Matthieu GRALL

Cours de C / C++

Partie 2

Attention : ce document est un complément aux séances de cours et de travaux pratiques.

Table des matières

Les notions abordées dans le présent document et celui qui le précède permettent de bien comprendre le langage. Il est cependant nécessaire de s'exercer afin que celles-ci deviennent familières. D'autre part, il ne faut pas oublier que le C++ est un vaste langage dont on ne fait pas le tour rapidement ! En effet, de nombreuses choses n'ont pas été étudiées ici, et c'est à chacun de découvrir ce qui l'intéresse (par exemple le traitement des fichiers, les interfaces, l'exploitation du préprocesseur, les différents environnements de développement, …).

1 La manipulation des pointeurs

1.1 Généralités

Un pointeur est une variable qui peut stocker la valeur d'une adresse en mémoire.

1.1.1 Exemple de déclaration et d'affectation

```
int * p ;   // déclaration de p, un pointeur sur une donnée de type int
int i=4 ;   // déclaration et affectation de la variable i de type int
p = &i ;    // affectation de l'adresse de i à p
cout<<(*p); // p est dépointé (indirection) et la valeur correspondant à
            // l'adresse est affichée
```

1.1.2 Pointeur sur agrégat : notation fléchée

La notation fléchée permet d'utiliser un attribut de ce que pointe le pointeur :

```
pe->c='A' ; // affectation de 'A' à l'attribut c de ce que pointe pe
            // (afin de modifier l'attribut c d'un agrégat, on se « place »
            // sur cet agrégat et non sur son adresse, donc on dépointe pe)
```

1.1.3 Pointeur sur objet cible d'une méthode de classe : this

```
void Point::afficher(){
      cout << "adresse : " << this ;            // pointeur sur cible
      cout << ", abscisse : " << this->x ;      // x de la cible
      cout << ", ordonnée : " << this->y << endl; // y de la cible
}
```

1.2 Le passage de paramètres

Dans les différentes fonctions et procédures, on peut passer en paramètres des données que l'on souhaitera modifier et d'autres qui ne devront pas changer. Ces données sont de taille plus ou moins importante, et on peut choisir de ne passer en paramètres que les adresses de celles-ci, et non les données elles-mêmes.

```
void g(int i1, int *i2, int &i3){
      i1++ ;      // i1 est incrémenté              → en donnée
      *i2++ ;     // x2 est incrémenté par pointeur  → en résultat
      i3++ ;      // x3 est incrémenté par référence → en résultat
}

void f(){
      int x1, x2, x3 ;
      x1=1 ;
      x2=2 ;
      x3=3 ;
      g(x1, &x2, x3) ; // les paramètres sont x1, un pointeur sur x2, et x3
}
```

Ici, lorsque g est appelée, une copie du contenu de la variable x1 est transmise à la fonction. A l'intérieur de la fonction, seule cette copie est altérée, et x1 garde sa valeur. Le second argument est un pointeur sur la variable x2, et puisqu'on incrémente la valeur de la variable pointée par celui-ci, x2 change. La transmission de x3 sous forme de référence n'est possible qu'en C++. i3 est traité comme i1, mais x3 est changé quand on modifie i3.

1.3 Pointeurs arithmétiques

Sachant que les pointeurs sont les adresses de données de différents types, et que ces types ont des tailles qui diffèrent, le pointeur du premier élément d'un tableau doit être incrémenté d'un nombre d'octets correspondant à la taille des données pour pointer sur l 'élément suivant. Pour pointer sur un élément quelconque en utilisant une notation de type pointeur, on utilise le pointeur arithmétique.

Incrémenter un pointeur consiste à en augmenter sa valeur de la taille de la donnée correspondante. Par exemple, si *ptr_int* pointe sur un élément de tableau de type *int*, l'instruction *ptr_int++* incrémente la valeur de ce pointeur de 2 pour qu'il pointe sur l'élément suivant (l'instruction *ptr_int += 4* augmente de 8 la valeur du pointeur pour qu'il pointe 4 éléments plus loin). La décrémentation suit le même principe.

Ainsi, la soustraction d'un pointeur par un autre dans un tableau donne le nombre d'éléments les séparant.

1.4 La gestion dynamique de la mémoire

Elle permet de ne réserver aux zones de données d'un programme que l'espace strictement nécessaire. De plus, elle permet d'en faire évoluer la taille, et évite de surdimensionner ou de sousdimensionner la taille réservée.

L'ensemble de la mémoire disponible est divisée en trois zones :
- la zone des données statiques et du code exécutable ;
- la zone réservée pour la pile du programme (variables locales, appels de fonctions) : stack ;
- la zone des allocations dynamiques : heap (tas).

1.4.1 Allocation dynamique : l'opérateur new

```
int * pi ;
pi = new int ;
```

L'opérateur *new* demande l'ollocation d'un espace mémoire nécessaire pour stocker un élément de type *int*. Il affecte ensuite à *pi* l'adresse correspondante, ou un pointeur nul (constante NULL ou NIL ou 0L) en cas d'échec.

1.4.2 Exemple d'allocations de blocs jusqu'au débordement

```
#include <iostream.h>
void main(){
        long taille;                    // taille du tableau
        int *pint;                      // pointeur sur entier
        int nbloc;                      // nombre de blocs
        cout<<"taille : ";
        cin>>taille;
        for(nbloc=1;;nbloc++){          // boucle sans fin
                pint=new int[taille];   // demande d'allocation pour un
                                        // tableau d'entiers
                if(pint)                // adresse non nulle
                        cout<<endl<<"bloc "<<nbloc;
                else{                   // adresse nulle
                        cout<<endl<<"IMPOSSIBLE";
                        break;          // sortie de la boucle
                }
         }
}
```

Exemple de résultat d'exécution (dépend de la taille de la mémoire disponible) :

```
taille : 5000000
bloc 1
bloc 2
bloc 3
IMPOSSIBLE
```

Remarque : certains environnements ou systèmes quitteront eux-mêmes l'exécution s'il y a un problème d'allocation.

