méthodes.

Les fonctions sont définies en dehors de la classe, comme nous l'avons vu plus haut pour des fonctions intégrées à des structures. Le nom de la classe de référence suivi de deux points est indiquée entre la valeur de retour et le nom de la fonction :

```
class test
{
   // ici déclarer ses données membres
   int val;

   // ensuite (ou avant) déclarer ses fonctions membres
```

```
   void affiche();


} ; // point virgule

// définition des fonctions ailleurs avec spécification de la
// classe de référence  test::
void test::affiche()
{
   cout<<val<<endl;
}
```

Une fois la classe définie pour obtenir un objet il faut une déclaration comme pour n'importe quelle variable :

```
int main()
{
test a;   // a est un objet du type class test

}
```

Pour accéder aux données et fonctions membres c'est l'opérateur point qui est utilisé et flèche dans le cas d'un pointeur comme pour les structures :

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

class test
{
   int val;

   void affiche();


} ;
void test::affiche()
{
   cout<<val<<endl;
}

int main()
{

test a;                 // objet
    a.val=10;           // accès avec point
    a.affiche();

test*ptr = new test;    // pointeur objet
    ptr->val=20;        // accès avec flèche
    ptr->affiche();
    return 0;
}
```

Si vous compilez ce code ça ne marche pas et l'erreur est :

```
error : int test::val is private
```

En effet, dans un soucis d'encapsulation des données, pour garantir plus de sécurité du code, il y a trois accès possibles aux données et fonctions membres. Ils sont spécifiés avec ces instructions : public, protected et private.

- □ public : accès interne à la classe et à partir de l'objet.

- protected : accès interne à la classe et ses classes dérivées (voir héritage)
- private : accès interne à la classe uniquement.

Par défaut c'est private d'où l'erreur dans notre programme. Si nous voulons pouvoir accéder à la variable val et la fonction affiche() à partir d'un objet déclaré dans le main() ou ailleurs il faut spécifier que ces membres sont public :

```
class test
{

   public:         // accès public ...
   int val;

   void affiche(); // ...tant qu'un autre accès n'est pas spécifié

};
```

Maintenant ça marche.

d. Un programme C muté en classe et objet

Reprenons l'automate cellulaire étudié plus haut pour en faire une classe. C'est très simple il suffit de prendre variables globales et déclarations de fonctions, la définition de la structure peut rester en dehors ou bien être intégrée (si elle reste en dehors elle pourra être utilisée éventuellement pour les définitions d'autres classes).

Cela donne :

```
#include <iostream>    // attention ne pas oublier les entêtes !
using namespace std;

#include <conio.h>     // pour getch() et kbhit()
#include <windows.h>   // pour textcolor() et gotoxy()

typedef struct pos{
    int val;
    int color;
}t_pos;

class autocell  // définition de la classe pour l'automate
{
    private :
    int TX;
    int TY;

    t_pos**MAT;
    t_pos**SEV;

    int     compte_voisins      (int y, int x);

    public :
    void    init_matrice        (int tx, int ty);
    void    calcul              (void);
    void    copie               (void);
    void    affiche             (void);
    void    destroy_mat         (void);
    void    gotoxy              (int x, int y);
    void    textcolor           (int color);
};
```

Nous avons placé en données private tout ce qui a trait au fonctionnement "interne" de l'automate, les tailles de la matrice et la fonction compte_voisin() qui n'est appelée que dans la fonction calcul.

Les fonctions sont définies ensuite. Rien ne change à part l'entête de chacune des fonctions qui doivent être référencées à la classe dont elles dépendent :

```
/**************************************/
void autocell::init_matrice(int tx, int ty)
{
int i ;
    TX=tx;
    TY=ty;
    MAT=(t_pos**)malloc(sizeof(t_pos*)*TY);
    SEV=(t_pos**)malloc(sizeof(t_pos*)*TY);
    for (i=0; i<TY;i++){
        MAT[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        SEV[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        memset(MAT[i],0,sizeof(t_pos)*TX);
        memset(SEV[i],0,sizeof(t_pos)*TX);
    }
   MAT[TY/2][TX/2].val=1;
   MAT[TY/2+1][TX/2].val=1;
   MAT[TY/2][TX/2+1].val=1;
   MAT[TY/2+1][TX/2+1].val=1;

   MAT[TY/2][TX/2].color=1;
   MAT[TY/2+1][TX/2].color=1;
   MAT[TY/2][TX/2+1].color=1;
   MAT[TY/2+1][TX/2+1].color=1;
}
/**************************************/
void autocell::destroy_mat()
{
int i;
    for (i=0; i<TY; i++){
        free(MAT[i]);
        free(SEV[i]);
    }
    free(MAT);
    free(SEV);
}
/**************************************/
void autocell::calcul()
{
int x,y,nb_voisins;

   for (y=1; y<TY-1; y++){
      for (x=1 ;x<TX-1 ;x++){
         nb_voisins=compte_voisins(y,x);
         if (nb_voisins <2 || nb_voisins>3){
            SEV[y][x].val=0;
            SEV[y][x].color = MAT[y][x].color;
         }
         else{
            SEV[y][x].val=1;
            if (MAT[y][x].color<15)
                SEV[y][x].color = MAT[y][x].color+1;
         }
      }
   }
```

```
}
/****************************************/
int autocell::compte_voisins(int y, int x)
{
int nb=0;

   if (MAT[y][x+1].val==1)
     nb++;
   if (MAT[y-1][x+1].val==1)
     nb++;
   if (MAT[y-1][x].val==1)
     nb++;
   if (MAT[y-1][x-1].val==1)
     nb++;
   if (MAT[y][x-1].val==1)
     nb++;
   if (MAT[y+1][x-1].val==1)
     nb++;
   if (MAT[y+1][x].val==1)
     nb++;
   if (MAT[y+1][x+1].val==1)
     nb++;

   return nb;
}
/****************************************/
void autocell::copie()
{
int i,j ;
   for (j=0; j<TY; j++){
      for (i=0 ;i<TX ;i++){
         MAT[j][i]= SEV[j][i];
      }
   }
}
/****************************************/
void autocell::affiche()
{
int x,y ;

   for (y=0; y<TY; y++){
      for (x=0 ;x<TX ;x++){
         gotoxy(x,y);
         textcolor(MAT[y][x].color*16);
         putchar(' ');
      }
   }
}
/****************************************/
void autocell::gotoxy(int x, int y)
{
  COORD c;

  c.X = x;
  c.Y = y;
  SetConsoleCursorPosition (GetStdHandle(STD_OUTPUT_HANDLE), c);
}
/****************************************/
void autocell::textcolor (int color)
{
```

```
    SetConsoleTextAttribute (GetStdHandle(STD_OUTPUT_HANDLE),
color);
}
/****************************************/
```

Dans le main(), l'action est similaire sauf qu'elle est maintenant pilotée à partir d'un objet :

```
// ici les entêtes

// ici définition de la class autocell

// ici définitions des fonctions

int main()
{
int fin=0;
autocell a;

  printf("Action  : appuyer sur n'importe quelle touche\n"
         "Quitter : q");

  a.init_matrice(80,23);
  while(fin!='q'){

     if(kbhit()){
        fin=getch();
        a.affiche();
        a.calcul();
        a.copie();
     }
  }
  a.destroy_mat();

  return 0;
}
```

Ce point est important. Quoique ce ne soit pas visible le programme a considérablement changé : notre automate ne constitue plus le programme entier mais il est devenu une variable dans un programme plus vaste. Ainsi nous pouvons par exemple avoir facilement plusieurs automates et même des tableaux statiques ou dynamiques d'automates :

```
autocell a,b,c;
autocell tab[100];
autocell *dyn = new autocell[1+rand()%1000];
```

La gestion de plusieurs automates simultanément soulève de nouvelles questions et nécessite des adaptations : comment les initialiser différemment ? Comment les afficher ? Interagissent-ils ensemble ? Etc. Par ailleurs nous pouvons également ajouter au programme d'autres types d'objets d'autres automates afin de les faire fonctionner ensemble. Le programme est incontestablement monté potentiellement en puissance avec les nouvelles perspectives qu'il met à portée de main. De plus la classe automate peut être très facilement intégrée dans un autre programme moyennant ou non des modifications.

Mais reprenons notre expérimentation. Nous allons maintenant explorer le fonctionnement d'un certain nombre d'outils et concepts de base associés à la pratique de l'objet.

2. Constructeurs

Un constructeur est simplement une fonction appelée à la déclaration d'un objet et qui permet de lui passer des valeurs spécifiques. Pour commencer nous allons ajouter deux constructeurs à l'automate ci-dessus.

a. Paramétrer un objet à sa déclaration

La syntaxe d'un constructeur c'est le nom de la classe comme une fonction mais sans retour. Il peut être défini dans ou hors la classe. La version ci-dessous définit le constructeur dans la classe en utilisant la fonction init_matrice()

```
class autocell
{
   (...)
   // constructeur : déclaration dans la classe
   public :
   autocell() { init_matrice(80,20);}
   (...)
}
```

Le constructeur est nécessairement public sinon il ne sera pas accessible par l'objet déclaré.

Dans le main la déclaration d'un objet autocell déclenche automatiquement un appel à ce constructeur :

```
int main()
{
autocell a;  // appel le constructeur. Pas de parenthèse après a.
   (...)
}
```

Le fait d'avoir un constructeur permet de modifier la fonction init_matrice(). A priori elle n'a plus besoin d'être accédée directement par l'objet alors nous allons la placer en private.

Grâce à la surcharge des fonctions il est possible d'avoir plusieurs constructeurs dans une classe. Le constructeur est reconnu à la déclaration de l'objet en fonction des paramètres donnés. Voici un constructeur complémentaire pour la classe autocell qui permet de spécifier la taille de l'automate. Cette fois, pour changer, il est déclaré dans la classe et il est défini en dehors de la classe :

```
class autocell
{
   (...)

