Le Langage C++

Version 2.0

Bases : Alain Dancel
Bases : Christian Casteyde
Modifications : Philippe Colantoni

Sommaire

1 Introduction

Ce poly a été conçu à partir d'informations provenant de plusieurs sources différentes (livre + web). Pour approfondir le sujet, je vous conseille la lecture d'un ouvrage consacré à C++, et je recommande particulièrement les deux livres suivants :

- Bjarne Stroustrup. The C++ programming Language, 3rd Edition. Addison-Wesley, 1997. La référence de base, par l'auteur du langage.
- Stanley B. Lippman, Josée Lajoie. C++ Primer, 3rd Edition. Addison-Wesley, 1998.

Il était une fois le C++

Le langage C++ a deux grands ancêtres :

- Simula, dont la première version a été conçue en 1967. C'est le premier langage qui introduit les principaux concepts de la programmation objet. Probablement parce qu'il était en avance sur son temps, il n'a pas connu à l'époque le succès qu'il aurait mérité, mais il a eu cependant une influence considérable sur l'évolution de la programmation objet.
  Développé par une équipe de chercheurs norvégiens, Simula-67 est le successeur de Simula I, lui-même inspiré d'Algol 60. Conçu d'abord à des fins de modélisation de systèmes physiques, en recherche nucléaire notamment, Simula I est devenu un langage spécialisé pour traiter des problèmes de simulation. Ses concepteurs faisaient aussi partie du groupe de travail Ifip1 qui poursuivait les travaux ayant donné naissance à Algol 60. Simula-67 est avec Pascal et Algol 68 un des trois langages issus des différentes voies explorées au sein de ce groupe. Son nom fut changé en Simula en 1986.
  Comme son prédécesseur Simula I, Simula permet de traiter les problèmes de simulation. En particulier, un objet est considéré comme un programme actif autonome, pouvant communiquer et se synchroniser avec d'autres objets. C'est aussi un langage de programmation général, reprenant les constructions de la programmation modulaire introduites par Algol 60. Il y ajoute les notions de classe, d'héritage et autorise le masquage des méthodes, ce qui en fait un véritable langage à objets.
- Le langage C a été conçu en 1972 aux laboratoires Bell Labs. C'est un langage structuré et modulaire, dans la philosophie générale de la famille Algol. Mais c'est aussi un langage proche du système, qui a notamment permis l'écriture et le portage du système Unix. Par conséquent, la programmation orientée système s'effectue de manière particulièrement aisée en C, et on peut en particulier accéder directement aux fonctionnalités du noyau Unix.
  C possède un jeu très riche d'opérateurs, ce qui permet l'accès à la quasi-totalité des ressources de la machine. On peut par exemple faire de l'adressage indirect ou utiliser des opérateurs d'incrémentation ou de décalage. On peut aussi préciser qu'on souhaite implanter une variable dans un registre. En conséquence, on peut écrire des programmes presque aussi efficaces qu'en langage d'assemblage, tout en programmant de manière structurée.

Le concepteur de C++, Bjarne Stroustrup, qui travaillait également aux Bell Labs, désirait ajouter au langage C les classes de Simula. Après plusieurs versions préliminaires, le langage a trouvé une première forme stable en 1983, et a très rapidement connu un vif succès dans le monde industriel. Mais ce n'est qu'assez récemment que le langage a trouvé sa forme définitive, confirmée par une norme (C++ ISO 14882).

Pourquoi utiliser le C++ ?

Le C++ est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Il souffre cependant de la réputation d'être compliqué et illisible. Cette réputation est en partie justifiée. La complexité du langage est inévitable lorsqu'on cherche à avoir beaucoup de fonctionnalités. En revanche, en ce qui concerne la lisibilité des programmes, tout dépend de la bonne volonté du programmeur.

Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, ce langage est, avec le C, idéal pour ceux qui doivent assurer la portabilité de leurs programmes au niveau des fichiers sources (pas des exécutables).

Les principaux avantages du C++ sont les suivants :

- grand nombre de fonctionnalités ;
- performances du C ;
- facilité d'utilisation des langages objets ;
- portabilité des fichiers sources ;
- facilité de conversion des programmes C en C++, et, en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C ;
- contrôle d'erreurs accru.

On dispose donc de quasiment tout : puissance, fonctionnalité, portabilité et sûreté. La richesse du contrôle d'erreurs du langage, basé sur un typage très fort, permet de signaler un grand nombre d'erreurs à la compilation. Toutes ces erreurs sont autant d'erreurs que le programme ne fait pas à l'exécution. Le C++ peut donc être considéré comme un super C Ż. Le revers de la médaille est que les programmes C ne se compilent pas directement en C++ : il est courant que de simples avertissements en C soient des erreurs blocantes en C++. Quelques adaptations sont donc souvent nécessaires, cependant, celles-ci sont minimes, puisque la syntaxe du C++ est basée sur celle du C. On remarquera que tous les programmes C peuvent être corrigés pour compiler à la fois en C et en C++.

D'autres langages, et en particulier Java, se sont fortement inspirés de la syntaxe de C++. Celle-ci est de ce fait devenue une référence. Nous supposons en particulier que les élèves qui ont déjà appris Java ne seront pas dépaysés par ce langage. Cependant, nous voulons mettre en garde contre plusieurs fausses ressemblances : si la syntaxe est la même ou très proche, plusieurs concepts sous-jacents sont différents.

2 Un meilleur C

Sommaire

2.1 Les commentaires

Le langage C++ offre un nouvelle façon d'ajouter des commentaires. En plus des symboles /* et */ utilisés en C, le langage C++ offre les symboles // qui permettent d'ignorer tout jusqu'à la fin de la ligne.

Exemple :

```
/* commentaire traditionnel
   sur plusieurs lignes
   valide en C et C++
*/

int main ( void ) { // commentaire de fin de ligne valide en C++
#if 0
     // une partie d'un programme en C ou C++ peut toujours
     // être ignorée par les directives au préprocesseur
     // #if  ....   #endif
#endif

     return 0;
}
```

Note : Il est préférable d'utiliser les symboles // pour la plupart des commentaires et de n'utiliser les commentaires C (/* */) que pour isoler des blocs importants d'instructions.

2.2 Les types

En C++, les types de base sont :

- bool : booléen , peut valoir true ou false,
- char : caractère (en général 8 bits), qui peuvent aussi être déclarés explicitement signés (signed char) ou non signés (unsigned char),
- int : entier (16 ou 32 bits, suivant les machines), qui possède les variantes short [int] et long [int], tous trois pouvant également être declarés non signés (unsigned),
- float : réel (1 mot machine),
- double : réel en double précision (2 mots machines), et sa variante long double (3 ou 4 mots machine),
- void qui spécifie un ensemble vide de valeurs.

Les constantes caractères s'écrivent entre quotes simples :

```
'a'  'G'  '3'  '*' '['
```

Certains caractères de contrôle s'écrivent par des séquences prédéfinies ou par leur code octal ou hexadécimal, comme par exemple :

```
\n  \t  \r  \135  \'  \x0FF
```

Les constantes entières peuvent s'écrire en notations décimale, hexadécimale (précédées de 0x3) ou octale (précédées de 04). Pour forcer la constante à être de type entier long, il faut ajouter un L à la fin, de même le suffixe u indique une constante non signée :

```
12  -43  85  18642  54L 255u 38ul
0xabfb  0x25D3a  0x3a
0321  07215  01526
```

Les constantes réelles s'écrivent avec point décimal et éventuellement en notation exponentielle :

```
532.652  -286.34  12.73
52e+4  42.63E-12  -28.15e4
```

Les constantes de type chaînes de caractères (voir plus loin) s'écrivent entre double-quotes :

```
"Home sweet home"
"Français, je vous ai compris."
```

2.3 Entrées/sorties avec *cin*, *cout* et *cerr*

Les entrées/sorties en langage C s'effectue par les fonctions scanf et printf de la librairie standard du langage C.

Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes, mais cependant les programmeurs C++ préfèrent les entrées/sorties par *flux* (ou *flot* ou *stream*).

Quatre flots sont prédéfinis lorsque vous avez inclus le fichier d'en-tête iostream :

- cout qui correspond à la sortie standard
- cin qui correspond à l'entrée standard
- cerr qui correspond à la sortie standard d'erreur non tamponné
- clog qui correspond à la sortie standard d'erreur tamponné.

L'opérateur (surchargé) << permet d'envoyer des valeurs dans un *flot de sortie*, tandis que >> permet d'extraire des valeurs d'un *flot d'entrée*.

Exemple :

```
#include <iostream>

using namespace std;

int main() {
  int i=123;
  float f=1234.567;
  char ch[80]="Bonjour\n", rep;

  cout << "i=" << i << "  f=" << f << "  ch=" << ch;
  cout << "i = ? ";
  cin >> i;          // lecture d'un entier
  cout << "f = ? ";
  cin >> f;          // lecture d'un réel
  cout << "rep = ? ";
  cin >> rep;        // lecture d'un caractère
  cout << "ch = ? ";
  cin >> ch;         // lecture du premier mot d'une chaîne
  cout << "ch= " << ch;  // c'est bien le premier mot ...

  return 0;
}
/*-- résultat de l'exécution -------------------
i=123  f=1234.57  ch=Bonjour
i = ? 12
f = ? 34.5
rep = ? y
ch = ? c++ is easy
ch= c++
-------------------------------------------------*/
```

Notes :

- tout comme pour la fonction *scanf*, les espaces sont considérés comme des séparateurs entre les données par le flux *cin*.
- notez l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

2.4 Intéret de *cin*, *cout* et *cerr*

- Vitesse d'exécution plus rapide : la fonction *printf* doit analyser à l'exécution la chaîne de formatage, tandis qu'avec les flots, la traduction est faite à la compilation.
- Vérification de type : pas d'affichage erroné

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

using namespace std;

int main ( void )
{
```

```
  int i=1234;
  double d=567.89;
  printf("i= %d   d= %d !!!!!\n", i, d);
  //                  ^erreur: %lf normalement
  cout << "i= " << i << "   d= " << d << "\n";

  return 0;
}
/* Résultat de l'exécution ************
   i= 1234   d= -5243 !!!!!!!
   i= 1234   d= 567.89
*******************************/
```

- Taille mémoire réduite : seul le code nécessaire est mis par le compilateur, alors que pour, par exemple printf , tout le code correspondant à toutes les possibilités d'affichage est mis.
- On peut utiliser les flux avec les types utilisateurs (surcharge possible des opérateurs `>>` et `<<`.

2.5 Les manipulateurs

Les manipulateurs sont des éléments qui modifient la façon dont les éléments sont lus ou écrits dans le flot.

Les principaux manipulateurs sont :

dec	lecture/écriture d'un entier en décimal
oct	lecture/écriture d'un entier en octal
hex	lecture/écriture d'un entier en hexadécimal
endl	insère un saut de ligne et vide les tampons
setw(int n)	affichage de n caractères
setprecision(int n)	affichage avec n chiffres avec éventuellement un arrondi
setfill(char)	définit le caractère de remplissage
flush	vide les tampons après écriture

Exemple :

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
  int i=1234;
  float p=12.3456;
  cout << "|" << setw(8) << setfill('*')
       << hex << i << "|\n" << "|"
       << setw(6) << setprecision(4)
       << p << "|" << endl;

  return 0;
}
/*-- résultat de l'exécution --------------------
|*****4d2|
|*12.35|
-----------------------------------------------*/
```

2.6 Les conversions explicites

- En C++, comme en langage C, il est possible de faire des conversions explicites de type, bien que le langage soit plus fortement typé :

```
double d;
int i;

i = (int) d;
```

- Le C++ offre aussi une notation fonctionnelle pour faire une conversion explicite de type :

```
double d;
int i;

i = int(d);
```

- Cette façon de faire ne marche que pour les types simples et les types utilisateurs.
- Pour les types pointeurs ou tableaux le problème peut être résolu en définissant un nouveau type :

```
double d;
int *i;

typedef int *ptr_int;

i = ptr_int(&d);
```

- La conversion explicite de type est surtout utile lorsqu'on travaille avec des pointeurs du type *void **.

2.7 Définition de variables

- En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code.
- La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Ceci permet :
  - de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité. C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.
  - d'initialiser un objet avec une valeur obtenue par calcul ou saisie.

