Le C++ (après le C)

Alain Plantec

2004–2011

Table des matières

Préambule

Ce cours n'est pas vraiment une production originale : on trouve plein de cours sur le C++, dans des livres, des polycopiés et sur des sites internet.

La première partie (du chapitre 1 au chapitre 5) est rédigée plus spécifiquement à l'objectif pédagogique du cours : comprendre l'essentiel pour être autonome et passer du C au C++ sans trop d'encombres.

La suite du document consiste plus en une compilation de différentes sources dont les principales sont :

- le cours de Frank Brokken, C++ Annotations (http ://www.icce.rug.nl/documents/cplusplus/)
- le cours de Christian Casteyde (http ://casteyde.christian.free.fr/cpp/cours/online/book1.html)
- la documentation de la *STL* (*Standard Template library*) chez sgi (http ://www.sgi.com/tech/stl).

Je remercie vivement les auteurs de ces documents.

Chapitre 1

Du C au C++

La première mise en œuvre d'un compilateur C++ à été effectuée au début des années 80. Il s'agit alors d'un pré-compilateur qui produit du code C. Cette idée de départ est probablement responsable du fait que le C++ soit un sur-ensemble du langage C.

Tout compilateur C++ est donc aussi un compilateur C ANSI. Le C ANSI diffère du C K&R principalement du fait qu'il est plus fortement typé et donc moins permissif. En effet, beaucoup plus de contrôles sont effectués par le compilateur.

Ce chapitre présente les ajouts au C K&R utilisés couramment pour l'écriture des programmes C++ mais aussi les spécificités du C++ qui n'ont rien à voir avec la programmation «orienté-objet».

1.1 Du C au C ANSI

1.1.1 Définition des constantes

Le mot clé *const* peut être utilisé pour geler la valeur d'une variable, d'un pointeur ou de la zone mémoire pointée. De plus, une constante peut être utilisée pour la définition de la taille des tableaux. Les constantes sont utilisées à la place des constantes du préprocesseur du C. Les deux principaux avantages sont :

- qu'il est possible de récupérer l'adresse d'une constante et
- que le contrôle de type est beaucoup plus stricte.

Exemples :

```
1  main(int argc, char ** argv) {
2
3    const int max_tab = 10 ; // max_tab est constant
4    max_tab = 20 ; // interdit
5
6    const int * iptr ; // iptr pointe sur un entier constant
7    int i = 4 ;
8    iptr = &i ; // ok
9    *iptr =100 ; // interdit, la zone memoire pointee est constante
10
11   const char * nom = "Bibi Fricotin" ; // nom pointe sur une chaine de caracteres
         constante
12   nom[0] = 'T' ; // interdit, la zone memoire pointee est constante
13   *nom = 'T' ; // interdit, la zone memoire pointee est constante
14   nom = "Tintin & milou" ; // ok
15
16   char * const kptr = "titi & grominet" ; // kptr est un pointeur constant
17   kptr[7] = 'R' // ok
18   kptr = "calimero" ; // interdit, kptr est constant
19
20   // bp est un pointeur constant qui pointe sur une zone □mmoire constante
21   const char * const bp = "Que j'aime le C++" ;
22   bp = "Que j'aime SmallTalk" ; // interdit, bp est un pointeur constant
23   for (int i = 0 ; i < strlen(bp) ; i++) {
24     bp[i] = ' ' ; // interdit, bp pointe sur une zone memoire constante
25   }
26
27   const size_t len = BUFSIZ ; // definition d'une constante positive
28   char iobuf[len] ; // utilisation de la constante pour □dfinir la taille d'un tableau
29 }
```

1.1.2 Les fonctions

1.1.2.1 Le prototype

Le prototype des fonctions diffère totalement de celui du C K&R ;

- pour la mise en oeuvre d'une fonction, le type et le nom des paramètres doivent être spécifiés entre les parenthèses. Pour les paramètres non utilisés dans le corps de la fonction, seul le type est obligatoire. Le séparateur de déclarations de paramètres est la virgule.
- pour la déclaration de la fonction (utilisation avant lecture de la mise en oeuvre par le compilateur ou déclaration en externe) seulement le type des arguments doit être spécifié entre parenthèses. Il est recommandé de préciser aussi le nom des paramètres.
- Le type de retour par défaut est *int*. Une fonction qui ne retourne rien (procédure) doit préciser *void* comme type de retour

Exemple :

```
1  #include <stdio.h>
2
3  int max (int i1, int i2) ; // Declaration de la fonction max
4
5  void printMax(FILE * file, int i1, int 12)
6  {
7    fprintf(file, "%d\n", max(i1,i2)) ; // Le prototype de fprintf a ete lu dans stdio.h
8  }
9
10 int max(int i1, int i2) // Mise en oeuvre de la fonction max
11 {
12   retum (i1 > i2) ? i1 :i2 ;
13 }
```

Utiliser (spécifier l'appel d'une fonction) sans avoir au préalable spécifié son prototype est une erreur de programmation détecté par les compilateurs. En effet, sans déclaration préalable (dans un fichier d'entête par exemple), le compilateur est dans l'impossibilité d'effectuer les contrôles de types et l'analyse des signature. Nous verrons que l'analyse des signatures est important dans le cas de redéfinitions de fonctions.

1.1.2.2 Contrôle des types

Un compilateur C ANSI est très strict pour le contrôle de type. Le prototype d'une fonction doit être connu avant son utilisation. De plus, l'appel doit respecter la définition donnée par le prototype pour le type des paramètres et le type du retour de la fonction. Par exemple, utiliser la fonction

printf sans au préalable inclure le fichier *stdio.h* dans lequel son prototype est déclaré constitue une erreur de compilation.

1.1.2.3 Fonction sans paramètres

L'absence de paramètre peut être explicitement spécifiée par *void* :

```
1 int f1() ; // fonction sans parametre
2 int f2(void) ; // fonction sans parametre
```

1.1.2.4 Fonctions avec nombre variable d'arguments

Il est toujours possible de mettre en oeuvre des fonctions à nombre variable d'arguments de type *printf*. Pour cela, on utilise l'ellipsis («...») comme dernier argument. Cependant, la fonction doit être déclarée avec au moins un paramètre typé et nommé avant l'emploi de «...».

Pour récupérer les arguments anonymes, le corps de la fonction doit mettre en oeuvre les macros *va_start()*, *va_arg()*, *va_end()*, ainsi que le type *va_list* fournis dans le fichier *stdarg.h*.

Attention, la programmation de ce type de fonction n'est pas recommandée car :

- il ne peut y avoir de contrôle de type sur les arguments lors de l'appel et
- le code s'avère souvent complexe et difficile à maintenir.

Exemple :

```
1  #include <stdio.h> // entrees/sorties standards
2  #include <stdarg,h> // inclusion du fichier standard pour le nombre variable d'
       arguments
3
4  enum LesTypesDeTruc { chaine, caractere, reel } ;
5
6  void afficheDesTrucs(int nbTrucs, ...) ; // Utilisation de l'ellipsis
7
8  main(int argc, char **argv)
9  {
10   afficheDesTrucs(3, reel, double(10), caractere, 'K', chaine, "Bibi Fricotin") ;
11   afficheDesTrucs(1, chaine, << Tintin & milou >>) ;
12 }
13
14 void afficheDesTrucs(int nbTrucs, ...)
```

```
{
  va_list args ; // La liste de arguments anonymes
  va_start(args, nbTrucs) ; //args pointe sur la pile (initialisation de la liste d'arguments)

  for (int cpt = 0 ; cpt < nbTrucs ; cpt++) { // récupération des arguments
    LesTypesDeTruc tdt ;

    tdt = va arg(args, LesTypesDeTruc) ; // récupération du type de l'argument suivant
    switch (tdt) {
      case chaine  : { // l'argument suivant est une chaîne de caractères
  const char * str = va_arg(args, const char*) ;
  printf("%s", str) ;
  break ;
      }
      case caractere  : { // l'argument suivant est un caractère
  const char k = va_arg(args, char) ;
  printf("%c", k) ;
  break ;
      }
      case reel  : { // l'argument suivant est un double
  const double dbl = va_arg(args, double) ;
  printf("%lf", dbl) ;
  break ;
      }
    }
  }
  va end(args) ; // Remise en état de la pile
}
```

1.2 Du C ANSI au C++

1.2.1 Les commentaires

```
/* ceci est un commentaire a la C */

/*
 * ceci aussi est un commentaire a la C
 * avec plusieurs lignes
 */

// ceci est un commentaire a la C++.
// Le commentaire debute apres les deux /
// et se termine en fin de ligne
```

Dans un source C, on ne peut utiliser que le commentaire «à la C». Dans un source C++, il est possible d'utiliser aussi bien le commentaire «à la C»

que le commentaire «à la C++».

1.2.2 Nom de structure, d'union et d'énumération

Le nom d'une structure, d'une union et d'une énumération constitue maintenant un type et peut donc être utilisé en tant que tel sans utilisation des mots clés *struct*, *union* ou *enum*. Ainsi, le mot clé *typedef*, utilisé en C pour la définition de tels types est maintenant obsolète.

Exemple :

```
#ifndef__cplusplus

/* Version C avec typedef */
typedef enum {_char_, _int_, _double_} types_primitifs;

typedef union {
   char c;
   int i;
   double d;
} primitif;

typedef struct {
   types_primitifs tp;
   primitif prim.
} val_primitive;

#else

// Version C++ equivalente, le typedef est inutile
enum types_primitifs {_char_, _int_, _double_};

union primitif {
   char c;
   int i;
   double d;
};

struct val_primitive {
   types_primitifs tp;
   primitif prim.
};

#endif

// L'utilisation du type en C ou en C++
main(int argc, char **argv)
{
   val_primitive v;
   v.tp = _char_;
   v.prim.c = 'A';
```

```
41 }
```

1.2.2.1 Union anonyme

Une union sans nom peut être utilisée dans une structure.

Exemple :

```
1  enum types_primitifs {_char_, _int_, _double_};
2
3  struct val_primitive {
4    types_primitifs tp;
5    union {
6      char c;
7      int i;
8      double d;
9    }; // La declaration de l'union dans la structure ne necessite pas son nommage.
10 };
11
12 main(int argc, char **argv)
13 {
14   val_primitive v;
15   v.tp = _char_;
16   v.c = 'A';  // On accede directement aux champs de l'union.
17 }
```

1.2.3 L'opérateur de résolution de contexte ::

L'opérateur *::* peut être utilisé pour résoudre des problèmes de conflits entre des définitions globales et des définitions locales.

Exemples :

```
1  #include <stdio.h>
2
3  const char * nom = "Bibi"; // variable globale
4  main()
5  {
6    const char * nom = "Toto"; // variable automatique
7    printf("%s", nom); // affiche "Toto"
8    printf("%s",  ::nom); // affiche "Bibi"
9  }
```

L'utilisation de cet opérateur n'est pas du tout recommandé dans le cas de cet exemple qui est un cas d'utilisation à éviter. Par contre, cet opérateur s'avèrera indispensable dans le contexte de la définition de classes C++.

1.2.4 Le pointeur NULL

En C++, un pointeur null à pour valeur 0 et non *NULL* comme c'est le cas en C. Il ne s'agit que d'une évolution de style d'écriture. En effet, en C, la valeur *NULL* est définie par la valeur 0 typée par *void* * ou bien *char* *. Voici par exemple, un extrait du fichier d'entête standard *stddef.h* d'un système *Linux* dans lequel est défini la valeur *NULL* :

```
#ifndef _STDDEF_H
#define _STDDEF_H

#undef NULL
#if defined(__cplusplus)
# define NULL 0
#else
# define NULL ((void *)0)
#endif
```

1.2.5 Le type bool

Le type booléen *bool* a été introduit comme nouveau type prédéfini. Le domaine de ce type comprend les deux valeurs réservées *true* et *false*. Si le compilateur est antérieur à l'introduction de ce type, il peut être remplacé par la déclaration suivante :

```
enum bool {false = 0, true = 1};
```

1.2.6 Les déclarations de variables

C++ considère une déclaration de variable comme une expression. En conséquence, une déclaration peut être spécifiée presque n'importe où dans le code :

- entre deux instructions,
- dans la déclaration d'une boucle *for*.

Dans le cas d'une déclaration dans une boucle *for* (voir l'exemple ci-après), les anciennes versions de compilateurs acceptent l'accès à la variable après la boucle. Cette possibilité est normale du point de vue d'un compilateur puisque le bloc contenant la déclaration n'est pas clos. Par contre, cette possibilité diminue la lisibilité du code. En conséquence, le C++ ANSI interdit maintenant l'utilisation d'une variable déclarée dans la spécification d'un *for* après la boucle.

On peut donc faire n'importe quoi. Ceci dit, nul ne nous y oblige. Il paraît en effet raisonnable, pour des raisons de lisibilité et de maintenabilité du code, de continuer à localiser les déclarations en début de bloc (juste après une accolade ouvrante) en s'autorisant la possibilité de déclaration du compteur de boucle dans les boucles *for*. Voici un exemple de ce qu'il est possible de faire en C++ :

```
#define MAXTAB 10

main ()
{
  char ktab[MAXTAB]; // declaration classique en debut de bloc
  initialiser_tab(ktab);

  printf("□caractre □recherch  : ");
  char k; // declaration d'une variable entre deux instructions (a eviter)
  scanf ("%c", &k);

  // declaration du compteur cptk dans la specification de la boucle
  for (int cptk = 0; cptk <MAXTAB; cptk++) {
    if (tab[cptk] == k) {
      break;
    }
  }
  // d'apres les regles du C++ ANSI, cptk n'est normalement plus visible ici
```

1.2.7 Conversion de type

Le nom d'un type T peut être utilisé comme un nom de fonction pour spécifier une conversion dans le type T.

Exemple :

```
int entier;
scanf("%d", &entier);

double reel;
reel = (double) entier; // en C, pour "caster" entier en double
```

```
6 reel = double(entier) ; // en C++, on a une fonction de conversion dont le nom est celui
      du type cible
```

1.2.8 Les fonctions

1.2.8.1 Surcharge des fonctions

La possibilité de surcharge autorise de donner le même nom à deux fonctions différentes qui se distinguent alors par le type de leurs paramètres. La surcharge est résolue à la compilation par l'analyse des types des arguments passés dans l'appel de la fonction.

Toute ambiguïté doit être levée à la compilation. Dans certains cas, il est indispensable de typer explicitement les valeurs d'arguments passés.

Un premier exemple :

```
1  void affiche (char c) // version 1
2  {
3    printf("%c", c) ;
4  }
5
6  void affiche (const char *s) // version 2
7  {
8    printf("%s", s) ;
9  }
10
11 void affiche (int d) // version 3
12 {
13   printf("%d", d) ;
14 }
15
16 main()
17 {
18   affiche ('K') ; // appel de la version 1
19   affiche("Bibi Fricotin") ; // appel de la version 2
20   affiche(10)  ; // appel de la version 3
21 }
```

Pour un type T, le compilateur est incapable de faire la différence entre un paramètre de type T ou de type T&. Enfin, le compilateur ne se sert ni du type de retour, ni du type des paramètres avec valeur par défaut pour différencier deux fonctions.

Exemple :

```
1  struct S { };
2
3  void f (S);
4  void f (const S); // OK, le compilateur fait la difference avec les autres versions de f
5  S f (S &); // OK
6
7  void g(const char *);
8  int g(const char *s); // Erreur de surcharge
9
10 void h(double d, char = ';');
11 void h(double d, int = 2);
12
13 main(int argc, char **argv)
14 {
15   S s;
16   f(s); // Erreur : le compilateur ne sait pas quelle fonction f() appeler
17   S s2 = f(s); // Erreur
18
19   h(double(3)); // Erreur : le compilateur ne sait pas quelle fonction h() appeler
20 }
```

1.2.8.2 Valeurs par défaut des paramètres de fonctions

Il est possible de préciser la valeur par défaut des paramètres. Les paramètres avec une valeur par défaut doivent obligatoirement se trouver en fin de liste des paramètres. La fonction peut alors être invoquée avec moins d'arguments que le nombre de paramètres déclarés.

Exemple :

```
1  #include <stdio.h>
2
3  void f(int , int = 0, int = 10);// Declaration de l'interface avec valeurs par defaut
4
5  void f(int x, int i, int j) // Declaration de la mise en oeuvre
6  {
7    fprintf(stdout,(const char*)"[%d  : %d  : %d]\n", x, i, j);
8  }
9
10 main(int argc, char **argv)
11 {
12   f(-20); // □rsultat sur la sortie standard : [-20  : 0  : 10]
13   f(-300, 10); // [-300  : 10  : 10]
14   f(-400, 30, 40); // [-400  : 30  : 40]
15 }
```

Autre exemple tordu mais valide :

```
#include <stdio.h>

void f(int, int = 10) ; // Declaration de la valeur par defaut du deuxieme parametre

void f(int i = 0, int j) // Declaration de la valeur par defaut du premier parametre
{
  fprintf(stdout,(const char*)"[%d  : %d]\n", i, j) ;
}

main (int argc, char **argv)
{
  f() ; // resultat sur la sortie standard :[0  : 10]
}
```

1.2.8.3 Les fonctions «inline»

La déclaration d'une fonction peut être précédée du mot clé *inline*. Le programmeur précise ainsi au compilateur qu'il aimerait bien que les appels de la fonction soient remplacés directement par la suite d'instructions contenue dans le corps de la fonction. La gain escompté est une plus grande efficacité puisqu'il y a économie de l'appel de la fonction.

Cependant, le remplacement effectif des appels par le compilateur n'est pas garanti. Ce remplacement ne sera effectué que si le compilateur considère qu'il est plus avantageux de le faire. Pour cette raison, il est inutile de spécifier inline de grosses fonctions puisque le compilateur n'en tiendra certainement pas compte.

L'utilisation des fonctions inline remplace avantageusement l'utilisation des macros, en effet le contrôle de types des paramètres est effectué par le compilateur et le risque d'effets de bord possible lors de l'utilisation de macros est dans ce cas complètement éliminé.

Exemple :

```
#define CARRE(i)(i*i) // Macro très dangereuse

inline int carre(int i) // Déclaration d'une fonction inline
{
  retum i*i ;
}
```

```
7
8  extern void fonctionEnLigne() ; // Impossible si fonctionEnLigne() est déclarée inline
9
10 main(int argc, char **argv)
11 {
12   const char * kptr = ”Bibi Fricotin” ;
13   int i ;
14   int result ;
15
16   i = 2 ;
17   result = CARRE(i++) ;
18   printf(”%d”, result) ; // Affichera 6 et non 4 comme on l’attend
19   printf(”%d”, i) ; // Affichera 4 et non 3
20
21   i = 2 ;
22   result = carre(i++) ; // Affichera bien 4
23   printf(”%d”, i) ; // Affichera bien 3
24
25   CARRE(kptr) ; // Aucune erreur de compilation
26   carre(kptr) ; // Erreur de compilation, type d’argument incorrecte
27
28   int (*fctPtr)(int i) ;
29   fctPtr = carre ; // Erreur, le compilateur ne connaît pas l’adresse de la fonction carre
30
31   result = fctPtr(3) ;
32 }
```

1.2.9 Allocation dynamique et libération de la mémoire

Le C++ introduit deux nouveaux opérateurs pour l’allocation et la libération de la mémoire : *new* permet l’allocation et *delete* la libération de la mémoire :

1.2.9.1 Allocation de la mémoire

new T : pour un type *T*, *new T* renvoie l’adresse d’une zone mémoire allouée. La taille de la zone est *sizeof(T)*. Le type de retour est *T**. Cette instruction remplace l’utilisation de *(T*) calloc(1, sizeof(T))* du C.

new T[N] : pour un type *T*, *new T[N]* renvoie l’adresse d’une zone mémoire allouée. La taille de la zone est *sizeof(T) * N*. Le type de retour est *T**. Cette instruction remplace l’utilisation de *(T*) calloc(N, sizeof(T) * N)* du C.

1.2.9.2 Libération de la mémoire

delete vptr : pour une variable *vptr*, pointeur sur un type *T*, *delete vptr* permet la désallocation de la mémoire pointée par *vptr*, soit *sizeof(T)* octets en mémoire dynamique alloués précédemment par *new T*. Cette instruction remplace l'utilisation de *free(vptr)* en C.

delete [] vtab : pour une variable *vtab*, pointeur sur un tableau de N éléments de type *T*, *delete vtab* désalloue la mémoire pointée par *vtab*, soit *sizeof(T) * N* octets en mémoire dynamique alloués précédemment par *new T[N]*. Cette instruction remplace l'utilisation de *free(vtab)* en C.

Exemple :

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct Personne
5  {
6    char nom[128];
7    char prenom[128];
8  };
9
10 main(int argc, char ** argv)
11 {
12   const char * kpt0 = "Bibi Fricotin";
13   // allocation d'un □caractre et affectation de l'adresse retournee dans un pointeur de
         □caractre.
14   char * kptr1 = new char;
15   *kptr1 = 'c';
16
17   // Allocation de la place necessaire au stockage de copie de chaine classique en C
18   char * kptr2 = new char[strlen(kpt0)+1];
19   printf("%s\n", kptr2); // Utilisation du pointeur
20   delete kptr1; // □libration de 1 octet en □mmoire
21
22   delete [] kptr2; // □libration de strlen(kpt0) + 1 octet
23
24   Personne * persPtr = new Personne [10]; // Allocation d'un tableau de 10 structures
         Personne
25   strcpy(persPtr[0].nom, "Fricotin"); // Utilisatlon du tableau
26   strcpy(persPtr[0].prenom, "Bibi");
27
28   delete [] persPtr; // liberation de sizeof(persPtr) * 10 octets
29 }
```

1.2.9.3 La fonction *set_new_handler*

En C++, lorsqu'une allocation retourne une erreur (plus de mémoire disponible), une fonction définie par défaut est automatiquement exécutée. Par défaut, cette fonction ne fait rien et le *new* retourne 0.

La fonction *set_new_handler* peut être utilisée pour redéfinir la fonction par défaut (à la manière de $signal$ sous unix). Ainsi, on indique quel est le code à exécuter si une allocation de mémoire retourne une erreur. La redéfinition de cette fonction doit respecter certaines contraintes qui ne sont malheureusement pas stables d'un compilateur à l'autre.

Pour le compilateur de *Microsoft*, les contraintes sont :

- La fonction prend un argument de type *size_t* qui, lorsque la fonction est appelée, prend comme valeur le nombre d'octets qui auraient du être alloués par *new*.
- La fonction retourne un entier. L'entier retourné est la valeur directement retournée par le *new*.

Pour le compilateur $g++$ de GNU :

- La fonction ne prend aucun argument,
- La fonction ne retourne rien.

Classiquement, dans la redéfinition de cette fonction, une message d'erreur est affiché et le programme se termine. Voici un exemple :

```
1  #include <new.h>
2  #include <iostream>
3
4  void plus_de_memoire()
5  {
6    cout << "□Mmoire □□puise. Sortie violente !" << endl;
7    exit(1);
8  }
9
10 int main()
11 {
12   // installation de la fonction d'erreur
13   set_new_handler(plus_de_memoire);
14
15   // On mange toute la □mmoire
16   long total = 0;
17   while (1) {
18     int * iptr = new int [10000];
19     total += 10000 * sizeof(int);
20     cerr << total << endl;
21   }
22   return (0);
23 }
```

Attention, en cas d'erreur, les fonction C telles que *strdup*, *malloc*, *realloc* n'exécutent pas forcément la fonction installée par *set_new_handler*. Là encore, cela dépend du compilateur. C'est une bonne raison pour ne pas les utiliser dans les développements C++.

1.2.10 Les références

Il est possible de définir une référence (ou alias, ou synonyme) d'une variable déjà existante. On utilise le symbole & pour la déclaration d'une référence.

1.2.10.1 Déclaration et utilisation des références

Déclaration d'une référence sur une variable : On peut déclarer une référence au même niveau qu'une variable dans un bloc d'instructions. Dans ce cas, la référence doit être initialisée dans la déclaration :

```
int unInt = 4 ;
int unAutre = 5 ;
int & unInt_bis = unInt ; // Déclaration d'une référence sur la variable unlnt

unInt = unAutre ;
printf("%d\n", unInt_bis) ; // Affichage de unlnt_bis : 5 sur stdout

unInt_bis = 50 ;
printf("%d\n", unInt) ; // Affichage de unlnt_bis : 50 sur stdout
```

La passage par variable : La référence est utilisée pour le passage de paramètres dont la valeur peut être modifiée par la fonction. On dispose donc du passage par variable du langage *Pascal* : le paramètre est alors déclaré en lecture-écriture.

Un exemple :

```
// programmation du passage par variable en C
void fc(int * pptr) .
{
  *pptr = *pptr + 1 ; // La fonction manipule explicitement *pptr
}

```

```
// Solution en C++
void fcpp (int & p)
{
  p = p + 1 ; // La fonction manipule implicitement *p
}

main(int, char **)
{
 int unInt = 4 ;
 printf("%d\n", unInt) ; // Affichage de unInt : 4 sur stdout
 fc(&unInt) ; // en C, on passe explicitement l'adresse d'une variable
 printf("%d\n", unInt) ; // Affichage de unInt : 5 sur stdout

 int unAutreInt = 4 ;
 printf("%d\n", unAutreInt) ; // Affichage de unAutreInt : 4 sur stdout
 fcpp(unAutreInt) ; // en C++, avec la reference, le passage de l'adresse est implicite
 printf("%d\n", unAutreInt) ; // Affichage de unAutreInt : 5 sur stdout
}
```

Un autre exemple :

```
const int Taille = 10 ;

struct UnTruc {
  int cumul ;
  int tab[Taille] ;
} ;

void initialiseTruc(UnTruc & truc, const int itab[])
{
  truc.cumul = 0 ;
  for (int cpt = 0 ; cpt < Taille ; cpt++) {
    truc.tab[cpt] = itab[cpt] ;
  }
}

void cumulTruc(UnTruc &truc)
{
  truc.cumul = 0 ;
  for(int cpt = 0 ; cpt < Taille ; cpt++) {
    truc.cumul += truc.tab[cpt] ;
  }
}

main(int, char **)
{
  UnTruc truc ;
  const int itab[] = {9,8,7,6,5,4,3,2,1,0} ;

  initialiseTruc(truc, itab) ;
  printf("%d\n", truc.cumul) ; // Affichage de truc.cumul : 0 sur stdout
```

```
31    cumulTruc(truc) ;
32    printf(”%d\n”, truc.cumul) ; // Affichage de truc.cumul : 45 sur stdout
33  }
```

Le retour de fonction : Une fonction peut aussi être spécifiée comme retournant une référence sur une variable existante. Une telle fonction peut alors être utilisée en partie gauche d'une affectation. Cette possibilité est montré dans les exemples ci-dessous. Bien que les utilisations montrées soient quelques peu exotiques, nous verrons qu'en C++, le retour d'une référence est beaucoup utilisé principalement pour la définition des opérateurs.

Exemple :

```
1   int globInt = 0 ;
2
3   int & flvalue1 ()// Fonction correcte, retourne la □□rfrence d'un entier global
4   {
5     retum globInt ;
6   }
7
8   int & flvalue2 ()// Fonction correcte, cpt est statique
9   {
10    static int cpt = 0 ;
11    return cpt ;
12  }
13
14  int & flvalue3 ()// Erreur (normalement □□dtecte par le compilateur)
15  {
16    int i ; // i est une variable automatique (dans la pile)
17    retum i ; // La fonction ne peut donc pas retourner i
18  }
19
20  main(int, char **)
21  {
22    flvaluel() = 4 ;
23
24    printf(”%d\n”, globInt) ; // Affiche 4
25    flvaIue2()++ ;
26    printf(”%d\n”, flvalue2()) ; // Affiche 1
27  }
```

Passage de gros objets en paramètres : Lorsqu'un paramètre n'est pas modifié par une fonction, ce paramètre peut être passé par copie comme

c’est le cas en C. Cependant, lorsqu’il s’agit de passer de grosses structures de données, le passage par copie peut induire des performances faibles en terme de rapidité d’exécution et d’utilisation de la mémoire. En C, la solution est de déclarer le paramètre comme un pointeur. En C++, le passage par référence peut être utilisé pour mettre en oeuvre le passage de telles structures de données sans les inconvénients précisés ci-dessus (le compilateur traite en effet une référence comme un pointeur). Pour garantir que la données passée en paramètre n’est pas modifiée par la fonction, le paramètre est déclaré comme une référence constante.

Exemple :