   // constructeur
   public :
   autocell () { init_matrice(80,20);}
   autocell (int tx, int ty);

   (...)
}
// définition du deuxième constructeur en dehors classe
autocell::autocell(int tx, int ty)
{
    init_matrice(tx,ty);
```

```
}
```

Dans le main() il y a maintenant possibilité de déclarer des objets autocell de tailles différentes :

```
int main()
{
autocell a;
autocell b(8,8);
autocell c(20,30);
   (...)


}
```

Remarque

Le constructeur fait appel à la fonction init_matrice() mais nous pouvons nous en passer et écrire directement le code dans le constructeur :

```
autocell::autocell(int tx, int ty)
{
    int i ;
    TX=tx;
    TY=ty;
    MAT=(t_pos**)malloc(sizeof(t_pos*)*TY);
    SEV=(t_pos**)malloc(sizeof(t_pos*)*TY);
    for (i=0; i<TY;i++){
        MAT[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        SEV[i]=(t_pos*)malloc(sizeof(t_pos)*TX);
        memset(MAT[i],0,sizeof(t_pos)*TX);
        memset(SEV[i],0,sizeof(t_pos)*TX);
    }
   MAT[TY/2][TX/2].val=1;
   MAT[TY/2+1][TX/2].val=1;
   MAT[TY/2][TX/2+1].val=1;
   MAT[TY/2+1][TX/2+1].val=1;

   MAT[TY/2][TX/2].color=1;
   MAT[TY/2+1][TX/2].color=1;
   MAT[TY/2][TX/2+1].color=1;
   MAT[TY/2+1][TX/2+1].color=1;
}
```

b. Le pointeur this

Si pour le second constructeur nous nommons les paramètres comme les variables concernées :

```
class autocell{

   private :
    int TX;
    int TY;

   (...)

   public :
```

```
   autocell (int TX, int TY);
   (...)
}
```

Alors ces variables locales à la fonction masquent les variables de même nom en données membres et les instructions :

```
autocell::autocell( int TX, int TY)
{
   TX=TX;  // sans effet !
   TY=TY;
   (...)
}
```

n'ont pas d'effet

Pour distinguer entre les paramètres et les données membres de même nom nous devons utiliser le pointeur this et de la façon suivante :

```
autocell::autocell( int TX, int TY)
{
   this->TX=TX;   // ok les valeurs TX et TY sont transmises
   this->TY=TY;   // aux données TX et TY
   (...)
}
```

Le pointeur this contient toujours l'adresse de l'objet courant (*this c'est l'objet courant) et il permet d'accéder à chacun des ses membres via l'opérateur flèche. C'est un opérateur,mot clé du langage, il est utile dans certaines situations (nous l'utilisons plus bas pour un constructeur de copie)

c. Constructeur sans paramètre

Si l'on ne définit aucun constructeur dans une classe le constructeur par défaut se contente de réserver l'espace mémoire pour l'objet et ces données membres mais elles ne sont pas initialisées.

Il peut être bien venu de définir un constructeur sans paramètre qui initialise les variables :

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


class test
{
    public:
    float*tab;
    int nb;
    // constructeur mise à 0
    test(){tab=NULL; nb=0;}

};

int main()
{
test t;
    cout<<t.tab<<endl;
```

```
    cout<<t.nb<<endl;
    return 0;
}
```

L'appel du constructeur sans paramètre à la déclaration de l'objet se fait sans parenthèse. Les parenthèses sont implicites et en rajouter provoque une erreur à la compilation.

Remarque

S'il s'agit d'un pointeur d'objet le pointeur n'est pas alloué. Ecrire :

```
int main()
{
test *ptr;
    cout<<ptr->tab<<endl;     // erreur à l'exécution
    cout<<ptr->nb<<endl;      // erreur à l'exécution

    return 0;
}
```

provoque la sortie du programme (access violation). Il est donc nécessaire d'ajouter une allocation mémoire avec l'opérateur new suivi du type de l'objet. Dans ce cas le constructeur sans paramètre est appelé également :

```
int main()
{
test *ptr = new test;
    cout<<ptr->tab<<endl;     // imprime 0
    cout<<ptr->nb<<endl;      // imprime 0

    return 0;
}
```

d. Constructeurs avec paramètres

Nous pouvons également définir un constructeur avec paramètres, ce que nous avons fait pour l'automate cellulaire afin de paramétrer sa taille. En reprenant l'exemple ci-dessus le constructeur alloue le tableau de float avec une taille donnée à la déclaration de l'objet. Nous avons également ajouté deux fonctions membres, une pour l'initialisation du tableau et une pour son affichage :

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

class test
{
    public:
    float*tab;
    int nb;
    // constructeur avec paramètre
    test(int nb);

    void init();
    void affiche();
};
//
```

```
test::test(int nb)
{
    this->nb = nb;
    tab=new float[nb];
}
//
void test::init()
{
    for(int i=0; i<nb;i++)
        tab[i]=rand()%1000/100.0;
}
//
void test::affiche()
{
    for (int i=0; i<nb; i++)
        cout<<tab[i]<<endl;
}
//
int main()
{
test a(10);
    a.init();
    a.affiche();

    return 0;
}
```

Remarque :

A partir du moment où un constructeur est défini il n'y a plus de constructeur par défaut. Dans le programme ci-dessus par exemple il n'est pas possible de déclarer un objet sans paramètre. Pour le faire il faut redéfinir un constructeur sans paramètre.

e. Constructeur et copie d'objet

Une autre possibilité de constructeur consiste à initialiser un objet avec les valeurs d'un autre objet déjà présent dans le programme. Pour ce faire il y a un constructeur par défaut qui effectue une copie membre à membre. Ce constructeur s'utilise avec l'opérateur d'affectation comme ci-dessous.

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

class M
{
   public :
   int val;

   M(int v){val=v;}                  // constructeur
   void affiche(){cout<<val<<endl;} // affichage
};
int main()
{
M m1(10), m2 = m1;     // m2 initialisé avec m1
    m1.val=20;         // modification de m1
    m1.affiche();      // imprime 20
```

```
    m2.affiche();      // imprime 10
    return 0;
}
```

Attention toutefois lorsqu'il est question de membres pointeurs et de données dynamiques. Par exemple en reprenant la classe test ci-dessus nous pouvons avoir un objet a avec un tableau de 5 float et un objet b copié à partir de a :

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

class test
{
    public:
    float*tab;
    int nb;
    // constructeur avec paramètre
    test(int nb);

    void init();
    void affiche();
};

(. . . ) // définitions des fonctions

int main()
{
test a(5);
test b = a;

   (...)
    return 0;
}
```

Vous pensez avoir maintenant deux tableaux de float ? Eh bien non car l'adresse mémoire du tableau a a été recopiée pour le tableau b. Les deux pointeurs pointent sur le même espace mémoire. Si vous initialisez le tableau de b ensuite vous pouvez constater que le tableau de a prend les mêmes valeurs que celui de b, en fait nous avons un seul tableau :

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

class test
{
    public:
    float*tab;
    int nb;
    // constructeur avec paramètre
    test(int nb);

    void init();
    void affiche();
};
test::test(int nb)
{
    this->nb = nb;
    tab=new float[nb];
}
//
```

```
void test::init()
{
    for(int i=0; i<nb;i++)
        tab[i]=rand()%1000/100.0;
}
//
void test::affiche()
{
    for (int i=0; i<nb; i++)
        cout<<tab[i]<<endl;
}

int main()
{
test a(5);
test b = a;

    b.init();
    cout<<"obj a :"<<endl;
    a.affiche();
    cout<<"obj b :"<<endl;
    b.affiche();

    return 0;
}
```

Pour éviter ce problème et avoir deux objets distincts il est nécessaire de réécrire le constructeur de copie. Sa forme, obligatoire est toujours la même :

```
nomClasse( référence objet à copier);
```

Ainsi dans la classe test nous avons :

```
class test
{
    public:
    float*tab;
    int nb;

    test(int nb);
    test(const test &a_copier);     // constructeur copie

    void init();
    void affiche();
};
```

et la définition du constructeur donne :

```
test::test(const test &a_copier)
{
    // copie complète de a_copier dans l'objet courant
    memcpy(this,&a_copier,sizeof(test));

    // allocation du pointeur tab de l'objet courant
    tab=new float[a_copier.nb];

    // copie du tableau de a_copier dans celui de l'objet courant
    memcpy(tab,a_copier.tab,sizeof(float)*nb);
}
```

Nous utilisons la fonction standart memcpy() qui permet de recopier un bloc de mémoire dans un autre (mais vous pouvez aussi recopier membre à membre). Le

premier appel fait la même copie que celle du constructeur par défaut. Toutes les données sont copiées. Ensuite il faut réallouer toutes les données dynamiques. Ici nous réallouons le tableau dynamique de l'objet courant.

Dans le main() si nous refaisons le test :

```
int main()
{
test a(5);

test b = a;
    b.init();
    cout<<"obj a :"<<endl;
    a.affiche();
    cout<<"obj b :"<<endl;
    b.affiche();
    return 0;
}
```

Nous constatons que les deux tableaux sont bien distincts. Le tableau de a contient ce qui traine en mémoire (il n'est pas initialisé) et le tableau de b des valeurs flottantes dans la fourchette spécifiée.

En fait c'est le fonctionnement de l'opérateur d'affectation qui a été modifié. L'exemple suivant le met en évidence :

```
int main()
{
test a(5);
    a.init();
    cout<<"obj a :"<<endl;
    a.affiche();

test b(1);
    b = a;
    cout<<"obj b :"<<endl;
    b.affiche();

    return 0;
}
```

Au départ l'objet a alloue à sa déclaration un tableau de 5 float, ce tableau est ensuite initialisé et affiché. Ensuite déclaration d'un objet b avec un tableau de 1 float. L'objet a est finalement copié dans b avec l'opérateur = d'affectation. Ici il ne s'agit plus en effet du constructeur de copie (voir la surcharge des opérateurs).

f. Constructeur avec conversion

C'est petit raccourci qui permet à la déclaration d'un objet d'initialiser une variable en utilisant l'opérateur d'affectation. Par exemple toujours pour notre classe test nous souhaitons pouvoir écrire :

```
test a = 5;
```

ou 5 donne la taille du tableau dynamique. Pour ce faire nous devons remplacer le constructeur à un paramètre que nous avons écrit précédemment par celui-ci :

```
class test
{

```

```
   public :
   float*tab;
   int nb;

   // constructeur avec conversion
   test(int n) : nb(n) {tab = new float[nb];};

   void init();
   void affiche();
};
```

L'entête du constructeur est suivie de deux points puis du nom de la variable à initialiser avec entre parenthèse la valeur à lui affecter et finalement un bloc d'instructions complémentaires s'il y a lieu.

