Jacquelin Charbonnel

Langage C++

la proposition de standard ANSI / ISO expliquée

MANUELS INFORMATIQUES MASSON

MASSON

Langage C++

La proposition de standard ANSI / ISO expliquée

CHEZ LE MÊME ÉDITEUR

Du même auteur

LANGAGE C. LES FINESSES D'UN LANGAGE REDOUTABLE, par J. CHARBONNEL. *Collection U-Informatique.* 1992, 200 pages.

Dans la même collection

COMPRENDRE ET UTILISER C++ POUR PROGRAMMER OBJET, par G. CLAVEL, I. TRILLAUD, L. VEILLON. 1994, 248 pages.

LE DÉVELOPPEMENT DE LOGICIEL EN C++, par R. WINDER. Traduit de la 2e édition anglaise par P.-Y. BONNETAIN. 1994, 568 pages.

LE LANGAGE C, NORME ANSI, par B.W. KERNIGHAN et D.M. RITCHIE. Traduit de l'anglais par J.F. GROFF et E. MOTTIER. 1994, 2e édition, 4e tirage, 296 pages.

LE LANGAGE C. Solutions aux exercices de la 2e édition de l'ouvrage de KERNIGHAN et RICHIE, par C.L. TONDO, S. E GIMPEL. Traduit de l'anglais par A. BERTIER. 1992, 2e édition, 2e tirage, 168 pages.

LANGAGE C, NORME ANSI. Vers une approche orientée objet, par Ph. DRIX. 1994, 2e édition revue et augmentée, 440 pages.

MÉTHODOLOGIE DE LA PROGRAMMATION EN LANGAGE C. Principes et applications, par J.-P. BRAQUELAIRE. 1995, 2e édition, 2e tirage, 528 pages.

Autres ouvrages

C++, par B. BEAUDOING et D. EDELSON. Préface de Ph. GAUTRON. *Collection Objectif.* 1994, 196 pages.

L'APPROCHE OBJET. CONCEPTS ET TECHNIQUES, par R. MOREAU. *Collection Méthodes Informatiques et Pratique des Systèmes.* 1994, 312 pages.

RÉUTILISATION DU LOGICIEL, par B. COULANGE. *Collection Méthodes Informatiques et Pratique des Systèmes.* 1996, 324 pages.

MANUELS INFORMATIQUES MASSON

Langage C++

La proposition de standard ANSI / ISO expliquée

Jacquelin CHARBONNEL
Enseignant, ingénieur à l'ESPCI

MASSON

Paris Milan Barcelone

Ce logo a pour objet d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, tout particulièrement dans le domaine universitaire, le développement massif du « photocopillage ».
Cette pratique qui s'est généralisée, notamment dans les établissements d'enseignement, provoque une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que la reproduction et la vente sans autorisation, ainsi que le recel, sont passibles de poursuites. Les demandes d'autorisation de photocopier doivent être adressées à l'éditeur ou au Centre français d'exploitation du droit de copie: 3, rue Hautefeuille, 75006 Paris. Tél.: 43 26 95 35.

Le lecteur pourra contacter l'auteur, notamment pour les exercices proposés, à l'adresse électronique suivante :
Charbonnel@espci.fr

ISBN : 2-225-85219-7

ISSN : 0249-6992

MASSON S.A. 120, bd Saint-Germain, 75280 Paris Cedex 06

TABLE DES MATIERES

LE STANDARD C++

COMPOSANTS PREDEFINIS

C++ EN PRATIQUE

ANNEXES

CONTENTS

PART ONE - STANDARD C++

PART TWO - PREDEFINED COMPONENTS

Language support library
Diagnostics library
General utilities library
String library
Containers library
Iterators library
Algorithms library
Numerics library
Input/output library

PART THREE - USING C++

Using references
- References and argument passing
- References and return statement
- Conclusion

Memory management
- Arrays allocation
- Handling allocation errors

Designing good classes
- Separating logically and physically interface from implementation

About constructors and destructors
- Performance of constructors
- Constructors and inheritance
- Suppressing synthetisations

Member functions : constant or not constant ?

Duplicating objects
- When and why user duplication must be implemented ?
- Duplicating derived objects
- Special kind of synthetisations
- More about copy constructors
- More about assignment operator

Implicit conversions
- Limiting implicit conversions
- User-defined conversions
- Implicit conversion sequences
- Ranking implicit conversion sequences
- Ambiguities

Overload resolution algorithm
- Candidate functions
- Viable functions
- Best viable function

Practice of inheritance
- Friend relation and inheritance
- Static members and inheritance
- Pointer to member and inheritance
- More about dynamic_cast
- Virtual functions

Using handling exceptions
- Creating exception types
- Examples of handler
- Warning about liberation of resources
- Keeping integrity of objects
- Conclusion

Using templates
- Friend relation and templates
- Static members and templates
- Inheritance and templates
- Meta-programs

Using the stream classes
- Overloading << and >> operators
- Writing manipulators
- Exercises

Others traps and pitfalls
- Lexical pitfalls
- Unlucky confusions
- Costly carelessness
- About scope of identifiers
- Overloading and namespaces
- Warning : temporary objects !

AVANT-PROPOS

Ce document a été conçu à partir de l'expérience acquise en plusieurs années de pratique et d'enseignement de la programmation en langage C++.

L'objectif de l'ouvrage est de faire comprendre au programmeur C la puissance et les nouveaux horizons offerts par le langage C++, tel qu'il est décrit aujourd'hui dans les spécifications de l'ANSI et de l'ISO.

Les moyens pour atteindre cet objectif sont doubles. Le premier, c'est offrir une référence *abordable*, aussi précise et complète que possible, de la syntaxe et de la sémantique du langage C++. Des exemples et des commentaires sont là pour faciliter la compréhension et l'assimilation des concepts. Le second, c'est accompagner les néophytes, c'est-à-dire les programmeurs C qui n'ont pas encore été confrontés aux subtilités de C++, dans la mise en œuvre du langage. A cet égard, des conseils sont proposés, les points délicats du langage sont discutés, les pièges sont mis en lumière, décortiqués et commentés.