Exemple :

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

using namespace std;

int main() {
  int i=0;      // définition d'une variable
  i++;          // instruction
  int j=i;      // définition d'une autre variable
  j++;          // instruction
  int somme(int n1, int n2); // déclaration d'une fonction
  printf("%d+%d=%d\n", i, j, somme(i, j)); // instruction
```

```
  cin >> i;
  const int k = i; // définition d'une constante initialisée
                   // avec la valeur saisie

 return 0;
}
```

2.8 Variable de boucle

On peut déclarer une variable de boucle directement dans l'instruction *for*. Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.

Exemple :

```
#include <iostream>

using namespace std;

int main() {
  for(int i=0; i<10; i++)
    cout << i << ' ';
  // i n'est pas utilisable à l'extérieur du bloc for

  return 0;
}
/*-- résultat de l'exécution --------
0 1 2 3 4 5 6 7 8 9
-------------------------------------*/
```

2.9 Visibilité des variables

L'opérateur de résolution de portée *::* permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream>

using namespace std;

int i = 11;

int main() {
    int i = 34;
    {  int i = 23;

       ::i = ::i + 1;
       cout << ::i << "  " << i << endl;
    }
    cout << ::i << "  " << i << endl;

    return 0;
}
/*-- résultat de l'exécution ------------------
12  23
12  34
```

Note : L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.

En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres ou pour accéder à un identificateur dans un espace de noms (cf espace de noms).

2.10 Les constantes

Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur *#define* pour définir des constantes.
Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter.

En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :
- inclusion de fichiers
- compilation conditionnelle.

Le mot réservé *const* permet de définir une constante.

L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.
Exemple :

```
const int N = 10; // N est un entier constant.
const int MOIS=12, AN=1995; // 2 constantes entières
int tab[2 * N];  // autorisé en C++ (interdit en C)
```

2.11 Constantes et pointeurs

Il faut distinguer ce qui est pointé du pointeur lui même.

- La donnée pointée est constante :
```
const char  *ptr1 = "QWERTY";
ptr1++;      // autorisé
*ptr1 = 'A'; // ERROR: assignment to const type
```
- Le pointeur est constant :
```
char * const ptr2 = "QWERTY";
ptr2++;      // ERROR: increment of const type
*ptr2 = 'A'; // autorisé
```
- Le pointeur et la donnée sont constants :
```
const char * const ptr3 = "QWERTY";
ptr3++;      // ERROR: increment of const type
*ptr3 = 'A'; // ERROR: assignment to const type
```

2.12 Les types composés

En C++, comme en langage C, le programmeur peut définir des nouveaux types en définissant des struct , enum ou union.
Mais contrairement au langage C, l'utilisation de typedef n'est plus obligatoire pour renommer un type.
Exemple :

```
struct FICHE {        // définition du type FICHE
  char *nom, *prenom;
```

```
  int  age;
};
// en C, il faut ajouter la ligne :
//     typedef struct FICHE FICHE;

FICHE adherent, *liste;

enum BOOLEEN { FAUX, VRAI };
// en C, il faut ajouter la ligne :
//     typedef enum BOOLEEN BOOLEEN;

BOOLEEN trouve;
trouve = FAUX;
trouve = 0; // ERREUR en C++ : vérification stricte des types
trouve = (BOOLEEN) 0; // OK

Chaque énumération enum est un type particulier,
différent de int et ne peut prendre que les valeurs énumérées
dans sa définition :

enum Jour {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};
enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};

Jour j;  // définition d'une variable de type Jour

j = LUNDI; // OK
j = 2;     // ERREUR en C++ (légal en C)

int i = MARDI;  // légal, il existe une conversion
                // implicite vers le type int

Couleur c; // définition d'une variable de type Couleur

c = j;     // ERREUR en C++ (légal en C)
```

2.13 Variables références

En plus des variables normales et des pointeurs, le C++ offre les variables références.

Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Dès lors, une modification de l'une affectera le contenu de l'autre.

```
int i;
int & ir = i; // ir est une référence à i
int *ptr;

i=1;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 1  ir= 1

ir=2;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 2  ir= 2
```

```
ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 3  ir= 3
```

Une variable référence doit obligatoirement être initialisée et le type de l'objet initial doit être le même que l'objet référence.

Intérèt :

- passage des paramètres par référence
- utilisation d'une fonction en lvalue

2.14 Allocation mémoire

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour remplacer respectivement les fonctions malloc et free (bien qu'il soit toujours possible de les utiliser).

2.14.1 L'opérateur *new*

L'opérateur *new* réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne l'adresse de début de la zone mémoire allouée.

```
int *ptr1, *ptr2, *ptr3;

// allocation dynamique d'un entier
ptr1 = new int;

// allocation d'un tableau de 10 entiers
ptr2 = new int [10];

// allocation d'un entier avec initialisation
ptr3 = new int(10);

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

// allocation dynamique d'une structure
ptr4 = new date;

// allocation dynamique d'un tableau de structure
ptr5 = new date[10];

// allocation dynamique d'une structure avec initialisation
ptr6 = new date(d);
```

En cas d'erreur d'allocation par *new*, une *exception bad_alloc* est lancée s'il n'y a pas de *fonction d'interception* définie par l'utilisateur.

L'allocation des tableaux à plusieurs dimensions est possible :

```
typedef char TAB[80]; // TAB est un synonyme de : tableau de 80 caractères
TAB *ecran;

ecran = new TAB[25]; // ecran est un tableau de 25 fois 80 caractères
ecran[24][79]='\$';
```

ou

```
char (*ecran)[80] = new char[25][80];

ecran[24][79]='\$';
```

Attention à ne pas confondre :

```
int *ptr = new int[10]; // création d'un tableau de 10 entiers
```

avec

```
int *ptr = new int(10); // création d'un entier initialisé à 10
```

2.14.2 L'opérateur delete

L'opérateur delete libère l'espace mémoire alloué par new à un seul objet, tandis que l'opérateur delete[] libère l'espace mémoire alloué à un tableau d'objets.

```
// libération d'un entier
delete ptr1;

// libération d'un tableau d'entier
delete[] ptr2;
```

L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraîne aucune conséquence fâcheuse (l'opération est tout simplement ignorée).

Notes :

- A chaque instruction new doit correspondre une instruction delete.
- Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire.
- La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.
- Tout ce qui est alloué avec *new []*, doit être libéré avec *delete[]*.

2.14.3 La fonction d'interception *set_new_handler*

Si une allocation mémoire par *new* échoue, une fonction d'erreur utilisateur peut être appelée.

La fonction *set_new_handler*, déclarée dans *new.h*, permet d'installer votre propre fonction d'erreur.

```
#include <iostream>
#include <stdlib.h>   // exit()
#include <new>      // set_new_handler()

using namespace std;

// fonction d'erreur d'allocation mémoire dynamique
void erreur_memoire( void) {
  cerr << "\nLa mémoire disponible est insuffisante !!!" << endl;
  exit(1);
}

int main() {
  set_new_handler( erreur_memoire );
  double *tab = new double [1000000000];

  return 0;
}
```

Si la fonction d'interception erreur_memoire ne comporte pas un *exit*, une nouvelle demande d'allocation mémoire est faite, et cela jusqu'à ce que l'allocation réussisse.

```
#include <iostream>
#include <new>      // set_new_handler()

using namespace std;

// fonction d'erreur d'allocation mémoire dynamique
void erreur_memoire(void) {
  static int n = 0;

  cerr << ++n << " fois, mémoire insuffisante" << endl;
  if ( n==10 )
     exit(1);
}

int main() {
  set_new_handler( erreur_memoire );
  double *tab = new double [100000000];

  return 0;
}
```

3

Les fonctions

Sommaire

3.1 Déclaration des fonctions

- Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.
- Ces déclarations sont identiques aux prototypes de fonctions de la norme C-ANSI.
- Cette déclaration est par ailleurs obligatoire avant utilisation (contrairement à la norme C-ANSI).
- La déclaration suivante *int f1();* où *f1* est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration *int f1(void);*
  (La norme C-ANSI considère que *f1* est une fonction qui peut recevoir un nombre quelconque d'arguments, eux mêmes de type quelconques, comme si elle était déclarée *int f1(...);* .)
- Une fonction dont le type de la valeur de retour n'est pas *void*, doit **obligatoirement** retourner une valeur.

3.2 Passage par référence

En plus du passage par valeur, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel.

Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple :

```
void echange(int &n1, int &n2) {
  // n1 est un alias du paramètre réel i
  // n2 est un alias du paramètre réel j
  int temp = n1;
  n1 = n2;   // toute modification de n1 est répercutée sur i
  n2 = temp; // toute modification de n2 est répercutée sur j
}

int main() {
  int i=2, j=3;
```

```
  echange(i, j);
  cout << "i= " << i << " j= " << j << endl;
  // affichage de :  i= 3  j= 2

  return 0;
}
```

Comme vous le remarquez, l'appel se fait de manière très simple.

- Utilisez les références quand vous pouvez,
- utiliser les pointeurs quand vous devez.

Cette facilité augmente la puissance du langage mais doit être utilisée avec précaution, car elle ne protège plus la valeur du paramètre réel transmis par référence.

L'utilisation du mot réservé *const* permet d'annuler ces risques pour les arguments de grande taille ne devant pas être modifiés dans la fonction.

```
struct FICHE {
  char nom[30], prenom[20], email[256];
};

void affiche(const FICHE  &f) {
  // passage par référence (plutôt que par valeur) pour des
  // raisons d'efficacité. Une modification de f provoque une
  // erreur de compilation
  cout << f.nom << " " << f.prenom;
  cout <<  " " << f.adresse << endl;
}

int main() {
  FICHE user = {
    "Dancel", "Alain", "alain.dancel@free.fr"
  };
  affiche(user);

  return 0;
}
```

3.2.1 Références et pointeurs peuvent se combiner :

```
int ouverture(FILE *&f, const char *nf, const char *mode) {
// passage par référence d'un pointeur.
  f = fopen(nf, mode);
  return (f==null) ? -1 : 0;
}

int main() {
  FILE *fic;

  if (ouverture(fic, "/tmp/toto.fic", "r") == -1)
    exit(1);

  int i;
  fscanf(fic, "%d", &i);
  // etc ...
}
```

3.3 Valeur par défaut des paramètres

Certains arguments d'une fonction peuvent prendre souvent la même valeur.

Pour ne pas avoir à spécifier ces valeurs à chaque appel de la fonction, le C++ permet de déclarer des valeurs par défaut dans le prototype de la fonction.

Exemple :

```
void print(long valeur, int base = 10);

int main() {
  print(16);    // affiche 16    (16 en base 10)
  print(16, 2); // affiche 10000 (16 en base 2)

  return 0;
}

void print(long valeur, int base){
  cout << ltostr(valeur, base) << endl;
}
```

- Les paramètres par défaut sont obligatoirement les derniers de la liste.
- Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

3.4 Fonction *inline*

Le mot clé *inline* remplace avantageusement l'utilisation de *#define* du préprocesseur pour définir des pseudo-fonctions.

Afin de rendre l'exécution plus rapide d'une fonction et à condition que celle ci soit de courte taille, on peut définir une fonction avec le mot réservé *inline*.

Le compilateur générera, à chaque appel de la fonction, le code de celle ci. Les fonctions inline se comportent comme des fonctions normales et donc, présentent l'avantage de vérifier les types de leurs arguments, ce que ne fait par la directive *#define*.

Exemple :

```
#include <iostream>

using namespace std;

inline int carre(int n); // déclaration

void main() {
  cout << carre(10) << endl;
}

// inline facultatif à la définition, mais préférable
inline int carre(int n) {
  return n * n;
}
```

Note : contrairement à une fonction normale, la portée d'une fonction inline est réduite au module dans lequel elle est déclarée.

3.5 Surcharge de fonctions

Une fonction se définit par :

- son nom,
- sa liste typée de paramètres formels,
- le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la signature de la fonction.

On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents :

```
int somme( int n1, int n2)
       { return n1 + n2; }

int somme( int n1, int n2, int n3)
       { return n1 + n2 + n3; }

double somme( double n1, double n2)
       { return n1 + n2; }

int main() {
  cout << "1 + 2 = " << somme(1, 2) << endl;
  cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
  cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;

  return 0;
}
```

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.

Ce choix se faisant à la compilation, fait que l'appel d'une fonction surchargée procure des performances identiques à un appel de fonction classique.

On dit que l'appel de la fonction est résolu de manière statique.