1.4.3 Libération de la mémoire : l'opérateur delete

```
delete pi ;
```

On essaie de libérer la mémoire dès qu'on n'a plus besoin d'un élément. On remet donc à la disposition du système la zone mémoire dont l'adresse est *pi*.

Syntaxe pour libérer un élément simple : `delete adresse`
Syntaxe pour libérer un tableau d'éléments : `delete [] adresse`

1.4.4 Gérer les débordements de mémoire : set_new_handler

```
#include <iostream.h>   // pour cout et cin
#include <new.h>        // pour set_new_handler
#include <stdlib.h>     // pour exit

void deborde();         // déclaration de la fonction deborde

void main(){
      long taille;
      int *pint;
      int nbloc;

      set_new_handler(&deborde);
      // à partir de maintenant, tout manque de mémoire lancera deborde
      // (ne fonctionne pas sous tous les systèmes)

      cout<<"taille : ";
      cin>>taille;

      for(nbloc=1;;nbloc++){
            pint=new int[taille];
            cout<<endl<<"bloc "<<nbloc;
      }
}

// fonction appelée automatiquement en cas de manque de mémoire
void deborde(){
      cout<<endl<<"IMPOSSIBLE";
      exit(1);
}
```

2 Les chaînes de caractères et les entrées / sorties

2.1 A retenir

- Pour représenter des chaînes de caractères, on peut considérer *char ** comme synonyme de *char[]*.
- Le premier caractère de la chaîne porte l'indice 0.
- Un caractère est écrit entre cotes ('c'), une chaîne de caractères est écrite entre guillements ("c").
- Une chaîne de caractères est une suite de caractères terminée par le caractère '\0', la marque de fin.

2.2 Quelques fonctions usuelles de string.h

2.2.1 Longueur utile : strlen

```
int strlen (const chat * tc) ;
```

→ compte le nombre de caractères jusqu'à '\0'.

2.2.2 Comparaison de deux chaînes : strcmp

```
int strcmp (const char * s1, const char * s2) ;
```

→ renvoie :

–1 si la chaîne s1 est avant s2 (ordre alphabétique),
0 si les deux chaînes sont identiques,
1 si la chaîne s1 est après s2.

2.2.3 Copie : strcpy

```
char * strcpy (char * td, const char * ts) ;
```

→ copie de la chaîne source *ts* vers la chaîne *td*, aucune vérification sur la taille de la mémoire associée à *td*

2.2.4 Ajout en fin de chaîne : strcat

```
char * strcat (char * td, const char * ts) ;
```

→ concaténation de la chaîne source *ts* à la fin de la chaîne *td*, aucune vérification sur la taille de la mémoire associée à *td*

Pensez à éviter le débordement :

```
char td[MAXT] ;
if(strlen(td)+strlen(ts<MAXT)
      strcat(td, ts) ;
```

2.3 Entrées / sorties

Dans les exemples suivants, on utilise une chaîne déclarée ainsi :

```
char tc[8] ;
```

2.3.1 Saisie

2.3.1.1 Saisie formatée avec débordement : cin

```
cin << tc ;
```

→ on tape " ab ®" tc vaut "abø "
→ on tape "abcdefghij®" tc vaut "abcdefgh"

2.3.1.2 Saisie non formatée sans débordement : cin.get

```
cin.get(tc,sizeof(tc)) ;
```

→ on tape " ab ®" tc vaut " ab ø "
→ on tape "abcdefghij®" tc vaut "abcdefgø"

2.3.1.3 Saisie formatée sans débordement : cin.width

```
cin.width(sizeof(tc)) ; cin >> tc ;
```

→ on tape " ab ®" tc vaut "abø "
→ on tape "abcdefghij®" tc vaut "abcdefgø"

2.3.1.4 Définition de la base pour interpréter des entiers : cin.setf et basefield

```
int i1, i2, i3 ;
// en hexadécimal
cin.setf(ios::hex,ios::basefield) ;
cin >> i1 ;
// en décimal
cin.setf(ios::dec,ios::basefield) ;
cin >> i2 ;
// en octal
cin.setf(ios::oct,ios::basefield) ;
cin >> i3 ;
```

2.3.2 Affichage

On utilise une chaîne tc :

```
strcpy (tc, "abc") ;
```

2.3.2.1 Affichage normal : cout

```
cout << ':' << tc << ':' ;    // :abc:
```

2.3.2.2 Définition de la largeur : cout.width

```
// sur 7 colonnes, cadré à droite (par défaut ios::right)
cout << ':' ;
cout.width(7) ;
cout << tc << ':' ;           // :    abc:
```

2.3.2.3 Définition du cadrage : cout.setf et adjustfield

```
// cadré à gauche
cout.setf(ios::left,ios::adjustfield) ;
cout << ':' ;
cout.width(7) ;
cout << tc << ':' ;           // :abc    :
// cadré à droite
cout.setf(ios::right,ios::adjustfield) ;
cout << ':' << tc << ':' ;    // :abc    :
```

2.3.2.4 Définition de la base pour afficher des entiers : cout.setf et basefield

```
int i1=10,i2=11,i3=12;
// en hexadécimal
cout.setf(ios::hex,ios::basefield) ;
cout << i1 << endl ;    // 10→a
// en décimal
cout.setf(ios::dec,ios::basefield) ;
cout << i2 << endl ;    // 11→11
// en octal
cout.setf(ios::oct,ios::basefield) ;
cout << i3 << endl ;    // 12→14
```

2.3.2.5 Définition de la précision pour les réels : cout.precision

```
double d=987.654321 ;
cout.precision(4) ;     // par défaut 6 chiffres
cout << d ;             // 987.7
```

2.3.2.6 Définition de la notation scientifique : cout.setf et floatfield

```
double d=987.654321 ;
cout.setf(ios::scientific,ios::floatfield) ;   // par défaut ios::fixed
cout << d ;             // 9.876543e+002
```

2.3.2.7 Affichage avec un caractère de remplissage : cout.fill

```
int i=4 ;
cout.fill('#');
cout.width(6);
cout<<i ;               // #####4
```

2.4 Programme de test

Copiez, compilez et exécutez ce programme, puis tapez " Reine 123®" (avec les espaces).