Pour déclarer un tel constructeur hors classe, si par exemple il y a beaucoup d'instructions complémentaires, ça donne :

```
class test
{
    public:
    float*tab;
    int nb;
    // déclaration constructeur
    test(int n);

    void init();
    void affiche();
};

// définition constructeur avec conversion
test::test(int n ) : nb(n)
{
    tab=new float[nb];
}
```

Dans le main() nous pouvons avoir :

```
int main()
{
test a=5;
    a.init();
    a.affiche();

    return 0;
}
```

3. Destructeur

Figure inverse du constructeur, le destructeur sert à libérer la mémoire allouée. Il ne peut y en avoir qu'un par classe. Le destructeur d'un objet est automatiquement appelé à l'issue du bloc dans lequel il a été déclaré. Le destructeur par défaut libère la mémoire allouée par la machine. Mais si des opérations en dynamiques ont été opérées il faut écrire son propre destructeur. La syntaxe du destructeur est la suivante :

```
~nomClasse(aucun paramètre);
```

tilde suivi du nom de la classe et pas de paramètre. Par exemple pour l'automate cellulaire nous pouvons remplacer la fonction destroy_mat() par un destructeur qui sera automatiquement appelé à la sortie du programme :

```
class autocell
{
    public :
    (...)

    // déclaration destructeur
    ~autocell();

    (...)
};
// définition
autocell::~autocell()
{
    for (int i=0; i<TY; i++){
        free(MAT[i]);
        free(SEV[i]);
    }
    free(MAT);
    free(SEV);
}
```

Dans la classe test nous pouvons également ajouter un destructeur puisqu'il y a une allocation dynamique. Le programme complet donne :

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

class test
{
    public:
    float*tab;
    int nb;

    test(int n) : nb(n){tab=new float[nb];};

    // destructeur
    ~test() { free(tab);}

    void init();
    void affiche();
};
void test::init()
{
    for(int i=0; i<nb;i++)
        tab[i]=rand()%1000/100.0;
}
//
void test::affiche()
{
    for (int i=0; i<nb; i++)
        cout<<tab[i]<<endl;
}
int main()
{
    test a=10;
    a.init();
    a.affiche();
```

```
    a.~test();

    return 0;
}
```

4. Classes et membres "static"

a. Qualificatif static en C

Rappelons que en C une variable static locale à un bloc d'instructions dans une fonction ne disparaît pas de la mémoire à l'issue du bloc et lors de l'appel suivant de la fonction nous pouvons retrouver la variable avec la dernière valeur qu'elle avait prise. Par exemple :

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

int test()
{
static int v=0;
    return v++;
}
int main()
{
    for(int i=0; i<10; i++)
        cout<<test()<<endl;
    return 0;
}
```

ce petit programme imprime 0 1 2 3 4 5 6 7 8 9.

Autre propriété, une variable static déclarée en globale dans un fichier est réservé à ce fichier et ne peut pas exister en dehors de ce fichier. C'est vrai également pour une fonction :

```
/* fichier test.cpp */

#include <iostream>
using namespace std;

static int VAL=0;            // visible que dans ce fichier

static int test()            // visible que dans ce fichier
{
static int v=0;
    return v++;
}
   (...)
```

Ces deux propriétés sont toujours applicables en C++ même si la seconde est remplacée par l'objet et les droits d'accès à ces membres (public, private protected).

b. Qualificatif static et objets

L'instruction static dans une classe signifie une variable ou une fonction indépendante ou commune à tous les objets de la classe. A savoir si aucun objet n'existe ces données et fonctions sont néanmoins accessibles et si un ou plusieurs

objets existent elles sont communes à tous. L'accès à un membre static d'une classe se fait avec :

```
nomClasse::nomMembre
```

Par exemple :

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

class CL
{
    public :
    int val;
    static int s_val;
    static void fonct();
    void affiche() {cout<<s_val<<endl;}
};

// définition de la fonction static de la classe CL
void CL::fonct()
{
    cout<<"la fonction static"<<endl;
}

/* attention :
les variables static d'une classe doivent obligatoirement être
initialisées en dehors de la définition de la classe sans répéter
le mot clé static :
*/

int CL::s_val=0;

int main()
{
    // aucun objet
    CL::s_val=10;
    cout<<CL::s_val<<endl;    // affiche 10
    CL::fonct();              // affiche "la fonction static"

    // plusieurs objets
CL a,b;
    a.s_val=20;
    b.affiche();              // affiche 20

    return 0;
}
```

Les variables static d'une classe s'initialisent en dehors, comme des variables globales. Sans objet, nous constatons que variables et fonctions static de la classe CL existent néanmoins et sont utilisables. Avec plusieurs objets de la classe CL, nous voyons que la variable s_val de l'exemple est commune à tous.

5. Surcharge des opérateurs

En C++ tous les opérateurs courants :

- □ arithmétiques + - * / % combinés += -= *= /= %=
- □ incrémentation ++ --
- □ bit à bit << >> ~ | ^ & combinés <<= >>= ~= |= ^= &=

- de comparaison > < >= <= == !=
- multicondition && ||
- pointeurs → * [ ]
- fonction ()
- liste , (la virgule)

peuvent être utilisés sur des objets. Ils sont alors considérés comme des fonctions bâties avec l'expression :

```
operator op
```

où operator est un mot clé et op un opérateur, par exemple + c'est *opérator+*, = c'est *operator=* etc. Une fonction opérateur peut être déclarée et appelée exactement de la même façon qu'une autre, l'utilisation de l'opérateur devenant un raccourci de l'appel explicite de cette fonction.

Notons par exemple les formes suivantes:

```
objet1 = objet2 + objet3            // si une redéfinition de +
objet1 = operator+(objet2,objet3);  // redéfinition hors class
objet1 = objet2.operator+(objet3);  // redéfinition dans class
```

Il faut distinguer ces deux possibilités :

- soit la redéfinition est faite hors classe, dans une fonction globale,
- soit la redéfinition est faite dans une classe

a. Fonction operator globale hors classe

Soit une classe M simple, voici pour des objets de la classe M une redéfinition de l'opérateur + en globale, hors classe :

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

class M
{
    public :
    int val;

    M(int v){val=v;}                // constructeur

    void affiche() {cout<<val<<endl;}     // fonct. membre
};

M operator+(M m1,M m2)       // fonct. Redefinition hors class
{
M m0(0);
    m0.val = (m1.val+m2.val)/2;
    return m0;
}
```

La fonction de redéfinition prend en argument deux objets de type M et retourne un objet M dont le champ val contient la moyenne des données val des deux paramètres.

Dans le main() voici quatre tests :

```
int main()
{
M a(10), b(20), c(30);
```

```
    a = a+b;         //15
    a.affiche();

    a = c+b;         //25
    a.affiche();
                    // (attention ici a vaut 25)
    a = a+b+c;      // a = (a+b)+c  // (((25+20)/2) + 30)/2 = 26
    a.affiche();

    (a+b).affiche(); // 23

    return 0;
}
```

Nous pouvons constater que le fonctionnement de l'opérateur + associé à des objets de type M produit bien le résultat souhaité. Attention lorsqu'il y a plus de deux opérandes la priorité se fait de la gauche vers la droite. Pour le dernier test l'objet en retour n'est pas stocké mais le type de l'expression est un objet M et donc il y a possibilité d'accéder à ses membres.

b. Fonction operator localisée dans une classe

Deuxième cas maintenant lorsque la redéfinition est faite dans une classe. Nous sommes alors restreints à écrire une fonction avec zéro ou un paramètre, pas plus, et la fonction peut avoir ou non une valeur de retour.

Dans l'exemple suivant la fonction opérator+ de la class M ne retourne rien et prend toujours en paramètre un objet M, l'opération réalisée est comme précédemment la moyenne des valeurs du champ val de l'objet courant et du champ val de l'objet passé en paramètre :

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

class M
{
    public :
    int val;

    M(int v){val=v;}

    void operator+(M m);

    void affiche() {cout<<val<<endl;}
};

// définition fonction sans retour de l'opérateur +
void M::operator+(M m)
{
    val = (val+m.val)/2;
}

int main()
{
M a(10), b(20), c(30);

    //a = a + b;    //  = ne fonctionne pas si pas de retour
```

```
                    //  l'expression a+b n'a pas de valeur
    a+b;            // 15
    a.affiche();

    a.operator+(b);  // 17
    a.affiche();

   // a+b+c;         // ne fonctionne puisque pas de retour
                     // (le calcule ne peut pas être décomposé)
    return 0;
}
```

Nous constatons que :

- puisque l'expression a+b ne retourne rien il n'est pas possible d'utiliser l'opérateur d'affectation =
- l'instruction a+b; effectue l'opération et la valeur de a.val est modifiée
- cette instruction est équivalente à a.operator+(b); qui effectue de même l'opération
- L'instruction a+b+c; ne peut pas être décomposée en (a+b)+c parce que a+b ne renvoie rien.

Deuxième exemple, cette fois la fonction operator retourne l'objet courant :

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

class M
{
    public :
    int val;

    M(int v){val=v;}   // constructeur

    M operator+(M m);

    void affiche() {cout<<val<<endl;}
};

// définition avec retour de la fonction operator+
M M::operator+(M m)
{
    val = (val+m.val)/2;
    return *this;
}
```

l'opérateur this est un pointeur qui contient toujours l'adresse de l'objet l'objet courant ainsi *this c'est l'objet courant et la fonction retourne l'objet courant. Voic quelques tests dans le main() :

```
int main()
{
M a(10), b(20), c(30);

    a = a+b;
    a.affiche();    // 15

    a+c;
    a.affiche();    // 22
```

```
    a.operator+(b);
    a.affiche();    // 21

    a = b+c;
    a.affiche();    // 25
    b.affiche();    // 25

    a.operator+(b+c);
    a.affiche();    // 26
    b.affiche();    // 27

    a=a+b+c;        // (a+b)+c
    a.affiche();    // 28
    b.affiche();    // 27


    return 0;
}
```