Cet ouvrage s'appuie constamment sur le document de travail des comités de normalisation ANSI et ISO. Celui-ci n'est pas à l'heure actuelle définitif, mais seuls des changements mineurs peuvent intervenir d'ici la publication officielle de la norme C++.

Ce document est composé de trois parties. La première, *Le standard C++*, décrit et commente les concepts du langage. La progression a été conçue dans une optique pédagogique, les exemples sont nombreux et des exercices sont proposés en fin de chapitres.

La seconde partie, *Composants prédéfinis*, est un extrait simplifié de la bibliothèque standard. Son but est d'abord de fournir les informations nécessaires à l'utilisation des classes indispensables à la réalisation des premiers programmes. Mais son but est aussi de donner une idée des grandes lignes qui ont guidé la conception de cette bibliothèque, ainsi que les principes généraux de son utilisation.

La troisième partie, *C++ en pratique*, approfondit la mise en œuvre des concepts, expose et commente les problèmes classiques, répertorie les fréquentes sources d'erreurs, et tente d'y apporter des solutions.

Je remercie Patrice Boizumault (Ecole des Mines de Nantes), Sophie Charbonnel-Moreau, Jean-Claude Royer (Université de Nantes), Serge Tahé (ISTIA, Université d'Angers) et Marie-Thérèse Charbonnel pour leur participation, contributions, critiques ou commentaires, ainsi que tous les étudiants qui ont relu et corrigé les premières versions de ce document alors sous forme de support de cours. Un merci particulier à Didier Moreau (Télis Cégélog, Nantes) pour ses corrections pointilleuses et implacables, ses nombreuses suggestions et l'apport de son expérience dans le domaine.

HISTORIQUE

Les prémices

Deux ans avant d'avoir l'idée d'ajouter les concepts de Simula au langage C, Bjarne Stroustrup travaille au Computing Laboratory de l'Université de Cambridge. C'est pendant cette période que mûrissent les idées qui mèneront à l'émergence de C++.

Son projet, optimisation des systèmes logiciels en environnements distribués, aboutit au développement d'un simulateur expérimental pour l'exécution de logiciels répartis.

La première version de ce simulateur est écrit en Simula. « *Son écriture fut un véritable plaisir...* » explique Stroustrup dans [AHC]. Les caractéristiques de Simula sont, en effet, idéales pour ce type de sujets : les concepts du langage aident à réfléchir aux problèmes spécifiques à l'application. En particulier, le concept de classe permet de représenter directement et clairement les composantes de l'application. Stroustrup apprécie également le typage souple et les capacités du compilateur à détecter les erreurs de type. Grâce aux classes et à la vérification intelligente des types, le nombre des erreurs augmente beaucoup moins vite que la taille du programme. La totalité du programme se comporte comme une collection de petits programmes, faciles à écrire, à comprendre et à déboguer.

Mais, à l'époque, l'implémentation de Simula n'est pas à la hauteur : l'édition des liens est très lente, et les temps d'exécution sont longs, à cause notamment du contrôle dynamique des types, du ramasse-miettes et du support de la concurrence. Le projet est finalement un désastre.

Pour le faire aboutir, le programme est recodé en BCPL. Mais cette fois, l'expérience est particulièrement désagréable : BCPL est un langage de bas niveau, sans contrôle de type ni support d'exécution. Par contre, les temps d'exécution sont excellents.

Cette expérience n'est pas sans conséquence pour C++. La leçon retenue, c'est qu'un outil pour la réalisation de tels projets doit :

— disposer d'un support pour l'organisation des programmes : des classes, différentes formes de hiérarchie, un support de la concurrence, un système de type fort, etc. ;

— générer des exécutables rapides (aussi rapide que les programmes BCPL) ;

— faciliter la combinaison d'unités compilées en C, Algol68, FORTRAN, assembleur, etc., en un unique programme non limité par une quelconque caractéristique de l'un de ces langages ;
— disposer d'implémentations hautement portables : avoir plusieurs sources d'implémentation, ne nécessitant pas un support d'exécution difficile à porter, et rester indépendant du système d'exploitation hôte.

La version 2.0 de C++ sera conforme à ces principes.

De C à C with classes

La première évolution de C s'appelle *C with classes*. L'expérience acquise avec elle, de 1979 à 1983, conduira aux fondements de C++.

En octobre 1979 est réalisé un préprocesseur qui ajoute les classes au langage C. En mars 1980, ce préprocesseur est utilisé pour plusieurs projets.

Pourquoi utiliser C comme langage de base, C n'étant ni le langage le plus élégant, ni le plus facile à manier et à maintenir ? Stroustrup justifie ce choix : le langage est massivement utilisé. Selon lui, cet état de fait s'explique par les qualités du langage C, à savoir principalement :

— la flexibilité : son champ d'application est très large, toute technique de programmation peut être utilisée, tout type de programme peut être réalisé ;
— l'efficacité : la sémantique étant de bas niveau (ses concepts sont ceux des ordinateurs traditionnels), il est facile pour le compilateur et pour le programmeur d'utiliser les ressources du matériel ;
— la disponibilité : la probabilité de trouver un compilateur C standard avec sa bibliothèque sur un système donné, du micro au super-calculateur, est grande ;
— la portabilité : le portage d'application C d'une machine à une autre est toujours possible, techniquement et économiquement.

Comparés à ces avantages, les inconvénients, comme la syntaxe bizarre des déclarations ou les constructions dangereuses, sont négligeables. Créer un meilleur C permettrait de corriger les inconvénients de C sans en perdre les avantages.

D'autres possibilités sont cependant étudiées : Pascal, Modula-2, Ada, Smalltalk, Mesa, CLU, qui furent sources d'inspiration pour C++. Cependant, seuls C, Simula (pour les classes), Algol68 (pour la surcharge d'opérateurs, les références, les déclarations en milieu de bloc) et BCPL (pour les commentaires //) laissent des traces dans la version C++ de 1985. L'évolution de C++ à partir de 1985 met en évidence d'autres influences : Ada, CLU et ML.