Autres exemples :

1.
```
enum Jour {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};
enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};

void f1(Jour j);
void f1(Couleur c);
// OK : les types énumérations sont tous différents
```

2.
```
void f2(char *str) { /* ... */ }
void f2(char ligne[80]) { /* ... */ }
// Erreur: redéfinition de la fonction f
//         char * et char [80] sont considérés de même type
```

3.
```
int somme1(int n1, int n2) {return n1 + n2;}
int somme1(const int n1, const int n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

4.
```
int somme2(int n1, int n2) {return n1 + n2;}
int somme2(int & n1, int & n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

5.
```
int    somme3(int n1, int n2) {return n1 + n2;}
double somme3(int n1, int n2) {return (double) n1 + n2;}
// Erreur: seul le type des paramètres permet de faire la distinction
//         entre les fonctions et non pas la valeur retournée.
```

6.
```
int somme4(int n1, int n2) {return n1 + n2;}
int somme4(int n1, int n2=8) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

7.
```
typedef int entier;     // entier est un alias de int
int somme5(int n1, int n2) {return n1 + n2;}
int somme5(entier e1, entier e2) {return e1 + e2;};
// Erreur : redéfinition de somme5 : des alias de type ne
//          sont pas considérés comme des types distincts
```

3.6 Retour d'une référence

3.6.1 Fonction retournant une référence

Une fonction peut retourner une valeur par référence et on peut donc agir, à l'extérieur de cette fonction, sur cette valeur de retour.

La syntaxe qui en découle est plutôt inhabituelle et déroutante au début.

```
#include <iostream>

using namespace std;

int t[20];  // variable globale -> beurk !

int & nIeme(int i) {
  return t[i];
}

int main() {
  nIeme(0) = 123;
  nIeme(1) = 456;
  cout << t[0] << " " << ++nIeme(1);

  return 0;
}
// -- résultat de l'exécution -----------
// 123 457
```

Tout se passe comme si nIeme(0) était remplacé par t[0]. Une fonction retournant une référence (non constante) peut être une *lvalue*. La variable dont la référence est retournée doit avoir une

durée de vie permanente.

En C, pour réaliser la même chose, nous aurions du écrire :

```
int t[20];  // variable globale -> beurk !

int * nIeme(int i) {
  return &t[i];
}

int main() {
  *nIeme(0) = 123;
  *nIeme(1) = 456;
  printf("%d %d\n", t[0] ++(*nIeme(1)) );

  return 0;
}
// -- résultat de l'exécution -----------
// 123 457
```

ce qui est moins lisible et pratique ...

3.6.2 Retour d'une référence constante

Afin d'éviter la création d'une copie, dans la pile, de la valeur retournée lorsque cette valeur est de taille importante, il est possible de retourner une valeur par référence constante.

Le préfixe const, devant le type de la valeur retournée, signifie au compilateur que la valeur retournée est constante.

```
const Big & f1() {
  static Big b;    // notez le static ...
  // ...
  return b;
}
int main() {
  f1() = 12;  // erreur
  return 0;
}
```

3.7 Utilisation d'une fonction écrite en C

Le compilateur C++ génère pour chaque fonction un nom dont l'éditeur de liens aura besoin. Le nom généré en C++ se fait à partir de la signature de la fonction.

Ainsi, à une fonction surchargée 2 fois, correspondra 2 fonctions de noms différents dans le module objet.

En C, la signature de la fonction ne comporte que le nom de la fonction.

Pour pouvoir utiliser dans un programme C++ des fonctions compilées par un compilateur C, il faut déclarer ces fonctions de la façon suivante :

```
extern "C" int f1(int i, char c);
extern "C" void f2(char *str);
```

ou

```
extern "C" {
  int f1(int i, char c);
  void f2(char *str);
}
```

Par contre, les fonctions f1 et f2 ne pourront pas être surchargées.

3.8 Fichier d'en-têtes pour C et C++

Le symbole *__cplusplus* est défini pour les compilateurs C++ uniquement.

Il facilite l'écriture de code commun aux langages C et C++, et plus particulièrement pour les fichiers d'en-têtes :

```
#ifdef __cplusplus
  extern "C" {
#endif
#include <stdio.h>
void fail(char *msg);

#ifdef __cplusplus
  }
#endif
```

Cet exemple peut être compilé aussi bien en C qu'en C++.

4 Les classes

Sommaire

4.1 Définition d'une classe

la classe décrit le modèle structurel d'un objet :

- ensemble des attributs (ou champs ou données membres) décrivant sa structure
- ensemble des opérations (ou méthodes ou fonctions membres) qui lui sont applicables.

Une classe en C++ est une structure qui contient :

- des fonctions membres
- des données membres

Les mots réservés public et private délimitent les sections visibles par l'application.

Exemple :

```
class Avion {
  public :   // fonctions membres publiques
    void init(char [], char *, float);
    void affiche();
  private :  // membres privées
    char  immatriculation[6], *type; // données membres privées
    float poids;
    void  erreur(char *message);     // fonction membre privée
};  // n'oubliez pas ce ; après l'accolade
```

4.2 Droits d'accès

L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe.

Elle est réalisée à l'aide des mots clés :

- **private** : les membres privés ne sont accessibles que par les fonctions membres de la classe. La partie privée est aussi appelée réalisation.
- **protected** : les membres protégés sont comme les membres privés. Mais ils sont aussi accessibles par les fonctions membres des classes dérivées (voir l'héritage).
- **public** : les membres publics sont accessibles par tous. La partie publique est appelée interface.

Les mots réservés *private* , *protected* et *public* peuvent figurer plusieurs fois dans la déclaration de la classe.

Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

4.3 Types de classes

4.3.1 *struct Classe1 { /* ... */ };*

Tous les membres sont par défaut d'accès public le contrôle d'accès est modifiable Cette structure est conservée pour pouvoir compiler des programmes écrits en C.

Exemple :

```
struct Date {
  // méthodes publiques (par défaut)
  void set_date(int, int, int);
  void next_date();
  // autres méthodes ....
private :             // données privées
  int _jour, _mois, _an;
};
```

4.3.2 *union Classe2 { /* ... */ };*

Tous les membres sont d'accès public par défaut le contrôle d'accès n'est pas modifiable.

4.3.3 *class Classe3 { /* ... */ };*

Tous les membres sont d'accès private (par défaut) le contrôle d'accès est modifiable.
C'est cette dernière forme qui est utilisée en programmation objet C++ pour définir des classes.

4.4 Définition des fonctions membres

- En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe.

```
class Avion {
  public :
  void init(char [], char *, float);
  void affiche();
  private :
```

```
  char  _immatriculation[6], *_type;
  float _poids;
  void  erreur(char *message);
  };
```

- Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source. Syntaxe de la définition hors de la classe d'une méthode :

```
type_valeur_retournée Classe::nom_méthode( paramètres_formels )
 {
   // corps de la fonction
 }
```

- Dans la fonction membre on a un accès direct à tous les membres de la classe.

  Exemple de définition de méthode de la classe Avion :

```
void Avion::init(char m[], char *t, float p) {
  if ( strlen(m) != 5 ) {
    erreur("Immatriculation invalide");
    strcpy(_immatriculation, "?????");
    }
  else
  strcpy(_immatriculation, m);
  _type = new char [strlen(t)+1];
  strcpy(_type, t);
  _poids = p;
  }

void Avion::affiche() {
  cout << _immatriculation << "  " << _type;
  cout << "  " << _poids << endl;
  }
```

- La définition de la méthode peut aussi avoir lieu à l'intérieur de la déclaration de la classe.

  Dans ce cas, ces fonctions sont automatiquement traitées par le compilateur comme des fonctions inline.

  Une fonction membre définie à l'extérieur de la classe peut être aussi qualifiée explicitement de fonction *inline*.

  Exemple :

```
class Nombre {
  public :
  void setnbre(int n) { nbre = n; } // fonction inline
  int  getnbre() { return nbre; }   // fonction inline
  void affiche();
  private :
  int nbre;
  };

inline void Nombre::affiche() {   // fonction inline
  cout << "Nombre = " << nbre << endl;
  }
```

Rappel : la visibilité d'une fonction inline est restreinte au module seul dans laquelle elle est définie.

4.5 Instanciation d'une classe

De façon similaire à une *struct* ou à une *union*, le nom de la classe représente un nouveau type de donnée.

On peut donc définir des variables de ce nouveau type; on dit alors que vous créez des objets ou des instances de cette classe.

Exemple :

```
Avion av1;    // une instance simple (statique)
Avion *av2;   // un pointeur (non initialisé)
Avion compagnie[10];  // un tableau d'instances
av2 = new Avion; // création (dynamique) d'une instance
```

4.6 Utilisation des objets

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux attributs et méthodes de la classe.

Cet accès se fait comme pour les structures à l'aide de l'opérateur . (point) ou ->(tiret supérieur).

Exemple :

```
av1.init("FGBCD", "TB20", 1.47);
av2->init("FGDEF", "ATR 42", 80.0);
compagnie[0].init("FEFGH","A320", 150.0);
av1.affiche();
av2->affiche();
compagnie[0].affiche();
av1.poids = 0; // erreur, poids est un membre privé
```

4.7 Fonctions membres constantes

Certaines méthodes d'une classe ne doivent (ou ne peuvent) pas modifier les valeurs des données membres de la classe, ni retourner une référence non constante ou un pointeur non constant d'une donnée membre :

on dit que ce sont des **fonctions membres constantes**.

Ce type de déclaration renforce les contrôles effectués par le compilateur et permet donc une programmation plus sûre sans coût d'exécution. Il est donc **très souhaitable** d'en déclarer aussi souvent que possible dans les classes.

Exemple :

```
class Nombre {
  public :
    void setnbre(int n) { nbre = n; }
    // méthodes constantes
```

```
    int  getnbre() const { return nbre; }
    void affiche() const;
  private :
    int nbre;
};

inline void Nombre::affiche() const {
  cout << "Nombre = " << nbre << endl;
}
```

Une fonction membre *const* peut être appelée sur des objets constants ou pas, alors qu'une fonction membre non constante ne peut être appelée que sur des objets non constants.

Exemple :

```
const Nombre n1;  // une constante
n1.affiche();    // OK
n1.setnbre(15);  // ERREUR: seule les fonctions const peuvent
                 // être appelées pour un objet constant
Nombre n2;
n2.affiche();    // OK
n2.setnbre(15);  // OK
```

Surcharge d'une méthode par une méthode constante :

Une méthode déclarée comme constante permet de surcharger une méthode non constante avec le même nombre de paramètres du même type.

Exemple :

```
class String {
  public :
    // etc ...
    char & nieme(int n);        // (1)
    char   nieme(int n) const;  // (2)
    // etc...
  private :
    char *_str;
};
```

S'il n'y a pas l'attribut *const* dans la deuxième méthode, le compilateur génère une erreur *"String::nieme() cannot be redeclared in class"*.

Cette façon de faire permet d'appliquer la deuxième méthode *nieme()* à des objets constants et la première méthode *nieme()* à des objets variables :

```
int main() {
  String ch1;
  // initialisation de ch1

  const String ch2;
  // initialisation de ch2

  cout << ch1.nieme(1);  // appel de la méthode (1)
  cout << ch2.nieme(1);  // appel de la méthode (2)
```

```
  return 0;
}
```

L'écriture d'une instruction comme :

```
ch2.nieme(3) = 'u';
```

ne sera pas compilable, car la méthode (2) ne retourne pas une référence.

Si pour des raisons d'efficacité, la méthode (2) doit retourner une référence , on retournera alors une référence constante et l'on écrira donc :

```
const char & nieme(int n) const;   // (2 bis)
```

Et que se passe-t-il si j'écris :

```
char & nieme(int n) const;   // (2 bis)
//on retourne une référence non constante
```

au lieu de :

```
const char & nieme(int n) const;   // (2 bis)
//on retourne une référence constante
```

le programme suivant se compile parfaitement et donne un résultat surprenant :

```
int main() {
  const String ch2;
  // initialisation de ch2 avec la chaîne "coco"

  ch2.nieme(3) = 'a';

  ch2.affiche(); // affiche "coca"

  return 0;
}
```

La question maintenant est :

Comment une méthode constante a-t-elle pu modifier la valeur d'un objet constant ?

Quand on déclare une méthode constante, le compilateur vérifie que la méthode ne modifie pas une donnée membre. Et c'est le cas de la méthode *nieme()*. Pour générer une erreur il aurait fallu qu'elle change la valeur de *_str*, chose qu'elle ne fait pas. Elle ne peut changer la valeur que d'un caractère pointé par *_str*.

4.8 Un exemple complet : Pile d'entiers(1)