Nous pouvons constater que :

- Maintenant l'opérateur d'affectation = peut être utilisé.
- L'expression a=a+b est équivalente aux expressions a+b; et a.operator(b);
- Si l'on écrit a=b+c; a est modifié mais b aussi. C'est équivalent à l'expression a.operator+(b+c);
- En revanche dans l'expression a+b+c seul a est modifié (le résultat de (a+b) est stocké dans a et le résultat de a+c est stocké dans a).

c. Fonction operator et données dynamiques

Dans le cas de données dynamiques il est nécessaire de veiller à ce que les allocations de mémoire soient correctement effectuées et faire très attention à ce que des pointeurs d'objets différents peuvent pointer sur les même données. Par exemple voici la surcharge de l'opérateur + pour concaténer des chaines de caractères :

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

class D
{
    public :
    char* s;

    public :
    D() {s = NULL;}
    D(char*s);

    void affiche();
    D operator+(D &concat);
};

D::D(char*s)
{
    this->s=new char[strlen(s)+1];
    strcpy(this->s,s);
}
```

```
D D::operator+(D &a)
{
char*s0;
    // attention si la chaine a.s est null
    if (a.s != NULL){

        // atention au cas s1=s1+s1 par exemple

        // (rappelons que &a avec a une reférence donne
        // l'adresse contenue par a)

        if (this==&a){
            s0= new char[strlen(s)+1];
            strcpy(s0,s);
        }
        else
            s0=a.s;

        s=(char*)realloc(s,strlen(s)+strlen(s0)+1);
        strcat(s,s0);
    }
    return *this;
}

void D::affiche()
{
    cout<<s<<endl;
}

int main()
{
D s0; // chaine NULL
D s1("toto");
D s2("tata");

    s1=s1+s0;
    s1.affiche();   // toto

    s1=s1+s2;
    s1.affiche();   // tototata

    s1=s1+s1;       // tototatatototata
    s1.affiche();

D s3("titi");
    s1=s2+s3;
    s1.affiche();   // tatatiti
    s2.affiche();   // tatatiti

    cout<< ((s1.s==s2.s)? "meme chaine": "autre chaine")<<endl;
    for(char*i=s1.s; *i!='\0';i++)
        (*i)++;
                    // meme chaine
    s1.affiche();   // ububujuj
    s2.affiche();   // ububujuj

    return 0;
}
```

Pour la première concaténation la fonction operator+ prend en compte l'addition d'une chaine NULL à savoir rien n'est touché.

L'addition de s2 avec s1 est normal, la chaine de s2 est concaténée à la suite de la chaine élargie suffisamment en mémoire de s1

L'addition de s1 avec s1 pose un problème : il faut doubler la taille de la chaine mais avant il faut garder une copie de la chaine d'origine afin de pouvoir l'ajouter ensuite.

Pour l'addition s1=s2+s3, l'objet s2 contient la concaténation et ensuite l'objet s2 est copiée dans l'objet s1. Or c'est une copie membre à membre simple de sorte que le pointeur s1.s prend la valeur de s2.s, les deux pointeurs pointent alors sur la même chaine. Pour remédier à cela il faut redéfinir également l'opérateur d'affectation.

Voici une proposition de surcharge de l'opérateur = ajoutée au programme précédent :

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

class D
{
    public :
    char* s;

    public :
    D() {s = NULL;}
    D(char*s);

    void affiche();
    D operator+(D &concat);
    D operator=(D o);
};

D::D(char*s)
{
    this->s=new char[strlen(s)+1];
    strcpy(this->s,s);
}

D D::operator+(D &a)
{
char*s0;
    // attention si la chaine a.s est null
    if (a.s != NULL){
        // atention au cas s1=s1+s1 par exemple
        if (this==&a){
            s0= new char[strlen(s)+1];
            strcpy(s0,s);
        }
        else
            s0=a.s;
        s=(char*)realloc(s,strlen(s)+strlen(s0)+1);
        strcat(s,s0);
    }
    return *this;
}

D D::operator=(D o)
{
```

```
    // attention au cas s1=s1, s'il s'agit des mêmes chaines
    if(s!=o.s ) {
        // libérer la chaine de l'objet courant
        delete s;

        // réallouer une chaine de la bonne taille
        // pour l'objet courant
        s = new char[strlen(o.s)+1];

        // copier la chaine pour l'objet courant
        strcpy(s,o.s);
    }
    return *this;
}

void D::affiche()
{
    cout<<s<<endl;
}

int main()
{
D s1("toto"),s2("tata"),s3("titi");

    s1=s1;
    s1.affiche();   // toto

    s1=s2;
    s1.affiche();   // tata

    s1=s1+s1;
    s1.affiche();   // tatatata

    s1=s2+s3;
    s1.affiche();   // tatatiti
    s2.affiche();   // tatatiti
    for(int i=0; s1.s[i]!='\0'; i++){
        s1.s[i]++;
        s2.s[i]--;
    }
    s1.affiche();   // ububujuj
    s2.affiche();   // s's'shsh

   system("PAUSE");
    return 0;
}
```

La fonction operator= commence par vérifier que l'affectation n'a pas lieu sur une même chaine, probablement un même objet avec une instruction comme s1 = s1; par exemple. Pourquoi ?

A cause de la ligne delete s; dans la fonction operator=. Si cette vérification n'est pas faite, s est libérée puis tentative d'accès sur s pour copie ce qui produit une erreur à l'exécution avec sortie du programme.

En conclusion, attention au passage de valeurs des paramètres qui provoquent des copies membre à membre, mais aussi à la valeur de retour qui est recopiée dans la variable de récupération. La surcharge des opérateurs pose différents problèmes

selon les opérateurs et les données concernées (dynamiques ou pas etc.). La mise en place peut s'avérer délicate, source de confusion. Il convient d'être prudent.

6. Classes génériques ("template" ou "patron")

a. Principe

Nous avons présenté les "template" de fonctions ou fonctions génériques nommées parfois patrons ou modèles en français. Ces fonctions peuvent utiliser différents types de variables, il suffit de les préciser au moment de l'appel. Le même principe peut être utilisé pour des classes. Typiquement une classe pile par exemple, capable de produire des piles de n'importe quel type, int, float, structures etc.

b. Syntaxe de base

La syntaxe pour définir une classe générique est la suivante, soit T le nom du type générique et C le nom de la classe :

```
template <class T> class C  { // les membres};
```

A la déclaration d'un objet il faut ensuite spécifier le ou les types souhaités :

```
C<int> e1;
```

Dans la classe, la déclaration des données et fonctions qui utilisent le type générique se fait avec le nom donné au type générique :

```
template <class T> class C
{
    // des données membres
    int val;
    T valGenerique;

    // des fonctions membres
    void affiche();
    T fonct(int a, T b);
};
```

Pour la définition des fonctions en dehors de la classe il faut spécifier que la fonction est paramétrée par le type générique comme pour n'importe quelle fonction générique (cf fonctions génériques) :

```
template<class T> ...
```

ensuite spécifier le type de la valeur de retour :

```
template<class T>  void...     // si rien
template<class T>  int...      // si int
template<class T>  T...        // si générique
```

ensuite le nom de la classe suivi du nom de type générique entre chevron :

```
template<class T> void   C<T>...
template<class T> int    C<T>...
template<class T> T      C<T>  ...
```

ensuite les deux points fois deux et le nom de la fonction suivie du nom de type générique entre chevrons :

```
template<class T> void   C<T> :: fonct1<T>  ...
template<class T> int    C<T> :: fonct2<T>  ...
template<class T> T      C<T> :: fonct3<T>  ...
```

Et pour finir la liste des paramètres et le bloc des instructions, par exemple :

```
template<class T> void   C<T> :: fonct1<T>  ( )  { ... }
template<class T> int    C<T> :: fonct2<T>  (int a) { ... }
template<class T> T      C<T> :: fonct3<T>  (T a, float b) { ... }
```

c. syntaxe constructeurs

En ce qui concerne les constructeurs, nous avons :

```
template<class T> C<T> :: C<T>  ( )  { ... }
template<class T> C<T> :: C<T>  (int a) { ... }
template<class T> C<T> :: C<T>  (T a, float b) { ... }
```

disparaissent la valeur de retour et le nom de la fonction qui est remplacé par le nom de la classe suffixé avec le nom de type générique entre chevrons.

d. Syntaxe avec plusieurs types génériques

S'il y a plusieurs types génériques c'est le même principe mais avec la liste des noms de types génériques séparés par une virgule entre les chevrons, pour la classe ça donne :

```
template <class T1, class T2, class Tn> class Cn
{
    // les membres
};
```

Et pour le reste il suffit de remplacer

```
<class T>
```

de la syntaxe de base par

```
<class T1, class T2, class T3> et <T> par <T1, T2, T3>
```

Pour la déclaration d'un objet il faut spécifier chacun des types :

```
Cn<int, float,char> e;
```

e. Exemple d'implémentation d'une pile générique

La pile est organisée autour d'un tableau dynamique dont la taille est définie à la déclaration via un constructeur. Elle comprend les primitives de base et une fonction d'affichage qui permet de visionner le contenu de la pile :

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

template <class T> class pile
{
    private :
    int max;
    int sommet;
    T *p;

    public :
    //
    pile();
    pile (int nb);
    //
    bool pile_vide()    {return sommet==0;}
    bool pile_pleine()  {return sommet==max;}
    void empiler(T e);
    T depiler();
```