La période d'avril à octobre 1980 est une période de transition, au cours de laquelle l'idée de préprocesseur fait place peu à peu à celle de langage. Mais dans les faits, *C with classes* n'est toujours qu'une extension primaire de C, permettant

simplement d'exprimer la modularité et la concurrence. Une première description de *C with classes* est néanmoins publiée comme un rapport des Bell Labs en avril 1980 et plus tard dans SIGPLAN Notices.

L'objectif de *C with classes* est surtout de faciliter l'organisation des programmes. Mais il est essentiel, pour Stroustrup, que cette faculté ne pénalise pas les qualités des exécutables générés. La contrainte fixée est de coïncider avec C au niveau des temps d'exécution, du volume de code généré et du volume des données générées. Cette exigence incontournable est très contraignante. Les domaines d'utilisation de *C with classes* doivent rester les mêmes que ceux de C. *C with classes* doit pouvoir être utilisé partout où C peut l'être. Cette obligation empêchera *C with classes* de rectifier certains points faibles du langage C.

Les concepts de *C with classes* en 1980 sont :

— les classes, classes dérivées,
— les contrôles d'accès public et privé,
— les constructeurs et destructeurs,
— la possibilité de définir une fonction implicitement appelée avant tout appel de fonction membre et une autre appelée implicitement avant tout retour de fonction membre (curieusement, cette caractéristique n'existe pas en C++[1]),
— les classes amies,
— la vérification du type des arguments de fonction,

puis en 1981 :

— les fonctions en ligne,
— les arguments par défaut,
— la surcharge de l'opérateur =.

De C with classes à C++

En 1982, il devient évident que *C with classes* n'obtiendrait qu'un succès mitigé : un succès pour Stroustrup et ses collègues, mais pas pour la communauté des développeurs d'applications.

Le succès de *C with classes* est dû sans aucun doute à l'aide apportée à l'organisation des gros programmes, sans perte d'efficacité par rapport à C, et sans nécessiter un changement de culture pour les programmeurs. Les facteurs d'échec sont d'une part le peu de nouveautés par rapport à C, et la technologie préprocesseur utilisée d'autre part. *C with classes* allait dans la bonne direction, mais n'était qu'un trop petit pas.

L'alternative est alors soit l'abandon pur et simple du projet, soit sa poursuite sous la forme du développement d'un nouveau langage, à partir de l'expérience acquise avec *C with classes*. Mais un tel projet ne peut être viable que si le

1 « *personne (sauf moi) ne les utilisait* » avoua Stroustrup dans [AHC].

nouveau langage est en mesure de satisfaire une communauté de développeurs fixée à 5000 utilisateurs industriels.

Stroustrup commence alors à imaginer un successeur de *C with classes*, implémenté à l'aide des technologies traditionnelles des compilateurs.

Le langage qui en découle est baptisé temporairement *C84*. En effet, l'inconvénient de l'appellation *C with classes*, c'est qu'elle devenait, au fil du temps, *new C*, puis simplement *C*. Du coup, l'appellation *C* se transformait peu à peu en *plain C*, en *straight C*, et finalement en *old C*. En définitive, le nom de C++ est choisi en 1984, pour sa *fine* interprétation : en C, ++, prononcé *plus plus*, est compris comme *suivant* ou *successeur*.

Les ajouts initiaux de C++ par rapport à *C with classes* sont :

— les fonctions virtuelles,
— la surcharge des noms de fonctions et des opérateurs,
— les références,
— les membres constants,
— le contrôle par l'utilisateur de la gestion dynamique de la mémoire (définition des opérateurs `new` et `delete`).

Le compilateur C++, appelé *Cfront*, est conçu et implémenté par Stroustrup entre le printemps 1982 et l'été 1983. C'est pendant cette période qu'est rédigé le manuel de référence, publié le 1er janvier 1984. La bibliothèque des nombres complexes et la première classe `string` sont également conçues pendant cette période.

Cfront est un compilateur traditionnel, réalisant une vérification complète de la syntaxe et de la sémantique du langage. Il est d'abord écrit en *C with classes* et bientôt traduit en C++. La première version de *Cfront* utilise massivement les classes, mais pas les fonctions virtuelles qui n'existent pas encore au début du projet.

L'aspect le moins habituel de *Cfront* est qu'il génère du C. Malgré cela, ce n'est ni un préprocesseur, ni un traducteur ligne à ligne de C++ en C, mais un véritable compilateur. Il génère du C pour obtenir une portabilité maximum, C étant considéré comme le plus portable des assembleurs. Cette stratégie sera reprise plus tard pour des langages comme Ada, CLOS, Eiffel, Modula-3 ou Smalltalk. Un haut niveau de portabilité est ainsi atteint pour un surcoût acceptable en temps de compilation. Le compilateur C n'est utilisé qu'en tant que générateur de code : tout message d'erreur du compilateur C aurait révélé soit une erreur dans *Cfront*, soit une erreur dans le compilateur C lui-même (!), mais jamais une erreur dans le code source C++. Le processus d'obtention des exécutables est donc le suivant :

code source C++ → `cpp`[2] → `cfront` → `cc`[3] → *code objet*

---

2 Le préprocesseur C.

3 Le compilateur C.

Versions

Les versions de C++ sont souvent référencées par les numéros de version de *Cfront*. La version 1.0 est le langage tel qu'il est défini dans *The C++ programming language*[4]. Les versions 1.1 (juin 1986) et 1.2 (février 1987) intègrent les pointeurs vers membres et les membres protégés.

Dès 1986, la direction dans laquelle doit évoluer C++ est tracée : type paramètré, héritage multiple et gestion des exceptions. C++ a alors environ 2000 utilisateurs. La version 2.0 sort en juin 1989. Les nouvelles caractéristiques sont :

— l'héritage multiple,
— le contrôle de type à l'édition des liens,
— une meilleure résolution des surcharges,
— une définition récursive des affectations et des clonages (réalisés membre à membre et non plus bit à bit),
— les classes abstraites,
— les fonctions membres statiques,
— les fonctions membres constantes,
— la surcharge de l'opérateur `->`.