```
1  // Une grosse structure de ☐donne
2  struct TAdresse
3  {
4    char nom [128];
5    char addresse1 [128];
6    char addresse2 [128];
7  };
8
9  // Passage d’une ☐☐rfrence constante pour ☐viter la recopie sur la pile
10 void affiche_adresse (const TAdresse & adr)
11 {
12   // La fonction ne modifie pas adr
13   printf (”nom %s\nadresse %s\n%s\n”, adr.nom, adr.adresse1, adr.adresse2);
14 }
15
16 int main ()
17 {
18   TAdresse monadresse
19   affiche_adresse (monadresse);
20 }
```

1.2.10.2 Pointeurs versus références

Une référence ne peut pas exister indépendamment d’une variable de laquelle elle est l’alias. Un paramètre peut être déclaré comme référence indépendamment d’une variable existante mais le lien entre une variable est le paramètre est effectué lors de l’appel de la fonction. Une référence sur une variable globale ou locale ne peut pas être changé en l’alias d’une autre variable. Le lien entre une référence et la variable est définit une fois pour toute.

Un pointeur est une variable qui peut ou non être initialisée. Il est possible de modifier l’adresse contenue dans un pointeur, il n’y donc pas de lien

immuable entre un pointeur et la zone mémoire pointée.

Il est possible de récupérer l'adresse d'une référence avec l'opérateur &. L'adresse récupérée est celle de la variable dont la référence est l'alias.

Il est aussi possible de récupérer l'adresse d'un pointeur. Cependant, la valeur retournée n'a rien à voir avec l'adresse pointée par le pointeur (contenue dans le pointeur).

1.2.11 Les flots : une première approche

1.2.11.1 Les flots standards

Comme en C, il est possible d'exploiter des flots d'entrées/sorties standards automatiquement ouverts lorsqu'un programme est exécuté. Il s'agit des flots *cin*, *cout* et *cerr* pour respectivement, en comparaison avec C, *stdin*, *stdout* et *stderr*. *cin* et de type *istream*, *cout* et *cerr* sont de type *ostream*.

Syntaxiquement, l'utilisation de ces flots implique une écriture C++. On utilise l'opérateur C++ de sortie (d'entrée) dans (depuis) un flot « (»). Les flots standards sont des objets déclarés dans *iostream.h* de la même façon que les flots standards du C sont des variables déclarées dans *stdio.h*.

Bien que l'on puisse établir un parallèle entre les flots standards du C et ceux de C++, ils demeurent physiquement différents. Par exemple, *stdout* est géré indépendamment de *cout* et vice-versa.

Vite un exemple :

```
1  #include <iostream>
2
3  void main()
4  {
5    int i;
6    char s[30];
7    double d;
8
9    cout << "Un entier  :" << endl; // endl représente la fin de ligne
10   cin >> i;
11   cout << "Une chaine  :" << endl;
12   cin >> s;
13   cout << "Un réèl  :" << endl;
14   cin >> d;
15
16   cout << "Entier  : " << i << ", chaine  : " << s << "et le réèl  :" << d << endl;
17 }
```

Cet exemple donne des utilisations pour des types simples et prédéfinis de C++. Pour des classes définies par un développeur, il sera aussi possible d'utiliser les flots de manière identique en redéfinissant les opérateurs pour ces nouvelles classes.

1.2.11.2 Association flot/fichier

Les flots de type *ofstream* et *ifstream* disposent des mêmes fonctionalités que les types *ostream* et *istream*. La différence majeure est qu'un *ofstream* et qu'un *ifstream* est associé à un fichier. Les opération de lecture et d'écriture s'effectuent dans ce fichier.

Pour déclarer et ouvrir un fichier, on utilise les déclarations suivantes :

ofstream fout; // déclaration d'un fichier dans lequel on veut ecrire
ifstream fin; // déclaration d'un fichier dans lequel on veut lire
fout.open(nom_fichier, mode);
fin.open(nom_fichier, mode);

La fonction *open* permet d'associer un fichier physique dont le nom est *nom_fichier* au fichier logique (ici, fin ou fout) déclaré comme une variable.

mode indique comment le fichier est ouvert, soit :

- *ios ::in* : ouverture en lecture (obligatoire pour ifstream)
- *ios ::out* : ouverture en écriture (obligatoire pour ofstream)
- *ios ::app* : ouverture en ajout de données (en fin de fichier)
- *ios ::are* : se place en fin de fichier après ouverture
- *ios ::trunc* : si le fichier existe, écrase son contenu
- *ios ::nocreate* : le fichier doit exister
- *ios ::noreplace* : le fichier ne doit pas exister

Tout comme avec *cin* et *cout*, on peut utiliser les opérateurs << et >> pour écrire et lire dans un fichier :

```
1 #include <iostream>
2 #include <fstream>
3
4 void main()
5 {
6   ofstream out;
7   ifstream in;
8
9   out.open("test.out", ios ::out);
```

```
  in.open("test.in", ios ::in);

  int i;
  char s[30];
  double d;

  in >> i;
  out << i;
  cin >> s;
  out << s;
  cin >> d;
  cout << d;
}
```

Dans cet exemple, un entier, une chaîne de caractères et un double sont lus dans le fichier *test.in* et écrits dans le fichier *test.out*. On note que la lecture dans un fichier avec >> avale les blancs tout comme *fscanf*.

Voici un exemple de programme qui copie un fichier dans un autre. Pour éviter d'avaler les blancs (et donc de les perdre) on utilise la fonction *get* pour lire caractère par caractère :

```
#include <stdlib.h>
#include <iostream.h> // inclusion du fichier standard pour les ostream
#include <fstream.h> // inclusion du fichier standard pour les ofstream

main(int argc, char * argv[])
{
  if (argc < 3) {
    cerr << "usage : cp nom_fichier_source nom_fichier_destination" << endl;
    exit(-1);
  }
  ifstream fin;
  fin.open(argv[1], ios ::in);
  if (!fin) {
    cerr << "Echec d'ouverture du fichier " << argv[1] << " en lecture" << endl;
    exit(-1);
  } else {
    ofstream fout;
    fout.open(argv[2], ios ::out);
    if (!fout) {
      cerr << "Echec d'ouverture du fichier " << argv[2] << " en ecriture" << endl;
      exit(-1);
    } else {
      while (!fin.eof()) {
  char c;
  fin.get(c);
  fout << c;
      }
```

```
28      }
29    }
30    exit(0);
31  }
```

1.2.12 Utiliser du code C dans du C++

Il est possible d'utiliser du code C dans des programmes C++. Il suffit de préciser au compilateur quelles sont les parties C. Pour cela, la directive *extern "C"* peut être utilisée de deux façons différentes :

- si *extern "C"* est suivie d'un bloc (entre deux accolades, l'une ouvrante et l'autre fermante) alors, toutes les déclarations contenues dans le bloc sont considérées comme du C par le compilateur C++,
- sinon, seule la déclaration qui suit *extern "C"* est considérée comme du C par le compilateur C++.

De plus, pour qu'un fichier d'entête pour du C puisse être accepté aussi bien avec un compilateur C que par un compilateur C++, il est possible d'utiliser la macro définition *__cplusplus* qui doit être définie par les compilateurs C++.

Exemple :

```
1  #ifndef __MON_ENTETE_H__
2  #define __MON_ENTETE_H__
3
4  #include <stdio.h>
5
6  /* la macro __cplusplus est □dfinie par le compilateur C++ */
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10
11   struct _MaStructC {
12     int i;
13     char c;
14   };
15   typedef struct_MaStructC MaStructC;
16
17   extern int fc1 (const char *);
18   extern int fc2(double, int);
19
20 #ifdef __cplusplus
21 }
22 #endif
23
24 #endif /* __MON_ENTETE_H__ */
```

Un autre exemple :

```
#ifndef __MON_ENTETE_H__
#define __MON_ENTETE_H__

#include <stdio.h>

/* la macro __cplusplus est □dfinie par le compilateur C++ */
#ifdef __cplusplus
# define EXTERNC extern ”C”
#else
# define EXTERNC extern
#endif

struct _MaStructC {
   int i;
   char c;
};
typedef struct_MaStructC MaStructC;

EXTERNC int fc1(const char *);
EXTERNC int fc2(double, int);

#endif /* __MON_ENTETE_H__ */
```

Chapitre 2

Vers la classe en C++

2.1 Des fonctions pour des structures

2.1.1 En C

En C, une structure contient des données appelées les champs de la structure. Des fonctions peuvent être mises en oeuvre pour gérer spécifiquement une structure. Le lien entre les fonctions et la structure est classiquement établit par le premier argument qui est un pointeur sur la structure.

Voici un exemple de définition d'une structure C et des fonctions associées pour la gestion de données concernant des personnes :

```
1 #ifndef __PERSONNE_H__
2 #define __PERSONNE_H__
3
4 #include <stdio.h>
```

```
5
6  typedef struct _personne
7  {
8    char * nom, *prenom, *adresse, * tel ;
9    int age ;
10 } TPersonne ;
11
12 void init_personne(TPersonne * p) ;
13 void detruire_presonne(TPersonne * p) ;
14 void ecrire_personne(TPersonne * p, FILE * f) ;
15 void lire_personne(TPersonne * p, FILE * f) ;
16 void affecter_nom_personne_avec(TPersonne * p, char * nom) ;
17
18 #endif /* __PERSONNE_H__ */
```

En voici une mise en oeuvre :

```
1  #include ”personne.h”
2
3  void init_personne(TPersonne * p)
4  {
5    p->nom = malloc(1) ;
6    *p->nom = 0 ;
7    p->prenom = malloc(1) ;
8    *p->prenom = 0 ;
9    p->adresse = malloc(1) ;
10   *p->adresse = 0 ;
11   p->tel = malloc(1) ;
12   *p->tel = 0 ;
13   p->age = 0 ;
14 }
15
16 void detruire_presonne(TPersonne * p)
17 {
18   free(p->nom) ;
19   free(p->prenom) ;
20   free(p->adresse) ;
21   free(p->tel) ;
22 }
23
24 void ecrire_personne(TPersonne * p, FILE * f)
25 {
26   fprintf(f, ”’%s’ :’%s’ :’%s’ :’%s’ :%d\n”,
27     p->nom, p->prenom, p->adresse, p->tel, p->age) ;
28 }
29
30 void lire_personne(TPersonne * p, FILE * f)
31 {
32   // la fonction lire_chaine lit une chaine entre deux □caractres ’\”
33   lire_chaine(f, &p->nom) ;
34   lire_car(f) ;
```

```
35    lire_chaine(f, &p->prenom) ;
36    lire_car(f) ;
37    lire_chaine(f, &p->adresse) ;
38    lire_car(f) ;
39    lire_chaine(f, &p->tel) ;
40    lire_car(f) ;
41    lire_entier(f, &p->age) ;
42  }
43
44  void affecter_nom_personne_avec(TPersonne * p, char * nom)
45  {
46    p->nom = realloc(p->nom, strlen(nom) + 1) ;
47    strcpy(p->nom, nom) ;
48  }
```

On obtient ainsi un ensemble proche de la définition de la classe *personne* avec les données et les fonctions associées :

- *init_personne* initialise une personne ; pour chaque variable, cette fonction doit être utilisée une fois, après la déclaration, pour assurer la validité des champs de la structure,
- *detruire_presonne* permet la destruction d'une personne (pour la récupération de la mémoire dynamique),
- *ecrire_personne* et *lire_personne* permettent l'écriture et la lecture d'une personne dans un fichier,
- *affecter_nom_personne_avec* permet la modification du nom d'une personne avec prise en charge de l'allocation dynamique.

Voici un exemple d'utilisation :

```
1   #include "personne.h"
2
3   main()
4   {
5     // Avec une variable de type TPersonne
6     TPersonne p ;
7     init_personne(&p) ; // les champs sont ☐initialiss
8     lire_personne(&p, stdin) ; // lecture des champs d'une personne sur l'☐entre standard
9     ecrire_personne(&p, stdout) ; // ecriture des champs sur la sortie standard
10    // Pour ☐accder au nom, on utilise directement le champ de la structure
11    printf("Nom  : %s\n", p.nom) ;
12    detruire_personne(&p) ; // ☐☐rcupration de la ☐mmoire dynamique ☐alloue
13
14    // Avec un pointeur sur un TPersonne
15    TPersonne * ptr = (TPersonne*) malloc(sizeof(TPersonne)) ;
16    init_personne(ptr) ; // les champs sont ☐initialiss
17    lire_personne(ptr, stdin) ; // lecture des champs d'une personne sur l'☐entre standard
```

```
18    ecrire_personne(ptr, stdout) ; // ecriture des champs sur la sortie standard
19    // Pour ☐accder au nom, on utilise directement le champ de la structure
20    printf(”Nom  : %s\n”, ptr->nom) ;
21    detruire_personne(ptr) ; // ☐☐rcupration de la ☐mmoire dynamique ☐alloue
22    free(ptr) ;
23  }
```

2.1.2 En C++

En C++, une structure peut encapsuler des fonctions. Les données et les fonctions de la structure sont appelées les membres de la structure. Une structure décrit ainsi une classe. Les fonctions membres peuvent être directement mises en oeuvre dans la déclaration de la structure.

Voici l’exemple de la structure *personne* avec la mise en oeuvre des fonctions membres dans la déclaration de la structure :

```
1   #ifndef __PERSONNE_H__
2   #define __PERSONNE_H__
3
4   #include <stdio.h>
5
6   struct TPersonne
7   {
8     char * nom, *prenom, *adresse, * tel ;
9     int age ;
10
11    TPersonne() // Le constructeur
12    {
13      nom = new char ;
14      *nom = 0 ;
15      prenom = new char ;
16      *prenom = 0 ;
17      adresse = new char ;
18      *adresse = 0 ;
19      tel = new char ;
20      *tel = 0 ;
21      age = 0 ;
22    }
23
24    ~TPersonne() // Le destructeur
25    {
26      delete nom ;
27      delete prenom ;
28      delete adresse ;
29      delete tel ;
30    }
31
32    void ecrire(FILE * f)
```

```
33    {
34      fprintf(f, "'%s' :'%s' :'%s' :'%s' :%d\n", nom, prenom, adresse, tel, age);
35    }
36
37    void lire(FILE * f)
38    {
39      // la fonction lire_chaine lit une chaine entre deux caractères '\"
40      lire_chaine(f, nom);
41      lire_car(f);
42      lire_chaine(f, prenom);
43      lire_car(f);
44      lire_chaine(f, adresse);
45      lire_car(f);
46      lire_chaine(f, tel);
47      lire_car(f);
48      lire_entier(f, age);
49    }
50
51    void affecter_nom_avec(char * n)
52    {
53      delete nom;
54      nom = new char [strlen(n)+1];
55      strcpy(nom, n);
56    }
57  };
58
59  #endif
```

On remarque que, d'un point de vue général, le nom des fonctions s'est simplifié : il ne contient plus la sous-chaîne «personne». La liaison entre la fonction membre et la structure est en effet maintenant indiquée par l'appartenance à la structure (à la classe).

Le paramètre de type *TPersonne* * qui, en C, est passé à chaque fonction est inutile en C++. En effet, dans le corps d'une fonction membre, l'objet courant peut être accédé au moyen du pointeur prédéfini *this* qui est initialisé avec l'adresse de l'objet courant au moment de l'appel d'une fonction membre. Pour une fonction membre d'une classe C, le pointeur prédéfini $this$ a pour type *C**. Une fonction membre peut soit directement accéder à une donnée ou à une fonction membre par son nom soit utiliser le pointeur *this* pour désigner l'objet courant. Par exemple, la fonction *affecter_nom_avec* donnée précédemment modifie la variable membre *nom*. Cette variable est accédée directement par son nom. La version suivante de cette même fonction utilise *this* et est strictement équivalente.

```
1   void affecter_nom_avec(char * nom)
2   {
3     this->nom = realloc(this->nom, strlen(this->nom) + 1);
4     strcpy(this->nom, nom);
```

```
5   }
```

le code des fonctions *init_personne* et *detruire_personne* se retrouve respectivement dans les fonctions membres *TPersonne* et *~TPersonne*. Ces deux fonctions sont particulières :

- la fonction qui porte le nom de la structure est appelée le constructeur ; son rôle est d'initialiser l'objet et son code est exécuté automatiquement lors de la création d'un objet (voir chapitre 3.2.1),
- la fonction qui porte de le nom de la structure précédé du *tilde* est appelée le destructeur ; son rôle est de détruire l'objet ; il est principalement utilisé pour récupérer la mémoire dynamique allouée pendant la durée de vie de l'objet et est exécuté lors de la destruction de l'objet en mémoire (voir chapitre 3.2.2).

Les fonctions membres peuvent aussi être mise en oeuvre dans un fichier séparé. Cette deuxième solution est préférable car elle permet la compilation séparée, le masquage de la mise en oeuvre, elle assure une meilleure lisibilité de la déclaration de la classe et une durée de compilation beaucoup moins longue. De plus, toute fonction dont le corps est développé dans la déclaration de la classe (comme c'est le cas dans l'exemple précédent) est implicitement déclarée «inline».

Par contre, le développement d'une classe en deux partie (la déclaration et la mise en oeuvre) est beaucoup plus fastidieux. Voici, pour le même exemple, la déclaration de la classe et la mise en oeuvre séparée des fonctions.

```
1   #ifndef __PERSONNE_H__
2   #define __PERSONNE_H__
3
4   #include <stdio.h>
5
6   struct TPersonne
7   {
8     char * nom, *prenom, *adresse, * tel ;
9     int age ;
10
11    TPersonne() ;
12    ~TPersonne() ;
13    void ecrire(FILE * f) ;
14    void lire(FILE * f) ;
15    void affecter_nom_avec(char * nom) ;
16  } ;
17
18  #endif
```

```
#include ”personne.h”

TPersonne ::
TPersonne()
{
  nom = new char;
  *nom = 0;
  prenom = new char;
  *prenom = 0;
  adresse = new char;
  *adresse = 0;
  tel = new char;
  *tel = 0;
  age = 0;
}

TPersonne ::
~TPersonne()
{
  delete nom;
  delete prenom;
  delete adresse;
  delete tel;
}

void TPersonne ::
ecrire(FILE * f)
{
  fprintf(f, ”’%s’ :’%s’ :’%s’ :’%s’ :%d\n”, nom, prenom, adresse, tel, age);
}

void TPersonne ::
lire(FILE * f)
{
  // la fonction lire_chaine lit une chaine entre deux □caractres ’\”
  lire_chaine(f, nom);
  lire_car(f);
  lire_chaine(f, prenom);
  lire_car(f);
  lire_chaine(f, adresse);
  lire_car(f);
  lire_chaine(f, tel);
  lire_car(f);
  lire_entier(f, age);
}

void TPersonne ::
affecter_nom_avec(char * n)
{
  delete nom;
  nom = new char [strlen(n)+1];
  strcpy(nom, n);
}
```

Dans la déclaration de la classe, la parenthèse qui ferme la déclaration des paramètres est suivie d'un ';'. Pour la mise en oeuvre d'une fonction membre, le nom de la classe suivi de ' ::' est indiqué juste après le type retourné par la fonction ou en premier pour les constructeurs et le destructeur.

Voici un exemple d'utilisation en C++ :

```
#include "personne.h"

main()
{
  // Avec un objet de type TPersonne
  TPersonne p ; // les champs sont □initialiss par le constructeur
  p.lire(stdin) ; // lecture des champs d'une personne sur l'□entre standard
  p.ecrire(stdout) ; // ecriture des champs sur la sortie standard
  // Pour □accder au nom, on peut toujours utiliser directement le champ de la structure
  printf("Nom  : %s\n", p.nom) ;

  // Avec un pointeur sur un TPersonne
  TPersonne * ptr = new TPersonne ; // les champs sont □initialiss par le constructeur
  ptr->lire(stdin) ; // lecture des champs d'une personne sur l'□entre standard
  ptr->ecrire(stdout) ; // ecriture des champs sur la sortie standard
  // Pour □accder au nom, on peut toujours utiliser directement le champ de la structure
  printf("Nom  : %s\n", ptr->nom) ;
  delete ptr ;
}
```

Le code du constructeur est exécuté automatiquement lorsqu'un objet est créé : ici, lors de sa déclaration. Pour indiquer l'appel d'une fonction membre, on utilise la notation pointée : le nom de l'objet suivi du point et du nom de la fonction avec les arguments. Si on manipule un pointeur sur un objet, alors on utilise la notation flèche (->).

2.2 Protection des membres d'une classe

Dans l'exemple de la classe *personne* donné précédemment, on observe qu'il est toujours possible d'accéder directement aux membres de la classe : toutes les fonctions et toutes les données. Il n'y a donc pas de protection des membres. Or, il est souvent très fortement souhaitable d'assurer la

protection de certaines fonctions et données membres. Observons l'exemple suivant :

```
#include "personne.h"

main()
{
  TPersone p ; // p.nom est □initialis
  affecter_nom_avec("Bibi Fricotin") ; // ok, nom est □□rallou
  fprintf("%s\n", p.nom) ; // c'est pas terrible mais ok
  free(p.nom) ; // Oups ! modification sans le controle d'une fonction membre
  // p est instable !!!
  affecter_nom_avec("Toto le coco") ; // Boum !, reallocation d'une pointeur □□libr
}
```

Cet exemple est valide du point de vue du compilateur par contre il ne l'est pas d'un point de vue fonctionnel. La protection des données membres permet d'éviter ce genre de situation à haut risque. Pour protéger une donnée ou une fonction membre, C++ dispose des mots clés *private* et *public* qui permettent respectivement d'interdire ou d'autoriser l'accès aux membres d'une classes. Ces deux mots clés permettent la définition de contextes privés ou publique :

- les membres spécifiés dans un contexte privé ne sont directement utilisables que par des fonctions membres privées ou publiques de la même classe,
- les membre spécifiés dans un contexte publique sont directement utilisables par toutes les fonctions membre de toutes les classes et par les fonctions globales.

Par défaut, dans une structure, le contexte est publique.

Voici notre classe *personne* revue pour assurer la protection des membres de la classe. Seul la déclaration de la classe doit être modifiée :

```
#ifndef __PERSONNE_H__
#define __PERSONNE_H__

#include <stdio.h>

struct TPersonne
{
private :
  char * nom, *prenom, *adresse, * tel ;
  int age ;

public :
  TPersonne() ;
  ~TPersonne() ;
  void ecrire(FILE * f) ;
```

```
16    void lire(FILE * f);
17    void affecter_nom_avec(char * nom);
18 };
19
20 #endif
```

2.3 Les accesseurs

2.3.1 Les accesseurs en lecture

Reprenons maintenant une partie de l'exemple d'utilisation donné précédemment.

```
1 #include "personne.h"
2
3 main()
4 {
5    TPersonne p;
6    p.lire(stdin);
7    p.ecrire(stdout);
8    printf("Nom  : %s\n", p.nom); // Erreur de compilation  : p.nom est prive
9 }
```

L'accès directe au nom de la personne est maintenant interdit puisque la variable membre *nom* est définie dans une contexte privé. La ligne 8 provoque une erreur de compilation. La mise en oeuvre d'une telle classe nécessite alors l'ajout de fonctions publiques particulières, couramment appelées les accesseurs en lecture. Ces fonctions permettent l'accès en lecture seule des variables membres autorisées à être lues depuis les fonctions extérieures à la classe. Pour notre classe personne, un accesseur en lecture serait classiquement mis en oeuvre pour chacune des cinq variables membres.

Voici la version de la classe avec les accesseurs en lecture :

```
1 #ifndef __PERSONNE_H__
2 #define __PERSONNE_H__
3
4 #include <stdio.h>
5
6 struct TPersonne
7 {
8 private :
```

```
9     char * nom, *prenom, *adresse, * tel ;
10    int age ;
11
12  public :
13    TPersonne() ;
14    ~TPersonne() ;
15
16    void ecrire(FILE * f) ;
17    void lire(FILE * f) ;
18
19    // Accesseurs en lecture
20    const char * nom () { return nom ; }
21    const char * prenom () { return prenom ; }
22    const char * adresse () { return adresse ; }
23    const char * tel () { return tel ; }
24    int age () { return age ; }
25  } ;
26
27  #endif
```

Chaque accesseur retourne directement la variable membre. Le type de retour est constant : un pointeur constant, une référence constante ou une copie (pour un entier par exemple). Ainsi, les données membres sont protégées contre des modifications directes. On remarque aussi que les accesseurs sont mis en oeuvre dans la déclaration de la classe. C'est largement admis car la fonction ne comporte qu'une seule instruction (le *return*).

2.3.2 Les accesseurs en écriture

Dans la classe *personne*, la fonction *affecter_nom_avec* à pour rôle de modifier la variable membre privée *nom*. L'intéret de cette fonction est qu'elle prend en charge la réallocation de la variable membre avant la copie du nom passé en argument. Cette fonction est couramment appelée un accesseur en écriture. Ces accesseurs sont développés pour permettre la modification des variables membres. Pour notre classe personne, un accesseur en écriture serait classiquement mis en oeuvre pour chacune des cinq variables membres.

Voici la version de la classe avec les accesseurs en écriture :

```
1   #ifndef __PERSONNE_H__
2   #define __PERSONNE_H__
3
4   #include <stdio.h>
5
6   struct TPersonne
```

```
{
private :
  char * nom, *prenom, *adresse, * tel;
  int age;

public :
  TPersonne();
  ~TPersonne();

  void ecrire(FILE * f);
  void lire(FILE * f);

  // Accesseurs en ecriture
  void affecter_nom_avec(const char *);
  void affecter_prenom_avec(const char *);
  void affecter_adrese_avec(const char *);
  void affecter_tel_avec(const char *);
  bool affecter_age_avec(int);

  // Accesseurs en lecture
  const char * nom () { return nom; }
  const char * prenom () { return prenom; }
  const char * adresse () { return adresse; }
  const char * tel () { return tel; }
  int age () { return age; }
};

#endif
```

Voici la mise en oeuvre des accesseurs en écriture.