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

// Procédure d'affiche de la chaîne
void afficher(char * t){
      cout << "tc=|";
      for(int i=0;i<8;i++)
            cout << t[i];
      cout << "| longueur=" << strlen(t) << endl;
}

// Programme principal
void main(){
      char tc[7+1];          // 7 caractères + caractère '\0'

      // Initialisation
      for(int i=0;i<8;i++)
            tc[i]=':';
      afficher(tc);          // |::::::::|

      // Saisie
      cin.width(sizeof(tc));
      cin >> tc;             // on tape "   Reine 123®"
      afficher(tc);          // |Reineø::|

      // Copie d'une autre chaîne sur la chaîne
      strcpy(tc,"fin");
      afficher(tc);          // |finøeø::|

      // Changement du quatrième caractère (tc[3])
      tc[3]='z';
      afficher(tc);          // |finzeø::|

      // Concaténation d'une autre chaîne sur la chaîne
      strcat(tc,"zz");
      afficher(tc);          // |finzezzø|
}
```

2.5 Allocation d'une suite de chaînes de caractères

Contrairement à un tableau dont la taille est fixe (donc en général trop petite ou trop grande), on va créer un tableau dynamique de pointeurs sur des chaînes de caractères dynamiques :

```
#include <iostream.h>   // pour cout, cin
#include <stdlib.h>     // pour exit
#include <string.h>     // pour strlen, strcpy
#include <new.h>        // pour set_new_handler

const int MAX = 256;

// Déclaration des fonctions
char * saisieNom();
void deborde();

// Main
void main(){

      // Déclaration des variables
      int i;
      char **pnoms; // pointeur sur le tableau de chaînes de caractères
      int nbnoms;

      // Gestion des manques de mémoire
      // (ne fonctionne pas sous tous les systèmes)
      set_new_handler(&deborde);

      // Allocation du tableau de pointeurs
      cout << "Nombre de noms à rentrer : ";
      cin >> nbnoms;
      cin.get();  // permet une saisie « propre » (avec retour-chariot)
      pnoms = new char *[nbnoms];

      // Saisies et affichage
      for(i=0;i<nbnoms;i++)
            pnoms[i]=saisieNom();
      cout << endl ;
      for(i=0;i<nbnoms;i++)
            cout << pnoms[i] << endl;
}

// Fonction de saisie d'un nom
char * saisieNom(){
      char nom[MAX];
      char * pnom;      // une chaîne de caractères

      cout << "Entrez le nom : ";
      cin.getline(nom, MAX);  // permet la saisie jusqu'au retour-chariot
      pnom = new char [strlen(nom)+1];
      strcpy(pnom,nom);       // copie de nom dans la zone pointée par pnom
      return (pnom);
}

// Procédure gérant les manques de mémoire
void deborde(){
      cout << "Allocation impossible" << endl;
      exit(1);
}
```

3 Programmation Orientée Objet (C++)

3.1 Classes

3.1.1 Généralités

L'utilisation des classes permet d'obtenir une parfaite modularité des différents objets qu'on veut manipuler. Chaque classe est une sorte de « moule à objets », qui permet de générer des **instances** possédant toutes les mêmes propriétés. Une classe contient des **attributs** (ou données, ou champs), et des **méthodes** (ou fonctions). Ce sont les **membres** de la classe :

```
class X{
      int x1 ;    // Attributs
      int x2 ;
      …
      int f3(…) ; // Méthodes (souvent uniquement les prototypes)
      int f4(…) ;
} ;

// Définitions des méthodes, le symbole '::' précisant le rattachement
// d'une fonction à une classe

int X::f3(…){
      …
}

int X::f4(…){
      …
}
```

3.1.2 Visibilité des membres d'une classe : public, private, protected

```
class X{
      int x1 ;
public :
      int x2 ;
private :
      int f3() ;
protected :
      int x4 ;
public :
      int x5 ;
      int f6() ;
} ;
```

Ici, x1 et f3 sont **privés**, x4 est **protégé**, et x2, x5 et f6 sont **publics**. La différence entre *class* et *struct* est que, dans le premier cas les membres sont par défaut privés, alors que dans le second ils sont publics par défaut.
Les membres publics sont accessibles librement par tous, comme en C. Les membres privés ou protégés (confondus pour l'instant) sont accessibles uniquement à l'intérieur des méthodes de la classe. Ainsi, un programme utilisant la classe *X* n'aura pas le droit d'accéder directement à *x1*.

3.1.3 Fonctions amies : friend

Le modèle de visibilité est parfois un peut trop rigide. Il n'est pas toujours pratique de tout faire avec des méthodes. On peut dans ce cas déclarer en même temps que le classe des fonctions amies, qui auront les mêmes droit d'accès que les méthodes de la classe, alors qu'elles ne font pas partie de la classe.