La version 2.1 (avril 1990) corrige diverses bogues par rapport à la définition spécifiée dans le document de référence *The Annoted C++ Reference Manual*. La version 3.0 (septembre 1991) ajoute les modèles (*templates*) spécifiés dans [ARM]. La version 4.0 intègre la gestion des exceptions, comme spécifiée dans [ARM].

L'explosion

C++ fut d'abord conçu pour être opérationnel auprès des utilisateurs, et non pour servir de prototype de laboratoire. Et de fait, il a de plus en plus d'utilisateurs :

	nombre d'utilisateurs
octobre 1979	1
octobre 1980	16
octobre 1981	38
octobre 1982	85
octobre 1985	500
octobre 1986	2 000
octobre 1987	4 000
octobre 1988	15 000
octobre 1989	50 000
octobre 1990	150 000
octobre 1991	400 000

[4] B. Stroustrup — *The C++ programming language*, 1986 — Addison-Wesley
dont la traduction française est :
B. Stroustrup — *Le langage C++*, 1989 — InterEditions

Après la version 1.0 d'AT&T, c'est-à-dire à partir de 1986, des revendeurs, notamment Glockenspiel en Irlande, commencent à faire de la publicité. L'arrivée de compilateurs C++ propriétaires (Oregon Software C++, Zortech C++) coïncide avec l'apparition massive de références à C++ dans les annonces. A partir de 1991, le recensement des utilisateurs devient difficile. Le nombre des utilisateurs d'implémentation comme G++ du GNU ou des versions universitaires de *Cfront* ne peut être déterminé précisément. En octobre 1991, Borland, le plus gros fournisseur de compilateur C++, fait publiquement état de 500 000 compilateurs en service.

Compilateurs

La conférence de Santa Fé, en novembre 1987, marque l'avènement d'une seconde vague d'implémentation de C++, et présente le futur compilateur Oregon Software C++ (lancé en janvier 1988), et la première version de G++ (sortie en décembre 1987).

Jusqu'en juin 1988, tous les compilateurs C++ sur PC sont des portages de *Cfront*. Puis Zortech lance son compilateur, suivi par Borland en mai 1990, puis Microsoft en mars 1992. DEC sort son compilateur en février 1992, IBM en mai 1992. Il existe maintenant plus d'une douzaine de versions propriétaires. Il faut y ajouter les portages de *Cfront* que l'on retrouve partout : Sun, HP, ParkPlace, Centerline, Glockenspiel, Comeau Computing...

Bibliothèques

Le tout premier code écrit en *C with classes* est la bibliothèque de tâches, fournissant un modèle de concurrence similaire à celui de Simula. Celle-ci reste toujours utilisée.

La bibliothèque standard C est également disponible en C++, sans surcoût ni complication supplémentaire par rapport à son utilisation à partir du langage C. De même pour toutes les autres bibliothèques C.

Les structures de données classiques (chaînes de caractères, tableaux à accès contrôlés, tableaux dynamiques, listes, etc.) sont conçues simplement pour servir de tests aux premières implémentations.

Les premiers travaux sur les classes *conteneurs* (listes, tableaux, etc.) sont entravés par l'absence de type paramètré jusqu'à la version 3.0.

La première implémentation de la bibliothèque de flux apparaît en 1984. Elle amène la différenciation de traitement des valeurs de type `char` et celle de type `short`.

Il existe beaucoup d'autres bibliothèques C++. La plus importante parmi les premières sorties est la bibliothèque de classes NIH (1990) qui fournit un ensemble de classes à la Smalltalk, et la bibliothèque graphique Interviews de

Linton (1987) pour XWindows. G++ arrive ensuite avec sa bibliothèque de classes. Rogue Wave & Dyad fournit un ensemble important de bibliothèques à usage scientifique. Glockenspiel fournit plusieurs bibliothèques : Commonview, bibliothèque de classes portables pour différentes interfaces graphiques (MSWindows, Motif, MacIntosh, HP New Wave, Presentation Manager), Container, et CommonBase, bibliothèque de classes portables pour différents SGBD. Rational diffuse une version C++ des composants de Booch (10000 lignes de code C++), à l'origine écrit en Ada (150000 lignes de code Ada). Il existe aujourd'hui beaucoup d'autres bibliothèques : il semble que nous sommes définitivement entrés dans l'ère de l'industrie du composant logiciel.

Publications

En 1983, on suggère à Stroustrup d'écrire un livre sur C++ dans la lignée du [K&R]. Le document, *The C++ programming language*, reprend les publications de Stroustrup, ses notes internes et un manuel de référence. Ce livre est donc la première définition de C++ et la première présentation de techniques et d'organisations relatives au langage.

Le premier journal dédié à C++, *C++ Report*[5], commence ses publications en janvier 1989. *The C++ Journal* apparaît pendant l'été 1990. Des articles sur C++ apparaissent également dans d'autres revues spécialisées. Des groupes de discussions (comp.lang.c++) produisent des milliers de messages à travers le monde, pour le plus grand plaisir (ou désespoir !) des lecteurs. Se tenir au courant aujourd'hui de tout ce qui s'écrit sur C++ est un travail à plein temps !

Standardisation

Dès 1988, il devient clair qu'une normalisation s'avère nécessaire. Il y a alors une poignée d'implémentations indépendantes en service ou en développement, et un effort de définition claire et précise s'impose.

L'initiative d'une standardisation formelle est prise par HP, en accord avec AT&T, DEC et IBM.