```
#include "personne.h"

void TPersonne ::
affecter_nom_avec(char * n)
{
  delete nom;
  nom = new char [strlen(n)+1];
  strcpy(nom, n);
}

void TPersonne ::
affecter_prenom_avec(char * p)
{
  delete prenom;
  prenom = new char [strlen(p)+1];
  strcpy(prenom, p);
}

void TPersonne ::
affecter_adrese_avec(char * a)
{
```

```
22    delete adresse;
23    adresse = new char [strlen(a)+1];
24    strcpy(adresse, a);
25  }
26
27  void TPersonne ::
28  affecter_tel_avec(char * t)
29  {
30    delete tel;
31    tel = new char [strlen(t)+1];
32    strcpy(tel, t);
33  }
34
35  bool TPersonne ::
36  affecter_age_avec(int a)
37  {
38    if (a > 0 && a < 160) {
39      age = a;
40      return true;
41    } else {
42      return false;
43    }
44  }
```

Dans l'accesseurs *affecter_age_avec*, un contrôle sur la validité du paramètre est effectué. Il s'agit en effet d'une possibilité qui est souvent utilisée pour les accesseurs en écriture : avant la modification de la variable membre, un contrôle de validité du paramètre peut être effectué afin de garantir la qualité des données membres. Dans le cas de notre fonction *affecter_age_avec*, la valeur retournée est *true* si le paramètre est correcte sinon *false*.

2.4 Quelques tours de boulon...

2.4.1 Les paramètres constants

```
1   #include "personne.h"
2
3   main()
4   {
5     FILE * fpers = fopen ("adresses.lst", "r"); // ouverture de mon fichiers
6     if (fpers) {
7       TPersonne p1; // construction d'un TPersonne
8       p1.lire(fpers); // lecture de la premiere adresse dans le fichier
9       TPersonne p2; // construction d'un autre TPersonne
10      p2.affecter_nom_avec(p1.nom()); // on veut copier le nom -> erreur de compilation
11      p2.ecrire(stdout); // envoit de p2 sur la sortie standard
```

```
12      fclose(fpers) ; // fermeture du fichier
13    } else {
14      // traitement de l'erreur
15    }
16    return 0 ;
17  }
```

Cet exemple montre un cas simple d'utilisation : on désire affecter le nom d'une personne dans celui d'une autre personne. Or, la compilation de cet exemple provoque une erreur ligne 10 : le type de retour de p1.nom() est *const char* * or, la fonction *affecter_nom_avec* a pour signature :

```
1   void affecter_nom_avec(char *) ;
```

Au regard de la mise en oeuvre de cet accesseur, la chaîne de caractères passée en argument n'est pas modifiée par la fonction. On peut donc déclarer cet accesseur avec un paramètre constant (*const char* *).

En règle générale, tout paramètre pointeur ou référence doit être déclaré constant lorsqu'il n'est pas modifié par la fonction. Voici donc la nouvelle version de la classe *personne* :

```
1   #ifndef __PERSONNE_H__
2   #define __PERSONNE_H__
3
4   #include <stdio.h>
5
6   struct TPersonne
7   {
8   private :
9     char * nom, *prenom, *adresse, * tel ;
10    int age ;
11
12  public :
13    TPersonne() ;
14    ~TPersonne() ;
15
16    void ecrire(FILE * f) ;
17    void lire(FILE * f) ;
18
19    // Accesseurs en ecriture
20    void affecter_nom_avec(const char *) ;
21    void affecter_prenom_avec(const char *) ;
22    void affecter_adrese_avec(const char *) ;
23    void affecter_tel_avec(const char *) ;
24    bool affecter_age_avec(int) ;
25
26    // Accesseurs en lecture
27    const char * nom () { return nom ; }
28    const char * prenom () { return prenom ; }
```

```
29    const char * adresse () { return adresse ; }
30    const char * tel () { return tel ; }
31    int age () { return age ; }
32  } ;
33
34  #endif
```

2.4.2 Les fonctions membres constantes

```
1   #include "personne.h"
2
3   void afficher_telephone(const TPersonne * p)
4   {
5     printf("%s tel %s", p->nom(), p->tel()) ; // erreur de compilation
6   }
7
8   void afficher_telephone(const TPersonne & p)
9   {
10    printf("%s tel %s", p.nom(), p.tel()) ; // erreur de compilation
11  }
12
13  main()
14  {
15    FILE * fpers = fopen ("adresses.lst", "r") ; // ouverture de mon fichiers
16    if (fpers) {
17      TPersonne p ; // construction d'un TPersonne
18      p.lire(fpers) ; // lecture de la premiere adresse dans le fichier
19      afficher_telephone(*p) ;
20      afficher_telephone(p) ;
21      fclose(fpers) ; // fermeture du fichier
22    } else {
23      // traitement de l'erreur
24    }
25    return 0 ;
26  }
```

Les deux fonctions *afficher_telephone* prennent en argument un paramètre constant : dans la première version, un pointeur constant et dans la deuxième, une référence constante. Ce choix se justifie car aucune de ces versions ne modifie le TPersonne qui lui est passé en argument. Or, le compilateur indique une erreur ligne 5 et 10. En effet, le paramètre étant constant il est impossible de le modifier et l'exécution de p->nom() et p->tel() par exemple, doit garantir la non modification de $*p$. Or ce n'est pas le cas. Pour ce faire, ces fonctions membres doivent être déclarées comme des fonctions membres constantes.

En règle générale, toute fonction membre qui ne modifie pas l'objet courant $(*this)$ doit être déclarée comme une fonction membre constante. Ceci s'effectue en ajoutant le mot clé $const$ après la parenthèse qui ferme la déclaration des paramètres, dans la déclaration de la classe et aussi dans sa mise en oeuvre. Dans une fonction membre constante d'une classe C, $this$ à pour type *const C* *. Une fonction membre déclarée constante ne peut faire appel qu'à des fonctions elles même déclarées constantes. Voici la nouvelle version de la classe *personne* :

```
#ifndef __PERSONNE_H__
#define __PERSONNE_H__

#include <stdio.h>

struct TPersonne
{
private :
   char * nom, *prenom, *adresse, * tel;
   int age;

public :
   TPersonne();
   ~TPersonne();

   void ecrire(FILE * f) const;
   void lire(FILE * f);

   // Accesseurs en ecriture
   void affecter_nom_avec(const char *);
   void affecter_prenom_avec(const char *);
   void affecter_adrese_avec(const char *);
   void affecter_tel_avec(const char *);
   bool affecter_age_avec(int);

   // Accesseurs en lecture
   const char * nom () const { return nom; }
   const char * prenom () const { return prenom; }
   const char * adresse () const { return adresse; }
   const char * tel () const { return tel; }
   int age () const { return age; }
};

#endif
```

2.4.3 Les fonctions membres privées

Jusqu'à présent, la ligne directrice qui ressort des différents exemples donnés consiste à spécifier les données en privé, à développer les accesseurs et les autres fonctions membres en public. C'est un point de départ.

Cependant, il est aussi très courant de mettre en oeuvre des fonctions membres privées. L'idée est de bénéficier de services pour les autres fonctions de la classe. Etant privée, elles ne peuvent en effet pas être exécutée depuis une d'autres fonctions autres que celles de la classe.

Dans notre exemple de la classe *personne*, on observe que les accesseurs en écriture ont en commun une grande partie de leur code. Une solution acceptable serait de factoriser ces lignes communes dans la mise en oeuvre d'une fonction membre privée. Voici la nouvelle version de la déclaration de la classe avec la fonction membre privée *reaffecter_chaine* :

```
#ifndef __PERSONNE_H__
#define __PERSONNE_H__

#include <stdio.h>

struct TPersonne
{
private :
  char * nom, *prenom, *adresse, * tel ;
  int age ;

public :
  TPersonne() ;
  // et le reste ...

private :
  void reaffecter_chaine(char *& dest, const char * src) ;

} ;

#endif
```

On remarque que le premier paramètre de *reaffecter_chaine* est de type référence sur un pointeur (char *&) ce qui permet de modifier le pointeur passé en argument. Voici un bout de la mise en oeuvre de la classe *personne*, avec la fonction *reaffecter_chaine* :

```
#include "personne.h"

void TPersonne ::
reaffecter_chaine(char *& dst, const char * src)
{
  delete dst ;
  dst = new char [strlen(src)+1] ;
  strcpy(dst, src) ;
```

```
}

void TPersonne ::
affecter_nom_avec(const char * n)
{
  reaffecter_chaine(nom, n);
}

void TPersonne ::
affecter_prenom_avec(const char * p)
{
  reaffecter_chaine(prenom, p);
}

// et le reste ...
```

Chapitre 3

La classe

3.1 Présentation générale d'une classe C++

3.1.1 Le mot réservé *class*

Le mot réservé *class* permet la définition d'une classe en C++. [1] La classe contient la déclaration des données et des fonctions membres. Dans la grande majorité des cas, les données membres ne sont accessibles que par les fonctions membres. Par comparaison à une structure, le contexte

1. c'est comme une structure sauf qu'à la place de *struct*, on met *class*

par défaut est privé, de plus, une classe définie avec le mot réservé *class* autorise l'héritage.

3.1.2 Des rappels

le mot réservé *public* permet de rendre publique (accessible à tous) des données ou des fonctions membres. En règle quasi générale, seules les fonctions membres sont rendues publiques. A l'inverse, le mot clé *private* permet de rendre privées des données ou des fonctions membres qui ne sont alors visibles que dans la mise en oeuvre des fonctions membres de la classe.

Exemple : déclaration de la classe *Values*

```
1  #ifndef __VALUES_H__
2  #define __VALUES_H__
3
4  class Values
5  {
6    // ☐donne membre implicitement ☐prive (aucun indicateur de ☐porte n'est ☐indiqu)
7    int ival;
8  private :
9    // ☐donne membre explicitement ☐prive (indicateur de ☐porte private ☐indiqu)
10   char cval;
11
12 public :
13   // fonctions membres publiques (Indicateur de ☐porte public ☐indiqu)
14   Values (); // constructeur
15   -Values (); // destructeur
16
17   // accesseurs en ☐criture
18   void set_val(int);
19   void set_val(char);
20
21   // accesseurs en lecture
22   int get_ival() const;
23   char get_cval() const;
24 };
25
26 #endif __VALUES_H_
```

La mise en oeuvre d'une classe fournit les définitions des fonctions membres. On indique le nom de la classe correspondant suivi de l'opérateur de portée ::. Dans la mise en oeuvre des fonctions membres, toutes les données membres (publiques ou privées) sont directement accessibles par leur nom.

Exemple : mise en oeuvre de la classe *Values*

```
1  #include "values.hh"
2
3  // On utilise l'□oprateur de □porte :: □□□prcd du nom de la classe pour □spcifier
4  // qu'il s'agit de la fonction membre de la classe
5
6  Values ::
7  Values()
8  {
9    // □Accs direct aux □donnes membres □prives ou publiques de la classe
10   ival = 0;
11   cval = 'A';
12 }
13
14 Values ::
15 -Values()
16 {
17 }
18
19 void Values ::
20 set_val(int _v)
21 {
22   ival = _v;
23 }
24
25 void Values ::
26 set_val(char _v)
27 {
28   cval = _v;
29 }
30
31 int Values ::
32 get_ival() const
33 {
34   return ival;
35 }
36
37 char Values ::
38 get_cval() const
39 {
40   return this->cval; // Utilisation de l'□□autorfrence, ici □quivalente a "return cval";
41 }
```

La première fonction membre *Values()*, porte le nom de la classe. Il s'agit d'un constructeur. Le corps de cette fonction est exécuté lors de la création d'un objet. Dans le cas de la classe *Values*, le constructeur met la valeur 0 dans la donnée membre privée *ival* et la valeur $'A'$ dans la donnée membre privée *cval*.

La deuxième fonction membre *~Values()*, porte le nom de la classe précédé du caractère ~. Cette fonction membre est appelée lors de la mort

d'un objet, en l'occurrence à la fermeture du bloc constituant la fonction *main()*.

Les deux fonctions membres *set_val* sont des accesseurs en écriture. Leur nom est préfixé par convention par set_. Ces fonctions modifient l'état de l'objet (les valeurs des données membres).

Les deux fonctions membres *get_cval* et *set_ival* sont des accesseurs en lecture. Leur nom est préfixé par convention par get_. Ces fonctions permettent l'accès en lecture des valeurs des données membres. Ces deux fonctions membres sont déclarées constantes par le mot clé *const* en fin de prototype, après la déclaration de la liste (vide) des paramètres. Cela garanti à l'utilisateur de la classe que l'exécution de la fonction membre ne modifie pas l'état de l'objet (les données membres). Une fonction membre déclarée constante ne peut faire appel qu'à des fonctions membres constantes.

Dans la mise en oeuvre des fonctions membres, le mot clé réservé *this* spécifie l'adresse de l'objet qui exécute la fonction (l'objet courant). L'utilisation du pointeur constant *this* est illustrée par la mise en oeuvre de la fonction membre *get_cval()*.

Voici un exemple d'utilisation de la classe *Values* :

```
1  #include <stdio.h>
2  #include "values.hh"
3
4  main(int, char**)
5
6  {
7    Values v ; // □Dclaration d'un objet de la classe Values -> □excution du constructeur
         pour v
8    printf ("ival %d, cval %c\n", v.get_ival(), v.get_cval()) ; // Affiche ival 0, cval A
9
10   // modification de l'objet par les accesseurs en □criture
11   v.set_val(2) ;
12   v.set_val('X') ;
13   printf ("ival %d, cval %c\n", v.get_ival(), v.get_cval()) ; // Affiche ival 2, cval X
14
15   // acces directe □une □donne membre
16   printf("ival %d\n", v.ival) ; // Erreur de compilation, ival est □prive
17
18   return 0 ;
19 } // Fin de bloc, □excution du destructeur ~Values() pour l'objet v
```

3.1.3 Les fonctions et classes amies

Normalement, les membres privés d'une classe ne sont accessibles que par les fonctions membres de la classe. Dans certains cas bien particuliers, une classe peut autoriser soit une(des) fonction(s) soit une(des) autre(s)

classe(s) à accéder directement à ses membres privés. Pour cela, dans la déclaration de la classe, on dispose du mot clé *friend*.

Voici un exemple :

```
1  // La class A □dclare deux amis  : la classe B_amie et la fonction affa_amie(const A&)
2  class A
3  {
4  public :
5    A()  : a(0) {}
6    friend class B_amie ;
7    friend void affa_amie(const A &) ;
8  private :
9    int a ;
10 } ;
11
12 // la classe B utilise directement le contenu de la variable
13 // membre □prive a.a ; c'est une erreur, les membres □privs de la classe A
14 // ne sont accessibles que par les fonctions de la classe A
15 class B
16 {
17 public :
18   B(A & a)  : ba(a.a) {} // Erreur, a.a est privee dans A
19 private :
20   int ba ;
21 } ;
22
23 // la classe B_amie est □□dclare amie de A, elle peut
24 // directement utiliser les membres □privs de A
25 // ici, on accede directement □la variable membre □prive a.a
26 // dans le constructeur
27 class B_amie
28 {
29 public :
30   B(A & a)  : ba(a.a) {} // ok ; B_amie est □□dclare "friend" dans A
31 private :
32   int ba ;
33 } ;
34
35 // affa(const A &) affiche directement le contenu de la variable
36 // membre □prive a.a ; c'est une erreur, les membres □privs de la classe A
37 // ne sont accessibles que par les fonctions de la classe A
38 void affa(const A & a)
39 {
40   cerr << a.a << endl ; // Erreur, a.a est privee dans A
41 }
42
43 // affa_amie(const A &) est amie de A, elle peut
44 // directement utiliser les membres □privs de A
45 // ici, on accede directement □la variable membre □prive a.a
46 void affa_amie(const A & a)
47 {
48   cerr << a.a << endl ; // ok ; affa est □□dclare "friend" dans A
49 }
```

Mais attention, en C++, si une classe A déclare la classe B amie, alors la classe A n'est pas implicitement déclarée amie de B. De plus, l'utilisation des «amis» casse l'encapsulation et la protection par contexte privé. On peut se trouver dans une situation inextricable, de classes fortement et intimement dépendantes les unes des autres d'une façon qui n'a rien à voir avec une conception propre de l'objet. On peut cependant dégager deux cas d'utilisation classiques :

- soit une classe D qui ne peut être instanciée que comme variable membre d'une classe A ; la raison peut être par exemple, que D comporte des variables membres que seule la classe A est en mesure d'initialiser (un cas d'aggrégation par exemple) ; la solution pour traduire cet état de fait est de déclarer les constructeurs/destructeur de D en privé (impossible de créer directement un objet D) est de déclarer la classe A amie de D ; la classe A peut ainsi créer un D et accéder aux membres privés de D ;
- le deuxième cas concerne très précisément les opérateurs externes aux classes ; il est souvent très commode de les déclarés amis dans les classes des opérandes ;

Comme règle de base, il est conseillé de ne pas utiliser les fonctions ou classes amies. Cela fait partie des partcularités du C++ qui compliquent la maintenance et dont on peut très bien se passer.

3.1.4 Qualification des données membres

Dans le cas où il y a ambiguïté entre par exemple le nom d'une donnée membre et le nom d'un paramètre, il est possible d'utiliser soit l'opérateur de portée ' ::' soit l'autoréférence avec le pointeur prédéfini this.

Voici un exemple :

```
1  class C
2  {
3    int i;
4    double d;
5  public :
6    C(int i)  : d(0.0)
7    {
8      this->i = i; // □□autorfrence pour lever l'□□ambigut
9    }
10   C(int i, double d)  : C ::i(i), C ::d(d) {} // □oprateur de □porte pour lever l'
         □□ambigut
11 };
```

3.1.5 Classe et allocation dynamique

Comme nous l'avons vu dans le paragraphe 1.2.9, C++ comprend les deux nouveaux opérateurs new et $delete$ pour l'allocation et la libération de la mémoire. Pour une classe, la syntaxe est la même que pour un type simple.

3.1.5.1 Allocation de la mémoire

new C : pour une classe *C*, *new C* renvoie l'adresse d'une zone mémoire allouée. La taille de la zone est *sizeof(C)*. Le type de retour est *C**.

new C[N] : pour une classe *C*, *new C[N]* renvoie l'adresse d'une zone mémoire allouée. La taille de la zone est *sizeof(C) * N*. Le type de retour est *C**.

3.1.5.2 Libération de la mémoire

delete cptr : pour une variable *cptr*, pointeur sur un objet de la classe *C*, *delete cptr* permet la désallocation de la mémoire pointée par *cptr*, soit *sizeof(C)* octets en mémoire dynamique alloués précédemment par *new C*.

delete [] ctab : pour une variable *ctab*, pointeur sur un tableau de N éléments de type *C*, *delete ctab* désalloue la mémoire pointée par *ctab*, soit *sizeof(C) * N* octets en mémoire dynamique alloués précédemment par *new C[N]*.

3.1.5.3 Particularités de l'allocation pour une classe

Pour les classes, les opérateurs new et $delete$ permettent plus qu'une simple allocation ou qu'une simple libération de mémoire :

- l'opérateur new connaît le type de l'objet pour lequel on veut allouer de la mémoire ; il connaît la taille du type et donc celle de la zone mémoire à allouer ; par contre, avec $malloc$, on doit utiliser l'expression $sizeof$ pour indiquer à malloc la taille à allouer,
- la fonction $malloc$ ne fait qu'allouer de la mémoire ; la mémoire allouée n'est pas initialisée ; la seule façon d'initialiser la mémoire allouée est d'utiliser calloc qui afecte 0 à tous les octets alloués ; L'allocation de mémoire pour une classe avec new est automatiquement suivie de l'exécution d'un constructeur de la classe (voir chapitre 3.2.1), ainsi, la mémoire allouée est initialisée par une fonction de la classe ;
- en C, on doit toujours contrôler le retour de la fonction d'allocation ($malloc$, $realloc$, $calloc$) pour gérer une éventuelle erreur d'allocation ; en C++, lorsque plus de mémoire n'est disponible, la fonction installée par $set\_new\_handler$ (voir chapitre 1.2.9.3) est automatiquement exécutée.

– la libération de mémoire avec $free$ est automatiquement précédée de l'exécution du destructeur (voir chapitre 3.2.2) de la classe pour laquelle la mémoire a été allouée.

3.2 Constructeurs et destructeur

3.2.1 Les constructeurs

3.2.1.1 Déclaration et exécution

Un constructeur est une fonction membre qui porte le même nom que sa classe. Pour un constructeur, on ne précise pas le type de retour. Dans son code on ne peut pas utiliser $return$. Son but est d'initialiser l'espace mémoire que l'objet occupe. Un constructeur ne renvoie aucune valeur, en conséquence, aucune indication de type de retour n'est spécifiée.

Plusieurs constructeurs peuvent être spécifiés en surcharge. Ils se différencient alors par le type des paramètres. Un constructeur peut prévoir des valeurs par défaut pour ses paramètres.

Le constructeur d'un objet est automatiquement exécuté :

- lors de la déclaration d'un objet automatique, statique ou global, pour initialiser la mémoire réservée (dans la pile ou en mémoire globale),
- lors de la déclaration d'un tableau d'objets automatique, statique ou global, pour initialiser chacun des objets du tableau,
- après l'allocation dynamique d'un objet, pour initialiser la mémoire allouée,
- après l'allocation dynamique d'un tableau d'objets, pour initialiser chacun des objets du tableau.

On peut indiquer quel constructeur utiliser pour la création d'un objet, en précisant les arguments du constructeur. Ceci n'est pas possible pour la déclaration de tableaux d'objets : pour chacun des objets du tableau, c'est le constructeur sans paramètre qui est exécuté.

Voici un exemple :

```
1 #include <iostream.h>
2
3 class Point
4 {
5 public :
```

```
6     Point() { x = 0 ; y = 0 ; } // constructeur sans argument
7     Point(int _x, int _y) { x = _x ; y = _y ; } // constructeur avec arguments
8     // Accesseurs en lecture
9     int get_x() { return x ; }
10    int get_y() { return y ; }
11
12  private :
13    int x, y ;
14  } ;
15
16  main()
17  {
18    // Pour des points
19    Point p1 ; // Appel du constructeur Point()
20    cout << p1.get_x() << " : " << p1.get_y() ; // affiche '0 : 0'
21
22    Point p2(10, 25) ; // Appel du constructeur Point(int,int)
23    cout << p2.get_x() << " : " << p2.get_y() ; // affiche '10 : 25'
24
25    const Point p2c(10, 25) ; // Appel du constructeur Point(int,int) pour un objet constant
26    cout << p2c.get_x() << " : " << p2c.get_y() ; // affiche '10 : 25'
27
28    Point * p3 = new Point (5, 6) ; // Allocation et appel du constructeur Point(int,int)
29    cout << p3->get_x() << " : " << p3->get_y() ; // affiche '5 : 6'
30
31    Point * p4 = &Point(8, 9) ; // Construction d'un objet anonyme et récupération de son
          adresse
32    cout << p4->get_x() << " : " << p4->get_y() ; // affiche '8 : 9'
33
34    // Pour des tableaux de points
35    Point tp1[10] ; // pour chaque point, appel du constructeur Point()
36    Point * tp2 = new Point [10] ; // Pour chaque point, appel du constructeur Point()
37    Point ** tp3 = new Point * [10] ; // Pas d'appel de constructeur (tableau de pointeurs)
38    // Exemple d'initialisation d'un tableau de pointeurs de Point
39    for (int i = 0 ; i < 10 ; i++) {
40      tp3[i] = new Point ;
41    }
42  }
```

Attention, la déclaration d'un objet global entraîne l'exécution du constructeur avant le début du programme principal. En C++, du code peut donc être éxécuté au lancement d'un programme, même si le corps de la fonction main() est vide !

Lorsqu'une classe est spécifiée avec un constructeur prenant un argument, il est possible d'utiliser l'affectation dans la déclaration avec en partie droite, une expression dont le type est compatible avec le type de l'argument du constructeur. Il ne s'agit cependant pas d'une affectation car il n'y a pas exécution de l'opérateur '='. Il s'agit bel et bien d'une déclaration

et c'est le constructeur qui est appelé :

```
#include <iostream.h>

class C {
public :
  C(int unEntier) { i = unEntier ; cerr << "C(" << i << ")" << endl ; }
  C & operator = (const C & _c) { i = _c.i ; cerr << "C& operator = (const C & _c)" <<
      endl ; }
private :
  int i ;
} ;

main(int, char**) {
  C c1(2) ; // Exécution du constructeur C(2)
  C c2 = 3 ; // ATTENTION  : Exécution du constructeur C(3)  !!!
  c1 = c2 ; // Affectation entre objets
}
```

Voici le résultat de l'exécution de ce programme :

```
~/COURS/C++/latex/SOURCES> a.out
C(2)
C(3)
C& operator = (const C & _c)
~/COURS/C++/latex/SOURCES>
```

3.2.1.2 Initialisation des variables membres

C++ prévoit une syntaxe particulière pour l'initialisation des variables membres. Dans les constructeurs de l'exemple précédent, les variables membres sont initialisées dans le code même des constructeurs. Il est possible, souvent souhaitable et parfois même indispensable (voir chapitre 3.4.1) d'initialiser les variables membres avant le début du code du constructeur : entre la parenthèse qui ferme la déclaration des paramètres et l'accolade ouvrante de début du code. Voici l'exemple de la classe *Point* revue pour tenir compte de cette possibilité.

```
// Si la mise en oeuvre est dans la □dclaration de la classe  :
class Point {
public :
  Point()  : x(0), y(0) { } // constructeur sans argument
  Point(int _x, int _y)  : x(_x), y(_y) { } // constructeur avec arguments
private :
  int x,y ;
} ;

// Si la mise en oeuvre est □□spare  :
```

```
class Point {
public :
   Point() ;
   Point(int _x, int _y) ;
private :
   int x,y ;
} ;

Point ::
Point()  : x(0), y(0)
{
}

Point ::
Point(int _x, int _y)  : x(_x), y(_y)
{
}
```

Les variables membres constantes ne peuvent être initialisées que dans le(s) constructeur(s) et la syntaxe précisée précédemment doit obligatoirement être utilisée. En effet, étant constante, la variable membre ne peut constituer la partie gauche d'une affectation. Donc, une modification de sa valeur dans un bloc (dans le corps du constructeur ou dans le corps d'un accesseur en écriture) est une erreur. Voici un exemple :

```
#include <iostream.h>

class c_ok
{
public :
   const int x ;
   c_ok()  : x(0) {} // Initialisation ok
   c_ok(int _x)  : x(_x) {} // ok
} ;

#if 0
class c_ko
{
public :
   const int x ;
   c_ko() { x = 0 ; } // erreur : affectation interdite
   c_ko(int _x) { x = _x ; } // meme erreur
   void set_x(int _x) { x = _x ; } // meme erreur
} ;
#endif

main()
{
   c_ok v ;
   cerr << v.x ; // affiche '0'
```

```
26    c_ok v2(999);
27    cerr << v2.x; // affiche '999'
28  }
```

Les variables membres qui sont des références ne peuvent être initialisées que dans le(s) constructeur(s). En effet, c'est à la création de l'objet qu'on peut établir le lien entre la référence et l'objet ou la variable existante dont la variable membre est l'alias. Si tel n'etait pas le cas, alors cela signifirait qu'une référence pourrait exister sans lien avec une variable existante.

Comme pour les variables membres constantes, la syntaxe précisée précédemment doit obligatoirement être utilisée. Voici un exemple :

```
1   #include <iostream.h>
2
3   #if 0
4   class c_ko1
5   {
6   public :
7     int & x;
8     c_ko1() {} // Erreur : x n'est pas initialise (pas d'alias)
9   };
10
11  class c_ko2
12  {
13  public :
14    int & x;
15    c_ko2()  : x(0) {} // Erreur : x ne peut pas être initialisé avec une valeur
16    c_ko2(int _x)  : x(_x) {} // Erreur : on initialise x avec une copie sur la pile
17    c_ko2(int & _x) { x = _x; } // Erreur : initialisation dans un bloc (reaffectation)
18    void set_x(int & _x) { x = _x; } // meme erreur (reaffectation)
19  };
20  #endif
21
22  class c_ok
23  {
24  public :
25    int & x;
26    c_ok(int & _x)  : x(_x) { } // OK
27    // c_ok ne peut pas avoir de constructeur sans argument
28  };
29
30  main()
31  {
32  #if 0
33    c_ok v(999); // Erreur, on doit passer une variable
34    c_ok v2; // Erreur, pas de constructeur sans argument
```

```
35    c_ok tab[10]; // Erreur, pas de constructeur sans argument
36  #endif
37    int vx = 999;
38    c_ok v2(vx);
39    cerr << v2.x; // affiche '999'
40  }
```

En conséquence, comme il est montré dans l'exemple, une classe avec une(des) variable(s) membre(s) référence(s) ne peut pas comporter de constructeur sans argument. On ne peut donc pas déclarer ou allouer de tableau pour une telle classe.

3.2.1.3 Constructeur par défaut

Si aucun constructeur n'est spécifié pour une classe, alors un constructeur par défaut, sans argument, est mis en place par le compilateur. Ce constructeur ne fait rien. Par contre, si un constructeur est spécifié pour la classe, alors aucun constructeur par défaut n'est mis en place. Voici une illustration :

```
1   #include <iostream.h>
2
3   class sans_constructeur
4   {
5   public :
6     int x;
7   };
8
9   class avec_constructeur_ko
10  {
11  public :
12    avec_constructeur_ko(int _x)  : x(_x) {}
13  public :
14    int x;
15  };
16
17  class avec_constructeur_ok1
18  {
19  public :
20    avec_constructeur_ok1(int _x = 0)  : x(_x) {}
21  public :
22    int x;
23  };
24
25  class avec_constructeur_ok2
26  {
27  public :
28    avec_constructeur_ok2()  : x(0) {}
```

```
29    avec_constructeur_ok2(int _x)  : x(_x) {}
30  public :
31    int x;
32  };
33
34  main()
35  {
36    sans_constructeur sc1; // OK, le constructeur par □dfaut est □□excut
37    cerr << sc1.x; // il y a n'importe quoi dans x
38    sans_constructeur sc1_tab[10]; // OK
39    sans_constructeur * sc1_ptr = new sans_constructeur[10]; // OK
40
41    avec_constructeur_ko sc1_ko; // Erreur de compilation , pas de constructeur sans
          argument
42    avec_constructeur_ko sc1_ko_tab[10]; // meme erreur
43    avec_constructeur_ko * sc1_ko_ptr = new avec_constructeur_ko[10]; // meme erreur
44    avec_constructeur_ko * sc1_ko_ptr2 = new avec_constructeur_ko(3); // OK
45    avec_constructeur_ko sc1_ko2(3); // OK
46
47    avec_constructeur_ok1 sc1_ok1; // OK
48    cerr << sc1.x; // affiche bien '0'
49    avec_constructeur_ok1 sc1_ok1_tab[10]; // OK
50    avec_constructeur_ok1 * sc1_ok1_ptr = new avec_constructeur_ok1[10]; // OK
51
52    avec_constructeur_ok2 sc1_ok2; // OK
53    cerr << sc1.x; // affiche bien '0'
54    avec_constructeur_ok2 sc1_ok2_tab[10]; // OK
55    avec_constructeur_ok2 * sc1_ok2_ptr = new avec_constructeur_ok2[10]; // OK
56  }
```

Dans cet exemple, on observe que :

- si on défini un constructeur avec argument(s) dans une classe, suivant l'utilisation désirée de cette classe, il peut alors devenir indispensable de spécifier un constructeur sans argument. C'est par exemple le cas si on veut déclarer un objet sans passer d'argument au constructeur ou si on veut déclarer un tableau d'objets ou encore allouer un tableau d'objets.

- un constructeur avec argument(s) et avec une valeur par défaut pour chaque argument est considéré aussi comme un constructeur sans argument.

Il en ressort qu'il est préférable de ne pas se poser de question et de systématiquement mettre en oeuvre un constructeur sans argument.

3.2.1.4 Constructeur par copie

Le constructeur d'une classe qui a pour unique paramètre la référence d'un objet de cette même classe est un constructeur particulier appelé constructeur par copie.

Ce constructeur est appelé lors de la déclaration d'un objet avec recopie d'un autre objet (passé en argument au constructeur). Il s'agit là d'un appel explicite. Ce constructeur est aussi appelé implicitement lorsque l'on passe un objet en argument à une fonction et qu'il y a recopie de l'objet sur la pile ou encore lorsqu'une fonction retourne un objet (et si là encore, il y a recopie sur la pile).

La version par défaut de ce constructeur procède membre à membre : si une donnée membre est elle-même un objet, elle est recopiée par l'exécution du constructeur par copie de sa classe, sinon, il s'agit d'une copie de bas niveau, octet par octet.

Voici quelques exemples :