```
// IntStack.cpp : pile d'entiers -------------------------------
#include <iostream>
#include <assert.h>
#include <stdlib.h>  // rand()

using namespace std;

class IntStack {
  public:
```

```
    void init(int taille = 10); // création d'une pile

    void push(int n);   // empile un entier au sommet de la pile
    int pop();          // retourne l'entier au sommet de la pile
    int vide() const;   // vrai, si la pile est vide
    int pleine() const; // vrai, si la pile est pleine
    int getsize() const { return _taille; }
  private:
    int _taille;  // taille de la pile
    int _sommet;  // position de l'entier à empiler
    int *_addr;   // adresse de la pile
};

void IntStack::init(int taille ) {
  _addr = new int [ _taille = taille ];
  assert( _addr != 0 );
  _sommet = 0;
}

void IntStack::push(int n) {
  if ( ! pleine() )
    _addr[ _sommet++ ] = n;
}

int IntStack::pop() {
  return  ( ! vide() ) ? _addr[ --_sommet ] : 0;
}

int IntStack::vide() const {
  return ( _sommet == 0 );
}

int IntStack::pleine() const {
  return ( _sommet == _taille );
}

int main() {
  IntStack pile1;
  pile1.init(15);            // pile de 15 entiers
  while ( ! pile1.pleine() ) // remplissage de la pile
    pile1.push( rand() % 100 );
  while ( ! pile1.vide() ) // Affichage de la pile
    cout << pile1.pop() << " ";
  cout << endl;

  return 0;
}
```

4.9 Constructeurs et destructeurs

Nous pouvons remarquer que :

- les données membres d'une classe ne peuvent pas être initialisées; il faut donc prévoir une méthode d'initialisation de celles-ci (voir la méthode init de l'exemple précédent).

- si l'on oublie d'appeler cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des surprises fâcheuses dans la suite de l'exécution.
- de même, après avoir fini d'utiliser l'objet, il est bon de prévoir une méthode permettant de détruire l'objet (libération de la mémoire dynamique ...).

4.9.1 Constructeur

Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet, assurant ainsi une initialisation correcte de l'objet.

Ce constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un *void*).

Exemple :

```
class Nombre {
  public :
    Nombre(); // constructeur par défaut
    // ...
  private :
    int _i;
};

Nombre::Nombre() {
  _i = 0;
}
```

On appelle **constructeur par défaut** un constructeur n'ayant pas de paramètre ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre {
  public :
    Nombre(int i=0); // constructeur par défaut
    // ...
  private :
    int _i;
};

Nombre::Nombre(int i) {
  _i = i;
}
```

- si le concepteur de la classe ne spécifie pas de constructeur, le compilateur générera un constructeur par défaut.
- comme les autres fonctions, les constructeurs peuvent être surchargés.

```
class Nombre {
  public :
  Nombre();      // constructeur par défaut
  Nombre(int i); // constructeur à 1 paramètre
  private :
  int _i;
  };
```

Le constructeur est appelé à l'instanciation de l'objet. Il n'est donc pas appelé quand on définit un pointeur sur un objet...

Exemple :

```
Nombre n1;     // correct, appel du constructeur par défaut
Nombre n2(10); // correct, appel du constructeur à 1 paramètre
Nombre n3 = Nombre(10); // idem que n2

Nombre *ptr1, *ptr2;  // correct, pas d'appel aux constructeurs
ptr1 = new Nombre;  // appel au constructeur par défaut
ptr2 = new Nombre(12); // appel du constructeur à 1 paramètre

Nombre tab1[10]; // chaque objet du tableau est initialisé
                 // par un appel au constructeur par défaut
Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) };
                 // initialisation des 3 objets du tableau
                 // par les nombres 10, 20 et 30
```

Un objet anonyme et temporaire peut être créé par appel au constructeur de la classe :

```
void analyse(Nombre n);

int main() {
  analyse( Nombre(123) );
  return 0;
}
```

évitant ainsi d'écrire :

```
int main() {
  Nombre n(123);
  analyse( n );
  return 0;
}
```

4.9.2 Destructeur

De la même façon que pour les constructeurs, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

Ce destructeur est une fonction :

- qui porte comme nom, le nom de la classe précédé du caractère (tilda)
- qui ne retourne pas de valeur (pas même un void)
- qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

Exemple :

```
class Exemple {
  public :
    // ...
    ~Exemple();
  private :
    // ...
};

Exemple::~Exemple() {
  // ...
}
```

Comme pour le constructeur, le compilateur générera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.

- Si des constructeurs sont définis avec des paramètres, le compilateur ne générera pas le constructeur par défaut.
- Les constructeurs et destructeurs sont les seules méthodes non constantes qui peuvent être appelées pour des objets constants.

Attention aux fautes de frappe, particulièrement dans l'identificateur du constructeur :

```
class Essai {
  private:
  char *_ptr;
  public:
  essai() { _ptr = new char[80];}  // constructeur
  void init(char c) {for(int i=0; i<80; i++) _ptr[i]=c;}
  ~Essai() { delete[] _ptr; }
  };

Essai e;
e.init('\0');  // Aïe, Aïe, Aïe ... core dump ...
```

A cause de la faute de frappe dans le nom du constructeur, celui ci n'a pas été appelé à la création de l'objet *e*.

Certains compilateurs signalent cette erreur :

```
warning: Essai has Essai::~Essai() but no constructor (273)
error: no value returned from Essai::essai() (1404)
```

4.10 Exemple : Pile d'entiers avec constructeurs et destructeurs

```
// IntStack2.cpp : pile d'entiers -------------------------------
#include <iostream>
#include <assert.h>
#include <stdlib.h>  // rand()

using namespace std;

class IntStack {
  public:
    IntStack(int taille = 10);     // constructeur par défaut
    ~IntStack() { delete[] _addr; } // destructeur

    void push(int n);   // empile un entier au sommet de la pile
    int pop();          // retourne l'entier au sommet de la pile
    int vide() const;   // vrai, si la pile est vide
    int pleine() const; // vrai, si la pile est pleine
    int getsize() const { return _taille; }
  private:
    int _sommet;
    int _taille;
    int *_addr;  // adresse de la pile
```

```
};

IntStack::IntStack(int taille) {
  _addr = new int [ _taille = taille ];
  assert( _addr != 0 );
  _sommet = 0;
}

// ...
// le reste des méthodes est sans changement
//

int main() {
  IntStack pile1(15);        // pile de 15 entiers
  while ( ! pile1.pleine() ) // remplissage de la pile
    pile1.push( rand() % 100 );

  while ( ! pile1.vide() ) // Affichage de la pile
    cout << pile1.pop() << "   ";
  cout << endl;
```

4.11 Constructeur copie

4.11.1 Présentation du problème

Reprenons la classe IntStack avec un constructeur et un destructeur et écrivons une fonction (AfficheSommet) qui affiche la valeur de l'entier au sommet de la pile qui est passée en paramètre.

Exemple d'appel de cette fonction :

```
int main() {
  IntStack pile1(15);  // création d'une pile de 15 entiers

  // ...

  AfficheSommet( pile1 );
  // ...
}
```

Une version fausse (pour l'instant) de cette fonction peut ressembler au code qui suit :

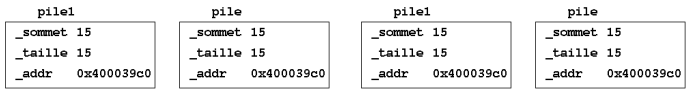
```
void AfficheSommet( IntStack pile ) {
  cout << "Sommet de la pile : " << pile.pop() << endl;
}
```

Ici, la pile est passée par valeur, donc il y a création de l'objet temporaire nommé pile créé en copiant les valeurs du paramètre réel pile1.

L'affichage de la valeur au sommet de la pile marche bien, mais la fin de cette fonction fait appel au destructeur de l'objet local *pile* qui libère la mèmoire allouée par l'objet *pile1* parce que les données membres de l'objet *pile* contiennent les mêmes valeurs que celles de l'objet *pile1* (cf 4.1).

Pour éviter cela :

- il faut définir la fonction AfficheSommet comme :
  `void AfficheSommet( IntStack & pile ) {`



(a) *pile1* (b) *pile*

Fig. 4.1 – *Valeurs des champs membres des objets de type pile*

```
  cout << "Sommet de la pile : " << pile.pop() << endl;
}
```

Mais une opération comme *pile.pop()* dépile un entier de la pile *pile1*!

il faut faire une copie intelligente : création du **constructeur de copie**.

- Le constructeur de copie est invoqué à la construction d'un objet à partir d'un objet existant de la même classe.

```
Nombre n1(10); // appel du constructeur à 1 paramètre
Nombre n2(n1); // appel du constructeur de copie
Nombre n3=n1;  // appel du constructeur de copie
```

- Le constructeur de copie est appelé aussi pour le passage d'arguments par valeur et le retour de valeur

```
Nombre traitement(Nombre n) {
  static Nombre nbre;
  // ...
  return nbre;      // appel du constructeur de copie
  }

int main() {
  Nombre n1, n2;
  n2 = traitement( n1 ); // appel du constructeur de copie
  }
```

- Le compilateur C++ génère par défaut un constructeur de copie bêteŻ.

4.11.2 Constructeur de copie de la classe IntStack

```
class IntStack {
  public:
    IntStack(int taille = 10);       // constructeur par défaut
    IntStack(const IntStack & s);    // constructeur de copie
    ~IntStack() { delete[] _addr; }  // destructeur
    // ...
  private:
    int _taille;  // taille de la pile
    int _sommet;  // position de l'entier à empiler
    int *_addr;   // adresse de la pile
};

// ...
```

```
IntStack::IntStack(const IntStack & s) {  // constructeur de copie
  _addr = new int [ _taille = s._taille ];
  _sommet = s._sommet;
  for (int i=0; i< _sommet; i++)  // recopie des éléments
    _addr[i] = s._addr[i];
}
```

4.12 Classes imbriquées

Il est possible de créer une classe par une relation d'appartenance : relation a un ou est composée de.

Exemple : une voiture a un moteur, a des roues ...

```
class Moteur { /* ... */ };
class Roue  { /* ... */ };
class Voiture {
  public:
    // ....
  private:
    Moteur _moteur;
    Roue   _roue[4];
    // ....
};
```

4.13 Affectation et initialisation

Le langage C++ fait la différence entre l'initialisation et l' affectation.

- l'affectation consiste à modifier la valeur d'une variable (et peut avoir lieu plusieurs fois).
- l'initialisation est une opération qui n'a lieu qu'une fois immédiatement après que l'espace mémoire de la variable ait été alloué. Cette opération consiste à donner une valeur initiale à l'objet ainsi créé.

4.14 Liste d'initialisation d'un constructeur

Soit la classe :

```
class Y { /* ... */ };

class X {
  public:
    X(int a, int b, Y y);
    ~X();
    // ....
  private:
    const int _x;
    Y _y;
    int _z;
};

X::X(int a, int b, Y y) {
  _x = a; // ERREUR: l'affectation à une constante est interdite
```

```
  _z = b; // OK : affectation
  // et comment initialiser l'objet membre _y ???
}
```

Comment initialiser la donnée membre constante $\_x$ et appeler le constructeur de la classe Y ?

Réponse : la liste d'initialisation.

La phase d'initialisation de l'objet utilise une liste d'initialisation qui est spécifiée dans la définition du constructeur.

Syntaxe :

```
nom\_classe::nom\_constructeur( args ... ) : liste\_d\_initialisation
{
  // corps du constructeur
}
```

Exemple :

```
X::X(int a, int b, Y y) : _x( a ) , _y( y ) , _z( b ) {
  // rien d'autre à faire
}
```

- L'expression *_x(a)* indique au compilateur d'initialiser la donnée membre $\_x$ avec la valeur du paramètre *a*.
- L'expression *_y(y)* indique au compilateur d'initialiser la donnée membre $\_y$ par un appel au constructeur (avec l'argument *y*) de la classe *Y*.

4.15 Le pointeur *this*

Toute méthode d'une classe X a un paramètre caché : le pointeur *this*.

Celui contient l'adresse de l'objet qui l'a appelé, permettant ainsi à la méthode d'accéder aux membres de l'objet.

Il est implicitement déclaré comme (pour une variable) :

```
X * const this;
```

et comme (pour un objet constant) :

```
const X * const this;
```

et initialisé avec l'adresse de l'objet sur lequel la méthode est appelée.