```
    void afficher();
};
```

Les définitions des constructeurs :

```
template<class T>pile<T>::pile()
{
    sommet=0;
    max=10;
    p=new T[max];
}

template<class T>pile<T>::pile(int nb)
{
    sommet=0;
    max=nb;
    p=new T[max];
}
```

Les définitions des fonctions membres :

```
template<class T>void pile<T>::empiler(T e)
{
    if(!pile_pleine())
        p[sommet++]=e;
}

template<class T> T pile<T>::depiler()
{
    return(!pile_vide())? p[--sommet]: -1;
}

template<class T> void pile<T>::afficher()
{
    if (pile_vide())
        cout<<"pile vide"<<endl;
    else{
        for (int i=0; i<sommet; i++)
            cout<<p[i]<<" ";
        cout<<endl;
    }
}
```

et un test dans le main()

```
int main()
{
pile<int> pi(20);

    pi.afficher();          // pile vide
    for(int i=0; i<10; i++)
        pi.empiler(i*10);
    pi.afficher();          // 0 10 20 30 40 50 60 70 80 90

    for(int i=0; i<5; i++)
        pi.depiler();
    pi.afficher();          // 0 10 20 30 40

pile<float> pf(20);

    pf.afficher();          // pile vide
```

```
    for(int i=0; i<10; i++)
        pf.empiler(i/10.0);
    pf.afficher();          // 0 0.1 0.2 0.3 0.4  ....  0.9

    for(int i=0; i<5; i++)
        pf.depiler();
    pf.afficher();          // 0 0.1 0.2 0.3 0.4

    return 0;
}
```

f. Spécialisation de fonction sur un type donné

Nous avons mentionné au départ que le type générique pouvait être instancié avec n'importe quel type. Certes mais dans notre pile des problèmes peuvent se poser pour certaines fonctions si l'on donne comme type une structure, par exemple la fonction d'affichage ne marche plus.

Pour remédier à cela il y a la possibilité d'écrire une fonction dédié à un type particulier et que le compilateur choisira lorsque ce type sera spécifié comme le type de l'objet. Pour le tester nous ajoutons une structure test au programme précédent cette hypothèse oblige à disposer de deux fonction spécialisées pour afficher() et dépiler (le retour suppose une struct test) .

Voici la structure :

```
typedef struct {int x, y;}test;
```

Attention elle devra être visible avant la définition des fonctions spécialisées qui l'utilise, juste au dessous de la classe ou au dessus par exemple.

Voici la fonction de dépilement spécialisée pour struct test :

```
// si structure test
test pile<test>::depiler()
{
   test e={-1,-1};
   return(!pile_vide())? p[--sommet]: e;
}
```

et la fonction d'affichage si struct test :

```
void pile<test>::afficher()
{
    if (pile_vide())
        cout<<"pile vide"<<endl;
    else{
        cout<<"pile de struct test :"<<endl;
        for (int i=0; i<sommet; i++){
            cout<<p[i].x<<endl;
            cout<<p[i].y<<endl;
                cout<<"----------------------"<<endl;
          }

    }
}
```

Voici une test dans le main() :

```
int main()
{
pile<char> pi(20);

    pi.afficher();          // pile vide
    for(int i=0; i<10; i++)
        pi.empiler(i+'A');
    pi.afficher();          // A B C D E F G H I J

    for(int i=0; i<5; i++)
        pi.depiler();
    pi.afficher();          // A B C D

pile<test> pp(20);
test e;
    pp.afficher();          // pile vide
    for(int i=0; i<10; i++){
        e.x=rand()%800;
        e.y=rand()%600;
        pp.empiler( e);
    }
    pp.afficher();          // pile structure test :
                            // et 10 structures test

    for(int i=0; i<5; i++)
        pp.depiler();
    pp.afficher();          // 5 structures test restantes

    return 0;
}
```

7. Héritage

l'héritage consiste à ajouter des champs ou des méthodes à une classe existante en définissant une nouvelle classe qui prolonge la première. La classe initiale est appelée classe de base, classe mère ou superclasse, et l'autre classe dérivée, classe fille ou sous-classe. En résumé un objet d'une classe dérivée contient :

- ses propres champs et ceux de la classe mère, hors les champs private.
- ses propres méthodes et celles de la classe mère, hors les méthodes private.

Seul l'opérateur private permet de réduire l'accés aux données et méthodes de la classe de base depuis une classe héritée.

Du point de vue de la modélisation, nous pouvons avoir un système d'ensembles et de sous-ensembles avec cette particularité informatique que le sous ensemble est plus large que l'ensemble de base parce qu'il lui ajoute des données et des fonctions. Le sous-ensemble précise l'ensemble de base en l'orientant dans un certain sens.

Par exemple à partir d'une classe véhicule, nous pouvons dériver une classe voiture, une classe camion, une classe deux roues. Les voitures sont des véhicules avec des spécificités propres, différents de celles des camions et des deux roues. La classe voiture elle-même peut être décomposée ensuite en plusieurs types de voitures avec une classe pour les voitures familiale, une classe pour les voitures de course, une autre classe pour les utilitaires. De même à partir de la classe camion il peut y avoir plusieurs ensembles dérivés, chacun précisant une catégorie de

camion. Idem pour les deux roues. Chaque classe dérivée implémente un nouveau type d'ensemble qui hérite des propriétés de la classe de base. La classe dérivée ajoute des précisions et oriente dans un certain sens l'ensemble de base. La totalité des données et fonctions du programme est organisée sous la forme d'un arbre.

a. Définir une classe dérivée

Soit la classe de base suivante :

```
class nomBase
{
    // données membres
    // fonctions membres
};
```

La syntaxe pour écrire une classe dérivée est

```
class herit : [accès public/protected/private] nomBase
{
    // données membres
    // fonctions membres
};
```

Pour un objet de la classe herit, les données et fonctions membres de la classe herit s'ajoutent aux données et fonctions membres de la classe base.

Exemple :

```
// CLASSE DE BASE
class base
{
    public :
    int v1;

    public :
    base(){v1=0;}
    base(int v) {v1=v;}

    void affiche() {cout<<"base v1 : "<<v1<<endl;}
};

// CLASSE DERIVEE :
class herit : private base
{
    public :
    int v2;

    public :
    herit(int v2)  { this->v2=v2;}

    void affiche2();
};
//
void herit::affiche2()
{
    cout<<"base v1  : "<<v1<<endl;
    cout<<"herit v2 : "<<v2<<endl;
}
//
int main()
{
    base b(5);
```

```
    b.affiche();    // imprime
                    // base v1 : 5

    herit h(100);
    h.affiche2();   // imprime
                    // base v1  : 0
                    // herit v2 : 100
    return 0;
}
```

La déclaration d'un objet de classe dérivée s'effectue comme pour n'importe quel objet. Le constructeur de la classe herit initialise la variable v2 de l'objet avec une valeur donnée à la déclaration. Pour ce qui concerne la variable v1 de la classe de base c'est le constructeur par défaut (sans paramètre) qui est implicitement appelé. Une déclaration comme :

```
herit h(50);
```

initialise v2 à 50 et v1 à 0.

Cet exemple fonctionne mais Il y a plusieurs choses à corriger et examiner :

- Il convient d'utiliser les constructeurs de la classe de base
- Éventuellement il serait plus clair d'avoir un seul nom "affiche" pour les fonctions d'affichage, c'est à dire de pouvoir redéfinir une fonction ou une variable avec le même nom qu'une fonction ou qu'une variable de la classe de base. Ce qui pose ensuite la question de pouvoir les distinguer.
- Par ailleurs il est possible de contrôler les droits d'accès aux données et fonctions membres de la classe de base à partir de la classe dérivée.

b. Appeler explicitement un constructeur de la classe de base

Il y a possibilité d'appeler le constructeur de la classe de base. Il suffit de faire suivre le constructeur de la classe dérivée par deux points et le constructeur souhaité de la classe de base. Voici trois constructeur ajoutés à notre classe herit :

```
class herit : public base
{
    public :
    int v2;

    public :
    herit() : base() {v2=0;}
    herit(int v2) : base(rand()%10) {this->v2=v2;}
    herit(int v1,int v2) : base (v1) { this->v2=v2;}

    void affiche2();
};
```

Le premier met les deux variables à 0.
le second initialise la variable v2 de l'objet avec une valeur passée à la déclaration et la variable v1 de la classe de base avec une valeur aléatoire entre 0 et 9
Le troisième initialise les variables v1 et v2 avec des valeurs passées à la déclaration. Par exemple :

```
int main()
{
    herit h1;
```

```
    h1.affiche2(); // imprime 0 0

    herit h2(10);
    h2.affiche2(); // un nombre entre 0 et 9 et 10

    herit h3(20,30);
    h3.affiche2(); // 20 et 30

    return 0;
}
```

c. Redéfinition de données ou de fonctions

Un autre point important est de pouvoir réutiliser les mêmes noms pour certaines fonctions ou données membres dans une classe dérivée. Ici par exemple avoir deux fonctions affiche(), une pour la classe base et une pour la classe dérivée :

```
class base
{
    public :
    int v1;

    public :
    base(){v1=0;}
    base(int v) {v1=v;}

    void affiche() {cout<<"base v1 : "<<v1<<endl;}
};
//
class herit : public base
{
    public :
    int v2;

    public :
    herit() : base() {v2=0;}
    herit(int v2) : base(rand()%10) {this->v2=v2;}
    herit(int v1,int v2) : base (v1) { this->v2=v2;}

    void affiche();
};
//
void herit::affiche()
{
    cout<<"base v1  : "<<v1<<endl;
    cout<<"herit v2 : "<<v2<<endl;
}
//
int main()
{
    base b(5);
    b.affiche();  // 5

    herit h(10,20);
    h.affiche(); // 10 20

    return 0;
}
```

Au moment de l'appel c'est la fonction membre la plus proche de l'objet qui a la priorité. Dans l'exemple l'objet h de la classe herit appel la fonction affiche() de la classe herit et l'objet b de la classe base appel la fonction affiche de la classe base.

d. Spécifier un membre de la classe de base

Il peut être nécessaire de forcer l'accès à une fonction ou une variable de la classe de base par exemple parce que le nom est repris dans la classe dérivée. Pour ce faire la syntaxe est :