```
class X{
      int x1 ;                // implicitement privé
      …
      friend … f(…) ;         // fonction f n'appartenant pas à la classe X
      friend … Y::g(…) ;      // fonction g d'une autre classe Y
} ;
```

Ici, f et g auront accès au champ privé x1 des objets X. On peut aussi déclarer d'un seul coup que toutes les méthodes d'une classe sont amies :

```
class X{
      …
      friend class Y ;
} ;
```

3.1.4 Fonctions inline

Habituellement, un appel de fonction est transformé par le compilateur en un branchement de langage machine. Le déroulement du programme est donc détourné, puis après exécution du corps de la fonction, le contrôle est rendu à l'instruction qui suit l'appel.

Lorsqu'une fonction est désignée comme *inline*, le compilateur ne génère pas de branchement mais recopie carrément en langage machine la définition de la fonction. L'exécution s'en trouve ainsi accélérée, et l'amélioration des performances est surtout appréciable dans les boucles qui font souvent appel à des fonctions. Mais l'inconvénient est qu'une fonction *inline* se trouve dupliquée, elle prend donc beaucoup de place. C'est pourquoi il ne faut pas en abuser !

Une fonction peut être désignée comme *inline* de deux façons :

- en mettant le mot-clé *inline* juste avant la définition de la fonction :

```
inline void f1(){…}

void f(){
      int i ;
      for(i=0;i<1000;i++)
            f1() ;      // appel inline
      …
}
```

- en définissant complètement une fonction membre à l'intérieur de sa classe :

```
class abc{
      int i ;
      char c ;
      void g1() {
            i=0 ;
            c='m' ;
      }                 // g1 est inline
      void g2() ;       // g2 n'est pas inline
} ;

void abc::g2(){
      i=1 ;
      c='k' ;
}
```

3.1.5 Membres statiques

Lorsque des instances (ou objets) d'une classe sont définis, ils reproduisent en mémoire la structure complète des attributs (ou données, ou champs). Les méthodes (ou fonctions) ne sont mémorisées qu'une seule fois et sont accessibles à toutes les instances (objets) de la classe.

Dans certains cas, il n'est cependant pas souhaitable qu'une variable soit redéfinie pour chaque instance d'une classe. On définit alors des variables membres statiques, créées une seule fois pour toute la classe et disponible par toutes les instances de la classe. Ces variables doivent être initialisées en dehors de la classe.

Les méthodes peuvent aussi être statiques. Elles ne peuvent accéder qu'aux variables statiques. Les appels à ces fonctions peuvent se faire avec une instance, ou avec l'indication de la classe et de l'opérateur '::', pour le même effet (l'instance n'a pas d'importance).

```
class abc{
      int i ;
      char c ;
      void g2() {…}
public :
      static int nombreTotal ;
      void g1() ;
      static void g2(){
            nombreTotal++ ;
      }
} ;

void abc::g1(){
      nombreTotal++ ;
      …
}

int abc::nombreTotal=0 ;// initialisation

void f1(){
      abc objet1, objet2 ;
      objet1.g1() ;
      objet2.g1() ;     // tous les appels incrémentent la même variable
      …
}

void f2(){
      abc objet1, objet2 ;
      objet1.g2() ;
      abc::g2() ;       // g2 étant statique, on peut écrire les deux
      …
}
```

3.1.6 Constructeur et destructeur

3.1.6.1 Le constructeur

Pour les types de données usuels, il existe des routines prévues pour réserver de la mémoire et remplir éventuellement les variables de valeurs nulles. Mais si le programmeur définit ses propres types de données, les objets correspondants sont traités comme les variables possédant les types de base, les membres étant initialisés comme des variables indépendantes. Il est cependant normal de vouloir garder entièrement le contrôle de ses propres structures de données.

Appelé automatiquement à l'instanciation d'un objet, le constructeur porte le même nom que la classe et ne retourne aucune valeur. Il peut comporter des paramètres et des valeurs par défaut.

3.1.6.2 Le constructeur par recopie

Il est appelé lors de la définition d'un objet qui doit être initialisé avec le contenu d'un autre objet de la même classe (affectation, passage en paramètre dans une fonction). Il doit avoir pour argument une référence sur un objet de sa classe, et porte le même nom que la classe.

3.1.6.3 Le destructeur

Appelé à la fin de la vie de chaque instance (fin du bloc où elle a été déclarée par exemple), il permet de restituer la place mémoire occupée et exécuter éventuellement d'autres actions. Tout comme le constructeur, cette fonction est définie implicitement pour chaque classe. Le nom du destructeur est celui de la classe précédé du symbole '~'. Il ne retourne aucune valeur et ne comporte aucun argument.

3.1.7 Comment créer une classe

Bien qu'il soit possible de créer ses classes dans le même fichier qu'un programmes ou autres fonctions, on retrouve, en règle générale, le découpage suivant :

- Un fichier *nom_classe.h* : spécifications de la classe, où seuls les prototypes des méthodes sont écrits :

```
#ifndef NOM_CLASSE // évite la relecture des informations si elles ont déjà
#define NOM_CLASSE // été prises en compte au cours de la même compilation

class nom_classe {
      …
      // Attributs
      …
      // Méthodes : par exemple constructeur, destructeur, accesseurs,…
      …
} ;

#endif
```

- Un fichier *nom_classe.cpp* : définition détaillée de la classe, où est développé le corps des méthodes :

```
#include "nom_classe.h" // on utilise des guillemets pour les classes crées

nom_classe::nom_classe(){…}

nom_classe::~nom_classe(){…}

void nom_classe::nom_procédure(liste_paramètres){…}

int nom_classe::nom_fonction(liste_paramètres){…}
```

- Un(des) fichier(s) *toto.cpp* utilisant des objets de la nouvelle classe :