Il peut être explicitement utilisé :

```
classe X {
  public:
    int f1() { return this->i; }
    // idem que :  int f1() { return i; }
  private:
    int i;
    // ...
};
```

Une fonction membre qui retourne le pointeur *this* peut être chaînée, étant donné que les opérateurs de sélection de membres . (point) et ->(tiret supérieur) sont associatifs de gauche à droite :

exemple :

```
classe X {
  public:
    X f1() { cout << "X "; return *this; }
    // ...
  private:
    // ...
};

int main() {
  X x;
  x.f1().f1().f1(); // affiche : X X X

  return 0;
}
```

La fonction membre *f1* a tout intérêt de retourner une référence :

```
 X &f1() { cout << "X "; return *this; }
```

Notes :

- La valeur ne this ne peut pas être changée.
- this ne peut pas être explicitement déclaré.

4.16 Les membres statiques

Ces membres sont utiles lorsque l'on a besoin de gérer des données communes aux instances d'une même classe.

4.16.1 Données membres statiques

Si l'on déclare une donnée membre comme static, elle aura la même valeur pour toutes les instances de cette classe.

```
class Ex1 {
  public:
  Ex1() { nb++; /* ... */ }
  ~Ex1() { nb--; /* ... */ }
  private:
  static int nb;  // initialisation impossible ici
  };
```

L'initialisation de cette donnée membre statique se fera en dehors de la classe et en global par une déclaration :

```
int Ex1::nb = 0; // initialisation du membre static
```

4.16.2 Fonctions membres statiques

De même que les données membres statiques, il existe des fonctions membres statiques.

- ne peuvent accéder qu'aux membres statiques,
- ne peuvent pas être surchargés,
- existent même s'il n'y a pas d'instance de la classe.

```
class Ex1 {
  public:
  Ex1() { nb++; /* ... */ }
  ~Ex1() { nb--; /* ... */ }
  static void affiche() {  // fonction membre statique
    cout << nb << endl;
    }
  private:
  static int nb;
  };

int Ex1::nb = 0; // initialisation du membre static (en global)

int main() {
  Ex1.affiche();   // affiche 0 (pas d'instance de Ex1)
  Ex1 a, b, c;
  Ex1.affiche();   // affiche 3
  a.affiche();     // affiche 3
  return 0;
}
```

4.17 Classes et fonctions amies

Dans la définition d'une classe il est possible de désigner des fonctions (ou des classes) à qui on laisse un libre accès à ses membres privés ou protégés.

C'est une infraction aux règles d'encapsulation pour des raisons d'efficacité.

- Exemple de fonction amie :

```
class Nombre {
  // ici je désigne les fonctions qui pourront accéder
  // aux membres privés de ma classe
  friend int manipule_nbre(); // fonction amie
  public :
  int getnbre();
  // ...
  private :
  int _nbre;
  };

int  manipule_nbre(Nombre n) {
  return n._nbre + 1;  // je peux le faire en tout légalité
  // parce que je suis une fonction amie
  // de la classe Nombre.
  }
```

- Exemple de classe amie :

```
class Window;  // déclaration de la classe Window
class Screen {
  friend class Window;
  public:
  //...
  private :
  //...
  };
```

Les fonctions membres de la classe *Window* peuvent accéder aux membres non-publics de la classe *Screen*.

5 Surchage d'opérateur

Sommaire

5.1 Introduction à la surcharge d'opérateurs

Le concepteur d'une classe doit fournir à l'utilisateur de celle ci toute une série d'opérateurs agissant sur les objets de la classe. Ceci permet une syntaxe intuitive de la classe.

Par exemple, il est plus intuitif et plus clair d'additionner deux matrices en surchargeant l'opérateur d'addition et en écrivant :

result = m0 + m1; que d'écrire *matrice_add(result, m0, m1);*

Règles d'utilisation :

- Il faut veiller à respecter l'esprit de l'opérateur. Il faut faire avec les types utilisateurs des opérations identiques à celles que font les opérateurs avec les types prédéfinis.
- La plupart des opérateurs sont surchargeables.
- les opérateurs suivants ne sont pas surchargeables : **:: . .* ?: sizeof**
  - il n'est pas possible de :
  - changer sa priorité
  - changer son associativité
  - changer sa pluralité (unaire, binaire, ternaire)
  - créer de nouveaux opérateurs

Quand l'opérateur + (par exemple) est appelé, le compilateur génère un appel à la fonction **operator+**.

Ainsi, l'instruction **a = b + c;** est équivalente aux instructions :

```
a = operator+(b, c);   // fonction globale
a = b.operator+(c);    // fonction membre
```

Les opérateurs *= () [] ->new delete* ne peuvent être surchargés que comme des fonctions membres.

5.2 Surcharge par une fonction membre

Par exemple, la surcharge des opérateurs + et = par des fonctions membres de la classe *Matrice* s'écrit :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice
class Matrice {  // matrice dim1 x dim2 d'entiers
  public:

    // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);
    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

// ...

void main() {
  Matrice a, b, c;
  b + c ;     // appel à : b.operator+( c );
  a = b + c;  // appel à : a.operator=(  b.operator+( c ) );
}
```

Une fonction membre (non statique) peut toujours utiliser le pointeur caché *this*.

Dans le code ci dessus, this fait référence à l'objet b pour l'opérateur + et à l'objet a pour l'opérateur =.

Définition de la fonction membre operator+ :

```
Matrice Matrice::operator+(const Matrice &c) {
  Matrice m;

  for(int i = 0; i < dim1; i++)
    for(int j = 0; j < dim2; j++)
      m._matrice[i][j] = this->_matrice[i][j] + c._matrice[i][j];
  return m;
}
```

Note : Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable. Une fonction membre renforce l'encapsulation. Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage (sauf l'affectation).

La fonction membre operator+ peut elle même être surchargée, pour dans l'exemple qui suit, additionner à une matrice un vecteur :

```
class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);
```

```
    Matrice operator+(const Vecteur &n2);
    // ...
};

// ...

Matrice b, c;
Vecteur v;

b + c; // appel à b.operator+(c);
b + v; // appel à b.operator+(v); addition entre une matrice
       // et un vecteur
v + b; // appel à v.operator+(b); --> ERREUR si la classe
       // Vecteur n'a pas défini l'addition entre un vecteur et
       // une matrice
```

5.3 Surcharge par une fonction globale

Cette façon de procéder est plus adaptée à la surcharge des opérateurs binaires.

En effet, elle permet d'appliquer des conversions implicites au premier membre de l'expression.

Exemple :

```
Class Nombre{
  friend Nombre operator+(const Nombre &, const Nombre &);

  public:
    Nombre(int n = 0) { _nbre = n; }
    //....
  private:
    int _nbre;
};

Nombre operator+(const Nombre &nbr1, const Nombre &nbr2) {
  Nombre n;

  n._nbre = nbr1._nbre + nbr2._nbre;
  return n;
}

void main() {
  Nombre n1(10);

  n1 + 20;  // OK appel à :  operator+( n1, Nombre(20) );
  30 + n1;  // OK appel à :  operator+( Nombre(30) , n1 );
}
```

5.4 Opérateur d'affectation

C'est le même problème que pour le constructeur de copie.

Le compilateur C++ construit par défaut un opérateur d'affectation "bête".

L'opérateur d'affectation est obligatoirement une fonction membre et il doit fonctionner correctement dans les deux cas suivants :

```
X x1, x2, x3; // 3 instances de la classe X
x1 = x1;
x1 = x2 = x3;
```

Exemple : Opérateur d'affectation de la classe Matrice :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice

class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    const Matrice &operator=(const Matrice &m);
    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

const Matrice &Matrice::operator=(const Matrice &m) {
  if ( &m != this ) {   // traitement du cas :   x1 = x1
    for(int i = 0; i < dim1; i++)   // copie de la matrice
      for(int j = 0; j < dim2; j++)
        this->_matrice[i][j] = m._matrice[i][j];
  }
  return *this;  // traitement du cas :    x1 = x2 = x3
}
```

5.5 Surcharge de ++ et −

Selon que l'opérateur est préfixé ou postfixé la notation sera différente :

- notation préfixée :
  - fonction membre : *X operator++();*
  - fonction globale : *X operator++(X &);*
- notation postfixée :
  - fonction membre : *X operator++(int);*
  - fonction globale : *X operator++(X &, int);*

Exemple :

```
class BigNombre {
  public:
    // ...
    BigNombre operator++();    // préfixée
    BigNombre operator++(int); // postfixée
    // ...
};

// ...
```

```
void main() {
  BigNombre n1;
  n1++; // notation postfixéé
  ++n1; // notation préfixée
}
```

5.6 Opérateurs de conversion

Dans la définition complète d'une classe, il ne faut pas oublier de définir des opérateurs de conversions de types.

Il existe deux types de conversions de types :

- la conversion de type prédéfini (ou défini préalablement) vers le type classe en question. Ceci sera effectué grâce au **constructeur de conversion**.
- la conversion du type classe vers un type prédéfini (ou défini préalablement). Cette conversion sera effectuée par des **fonctions de conversion**.

5.6.1 Constructeur de conversion

Un constructeur avec un seul argument de type T permet de réaliser une conversion d'une variable de type T vers un objet de la classe du constructeur.

```
class Nombre {
  public:
    Nombre(int n) { _nbre = n; }
  private:
    int _nbre;
};

void f1(Nombre n) { /* ... */ }

void main() {
  Nombre n = 2; // idem que :  Nombre n(2);
  f1(3); // Appel du constructeur Nombre(int) pour réaliser
         // la conversion de l'entier 3 en un Nombre.
         // Pas d'appel du constructeur de copie
}
```

Dans cet exemple, le constructeur avec un argument permet donc d'effectuer des conversions d'entier en *Nombre*.

5.6.2 Fonction de conversion

Une fonction de conversion est une méthode qui effectue une conversion vers un type T. Elle est nommé operator *T()*. Cette méthode n'a pas de type de retour (comme un constructeur), mais doit cependant bien retourner une valeur du type T.

```
class Article {
  public :
    Article(double prix=0.0):_prix(prix) {}
    operator double() const { return _prix; }
  private :
    double _prix;
```

```
};

void main() {
  double total;
  Article biere(17.50);
  // utilisation implicite de la conversion Article -> double
  total = 7 * biere;
}
```

6 Les modèles

Sommaire

6.1 Les patrons de fonctions

Les termes : fonction générique, fonction modèle ou fonction template définissent la même notion.

Lorsque l'algorithme est le même pour plusieurs types de données, il est possible de créer un patron de fonction.

C'est un modèle à partir duquel le compilateur générera les fonctions qui lui seront nécessaires.

Exemple 1 :

```
template <class T>
void affiche(T *tab, unsigned int nbre) {
  for(int i = 0; i < nbre; i++)
    cout << tab[i] << " ";
  cout << endl;
}

void main() {
  int tabi[6] = {25, 4, 52, 18, 6, 55};
  affiche(tabi, 6);
  double tabd[3] = {12.3, 23.4, 34.5};
  affiche(tabd, 3);

  char *tabs[] = {"Bjarne", "Stroustrup"};
  affiche(tabs, 2);
}
```

Exemple 2 :

```
template <class T>     // déclaration du patron
T min(T, T);

// ...    la suite de votre programme

template <class T>     // définition du patron
T min(T t1, T t2) {
  return t1 < t2 ? t1 : t2;
}
```

Exemple 3 :

```
template <class X, class Y>
int valeur(X x, Y y)
{  // ... }
```

Les fonctions génériques peuvent, tout comme les fonctions normales, être surchargées.

Ainsi il est possible de spécialiser une fonction à une situation particulière, comme dans l'exemple qui suit :

```
template <class T>
T min(T t1, T t2) {
  return t1 < t2 ? t1 : t2;
}

char *min(char *s1, char *s2) {
  // fonction spécialisée pour les chaînes
  // il est évident la comparaison des adresses
  // des chaînes est absurde ...
  return strcmp(s1, s2)<0 ? s1 : s2;
}

void main() {
  cout << min(10, 20) << "  " << min("Alain", "Bjarne") << endl;
}
```

6.2 Les classes paramétrées

Les classes paramétrées permettent de créer des classes générales et de transmettre des types comme paramètres à ces classes pour construire une classe spécifique.

6.2.1 Définition d'un modèle de classe

Le mot clé template suivi des paramètres du type du modèle débute la définition de la classe modèle.

Exemple :

La classe *List* n'a pas à se soucier du type réel de ses éléments.

Elle ne doit s'occuper que d'ajouter, supprimer, trier ses éléments.