```
1  #include <iostream.h>
2
3  class C
4  {
5  public :
6    C(char * _s = "")  : s(_s) { cerr << "+++ C(\"" << s << "\")" << endl ; }
7    C(const C & _autre)  : s(_autre.s) { cerr << "+++ Constructeur par copie C(\"" << s
         << "\")" << endl ; }
8  private :
9    char * s ;
10 } ;
11
12 void copie_argument (C c)
13 {
14   cerr << " Dans copie_argument" << endl ;
15 }
16
17 C retourne_copie (const C &c)
18 {
19   cerr << " Dans retourne_copie" << endl ;
20   return c ;
21 }
22
23 main()
24 {
25   C c1("c1") ;
26   C c2(c1) ;
27   C c2bis = c2 ;
28
29   cerr << " Appel de copie_argument" << endl ;
30   copie_argument(c1) ;
31
32   cerr << " Appel de retourne_copie" << endl ;
33   retourne_copie(c1) ;
```

```
34 }
```

Voici le résultat de l'exécution de ce programme :

```
1  ~/COURS/C++/latex/SOURCES> a.out
2  +++ C("c1")
3  +++ Constructeur par copie C("c1")
4  +++ Constructeur par copie C("c1")
5       Appel de copie_argument
6  +++ Constructeur par copie C("c1")
7       Dans copie_argument
8       Appel de retourne_copie
9       Dans retourne_copie
10 +++ Constructeur par copie C("c1")
11 ~/COURS/C++/latex/SOURCES>
```

La version par défaut de ce constructeur suffit si aucune donnée membre n'est un pointeur qui pointe sur une zone mémoire allouée dynamiquement. Si, par contre, au moins une donnée membre est un pointeur qui pointe sur une adresse allouée dynamiquement pendant la durée de vie de l'objet, il faut définir explicitement ce constructeur de façon à ce qu'il alloue et recopie correctement ses pointeurs (voir le chapitre 3.6.2).

3.2.2 Le destructeur

Le destructeur est une fonction membre qui porte le même nom que la classe précédée du caractère ~. Cette fonction membre est unique dans la classe et ne comporte aucun paramètre. Pour un destructeur, on ne précise pas le type de retour. Dans son code on ne peut pas utiliser de $return$. Classiquement, le destructeur permet la libération de la mémoire dynamique allouée pendant la vie d'un objet. Le destructeur est appelé avant la libération de la mémoire occupée par l'objet soit :

- pour un objet automatique, juste avant la fermeture du bloc dans lequel il est créé,
- pour un objet statique ou global, juste avant la fin de programme.
- pour un tableau d'objets automatique, le destructeur est appelé pour chacun des objets du tableau juste avant la fermeture du bloc dans lequel il est créé,
- pour un tableau d'objets statique ou global, le destructeur est appelé pour chacun des objets du tableau juste avant la fin de programme,
- pour un objet alloué dynamiquement, le destructeur est appelé avant la libération du pointeur par *delete*,
- pour un tableau d'objets alloué dynamiquement, le destructeur est appelé pour chacun des objets du tableau avant la libération du pointeur par *delete* **si on n'omet pas d'indiquer le** [] **entre le *delete***

et le nom du pointeur (sinon, le destructeur est appelé seulement pour le premier objet du tableau).

Voici quelques exemples :

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

class C
{
public :
  C(const char * _s = "")  : s(strdup(_s)) { cerr << "+++ C(\"" << s << "\")" << endl ;
      }
  ~C() { cerr << "--- ~C()[s==\"" << s << "\"]" << endl ; free (s) ; }
private :
  char * s ;
} ;

C GlobC("GlobC") ;
main()
{
  cerr << "DEBUT DE BLOC (main)" << endl ;
  C c1("c1") ;
  C ctab1[2] ;
  C * cptr1 = new C("cptr1") ;
  C * cptr2 = new C[2] ;
  C * cptr3 = new C[2] ;

  delete cptr1 ; // ☐excution de ~C() pour *cptr1
  delete cptr2 ; // ☐excution de ~C() pour cptr2[0]
  delete [] cptr3 ; // 2 ☐excutions de ~C() pour cptr2[0] et cptr[1]

  for (int no = 0 ; no < 2 ; no++) {
    cerr << "DEBUT DE BLOC" << endl ;
    C c3("c3") ;
    C ctab2[2] ;
    cerr << "FIN DE BLOC" << endl ;
  } // ☐excution de ~C() pour c3, ctab2[0] et ctab2[1]

  cerr << "FIN DE BLOC (main)" << endl ;
} // ☐excution de ~C() pour c1, 2 fois ~C() pour ctab1
```

Voici le résultat de l'exécution du programme donné dans l'exemple ci-dessus. Chaque ligne du résultat est commentée. Ces commentaires ont été rajouté a postériori.

```
~/COURS/C++/latex/SOURCES> a.out
+++ C("GlobC") # ☐cration de la globale GlobC
DEBUT DE BLOC (main) # ouverture du bloc principal
+++ C("c1") # ☐cration de c1
+++ C("") # ☐cration de ctab1[0]
```

```
+++ C("") # □cration de ctab1[1]
+++ C("cptr1") # □cration de *cptr1 (□appel par le new )
+++ C("") # □cration de cptr2[0] (□appel par le new)
+++ C("") # □cration de cptr2[1] (□appel par le new)
+++ C("") # □cration de cptr3[0] (□appel par le new)
+++ C("") # □cration de cptr3[1] (□appel par le new)
--- ~C()[s=="cptr1"] # destruction de *cptr1 (□appel par le delete)
--- ~C()[s==""] # destruction de cptr2[0] (□appel par le delete)
--- ~C()[s==""] # destruction de cptr3[1] (□appel par le delete [])
--- ~C()[s==""] # destruction de cptr3[0] (□appel par le delete [])
DEBUT DE BLOC # □dbut de bloc boucle for (no == 0)
+++ C("c3") # □cration de c3
+++ C("") # □cration de ctab2[0]
+++ C("") # □cration de ctab2[1]
FIN DE BLOC # fin de bloc boucle for (no == 0)
--- ~C()[s==""] # destruction de ctab2[1]
--- ~C()[s==""] # destruction de ctab2[0]
--- ~C()[s=="c3"] # destruction de c3
DEBUT DE BLOC # □dbut de bloc boucle for (no == 1)
+++ C("c3") # □cration de c3
+++ C("") # □cration de ctab2[0]
+++ C("") # □cration de ctab2[1]
FIN DE BLOC # fin de bloc boucle for (no == 1)
--- ~C()[s==""] # destruction de ctab2[1]
--- ~C()[s==""] # destruction de ctab2[0]
--- ~C()[s=="c3"] # destruction de c3
FIN DE BLOC (main) # fin du bloc principal
--- ~C()[s==""] # destruction de ctab1[1]
--- ~C()[s==""] # destruction de ctab1[0]
--- ~C()[s=="c1"] # destruction de c1
--- ~C()[s=="GlobC"] # destruction de la globale GlobC
~/COURS/C++/latex/SOURCES>
```

En remarque générale sur ces résultats, on peut ajouter, qu'en C++, une grande quantité de code est susceptible d'être exécuté sans que, pour autant, le programmeur en soit vraimment conscient. Comme le montre ces résultats, cet état de fait est en partie dû à l'exécution du constructeur et du destructeur. Il convient donc de faire très attention à la déclaration des variables locales et surtout des tableaux. La création et la destruction de ces derniers peuvent être relativement coûteux.

3.3 Les membres *static*

Jusqu'à présent, toutes les données et fonctions membres considérées etaient relatives à un objet. On parle *de membres d'instance*. L'exécution d'une fonction membre correspond à l'exécution de l'envoi d'un message à un objet. Cette exécution peut manipuler les variables de l'instance qui reçoit le message.

Dans certains cas, on a besoin de manipuler des données communes à tous les objets d'une classe. On parle *de membres de classe*. Par exemple pour des valeurs par défaut ou encore des constantes. On peut régler ce besoin en déclarant des variables globales. Or ce n'est pas très statisfaisant car, surtout du point de vue de l'intelligibilité et de la maintenabilité du code, on ne domine pas bien la portée des variables globales. A quoi ces variables ce rapportent ? De plus, on se retrouve avec des variables et des fonctions comme en C. C'est dommage de ne pas pouvoir rester «objet».

La solution «objet» avec C++ pour manipuler des membres de classe consiste à utiliser des membres *static*. Les données et fonctions membres *static* sont communes à tous les objets de la classe. Simplement, ce sont des variables globales ou des fonctions accessibles en utilisant le nom de la classe et l'opérateur de portée.

3.3.1 Les variables membres *static*

3.3.1.1 Déclaration

Une variable membre déclarée *static* est visible pour tous les objets de la classe. Elle est initialisée une seule fois au lancement du programme à la différence des variables membres d'instance qui sont initialisées à chaque création d'objet par le constructeur.

Une variable membre *static* déclarée dans la partie *public* de la classe est visible comme une globale sans aucune restriction.

Une variable membre *static* déclarée dans la partie *private* de la classe n'est visible que par les fonctions membres de la classe.

Voici un exemple qui montre ces deux possibilités :

```
1
2  // Déclaration des variables membres static
3  class MaClasse
4  {
5  public :
6     static size_t DefaultSize ;
7  private :
8     static char * Path ;
9
10 public :
11    // Utilisation d'une variable static par une fonction membre
12    const char * defaultPath() const { return Path ; }
13 } ;
14
```

```
15 main() {
16   cout << "Taille par defaut " << MaClasse ::DefaultSize << endl; // OK, DefaultSize
         est public
17   cout << "Chemin par defaut " << MaClasse ::Path << endl; // Compilation impossible
         , C ::Path est prive
18
19   MaClasse mc;
20   cout << "Chemin par defaut " << mc.defaultPath() << endl; // OK, DefaultPath est
         public
21 }
```

3.3.1.2 Initialisation

Quelque-soit sa portée *public* ou *private*, une variable membre *static* est initialisée une seule fois lors du chargement du programme. Typiquement, pour un fichier entête «MaClasse.h», dans lequel on trouve la déclaration de l'interface de la classe *MaClasse*, et on a un fichier «MaClasse.cpp» dans lequel on trouve la mise en œuvre de la classe *MaClasse*. L'initialisation des variables *static* est placée dans ce dernier.

Voici le fichier «MaClasse.h» qui contient l'interface de la classe *MaClasse* :

```
1
2 // ☐Dclaration des variables membres static
3 class MaClasse
4 {
5 public :
6   static size_t DefaultSize;
7 private :
8   static char * Path;
9
10 public :
11   // Utilisation d'une variable static par une fonction membre
12   const char * defaultPath() const { return Path; }
13 };
```

Et voici le fichier «MaClasse.cpp» qui contient l'initialisation des variables *static* :

```
1 #include <string.h>
2 #include "MaClasse.h"
3
```

```
// Initialisation des variables membres static
size_t MaClasse ::DefaultSize = 256 ;
char * MaClasse ::Path = strdup(”/tmp”) ;
```

3.3.2 Les fonctions membres *static*

Une fonction membre *static* est relative à la classe. Une telle fonction est complètement indépendante de tout objet : la variable prédéfinie *this* n'existe pas, et par conséquent, une fonction membre *static* ne peut pas manipuler de variables d'instances et ne peut pas être déclarée constante.

Une fonction membre *static* déclarée dans la partie *public* de la classe est exécutable comme une fonction normale sans aucune restriction.

Une fonction membre *static* déclarée dans la partie *private* de la classe n'est exécutable que par les fonctions membres de la classe (*static* ou non *static*).

Classiquement, on déclare des variables *static* dans la partie *private* de la classe et des fonctions *static*, *private* ou *public* pour manipuler ces variables. Voici l'exemple précédent remanié pour coller à ce cadre :

```

// □Dclaration des variables membres static
class MaClasse
{
private :
   static size_t DefaultSize ;
   static char * Path ;
 public :
   static size_t GetDefaultSize() ;
   static const char * GetDefaultPath() ;
   static void SetDefaultSize(size_t) ;
   static void SetDefaultPath(const char *) ;

   // et le reste ...
} ;
```

```
#include <string.h>
#include <stdio.h>
#include ”MaClasse.h”

```

```
size_t MaClasse ::DefaultSize = 256 ;
char * MaClasse ::Path = strdup(”/tmp”) ;

size_t MaClasse ::
GetDefaultSize()
{
  return DefaultSize ;
}

const char * MaClasse ::
GetDefaultPath()
{
  return Path ;
}

void MaClasse ::
SetDefaultSize(size_t s)
{
  DefaultSize = s ;
}

void MaClasse ::
SetDefaultPath(const char * p)
{
  free(Path) ;
  Path = strdup(p) ;
}

// et le reste ...
```

```
#include ”MaClasse.h”

main()
{
  MaClasse ::SetDefaultSize(1024) ;
  MaClasse ::SetDefaultPath(”/usr/local/tmp”) ;
  cout << ”Valeurs par defaut  : ”
       << MaClasse ::GetDefaultSize()
       << ” ”
       << MaClasse ::GetDefaultPath()
       << endl ;
}
```

On observe qu’il n’y syntaxiquement aucune différence entre la mise en œuvre d’une fonction membre d’instance et une fonction membre *static*.

3.3.3 Un exemple classique

Le comptage des instances d'une classes en utilisant une variable *static* est une utilisation classique. On a une variable *static* NbInstances qui contient le nombre d'instances et deux fonction membres static pour incrémenter et décrémenter cette variable :

```
1  #include <iostream.h>
2
3  class ACompter
4  {
5  private :
6    static size_t NbInstances ;
7    static void IncrNbInstances() ;
8    static void DecrNbInstances() ;
9
10 public :
11   static size_t GetNbInstances() ;
12   // Dans le constructeur, on □incrmente le nombre d'instances
13   ACompter() {
14     IncrNbInstances() ;
15   }
16   // et dans le destructeur, on □□dcrmente le nombre d'instances
17   ~ACompter() {
18     DecrNbInstances() ;
19   }
20 } ;
21
22 size_t ACompter ::NbInstances = 0 ;
23
24 void ACompter ::
25 IncrNbInstances()
26 {
27   NbInstances++ ;
28 }
29
30 void ACompter ::
31 DecrNbInstances()
32 {
33   NbInstances-- ;
34 }
35
36 size_t ACompter ::
37 GetNbInstances()
38 {
39   return NbInstances ;
40 }
41
42 main()
43 {
44   ACompter ac1 ;
45   cout << ACompter ::GetNbInstances() << endl ; // affiche 1
46
```

```
47    ACompter * tab = new ACompter[10]; // 10 ☐excutions du constructeur
48    cout << ACompter ::GetNbInstances() << endl; // affiche 11
49
50    if (1) {
51       ACompter tab2[10];
52       cout << ACompter ::GetNbInstances() << endl; // affiche 21
53    } // ici, ☐excution du destructeur pour les 10 objets de tab2
54
55    cout << ACompter ::GetNbInstances() << endl; // affiche 11
56
57    delete [] tab; // avec le [] on a destruction de chaque objet du tableau ☐point
58    cout << ACompter ::GetNbInstances() << endl; // affiche 1
59 }
```

Voici la trace d'exécution :

```
1  > ./a.out
2  1
3  11
4  21
5  11
6  1
7  >
```

3.4 La composition entre objets

Lorsque le type d'une variable membre d'une classe et lui même une classe, on parle de composition d'objets.

3.4.1 Le constructeur et la composition

Prenons le cas de la classe C qui comprend une variable membre d dont le type est la classe D. Pour la spécification et l'exécution du constructeur de C, les deux cas suivants sont à prendre en considération :

- si rien n'est indiqué dans le constructeur de C pour le choix du constructeur de d, alors, le constructeur sans argument de la classe D est exécuté pour la création de la variable membre d ; Cette exécution intervient avant l'exécution du code du constructeur de C ; en conséquence, dans ce cas, un constructeur sans argument doit exister dans la classe D ;
- il est possible d'indiquer quel constructeur exécuter pour initialiser d dans le constructeur de C ; cet appel de constructeur doit être obligatoirement inséré entre la parenthèse qui ferme la déclaration des arguments et l'accolade ouvrante du bloc du constructeur de C.

Voici un exemple :

```
class D
{
public :
  D()  : i(0) {}
  D(int _i)  : i(_i) {}
  ~D() { }
public :
  int i;
};

// une classe composée
class C
{
public :
  C()  : f(0.0) {} // creation de d avec D()
  C(int _i)  : d(_i) {} // creation de d avec D(int)
  C(float _f)  : f(_f) {} // creation de d avec D()
  ~C() {}
private :
  D d;
  float f;
};

main()
{
  C c; // exécution de D() et ensuite de C()
  C c(3); // exécution de D(3) et ensuite de C(3)
  C c(1.5); // exécution de D(), initialisation de c.f avec 1.5 et ensuite exécution de C
      (1.5)
}
```

En rappel : il est obligatoire d'indiquer un constructeur pour un objet composant qui est une référence. Il peut aussi être obligatoire d'indiquer un constructeur pour un objet composant constant (voir chapitre 3.2.1.2).

D'autre part, il peut être très inefficace de na pas indiquer de constructeur pour une variable membre. En effet, comme le montre l'exemple suivant, cela peut aboutir à une double initialisation d'une variable membre.

```
class D
{
public :
  D()  : i(0) {}
  D(int _i)  : i(_i) {}
  ~D() { }
  void set_i (int _i) { i = i; }
public :
  int i;
};
```

```
11
12  #if 0
13  // Version 1 de la classe C
14  class C
15  {
16  public :
17    C()  : {} // creation de d avec D()
18    C(int _i) // creation de d avec D(int)
19    {
20      d.set_i(_i) ; // modification de d avec _i
21    } // il y a eut double initialisation de d
22    ~C() {}
23  private :
24    D d ;
25  } ;
26  #endif
27
28  // Version 2 de la classe C
29  class C
30  {
31  public :
32    C()  : {} // creation de d avec D()
33    C(int _i)  : d(_i) // creation de d avec D(int)
34    {}
35    ~C() {}
36  private :
37    D d ;
38  } ;
```

Dans cet exemple, les deux versions de la classes C sont valides. Par contre, dans la première version, le constructeur $C(int)$ est moins efficace puisque que la variable d est d'abord initialisée avec le constructeur $D()$ et ensuite dans le corps du constructeur $C(int)$. Dans la deuxième version, la variable membre d n'est initialisée qu'une seule fois.

3.4.2 Le destructeur et la composition

Le destructeur des objets composants est exécuté après celui de l'objet composé. Pour le cas du chapitre précédent (classe C contient une variable membre d dont la classe est D), lors de la mort d'un objet de la classe C, le destructeur $\sim C()$ est exécuté avant $\sim D()$. Pour l'exemple du chapitre précédent, cela donne :

```
1  main()
2  {
3    C c1 ;
4    C c2(3) ;
5    C c3(1.5) ;
6  }
```

```
// ☐excution de ~C() pour c3
// ☐excution de ~D() pour c3.d
// ☐excution de ~C() pour c2
// ☐excution de ~D() pour c2.d
// ☐excution de ~C() pour c1
// ☐excution de ~D() pour c1.d
```

3.5 Affectation entre objets

Comme en C dans le cas d'une structure, l'affectation entre objets est autorisée. Il y a alors recopie directe, octet par octet, de la valeur des données membres qu'elles soient publiques ou privées.

Mais attention :

- si une donnée membre est un pointeur, il y a recopie de la valeur du pointeur et non allocation et recopie de la zone mémoire pointée ; la copie d'objet par défaut est donc une copie légère ; pour recopier correctement les objets, il faudra alors redéfinir l'opérateur d'affectation mais aussi le constructeur par copie (voir le chapitre 3.6.2),
- comme il est montré dans le chapitre 3.2.1.1, il peut y avoir ambiguïté entre une affectation et un appel de constructeur ; l'initialisation par affectation d'un objet lors de sa déclaration n'exécute pas une affectation mais un constructeur.

3.6 Le cas des variables membres pointeurs

3.6.1 La règle de base

Il est très important de considérer le cas des variables membres pointeurs avec beaucoup d'attention, particulièrement si ces pointeurs sont alloués et libérés pendant la durée de vie d'un objet. C'est en effet souvent une source de problème importante.

Comme règle de base, dans un tel cas, le rôle des constructeurs est d'allouer la mémoire dynamique pour initialiser le(s) pointeur(s) puis d'initialiser la mémoire allouée (la mémoire pointée par les pointeurs). Le rôle du destructeur est de libérer la mémoire allouée. Il est très important, autant que possible de conserver cette symétrie. Voici un exemple :

```
class Tel
{
  public :
  Tel()  : nom(new char), tel(new char)
```

```
  {
    *nom = '\0';
    *tel = '\0';
  }
  Tel(const char * _nom, const char * _tel)
     : nom(new char[strlen(_nom)+1]), tel(new char[strlen(_tel)+1])
  {
    strcpy(nom, _nom);
    strcpy(tel, _tel);
  }
  ~Tel()
  {
    delete nom;
    delete prenom;
  }

private :
  char * nom;
  char * tel;
};
```

Dans cet exemple, on observe que les variables membres de Tel sont allouées dynamiquement et la zone mémoire allouée est initialisée même dans le constructeur sans argument. De cette façon, le destructeur peut systématiquement désallouer les pointeurs.

Pour le constructeur sans argument, une autre solution consiste à initialiser les pointeurs à 0. En effet, normalement, l'opérateur $delete$ ignore les pointeurs nuls. Voici une deuxième version de la classe Tel.

```
class Tel
{
  public :
  Tel()  : nom(0), tel(0) { }
  Tel(const char * _nom, const char * _tel)
     : nom(new char[strlen(_nom)+1]), tel(new char[strlen(_tel)+1])
  {
    strcpy(nom, _nom);
    strcpy(tel, _tel);
  }
  ~Tel()
  {
    delete nom;
    delete prenom;
  }

private :
  char * nom;
  char * tel;
};
```

3.6.2 Le problème de l'affectation et de la construction par copie

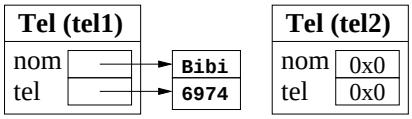
3.6.2.1 L'affectation

Voici la classe *Tel*. Cette classe comprend deux variables membres pointeurs alloués dans un constructeur et libérés par le destructeur. Il s'agit donc d'une classe avec pointeurs alloués pendant la durée de vie d'un objet. Cette première version utilise l'opérateur d'affectation par défaut qui copie octet par octet les pointeurs membres.

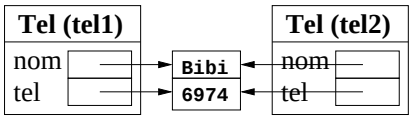
```
1  #ifndef __TEL_H__
2  #define __TEL_H__
3  #include <string.h>
4  #include <stdlib.h>
5  class Tel {
6  public :
7    Tel ()  : nom(0), tel(0) {}
8    Tel (const char * _nom, const char * _tel)
9        : nom(new char [strlen(_nom)+1]),
10         tel(new char [strlen(_tel)+1])
11   {
12     strcpy(nom, _nom) ;
13     strcpy(tel, _tel) ;
14   }
15   ~Tel()
16   {
17     delete nom ;
18     delete tel ;
19   }
20   const char * get_nom() const { return nom ; }
21   const char * get_tel() const { return tel ; }
22 private :
23   char * nom ;
24   char * tel ;
25 } ;
26 #endif // __TEL_H__
```

Voici à gauche un exemple de programme avec affectation entre deux variables de type *Tel* et à droite un schéma qui illustre ce qui se passe en mémoire.

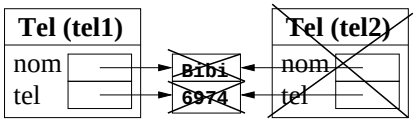
```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2;
```




```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2;
    tel2 = tel1; // Affectation
```




```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2;
    tel2 = tel1;
  } // Fin de bloc : tel2.~Tel()
 cerr << tel1.get_nom() << endl; // Boum!!!
}
```



On observe que la copie brutale des pointeurs membres a pour conséquence fâcheuse le partage des adresses par les deux objets $tel1$ et $tel2$ après la copie. Lors de la mort de $tel2$, le destructeur désalloue les pointeurs membres. En conséquence, les adresses stockées dans $tel1$ deviennent invalides (car libérées par l'exécution du destructeur *tel2.~Tel()*) en fin de bloc. $tel1$ est alors invalide.

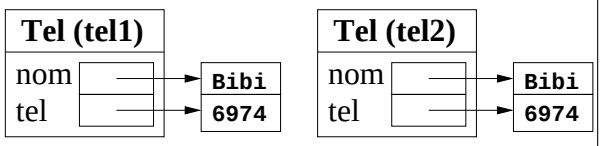
La solution consiste à définir l'opérateur d'affectation de la classe de façon à ce que les pointeurs soient correctement dupliqués. La copie ne provoque alors plus de partage d'adresses. Chaque objet gère ses propres allocations et libérations de mémoire sans incidence sur l'état d'autres objets.

Voici notre exemple, avec la définition de l'opérateur d'affectation (la syntaxe de la définition des opérateurs est abordée dans le chapitre 5).

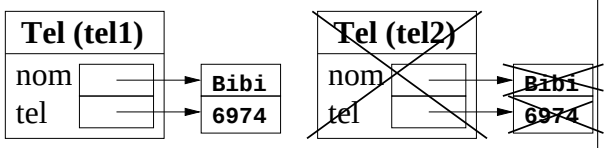
```
1 #ifndef __TEL_H__
2 #define __TEL_H__
3
4 #include <string.h>
5 #include <stdlib.h>
6 class Tel {
7 public :
8   Tel ()  : nom(0), tel(0) {}
```

```
9     Tel (const char * _nom, const char * _tel)
10        : nom(new char [strlen(_nom)+1]),
11          tel(new char [strlen(_tel)+1])
12    {
13      strcpy(nom, _nom);
14      strcpy(tel, _tel);
15    }
16    ~Tel()
17    {
18      delete nom;
19      delete tel;
20    }
21    const char * get_nom() const { return nom; }
22    const char * get_tel() const { return tel; }
23
24    // □dfinition de l'□oprateur =
25    Tel & operator = (const Tel & _autre)
26    {
27      if (this  != &_autre) { // si on ne copie pas soit meme
28        // liberation des pointeurs
29        delete nom;
30        delete tel;
31        // allocation
32        nom = new char [strlen(_autre.nom)+1];
33        tel = new char [strlen(_autre.tel)+1];
34        // copie
35        strcpy(nom, _autre.nom);
36        strcpy(tel, _autre.tel);
37      }
38      return *this;
39    }
40  private :
41    char * nom;
42    char * tel;
43  };
44  #endif // __TEL_H__
```

```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2;
    tel2 = tel1; // Affectation
```



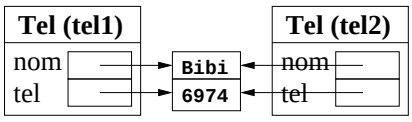

```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2;
    tel2 = tel1;
  } // Fin de bloc : tel2.~Tel()
 cerr << tel1.get_nom() << endl; // OK !
}
```



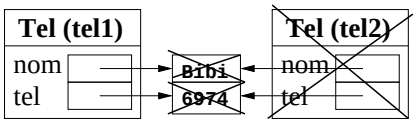
3.6.2.2 La construction par copie

Dans le chapitre 3.2.1.4, nous avons vu que la version par défaut de ce constructeur procède membre à membre : si une donnée membre est elle-même un objet, elle est recopiée par l'exécution du constructeur par copie de sa classe, sinon, il s'agit d'une copie octet par octet. Le problème posé pour le cas des variables membres pointeurs est similaire au problème posé par l'affectation entre objets.

```
main()
{
  Tel tel1("Bibi", "6974");
  {
   // construction de tel2 par copie de tel1
    Tel tel2(tel1);
```




```
main()
{
  Tel tel1("Bibi", "6974");
  {
    Tel tel2(tel1);
  } // Fin de bloc : tel2.~Tel()
 cerr << tel1.get_nom() << endl; // Boum!!!
}
```



La solution consiste à redéfinir le constructeur par copie :

```
1  #ifndef __TEL_H__
2  #define __TEL_H__
3
4  #include <string.h>
5  #include <stdlib.h>
```

```
6  class Tel {
7  public :
8    Tel ()  : nom(0), tel(0) {}
9    Tel (const char * _nom, const char * _tel)
10       : nom(new char [strlen(_nom)+1]),
11         tel(new char [strlen(_tel)+1])
12   {
13     strcpy(nom, _nom) ;
14     strcpy(tel, _tel) ;
15   }
16   // constructeur par copie
17   Tel (const Tel & _autre)
18       : nom(new char [strlen(_autre.nom)+1]),
19         tel(new char [strlen(_autre.tel)+1]),
20   {
21     strcpy(nom, _autre.nom) ;
22     strcpy(tel, _autre.tel) ;
23   }
24   ~Tel()
25   {
26     delete nom ;
27     delete tel ;
28   }
29   const char * get_nom() const { return nom ; }
30   const char * get_tel() const { return tel ; }
31
32   // □dfinition de l'□oprateur =
33   Tel & operator = (const Tel & _autre)
34   {
35     if (this  != &_autre) { // si on ne copie pas soit meme
36       // liberation des pointeurs
37       delete nom ;
38       delete tel ;
39       // allocation
40       nom = new char [strlen(_autre.nom)+1] ;
41       tel = new char [strlen(_autre.tel)+1] ;
42       // copie
43       strcpy(nom, _autre.nom) ;
44       strcpy(tel, _autre.tel) ;
45     }
46     return *this ;
47   }
48 private :
49   char * nom ;
50   char * tel ;
51 } ;
52 #endif // __TEL_H__
```

3.6.2.3 Constructeur par copie versus opérateur d'affectation

Dans ces deux fonctions, on trouve du code pour recopier les données membres d'un objets vers un autre et du code pour détruire les données

membres. De plus, le code pour détruire se retrouve à nouveau dans le destructeur. Pour éviter de réécrire complètement le code similaire, deux fonctions privées, une pour la copie et l'autre pour la destruction peuvent être mise en oeuvre. En bénéfice, le code des classes est plus claire et moins volumineux.