```
#include "nom_classe.h"
…

void main(){
      // déclaration (instanciation) d'objets de la classe
      // c'est ici que le constructeur est appelé pour chaque instance
      nom_classe instance_1, instance_2, …, instance_N ;
      …
      // exemple d'appel d'une méthode publique de la classe
      instance_i.méthode_j(liste_paramètres);
      …
}     // c'est ici que le destructeur est appelé pour chaque instance
```

On peut très bien se passer des fichiers *nom_classe.h* et *nom_classe.cpp* pour les incorporer dans *toto.cpp*, mais on ne pourra pas réutiliser la classe dans d'autres programmes. D'autre part, on peut aussi se contenter du fichier *nom_classe.h*, dans lequel on aura développé le corps de toutes les méthodes de la classe, mais ceci signifie qu'elles sont inline, c'est-à-dire chargées en mémoire. De plus, pour de simples questions de lisibilité, ce principe est déconseillé.

Le découpage proposé est le plus courant et permet une parfaite modularité. Il suffit ensuite d'inclure (*#include "nom_classe.h"*) la nouvelle classe dans de nouveaux programmes ou d'autres classes, pour utiliser des objets (instances) de la classe.

3.1.7.1 Exemple de la classe Point

```
#include <iostream.h>
class Point{
      int x,y ;                     // abscisse et ordonnée
      static int nbPoints ;         // nombre total de points
public :
      Point(int a=0, int b=0){      // constructeur avec valeurs par défaut
            x=a ; y=b ;
            cout << "++ nombre total de points : " << ++nbPoints << endl ;
      }
      Point(Point &p){              // constructeur par recopie
            cout  << " = nombre total de points : " << ++nbPoints
                  << " [recopie de (" << p.x << "," << p.y << ")]" << endl;
            x=p.x ; y=p.y ;
      }
      ~Point(){                     // destructeur
            cout  << "-- nombre total de points : " << --nbPoints
                  << " [destruction de (" << x << "," << y << ")]" << endl;
      }
};
int Point::nbPoints = 0 ;           // initialisation de la donnée statique

void f(int) ;                       // juste une fonction pour voir

void main(){
      Point p1,
            p2(3,4),
            p3=p2 ;
      f(5) ;
}

void f(int i){
      Point p(i,i) ;
}
```

3.1.7.2 Résultat d'exécution

```
++ nombre total de points : 1
++ nombre total de points : 2
 = nombre total de points : 3 [recopie de (3,4)]
++ nombre total de points : 4
-- nombre total de points : 3 [destruction de (5,5)]
-- nombre total de points : 2 [destruction de (3,4)]
-- nombre total de points : 1 [destruction de (3,4)]
-- nombre total de points : 0 [destruction de (0,0)]
```

3.2 Surcharges

3.2.1 Surcharge de fonctions

3.2.1.1 Définition d'une fonction surchargée

C++ offre la possibilité de surcharger des fonctions : une fonction peut être définie plusieurs fois sous le même nom. L'identification se fait alors par la liste des paramètres, la version appropriée au(x) type(s) en paramètre(s) est déclenchée.

```
int f(int i){
      …
}

int f(char c){
      …
}

void main(){
      f(0) ;      // appel avec paramètre int
      f('a') ;    // appel avec paramètre char
}
```

3.2.1.2 Portée d'une fonction

On ne peut appeler une fonction que lorsque sa portée s'étend jusqu'à l'endroit de l'appel. En C, la portée n'est jamais un problème car il n'existe que des fonctions globales. Or ce n'est pas le cas en C++.
Si une classe contient une fonction de même nom qu'une fonction de l'une de ses classes de base, la fonction de la classe de base est **occultée**. En revanche, il y a **surcharge** lorsque plusieurs fonctions de même portée portent le même nom.

```
int f(int i){} ;
int f(char c){} ;

class a{
public :
      int f(int i){} ;
      int f(char c){} ;
} ;

class b : public a{     // b hérite de a (voir la partie Héritage)
public :
      int f(int i){} ;
      int f(char c){} ;
} ;

void main(){
      // Appels dans la portée globale
      int i ;
      i=f(1) ;
      i=f('a') ;
      // Appels dans la portée de a
      a a1 ;
      a1=f(1) ;
      a1=f('a') ;
      // Appels dans la portée de b
      b b1 ;
      b1=f(1) ;
      b1=f('a') ;
}
```

3.2.1.3 Règles de conversion des paramètres

Lorsque plusieurs fonctions sont candidates lors d'un appel, il faut que le compilateur fasse le meilleur choix possible. Le type des paramètres de la fonction sélectionnée doit donc être aussi proche que possible des paramètres transmis à la fonction appelée. Les règles suivantes sont mises en œuvre pour convertir les paramètres transmis. Chacune des règle n'est applicable que si les précédentes n'ont pas permis la sélection d'une fonction :

- Il n'y a pas de conversion s'il existe une fonction avec les types de paramètres appropriés. Une conversion n'est tentée que si aucune fonction présentant les mêmes types n'est disponible.
- Les conversions entières et de *float* à *double* s'effectuent ensuite, en cherchant à éviter la perte d'information (ceci n'est possible que d'un type vers un autre dont l'intervalle de définition est plus étendu).
- Ensuite viennent les règles de conversion standard sur les types de base (par exemple un *double* peut être converti en *int* si une fonction est définie en prenant un *int* en paramètre, et qu'on appelle cette fonction avec un *double*).
- Si les règles précédentes n'ont toujours pas permis de choisir une fonction, on applique les conversions de type définies par l'utilisateur (dans le constructeur d'une classe).
- Enfin, la dernière possibilité est l'appel d'une fonction à ellipse (séquence de trois points (…) qui dans une définition de fonction permet de lui passer n'importe quel série de paramètres, c'est la fonction qui porte en elle-même la responsabilité du traitement des paramètres en fonction de leur type).

3.2.2 Surcharge d'opérateurs

Les opérateurs servent à manipuler les données, ils ont une fonctionnalité fixe, mais dépendent des opérandes qui leur sont soumis (polymorphisme). Pour les types de données de base, l'effet des opérateurs est prédéterminé. En revanche, on peut définir des opérateurs surchargés à l'intérieur d'une classe, qui ne seront applicables qu'aux objets de cette classe. Il faut néanmoins tenir compte des restrictions suivantes :

- Seuls les opérateurs correspondant à des symboles définis en C++ peuvent être surchargés, il n'est pas possible d'inventer de nouveaux opérateurs
- L'associativité et la syntaxe des opérateurs ne peuvent pas être modifiés
- L'action des opérateurs sur les types de base ne peut pas être changée
- Les opérateurs suivants ne peuvent pas être surchargés :

  . .* :: ? : sizeof

Les exemples qui suivent décrivent la surcharge de quelques opérateurs.

3.2.2.1 Exemple : l'opérateur d'addition