C'est dans cette optique que Stroustrup et M. A. Ellis entreprennent de réécrire le manuel de référence et de le faire circuler dans la communauté des développeurs C++ du monde entier. C'est ainsi que naît *The Annoted C++ Reference Manual*. Ce manuel de référence fournit une définition complète de C++ ainsi que des discussions sur ce qui n'est pas dans le langage, le pourquoi de telle ou telle caractéristique, le comment de certaines implémentations. Ces remarques, intéressantes pour la plupart des programmeurs, sont insérées sous

---

5 C++ Report — SIGS Publications.

forme d'annotations ou de commentaires dans le document. Ce document servira de base aux travaux de normalisation.

La première réunion du comité de normalisation ANSI C++ (X3J16) a lieu en décembre 1989 à Washington DC, et réunit une quarantaine de personnes (le comité compte maintenant plus de 250 membres). Parmi elles, des gens qui ont pris part à la normalisation de C, et des programmeurs C++ de longue date.

Les raisons officielles données pour une standardisation immédiate sont :

— le vif engouement du public pour C++,
— le danger de voir rapidement apparaître des dialectes,
— le besoin d'une définition détaillée de la sémantique pour chaque caractéristique du langage,
— des lacunes à combler (absence de gestion des exceptions, d'héritage multiple, de type paramètré et de bibliothèque standard),
— la nécessité d'une compatibilité avec le C ANSI.

La standardisation n'étant pas une affaire spécifiquement américaine, des représentations d'autres pays assistent depuis le début aux réunions ANSI C++. En juin 1991, le comité ISO C++ (SC22/WG21), composé de délégations européenne, canadienne, japonaise, etc., se réunit et les deux comités ANSI et ISO décident de joindre leurs meetings.

Le *comitee draft* (CD) est déposé en septembre 1994 : son contenu est alors gelé. Les ajouts ne sont plus acceptés, et le travail du comité se limite alors à affiner et à clarifier le contenu du *draft*. Puis, le CD est remis à la critique publique en 1995. Le rôle du comité est alors de répondre aux commentaires publics, ce qui amène retouches et clarification du CD. Si tout va bien, le *Document International Standard* (DIS), le standard officiel C++, sortira fin 1996. Par contre, si une deuxième consultation publique s'avère nécessaire, le standard ne sortira pas avant fin 1997.

CARACTERISTIQUES

C++ se présente comme un langage à objets, sur-ensemble du langage C ANSI. A quelques incompatibilités près, les programmes C sont compilables par un compilateur C++. Cela permet au programmeur de passer progressivement de la programmation structurée à la programmation par objets. En contrepartie, rien n'impose au développeur C++ d'appliquer les principes de la programmation par objets.

Un des aspects les plus positifs est sans doute l'absence de dégradation des performances entre l'exécution d'un programme écrit en C et celle du même programme écrit en C++ : C++ n'impose aucun mécanisme sous-jacent qui soit pénalisant en vitesse d'exécution ou en occupation mémoire[6]. C++ peut même parfois générer un exécutable plus rapide et plus compact du fait d'une meilleure conception.

C'est en effet à la conception, mais aussi à la maintenance que C++ apporte le plus, car :

- — les modules sont plus facilement réutilisables,
- — le code est plus portable (en raison notamment de conversions plus rigoureuses),
- — le code est plus lisible,
- — les contrôles du compilateur sont plus stricts, ce qui permet de déceler plus d'erreurs à la compilation.

C++ reste malgré tout un langage relativement complexe, conséquence de son histoire et de sa richesse : trois types d'héritage (public, protégé, privé), trois contrôles d'accès pour les membres des classes, deux types de liaison (statique, dynamique) pour les méthodes, concepts de classes amies, de fonctions amies, de classe de base virtuelle, de classe polymorphe...

---

6 Cependant, des mécanismes consommateurs de temps et de mémoire (gestion des exceptions, identification dynamique des types, polymorphisme) peuvent être mis en oeuvre à la demande explicite du programmeur.

— Première partie —

Le standard C++

I - C++ : UN C REVU ET CORRIGE

C++ introduit un certain nombre d'améliorations syntaxiques, d'extensions et de nouveautés par rapport au langage C. Indépendamment de l'orientation objet de C++, ce premier chapitre présente les points qui contribuent à faire que C++ soit un meilleur C. Il expose également les principales sources d'incompatibilité entre les deux langages.

1 - Nouveaux mots clés

C++ dispose de nouveaux mots clés :

```
bool          catch             class          const_cast
delete        dynamic_cast      explicit       false
friend        inline            mutable        namespace
new           operator          private        protected
public        reinterpret_cast  static_cast    template
this          throw             true           try
typeid        typename          using          virtual
wchar_t
```

Cette liste doit être complétée par les représentations alternatives de certains opérateurs :

```
and (&&)      and_eq (&=)     bitand (&)     bitor (|)
compl (~)     not (!)         not_eq (!=)    or (||)
or_eq (|=)    xor (^)         xor_eq (^=)
```

Tout ces littéraux sont réservés et ne peuvent être utilisés comme identificateurs. Cela introduit par conséquent des incompatibilités inévitables avec le langage C. Les déclarations suivantes sont illégales en C++ :

```
char *friend="Jérémie" ;                    // illégal
char *operator="+-*/" ;                     // illégal
void delete(void *ptr) ;                    // illégal
int new ;                                   // illégal
```

2 - Les extensions syntaxiques de C

i. Commentaires

Les commentaires en C++ commencent par // et finissent à la fin de la ligne. Les commentaires du langage C sont toujours valides :

en C

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

en C++

```
// ceci est
// un commentaire C++

/* ceci est aussi un
   commentaire acceptable
   en C++ */
```

ii. Transtypages

cast

C++ possède une nouvelle syntaxe pour le *transtypage* explicite, bien que l'opérateur de *cast* du langage C soit toujours valable :

en C

```
int i=1;
float x=3.2;


i = (int) x;
x = (float) i;
```

en C++

```
int i=1;
float x=3.2;

// nouvelle syntaxe:
i = int(x);
x = float(i);

// toujours valable:
i = (int) x;
x = (float) i;
```

L'ancien opérateur de *cast* reste malgré tout indispensable en C++ lorsque le type est composé :

```
ptr1 = char * (ptr2) ;          // syntaxiquement incorrect
ptr1 = (char *) ptr2 ;                  // d'accord comme çà
```

L'opération de *cast* a beaucoup de défauts. D'abord syntaxiquement, elle utilise les parenthèses, déjà surexploitées, et sa syntaxe trop concise la rend difficilement repérable. Ensuite, cette opération est utilisée à plusieurs fins : réinterprétation de séquence de bits, conversions arithmétiques, suppression de `const`, suppression de `volatile`... Son utilisation obscurcit les programmes et complique la maintenance.