On a donc intérêt à paramétrer la classe List par le type des éléments qu'elle stocke.

```
template <class T> class List {
  public:
    List();              // constructeur
    List(const List &l); // constructeur de copie
    ~List();
    void add(T elmt);    // ajoute un élément à la liste
    T &get();            // retourne l'élément courant
    // ...
  private:
    // ...
```

```
};
```

La définition des méthodes se fait de la façon suivante :

```
template <class T> void List<T>::add(T elemt) {
  /*...*/
}
template <class T> T &List<T>::get() {
  /*...*/
}
```

Une utilisation de cette classe sera, par exemple :

```
void main() {
  List<int> resultat;
  List<Employe> repertoire;
  ...
```

7

L'héritage

Sommaire

7.1 L'héritage simple

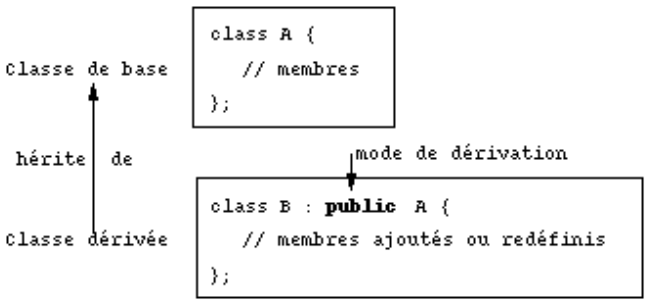
L'héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d'une classe déjà existante, textbfla classe de base (ou super classe).

"**Il est plus facile de modifier que de réinventer**"

La nouvelle classe (ou **classe dérivée** ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et ainsi réutiliser le code déjà écrit pour la classe de base.

On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

Syntaxe :



La classe B hérite de façon publique de la classe A.

Tous les membres publics ou protégés de la classe A font partis de l'interface de la classe B.

7.2 Mode de dérivation

Lors de la définition de la classe dérivée il est possible de spécifier le **mode de dérivation** par l'emploi d'un des mots-clé suivants :

public, *protected* ou *private*.

Ce mode de dérivation détermine quels membres de la classe de base sont accessibles dans la classe dérivée.

Au cas où aucun mode de dérivation n'est spécifié, le compilateur C++ prend par défaut le mot-clé *private* pour une classe et public pour une structure.

Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

7.2.1 Héritage public

Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.

C'est la forme la plus courante d'héritage, car il permet de modéliser les relations "Y est une sorte de X" ou "Y est une spécialisation de la classe de base X".

Exemple :

```
class Vehicule {
  public:
    void pub1();
  protected:
    void prot1();
  private:
    void priv1();
};

class Voiture : public Vehicule {
  public:
    int pub2() {
      pub1();      // OK
      prot1();     // OK
      priv1();     // ERREUR
    }
};

Voiture safrane;
safrane.pub1();  // OK
safrane.pub2();  // OK
```

7.2.2 Héritage privé

Il donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée.

Il permet de modéliser les relations "Y est composé de un ou plusieurs X".

Plutôt que d'hériter de façon privée de la classe de base X, on peut faire de la classe de base une donnée membre (composition).

Exemple :

```
class String {
  public:
    int length();
  // ...
};

class Telephone_number : private String {
   void f1() {
     // ...
     l = length();  // OK
};

Telephone_number tn;
cout << tn.length();      // ERREUR
```

7.2.3 Héritage protégé

Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée.

L'héritage fait partie de l'interface mais n'est pas accessible aux utilisateurs.

Exemple :

```
class String {
  protected:
    int n;
};

class Telephone_number : protected String {
  protected:
    void f2() { n++; }  // OK
};

class Local_number : public Telephone_number {
  protected:
    void f3() { n++; } // OK
};
```

Tableau résumé de l'accès aux membres

		Statut dans la classe de base	Statut dans la classe dérivée
mode de dérivation	public	public	public
		protected	protected
		private	inaccessible
	protected	public	protected
		protected	protected
		private	inaccessible
	private	public	private
		protected	private
		private	inaccessible

7.3 Redéfinition de méthodes dans la classe dérivée

On peut redéfinir une fonction dans une classe dérivée si on lui donne le même nom que dans la classe de base.

Il y aura ainsi, comme dans l'exemple ci après, deux fonctions f2(), mais il sera possible de les différencier avec l'opérateur :: de résolution de portée.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    int xxx;
};

class Y : public X {
  public:
    void f2();
    void f3();
};

void Y::f3() {
  X::f2();     // f2 de la classe X
  X::xxx = 12; // accès au membre xxx de la classe X
  f1();        // appel de f1 de la classe X
  f2();        // appel de f2 de la classe Y
}
```

7.4 Ajustement d'accès

Lors d'un héritage protégé ou privé, nous pouvons spécifier que certains membres de la classe ancêtre conservent leur mode d'accès dans la classe dérivée.

Ce mécanisme, appelé déclaration d'accès, ne permet en aucun cas d'augmenter ou de diminuer la visibilité d'un membre de la classe de base.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    void f3();
    void f4();
};

class Y : private X {
  public:
    X::f1; // f1() reste public dans Y
    X::f3; // ERREUR: un membre protégé ne peut pas devenir public
  protected:
    X::f4; // f3() reste protégé dans Y
    X::f2; // ERREUR: un membre public ne peut pas devenir protégé
};
```

7.5 Héritage des constructeurs/destructeurs

Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.

Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.

Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une **liste d'initialisation**.

L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

Exemple 1 :

```
class Vehicule {
  public:
    Vehicule() { cout<< "Vehicule" << endl; }
    ~Vehicule() { cout<< "~Vehicule" << endl; }
};

class Voiture : public Vehicule {
  public:
    Voiture() { cout<< "Voiture" << endl; }
    ~Voiture() { cout<< "~Voiture" << endl; }
};

void main() {
  Voiture *R21 = new Voiture;
  // ...
  delete R21;
```

```
}
/********** se programme affiche :
Vehicule
Voiture
~Voiture
~Vehicule
*********************************************/
```

Exemple d'appel des constructeurs avec paramètres :

```
class Vehicule {
  public:
     Vehicule(char *nom, int places);
     //...
};

class Voiture : public Vehicule {
  private:
    int _cv;    // puissance fiscale
  public:
    Voiture(char *n, int p, int cv);
    // ...
};

Voiture::Voiture(char *n, int p, int cv): Vehicule(n, p), _cv(cv)
{ /* ... */ }
```

7.6 Héritage et amitié

L'amitié pour une classe s'hérite, mais uniquement sur les membres de la classe hérités, elle ne se propage pas aux nouveaux membres de la classe dérivée et ne s'étend pas aux générations suivantes.

Exemple :

```
class A {
  friend class test1;
  public:
  A( int n= 0): _a(n) {}
  private:
  int _a;
  };

class test1 {
  public:
  test( int n= 0): a0(n) {}
  void affiche1() {
    cout << a0._a << // OK: test1 est amie de A
    }
  private:
  A a0;
  };

class test2: public test {
```

```
  public:
  test2( int z0= 0, int z1= 0): test( z0), a1( z1) {}
  void Ecrit() {
    cout << a1._a; // ERREUR: test2 n'est pas amie de A
    }
  private:
  A a1;
  };
```

L'amitié pour une fonction ne s'hérite pas.

A chaque dérivation, vous devez redéfinir les relations d'amitié avec les fonctions.

7.7 Conversion de type dans une hiérarchie de classes

Il est possible de convertir implicitement une instance d'une classe dérivée en une instance de la classe de base si l'héritage est public.

L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres de la classe dérivée.

Exemple :

```
class Vehicule {
  public:
    void f1();
  // ...
};

class Voiture : public Vehicule {
  public:
    int f1();
  // ...
};

void traitement1(Vehicule v) {
  // ...
  v.f1();   // OK
  // ...
}

void main() {
  Voiture R25;
  traitement1( R25 );
}
```

De la même façon on peut utiliser des pointeurs :

Un pointeur (ou une référence) sur un objet d'une classe dérivée peut être implicitement converti en un pointeur (ou une référence) sur un objet de la classe de base.

Cette conversion n'est possible que si l'héritage est public, car la classe de base doit posséder des membres public accessibles (ce n'est pas le cas d'un héritage *protected* ou *private*).

C'est le type du pointeur qui détermine laquelle des méthodes f1() est appelée.

```
void traitement1(Vehicule *v) {
  // ...
  v->f1();   // OK
  // ...
}

void main() {
  Voiture R25;
  traitement1( &R25 );
}
```

7.8 Héritage multiple

En langage C++, il est possible d'utiliser l'héritage multiple.

Il permet de créer des classes dérivées à partir de plusieurs classes de base.

Pour chaque classe de base, on peut définir le mode d'héritage.

```
class A {
  public:
    void fa() { /* ... */ }
  protected:
    int _x;
};

class B {
  public:
    void fb() { /* ... */ }
  protected:
    int _x;
};

class C: public B, public A {
  public:
    void fc();
};

void C::fc() {
  int i;
  fa();
  i = A::_x + B::_x; // résolution de portée pour lever l'ambiguïté
}
```

7.8.1 Ordre d'appel des constructeurs

Dans l'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.

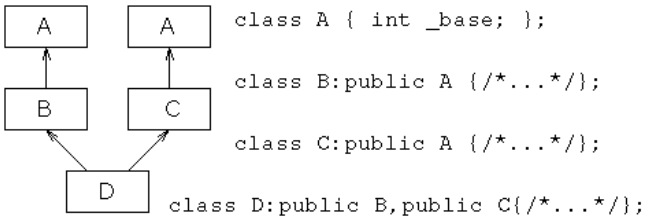
Dans l'exemple suivant, le constructeur par défaut de la classe C appelle le constructeur par défaut de la classe B, puis celui de la classe A et en dernier lieu le constructeur de la classe dérivée, même si une liste d'initialisation existe.

```
class A {
```

```
  public:
    A(int n=0) { /* ... */ }
  // ...
};

class B {
  public:
    B(int n=0) { /* ... */ }
  // ...
};

class C: public B, public A {
//       ^^^^^^^^^^^^^^^^^^
//       ordre d'appel des constructeurs des classes de base
//
  public:
    C(int i, int j) : A(i) , B(j) { /* ... */ }
  // ...
};

void main() {
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs.

7.9 Héritage virtuel



Un objet de la classe D contiendra deux fois les données héritées de la classe de base A, une fois par héritage de la classe B et une autre fois par C.

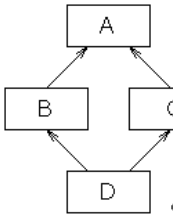
Il y a donc deux fois le membre *_base* dans la classe D.

L'accès au membre _base de la classe A se fait en levant l'ambiguïté.

```
void main() {
  D od;
  od._base = 0;    // ERREUR, ambiguïté
  od.B::_base = 1; // OK
  od.C::_base = 2; // OK
}
```

Il est possible de n'avoir qu'une occurrence des membres de la classe de base, en utilisant l'**héritage virtuel**.

Pour que la classe D n'hérite qu'une seule fois de la classe A, il faut que les classes B et C héritent virtuellement de A.




```
class A { int _base; };
class B:virtual public A { };
class C:virtual public A { };
class D:public B,public C{/*...*/};
```

Permet de n'avoir dans la classe D qu'une seule occurrence des données héritées de la classe de base A.