```
class Point{
      int x,y ;
public :
      …
      Point operator + (const Point &) ;
} ;

Point Point::operator + (const Point &p){
      Point pres ;
      pres.x = x + p.x ;
      pres.y = y + p.y ;
      return (pres) ;
}
```

On peut ainsi écrire le programme suivant :

```
#include <iostream.h>
#include "point.h"

void main(){
      Point p1(1,3),p2(4,5), p3(0,1), p4 ;
      p4 = p1 + p2 + p3 ;
      cout << "p4 = p1 + p2 + p3\n" ;
      p4.afficher() ;
}
```

L'opérande de gauche est nécessairement un objet instance de la classe Point. L'opérande de droite peut être un Point, ou tout autre type qui peut être converti en Point (s'il existe un constructeur de la classe Point qui prend un type T en paramètre) :

```
p3 = p3 + 3 ;
cout << "p3 = p3 + 3\n" ;
p3.afficher() ;
```

On peut écrire les lignes précédentes car il existe un constructeur de la classe Point permettant de convertir un int en Point.

Que faire si on veut pouvoir écrire le programme suivant ?

```
p3 = 3 + p3 ;
cout << "p3 = 3 + p3\n" ;
p3.afficher() ;
```

Dans ce cas, il faudrait modifier la définition de l'opérateur + pour les entiers. Or, on ne peut pas modifier cette définition car l'accès est impossible sur les types de base. Il faut donc déclarer l'opérateur + comme fonction amie de la classe Point (une fonction amie est une fonction non membre d'une classe qui peut utiliser les données privées de la classe) :

```
class Point{
      int x,y ;
public :
      …
      Point operator + (const Point &) ;  // pour p1 + p2 ou p1 + 2
      friend Point operator + (int, const Point &) ; // pour 2 + p2
} ;
// fonction membre
Point Point::operator + (const Point &p){
      Point pres ;
      pres.x = x + p.x ;
      pres.y = y + p.y ;
      return (pres) ;
}
// fonction amie
Point operator + (int i, const Point &p){
      Point pres ;
      pres.x = i + p.x ;
      pres.y = y + p.y ;
      return (pres) ;
}
```

3.2.2.2 Le cas particulier de l'opérateur d'affichage

```
class Point{
      int x,y ;
public :
      …
      friend ostream & operator << (ostream &, const Point &) ;
} ;

ostream & operator << (ostream &o, const Point &p){
      o << "abscisse : " << p.x ;
      o << " ordonnée : " << p.y ;
      return o ;  // pour pouvoir faire des << en cascade
}
```

On peut ainsi écrire :

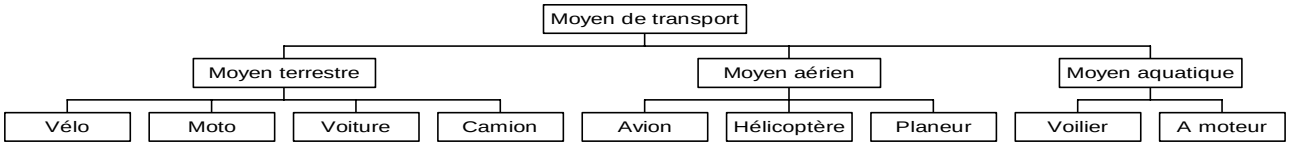
```
cout << "p3 : " << p3 << "\np1 : " << p1 << endl ;
```

3.3 Héritage

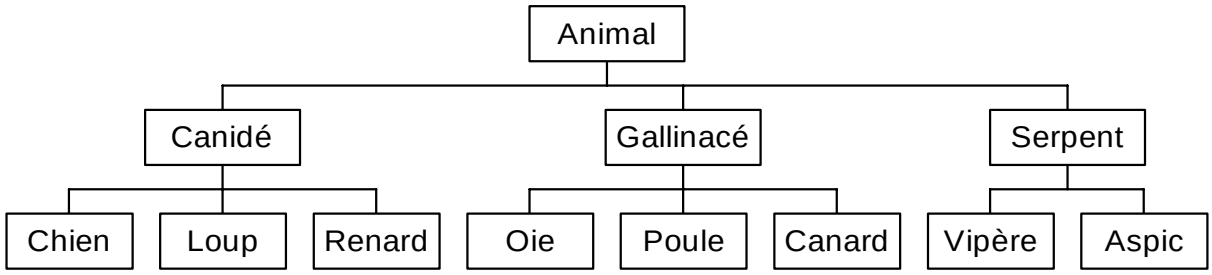
3.3.1 Généralités

- Les classes ont souvent des éléments en commun. En réunissant ces derniers, on est amené à former une **superclasse** (ou **mère**) qui peut servir de base aux autres **sous-classes** (ou **filles**). Les classes forment alors une hiérarchie, toute classe dérivée constituant une spécialisation de la superclasse.

  - Exemple 1 : Les moyens de transport



  - Exemple 2 : Les animaux



- Chaque classe possède des attributs, et l'intersection des attributs des sous-classes forme une superclasse.

- Les filles héritent des membres de leur(s) mère(s), c'est-à-dire que tous les membres de la (des) mère(s) deviennent automatiquement membres des filles. On y ajoutera alors seulement d'éventuelles spécificités (déclaration de nouveaux membres, redéclaration des membres des parents pour changer leur visibilité, redéclaration des membres des parents pour les masquer) :