```
nomClassBase :: nomVar
nomClassBase :: nomFonct
```

Dans notre exemple la fonction affiche() de la classe herit peut utiliser la fonction affiche() de la classe base pour afficher la valeur de la variable v1 :

```
void herit::affiche()
{
    base::affiche();
    cout<<"herit v2 : "<<v2<<endl;
}
```

De même en ce qui concerne les constructeurs de la classe herit. C'est un autre moyen pour accéder aux variables de la classe de base :

Nous pouvons remplacer :

```
herit() : base() {v2=0;}
par
herit() {base::v1=0; v2=0;}
//
herit(int v2) : base(rand()%10) {this->v2=v2;}
par
herit(int v2) {base::v1=rand()%10; this->v2=v2;}
//
herit(int v1,int v2) : base (v1) { this->v2=v2;}
par
herit(int v1,int v2) {base::v1=v1; this->v2=v2;}
```

e. Droits d'accès de la classe héritée

Il reste à se pencher sur les droits d'accès aux membres d'une classe dérivée et à partir de la classe dérivée aux membres de la classe de base.

Pour une classe de base nous avons vu des membres :

public : accès interne à la classe et à partir des objets de cette classe
protected : accès interne à la classe et aux classes dérivées mais fermé aux objets
private : accès en interne à la classe mais fermé aux classes dérivées et aux objets.

Par exemple :

```
class base
{
    private :
    int v1;

    public :
    base(){v1=0;}
```

```
    base(int v) {v1=v;}

    void affiche() {cout<<"base v1 : "<<v1<<endl;}
};

int main()
{
    base b(5);
    b.affiche();

    b.v1=10 ; // erreur à la compilation, v1 inaccessible
              // si private ou protected
    return 0;
}
```

la variable v1 est inaccessible par la variable b si déclarée private ou protected.

Ces restrictions demeurent depuis une classe dérivée. Avec v1 private :

```
class base
{
    private :
    int v1;

    public :
    base(){v1=0;}
    base(int v) {v1=v;}

    void affiche() {cout<<"base v1 : "<<v1<<endl;}
};

class herit : public base
{
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit h(20);

    h.affiche();       // erreur si v1 private
    h.v1=rand()%100;   // erreur si v1 private

    return 0;
}
```

L'accès est impossible de la classe herit avec la fonction affiche et d'un objet de la classe herit

Avec v1 protected :

```
class base
{
    protected :
    int v1;

    (. . .)
};
```

```
class herit : public base
{
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit h(20);

    h.affiche();       // ok si v1  protected
    h.v1=rand()%100;   // erreur si v1 private

    return 0;
}
```

L'accès est possible de la classe herit avec la fonction affiche, mais pas d'un objet.

Par ailleurs un de ces droits d'accès est également à spécifier globalement pour tous les éléments de la classe de base au moment de la définition de la classe dérivée. Soit une classe de base :

```
class test
{
    // membres base
};
```

la classe dérivée peut avoir globalement un accès public :

```
class deriv : public base
{
    // membres deriv
}
```

ou protected

```
class deriv : protected base
{
    // membres deriv
}
```

ou private

```
class deriv : public base
{
    // membres deriv
}
```

aux membres de la classe de base.

Comment s'articulent droits d'accès globaux et droits d'accès locaux aux membres de la classe de base ?

La règle est qu'ils ne peuvent que être restreints. Il n'est pas possible d'élargir les droits il est seulement possible de les diminuer. Le mieux est de récapituler ces interactions dans un tableau :

Accès local dans	**Accès global à parti de la classe dérivée**

classe de base	public	protected	private
public	*public*	*protected*	*private*
protected	*protected*	*protected*	*private*
private	*Inaccessible si private dans la base*		

Admettons la variable v1 public dans la base, si la classe de base est mentionnée public pour la classe dérivée :

```
class herit : public base
{
    public :
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit h(20);
    h.affiche();       // ok pas de restriction
    h.v1=rand()%100;   // ok pas de restriction
    return 0;
}
```

si la classe de base est mentionnée protected pour la classe dérivée :

```
class herit : protected base
{
    public :
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit h(20);
    h.affiche();       // ok  la restriction n'empêche pas
    h.v1=rand()%100;   // erreur v1 protected à partir de h
    return 0;
}
```

si la classe de base est mentionnée private pour la classe dérivée :

```
class herit : private base
{
    public :
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
```

```
{
    herit h(20);
    h.affiche();       // ok la restriction n'empêche pas
    h.v1=rand()%100;   // erreur, v1 private à partir de h
    return 0;
}
```

Les résultats avec
`class herit : protected base`

et
`class herit : private base`

sont les mêmes. Il faut bien comprendre que la restriction effectuée commence avec la classe dérivée et ses objets. Si nous dérivons une autre classe à partir de la classe herit alors cette restriction sera sensible.

Avec base protected pour herit et herit2 dérivée de herit :

```
class herit : protected base
{
    public :
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

class herit2 : public herit
{
    public :
    herit2() : herit(){}
    herit2(int v) : herit(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit2 h(20);
    h.affiche();       // ok, v1 accessible dans la class
    h.v1=rand()%100;   // erreur, v1 inaccessible de l'objet
    return 0;
}
```

la variable v1 à partir d'un objet herit2 est bien considérée comme protected, accessible en interne à la classe mais inaccessible par un objet.

Avec base private pour herit et herit2 dérivée de herit :

```
class herit : private base
{
    public :
    herit() : base(){}
    herit(int v) : base(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

class herit2 : public herit
{
```

```
    public :
    herit2() : herit(){}
    herit2(int v) : herit(v){}

    void affiche() { cout<<"herit v1 : "<<v1<<endl;}
};

int main()
{
    herit2 h(20);
    h.affiche();       // erreur v1 private
    h.v1=rand()%100;   // erreur v1 private
    return 0;
}
```

La variable v1 de l'objet h est considérée comme private, inaccessible de herit2 et d'un objet. Il convient d'^tre méthodique avec les accès et ses réductions possibles sous peine de rendre rapidement un programme difficile à lire.

f. Héritage multiple

Une classe peut hériter de plusieurs classes simultanément. Il suffit de le préciser à la définition de la classe dérivée. Soit les classes A1, A2 et B dérivée de A1 et A2 la syntaxe est la suivante :

```
class A1 { . . . };
class A2 { . . . };
class B : [public ou protected ou private] A1, [public ou ...] A2
{
    . . .
};
```

B hérite de A1 et A2. Tout ce que nous avons vu de l'héritage est vrai avec un héritage multiple. Juste deux précisions. Dans la classe dérivée les constructeurs des classes de base peuvent être appelés en liste (séparés chacun par une virgule) et quelque soit l'ordre de la liste, l'ordre d'appel est donné par l'ordre des déclarations des classes de base, dans l'exemple ci-dessus A1 puis A2.

Pour tester voici la classe C dérivée des classes A1 et B2, avec B2 elle-même dérivée de A2. Chaque classe comporte une variable val, un constructeur et une fonction affiche() :

```
class A1
{
    public:
    int val;
    A1(int v){val = v;}
    void affiche(){cout<<"A1 : "<<val<<endl;}
};
class A2
{
    public:
    int val;
    A2(int v){val = v;}
    void affiche(){cout<<"A2 : "<<val<<endl;}
};
class B2 : public A2
{
    public:
```

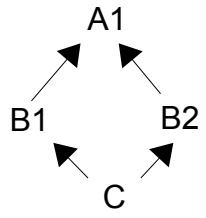
```
    int val;
    B2(int bv, int av) : A2(av){val = bv;}
    void affiche();
};
void B2::affiche()
{
    A2::affiche();
    cout<<"B2 : "<<val<<endl;
}
class C : public A1, public B2
{         // donne l'ordre de l'appel des constructeurs
          // dans la liste d'initialisation
    public :
    int val;
    C(int a1v,int a2v,int bv,int cv ) : B2(a2v,bv) , A1(a1v)
    {
          val = cv;
    }
    void affiche();
};
void C::affiche()
{
    A1::affiche();
    B2::affiche();
    cout<<"C : "<<val<<endl;
}

int main()
{
C c(1,2,3,4);
    c.affiche();
    return 0;
}
```

Le programme imprime :

```
A1 : 1
A2 : 2
B2 : 3
C : 4
```

Nous pouvons constater que c'est bien le constructeur de A1 qui est appelé premièrement avant le constructeur de B2 depuis la classe C.

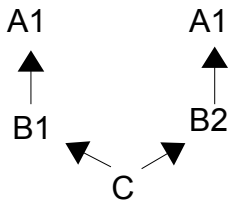
L'héritage multiple peut permettre de créer des relations transversales dans un arbre de classes. Toutefois il faut veiller à ne pas s'y perdre sachant qu'il peut devenir complexe à manipuler, qu'un certain nombre d'ambiguïtés peuvent être difficiles à repérer. Par exemple, admettons une classe de base A1 et deux classes dérivées B1 et B2, puis une classe C dérivée de B1 et B2, ça donne :



Lorsque nous déclarons un objet de la classe C combien de fois contient-il des données de la classe A1, une ou deux fois ?

Deux fois.

Le schéma est correcte pour le modèle mais en mémoire ça donne :



Nous pouvons le tester :