Voici l'exemple de notre classe Tel avec ces deux fonctions *copy* et *destroy* :

```
#ifndef __TEL_H__
#define __TEL_H__

#include <string.h>
#include <stdlib.h>
class Tel {
public :
  Tel ()  : nom(0), tel(0) {}
  Tel (const char * _nom, const char * _tel)
     : nom(new char [strlen(_nom)+1]),
       tel(new char [strlen(_tel)+1])
  {
    strcpy(nom, _nom) ;
    strcpy(tel, _tel) ;
  }
  // constructeur par copie
  Tel (const Tel & _autre)
  {
    copy(_autre) ;
  }
  ~Tel()
  {
    destroy()
  }
  const char * get_nom() const { return nom ; }
  const char * get_tel() const { return tel ; }

  // □dfinition de l'□oprateur =
  Tel & operator = (const Tel & _autre)
  {
    if (this  != &_autre) {
      destroy() ;
      copy(_autre) ;
    }
    return *this ;
  }

private :
  void copy(const Tel & _autre)
  {
    nom = new char [strlen(_autre.nom)+1] ;
    tel = new char [strlen(_autre.tel)+1] ;
    strcpy(nom, _autre.nom) ;
    strcpy(tel, _autre.tel) ;
  }
```

```
46    void destroy()
47    {
48      delete nom ;
49      delete tel ;
50    }
51
52    char * nom ;
53    char * tel ;
54  } ;
55  #endif // __TEL_H__
```

3.6.2.4 En conclusion

Le problème de l'affectation et de la construction par copie lorsqu'une ou plusieurs variables membres sont des pointeurs est une source de bogue très fréquente qu'il s'agit de traiter avec soin. La règle d'or est de systématiquement redéfinir le constructeur par copie et l'opérateur d'affectation si le ou les pointeurs sont alloués dynamiquement pendant la durée de vie de l'objet et libérés par le destructeur.

Chapitre 4

Chaînes de caractères et collections standards

On peut maintenant présenter quelques classes pour la gestion des chaînes de caractères et des collections. Elles sont pratiques ici pour les exemples et facilitent grandement le travail de l'étudiant en TP.

Les classes présentées sont celles de la *STL* pour *Standard Template Library* (voir http://www.sgi.com/tech/stl/). La *STL* se constitue d'un ensemble de classes dont l'objectif est de standardiser la gestion des collections en C++. On y trouve les tableaux, les listes chainées, les tables de hash, etc. On ne sait pas encore ce qu'est un *template* mais c'est pas fondamental ici. Le lecteur est invité à accepter la syntaxe sans tout comprendre. Les *templates* sont présentés dans le chapitre **??**.

4.1 La classe *string*

La classe *string* permet la manipulation de chaînes de caractères sans que le programmeur ait a se préoccuper de l'allocation dynamique comme en C. Une chaîne est un objet qui peut s'initialiser à partie d'une chaîne constante, se copier dasn une autre. On peut toujours accéder à un caractère

particulier par sa position dans la chaîne. On peut aussi manipuler des sous chaînes. Cette classe est décrite dans la documentation en ligne : http://www.sgi.com/tech/stl/basic_string.html.

4.1.1 Manipulation simple d'un *string*

Voici un premier exemple : la déclaration et l'initialisation de deux chaînes.

```
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string

main()
{
  string maPremiereChaine = "ouf, plus de char*";
  string maDeuxiemeChaine;
  maDeuxiemeChaine = "on respire";
  cout << maPremiereChaine << " " << maDeuxiemeChaine << endl;
}
```

On remarque que *maPremiereChaine* est initialisée directement lors de sa déclaration avec une chaîne C constante (const char *). C'est donc un constructeur de *string* qui est utilisé. *maDeuxiemeChaine* est déclarée simplement sans être initialisée. C'est le constructeur sans argument qui est donc exécuté. La chaîne est vide. On lui affecte ensuite une chaîne C constante.

4.1.2 Manipulation des caractères d'une chaîne

Voici un exemple qui montre qu'on manipule très naturellemnt les caractères d'une chaîne :

```
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string

main()
{
  string maPremiereChaine = "ouf, plus de char*";

  // affichage caractere par caractere
  for (size_t n = 0; n < maPremiereChaine.size(); n++) {
    cout << maPremiereChaine[n];
  }
  cout << endl;
  // on modifie le premier caractere
  maPremiereChaine[0] = 'O';
  cout << maPremiereChaine << endl;
```

```
}
```

4.1.3 Modification d'une chaîne

On peut ajouter des caractères ou des chaînes en fin, au debut, au milieu :

```
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string

main()
{
  string maPremiereChaine = "ouf, plus de char*";

  // Ajouter un caractere
  maPremiereChaine += '.';

  // Ajouter une chaine C constante
  maPremiereChaine.append(" On respire");
  // ou encore
  maPremiereChaine += ".";
  // ou encore
  maPremiereChaine = maPremiereChaine + " C'est presque aussi bien qu'en Java.";

  // Inserer des chaines C n'importe ou
  // par exemple en □dbut de chaine
  maPremiereChaine.insert(0, "Youpii !!!, ");
  // ou encore
  maPremiereChaine = string("Youpii") + maPremiereChaine;
  // ou encore apres "Youpii"
  maPremiereChaine.insert(strlen("Youpii"), "!!! ");

  cout << maPremiereChaine << endl;
  // on affiche  : Youpii !!! Youpii !!!, ouf, plus de char*. On respire. C'est presque aussi
      bien qu'en Java.
}
```

4.1.4 Récupération de la chaîne C

On peut accéder en lecture à la représentation en const char * d'un string. C'est très utile notament pour le couplage C/C++ .

```
#include <stdio.h> // pour les E/S standards du C
#include <iostream.h> // pour les E/S standards du C++
#include <string> // definition de la classe string
```

```
4
5  main()
6  {
7    string maPremiereChaine = "ouf, plus de char*";
8    const char * la_chaine_C = maPremiereChaine.c_str();
9    char s[128];
10   strcpy(s, maPremiereChaine.c_str());
11
12   printf("%s - %s\n", la_chaine_C, s);
13   // affichage de  : ouf, plus de char* - ouf, plus de char*
14
15   // On peut aussi toujours ecrire  :
16   cout << la_chaine_C << " - " << s << endl;
17   // et obtenir la meme chose sur la sortie standard
18 }
```

4.2 Les *template* collections

On dispose de *template* pour la gestion des tableaux, des listes, des ensembles et des tables de hash. On peut ainsi manipuler des collections d'entiers, de caractères, d'objets ou de pointeurs sur des objets. On ne se soucie pas de la gestion de la mémoire pour le stockage des éléments dans les collections : c'est géré par les classes.

Dans la suite est présenté un sous-ensemble minimal de ce qui est disponible dans la *STL* : la gestion des tableaux avec le *template vector*, la gestion des listes avec le *template list*, et la gestion des tables de hash avec le *template hash_map*.

4.2.1 Le *template vector*

Il est très utile pour gérer des tableaux de n'importe quoi. Ce *template* est décrit dans la documentation en ligne : http://www.sgi.com/tech/stl/Vector.html.

4.2.1.1 Déclaration d'un *vector*

Là, on est obligé d'utiliser une syntaxe bizaroïde sans la comprendre : on indique toujours le type des éléments stockées dans le tableau entre $<$ et $>$ après *vector*.

```
1  #include <iostream.h> // pour les E/S standards
2  #include <string> // definition de la classe string
3  #include <vector> // definition du template vector
4
5  class Personne
```

```
{
public :
  Personne() {} // constructeur sans argument obligatoire pour declarer un vector
protected :
  string nom, prenom ;
} ;

main()
{
  // un tableau d'entiers
  vector<int> tabofint ;
  // un tableau de caracteres
  vector<char> tabofchar ;
  // un tableau de string
  vector<string> tabofstring ;
  // un tableau de Personnes
  vector<Personne> tabofpersonne ;
  // un tableau de pointeurs sur des Personnes
  vector<Personne*> tabofpersonneptr ;
}
```

4.2.1.2 Ajout et récupération d'éléments

Pour ajouter un élément, c'est la fonction *push_back* qui est utilisée : l'élément passé en argument est ajouté à la fin du tableau. Pour récupérer un élément, c'est classique comme pour un tableau C.

```
#include <stdio.h> // pour le C
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string
#include <vector> // definition du template vector

class Personne
{
  friend ostream & operator << (ostream & out, const Personne & p) ;

public :
  Personne() {} // constructeur sans argument obligatoire pour declarer un vector
  Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
protected :
  string nom_, prenom_ ;
} ;

ostream & operator << (ostream & out, const Personne & p)
{
  out << "personne(" << p.nom_ << " " << p.prenom_ << ")" ;
}

main()
{
```

```
24    // un tableau d'entiers
25    vector<int> tabofint;
26
27    // Ajout d'entiers
28    for (size_t n = 0; n < 4; n++) {
29      tabofint.push_back(n);
30    }
31    // recuperation des entiers
32    size_t cum = 0;
33    for (size_t n = 0; n < tabofint.size(); n++) {
34      cum += tabofint[n];
35    }
36    cerr << cum << endl;
37
38    // un tableau de string
39    vector<string> tabofstring;
40
41    // Ajout de chaines
42    for (size_t n = 0; n < 4; n++) {
43      char buf[512];
44      sprintf(buf, "%d", n);
45      tabofstring.push_back(string("Chaine #") + buf);
46    }
47
48    // recuperation des chaines
49    for (size_t n = 0; n < tabofstring.size(); n++) {
50      cout << tabofstring[n] << endl;
51    }
52
53    // un tableau de pointeurs sur des Personnes
54    vector<Personne*> tabofpersonneptr;
55
56    // Ajout de Personne*
57    const char * lesprenoms[] = {"bibi", "toto", "momo", 0};
58    const char * lesnoms[] = {"fricotin", "laribo", "latronche", 0};
59
60    for (size_t n = 0; lesnoms[n] != 0; n++) {
61      Personne * p = new Personne(lesnoms[n], lesprenoms[n]);
62      tabofpersonneptr.push_back(p);
63    }
64
65    // affichage des personnes
66    for (size_t n = 0; n < tabofpersonneptr.size(); n++) {
67      cout << *tabofpersonneptr[n] << endl;
68    }
69
70    // on fini proprement en vidant le tableau
71    for (size_t n = 0; n < tabofpersonneptr.size(); n++) {
72      delete tabofpersonneptr[n];
73    }
74    tabofpersonneptr.clear();
75  }
```

Et voici le résultat de l'éxécution du programme :

```
~/COURS/C++/latex/SOURCES> a.out
6
Chaine #0
Chaine #1
Chaine #2
Chaine #3
personne(fricotin bibi)
personne(laribo toto)
personne(latronche momo)
~/COURS/C++/latex/SOURCES>
```

On remarque que pour un tableau de pointeurs, on doit faire bien attention à récupérer la mémoire dynamique éventuellement allouée pour les éléments. La fonction *clear* ne sert qu'à vider le tableau et non détruire (désallouer) les éléments contenus.

4.2.1.3 Insertion/suppression d'éléments et copie de tableaux

On peut insérer/supprimer un élément à une certaine position. On peut aussi directement copier un tableau dans un autre.

```
#include <stdio.h> // pour le C
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string
#include <vector> // definition du template vector

main()
{
  // un tableau de string
  vector<string> tabofstring;

  // Insertion d'une chaine n'importe ou
  tabofstring.insert(tabofstring.begin() + 0, "toto");
  tabofstring.insert(tabofstring.begin() + 0, "bibi");
  tabofstring.insert(tabofstring.begin() + 2, "momo");
  tabofstring.insert(tabofstring.begin() + 2, "tata");
  tabofstring.insert(tabofstring.begin() + 4, "papa");

  for (size_t n = 0; n < tabofstring.size(); n++) {
    cout << tabofstring[n] << endl;
  }
  cout << endl;

  // Enlever un element du tableau
  tabofstring.erase(tabofstring.begin() + 2);
  for (size_t n = 0; n < tabofstring.size(); n++) {
    cout << tabofstring[n] << endl;
  }
  cout << endl;

```

```
30    // Enlever un element du tableau par la fin
31    tabofstring.pop_back();
32    for (size_t n = 0; n < tabofstring.size(); n++) {
33       cout << tabofstring[n] << endl;
34    }
35    cout << endl;
36
37    // Copier un tableau dans un autre
38    vector<string> tabofstring2(tabofstring); // construction par copie
39    // ou encore
40    vector<string> tabofstring3;
41    tabofstring3 = tabofstring; // affectation
42    for (size_t n = 0; n < tabofstring.size(); n++) {
43       cout << tabofstring[n] << " " << tabofstring2[n] << " " << tabofstring3[n] <<
          endl;
44    }
45    cout << endl;
46 }
```

Et voici le résultat de l'éxécution du programme :

```
1  ~/COURS/C++/latex/SOURCES> a.out
2  bibi
3  toto
4  tata
5  momo
6  papa
7
8  bibi
9  toto
10 momo
11 papa
12
13 bibi
14 toto
15 momo
16
17 bibi bibi bibi
18 toto toto toto
19 momo momo momo
20
21 ~/COURS/C++/latex/SOURCES>
```

4.2.2 Le *template list*

On dispose de listes chainées bidirectionnelles avec les *template list*. Ce *template* est décrit dans la documentation en ligne : http://www.sgi.com/tech/stl/List.html.

4.2.2.1 Déclaration d'un *list*

Là encore, on utilise la syntaxe bizaroïde : on indique toujours le type des éléments stockées dans le tableau entre $<$ et $>$ après *list*.

```
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string
#include <list> // definition du template list

class Personne
{
public :
  Personne() {} // constructeur sans argument obligatoire pour declarer une list
protected :
  string nom, prenom ;
} ;

main()
{
  // une liste d'entiers
  list<int> listofint ;
  // une liste de caracteres
  list<char> listofchar ;
  // une liste de string
  list<string> listofstring ;
  // une liste de Personnes
  list<Personne> listofpersonne ;
  // une liste de pointeurs sur des Personnes
  list<Personne*> listofpersonneptr ;
}
```

4.2.2.2 Ajout et récupération d'éléments

Pour ajouter un élément, c'est encore la fonction *push_back* qui est utilisée : l'élément passé en argument est ajouté en fin de liste. Pour insérer en tête le liste, on utilise *push_front*.

Pour récupérer un élément, c'est pas classique du tout. On utilise un itérateur. Un itérateur peut être vu comme un pointeur sur les éléments stockés. Pour une liste l, on initialise un itérateur avec le résultat de la fonction *l.begin()*. Les valeurs possibles de l'itérateur vont de *l.begin()* jusqu'à *l.end() - 1*. Pour avancer dans la liste on incrémente l'itérateur avec l'opérateur $++$.

```
#include <stdio.h> // pour le C
#include <iostream.h> // pour les E/S standards
#include <string> // definition de la classe string
```

```
4  #include <list> // definition du template list
5
6  class Personne
7  {
8    friend ostream & operator << (ostream & out, const Personne & p);
9
10 public :
11   Personne() {} // constructeur sans argument obligatoire pour declarer une list
12   Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
13 protected :
14   string nom_, prenom_;
15 };
16
17 ostream & operator << (ostream & out, const Personne & p)
18 {
19   out << "personne(" << p.nom_ << " " << p.prenom_ << ")";
20 }
21
22 main()
23 {
24   // une liste d'entiers
25   list<int> listofint;
26
27   // Ajout d'entiers
28   for (size_t n = 0; n < 4; n++) {
29     listofint.push_back(n);
30   }
31   // recuperation des entiers
32   size_t cum = 0;
33   list<int> ::iterator iitor;
34   for (iitor = listofint.begin(); iitor != listofint.end(); iitor++) {
35     cum += *iitor;
36   }
37   cerr << cum << endl;
38
39   // une liste de string
40   list<string> listofstring;
41
42   // Ajout de chaines
43   for (size_t n = 0; n < 4; n++) {
44     char buf[512];
45     sprintf(buf, "%d", n);
46     listofstring.push_front(string("Chaine #") + buf); // insertion en tete de liste
47   }
48
49   // recuperation des chaines
50   list<string> ::iterator sitor;
51   for (sitor = listofstring.begin(); sitor != listofstring.end(); sitor++) {
52     cout << *sitor << endl;
53   }
54
55   // une liste de pointeurs sur des Personnes
56   list<Personne*> listofpersonneptr;
57
58   // Ajout de Personne*
```

```
59    const char * lesprenoms[] = {"bibi", "toto", "momo", 0};
60    const char * lesnoms[] = {"fricotin", "laribo", "latronche", 0};
61
62    for (size_t n = 0; lesnoms[n] != 0; n++) {
63       Personne * p = new Personne(lesnoms[n], lesprenoms[n]);
64       listofpersonneptr.push_back(p); // insertion en queue de liste
65    }
66
67    // affichage des personnes
68    list<Personne*> ::iterator pitor;
69    for (pitor = listofpersonneptr.begin(); pitor != listofpersonneptr.end(); pitor++) {
70       cout << **pitor << endl;
71    }
72
73    // on fini proprement en vidant a liste
74    while (!listofpersonneptr.empty()) {
75       delete listofpersonneptr.back();
76       listofpersonneptr.pop_back();
77    }
78 }
```

Et voici le résultat de l'éxécution du programme :

```
1  ~/COURS/C++/latex/SOURCES> a.out
2  6
3  Chaine #3
4  Chaine #2
5  Chaine #1
6  Chaine #0
7  personne(fricotin bibi)
8  personne(laribo toto)
9  personne(latronche momo)
10 ~/COURS/C++/latex/SOURCES>
```

4.2.2.3 Insertion/suppression d'éléments et copie de listes

On peut insérer/supprimer un élément à une certaine position en utilisant un itérateur. On peut aussi directement copier une liste dans une autre.

```
1  #include <stdio.h> // pour le C
2  #include <iostream.h> // pour les E/S standards
3  #include <string> // definition de la classe string
4  #include <list> // definition du template list
5
6  main()
7  {
8    // un tableau de string
9    list<string> listofstring;
10
11   // Ajout d'une chaine
```

```
12    listofstring.push_back("toto");
13    // ou encore
14    listofstring.insert(listofstring.begin()++, "toto");
15    listofstring.insert(listofstring.begin()++, "bibi");
16    listofstring.insert(listofstring.begin()++, "momo");
17    listofstring.insert(listofstring.begin()++, "tata");
18    listofstring.insert(listofstring.begin(), "papa");
19
20    // Parcour d'une liste  : on utilise un iterateur
21    for (list<string> ::iterator itor = listofstring.begin(); itor != listofstring.end(); itor++)
          {
22      cout << *itor << endl; // on □rcupere l□'□lment qui est pointe par l'iterateur
23    }
24    cout << endl;
25
26    // Enlever un element du tableau
27    for (list<string> ::iterator itor = listofstring.begin(); itor != listofstring.end(); itor++)
          {
28      string n(*itor); // on □rcupere l□'□lment qui est pointe par l'iterateur
29      if (n == string("tata")) {
30        listofstring.erase(itor); // attention, il faut sortir, itor n'est plus valide
31        break;
32      }
33    }
34
35    for (list<string> ::iterator itor = listofstring.begin(); itor != listofstring.end(); itor++)
          {
36      cout << *itor << endl;
37    }
38    cout << endl;
39
40    // Enlever un element de la liste tableau par la fin
41    listofstring.pop_back();
42    for (list<string> ::iterator itor = listofstring.begin(); itor != listofstring.end(); itor++)
          {
43      cout << *itor << endl;
44    }
45    cout << endl;
46
47    // Copier une liste dans une autre
48    list<string> listofstring2(listofstring); // construction par copie
49    // ou encore
50    list<string> listofstring3;
51    listofstring3 = listofstring; // affectation
52
53    list<string> ::iterator itor = listofstring.begin();
54    list<string> ::iterator itor2 = listofstring2.begin();
55    list<string> ::iterator itor3 = listofstring3.begin();
56    for (; itor != listofstring.end(); itor++, itor2++, itor3++) {
57      cout << *itor << " " << *itor2 << " " << *itor3 << endl;
58    }
59    cout << endl;
60  }
```

Et voici le résultat de l'éxécution du programme :

```
~/COURS/C++/latex/SOURCES> a.out
papa
tata
momo
bibi
toto
toto

papa
momo
bibi
toto
toto

papa
momo
bibi
toto

papa papa papa
momo momo momo
bibi bibi bibi
toto toto toto

~/COURS/C++/latex/SOURCES>
```

4.2.3 Le *template hash_map*

hash_map est une collection qui permet d'associer des objets clés à d'autre objets :

- un *hash_map* gère une collection de paires (voir http ://www.sgi.com/tech/stl/pair.html ; chaque paire contient un objet clé et un autre objet associé : <*const clé, donnée*> ;
- pour l'ensemble des couples <*const clé, donnée*>, il ne peut pas y avoir deux couples pour lesquels les clés sont égales ; ce qui signifie qu'avec une clé, une *hash_map* dispose des fonctions pour retrouver le couple et la donnée associée.

La recherche d'un élément est très efficace. Ce *template* est très utile pour construire des dictionnaires pour lesquels l'accès direct et efficace aux données indexées est important. Par contre, dans un *hash_map*, aucune relation d'ordre entre les couples ne peut être exploitée.

Ce *template* est décrit dans la documentation en ligne : http ://www.sgi.com/tech/stl/hash_map.html.

4.2.3.1 Déclaration d'un *hash_map*

C'est peut être ici que c'est le plus dur de comprendre sans avoir vu les *template* : si on désire utiliser le type T comme clé, on doit disposer de trois choses :

1. le template suivant :

   ```
   struct hash<T> {
     size_t operator()(const T & t) const
     {
       return entier_positif_a_partir_de_t(t);
     }
   };
   ```

   C'est un *template* de structure qui contient le code de l'opérateur $()$. Cet opérateur doit ici retourner un entier positif calulé à partir d'une instance ou d'une valeur de type T. C'est ce que doit calculer l'expression entier_positif_a_partir_de_t(t).

   Si T est *const char* *, alors un tel *template* est déjà donné par la STL. On peut donc directement utiliser ce type pour les clés sans donner struct hash<const char *>

2. un constructeur sans argument si T est une classe ;
3. l'operateur d'égalité $==$.

Voici quelques exemples :

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

struct Date {
  Date()  : jour_(0), mois_(0), annee_(0) {}
  Date(size_t _jour, size_t _mois, size_t _annee)
     : jour_(_jour), mois_(_mois), annee_(_annee) {}
  size_t jour_;
  size_t mois_;
  size_t annee_;
};

bool operator == (const Date & _d1, const Date & _d2)
{
  return _d1.jour_ == _d2.jour_ && _d1.mois_ == _d2.mois_ && _d1.annee_ == _d2.
      annee_;
}

ostream & operator << (ostream & _out, const Date & _date)
{
  return _out << _date.jour_ << "/" << _date.mois_ << "/" << _date.annee_;
}
```

```
23
24 struct hash<string> {
25   size_t operator()(const string & _s) const { return __stl_hash_string(_s.c_str()); }
26 };
27
28 struct hash<Date> {
29   size_t operator()(const Date & _d) const { return _d.jour_ + _d.mois_ + _d.annee_; }
30 };
31
32 int main()
33 {
34   // Déclaration d'un index qui associe un string (la cle) et une date (l'objet associe).
35   // Pour cette class cle, on doit definir la structure hash<string> (voir plus haut)
36   hash_map<string, Date> anniversaires;
37
38   // Déclaration d'un index qui associe une Date (la cle) àun string (l'objet associe).
39   // Pour cette class cle, on doit definir la structure hash<Date> (voir plus haut)
40   hash_map<Date, string> blague_du_jour;
41
42   // Déclaration d'un index qui associe une chaine C (la cle) et une date (l'objet associe)
43   // la STL definit un struct hash<const char *> par defaut
44   hash_map<const char *, string> ip_addresses;
45 }
```

4.2.3.2 Insertion/suppression et récupération d'éléments

Pour ajouter un couple, on peut utiliser l'opérateur []. Ce même opérateur est utilisé pour retrouver l'objet associé à une clé.

```
1  #include <iostream.h>
2  #include <string>
3  #include <hash_map>
4
5  struct Personne {
6    Personne() {}
7    private
8    string nom_, prenom_, adresse_;
9  };
10
11 bool operator == (const Personne & _p1, const Personne & _p2)
12 {
13   return _p1.nom_ == _p2.nom_
14 }
15
16 ostream & operator << (ostream & _out, const Date & _date)
17 {
18   return _out << _date.jour_ << "/" << _date.mois_ << "/" << _date.annee_;
19 }
20
21 struct hash<string> {
```

```
22    size_t operator()(const string & _s) const { return __stl_hash_string(_s.c_str()); }
23  };
24
25  struct hash<Date> {
26    size_t operator()(const Date & _d) const { return _d.jour_ + _d.mois_ + _d.annee_; }
27  };
28
29  int main()
30  {
31    // □Dclaration d'un index qui associe une chaine C (la cle) et une date (l'objet associe)
32    // la STL definit un struct hash<const char *> par defaut
33    hash_map<const char *, Date> aniversaires;
34
35    // Stockage dans l'index
36    aniversaires["Louise"] = Date(27,6,1995);
37    aniversaires["Martin"] = Date(25,5,1999);
38    aniversaires["□Adle"] = Date(24,12,2000);
39
40    // Recuperation des dates □partir d'une chaine C
41    cout << "Les anniversaires" << endl;
42    cout << "Louise -> " << aniversaires["Louise"] << endl;
43    cout << "Martin -> " << aniversaires["Martin"] << endl;
44    cout << "□Adle -> " << aniversaires["□Adle"] << endl;
45    cout << "toto -> " << aniversaires["toto"] << endl;
46
47    // □Dclaration d'un index qui associe une Date (la cle) □un string (l'objet associe).
48    // Pour cette class cle, on doit definir la structure hash<Date> (voir plus haut)
49    hash_map<Date, string> blague_du_jour;
50    blague_du_jour[Date(3,9,2001)] = string("C'est la □rentre de l'IUP3 □Brest");
51    blague_du_jour[Date(18,2,2002)] = string("C'est l'exam de POM/IUP3"); // a □vrifier
52
53    // Recuperation des blagues □partir des dates
54    cout << "Les bonnes blagues" << endl;
55    cout << Date(3,9,2001) << " -> " << blague_du_jour[Date(3,9,2001)] << endl;
56    cout << Date(18,2,2002) << " -> " << blague_du_jour[Date(18,2,2002)] << endl;
57    cout << Date(22,1,1964) << " -> " << blague_du_jour[Date(22,1,1964)] << endl;
58
59    // □Dclaration d'un index qui associe un string (la cle) et un string (l'objet associe)
60    // Pour cette class cle, on doit definir la structure hash<string> (voir plus haut)
61    hash_map<string, string> ip_addresses;
62    ip_addresses["cassoulet"] = "13.598.87.8";
63    ip_addresses["kelenn"] = "13.598.87.9";
64  }
```

Voici une trace d'exécution de ce programme :

```
1  ~/COURS/C++/latex/SOURCES> hash_map2
2  Les anniversaires
3  Louise -> 27/6/1995
4  Martin -> 25/5/1999□
5  Adle -> 24/12/2000
6  toto -> 0/0/0
7  Les bonnes blagues
8  3/9/2001 -> C'est la □rentre de l'IUP3 □Brest
```

```
18/2/2002 -> C'est l'exam de POM/IUP3
22/1/1964 ->
~/COURS/C++/latex/SOURCES>
```

En observant cette trace, on peut faire deux constatations :

- pour le *hash_map anniversaires*, pour la clé «toto», aucune date n'a été explicitement affectée dans le *hash_map*, par contre, on constate qu'on obtient une trace lorqu'on lit anniversaires["toto"] ;
- pour les bonnes blagues, on fait la même constatation pour la clé Date(22,1,1964).

En fait, il y a création d'un couple <*const clé, donnée*> dans un *hash_map* dès qu'on exécute un accès direct avec l'opérateur []. La clé du nouveau couple est l'objet passé entre [] et la donnée est construite avec le constructeur sans argument. Après la création du couple, l'opérateur [] retourne la référence de la donnée.

En conséquence, on ne doit pas utiliser directement l'opérateur [] pour tester si une clé est présente ou non dans un *hash_map*. Pour cela on utilise la fonction membre *find*.

Voici un exemple d'utilisation pour la gestion d'une liste d'un carnet d'adresses. L'utilisateur à le choix entre «ajouter», «retirer», «afficher» une adresse ou «lister» toutes les adresses du carnet. Si l'utilisateur demande de retirer ou d'afficher une adresse, le nom est saisi. Si ce nom est une clé dans le *hash_map* qui stocke les adresses, alors l'adresse correspondante est retirée ou affichée suivant le choix.

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

struct Personne {
  friend ostream & operator << (ostream & _out, const Personne &) ;
  friend bool operator == (const Personne &, const Personne &) ;
public :
  Personne() {}
  Personne(string _nom, string _prenom, string _email)
     : nom_(_nom), prenom_(_prenom), email_(_email) {}
private :
  string nom_, prenom_, email_ ;
} ;

bool operator == (const Personne & _p1, const Personne & _p2)
{ return _p1.nom_ == _p2.nom_ ; }

ostream & operator << (ostream & _out, const Personne & _p)
```

```
20 { return _out << _p.nom_ << " " << _p.prenom_ << " " << _p.email_ ; }
21
22 struct hash<string> {
23   size_t operator()(const string & _s) const { return __stl_hash_string(_s.c_str()) ; }
24 } ;
25
26 int main() {
27   hash_map<string, Personne> carnet ;
28   char c = 0 ;
29   while (c != 'q') {
30     cout << "(a)jouter (s)upprimer a(f)ficher (l)ister (q)uitter\n" ;
31     cin >> c ;
32     switch (c) {
33       case 'a'  : {
34   string nom, prenom, email ;
35   cout << "nom, prenom et email : " ;
36   cin >> nom >> prenom >> email ;
37   carnet[nom] = Personne(nom, prenom, email) ;
38   break ;
39       }
40       case 'l'  : {
41   // déclaration et initialisation d'un iterateur
42   hash_map<string, Personne> ::iterator itor = carnet.begin() ;
43   for (; itor != carnet.end() ; itor++) {
44     cout << (*itor).second << endl ;
45   }
46   break ;
47       }
48       case 's'  :
49       case 'f'  : {
50   cout << "nom : " ;
51   string nom ;
52   cin >> nom ;
53   if (carnet.find(nom) != carnet.end()) { // si on trouve la cle nom dans le carnet
54     if (c == 'f') {
55       cout << carnet[nom] << endl ;
56     } else {
57       carnet.erase(nom) ;
58     }
59   } else {
60     cout << "aucune adresse pour '" << nom << "'" << endl ;
61   }
62   break ;
63       }
64       default  : if (c != 'q') cout << "choix invalide." << endl ;
65     }
66   }
67 }
```

On remarque que pour parcourir un *hash_map*, on utilise un iterateur (quand on choisi «liste»). L'élément pointé par l'itérateur à chaque tour de

boucle est un couple. Pour un couple pair<const cle &, data &> cpl, la clé est accédé par cpl.first, et la donnée associée par cpl.second.