```
class Canidé : Animal{
      … // membres supplémentaires
} ;
```

ou d'une manière plus générale :

```
class Fille : Mère1, public Mère2, …, private MèreN{
      … // membres supplémentaires
} ;
```

Exemple de redéfinition :

```
class A {
      void f() ;
} ;

class B {
      void f() ;
} ;

class C : public A, public B {     // C hérite de A et de B
      void f() {
            A::f() ;    // f est redéfinie pour appeler successivement
            B::f() ;    // les versions définies dans A et B
            }
} ;
```

- L'héritage définit un nouveau type par combinaison et modification de types existants. Le **polymorphisme** offre la possibilité de définir les mêmes opérations sur plusieurs types, et d'écrire des traitements génériques s'appliquant à tous ces types, en laissant au compilateur le soin de choisir le cas approprié.

3.3.2 Visibilité et héritage

Les classes parentes peuvent être publiques ou privées, l'absence de mot clé signifiant *private* pour les *class* et *public* pour les *struct*.
Quelques règles :

- Classe héritée avec *public* : tous les membres hérités gardent la même visibilité que dans la mère.
- Classe héritée avec *private* : tous les membres hérités sont privés, même ceux qui étaient publics à l'origine. Même si un parent est privé, il est possible de rétablir la visibilité de certains de ses membres en les redéclarant. Cependant, cela ne permet pas de remettre en cause les droits d'accès initiaux :

```
class X{
…
public :
      int x1, x2 ;
…
} ;

class Y : private X {
…
public :
      int X::x1 ; // x1 (et pas x2) est accessible par les clients de Y
…
} ;
```

- Membres déclarés avec *protected* : niveau intermédiaire de visibilité, il fait en sorte que les membres protégés soient privés, sauf pour les sous-classes
- Fonctions amies : une sous-classe hérite des amis de ses ancêtres
- Une classe sera un **sous-type** de chacun de ses parents publics.

3.3.3 Classes virtuelles

Considérons l'exemple suivant :

```
class A {
      …
      void f() ;
      …
} ;
class B : public A {…} ;
class C : public A {…} ;
class D : public B, public C {…} ;
```

La classe D contient deux fois les membres de la classes A, une fois par B, et une fois par C. Cela peut être voulu, mais en général cela pose des problèmes. Par exemple, en supposant que seule A définit la méthode f, un appel à f pour un objet de type D est une erreur. En effet il y a ambiguïté entre f héritée de A à travers B, et f héritée de A à travers C. C'est d'autant plus stupide qu'il s'agit de le même méthode. On voudrait pouvoir dire simplement que B, C, et D héritent de A « en un seul exemplaire », et sans limiter les possibilités de combinaisons de classes.

Dans ce but, C++ propose la notion de classe virtuelle. L'exemple précédent se réécrit ainsi :

```
class A {
      …
      void f() ;
      …
} ;
class B : public virtual A {…} ;
class C : public virtual A {…} ;
class D : public B, public C {…} ; // A ne sera présente qu'une seule fois
```

3.3.4 Méthodes virtuelles

Considérons une classe Forme, avec les opérations Dessiner, Effacer et Déplacer, et ses classes filles Rectangle, Cercle, Triangle. Supposons que l'on ait défini les différentes classes et leurs méthodes. Si l'on désire dessiner une forme quelconque, on peut écrire :

```
Forme *p ;        // déclaration d'un pointeur sur Forme
…
p->Dessiner() ;   // appel de la procédure de dessin
```

L'idée du polymorphisme est que, suivant que p pointe sur un rectangle, un triangle ou une autre forme, c'est la méthode appropriée qui sera appelée. Mais ce n'est pas du tout ce qui se passe ici : le choix de la méthode est fait à la compilation, en fonction du type du pointeur. On appelle cela une **liaison statique**. Ce que l'on voudrait, c'est une **liaison dynamique**, avec un test à l'exécution déterminant la classe de l'objet, et donc la méthode à appeler. Pour cela, il faut définir des méthodes virtuelles :

```
class Forme {
public :
      virtual void Dessiner() ;
…
} ;

class Rectangle : public Forme {
public :
      void Dessiner() ;
…
} ;
```

3.3.5 Classes abstraites

Considérons encore une fois la classe Forme, avec les opérations Dessiner, Effacer, et Déplacer. Une telle classe est dite **abstraite**, car certaines opérations, comme Dessiner, ne peuvent pas être décrite complètement. Par contre, on peut très bien imaginer que la méthode Déplacer soit définie en fonction d'Effacer et de Dessiner. Une bonne bibliothèque Forme capturerait ainsi tout ce qu'il y a de commun entre les formes.

On peut alors définir des classes **réelles** Rectangle, Cercle, ou Triangle, qui héritent de Forme, et complètent la définition des opérations pour chaque cas. Mais le comportement commun est hérité, et n'a pas besoin d'être redéfini. Ainsi l'héritage permet de Déplacer un Triangle. Déplacer est défini pour une Forme quelconque, et fait appel à Dessiner. Puisque la Forme est en fait un Triangle, c'est (par le polymorphisme) la méthode de dessin du Triangle qui sera exécutée.

La définition de nos classes graphiques peut encore être améliorée. En effet, il faudrait pouvoir interdire la création d'objets de type Forme, puisque seuls les types dérivés ont une signification géométrique. Forme est une classe abstraite qui regroupe le comportement commun à toutes les formes, mais ne peut pas tout définir.

C++ propose la solution suivante :