```
class A1
{
    public :
    int val;
    A1(int v){val=v;}
    void affiche(){cout<<"A1 : "<<val<<endl;}
};
class B1 : public A1
{
    public :
    int val;
    B1(int a, int b) : A1(a){val=b;}
    void affiche()
    {
        A1::affiche();
        cout<<"B1 : "<<val<<endl;
    }
};
class B2 : public A1
{
    public :
    int val;
    B2(int a, int b) : A1(a){val=b;}
    void affiche()
    {
        A1::affiche();
        cout<<"B2 : "<<val<<endl;
    }
};
class C : public B1, public B2
{
    public :
    int val;
    C(int a1,int b1,int a2,int b2,int c) : B1(a1,b1),B2(a2,b2)
{val=c;}
    void affiche()
    {
        B1::affiche();
        B2::affiche();
        cout<<"C  : "<<val<<endl;
    }
};
int main()
```

```
{
C c(1,2,3,4,5);
    c.affiche();
    return 0;
}
```

Le programme imprime :

```
A1 : 1
B1 : 2
A1 : 3
B2 : 4
C  : 5
```

Pour éviter ce fonctionnement et n'avoir qu'une seule fois les données et fonctions de la classe de base A1 il faut utiliser la notion de virtualité et réaliser un héritage virtuel.

g. Héritage multiple avec une base virtuelle

Au besoin, dans un cas d'héritage multiple il est possible de n'avoir qu'une fois les données de la classe de base. Pour ce faire il faut que les classes dérivées soient dérivées en virtuel avec l'instruction virtual de la façon suivante :

```
class A    // base
{
    // membres A
};
class B1 : virtual public A
{
    // membres B1
};
class B2 : virtual public A
{
    // membres B2
};
class C : public B1,public B2
{
    // membres C
};
```

Mentionner A virtual, dans un héritage en B1 par exemple, signifie que A ne devra apparaître qu'une fois en C. Si B2 n'existe pas ça ne change rien en C. Si B2 existe et n'est pas virtual il y a bien une seule fois pour B1 mais aussi de toute façon une fois pour B2. Ça fait deux en C et l'objectif n'est pas atteint. Si en revanche A est déclarée virtual en B1 et virtual en B2 alors seulement le compilateur considérera que les données de A ne doivent apparaître qu'une fois en C. C'est pourquoi il faut répéter virtual pour B1 et B2.

Le programme suivant met en oeuvre l'héritage virtuel :

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

// heritage virtuel pour non dupplication des données
// de la classe de base A dans un héritage multiple
class A
{
    public :
    int val;
```

```
    A(){val=0;}        // constructeur par défaut à ajouter
    A(int v){val=v;}
    void affiche(){cout<<"A : "<<val<<endl;}
};
class B1 :  virtual public A
{
    public :
    int val;
    B1(int a, int b) : A(a){val=b;}
    void affiche()
    {
        A::affiche();
        cout<<"B1 : "<<val<<endl;
    }
};
class B2 :  virtual public A
{
    public :
    int val;
    B2(int a, int b) : A(a){val=b;}
    void affiche()
    {
        A::affiche();
        cout<<"B2 : "<<val<<endl;
    }
};
class C : public B1, public B2
{
    public :
    int val;
    C(int a1,int b1,int a2,int b2,int c) : B1(a1,b1),B2(a2,b2)
{val=c;}
    void affiche()
    {
        B1::affiche();
        B2::affiche();
        cout<<"C  : "<<val<<endl;
    }
};
int main()
{
C c(1,2,3,4,5);

    c.affiche();            // 0 2 0 4 5
    cout<<"-----------"<<endl;
    c.A::val=9;             // 9 2 9 4 5
    c.affiche();
    return 0;
}
```

Le premier affichage de l'objet c donne :

```
A : 0
B1 : 2
A : 0
B2 : 4
C : 5
```

et le second :

```
A : 9
B1 : 2
A : 9
```

```
B2 : 4
C : 5
```

Le second montre bien, avec une affectation de la valeur 9, que la variable val de la base est bien unique. Mais le premier met en évidence un problème avec les constructeurs. Le compilateur ne sait pas choisir entre l'appel du constructeur de A en B1 ou en B2. Tel que nous avons écrit le programme les appels constructeurs de A en B1 et B2 sont ignorés. En fait c'est le constructeur par défaut de A qui est appelé en C. En effet dans le cas ou une classe petite fille hérite de plusieurs classes filles déclarées virtual, C++ autorise et nécessite un appel constructeur de la classe mère (base) dans la classe petite fille. Voici le programme avec des constructeurs adaptés :

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

class A
{
    public :
    int val;
    A(){val=0;}
    A(int v){val=v;}
    void affiche(){cout<<"A : "<<val<<endl;}
};
class B1 :  virtual public A
{
    public :
    int val;
    B1(int b){val=b;}
    B1(int a, int b) : A(a){val=b;}
    void affiche(){cout<<"B1 : "<<val<<endl;}
};
class B2 :  virtual public A
{
    public :
    int val;
    B2(int b){val=b;}
    B2(int a, int b) : A(a){val=b;}
    void affiche(){cout<<"B2 : "<<val<<endl;}
};
class C : public B1, public B2
{
    public :
    int val;
    C(int a,int b1,int b2,int c) : A(a),B1(b1),B2(b2){val=c;}
    void affiche()
    {
        A::affiche();
        B1::affiche();
        B2::affiche();
        cout<<"C  : "<<val<<endl;
    }
};
int main()
{
C c(1,2,3,4);

    c.affiche();            // 1 2 3 4
    return 0;
}
```

Les fonctions d'affichage sont également modifiées, chaque classe affiche uniquement sa valeur val sauf la classe C qui affiche toutes les valeurs val. Le programme ci-dessus imprime :

```
A : 1
B1 : 2
B2 : 3
C : 4
```

8. Pointeurs polymorphes, virtualité

En C++ est apparaît la notion de typage dynamique également appelée "polymorphisme". L'idée est qu'un pointeur puisse changer de type en fonction de l'objet sur lequel il pointe. C'est à dire qu'il soit possible avec un pointeur de reconnaître dynamiquement un objet afin d'accéder à ses membres. Cette propriété est introduite en C++ pour des fonctions redéfinies dans des classes dérivées. Elle permet d'introduire des listes d'objets de type différents dans les programmes. Typiquement c'est un tableau de pointeurs dans lequel chacun non seulement pointe sur un objet différent mais surtout peut sans cast accéder à certaines fonctions de cet objet. Ce changement important obéit à des règles strictes que nous allons explorer et vérifier.

a. Accès pointeurs par défaut aux fonctions redéfinies

Soit une classe A et une classe B dérivée de A :

```
class A { . . . }
class B : public A { . . . }
```

Par défaut un pointeur de type A* auquel est affectée une adresse de type B* n'accède que aux données et fonctions membres de A :

```
B b;
A *ptr = &b;     // ptr reste de type A*
```

Le programme suivant permet de le constater :

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

class A
{
    public :
    int val;
    A(){val=0;}
    A(int v):val(v){}
    void affiche(){cout<<"A : "<<val<<endl;}
};
class B :  public A
{
    public :
    int val;
    B(int a, int b) : A(a),val(b){}
    void affiche(){cout<<"B : "<<val<<endl;}
};

int main()
{
A *ptr;
B b(1,2);
```

```
    ptr=&b;
    ptr->affiche();   // affiche de A

    return 0;
}
```

Le programme affiche :

```
A : 1
```

Quoique le pointeur a contienne l'adresse d'un objet B.

Mais il y a en C++ une possibilité pour que ce pointeur accède néanmoins à des fonctions membres redéfinies en B. Il faut que ces fonctions soient redéfinies en B et déclarées virtual en A.

b. Accès pointeur aux fonctions virtuelles redéfinies

Le pointeur de type A* de l'exemple précédent peut accéder à des fonctions de B si :

- ces fonctions sont des redéfinitions de fonctions de A
- les fonction initiales en A sont déclarées virtuelles avec l'instruction virtual

Voici le programme précédent modifié : la fonction affiche() de A est déclarée virtuelle :

```
class A
{
    public :
    int val;
    A(){val=0;}
    A(int v):val(v){}
    virtual void affiche(){cout<<"A : "<<val<<endl;}
};
class B :  public A
{
    public :
    int val;
    B(int a, int b) : A(a),val(b){}
    void affiche(){cout<<"B : "<<val<<endl;}
};

int main()
{
A *ptr;
B b(1,2);

    ptr=&b;
    ptr->affiche();   // affiche de B

    return 0;
}
```

le programme affiche :

```
B : 2
```

Avec une fonction initiale déclarée virtuelle dans une classe, la détermination de la fonction effective, lors d'un appel via un pointeur d'objet, se fait à l'exécution en fonction du type de l'objet dont le pointeur a l'adresse. Ici le type de l'objet dont le

pointeur a contient l'adresse est B et comme la fonction affiche() de A est virtuelle, c'est la fonction affiche() de B qui est appelée au moment de l'exécution.