Chapitre 5

La surcharge des opérateurs

Pour C++, un opérateur est une fonction qui peut être surchargée. Il est possible de surcharger les opérateurs préexistants lorsqu'ils portent sur des objets de type classe. Il n'est par contre pas possible de créer de nouveaux opérateurs ou de surcharger les opérateurs préexistants sur les types prédéfinis.

Les opérateurs suivants peuvent être surchargés :

```
+     -     *     /     %     ^     &     |
~     !     ,     =     <     >     <=    >=
++    --    <<    >>    ==    !=    &&    ||
+=    -=    *=    /=    %=    ^=    &=    |=
<<=   >>=   []    ()    ->    ->*   new   delete
```

Les opérateurs suivant ne peuvent pas être surchargés :

```
::      .     ?:
sizeof  typeid
static_cast     dynamic_cast   const_cast    reinterpret_cast
```

En règle générale, on dispose de deux moyens exclusifs l'un de l'autre pour surcharger un opérateur. On peut surcharger un opérateur

1. soit comme une fonction membre de la classe sur laquelle porte l'opérateur, on parle de surcharge interne ;
2. soit comme une fonction exterieure à toute classe (une fonction normale quoi), on parle de surcharge externe ;

Evidemment, il y a des exceptions : les opérateurs $=$, $[]$, $()$ et $->$ ne peuvent être surchargés **que comme une fonction membre de la classe de l'opérande gauche**.

5.1 La surcharge interne

Les opérateurs sont considérés comme des fonctions membres normales de la classe sur laquelle ils s'appliquent. Le nom de ces méthodes est donné par le mot-clé $operator$, suivi de l'opérateur à redéfinir (le symbole, +=, -=, etc).

5.1.1 Syntaxe

Dans l'interface de la classe :

```
class C {
  ...
  type-retour operator op (paramètres);
  ...
};
```

- *type-retour* est le type du résultat de l'opérateur,
- *operator* est un mot clé,
- *op* est l'opérateur du langage (le symbole, +=, -=, etc),

Dans la mise en oeuvre :

```
type-retour C::
operator op (paramètres)
{
  // code de l'operateur
}
```

C'est donc complètement classique, horsmis le nom op précédé par $operator$.

Appel d'un opérateur : A op B est équivalent à la forme fonctionnelle A.operator op(B).

On voit bien alors que le premier opérande est toujours un objet de la classe C. Bien que cela ne soit aucunement obligatoire, le type de retour est souvent une référence sur la classe C. Dans ce cas, on retourne *this.

5.1.2 Exemples d'opérateurs internes

Voici à nouveau (voir 3.6.2.3) l'exemple de la classe *Tel* avec son opérateur d'affectation et son constructeur par copie correctement spécifiés.

```
1  #ifndef __TEL_H__
2  #define __TEL_H__
3
4  #include <string.h>
5  #include <stdlib.h>
6  class Tel {
7  public :
8    Tel ()  : nom(0), tel(0) {}
9    Tel (const char * _nom, const char * _tel)
10       : nom(new char [strlen(_nom)+1]),
11         tel(new char [strlen(_tel)+1])
12   {
13     strcpy(nom, _nom) ;
14     strcpy(tel, _tel) ;
15   }
16   // constructeur par copie
17   Tel (const Tel & _autre)
18   {
19     copy(_autre) ;
20   }
21   ~Tel()
22   {
23     destroy()
24   }
25   const char * get_nom() const { return nom ; }
26   const char * get_tel() const { return tel ; }
27
28   // □dfinition de l'□oprateur =
29   Tel & operator = (const Tel & _autre)
30   {
31     if (this != &_autre) {
32       destroy() ;
33       copy(_autre) ;
34     }
35     return *this ;
36   }
37
38 private :
39   void copy(const Tel & _autre)
40   {
41     nom = new char [strlen(_autre.nom)+1] ;
42     tel = new char [strlen(_autre.tel)+1] ;
```

```
43      strcpy(nom, _autre.nom);
44      strcpy(tel, _autre.tel);
45    }
46    void destroy()
47    {
48      delete nom;
49      delete tel;
50    }
51
52    char * nom;
53    char * tel;
54  };
55  #endif // __TEL_H__
```

5.1.2.1 Operateur =

Cet opérateur est obligatoirement définit et mis en œuvre comme fonction membre. Le type de retour est souvent une référence sur la classe dans laquelle il est défini pour autoriser l'appel en cascade comme *a = b = c;*. Le type de retour peut tout aussi bien être $void$. Dans ce cas, l'appel en cascade ne peut être exécuté (est ce vraiment gênant ??).

Le type de l'opérande de droite est spécifié par le type du paramètre. Ici, on veut pouvoir affecter un objet de la classe *Tel* dans un autre.

```
Tel & operator = (const Tel & _autre)
{
  if (this != &_autre) { // (1)
    destroy();           // (2)
    copy(_autre);        // (3)
  }
  return *this;          // (4)
}
```

Comme le montre l'exemple, le code comporte 4 parties :

1. on test tout d'abord qu'on a bien this != &_autre, ceci de façon à éviter de détruire l'objet en cas d'auto-affectation ;
2. ensuite, on récupère la mémoire dynamique allouée pour les variables membres ; on trouve ici le même code que dans le destructeur ;
3. on copie effectivement le contenu de l'opérande de droite dans celui de gauche (*this) ; c'est le même code que dans le constructeur par copie ;
4. et enfin, on retourne (*this)

5.1.2.2 Operateur []

Cet opérateur est habituellement utilisé pour accéder à un élément dans un tableau et plus généralement une collection. L'index est souvent numérique mais en fait ca peut être n'importe quel type (voir par exemple le chapitre 4.2.3).

Pour donner un exemple, on introduit la classe *Carnet* qui comprend une collection d'adresses :

```
1  class Carnet {
2   public :
3    Carnet()  : liste_(0), nb_(0) {}
4    void ajouter(Tel * _t) {
5      Tel ** buf = new Tel*[nb_+1];
6      for (size_t n = 0; n < nb_; n++) {
7        buf[n] = liste_[n];
8      }
9      buf[nb_++] = _t;
10     delete liste_;
11     liste_ = buf;
12   }
13
14  private :
15   Tel ** liste_;
16   size_t nb_;
17 };
```

On peut mettre en oeuvre l'opérateur [] qui renvoit l'adresse ou la référence à l'adresse du Tel placé à une certaine position dans le liste gérée par le carnet :

```
1  class Carnet {
2   public :
3    Carnet()  : liste_(0), nb_(0) {}
4    void ajouter(Tel * _t);
5
6    // Retourne le Tel stocke ☐la position _pos
7    Tel * operator [] (size_t _pos) const {
8      assert(_pos < nb_);
9      return liste_[_pos];
10   }
11
12   // Retourne le Tel stocke ☐la position _pos
13   Tel *& operator [] (size_t _pos) {
14     assert(_pos < nb_);
15     return liste_[_pos];
```

```
16    }
17
18   private :
19    Tel ** liste_;
20    size_t nb_;
21  };
```

On a deux versions du même opérateur.

- la version constante (avec le $const$ situé après la parenthèse fermante des paramètres formels) est indispensable dans une fonction membre de la classe $Carnet$ constante ou lorsque le $Carnet$ receveur est constant ;
- l'autre version, la non constante, permet de directement utiliser l'opérateur en partie gauche d'affectation.

Attention : l'utilisation de la seconde version (celle qui retourne une référence) peut être problématique. Considérons le code suivant :

Carnet c.
c.ajouter(new Tel("Bibi", "02020298"));
//...
c[0] = 0;

On a introduit un pointeur null directement dans le tableau de Tel, il faut donc que les fonctions membres de la classe $Carnet$ prennent en compte cette possibilité. De plus, on peut avoir perdu toute possibilité de libérer la mémoire dynamique occupée par l'ex $c[0]$.

On aussi peut mettre en oeuvre l'opérateur [] qui renvoie l'adresse ou la référence à l'adresse du Tel dont le nom est passé en argument. L'index est alors un $string$:

```
1  class Carnet {
2   public :
3    Carnet()  : liste_(0), nb_(0) {}
4    void ajouter(Tel * _t);
5    Tel * operator [] (size_t _pos) const;
6    Tel *& operator [] (size_t _pos);
7
8    // Retourne le Tel dont le nom est _nom
9    // ou bien 0 si non trouve
10   Tel * operator [] (string _nom) const {
11     for (size_t n = 0; n < nb_; n++) {
12       if (_nom == liste_[n]->get_nom()) {
```

```
13    return liste_[n];
14        }
15      }
16      return 0;
17    }
18
19    // Retourne le Tel dont le nom est _nom
20    // ou bien cree un nouveau Tel avec le nom _nom,
21    // l'ajoute au tableau liste_ puis le retourne
22    Tel *& operator [] (string _nom) {
23      for (size_t n = 0; n < nb_; n++) {
24        if (_nom == liste_[n]->get_nom()) {
25    return liste_[n];
26        }
27      }
28      Tel * nouv = new Tel(_nom.c_str(), "");
29      ajouter(nouv);
30      return liste_[nb_-1];
31    }
32
33   private :
34    Tel ** liste_;
35    size_t nb_;
36  };
```

On a encore deux versions du même opérateur.

- la version constante qui renvoit 0 en cas de recherche infructueuse ;
- l'autre version, la non constante, ne peut pas renvoyer la valeur 0 puisque le résultat est forcément la référence à un pointeur existant ; une solution serait de ne pas définir cet opérateur (bonne solution à mon avis) ; une autre solution est, dans le cas d'une recherche infructueuse, de crée un nouveau Tel avec le nom et de l'ajouter à la liste ; le résultat retourné est alors la référence à ce nouveau pointeur ;

5.1.2.3 Un opérateur arithmétique

On peut par exemple définir l'opérateur += de façon à pouvoir écrire c += new Tel("Bibi", "02020298"); au lieu de c.ajouter(new Tel("Bibi", "02020298"));. On peut aussi définir les autres opérateurs arithmétiques comme -=, *=, /=. La question est, **quel est l'intéret ?**.

```
1   class Carnet {
2    public :
3     Carnet()  : liste_(0), nb_(0) {}
4     void ajouter(Tel * _t);
5     Tel * operator [] (size_t _pos) const;
```

```
6     Tel *& operator [] (size_t _pos);
7     Tel * operator [] (string _nom) const;
8     Tel *& operator [] (string _nom);
9     Carnet & operator += (Tel * _t) {
10      ajouter(_t);
11      return *this;
12    }
13
14   private :
15    Tel ** liste_;
16    size_t nb_;
17  };
```

5.1.2.4 Incrémentation et décrémentation

Les opérateurs $++$ et $--$ peuvent être utilisés de deux façons : avant ou après une expression. Ces opérateurs n'ont pas d'argument. Cependant, un argument fictif est introduit pour indiquer qu'il s'agit d'une post-incrémentation ou d'une post-décrémentation :

```
1   class Entier {
2   public :
3     Entier(int j)  : i(j) {}
4     // post-incrmentation
5     Entier operator++(int) {
6       Entier tmp(i++);
7       return tmp;
8     }
9     // pr-incrmentation
10    Entier operator++() {
11      Entier tmp(++i);
12      return tmp;
13    }
14   private :
15      int i;
16  };
```

5.1.2.5 Transtypage

Dans une classe C, un constructeur peut être utilisé pour convertir des objets du type de leur paramètre (par exemple une classe D) vers le type de leur classe (C). Pour pouvoir écrire un tel constructeur, il faut pouvoir modifier la classe (disposer des sources de la classe C). C'est pas toujours le cas (par exemple avec les classes standards ou avec des classes achetées et disponibles sous-licence).

Si on peut modifier la classe D, pour convertir un D vers un C une autre solution consiste à définir un opérateur de transtypage ou de $cast$: D::operator C();. Le code suivant permet d'exécuter cet opérateur :

```
D d;
// ...
C c = (C) d; // transtypage de D vers C.
```

Notons qu'on peut aussi transtyper vers un type de base (un type entier par exemple) ou vers un type pointeur (const char * par exemple).

Reprenons l'exemple de notre classe $Entier$. Supposons que l'on veuille pouvoir convertir une objet de cette classe vers un $string$. La première solution consiste donc à développer un nouveau constructeur dans la classe $string$: string::string(const Entier &);. Or ce n'est pas possible si on utilise une classe $string$ standard. L'autre solution consiste à développer l'opérateur de transtypage dans la classe $Entier$:

```
1  class Entier {
2  public :
3    Entier(int j)  : i(j) {}
4    Entier operator++(int);
5    Entier operator++();
6
7    // Transtypage vers string
8    operator string () const {
9      char buf[512];
10     sprintf(buf, "%d", i);
11     return buf;
12   }
13
14  private :
15     int i;
16 };
```

5.2 La surcharge externe

Les opérateurs sont considérés comme des fonctions globales normales. Le nom de ces fonctions est donné par le mot-clé $operator$, suivi de l'opérateur à redéfinir (le symbole, ==, >, etc). On indique ensuite les paramètres. Le nombre de paramètre(s) dépend de l'arité de l'opérateur. Ces opérateurs sont souvent indiqués comme amis ($friend$) de la classe d'un des opérandes.

5.2.1 Syntaxe

Le prototype C'est une fonction normale :

```
type-retour operator op (opérandes);
```

Comme fonction amie de la classe C :

```
class C {
  friend type-retour operator op (operandes);
  // ...
};
```

Dans la mise en oeuvre :

```
type-retour operator op (opérandes)
{
  // code de l'operateur
}
```

Appel d'un opérateur : A op B est équivalent à la forme fonctionnelle operator op(A,B).

5.2.2 Exemples d'opérateurs externes

5.2.2.1 Empilement dans un flot

En C++, la classe standard *ostream* met en oeuvre les flots de sorties. Les flots *cout* et *cerr* sont disponibles à la place de respectivement *stdout* et *stderr*.

L'opérateur d'empilement $<<$ est une fonction qui permet la sortie d'une valeur, d'une expression ou d'un objet sur un flot *ostream*.

La ligne de code suivante

```
cout << une_expression;
```

permet d'envoyer (d'empiler) l'expression *une_expression* sur la sortie standard. Pour cela, il faut que l'opérateur $<<$ soit définit pour le type de l'expression. En standard, cet opérateur est disponible pour tous les types de base du C++ (*int, char, const char*, double, float*). Pour les autres types (les classes applicatives par exemple), c'est à la charge du développeur de mettre en œuvre cet opérateur.

La classe *ostream* ne peuvent pas être enrichies par un développeur lambda pour disposer d'opérateurs pour ses propres types. La seule solution est donc bien de définir ces opérateurs en externe.

Pour une classe *C*, le squelette de la mise en oeuvre de l'opérateur << est :

```
ostream & operator << (ostream & out, const C & obj)
{
  // empilement sur out des variables membres de l'objet en utilisant <<
  return out;
}
```

Voici un exemple :

```
#include <iostream.h >>

const size_t Max = 10000 ;

class T {
   int tab[Max] ;
   size_t nb ;

public  :
   T() : nb(0) {} ;
   T(int _nb, int * _values) ;
   // ...
   friend ostream & operator << (ostream &, const T&) ;
} ;

ostream & operator << (ostream & out, const T & t)
{
   out << t.nb ;
   for (size_t i = 0 ; i < t.nb ; i++) {
      out << t[i] << " " ;
   }
   return out ;
}

int main(void)
{
   int t[] = {1,2,3} ;
   T tab(3, t) ;
   cout << tab ; // affichera "3 1 2 3" sur la sortie standard
   return 0 ;
}
```

Remarque : Cet opérateur est souvent mis en œuvre pour faciliter le déverminage du code. En effet, il permet assez simplement de contrôler

la valeur des variables membres d'un objet.

Pour nos classes Tel et $Carnet$, voici la déclaration de l'opérateur d'empilement :

```
class Tel {
  friend ostream & operator << (ostream & _out, const Tel &);
 public :
  Tel ()  : nom(0), tel(0) {}
  Tel (const char * _nom, const char * _tel);
  // ...
 private :
  char * nom;
  char * tel;
};

class Carnet {
  friend ostream & operator << (ostream & _out, const Carnet &);
 public :
  Carnet()  : liste_(0), nb_(0) {}
  void ajouter(Tel * _t);

 private :
  Tel ** liste_;
  size_t nb_;
};
```

Et voici leur mise en œuvre :

```
ostream & operator << (ostream & _out, const Tel & _tel)
{
  _out << _tel.nom << " " << _tel.tel;
  return _out;
}

ostream & operator << (ostream & _out, const Carnet & _carnet)
{
  _out << _carnet.nb_ << " ";
  for (size_t n = 0; n < _carnet.nb_; n++) {
    _out << *_carnet[n] << endl;
  }
  return _out;
}
```

On peut maintenant ecrire :

Carnet c;

```
c.ajouter(new Tel(”Bibi”, ”02020298”));
c.ajouter(new Tel(”Toto”, ”00030268”));
cout << c;
```

5.2.2.2 Dépilement d’un flot

En C++, la classes standard *istream* met en oeuvre les flots d’entrées. Le flot *cin* est disponible à la place de *stdin*.

L’opérateur de dépilement >> est une fonction qui permet la lecture d’une expression ou d’un objet depuis un flot *istream*.

La ligne de code suivante

```
cin >> une_variable
```

permet de lire (de dépiler) une valeur depuis l’entrée standard et de l’affecter à *une_variable*. Pour cela, il faut que l’opérateur >> soit définit pour le type de *une_variable*. En standard, cet opérateur est disponible pour tous les types de base du C++ (*int, char, char[], double, float*). Tout comme pour la classe *ostream*, pour les autres types (les classes applicatives par exemple), c’est à la charge du développeur de mettre en œuvre cet opérateur.

Pour une classe *C*, le squelette de la mise en oeuvre de l’opérateur >> est :

```
istream & operator >> (istream & in, C & obj)
{
  // dépilement depuis in des variables membres de l’objet en utilisant >>
  return in;
}
```

Voici un exemple :

```
1  #include <iostream.h >>
2
3  const size_t Max = 10000 ;
4
5  class T {
6    int tab[Max] ;
7    size_t nb ;
8
9  public  :
10   T() : nb(0) {} ;
11   T(int _nb, int * _values) ;
12   // ...
```

```
13    friend ostream & operator << (ostream &, const T&) ;
14    friend istream & operator >> (istream &, T&) ;
15  } ;
16
17  istream & operator >> (istream & in, T & t)
18  {
19    in >> t.nb ;
20    for (size_t i = 0 ; i < t.nb ; i++) {
21      in >> t[i] ;
22    }
23    return in ;
24  }
25
26  int main(void)
27  {
28    T tab ;
29    cin >> tab ; // lecture du tableau sur l'entree standard
30    cout << tab ;
31    return 0 ;
32  }
```

Pour nos classes Tel et $Carnet$, voici la déclaration de l'opérateur de dépilement :

```
1   class Tel {
2     friend ostream & operator << (ostream & _out, const Tel &) ;
3     friend istream & operator >> (istream & _in, Tel &) ;
4    public :
5     Tel ()  : nom(0), tel(0) {}
6     Tel (const char * _nom, const char * _tel) ;
7     // ...
8    private :
9     char * nom ;
10    char * tel ;
11  } ;
12
13  class Carnet {
14    friend ostream & operator << (ostream & _out, const Carnet &) ;
15    friend istream & operator >> (istream & _in, Carnet &) ;
16   public :
17    Carnet()  : liste_(0), nb_(0) {}
18    void ajouter(Tel * _t) ;
19
20   private :
21    Tel ** liste_ ;
22    size_t nb_ ;
23  } ;
```

Et voici leur mise en œuvre :

```
istream & operator >> (istream & _in, Tel & _tel)
{
  // Attention, on considere ici qu'un nom ne peut □dpasser 1023 caracteres
  // pour □□gnraliser il faudrait lire caractere par caractere avec _in.get()
  char nom[1024];
  char tel[10];
  _in >> nom >> tel; // lecture des buffers depuis _in
  _tel.set_nom(nom); // on utilise les accesseurs pour modifier les variables
  _tel.set_tel(tel); // membres nom et tel
  return _in;
}

istream & operator >> (istream & _in, Carnet & _carnet)
{
  size_t nb;
  _in >> nb; // Combien de Tel a lire
  for (size_t n = 0; n < nb; n++) {
    Tel * t = new Tel; // un nouveau Tel vide
    _in >> (*t); // on lit le Tel
    _carnet.ajouter(t); // et on l'ajoute au carnet
  }
  return _in;
}
```

On peut maintenant ecrire :

```
Carnet c;
Tel * t = new Tel;
cin >> (*t);
c.ajouter(t);
cout << c;
```

Ou encore :

```
Carnet c;
cin >> c; // lecture au clavier d'un carnet
cout << c; //ecriture sur l'écran
```

5.2.2.3 Opérateurs arithmétiques et logiques

Certains opérateurs arithmétiques et logiques sont le plus souvent définis en externes. Voici des exemples pour la classe Entier.

La déclaration :

```
class Entier {
  friend Entier operator + (const Entier &, const Entier &);
  friend bool operator > (const Entier &, const Entier &);
public :
  Entier(int j)  : i(j) {}
  Entier operator++(int);
  Entier operator++();
  operator string () const;
 private :
    int i;
};
```

La mise en œuvre :

```
Entier operator + (const Entier & _e1, const Entier & _e2)
{
  return Entier(_e1.i + _e2.i);
}

bool operator > (const Entier & _e1, const Entier & _e2)
{
  return _e1.i > _e2.i;
}
```

On peut maintenant ecrire :

```
Entier eg(8);
Entier ed(87);
Entier er(0);
if (eg > ed) {
  er = eg + ed;
}
```

Autres exemples : voici des exemples donnés dans le cours de *Christian Casteyde* (http ://casteyde.christian.free.fr/cpp/cours/online/book1.html)

```
class complexe
{
    friend complexe operator+(const complexe &, const complexe &);
    friend complexe operator-(const complexe &, const complexe &);
    friend complexe operator*(const complexe &, const complexe &);
    friend complexe operator/(const complexe &, const complexe &);

    double m_x, m_y; // Les parties □relles et imaginaires.
public :
    // Constructeurs et □oprateur de copie  :
    complexe(double x=0, double y=0);
```

```
12      complexe(const complexe &);
13      complexe &operator=(const complexe &);
14
15      // Fonctions permettant de lire les parties □relles
16      // et imaginaires  :
17      double re(void) const;
18      double im(void) const;
19
20      // Les □oprateurs de base :
21      complexe &operator+=(const complexe &);
22      complexe &operator-=(const complexe &);
23      complexe &operator*=(const complexe &);
24      complexe &operator/=(const complexe &);
25  };
26
27  complexe &complexe ::operator+=(const complexe &c)
28  {
29      m_x += c.m_x;
30      m_y += c.m_y;
31      return *this;
32  }
33
34  complexe &complexe ::operator-=(const complexe &c)
35  {
36      m_x -= c.m_x;
37      m_y -= c.m_y;
38      return *this;
39  }
40
41  complexe &complexe ::operator*=(const complexe &c)
42  {
43      double temp = m_x*c.m_x -m_y*c.m_y;
44      m_y = m_x*c.m_y + m_y*c.m_x;
45      m_x = temp;
46      return *this;
47  }
48
49  complexe &complexe ::operator/=(const complexe &c)
50  {
51      double norm = c.m_x*c.m_x + c.m_y*c.m_y;
52      double temp = (m_x*c.m_x + m_y*c.m_y) / norm;
53      m_y = (-m_x*c.m_y + m_y*c.m_x) / norm;
54      m_x = temp;
55      return *this;
56  }
57
58  complexe operator+(const complexe &c1, const complexe &c2)
59  {
60      complexe result = c1;
61      return result += c2;
62  }
63
64  complexe operator-(const complexe &c1, const complexe &c2)
65  {
66      complexe result = c1;
```

```
67      return result -= c2 ;
68  }
69
70  complexe operator*(const complexe &c1, const complexe &c2)
71  {
72      complexe result = c1 ;
73      return result *= c2 ;
74  }
75
76  complexe operator/(const complexe &c1, const complexe &c2)
77  {
78      complexe result = c1 ;
79      return result /= c2 ;
80  }
```

5.3 Choisir entre surcharge interne et externe

Pour certains opérateurs, il n'y a pas le choix :

- les opérateurs $=$, $[]$, $()$ et $->$ ne peuvent être surchargés que comme une fonction membre de la classe de l'opérande gauche ;
- les operateurs pour lesquels on ne peut modifier la classe de l'opérande gauche (comme par exemple $<<$ et $>>$ avec les *stream* ou pour le transtypage) sont mis en œuvre en externe.

Pour les autres, on a le choix : on choisit plutot la mise en œuvre en interne lorsque l'opérateur modifie l'opérande gauche. C'est par exemple le cas des opérateurs +=, *=, ...

Chapitre 6

Les exceptions

Le C++ dispose des exceptions logicielles dont le but est, pour une utilisation classique de celles ci, de gérer les erreurs qui surviennent lors de l'exécution des programmes.

Les exceptions permettent une gestion simplifiée des erreurs, parce qu'elles en reportent le traitement. Le code peut alors être écrit sans se soucier des cas particuliers, ce qui le simplifie grandement. Les cas particuliers sont traités dans les gestionnaires d'exception.

Le mécanisme des exceptions du C++ garantit que tous les objets de classe de stockage automatique sont détruits lorsque l'exception qui remonte sort de leur portée. Ainsi, si toutes les ressources sont encapsulées dans des classes disposant d'un destructeur capable de les détruire ou de les ramener dans un état cohérent, la remontée des exceptions effectue automatiquement le ménage. De plus, les exceptions peuvent être typées, et caractériser ainsi la nature de l'erreur qui s'est produite. Ce mécanisme est donc strictement équivalent en termes de fonctionnalités aux codes d'erreurs utilisés précédemment.

6.1 Lancement et récupération d'une exception

Pour gérer une erreur, on doit disposer d'une classe qui caractérise cette erreur. Lorsqu'une erreur est detectée, une exception doit être lancée. Une exception est un objet de la classe qui caractérise l'erreur. Pour lancer l'exception on utilise l'instruction *throw* suivit de l'exception (l'objet de la classe qui caractérise l'erreur).

throw objet;

L'exception doit alors être traitée par la routine d'exception correspondante. On ne peut attraper que les exceptions qui sont apparues dans une zone de code limitée (cette zone est dite protégée contre les erreurs d'exécution), pas sur tout un programme. On doit donc placer le code susceptible de lancer une exception d'un bloc d'instructions particulier. Ce bloc est introduit avec le mot-clé *try*. Les gestionnaires d'exceptions doivent suivre le bloc *try*. Ils sont introduits avec le mot-clé *catch*.

```
try {
   // Code susceptible de générer des exceptions...
} catch (classe temp) {
   // Traitement de l'exception associée à la classe
}
```

Il peut y avoir plusieurs gestionnaires d'exceptions. Chacun traitera les exceptions qui ont été générées dans le bloc try et dont l'objet est de la classe indiquée par son paramètre.