Les opérateurs `static_cast`, `reinterpret_cast`, `const_cast` et `dynamic_cast`[7] ont été introduits afin de rendre plus explicites et moins sournois les transtypages. Leurs noms ont d'ailleurs été choisis volontairement longs, non seulement pour décourager les programmeurs de les utiliser, mais surtout pour les repérer rapidement dans le programme.

`static_cast`

Sa syntaxe est :

`static_cast``<`*T*`>(`*exp*`)`

où *exp* est l'expression à convertir en type *T*.

Cet opérateur ne peut réaliser que des transtypages au comportement parfaitement défini, c'est-à-dire soit des transtypages sûrs, soit des transtypages non sûrs, mais bien définis sur toute implémentation.

L'opérateur `static_cast` est utilisé pour effectuer des conversions standard, des conversions définies par l'utilisateur[8], ou certaines conversions douteuses mais dont le résultat est parfaitement défini sur toute implémentation, comme :

```
long l=12345 ;
int i = static_cast<int>(l) ;
```

ou :

```
enum E { un, deux, trois } ;
enum E e = static_cast<enum E>(1) ;
```

ou encore des conversions de `void *` vers `T *`[9].

Par contre, cet opérateur n'effectue pas de transtypage entre des types n'ayant aucune relation entre eux, ni entre des pointeurs vers des types n'ayant aucune relation entre eux. Il ne permet pas de passer outre les spécifications `const` et `volatile`. Une telle tentative conduit à une erreur de compilation :

```
struct A {} a ;
struct B {} *pb = static_cast<B *>(&a) ;          // illégal
```

`reinterpret_cast`

Sa syntaxe est :

`reinterpret_cast``<`*T*`>(`*exp*`)`

où *exp* est l'expression à convertir en type *T*.

Cet opérateur est utilisé pour réaliser des transtypages intrinsèquement dangereux et mal définis. Il renvoie une suite de bits, sans interprétation, à partir

---

7 Voir *L'opérateur dynamic_cast* page 126.

8 Voir *Conversions de l'utilisateur* page 99.

9 Voir d'autres cas d'application au paragraphe *L'opérateur dynamic_cast* page 126.

de la valeur de son opérande. La valeur renvoyée dépend de l'implémentation : cette conversion peut ou non produire une suite de bits différente de celle fournie en opérande.

Il peut être utilisé pour effectuer des conversions entre des types n'ayant aucune relation entre eux, entre pointeurs vers des types n'ayant aucune relation entre eux, des conversions d'entiers en pointeurs et inversement :

```
struct A {} a ;
struct B {} *pb = &a ;                                  // illegal
struct C {} *pc = reinterpret_cast<C *>(&a) ;             // ok

long l1 = &a ;                                          // illégal
long l2 = reinterpret_cast<long>(&a) ;                    // ok

void *ptr1 = 0xA0000l ;                                 // illégal
void *ptr2 = reinterpret_cast<void *>(0xA0000l) ;         // ok
```

Il ne permet pas de passer outre les spécifications `const` et `volatile` :

```
const int i = 10 ;
int *ptr = reinterpret_cast<int *>(&i) ;                  // erreur
```

`const_cast`

Sa syntaxe est :

`const_cast``<`*T*`>(`*exp*`)`

où *exp* est l'expression à convertir en type *T*. *T* est un type pointeur ou référence, et doit être le type de *exp*, aux `const` et `volatile` près.

Cet opérateur effectue des transtypages consistant à supprimer ou ajouter des spécifications `const` ou `volatile` :

```
const int x = 16 ;
int *ptr1 = &x ;                                        // illégal
int *ptr2 = const_cast<int *>(&x) ;                       // ok
```

iii. Nouveaux suffixes pour les constantes numériques littérales

Outre les suffixes `l` ou `L` existant déjà en C, et spécifiant qu'une constante numérique est de type `long`, on trouve en C++ les suffixes `f` ou `F`, spécifiant qu'une constante est de type `float`[10], et `u` ou `U` spécifiant qu'une constante est `unsigned` :

```
unsigned long l=1ul ;
float x=1.2f ;
```

[10] Par défaut, les constantes réelles sont de type `double`, en C comme en C++.

3 - Pointeurs génériques

En C ANSI, un pointeur de type `void *` est compatible avec les autres types pointeurs, et tout type pointeur est compatible avec le type `void *`. Quel que soit le type *T*, il y a conversion implicite :

— de *T* * vers `void *`,

— de `void *` vers *T* *.

En C++, seule existe la conversion d'un pointeur d'un type particulier vers un pointeur générique. C'est-à-dire que la conversion implicite n'existe que dans le sens *T* * vers `void *`. Cette restriction renforce le système de type : toute opération sur un `void *` est possible sur un *T* *, l'inverse n'étant pas vrai.

```
void *generic;
int *ptr;

generic = ptr;                    /* légal en C et en C++ */
ptr = generic;                   /* légal C, illégal en C++*/

ptr = malloc(sizeof(int));    /* légal C, illégal en C++ */
ptr = (int *) malloc(sizeof(int));    /* légal C et C++ */
```

Et encore faut-il que ce pointeur ne soit pas un pointeur vers un objet constant ou volatile, sinon la conversion de ce pointeur en `void *` est illégale :

```
void *generic ;
const int *ptr ;

generic = ptr ;            /* légal en C, illégal en C++ */
```

4 - Constantes

i. Constante caractère

Les constantes caractères sont de type `char` en C++, alors qu'elles sont de type `int` en C :

```
int b = sizeof('A')==sizeof(char);
```

`b` vaut `0` en C, alors qu'il vaut `1` en C++.

ii. Portée des constantes

On appelle *unité de compilation* l'ensemble composé du code d'un fichier source et du code de tous les fichiers inclus, à l'exception des lignes exclues par les directives conditionnelles du préprocesseur.