```
void main() {
  D od;
  od._base = 0; // OK, pas d'ambiguïté
}
```

Note : Il ne faut pas confondre ce statut virtuaŻ de déclaration d'héritage des classes avec celui des membres virtuels que nous allons étudier. Ici, Le mot-clé *virtual* précise au compilateur les classes à ne pas dupliquer.

7.10 Polymorphisme

L'**héritage** nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de votre application.

Le **polymorphisme** rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes.

En C++, le polymorphisme est mis en oeuvre par l'utilisation des **fonctions virtuelles**.

7.10.1 Fonctions virtuelles

```
class ObjGraph {
 public:
   void print() const { cout <<"ObjGraph::print()"; }
};

class Bouton: public ObjGraph {
  public:
    void print() const { cout << "Bouton::print()"; }
};

class Fenetre: public ObjGrap {
  public:
    void print() const { cout << "Fenetre::print()"; }
};

void traitement(const ObjGraph &og) {
  // ...
  og.print();
  // ...
```

```
}

void main() {             // Qu'affiche ce programme ???
  Bouton OK;
  Fenetre windows97;
  traitement(OK);         // affichage de .....
  traitement(Window97);   // affichage de .....
}
```

Comme nous l'avons déjà vu, l'instruction *og.print()* de *traitement()* appelera la méthode *print()* de la classe *ObjGraph*.

La réponse est donc :

```
  traitement(OK);          // affichage de ObjGraph::print()
  traitement(Window97);    // affichage de ObjGraph::print()
}
```

Si dans la fonction *traitement()* nous voulons appeler la méthode *print()* selon la classe à laquelle appartient l'instance, nous devons définir, dans la classe de base, la méthode *print()* comme étant **virtuelle** :

```
class ObjGraph {
  public:
    // ...
    virtual void print() const {
       cout<< "ObjetGraphique::print()" << endl;}
};
```

Pour plus de clarté, le mot-clé *virtual* peut être répété devant les méthodes *print()* des classes *Bouton* et *Fenetre* :

```
class Bouton: public ObjGraph {
  public:
    virtual void print() const {
      cout << "Bouton::print()";
    }
};

class Fenetre: public ObjGrap {
  public:
    virtual void print() const {
      cout << "Fenetre::print()";
    }
};
```

On appelle ce comportement, le *polymorphisme*.

Lorsque le compilateur rencontre une méthode virtuelle, il sait qu'il faut attendre l'exécution pour déterminer la bonne méthode à appeler.

7.10.2 Destructeur virtuel

Il ne faut pas oublier de définir le destructeur comme "virtual" lorsque l'on utilise une méthode virtuelle :

```
class ObjGraph {
```

```
  public:
    //...
    virtual ~ObjGraph() { cout << "fin de ObjGraph\n"; }
};
class Fenetre : public ObjGraph {
  public:
    // ...
    ~Fenetre() { cout << "fin de Fenêtre  "; }
};

void main() {
  Fenetre *Windows97 = new Fenetre;
  ObjGraph *og = Windows97;
  // ...
  delete og;  // affichage de :    fin de Fenêtre  fin de ObjGraph
              // si le destructeur n'avait pas été virtuel,
              // l'affichage aurait été :          fin de ObjGraph
}
```

Notes :

- Un constructeur, par contre, ne peut pas être déclaré comme virtuel.
- Une méthode statique ne peut, non plus, être déclaré comme virtuelle.
- Lors de l'héritage, le statut de l'accessibilité de la méthode virtuelle (public, protégé ou privé) est conservé dans toutes les classes dérivée, même si elle est redéfinie avec un statut différent. Le statut de la classe de base prime.

7.11 Classes abstraites

Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de cadre générique pour les méthodes virtuelles des classes dérivées. Ceci permet de garantir une bonne homogénéité de votre architecture de classes.

Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure.

On ne peut pas créer d'instance d'une classe abstraite et une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction.

Par contre, les pointeurs et les références sur une classe abstraite sont parfaitement légitimes et justifiés.

7.11.1 Méthode virtuelle pure

une telle méthode se déclare en ajoutant un $= \mathbf{0}$ à la fin de sa déclaration.

```
class ObjGraph {
  public:
    virtual void print() const = 0;
};

void main() {
  ObjGraph og;  // ERREUR
  // ...
}
```

Notes :

- On ne peut utiliser une classe abstraite qu'à partir d'un pointeur ou d'une référence.
- Contrairement à une méthode virtuelle "normale", une méthode virtuelle pure n'est pas obligé de fournir une définition pour ObjGraph::print().
- Une classe dérivée qui ne redéfinit pas une méthode virtuelle pure est elle aussi abstraite.

```
class ObjGraph {
  public:
    virtual void print() const = 0;
};

void main() {
  ObjGraph og;  // ERREUR
  // ...
}
```

8 Les exceptions

Sommaire

8.1 Généralités

Le langage C++ nous propose une gestion efficace des erreurs pouvant survenir pendant l'exécution :

- erreurs matérielles : saturation mémoire, disque plein ...
- erreurs logicielles : division par zéro ...

La solution habituellement pratiquée est l'affichage d'un message d'erreur et la sortie du programme avec le renvoi d'un code d'erreur au programme appelant.

Exemple :

```
void lire_fichier(const char *nom) {
  ifstream f_in(nom); // ouverture en lecture
  if ( ! f_in.good()) {
    cerr << "Problème à l'ouverture du fichier " << nom << endl;
    exit(1);
  }
  // lecture du fichier ...
}
```

En C++, une nouvelle structure de contrôle permet donc la gestion des erreurs d'exécution, la structure try ... catch.

8.2 Schéma du mécanisme d'exception

Le mécanisme mis en œuvre pour une exception est les suivant :

1. constuction d'un objet (d'un type quelconque) qui représente l'erreur
2. lancement de l'exception (throw)
3. l'exception est alors propagée dans la structure de contrôle try ... catch englobante
4. cette structure essaye attraper (catch) l'objet

5. si elle n'y parvient pas, la fonction terminate() est appelée.

```
try {
  // ...
  throw objet  // lancement de l'exception
  }
catch ( type ) {
  // traitement de l'erreur
  }
```

8.3 La structure *try...catch*

Il faut noter que :

- Quand une exception est détectée dans un bloc try, le contrôle de l'exécution est donné au bloc catch correspondant au type de l'exception (s'il existe).
- Un bloc try doit être suivi d'au moins un bloc catch
- Si plusieurs blocs catch existent, ils doivent intercepter un type d'exception différent.
- Quand une exception est détectée, les destructeurs des objets inclus dans le bloc try sont appelés avant d'appeler un bloc catch.
- A la fin du bloc catch, le programme continue son exécution sur l'instruction qui suit le dernier bloc catch.

Exemple : interception d'une exception de type bad_alloc :

Cette exception est lancée en cas d'echec d'allocation mémoire.

```
#include <exception>

// ...
try {
  char *ptr = new char[1000000000];  /
  // ... suite en cas de succès de new (improbable ...)
}
catch ( bad_alloc ) {
  // en cas d'échec d'allocation mémoire par new
  // une exception bad_alloc est lancée par new

  // traitement de l'erreur d'allocation
}
```

8.4 Syntaxe du *catch*

catch permet d'intercepter une exception et de la traiter :

- catch (TYPE) : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées.
- catch (TYPE o) : intercepte les exceptions du type TYPE, ainsi que celles de ses classes dérivées. Dans le catch un objet o est utilisable pour extraire d'autres informations sur l'exception.
- catch (...) : intercepte les exceptions de tous types, non traitées par les bocs catch précédents.

Exemple :

```
class Erreur {
  // ...
  int get_erreur() { /* ... */ }
};

try {
  // instruction(s) douteuse(s) qui peu(ven)t lancer une exception
}
catch ( xalloc ) {
  // ...
}
catch ( Erreur err ) {
  cerr << err.get_erreur() << endl;
  // ...
}
catch (...) {
  // ...
}
```

8.5 Syntaxe de *throw*

throw permet de lever une exception :

- throw obj : elle permet de lever une exception donnée
  Exemple :

```
#include <iostream>

using namespace std;

class Essai {
  public :
  class Erreur {  };
  // ...
  void f1() {
    throw Erreur(); // construction d'une instance de Erreur
    // et lancement de celle-ci
    }
  };

int main() {
  try {
    Essai e1;
    e1.f1();
    }
  catch ( Essai::Erreur ) {
    cout << "interception de Erreur" << endl;
    }
  return 0;
  }
```

- throw : l'instruction *throw* sans paramètre s'utilise si l'on ne parvient pas à resoudre une exception dans un bloc catch. Elle permet de relancer la dernière exception.

8.6 Déclaration des exceptions levées par une fonction

Cette déclaration permet de spécifier le type des exceptions pouvant être éventuellement levées par votre fonction (ou méthode).

Exemples de déclaration :

- *void f1() throw (Erreur);*
- *void f2() throw (Erreur, Division_zero);*
- *void f3() throw ();*
- *void f4();*

Remarques :

- Par défaut de déclaration (comme dans le cas de la fonction *f4()*), toute exception peut être lancée par une fonction.
- La fonction *f3()*, déclarée ci dessus, ne peut lancer aucune exception.
- La fonction *f1()* ne peut lancer que des exceptions de la classe Erreur (ou de classes dérivées).
- Si une fonction lance une exception non déclarée dans son entête, la fonction *unexpected()* est appelée.
- A la définition de la fonction, *throw* et ses paramètres doivent être de nouveau spécifié.

8.7 La fonction *terminate()*

Si aucun bloc catch ne peut attraper l'exception lancée, la fonction *terminate()* est alors appelée. Par défaut elle met fin au programme par un appel à la fonction *abort()*.

On peut définir sa propre fonction *terminate()* par un appel à la fonction *set_terminate()* définie dans *except.h* comme :

```
typedef void (*terminate_handler)();
terminate_function set_terminate(terminate_handler t_func);
```

Exemple :

```
#include <exception>
#include <iostream>

using namespace std;

class Erreur{};

void my_terminate() {
  cout << "my_terminate" << endl;
  exit(1);  // elle ne doit pas retourner à son appelant
}

int main() {
  try {
     set_terminate((terminate_handler) my_terminate);
     throw;
     // ...
  }
  catch ( Erreur ) {
    // ...
```

```
  }
  return 0;
}
```

8.8 La fonction *unexpected()*

Si une fonction (ou une méthode) lance une exception qui n'est pas déclarée par un *throw* dans son entête, la fonction *unexpected()* est alors appelée. Par défaut elle met fin au programme par un appel à la fonction *abort()*.

On peut définir sa propre fonction *unexpected()* par un appel à la fonction *set_unexpected()* définie dans *except.h* comme :

```
typedef void (*unexpected_handler)();
unexpected_function set_unexpected(unexpected_handler t_func);
```

Exemple :

```
#include <exception>
#include <iostream>

using namespace std;

void my_unexpected() {
  cout << "my_unexpected" << endl;
  exit(1);   // elle ne doit pas retourner à son appelant
}

class Erreur {};

class Toto {};

class Essai {
  public:
    void f1() throw (Erreur);
};

void Essai::f1() throw (Erreur) {
  throw Toto();
}

int main() {
  try {
     set_unexpected( (unexpected_handler) my_unexpected);
     Essai e1;
     e1.f1();
  }
  catch ( Erreur ) {
    // ...
  }
  return 0;
}
```

8.9 Exemple complet

```
#include <exception>
#include <iostream>

using namespace std;

class Essai {
  public :
    class Erreur {
      public:
        Erreur(int n=0): _val(n) { }
        int get_val() { return _val; }
      private:
        int _val;
    };

    Essai() { cout << "Constructeur d'Essai" << endl; }
    ~Essai() { cout << "destructeur d'Essai" << endl; }
    // ...
    void f1() {
      throw Erreur(10); // construction d'une instance de Erreur
                        // initialiée à 10 et lancement de celle-ci
  }
};

int main() {
  try {
    Essai e1;
    e1.f1();
    cout << "bla bla bla" << endl;    //
  }
  catch ( Essai::Erreur e ) {
    cout << "Erreur numéro : " << e.get_val() << endl;
  }
  return 0;
}
/* résultat de l'exécution ***********
Constructeur d'Essai
destructeur d'Essai
Erreur numéro : 10
*********************************/
```

9 Les espaces de nommage

Sommaire

9.1 Introduction

Les espaces de nommage sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur. Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet. Par exemple, si deux programmeurs définissent différemment une même structure dans deux fichiers différents, un conflit entre ces deux structures aura lieu au mieux à l'édition de liens, et au pire lors de l'utilisation commune des sources de ces deux programmeurs. Ce type de conflit provient du fait que le C++ ne fournit qu'un seul espace de nommage de portée globale, dans lequel il ne doit y avoir aucun conflit de nom. Grâce aux espaces de nommage non globaux, ce type de problème peut être plus facilement évité, parce que l'on peut éviter de définir les objets globaux dans la portée globale.

9.2 Définition des espaces de nommage

9.2.1 Espaces de nommage nommées

Lorsque le programmeur donne un nom à un espace de nommage, celui-ci est appelé un espace de nommage nommé. La syntaxe de ce type d'espace de nommage est la suivante :

```
namespace nom
{
    déclarations | définitions
}
```

nom est le nom de l'espace de nommage, et déclarations et définitions sont les déclarations et les définitions des identificateurs qui lui appartiennent.

Contrairement aux régions déclaratives classiques du langage (comme par exemple les classes), un namespace peut être découpé en plusieurs morceaux. Le premier morceaux sert de déclaration, et les suivants d'extensions. La syntaxe pour une extension d'espace de nommage est exactement la même que celle de la partie de déclaration.

Exemple : Extension de namespace

```
namespace A   // Déclaration de l'espace de nommage A.
{
   int i;
```

```
}

namespace B   // Déclaration de l'espace de nommage B.
{
    int i;
}

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

Les identificateurs déclarés ou définis à l'intérieur d'un même espace de nommage ne doivent pas entrer en conflit. Ils peuvent avoir les mêmes noms, mais seulement dans le cadre de la surcharge. Un espace de nommage se comporte donc exactement comme les zones de déclaration des classes et de la portée globale.