```
class Forme {
public :
      virtual void Dessiner() = 0 ;
…
} ;

class Rectangle : public Forme {
public :
      void Dessiner(){…} ;
…
} ;
```

Les méthodes de Forme qui ne peuvent être définies ont leur corps remplacé par "= 0". Cela indique qu'il s'agit d'une classe abstraite, et la création d'objets de cette classe est interdite. De plus, les classes dérivées devront soit définir les méthodes abstraites, soit à nouveau les déclarer comme abstraites.

3.4 Classes génériques (patrons)

La surcharge des fonctions permet d'utiliser des paramètres de différents types pour une même fonction. Mais il est intéressant d'aller encore plus loin, en créant des classes dont certains attributs peuvent être de n'importe quel type. Pour cela, on utilise des classes génériques (ou patrons).

3.4.1 Exemple d'une liste chaînée d'entiers

Nous allons créer une classe *CListe* dont les éléments sont liés par des pointeurs monodirectionnels. Elle maintient un pointeur sur son début et un autre sur son élément courant, celui qui est en cours de traitement.

La classe dispose d'un constructeur pour initialiser toutes les données et d'un destructeur pour libérer à nouveau les zones allouées.

Une fonction appropriée doit se charger d'insérer un élément dans la liste. D'autres fonctions doivent respectivement ramener en début de liste le pointeur sur l'élément courant, et renvoyer la valeur de l'élément suivant.

```
// Fichier CListe.cpp
#include <iostream.h>

// CLASSE CLISTE
class CListe{
      struct element{   // définition du nouveau type element
            int e;
            element * suivant;
      };
      element *debut;
      element *courant;

public:
      CListe(){         // constructeur
            debut=NULL;
            courant=NULL;
      }
      ~CListe(){        // destructeur
            element *pointeur, *tmp;
            pointeur=debut;
            while(pointeur!=NULL){
                  tmp=pointeur->suivant;
                  delete pointeur;
                  pointeur=tmp;
            }
      }

      int & nouvo(int info){  //insertion d'un nouvel élément au début
            static int dummy;
            element * & pointeur=debut;
            element * tmp=debut;
            while(pointeur!=NULL)
                  pointeur=pointeur->suivant;
            if((pointeur=new element)!=NULL){
                  pointeur->suivant=tmp;
                  pointeur->e=info;
                  courant=pointeur;
                  return(pointeur->e);
            }
            else
                  return(dummy);
      }
```

```
      int & premier(){  // affectation du pointeur courant au début
            courant=debut;
            return(courant->e);
      }

      int & suivant(){  // affectation du pointeur courant au suivant
            if(courant->suivant!=NULL)
                  courant=courant->suivant;
            return(courant->e);
      }
};

// PROGRAMME PRINCIPAL DE TEST
void main(){
      int i;
      CListe list;

      for(i=1;i<=100;i++)
            list.nouvo(i);

      cout<<list.premier()<<endl;
      for(i=1;i<=99;i++)
            cout<<list.suivant()<<endl;
}
```

3.4.2 Les changements à effectuer

La transformation de la classe en classe générique (patron) va permettre la manipulation de listes chaînées de tout ce qu'on souhaite, et non plus seulement d'entiers.

Tout d'abord, on choisit un nom générique, par exemple T, ensuite :

- On change le type qu'on souhaite maintenant générique par *T*
- On ajoute <*T*> au nom de la classe
- On ajoute *template*<*class T*> devant la classe, et devant chaque méthode qui n'est pas inline
- On spécifie le type qu'on souhaite entre <> dans les programmes utilisateurs

3.4.3 Exemple d'une liste chaînée générique

Appliquons maintenant les modifications à la liste chaînée, afin de faire une liste d'entiers (comme avant), mais aussi une liste de caratères, ou de tout autre type (double, float, Etudiant,…) :

```
// Fichier CListeG.cpp
#include <iostream.h>

// CLASSE CLISTE GENERIQUE
template<class T>class CListe{
      struct element{   // définition du nouveau type element
            T e;
            element * suivant;
      };
      element *debut;
      element *courant;

public:
      CListe(){         // constructeur
            debut=NULL;
            courant=NULL;
      }
```

```
	~CListe(){		// destructeur
		element *pointeur, *tmp;
		pointeur=debut;
		while(pointeur!=NULL){
			tmp=pointeur->suivant;
			delete pointeur;
			pointeur=tmp;
		}
	}

	T & nouvo(T info){	//insertion d'un nouvel élément au début
		static T dummy;
		element * & pointeur=debut;
		element * tmp=debut;
		while(pointeur!=NULL)
			pointeur=pointeur->suivant;
		if((pointeur=new element)!=NULL){
			pointeur->suivant=tmp;
			pointeur->e=info;
			courant=pointeur;
			return(pointeur->e);
		}
		else
			return(dummy);
	}

	T & premier(){	// affectation du pointeur courant au début
		courant=debut;
		return(courant->e);
	}

	T & suivant(){	// affectation du pointeur courant au suivant
		if(courant->suivant!=NULL)
			courant=courant->suivant;
		return(courant->e);
	}
};

// PROGRAMME PRINCIPAL DE TEST
void main(){
	int i;
	CListe<int> list1;
	CListe<char> list2;

	for(i=1;i<=10;i++)
		list1.nouvo(i);
	cout<<list1.premier()<<endl;
	for(i=1;i<=9;i++)
		cout<<list1.suivant()<<endl;

	for(i=65;i<=70;i++)
		list2.nouvo(i);	// conversion int-char (65-->A, 66-->B, ...)
	cout<<list2.premier()<<endl;
	for(i=1;i<=5;i++)
		cout<<list2.suivant()<<endl;
}
```