Le modificateur de type `const` spécifie que la valeur de l'objet ainsi déclaré ne peut pas être modifiée. Mais il existe quelques différences entre le `const` de C et celui de C++.

En C, un identificateur global spécifié `const` est traité comme un identificateur global standard : si `static` n'est pas mentionné, sa portée[11] est externe à l'unité de compilation la contenant. On peut alors y faire référence à partir d'une autre unité de compilation.

Prenons trois fichiers sources `f.h`, `f1.c` et `f2.c`, organisés en deux unités de compilation (`f.h`, `f1.c`) et (`f.h`, `f2.c`), appartenant à la même application C. Pour créer une constante `K` commune aux deux unités de compilation, on définit `K` dans `f1.c` ou `f2.c`, de la sorte :

f.h	f1.c	f2.c
	`#include "f.h"`	`#include "f.h"`
	`const int K=255;`	`extern const int K;`
`/*...*/`	`/*...*/`	`/*K utilisable ici*/`

De ce point de vue, une constante est donc traitée comme une variable.

Par contre, en C++, lorsqu'un identificateur global est spécifié `const`, sa portée se voit restreinte à l'unité de compilation contenant sa définition, comme si `static` avait été spécifié. L'exemple précédent conduit donc à une erreur en C++ au moment de l'édition des liens, puisque la constante `K`, invisible hors de (`f.h`, `f1.c`), ne peut être référencée dans (`f.h`, `f2.c`).

En C++, `K` doit donc être explicitement déclarée de portée externe, ou alors elle doit être définie dans un fichier en-tête inclus par chaque source l'utilisant :

f.hpp	f1.cpp	f2.cpp
`const int K=255;`	`#include "f.hpp"`	`#include "f.hpp"`
	`// K visible ici`	`// K visible ici`

Cela rapproche l'utilisation des constantes de celles des symboles définis par `#define`.

Par conséquent, il est possible de définir deux constantes de même nom, dans deux unités de compilation de la même application, sans qu'il y ait collision entre les deux noms :

f.hpp	f1.cpp	f2.cpp
	`const int K=255;`	`const int K=1;`
	`// K vaut 255`	`// K vaut ici 1`

[11] La portée d'un identificateur est la partie du programme dans laquelle il est possible d'accéder à l'objet représenté par l'identificateur (voir *Portée et visibilité* page 170).

iii. Expression constante

Une expression constante est une expression qui peut être évaluée à la compilation. Si `MAX` est défini par :

```
const int MAX=1000 ;
```

alors `MAX` n'est pas une expression constante en C, alors qu'il en est une en C++. De même pour des expressions du type `2*(MAX+1)`.

Là encore, les constantes en C++ se rapprochent des symboles définis avec `#define`[12].

Cela se répercute dans toutes les utilisations d'expressions constantes, en particulier les définitions de tableaux :

en C	en C++
`#define MAX 1000`	`const int MAX=1000;`
`char tab[MAX];`	`char tab[MAX]; //illégal en C`

L'utilisation d'une constante doit être préférée à celle d'un symbole. Typées, les constantes s'intègrent mieux au langage. De plus, leur utilisation facilite le débogage[13] et laisse plus de possibilité d'optimisation au compilateur.

5 - Points de déclarations

En C++, tout identificateur doit être déclaré avant d'être utilisé. Mais les déclarations en C++ ne sont pas nécessairement regroupées en début de bloc : elles peuvent être précédées d'instructions. La portée de l'identificateur est alors limitée à la portion de code comprise entre sa déclaration et la fin de la *région déclarative*[14] dans laquelle il est défini

Cela permet en particulier d'initialiser un objet avec des valeurs obtenues (par calcul, par saisie, etc.) dans la même région déclarative. Cela est surtout utile pour des entités nécessitant obligatoirement des valeurs d'initialisation[15] :

---

12 Elles sont pourtant différentes : une constante définie avec `const` est une lvalue, c'est-à-dire qu'elle correspond à une entité stockée en mémoire, alors qu'une constante définie avec `#define` n'a pas d'existence propre, puisqu'elle disparaît lors du traitement par le préprocesseur.

13 Une constante peut être traitée par un débogueur comme un objet, et non pas simplement comme une valeur.

14 Une *région déclarative* est une portion de programme contiguë représentant la zone la plus grande dans laquelle les identificateurs qui y sont déclarés sont susceptibles d'être valides (voir *Portée et visibilité* page 170).

15 Par exemple, des constantes, des références et des objets sans constructeur par défaut.

```
int main()
{
  int i ;
  scanf("%d",&i) ;
  const int k=i ;
  /* ... */
}
```

Des identificateurs peuvent être déclarés dans l'expression d'initialisation d'une boucle `for`[16], et dans les expressions conditionnelles des boucles `if`, `for`, `while` et `do`[17] :

```
for (int i=0 ; i<MAX ; i++) { /* ... */ }
```

En C++, il est possible d'initialiser des variables statiques avec des valeurs inconnues lors de la compilation :

```
int a=1 ;
int b=a ;                    // si a et b sont globales, ceci
                          // est illégal en C et légal en C++

int f(int x)
{
  static int y=x ;                  // également illégal en C
                                       // et légal en C++
  /* ... */
}
```

et même d'initialiser une variable globale avec la valeur de retour d'une fonction :

```
FILE *f = fopen("donnees","r") ;
```

Cette initialisation est illégale en C si `f` est globale, mais légale en C++.