L'accès aux identificateurs des espaces de nommage se fait par défaut grâce à l'opérateur de résolution de portée (::), et en qualifiant le nom de l'identificateur à utiliser du nom de son espace de nommage. Cependant, cette qualification est inutile à l'intérieur de l'espace de nommage lui-même, exactement comme pour les membres des classes à l'intérieur de leur classe.

Exemple : Accès aux membres d'un namespace

```
int i=1;    // i est global.

namespace A
{
    int i=2; // i de l'espace de nommage A.
    int j=i; // Utilise A::i.
}

int main(void)
{
    i=1;     // Utilise ::i.
    A::i=3;  // Utilise A::i.
    return 0;
}
```

Les fonctions membres d'un espace de nommage peuvent être définies à l'intérieur de cet espace, exactement comme les fonctions membres de classes. Elles peuvent également être définies en dehors de cet espace, si l'on utilise l'opérateur de résolution de portée. Les fonctions ainsi définies doivent apparaître après leur déclaration dans l'espace de nommage.

Exemple : Définition externe d'une fonction de namespace

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

int A::f(void)     // Définition de A::f.
{
    return 0;
}
```

Il est possible de définir un espace de nommage à l'intérieur d'un autre espace de nommage. Cependant, cette déclaration doit obligatoirement avoir lieu au niveau déclaratif le plus externe de l'espace de nommage qui contient le sous-espace de nommage. On ne peut donc pas déclarer d'espaces de nommage à l'intérieur d'une fonction ou à l'intérieur d'une classe.

Exemple : Définition de namespace dans un namespace

```
namespace Conteneur
{
    int i;              // Conteneur::i.
    namespace Contenu
    {
        int j;          // Conteneur::Contenu::j.
    }
}
```

9.2.2 Espaces de nommage anonymes

Lorsque, lors de la déclaration d'un espace de nommage, aucun nom n'est donné, un espace de nommage anonyme est créé. Ce type d'espace de nommage permet d'assurer l'unicité du nom de l'espace de nommage ainsi déclaré. Les espaces de nommage anonymes peuvent donc remplacer efficacement le mot clé static pour rendre unique des identificateurs dans un fichier. Cependant, elles sont plus puissantes, parce que l'on peut également déclarer des espaces de nommage anonymes à l'intérieur d'autres espaces de nommage.

Exemple : Définition de namespace anonyme

```
namespace
{
    int i;          // Équivalent à unique::i;
}
```

Dans l'exemple précédent, la déclaration de i se fait dans un espace de nommage dont le nom est choisi par le compilateur de manière unique. Cependant, comme on ne connaît pas ce nom, le compilateur utilise une directive using (voir plus loin) afin de pouvoir utiliser les identificateurs de cet espace de nommage anonyme sans préciser leur nom complet avec l'opérateur de résolution de portée.

Si, dans un espace de nommage, un identificateur est déclaré avec le même nom qu'un autre identificateur déclaré dans un espace de nommage plus global, l'identificateur global est masqué. De plus, l'identificateur ainsi défini ne peut être accédé en dehors de son espace de nommage que par un nom complètement qualifié à l'aide de l'opérateur de résolution de portée. Toutefois, si l'espace de nommage dans lequel il est défini est un espace de nommage anonyme, cet identificateur ne pourra pas être référencé, puisqu'on ne peut pas préciser le nom des espaces de nommage anonymes.

Exemple : Ambiguïtés entre namespaces

```
namespace
{
    int i;          // Déclare unique::i.
}

void f(void)
{
    ++i;            // Utilise unique::i.
```

```
}

namespace A
{
    namespace
    {
        int i;     // Définit A::unique::i.
        int j;     // Définit A::unique::j.
    }

    void g(void)
    {
        ++i;       // Erreur : ambiguïté entre unique::i
                   // et A::unique::i.
        ++A::i;    // Erreur : A::i n'est pas défini
                   // (seul A::unique::i l'est).
        ++j;       // Correct : ++A::unique::j.
    }
}
```

9.2.3 Alias d'espaces de nommage

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;
```

nom_alias est ici le nom de l'alias de l'espace de nommage, et nom est le nom de l'espace de nommage lui-même.

Les noms donnés aux alias d'espaces de nommage ne doivent pas entrer en conflit avec les noms des autres identificateurs du même espace de nommage, que celui-ci soit l'espace de nommage de portée globale ou non.

9.3 Déclaration using

Les déclarations using permettent d'utiliser un identificateur d'un espace de nommage de manière simplifiée, sans avoir à spécifier son nom complet (c'est-à-dire le nom de l'espace de nommage suivi du nom de l'identificateur).

9.3.1 Syntaxe des déclarations using

La syntaxe des déclarations using est la suivante :

```
using identificateur;
```

où identificateur est le nom complet de l'identificateur à utiliser, avec qualification d'espace de nommage.

Exemple : Déclaration using

```
namespace A
```

```
{
    int i;        // Déclare A::i.
    int j;        // Déclare A::j.
}

void f(void)
{
    using A::i;   // A::i peut être utilisé sous le nom i.
    i=1;          // Équivalent à A::i=1.
    j=1;          // Erreur ! j n'est pas défini !
    return ;
}
```

Les déclarations using permettent en fait de déclarer des alias des identificateurs. Ces alias doivent être considérés exactement comme des déclarations normales. Cela signifie qu'ils ne peuvent être déclarés plusieurs fois que lorsque les déclarations multiples sont autorisées (déclarations de variables ou de fonctions en dehors des classes), et de plus ils appartiennent à l'espace de nommage dans lequel ils sont définis.

Exemple : Déclarations using multiples

```
namespace A
{
    int i;
    void f(void)
    {
    }
}

namespace B
{
    using A::i;  // Déclaration de l'alias B::i, qui représente A::i.
    using A::i;  // Légal : double déclaration de A::i.

    using A::f;  // Déclare void B::f(void),
                 // fonction identique à A::f.
}

int main(void)
{
    B::f();      // Appelle A::f.
    return 0;
}
```

L'alias créé par une déclaration using permet de référencer uniquement les identificateurs qui sont visibles au moment où la déclaration using est faite. Si l'espace de nommage concerné par la déclaration using est étendu après cette dernière, les nouveaux identificateurs de même nom que celui de l'alias ne seront pas pris en compte.

Exemple : Extension de namespace après une déclaration using

```
namespace A
{
    void f(int);
}
```

```
using A::f;            // f est synonyme de A::f(int).

namespace A
{
    void f(char);      // f est toujours synonyme de A::f(int),
                       // mais pas de A::f(char).
}

void g()
{
    f('a');            // Appelle A::f(int), même si A::f(char)
                       // existe.
}
```

Si plusieurs déclarations locales et using déclarent des identificateurs de même nom, ou bien ces identificateurs doivent tous se rapporter au même objet, ou bien ils doivent représenter des fonctions ayant des signatures différentes (les fonctions déclarées sont donc surchargées). Dans le cas contraire, des ambiguïtés peuvent apparaître et le compilateur signale une erreur lors de la déclaration using.

Exemple : Conflit entre déclarations using et identificateurs locaux

```
namespace A
{
    int i;
    void f(int);
}

void g(void)
{
    int i;        // Déclaration locale de i.
    using A::i;   // Erreur : i est déjà déclaré.
    void f(char); // Déclaration locale de f(char).
    using A::f;   // Pas d'erreur, il y a surcharge de f.
    return ;
}
```

Note : Ce comportement diffère de celui des directives using. En effet, les directives using reportent la détection des erreurs à la première utilisation des identificateurs ambigus.

9.3.2 Utilisation des déclarations using dans les classes

Une déclaration using peut être utilisée dans la définition d'une classe. Dans ce cas, elle doit se rapporter à une classe de base de la classe dans laquelle elle est utilisée. De plus, l'identificateur donné à la déclaration using doit être accessible dans la classe de base (c'est-à-dire de type protected ou public).

Exemple : Déclaration using dans une classe

```
namespace A
{
    float f;
}
```

```
class Base
{
    int i;
public:
    int j;
};

class Derivee : public Base
{
    using A::f;      // Illégal : f n'est pas dans une classe
                     // de base.
    using Base::i;   // Interdit : Derivee n'a pas le droit
                     // d'utiliser Base::i.
public:
    using Base::j;   // Légal.
};
```

Dans l'exemple précédent, seule la troisième déclaration est valide, parce que c'est la seule qui se réfère à un membre accessible de la classe de base. Le membre j déclaré sera donc un synonyme de Base::j dans la classe Derivee.

En général, les membres des classes de base sont accessibles directement. Quelle est donc l'utilité des déclarations using dans les classes ? En fait, elles peuvent être utilisées pour rétablir les droits d'accès, modifiés par un héritage, à des membres de classes de base. Pour cela, il suffit de placer la déclaration using dans une zone de déclaration du même type que celle dans laquelle le membre se trouvait dans la classe de base. Cependant, comme on l'a vu ci-dessus, une classe ne peut pas rétablir les droits d'accès d'un membre de classe de base déclaré en zone private.

Exemple : Rétablissement de droits d'accès à l'aide d'une directive using

```
class Base
{
public:
    int i;
    int j;
};

class Derivee : private Base
{
public:
    using Base::i;  // Rétablit l'accessibilité sur Base::i.
protected:
    using Base::i;  // Interdit : restreint l'accessibilité
                    // sur Base::i autrement que par héritage.
};
```

Note : Certains compilateurs interprètent différemment la section 9.4 de la norme C++, qui concerne l'accessibilité des membres introduits avec une déclaration using. Selon eux, les déclarations using permettent de restreindre l'accessibilité des droits et non pas de les rétablir. Cela implique qu'il est impossible de redonner l'accessibilité à des données pour lesquelles l'héritage a restreint l'accès. Par conséquent, l'héritage doit être fait de la manière la plus permissive possible, et les accès doivent être ajustés au cas par cas. Bien que cette interprétation soit tout à fait valable, l'exemple donné dans la norme C++ semble indiquer qu'elle n'est pas correcte.

Quand une fonction d'une classe de base est introduite dans une classe dérivée à l'aide d'une déclaration using, et qu'une fonction de même nom et de même signature est définie dans la classe

dérivée, cette dernière fonction surcharge la fonction de la classe de base. Il n'y a pas d'ambiguïté dans ce cas.

9.4 Directive using

La directive using permet d'utiliser, sans spécification d'espace de nommage, non pas un identificateur comme dans le cas de la déclaration using, mais tous les identificateurs de cet espace de nommage.

La syntaxe de la directive using est la suivante :

```
using namespace nom;
```

où nom est le nom de l'espace de nommage dont les identificateurs doivent être utilisés sans qualification complète.

Exemple : Directive using

```
namespace A
{
    int i;        // Déclare A::i.
    int j;        // Déclare A::j.
}

void f(void)
{
    using namespace A; // On utilise les identificateurs de A.
    i=1;          // Équivalent à A::i=1.
    j=1;          // Équivalent à A::j=1.
    return ;
}
```

Après une directive using, il est toujours possible d'utiliser les noms complets des identificateurs de l'espace de nommage, mais ce n'est plus nécessaire. Les directives using sont valides à partir de la ligne où elles sont déclarées jusqu'à la fin du bloc de portée courante. Si un espace de nommage est étendu après une directive using, les identificateurs définis dans l'extension de l'espace de nommage peuvent être utilisés exactement comme les identificateurs définis avant la directive using (c'est-à-dire sans qualification complète de leurs noms).

Exemple : Extension de namespace après une directive using

```
namespace A
{
    int i;
}

using namespace A;

namespace A
{
    int j;
}

void f(void)
{
```

```
    i=0;    // Initialise A::i.
    j=0;    // Initialise A::j.
    return ;
}
```

Il se peut que lors de l'introduction des identificateurs d'un espace de nommage par une directive using, des conflits de noms apparaissent. Dans ce cas, aucune erreur n'est signalée lors de la directive using. En revanche, une erreur se produit si un des identificateurs pour lesquels il y a conflit est utilisé.

Exemple : Conflit entre directive using et identificateurs locaux

```
namespace A
{
    int i;  // Définit A::i.
}

namespace B
{
    int i;  // Définit B::i.
    using namespace A;   // A::i et B::i sont en conflit.
                         // Cependant, aucune erreur n'apparaît.
}

void f(void)
{
    using namespace B;
    i=2;    // Erreur : il y a ambiguïté.
    return ;
}
```

Table des matières