Il est possible de définir un gestionnaire d'exceptions universel, qui récupérera toutes les exceptions possibles, quels que soient leurs types. Ce gestionnaire d'exception doit prendre comme paramètre trois points de suspension entre parenthèses dans sa clause catch. Bien entendu, dans ce cas, il est impossible de spécifier une variable qui contient l'exception, puisque son type est indéfini.

Dans l'exemple suivant, selon ce qu'entre l'utilisateur, une exception du type *erreur*, *int* ou *other* est générée.

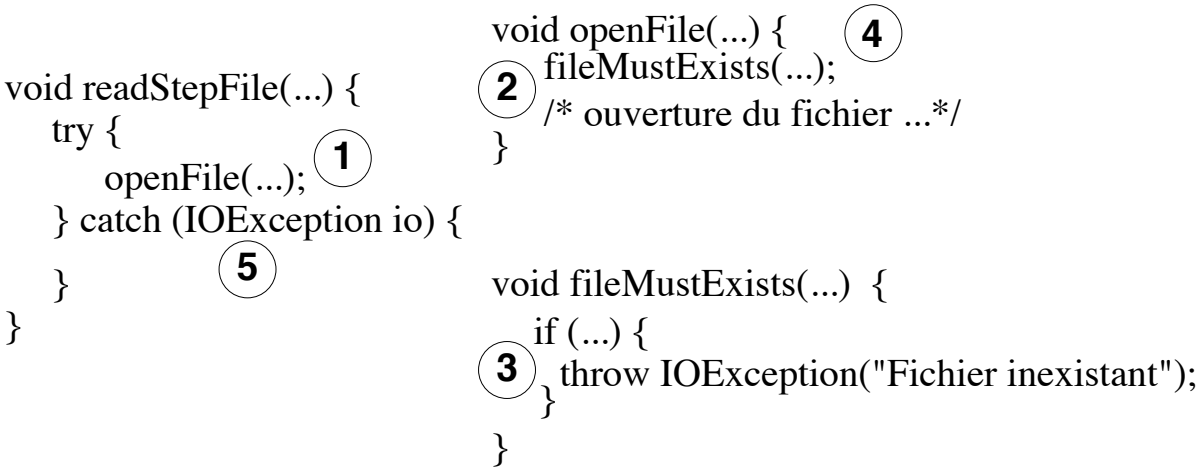
```
1  #include <iostream>
2
3  class erreur {
4  public :
5    int cause ; // Entier □spcifiant la cause de l'exception.
6    erreur(int c)  : cause(c) {}
7  } ;
```

```
8
9  class other {} ; // Objet correspondant □toutes les autres exceptions.
10
11 int main() {
12   int i ; // Type de l'exception □□□gnrer.
13   cout << "Tapez 0 pour □□gnrer une exception Erreur, 1 pour une □Entire  :" ;
14   cin >> i ; // On va □□gnrer une des trois exceptions possibles.
15
16   // Lancement des exceptions
17   try { // Bloc □o les exceptions sont prises en charge.
18     switch (i) { // Selon le type d'exception □□dsire,
19       case 0 : {
20   throw erreur(0) ; // on lance l'objet correspondant (ici, de classe erreur).
21       }
22       case 1 : {
23   throw (i) ; // Exception de type entier.
24       }
25       default : {
26   throw other() ;
27       }
28     }
29   } catch (erreur tmp) { // Traitement de l'exception erreur
30     cout << "Erreur erreur ! (cause " << tmp.cause << ")" << endl ;
31   } catch (int tmp) { // Traitement de l'exception int
32     cout << "Erreur int ! (cause " << tmp << ")" << endl ;
33   } catch (...) { // Traitement de toutes les autres exceptions
34     cout << "Exception inattendue !" << endl ;
35   }
36
37   return 0 ;
38 }
```

6.2 Remontée des exceptions

Lorsqu'une exception est lancée, arrête l'exécution et recherche dans le contexte courant un éventuel bloc *catch* pour l'exception lancée. Si le bloc est trouvé, alors le traitement du bloc est exécuté. Sinon, il y a remontée et recherche du *catch* dans le contexte de l'appelant.

Dans la figure ci-dessous, l'arbre d'appel est explicitement indiqué par un numéro d'ordre. Lorque l'exception *IOException* est lancée, il y a remontée jusqu'au premier bloc *catch* qui traite les *IOException* :



Voici un exemple qui montre la remontée avec des traitements d'erreur à différents niveaux de l'arbre d'appel :

```
#include <iostream>

class erreur {
public :
  int cause ; // Entier □spcifiant la cause de l'exception.
  erreur(int c)  : cause(c) {}
} ;

class MaClasseTest {
public :

  // Fonction qui lance les exceptions
  void traiter() {
    cout << "Tapez 0 pour □□gnrer une exception Erreur, 1 pour une □Entire  :" ;
    cin >> i ;
    switch (i) {
      case 0 : {
  throw erreur(0) ;
      }
      case 1 : {
  throw (i) ;
      }
      default : {
  throw "Erreur" ; // // Exception de type const char *
      }
    }
  }

  // fonction intermediaire qui ne traite qu'un type d'exception
  // et relaye les autre
  void sous_sous_traiter() {
    bool ok = false ;
    while (!ok) {
```

```
34          try {
35     sous_traiter();
36     ok = true; // pas d'exception, on peut sortir
37          } catch (erreur tmp) {
38     cout << "Erreur erreur ! (cause " << tmp.cause << ")" << endl;
39          }
40       }
41     }
42
43     // fonction appelante qui ne traite aucune exception
44     void sous_traiter() {
45       traiter();
46     }
47
48  private :
49     int i;
50  };
51
52  int main() {
53     MaClasseTest t;
54
55     try {
56       t.sous_sous_traiter();
57     } catch (int tmp) {
58       cout << "Erreur int ! (cause " << tmp << ")" << endl;
59     } catch (...) {
60       cout << "Exception inattendue !" << endl;
61     }
62     return 0;
63  }
```

Pour relancer l'exception en cours de traitement dans un gestionnaire d'exception, il faut utiliser le mot-clé *throw*. La syntaxe est la suivante :

throw;

L'exception est alors relancée, avec comme valeur l'objet que le compilateur a construit en interne pour propager l'exception. On peut ainsi disposer de plusieurs bloc *catch* pour la même exception mais a des niveaux différents de l'arbre d'appel. Voici l'exemple précédent avec plusieurs blocs *catch* pour l'exception *erreur* :

```
1  #include <iostream>
2
3  class erreur {
4  public :
5     int cause; // Entier □spcifiant la cause de l'exception.
6     erreur(int c)  : cause(c) {}
7  };
8
```

```
9  class MaClasseTest {
10 public :
11
12   // Fonction qui lance les exceptions
13   void traiter() {
14     cout << ”Tapez 0 pour □□gnrer une exception Erreur, 1 pour une □Entire  :”;
15     cin >> i;
16     switch (i) {
17       case 0 : {
18   throw erreur(0);
19       }
20       case 1 : {
21   throw (i);
22       }
23       default : {
24   throw ”Erreur”; // // Exception de type const char *
25       }
26     }
27   }
28
29   // fonction intermediaire qui ne traite qu’un type d’exception
30   // et relaye les autre
31   void sous_sous_traiter() {
32     try {
33       sous_traiter();
34     } catch (erreur tmp) {
35       cout << ”Erreur erreur ! (cause ” << tmp.cause << ”) traitement dans
             sous_sous_traiter” << endl;
36       throw; // on relance la □mme exception
37     }
38   }
39
40   // fonction appelante qui ne traite aucune exception
41   void sous_traiter() {
42     try {
43       traiter();
44     } catch (erreur tmp) {
45       cout << ”Erreur erreur ! (cause ” << tmp.cause << ”) traitement dans
             sous_traiter” << endl;
46       throw; // on relance la □mme exception
47     }
48   }
49
50 private :
51   int i;
52 };
53
54 int main() {
55   MaClasseTest t;
56
57   try {
58     t.sous_sous_traiter();
59   } catch (erreur tmp) {
60     cout << ”Erreur erreur ! (cause ” << tmp.cause << ”) traitement dans main” <<
         endl;
```

```
61    } catch (int tmp) {
62      cout << ”Erreur int ! (cause ” << tmp << ”)” << endl;
63    } catch (...) {
64      cout << ”Exception inattendue !” << endl;
65    }
66    return 0;
67  }
```

6.3 Traitement des exceptions non attrapées

Si, lorsqu'une exception se produit dans un bloc *try*, il est impossible de trouver le bloc *catch* correspondant à la classe de cette exception, il se produit une erreur d'exécution. La fonction prédéfinie *terminate* est alors appelée. Elle se contente d'appeler une fonction de traitement de l'erreur, qui elle-même appelle la fonction *abort* de la librairie C. Cette fonction termine l'exécution avec *abort()*. Si ce n'est pas le comportement désiré, il est possible de le modifier en indiquant une autre fonction à appeler.

Pour cela, on dispose de la fonction *set_terminate*, qui attend en paramètre un pointeur sur la fonction de traitement d'erreur, qui ne prend aucun paramètre et renvoie *void*. La valeur renvoyée par *set_terminate* est le pointeur sur la fonction de traitement d'erreur précédente. *set_terminate* est déclarée dans le fichier d'en-tête *exception*.

Dans l'exemple suivant, l'utilisateur choisi ou non d'intaller un autre gestionnaire d'exception par defaut. Si *mon_gestionnaire* n'est pas installé, alors le programme se termine par *abort()*, sinon, le programme se termine par l'appel à *mon_gestionnaire*.

```
1   #include <iostream>
2   #include <exception>
3
4   void mon_gestionnaire(void)
5   {
6       cout << ”Exception non □□gre □reue !” << endl;
7       cout << ”Je termine le programme proprement...” << endl;
8       exit(-1);
9   }
10
11  int lance_exception(void)
12  {
13      throw 2;
14  }
15
16  int main(void)
17  {
18    bool rep;
```

```
19    cout << "Avec gestionnaire ? " ;
20    cin >> rep ;
21    if (rep) {
22       set_terminate(&mon_gestionnaire) ;
23    }
24    lance_exception() ;
25    return 0 ;
26 }
```

6.4 Déclaration des exceptions autorisées pour une fonction

Il est possible de spécifier les exceptions qui peuvent apparaître dans une fonction. Pour cela, il faut faire suivre son en-tête du mot-clé *throw*, avec entre parenthèses, et séparées par des virgules, les classes des exceptions envisageables. Par exemple, la fonction suivante :

```
int fonction_sensible(void) throw (int, double, erreur)
{
  // ...
}
```

La fonction *fonction_sensible* n'a le droit de lancer que des exceptions du type int, double ou erreur. Si une autre exception est lancée, par exemple une exception du type *char* *, la fonction *unexpected* est appelée. Cette fonction appele la fonction *terminate* (voir chapitre 6.3).

On peut changer ce comportement par défaut en remplaçant la fonction appelée avec *set_unexpected*. Cette fonction standard est déclarée dans le fichier d'en-tête *exception*. Elle attend en paramètre un pointeur sur la fonction de traitement d'erreur. Cette dernière ne prend aucun paramètre et renvoie *void*.

Voici un exemple d'installation d'un nouveau gestionnaire :

```
1  #include <iostream>
2  #include <exception>
3
4  void mon_gestionnaire(void)
5  {
6     cout << "Une exception □illgale a □□t □lance." << endl ;
7     cout << "Je relance une exception de type int." << endl ;
8     throw 2 ;
9  }
```

```
10
11  class erreur {
12  public :
13    int cause ; // Entier □spcifiant la cause de l'exception.
14    erreur(int c)  : cause(c) {}
15  } ;
16
17  class MaClasseTest {
18  public :
19
20    // Fonction qui lance les exceptions
21    void traiter() throw (erreur, int) {
22      cout << "Tapez 0 pour □□gnrer une exception Erreur, 1 pour une □Entire  :" ;
23      cin >> i ;
24      switch (i) {
25        case 0 : {
26    throw erreur(0) ;
27        }
28        case 1 : {
29    throw (i) ;
30        }
31        default : {
32    throw "Erreur" ; // // Exception de type const char *
33        }
34      }
35    }
36  private :
37    int i ;
38  } ;
39
40  int main() {
41    set_unexpected(&mon_gestionnaire) ;
42
43    MaClasseTest t ;
44    try {
45      t.traiter() ;
46    } catch (erreur tmp) {
47      cout << "Erreur erreur  ! (cause " << tmp.cause << ") traitement dans main" <<
          endl ;
48    } catch (int tmp) {
49      cout << "Erreur int  ! (cause " << tmp << ")" << endl ;
50    } catch (...) {
51      cout << "Exception inattendue  !" << endl ;
52    }
53    return 0 ;
54  }
```

Chapitre 7

Les espaces de nommage

7.1 Définition

Les espaces de nommage permettent de définir des contextes de validité pour des symboles. Il est ainsi possible d'éviter des problèmes de conflit de nom entre plusieurs composants logiciels.

Pour déclarer un espace de nommage, on utilise l'instruction *namespace* suivit du nom de l'espace :

```
namespace nom
{
  // les déclarations
}
```

Les déclarations peuvent être des classes, des fonctions, des constantes ou des variables (comme d'habitude quoi). La différence avec la déclaration d'une classe ou d'une structure est qu'un espace de nommage peut être scindé en plusieurs parties disjointes. On peut donc étendre un espace de nommage :

```
1 namespace A // ☐Dclaration de l'espace de nommage A.
2 {
3     int i;
4 }
5
6 namespace B // ☐Dclaration de l'espace de nommage B.
```

```
{
    int i ;
}

namespace A // Extension de l'espace de nommage A.
{
    int j ;
}
```

L'accès aux identificateurs des espaces de nommage se fait par défaut grâce à l'opérateur de résolution de portée *::* :

```
int i = 1 ; // variable globale

class C { // une classe C
} ;

void f(void) ; // une fonction f

namespace A
{
  int i = 2 ; // i de l'espace de nommage A.
  int j = i ; // Utilise A ::i.
  class C { // la classe A ::C
  } ;
  void f(void) ; // la fonction A ::f
}

int main(void)
{
  i=1 ; // Utilise  ::i, soit la variable globale.
  A ::i=3 ; // Utilise A ::i, soit la variable i □□dclare dans l'espace A
  C c ; // Un objet de la classe  ::C
  A ::C c2 ; // Un objet de la classe A ::C
  A ::f() ; // appel de la fonction f de l'espace A
  return 0 ;
}
```

On peut scinder la déclaration des prototypes de celle de la mise en oeuvre des fonctions :

```
namespace A
{
  int f(void) ; // prototype de A ::f.
  int g(void) ; // prototype de A ::g.
}

int A ::f(void) // mise en oeuvre de A ::f.
{
```

```
  return 0;
}

namespace A
{
  // mise en oeuvre de A ::g.
  int g(void)
  {
  }
}
```

Lorsqu'un espace de nommage porte un nom très compliqué, il peut être avantageux de définir un alias pour ce nom. L'alias aura alors un nom plus simple. Cette opération peut être réalisée à l'aide de la syntaxe suivante :

namespace nom_alias = nom;

On utilise ensuite *nom_alias* à la place de *nom*.

7.2 Utilisation d'un espace de nommage

Pour utiliser les déclaration d'une espace de nommage, on a vu qu'on doit utiliser l'opérateur de portée précédé du nom de l'espace de nommage. Cela peut être contraignant. Avec *using namespace* suivit du nom d'un espace de nommage, on peut utiliser les déclarations de l'espace sans utiliser l'opérateur de portés. Cela permet de simplifier l'écriture des programmes.

Cependant, comme le montre l'exemple suivant, les ambigüités doivent être levées explicitement avec l'opérateur de portée :

```
#include <iostream.h>

namespace A
{
  int f(void); // prototype de A ::f.
  int g(void); // prototype de A ::g.
}

namespace A
{
  int f(void)
  {
    cout << "A ::f()" << endl;
  }
  int g(void)
  {
    cout << "A ::g()" << endl;
  }
```

```
19 }
20
21 void g(void)
22 {
23   cout << " ::g()" << endl;
24 }
25
26 main()
27 {
28   using namespace A;
29   f(); // appel de A ::f();
30   A ::g(); // appel de A ::g();
31   g(); // !!! Compilation impossible, □ambiguit entre  ::g() et A ::g()
32    ::g(); // appel de  ::g() OK
33 }
```

Chapitre 8

L'héritage

L'héritage permet de définir des nouvelles classes dites dérivées à partir de la définition d'une ou plusieurs autres classes appelées alors classes de base. L'héritage constitue une forme de réutilisation puissante car tout ou partie des classes de bases sont réutilisés par les classes dérivées.

Avec l'héritage s'ajoute un nouveau type de contexte de déclaration des membres d'une classe : le contexte *protected*. Il se spécifie de la même façon que les contextes *private* et *public*, dans la déclaration de la classe. Ce contexte spécifie que les membres sont accessibles par les fonctions membres de la classe et par les fonctions membres des classes dérivées. Du point de vue de l'extérieur de la classe, le contexte *protected* est par contre équivalent au contexte *private*.

Attention : l'opérateur d'affectation n'est pas hérité, même s'il est déclaré dans le contexte public de la classe de base.

8.1 Accès à la classe de base

On dispose de deux types d'héritage : l'héritage publique et l'héritage privé :

- si l'héritage est public, le mot clé *public* est spécifié avant le nom de la classe de base :

  class B : public A {...};

  signifie que *B* hérite publicment de la classe *A* ; les membres *private* de *A* sont alors inaccessibles par *B* et ses sous-classes, les membres *protected* et *public* de *A* sont accessibles par les fonctions membres de *B* ;
- si l'héritage est privé, le mot clé *private* est spécifié avant le nom de la classe de base :

  class B : private A {...};

  du point de vue de *B*, rien ne change par rapport à l'héritage public. C'est du point de vue des éventuelles ou futures sous-classes de *B* que cela change ; en effet, les fonctions membres des sous-classes de *B* ne peuvent plus accéder aux membres *protected* de la classe *A*.

C'est pratiquement toujours l'héritage public qui est utilisé.

Voici un exemple récapitulatif :

```
1  class A {
2  private :
3    int pva ;
4  protected :
5    int pra ;
6  public :
7    int pua ;
8  } ;
9
10 // La classe B ☐hrite publiquement de la classe A
11 class B  : public A {
12 protected :
13   int prb ;
14 public :
15   int pub ;
16   int fpra() { pra = 2 ; } // OK '
17   int fpua() { pua = 3 ; }// OK
18   int fpva() { pva = 4 ; } // Impossible car ☐porte private de A ::pva
19 } ;
20
21 // La classe C ☐hrite de la classe B, l'☐hritage est ☐priv
```

```
class C  : private B {
protected :
   int prc;
public :
   int puc;
   int fpra() { pra = 2; } // OK .
   int fpua() { pua = 3; } // OK
   int fpva() { pva = 4; } // Impossible, ☐porte private de A ::pva
};

// La classe D ☐hrite publiquement de la classe C
class D  : public C {
   int fpra() { pra = 2; } // Impossible, ☐hritage ☐priv de C vers B
   int fpua() { pua = 3; } // Impossible, ☐hritage ☐priv de C vers B
   int fpva() { pua = 4; } // Impossible, ☐porte private de A ::pva
};

main()
{
   A a;
   a.pua = 0; // OK
   a.pra = 0; // Impossible, ☐porte protected de A ::pra

   B b;
   b.pua = 5; // OK
   b.pub = 6; // OK
   b.pra = 2; // Impossible, ☐porte protected de A ::pra

   C c;
   c.pub = 6; // Impossible : ☐hritage ☐priv de C vers B
   c.pua =10; // Impossible  : ☐hritage ☐priv de C vers B
   c.pra = 12; // Impossible, ☐porte protected de A ::pra
}
```

Voici un autre exemple avec de vrais objets : un étudiant, un enseignant et un chercheur sont des personnes.

```
class Personne {
public :
   Personne(string _nom, string _prenom);
   string nom() const;
   string prenom() const;
protected :
   string nom_;
   string prenom_;
};

class Etudiant  : public Personne {
public :
   Etudiant(string _nom, string _prenom, string _numet);
```

```
14 protected :
15   string numet_ ;
16 } ;
17
18 class Enseignant  : public Personne {
19 public :
20   Enseignant(string _nom, string _prenom) ;
21 } ;
22
23 class Chercheur  : public Personne {
24 public :
25   Chercheur(string _nom, string _prenom) ;
26 } ;
```

8.2 Construction de la classe de base

Si une classe *B* hérite d'une classe *A*, alors, la construction d'un objet *B* implique la construction d'un objet *A* :

- si aucun constructeur particulier de *A* n'est spécifié dans le constructeur de *B*, le constructeur sans argument de *A* est appelé avant la construction de *B* ;
- il est possible de spécifier un constructeur particulier de la classe *A* de la même façon que pour la construction des données membres objets (voir chapitre 3.2.1.2).

Dans l'exemple suivant, pour un objet de la classe B, on dispose de deux constructeurs :

- le premier ne prend aucun argument, il exécute donc le constructeur sans argument de la classe A ;
- le second prend deux entiers en argument ; le premier de ces entiers est passé en argument au constructeur de A indiqué avant le bloc.

```
1  class A {
2  protected :
3    int i ;
4  public :
5    A()  : i(0) {}
6    A(inti)  : i(i) {}
7  } ;
8
9  class B  : public A {
10 protected :
11   int j ;
12 public :
13   B () {} // A() est □□excut puis le corps de B()
```

```
14    B(int i, int j)  : A(i), j(j) {} // A(int) est □□excut puis j(j) et le corps de B(int,int)
15  };
```

Concernant notre second exemple :

```
1   class Personne {
2   public :
3     Personne(string _nom, string _prenom)
4         : nom_(_nom), prenom_(_prenom) {}
5     // ...
6   };
7
8   class Etudiant  : public Personne {
9   public :
10    Etudiant(string _nom, string _prenom, string _numet)
11        : Personne(_nom, _prenom), numet_(_numet) {}
12    // ...
13  };
14
15  class Enseignant  : public Personne {
16  public :
17    Enseignant(string _nom, string _prenom)
18        : Personne(_nom, _prenom) {}
19  };
20
21  class Chercheur  : public Personne {
22  public :
23    Chercheur(string _nom, string _prenom)
24        : Personne(_nom, _prenom) {}
25  };
```

8.3 Destruction des objets

Simplement, l'exécution des destructeurs s'effectue dans l'ordre inverse de celui de la construction, soit d'abord celui de la classe dérivée puis celui de la classe de base.

8.4 Redéfinition des fonctions membres

Toute fonction public ou protégée d'une super-classe peut être redéfinie dans une sous-classe. Il s'agit d'une redéfinition, et non d'une surcharge car la signature de la fonction dans la sous-classe est la même que celle dans

la super-classe. Ceci permet d'affiner (de spécialiser) le comportement des objets.

Supposons que la hiérarchie de la super-classe *Personne* soit utilisée dans un carnet d'adresse. La classe *Personne* dispose d'une fonction membre *signature* qui retourne, sous la forme d'une *string*, les informations affichées dans l'interface utilisateur. Cette fonction peut être redéfinie dans chacune des sous-classes de *Personne* pour tenir compte de la spécialisation :

```
class Personne {
public :
  Personne(string _nom, string _prenom);
  string nom() const;
  string prenom() const;
  string signature() const {
    return nom_ + " " + prenom_;
  }
protected :
  string nom_;
  string prenom_;
};

class Etudiant  : public Personne {
public :
  Etudiant(string _nom, string _prenom, string _numet);
  string signature() const {
    return string("Etudiant ") + Personne ::signature() + " (no " + numet_ + ")";
  }
protected :
  string numet_;
};

class Enseignant  : public Personne {
public :
  Enseignant(string _nom, string _prenom);
  string signature() const {
    return string("Enseignant ") + Personne ::signature();
  }
};

class Chercheur  : public Personne {
public :
  Chercheur(string _nom, string _prenom);
  string signature() const {
    return string("Chercheur ") + Personne ::signature();
  }
};
```

On observe deux choses :

– dans la classe *Etudiant* (et dans chacune des sous-classes de *Personne*), la fonction *signature* retourne une chaîne de caractère plus précise, tenant compte du fait qu'il s'agit d'un étudiant,

– on note que chacune des redéfinitions dans les sous-classes réutilisent la fonction *signature* de la super-classe *Personne* en utilisant l'opérateur de portée (appel de Personne::signature()) ; ce point n'est pas obligatoire mais dénote d'une bonne utilisation de la redéfinition.

8.5 Compatibilité des types

Si *A* est la classe de base de la classe *B*, alors, on peut toujours utiliser un objet de la classe *B* à la place d'un objet de la classe *A*. L'inverse est impossible.

De même pour les pointeurs, on peut toujours utiliser un pointeur de type *B* * à la place d'un pointeur de type *A* *. L'inverse est possible avec un *cast* explicite mais il faut savoir ce qu'on fait dans ce cas.

```
1
2  class A {
3  };
4
5  class B  : public A {
6  };
7
8  main(int, char **)
9  {
10   A a;
11   B b;
12   a = b; // OK
13   b = a; // Erreur ☐la compilation
14
15   A * aptr = new A;
16   B * bptr = new B;
17   aptr = bptr; // OK
18   bptr = aptr; // Erreur ☐la compilation
19
20   bptr = (B*) aptr; // OK, mais il faut ☐tre ☐sr de ce qu'on fait
21   A * aptrOK = new B; // Allocation d'un B, rangement de l'adresse dans un pointeur sur
         A
22   bptr = (B*) aptrOK; // OK
23 }
```

8.6 Héritage et généricité

On vient de voir qu'on peut toujours utiliser une pointeur d'une sous-classe à la place d'un pointeur de sa super-classe. Cette propriété permet des développement génériques. Un exemple typique de code générique est le suivant :

Supposons qu'on ait une super-classe A et deux sous-classes B et C. On veut pouvoir stocker un tableau contenant soit des B, soit des C et même les deux mélangés.

Avec l'héritage c'est tout à fait possible en déclarant un tableau dont les éléments sont des pointeurs sur A (A*). Dans ce tableau, on peut tout a fait placer des pointeurs de B ou de C car ces pointeurs sont compatibles avec A*.

Dans le futur, si on introduit une nouvelle sous-classe de A+, par exemple la classe X, on pourra toujours exploiter notre tableau de A* pour y stocker des X*.

Prenons l'exemple concret de notre carnet d'adresses : un carnet comporte une collection pour stocker des instances de *Personne*. Dans cette collection, on peut stoker indifféremment des instances de *Etudiant*, *Chercheur* et de *Enseignant*.

Attention, on parle ici d'instances pour simplifier. **Il s'agira toujours de pointeur sur des objets**. En effet, on ne peut pas déclarer un tableau d'objets de la classe *Personne* et y stocker des instances de la classe *Etudiant*. On va déclarer un tableau de pointeurs sur des objets de la classe *Personne*.

Voici le code de cet exemple avec un programme de test :

```
1  #include <string>
2  #include <vector>
3  #include <iostream.h>
4
5  class Personne {
6  public :
7    Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
8    string nom() const { return nom_; }
9    string prenom() const { return prenom_; }
10   string signature() const {
11     return nom_ + " " + prenom_;
```

```
12    }
13  protected :
14    string nom_;
15    string prenom_;
16  };
17
18  class Etudiant  : public Personne {
19  public :
20    Etudiant(string _nom, string _prenom, string _numet)  : Personne(_nom, _prenom),
          numet_(_numet) {}
21    string signature() const {
22      return string("Etudiant ") + Personne ::signature() + " (no " + numet_ + ")";
23    }
24  protected :
25    string numet_;
26  };
27
28  class Enseignant  : public Personne {
29  public :
30    Enseignant(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
31    string signature() const {
32      return string("Enseignant ") + Personne ::signature();
33    }
34  };
35
36  class Chercheur  : public Personne {
37  public :
38    Chercheur(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
39    string signature() const {
40      return string("Chercheur ") + Personne ::signature();
41    }
42  };
43
44  class Carnet {
45  public :
46    Carnet() {}
47    void ajouterPersonne(Personne * p) {
48      personnes.push_back(p);
49    }
50    Personne * operator [](size_t _pos) {
51      return personnes[_pos];
52    }
53    size_t nbPersonnes() const {
54      return personnes.size();
55    }
56  protected :
57    vector<Personne*> personnes; // Tableau de pointeurs sur des Personne
58  };
59
60  main() {
61    Carnet c;
62    c.ajouterPersonne(new Etudiant("Bobo", "Larubule", "4356"));
63    c.ajouterPersonne(new Enseignant("Tarlite","Afrite"));
64    c.ajouterPersonne(new Chercheur("Gloubar","Nobar"));
65
```

```
66    for (size_t n = 0 ; n < c.nbPersonnes() ; n++) {
67      Personne * p = c[n] ;
68      cout << p->signature() << endl ;
69    }
70    exit(0) ;
71  }
```

Et voici le résultat de l'exécution sur la sortie standard :

```
1  > ./a.out
2  Bobo Larubule
3  Tarlite Afrite
4  Gloubar Nobar
5  >
```

On remarque d'abord que ca compile et que ca marche. C'est pas mal déjà. Par contre on a quelques petits soucis :

- dans la boucle d'affichage des *Personne*, on récupère des pointeurs *Personne* * ; et avec ce pointeur, on ne peut utiliser que les fonctions de la classe *Personne* ;
- toujours dans la boucle d'affichage, on exécute la fonction *signature* ; cette fonction est redéfinie dans toutes les sous-classes et on a stocké que des pointeurs d'objets des sous-classes de *Personne* ; or, c'est la version de la classe *Personne* qui est exécutée.