Remarque

Cette notion de virtualité des fonctions peut s'appliquer à un destructeur afin de pouvoir libérer la mémoire de l'objet réellement pointé par un pointeur :

```
class A
class A
{
    public :
    int val;
    A(){val=0;}
    A(int v):val(v){}
    virtual void affiche(){cout<<"A : "<<val<<endl;}

    virtual~A(){cout<<"destructeur A"<<endl;}
};
class B :  public A
{
    public :
    int val;
    B(int a, int b) : A(a),val(b){}
    void affiche(){cout<<"B : "<<val<<endl;}

    ~B(){cout<<"destructeur B"<<endl;}
};

int main()
{
A *ptr;
A a(1);
B b(2,3);
    ptr=&a;
    ptr->affiche();   // affiche de A
    delete ptr;       // destructeur A

    ptr=&b;
    ptr->affiche();   // affiche de B
    delete ptr;       // destructeur de B puis de A
    system("PAUSE");

    return 0;
}
```

Le programme imprime :

```
A : 1
destructeur A
B : 3
destructeur B
destructeur A
appuyez sur une touche pour continuer...
destructeur B
destructeur A
destructeur A
```

Après l'instruction system("PAUSE"); qui permet d'arrêter le programme nous pouvons constater que les destructeurs sont appelés automatiquement à l'issu du bloc pour tous les objets qui y ont été déclarés.

c. Intérêt des accès pointeurs aux fonctions virtuelles

L'apport important de ce concept est de permettre de constituer des listes d'objets différents. Par exemple :

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

class A
{
    public :

    virtual void affiche(){cout<<"A";}
};
class B : public A
{
    public :
    void affiche() {cout<<"B";}
};
class C : public A
{
    public :
    void affiche() {cout<<"C";}
};
class D : public A
{
    public :
    void affiche() {cout<<"D";}
};
class E : public A
{
    public :
    void affiche() {cout<<"E";}
};
int main()
{
A*tab[20];

    for (int i=0; i<20; i++){

        switch(rand()%4){
            case 0 : tab[i]=new B; break;
            case 1 : tab[i]=new C; break;
            case 2 : tab[i]=new D; break;
            case 3 : tab[i]=new E; break;
        }
    }
    for (int i=0; i<20; i++)
        tab[i]->affiche();
    cout<<endl;
    return 0;
}
```

Dans ce programme nous dérivons de la classe de base A les classes B, C, D, E. Toutes les classes comportent uniquement une fonction redéfinie, la fonction

affiche() déclarée en virtuel dans la classe de base. A partir de là, dans le main() nous déclarons un tableau de pointeurs sur la classe de base A. Nous initialisons ensuite ce tableau avec au hasard pour chaque pointeur l'adresse d'un objet de type B, C, D ou E. Pour finir nous pouvons afficher correctement chaque objet du tableau et tout le tableau avec une simple boucle. Cette technique peut être très utile s'il s'agit de gérer simultanément des objets de types différents. Par exemple un système graphique d'interface avec des objets bouton, menus contextuels, déroulants, des listes etc. le tout devant être réuni et contrôlé en même temps.

d. Classes abstraites, fonctions virtuelles pures

Dans l'exemple précédent la fonction virtuelle de la classe de base A est uniquement prétexte à des redéfinitions dans les classes dérivées et ses instructions dans la classe de base ne servent pas. De plus la classe de base sert uniquement à dériver des classes filles et n'est pas utilisée autrement, il n'y a pas d'objet de la classe A. C'est en fait le principe des classes abstraites. En voici les caractéristiques :

- Une classe abstraite est une classe de base qui va servir uniquement à dériver des classes complémentaires.
- Elle ne permet pas d'obtenir des objets
- Elle contient des fonctions virtuelles dites "pures" c'est à dire uniquement les déclarations des fonctions mais pas leurs définitions. Les définitions sont faites dans chacune des classes dérivées.
- La déclaration d'une fonction virtuelle pure se fait en ajoutant = 0 après. Par exemple :
  ```
  virtual void affiche() = 0;
  ```
- Pour obtenir une classe abstraite il suffit de déclarer une fonction virtuelle pure dedans.

A partir de l'exemple précédent nous pouvons transformer la classe de base A en une classe abstraite :

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

class A
{
   public :
   virtual void affiche()=0;  // pas de définition dans la classe
                              // de base A
};

class B : public A
{
    public :
    void affiche();
};
void B::affiche()      // définition en B
{
   cout<<"B";
}

( . . . )              // idem pour C, D, E

int main()
```

```
{
// A a;    // erreur, classe abstraite, objets impossibles

A*tab[20];

    for (int i=0; i<20; i++){

        switch(rand()%4){
            case 0 : tab[i]=new B; break;
            case 1 : tab[i]=new C; break;
            case 2 : tab[i]=new D; break;
            case 3 : tab[i]=new E; break;
        }
    }
    for (int i=0; i<20; i++)
        tab[i]->affiche();
    cout<<endl;


    return 0;
}
```

La fonction affiche() est une fonction virtuelle pure ce qui fait de la classe A une classe abstraite.

Dans la perspective d'utiliser le polymorphisme pour contrôler des objets différents à partir d'un tronc commun de fonctions ayant les mêmes noms, l'intérêt d'une classe abstraite est de normaliser et regrouper ces contrôles. Éventuellement des variables peuvent également être associées au tronc commun des fonctions dans la classe de base. Les fonctions virtuelles pures constituent l'interface des contrôles communs à toutes les classes dérivées. Voici à titre illustratif, l'amorce d'un programme susceptible de gérer un ensemble de formes géométriques.

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

enum{POINT, LIGNE, RECTANG, CERCLE };

class graph
{
    protected :
    int x1, y1, color, lettre;

    graph() : x1(0),y1(0),color(0),lettre(' '){}

    public :
    virtual void dessine()=0;
    virtual int type()=0;    // retourne le type de la forme
};
/*************************/
class point : public graph
{
    public :
    point() : graph(){}

    int type() { return POINT; };// retourne le type de la forme
    void dessine();
};
void point::dessine()
```

```
{
    cout<<"POINT : "<<endl;
    cout<<"\tx1 : "<<x1<<endl;
    cout<<"\ty1 : "<<y1<<endl;
    cout<<"\tcolor : "<<color<<endl;
    cout<<"\tlettre : "<<lettre<<endl;
    cout<<"--------------"<<endl;
}
/*************************/
class ligne : public graph
{
    private :
    int x2,y2; // fin ligne

    public :
    ligne() : graph(),x2(0),y2(0){}

    int type() { return LIGNE; };
    void dessine();
};
void ligne::dessine()
{
    cout<<"LIGNE : "<<endl;
    cout<<"\tx1 : "<<x1<<endl; // début ligne
    cout<<"\ty1 : "<<y1<<endl;
    cout<<"\tx2 : "<<x2<<endl; // fin ligne
    cout<<"\ty2 : "<<y2<<endl;
    cout<<"\tcolor : "<<color<<endl;
    cout<<"\tlettre : "<<lettre<<endl;
    cout<<"--------------"<<endl;
}
/*************************/
class rect : public graph
{
    private :
    int x2,y2; // coin bas droite rect

    public :
    rect() : graph(),x2(0),y2(0){}

    int type() { return RECTANG; };
    void dessine();
};
void rect::dessine()
{
    cout<<"RECTANGLE : "<<endl;
    cout<<"\tx1 : "<<x1<<endl; // coin haut gauche rect
    cout<<"\ty1 : "<<y1<<endl;
    cout<<"\tx2 : "<<x2<<endl; // coin bas droite rect
    cout<<"\ty2 : "<<y2<<endl;
    cout<<"\tcolor : "<<color<<endl;
    cout<<"\tlettre : "<<lettre<<endl;
    cout<<"--------------"<<endl;
}
/*************************/
class cercle : public graph
{
    private :
    int r;      // rayon du cercle
```

```
    public :
    cercle() : graph(),r(0){}

    int type() { return CERCLE; };
    void dessine();
};
void cercle::dessine()
{
    cout<<"CERCLE : "<<endl;
    cout<<"\tx1 : "<<endl;  // position du centre
    cout<<"\ty1 : "<<endl;
    cout<<"\tr  : "<<endl;  // le rayon
    cout<<"\tcolor : "<<endl;
    cout<<"\tlettre : "<<endl;
    cout<<"-------------"<<endl;
}
/*************************/

int main()
{
graph **all;
int nb;
    srand(time(NULL));

    nb=1+rand()%20;
    all=new graph*[nb];

    for (int i=0; i<nb; i++)
        switch(rand()%4){
            case 0 : all[i] = new point();  break;
            case 1 : all[i] = new ligne();  break;
            case 2 : all[i] = new rect();   break;
            case 3 : all[i] = new cercle(); break;
        }

    for (int i=0; i<nb; i++)
        all[i]->dessine();

    switch(rand()%nb){
        case POINT :    cout<<"trouve un point"<<endl;     break;
        case LIGNE :    cout<<"trouve une ligne"<<endl;    break;
        case RECTANG :  cout<<"trouve un rectangle"<<endl; break;
        case CERCLE :   cout<<"trouve un cercle"<<endl;    break;
    }

    return 0;
}
```

La classe abstraite graph détermine une interface composée des fonctions dessine() et type() ainsi que qu'une base de variables communes à tous les types de formes prévus. Chaque type dérivée de forme ajoute ces propres variables et ses redéfinitions des fonctions de contrôles dessine() et type().

9. Classes et fonctions "amies" (friend)

Une classe ou une fonction déclarée "amie" d'une classe peut accéder à toutes les données private ou protected de cette classe. Ce procédé permet de lever un interdit d'accès lorsque c'est nécessaire pour des raisons d'architecture afin

d'accorder des droits entre des classes internes. En principe ça ne devrait pas être utile et à priori c'est à manier avec précaution, voire plutôt à éviter. Mais le moyen est là et nous allons le tester.

Pour déclarer dans une classe une fonction amie il suffit de faire précéder la déclaration de cette fonction par l'instruction friend dans cette classe.

```
class A
{
    // ...
    friend void fonc()  // fonct amie de A
    // ...
};
```

Pour déclarer une classe amie dans une autre classe il faut déclarer cette classe précédée de l'instruction friend dans l'autre classe :

```
class A
{
    // ...
    friend class B;  // B amie de A
    // ...
};
```

Fonction et classe amies de la classe A ont accès aux fonctions et données privates de la classe A :

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

class A
{
    // déclaration d'une fonction amie qui
    // peut accéder à toutes les données
    // private ou protected de la classe A
    friend void fonct();

    // déclaration d'une classe amie peut
    // accéder à toutes lesdonnées private
    // ou protected de la classe A
    friend class B;

    private :
    int val;
    void affich(){cout<<"A : "<<val<<endl;}

    public :
    A(){val=0;}
    A(int v){val=v;}

    // ...
};

void fonct()
{
A a(5);
    cout<<"fonct accede a val private : "<<a.val<<endl;
}

class B
{
    public :
```

```
    A a;
    B(int n){a.val=n;}
    void affich()
    {
        cout<<"B accede a fonction private : ";
        a.affich();
        cout<<"B accede a val private : "<<a.val<<endl;

    }
};

int main()
{
    fonct();

    B b(10);
    b.affich();

    return 0;
}
```

Le programme imprime :

```
fonct accede a val private : 5
B accede a fonction private : A : 10
B acced a val private : 10
```