Concernant le premier point, si le pointeur récupéré est un *Etudiant** et si on veut utiliser une fonction membre de *Etudiant*, il faut *caster* le pointeur récupéré en *Etudiant**. On devrait faire quelque chose du style :

```
for (size_t n = 0; n < c.nbPersonnes(); n++) {
  Personne * p = c[n];
  cout << p->signature();
  if (p est un pointeur sur un Etudiant?) {
    Etudiant * e = (Etudiant*) p;
    cout << " Numéro d'étudiant: " << e->numen();
  }
  cout << endl;
}
```

Pour connaître le type du pointeur (pour répondre à «p est un pointeur sur un Etudiant ?»), on peut utiliser l'identification dynamique de type possible en C++ (voir chapitre 9).

Par contre, il faut bien être conscient qu'ainsi on diminue la généricité du code et sa maintenabilité. En effet, on le spécialise pour certains types. Mais on est parfois obligé d'utiliser une telle solution car la classe d'un objet n'est pas un objet en C++ (grOOSSe lacune !).

Pour le second point, effectivement c'est dommage, on se fatigue à redéfinir la fonction signature et finalement les redéfinitions ne sont pas exécutées. Lorsqu'on récupère un pointeur sur un *Etudiant* par exemple (c'est le cas du premier pointeur dans le carnet), on voudrait bien que ca exécute Etudiant::signature() et non pas Personne::signature(). Et bien pour ca il faut aller voir du coté des fonctions virtuelles (voir chapitre 8.8.1) car c'est possible mais il faut en savoir d'avantage.

8.7 L'héritage multiple

8.7.1 Mise en œuvre

L'héritage multiple signifie qu'une classe peut avoir plusieurs classes de base : la classe dérivée hérite ainsi des propriétés de toutes ses classes de base, des fonctions mais aussi des données membres :

```
1  class A {
2  public :
3    A() ;
4    int ai() const ;
5    void ai(int) ;
6  protected :
7    int ai ;
8  } ;
9
10 class B {
11 public :
12   B() ;
13   int bi() const ;
14   void bi(int) ;
15 protected :
16   int bi ;
17 } ;
18
19 // La classe C □hrite de A et de B
20 class C  : public A, public B {
21 public :
22   C() ;
23   int ci() const ;
24   void ci(int) ;
25   int abci() const
26   {
27     return ai + bi + ci ;
```

```
28    }
29  protected :
30    int ci;
31  };
32
33  main()
34  {
35    C c;
36    c.ai(2); // fonction membre ☐☐hrite de A
37    c.bi(2); // fonction membre ☐☐hrite de B
38    c.ci(2);
39  }
```

L'héritage multiple peut introduire des conflits de nom si un même nom de variable ou de fonction membre est utilisé dans une classe de base et dans une classe dérivée.

Il est évident qu'il vaut mieux éviter un telle situation. Mais si on ne peut pas l'éviter, alors les conflits de nommage sont réglés par une indication explicite du nom de la classe avec l'opérateur de portée ::. Voici le même exemple mais avec une variable i dans chacune des classes :

```
1   class A {
2   public :
3     A();
4     int i() const;
5     void i(int);
6   protected :
7     int i;
8   };
9
10  class B {
11  public :
12    B();
13    int i() const;
14    void i(int);
15  protected :
16    int i;
17  };
18
19  // La classe C ☐hrite de A et de B
20  class C  : public A, public B {
21  public :
22    C();
23    int i() const;
24    void i(int);
25
26    // Conflit de nom gere avec  ::
27    int abci() const
```

```
28    {
29      return C ::i + B ::i + C ::i;
30    }
31  protected :
32    int i;
33  };
34
35  main()
36  {
37    C c;
38    c.A ::i(2); // fonction membre ☐☐hrite de A
39    c.B ::i(2); // fonction membre ☐☐hrite de B
40    c.i(2); // fonction membre de C
41  }
```

8.7.2 Sémantique de l'héritage multiple

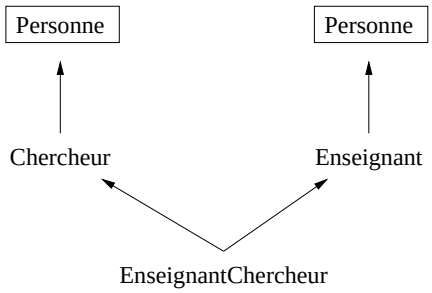
Voici à nouveau notre exemple des étudiants et des enseignants. On peut introduire la classe *EnseignantChercheur* qui hérite d'*Enseignant* et de *Chercheur* :

```
1   class Personne {
2   public :
3     Personne(string _nom, string _prenom);
4     // ...
5   };
6
7   class Etudiant  : public Personne {
8   public :
9     Etudiant(string _nom, string _prenom, string _numet);
10    // ...
11  };
12
13  class Enseignant  : public Personne {
14  public :
15    Enseignant(string _nom, string _prenom);
16  };
17
18  class Chercheur  : public Personne {
19  public :
20    Chercheur(string _nom, string _prenom);
21  };
22
23  class EnseignantChercheur  : public Enseignant, public Chercheur {
24    public :
25    EnseignantChercheur(string _nom, string _prenom)
26        : Enseignant(_nom, _prenom), Chercheur(_nom, _prenom) {}
27  };
```

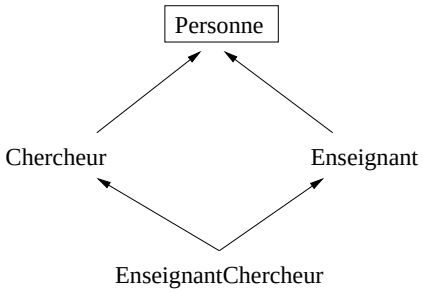
Bon d'accord mais il se pose un problème d'interprétation du graphe d'héritage parce qu'il y a deux chemins pour accéder aux membres de la classe *Personne* depuis la classe *EnseignantChercheur* :

- par *Enseignant*, *EnseignantChercheur ::Enseignant ::Personne*,
- par *Chercheur*, *EnseignantChercheur ::Chercheur ::Personne*.

On a deux interprétations possibles de cette hiérarchie :



Interprétation 1



Interprétation 2

Le problème qui se pose au compilateur est : dois-je interpréter l'héritage (des classes dérivées vers les classes de base) comme la définition d'un arbre (interprétation 1) ou d'un graphe (interprétation 2) ?

Le problème est important car :

- si on a un arbre, alors cela signifie que pour un objet de la classe *EnseignantChercheur*, on a deux *Personnes* (on a deux chemins différents pour accéder à la classe *Personne* depuis *EnseignantChercheur*).
- si on a un graphe, alors cela signifie que pour un objet de la classe *EnseignantChercheur*, on a qu'un seul *Personne*.

En fait, C++ prévoit les deux. Le cas par défaut (appelé ici *héritage multiple simple*) est celui de la première interprétation. Pour la seconde interprétation, on parle *d'héritage virtuel*.

8.7.2.1 L'héritage multiple simple

On a donc deux objets *Personne* pour un objet *EnseignantChercheur*, les données membres *nom_* et *prenom_*sont en double exemplaire :

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

class Personne {
public :
  Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
  void nom(string s) { nom_ = s ; }
  string nom() const { return nom_ ; }
private :
  string nom_, prenom_ ;
} ;

class Enseignant  : public Personne {
public :
  Enseignant(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
} ;

class Chercheur  : public Personne {
public :
  Chercheur(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
} ;

class EnseignantChercheur  : public Enseignant, public Chercheur {
  public :
  EnseignantChercheur(string _nom, string _prenom)
     : Enseignant(_nom, _prenom), Chercheur(_nom, _prenom) {}
} ;

main() {
  EnseignantChercheur ec("Fricotin","Bibi") ;
  cerr << ec.Enseignant ::nom() << " " << ec.Chercheur ::nom() << endl ;
  ec.Chercheur ::nom("toto") ;
  cerr << ec.Enseignant ::nom() << " " << ec.Chercheur ::nom() << endl ;
}
```

Voici le résultat de l'éxécution du programme :

```
> a.out
Fricotin Fricotin
Fricotin toto
>
```

On observe bien qu'après la modification du nom par un des deux chemins possibles, il y a deux nom différents :

- par le chemin *EnseignantChercheur ::Chercheur ::Personne*, le nom est «toto»,

– et par le chemin *EnseignantChercheur ::Enseignant ::Personne*, le nom est «Fricotin».

Dans certain cas, cette interprétation peut tout à fait être la bonne. Par exemple, si on veut prévoir le cas d'une personne (une dame) qui utilise son nom de jeune fille quand elle intervient en tant que chercheuse dans les conférences et qui utilise son nom marital autrement. On dispose ainsi des nom-prénom de la personne enseignante et des nom-prénom de la personne chercheuse.

Par contre, pour beaucoup de cas, cette interprétation est génante. Il y a en effet duplication des données au niveau de la ou des super-classes. En conséquence, on a une consommation de mémoire inutile et des difficultés suplémentaires non négligeables d'écriture et de maintenance du code.

8.7.2.2 L'héritage virtuel

L'héritage virtuel permet de représenter l'héritage (voir le début du chapitre 8.7.2) comme un graphe. Il n'y a donc pas duplication de la ou des classes de base communes. Cette forme d'héritage se spécifie avec le mot clé *virtual* :

```
1  #include <iostream.h>
2  #include <string>
3
4  class Personne {
5  public :
6    Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
7    void nom(string s) { nom_ = s ; }
8    string nom() const { return nom_ ; }
9  private :
10   string nom_, prenom_ ;
11 } ;
12
13 // Dans cette ☐hirarchie, la classe Personne est virtuelle
14 class Enseignant  : virtual public Personne {
15 public :
16   Enseignant(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
17 } ;
18
19 // Dans cette ☐hirarchie, la classe Personne est virtuelle (ici aussi, forcement)
20 class Chercheur  : virtual public Personne {
21 public :
22   Chercheur(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
23 } ;
24
25 class EnseignantChercheur  : public Enseignant, public Chercheur {
26   public :
27   EnseignantChercheur(string _nom, string _prenom)
```

```
28        : Personne(_nom, _prenom), Enseignant(_nom, _prenom), Chercheur(_nom,
              _prenom) {} // 3 initialisations pour Personne !!!
29  };
30
31  main() {
32    EnseignantChercheur ec("Fricotin","Bibi");
33    cerr << ec.Enseignant ::nom() << " " << ec.Chercheur ::nom() << endl;
34    ec.Chercheur ::nom("toto");
35    cerr << ec.Enseignant ::nom() << " " << ec.Chercheur ::nom() << endl;
36  }
```

Voici le résultat de l'éxécution du programme :

```
1  > a.out
2  Fricotin Fricotin
3  toto toto
4  >
```

On observe bien qu'il n'y a plus duplication des variables membres de la classes *Personne*. Après la modification du nom par un des deux chemins possibles, on obtient le meme nom par le second chemin :

- par le chemin *EnseignantChercheur ::Chercheur ::Personne*, le nom est «toto»,
- et par le chemin *EnseignantChercheur ::Enseignant ::Personne*, le nom est encore «toto».

Bon, ca paraît joli comme ca, mais il y a des restrictions :

- il est impossible de *caster* directement un pointeur sur un objet d'une classe de base virtuelle en un pointeur sur un objet d'une de ses classes dérivées ; il faut impérativement utiliser l'opérateur de transtypage dynamique *dynamic_cast* (pour les kamikazes, voir le chapitre 9.1).
- chaque classe dérivée directement ou indirectement d'une classe virtuelle doit en appeler le constructeur explicitement dans son constructeur, si elle ne désire pas que le constructeur par défaut soit appelé ; on peut observer cela dans notre exemple, chacun constructeur des classes *Chercheur*, *Enseignant* et *EnseignantChercheur* exécute explicitement le constructeur de *Personne* Personne::Personne(string,string) alors que jusqu'à présent, seules les sous-classes immédiates, *Enseignant* et *Chercheur*, doivent exécuter ce constructeur.

8.7.3 Comment éviter l'héritage multiple

Ce type d'héritage est souvent source de problèmes de mise en oeuvre et surtout de maintenance. Il est déconseillé.

On peut souvent l'éviter : demandez vous toujours si on ne peut pas utiliser la composition à la place de l'héritage. Dans beaucoup de cas, cette technique permet d'éviter l'introduction de l'héritage multiple dans une hiérarchie de classes.

8.8 Polymorphisme, lien dynamique et fonctions virtuelles

Nous avons vu dans le chapitre 8.6 qu'on peut déclarer des pointeurs d'une super-classe et leur affecter des adresses d'objets de sous-classes. On a ainsi la possibilité de développer des classes génériques.

Par contre, seule les fonctions membre de la super-classe peuvent être exécutées. La raison est claire, on déclare un pointeur de *Personne* et c'est bien la fonction *Personne ::signature()* qui est exécutée car les liens sont déterminés à la compilation.

En effet, le comportement par défaut pour l'exécution d'une fonction membre via un pointeur est que c'est le type du pointeur qui détermine la fonction à exécuter. Et ceci même si la classe de l'objet pointé est une sous-classe. C'est ce qu'on appele le lien statique.

Obtenir que lorsqu'un pointeur de *Personne* contient l'adresse d'un objet *Etudiant* fasse qu'on exécute la fonction membre de *Etudiant* et non celle de *Personne* est possible en C++. Il s'agit de la mise en œuvre du polymorphisme et de l'édition de lien dynamique :

- le polymorphisme parce qu'une personne peut être soit un étudiant soit un chercheur soit un enseignant et que chacun possède sa propre version de la fonction *signature* ;
- édition de lien dynamique car le programme doit tenir compte du type de l'objet pointé pendant l'exécution plutôt que du type déclaré du pointeur.

8.8.1 les fonctions membres virtuelles

Pour faire en sorte que ce soit, pendant l'exécution, le type de l'objet pointé qui est pris en compte pour le choix de la fonction à exécuter, il faut utiliser le lien dynamique : on déclare la fonction virtuelle (mot clé *virtual*).

Une fonction est virtuelle si sa déclaration dans l'interface de la classe est précédé du mot clé *virtual*.

Un premier exemple :

```
1  #include <iostream.h>
2
3  class A {
4  protected :
5    int i ;
6  public :
7    A()  : i(0) {}
8    A(int _i)  : i(_i) {}
9    int computeStatic() const { return i ; } // Lien statique
10   virtual int computeDyn() const { return 1 ; } // fonction virtuelle, lien dynamique
11 } ;
12
13 class B  : public A {
14   int j ;
15 public :
16   B () {}
17   B(int i, int j)  : A(i), j(i) { }
18   int computeStatic() const { return i* 10 ; } // □Redfinition, Lien statique
19   virtual int computeDyn() const { return i* 10 ; } // □Redfinition, lien dynamique
20 } ;
21
22 int cStatic (const A * aptr)
23 {
24   return aptr->computeStatic() ;
25 }
26
27 int cDyn (const A * aptr)
28 {
29   return aptr->computeDyn() ;
30 }
31
32 main(int, char **)
33 {
34   B b(34, 0) ;
35   cout << cStatic(&b) ;// Affiche 34, lien statique -> □excution de int A ::
         computeStatic0 const
36   cout << cDyn(&b) ; // Affiche 340, lien dynamique -> □excution de int B ::
         computeDyn0 const ;
37 }
```

La classe *B* hérite publiquement de *A*. Ces deux classes définissent les fonctions membres *computeStatic()* et *computeDyn()*. Les fonctions *cStatic()* et *cDyn()* prennent en argument, un objet de type pointeur sur *A*.

Si dans le programme, on exécute la fonction *cStatic()*, quelque-soit le type (*A* ou *B*) de l'argument effectivement passé, ce sera toujours la fonction membre de la classe *A*, *int A ::computeStatic() const*, qui sera exécutée. En effet, les liens étant décidés à la compilation, c'est la fonction membre de la classe *A* qui est exécutée dans la fonction *cStatic()*.

Observons maintenant le cas de la fonction membre *computeDyn()* et de la fonction *cDyn()*. Le contexte est exactement le même que précédemment sauf que la fonction membre est déclarée *virtual* dans la classe *A*. Le résultat de l'exécution de la fonction *cDyn()* est alors différent. C'est bien la fonction membre de la classe *B* qui est exécutée (classe de l'argument effectivement passé) et non celle de la classe *A* (classe du paramètre déclaré pour la fonction *cDyn()*). Le choix de la méthode à exécuter est bien dynamique, c'est à dire qu'il s'effectue pendant l'exécution du programme.

Pour enfoncer le clou à la masse, voici l'exemple du chapitre 8.6 revu et ammélioré à l'aide de fonctions virtuelles :

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

class Personne {
public :
  Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
  string nom() const { return nom_ ; }
  string prenom() const { return prenom_ ; }
  virtual string signature() const {
    return nom_ + " " + prenom_ ;
  }
protected :
  string nom_ ;
  string prenom_ ;
} ;

class Etudiant  : public Personne {
public :
  Etudiant(string _nom, string _prenom, string _numet)  : Personne(_nom, _prenom),
      numet_(_numet) {}
  virtual string signature() const {
    return string("Etudiant ") + Personne ::signature() + " (no " + numet_ + ")";
  }
protected :
  string numet_ ;
} ;

class Enseignant  : public Personne {
public :
  Enseignant(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
  virtual string signature() const {
```

```
32      return string("Enseignant ") + Personne ::signature();
33    }
34  };
35
36  class Chercheur  : public Personne {
37  public :
38    Chercheur(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
39    virtual string signature() const {
40      return string("Chercheur ") + Personne ::signature();
41    }
42  };
43
44  class Carnet {
45  public :
46    Carnet() {}
47    void ajouterPersonne(Personne * p) {
48      personnes.push_back(p);
49    }
50    Personne * operator [](size_t _pos) {
51      return personnes[_pos];
52    }
53    size_t nbPersonnes() const {
54      return personnes.size();
55    }
56  protected :
57    vector<Personne*> personnes; // Tableau de pointeurs sur des Personne
58  };
59
60  main() {
61    Carnet c;
62    c.ajouterPersonne(new Etudiant("Bobo", "Larubule", "4356"));
63    c.ajouterPersonne(new Enseignant("Tarlite","Afrite"));
64    c.ajouterPersonne(new Chercheur("Gloubar","Nobar"));
65
66    for (size_t n = 0; n < c.nbPersonnes(); n++) {
67      Personne * p = c[n];
68      cout << p->signature() << endl;
69    }
70    exit(0);
71  }
```

La fonction *signature()* est déclarée virtuelle dans toutes les classes. A l'exécution, comme le montre la trace données ci-après, c'est bien la fonction menbre des objets pointés qui est exécutée, et non toujours celle de *Personne*.

```
1  > ./a.out
2  Etudiant Bobo Larubule (no 4356)
3  Enseignant Tarlite Afrite
4  Chercheur Gloubar Nobar
5  >
```

8.8.2 Les classes abstraites

Dans les langages à objet, une classe abstraite est une classe générique qui ne peut pas être instanciée. On doit forcément passer par l'héritage pour instancier des sous-classes.

Pour spécifier une classe abstraite en C++, on utilise la possibilité de ne pas donner le corps d'une fonction membre virtuelle en la déclarant virtuelle pure. Pour cela, on affecte la valeur 0 à sa déclaration dans l'interface de la classe.

Dans l'exemple suivant, on désire manipuler des objets graphiques au travers d'une collection. Tous les objets graphiques héritent d'une même classe (abstraite) *Obj* dont la spécification est la suivante :

```
class Obj {
 public :
   virtual const char * est_quoi() const = 0;
   virtual ostream & write (ostream &) const = 0;
};
```

Etant donné que la classe *Obj* est abstraite, on ne peut pas déclarer de variable dont le type est *Obj* ou de tableau dont le type des éléments est *Obj*. On peut par contre déclarer un pointeur sur une *Obj* ou un tableau dont le type des éléments est *Obj**.

Cette classe représente un objet graphique dont on ne connaît à priori pas grand chose. On ne sait donc pas mettre en œuvre les fonctions membres *est_quoi* et *write*. Ces deux fonctions sont déclarées virtuelles pures.

Un rond se caractérise par les positions x et y de son centre et par son rayon, un carré se spécifie par les positions x et y de son coin haut à gauche et par sa largeur.

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

class Obj {
public :
   Obj() {}
   virtual const char * est_quoi() const = 0;
   virtual ostream & write (ostream &) const = 0;
};

```

```
11 class CarreObj  : public Obj
12 {
13 public :
14   CarreObj()  : x_(0), y_(0), largeur_(0) {}
15   CarreObj(int _x, int _y, size_t _largeur)  : x_(_x), y_(_y), largeur_(_largeur) {}
16   virtual const char * est_quoi() const { return "carre"; }
17   virtual ostream & write (ostream & _out) const
18   { return _out << x_ << " " << y_ << " " << largeur_ << " "; }
19 protected :
20   int x_;
21   int y_;
22   size_t largeur_;
23 };
24
25 class RondObj  : public Obj
26 {
27 public :
28   RondObj()  : x_(0), y_(0), rayon_(0) {}
29   RondObj(int _x, int _y, size_t _rayon)  : x_(_x), y_(_y), rayon_(_rayon) {}
30   virtual const char * est_quoi() const { return "rond"; }
31   virtual ostream & write (ostream & _out) const
32   { return _out << x_ << " " << y_ << " " << rayon_ << " "; }
33 protected :
34   int x_;
35   int y_;
36   size_t rayon_;
37 };
38
39 main ()
40 {
41   vector<Obj*> graphique;
42   graphique.push_back(new CarreObj(5,6,7));
43   graphique.push_back(new RondObj(7,9,1));
44   graphique.push_back(new CarreObj(9,6,0));
45
46   for (size_t n = 0; n < graphique.size(); n++) {
47     cout << graphique[n]->est_quoi() << " ";
48     graphique[n]->write(cout);
49     cout << endl;
50   }
51   exit(0);
52   return 0;
53 }
```

Voici la trace d'exécution du programme :

```
1 > a.out
2 carre 5 6 7
3 rond 7 9 1
4 carre 9 6 0
5 >
```

8.8.2.1 Destructeur virtuel

Quand on utilise des hiérarchies de classes, pour garantir que c'est le destructeur de l'objet réellement pointé et non seulement celui de la super-classe qui est exécuté, il est possible de déclarer le destructeur comme une fonction virtuelle. Par exemple dans la classe *Obj* :

```
1 class Obj {
2  public :
3    virtual ~Obj() ;
4    virtual const char * est_quoi() const = 0 ;
5    virtual ostream & write (ostream &) const = 0 ;
6 } ;
```

Chapitre 9

L'identification dynamique des types

Avec C++, il est possible de déterminer le type des objets et des expressions de deux façons : par le *cast* dynamique (*dynamic cast*, ca fait mieux avec des amis) et l'opérateur *typeid*.

9.1 Le *cast* dynamique

Il peut être utilisé pour convertir un pointeur ou une référence d'une classe de base vers un pointeur ou une référence d'une sous-classe. Le *cast* dynamique nécessite que la classe de base de l'objet pointé comporte au moins une fonction virtuelle. Il est intéressant dans la mesure ou la conversion n'est pas effectuée si le *cast* est invalide.

Voici un exemple d'utilisation avec des pointeurs :

```
1  #include <iostream.h>
2
3  class A
4  {
5  public :
6    virtual ~A() {}
7  };
8
9  class B : public A
```

```
{
public :
  char const * toString() {
    return ("B object");
  }
};

int main()
{
  B *b1;

  B b2;
  A* a = &b2;

  // Tentative de cast
  if ((b1 = dynamic_cast<B*>(a))) {
    // OK ca marche, les pointeurs sont compatibles
    cout << b1->toString() << endl;
  } else {
    // KO
    cout << "Le cast dynamique a echoue\n";
  }
  return (0);
}
```

9.2 L'opérateur *typeid*

9.2.1 Utilisation de *typeid*

Le C++ fournit l'opérateur *typeid* afin de récupérer les informations de type des expressions. La syntaxe est :

typeid(expression)

où expression est l'expression dont il faut déterminer le type.

Le résultat de l'opérateur *typeid* est une référence sur un objet constant de la classe *type_info*. Voici un exemple d'utilisation avec notre super exemple du carnet introduit dans le chapitre 8.6.

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

class Personne {
public :
  Personne(string _nom, string _prenom)  : nom_(_nom), prenom_(_prenom) {}
```

```
9     virtual ~Personne() {}
10    string nom() const { return nom_; }
11    string prenom() const { return prenom_; }
12    string signature() const {
13      return nom_ + " " + prenom_;
14    }
15  protected :
16    string nom_;
17    string prenom_;
18  };
19
20  class Etudiant  : public Personne {
21  public :
22    Etudiant(string _nom, string _prenom, string _numet)  : Personne(_nom, _prenom),
          numet_(_numet) {}
23    string signature() const {
24      return string("Etudiant ") + Personne ::signature() + " (no " + numet_ + ")";
25    }
26  protected :
27    string numet_;
28  };
29
30  class Enseignant  : public Personne {
31  public :
32    Enseignant(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
33    string signature() const {
34      return string("Enseignant ") + Personne ::signature();
35    }
36  };
37
38  class Chercheur  : public Personne {
39  public :
40    Chercheur(string _nom, string _prenom)  : Personne(_nom, _prenom) {}
41    string signature() const {
42      return string("Chercheur ") + Personne ::signature();
43    }
44  };
45
46  class Carnet {
47  public :
48    Carnet() {}
49    void ajouterPersonne(Personne * p) {
50      personnes.push_back(p);
51    }
52    Personne * operator [](size_t _pos) {
53      return personnes[_pos];
54    }
55    size_t nbPersonnes() const {
56      return personnes.size();
57    }
58  protected :
59    vector<Personne*> personnes; // Tableau de pointeurs sur des Personne
60  };
61
62  main() {
```

```
63    Carnet c;
64    c.ajouterPersonne(new Etudiant("Bobo", "Larubule", "4356"));
65    c.ajouterPersonne(new Enseignant("Tarlite","Afrite"));
66    c.ajouterPersonne(new Chercheur("Gloubar","Nobar"));
67
68    for (size_t n = 0; n < c.nbPersonnes(); n++) {
69       Personne * p = c[n];
70       if (typeid(*p) == typeid(Etudiant)) {
71          Etudiant * e = (Etudiant*) p;
72          cout << e->signature() << endl;
73       } else if (typeid(*p) == typeid(Enseignant)) {
74          Enseignant * e = (Enseignant*) p;
75          cout << e->signature() << endl;
76       } else if (typeid(*p) == typeid(Chercheur)) {
77          Chercheur * c = (Chercheur*) p;
78          cout << c->signature() << endl;
79       } else {
80          throw "Erreur bizarre!";
81       }
82    }
83    exit(0);
84 }
```

Voici la trace d'exécution :

```
1  > a.out
2  Etudiant Bobo Larubule (no 4356)
3  Enseignant Tarlite Afrite
4  Chercheur Gloubar Nobar
5  >
```

Bien que la fonction membre *signature* ne soit pas virtuelle, on obtient cependant le même résultat qu'avec *signature* virtuelle (voir exemple du chapitre 8.8.1). Cependant, *yapasphoto*. Il est largement préférable d'utiliser la version du chapitre 8.8.1 avec des fonctions virtuelles.

Le code présenté ici est peu maintenable car il fait l'hypothèse qu'il n'y a que trois sous-classes à *Personne*. De plus, c'est beaucoup plus lourd, car on doit tester tous les cas.

On note de plus qu'il faut qu'au moins une fonction membre de *Personne* (la classe de base) soit virtuelle. C'est le cas ici, le destructeur de *Personne* est virtuel.

9.2.2 La classe *typeinfo*

Les informations de type sont enregistrées dans des objets de la classe *type_info*, prédéfinie. Cette classe est déclarée dans l'en-tête *typeinfo* :

```
1
2  class type_info
3  {
4  public :
5       virtual ~type_info();
6       bool operator==(const type_info &rhs) const;
7       bool operator!=(const type_info &rhs) const;
8       bool before(const type_info &rhs) const;
9       const char *name() const;
10 private :
11      type_info(const type_info &rhs);
12      type_info &operator=(const type_info &rhs);
13 };
```

Les objets de la classe *type_info* ne peuvent pas être copiés, puisque l'opérateur d'affectation et le constructeur de copie sont tous les deux déclarés *private*. Par conséquent, le seul moyen de générer un objet de la classe *type_info* est d'utiliser l'opérateur typeid.

Les opérateurs de comparaison permettent de tester l'égalité et la différence de deux objets *type_info*.

Les objets *type_info* contiennent des informations sur les types sous la forme de chaînes de caractères (on a uniquement une description minimale de la classe). Le format de la chaîne de caractères est dépendante du compilateur et du système.