6 - Les fonctions

i. En-têtes de fonctions

La syntaxe K&R

La syntaxe K&R de l'en-tête de fonction, acceptée par le C ANSI, n'est plus autorisée en C++. L'en-tête de la fonction suivante :

---

16 Voir *Déclarations dans l'instruction d'initialisation d'un for* page 376.

17 Voir *Déclarations dans une condition* page 377.

```
void swap(x,y)
int *x, *y ;
{
  /* ... */
}
```

est rejeté par le compilateur C++, et doit être mis sous la forme :

```
void swap(int *x, int *y)
{
  /* ... */
}
```

Les fonctions sans valeur de retour

En C++, une fonction dont le type de retour n'est pas `void` doit obligatoirement renvoyer une valeur :

```
char f()
{
  /* ... */
  return ;                                  // erreur en C++
}
```

L'instruction `return` ci-dessus provoque une erreur de compilation en C++, alors qu'elle ne conduit qu'à un avertissement en C ANSI.

ii. Prototypes de fonctions

Contrôle du nombre et du type des arguments

C++ est fortement typé : le nombre et le type des arguments[18] de fonctions est rigoureusement contrôlé à chaque appel par rapport aux paramètres[19] déclarés dans les prototypes précédant l'appel, et éventuellement dans l'en-tête de la définition. Ainsi, et contrairement au langage C, tout appel de fonction doit obligatoirement être précédé d'au moins un prototype, ou de la définition même de la fonction.

Fonctions sans paramètre

En C, un prototype de fonction sans paramètre signifie que la fonction peut être appelée avec n'importe quels arguments. Dans ce cas, aucun contrôle de cohérence n'est effectué par le compilateur entre les paramètres et les arguments

---

18 Appelés aussi paramètres effectifs.

19 Appelés aussi paramètres formels.

fournis. Pour indiquer qu'il n'y a aucun paramètre, `void` doit être spécifié dans le prototype.

En C++, un prototype de fonction sans paramètre signifie que la fonction n'a pas de paramètre. La comparaison suivante illustre cette différence de sémantique :

en C	en C++
`/* ce prototype n'indique ni le nombre, ni le type des paramètres: */` `int f();`	`// ce prototype signifie que f n'a pas de paramètre:` `int f();`

`f();  /* légal */`
`f(1);  /* légal */`
`f(1,2);  /* légal */`
	`f();  // légal`
	`f(1);  // illégal`
	`f(1,2);  // illégal`

`/* ce prototype signifie que g n'a pas de paramètre:*/` `int g(void);`	`//ce prototype signifie aussi que g n'a pas de paramètre:` `int g(void);`

`g();  /* légal */`
`g(1);  /* illégal */`
	`g();  // légal`
	`g(1);  // illégal`

iii. Paramètres anonymes

En C, tout paramètre doit être nommé, qu'il soit utilisé ou non dans la fonction. Pour chaque paramètre inutilisé, le compilateur émet en principe un avertissement.

En C++, un paramètre inutilisé peut être anonyme, ce qui supprime tout avertissement de la part du compilateur.

Dans l'exemple suivant, la fonction `appliquer` applique une fonction `*f` à tous les éléments d'un tableau `t`. `*f` possède deux paramètres de type `int` et `void *` :

```
void appliquer(int t[], int nb, int (*f)(int,void *)) ;
```

On veut appliquer la fonction `Max` suivante à un tableau `tab`. Bien que `Max` n'ait besoin que d'un paramètre, il est nécessaire de lui en déclarer deux : `Max` doit en effet être du type de `*f`. En C++, ce paramètre inutilisé peut rester anonyme :

```
int Max(int x, void *)
{
  static int max=0 ;

  if (max<x) max = x ;
  return max ;
}
```

```
void main()
{
  int tab[100] ;
  /* ... */
  appliquer(tab,100,Max) ;
}
```

iv. Arguments par défaut

Un paramètre d'une fonction peut être doté d'une valeur par défaut, de la façon suivante :

```
void saut(FILE *f=stdout)
{
  putc('\n',f) ;
}
```

Dans cet exemple, le paramètre `f` a une valeur par défaut égale à `stdout`. La fonction `saut` peut être appelée de façon habituelle :

```
saut(stderr) ;
```

auquel cas le paramètre `f` est initialisé normalement avec `stderr`, mais elle peut être aussi appelée sans argument :

```
saut() ;
```

auquel cas `f` est initialisée avec sa valeur par défaut. `stdout` est donc un argument par défaut pour `saut`.

Plus précisément, pour affecter une valeur par défaut à un paramètre, il faut que ce paramètre soit le dernier de la liste, ou qu'il ne soit suivi que par d'autres paramètres ayant eux aussi une valeur par défaut :

```
void f (float x, int i=0) ;                          // ok
void g (float x, int i=0, char *s="") ;              // ok
int h (int i=0, float x) ;                      // illégal
```

Un argument est optionnel, pour un appel de fonction donné, si les deux conditions suivantes sont réunies :

— son paramètre correspondant possède une valeur par défaut,

— il est le dernier argument de la liste[20].

Par exemple :

```
void f (float, int=2, char * ="");
```

[20] C'est-à-dire qu'ou bien le paramètre correspondant est le dernier paramètre de la liste, ou bien, pour cet appel de fonction donné, tous les arguments suivants ont été omis.

```
f(2.71828,10,"Anaïs") ;          // "Anaïs" est optionnel,
                                 // 10 n'est pas optionnel
f(2.71828,10) ;                     // 10 est optionnel
f(2.71828) ;              // 2.71828 n'est pas optionnel
```

Un argument peut être omis s'il est optionnel :

```
void f (float, int=2, char * ="");

f(2.71828,10) ;                                          // ok
f(2.71828,"Anaïs") ;                                // illégal
f(2.71828) ;                                             // ok
f() ;                                               // illégal
```

Pour un appel donné, un paramètre avec valeur par défaut est initialisé avec sa valeur par défaut lorsque l'argument correspondant a été omis :

```
void f (float, int=2, char * ="");

f(2.71828,10,"Anaïs") ;                  // appel classique
f(2.71828,10) ;          // identique à : f(2.71828,10,"");
f(2.71828) ;              // identique à : f(2.71828,2,"");
```

Les arguments par défaut peuvent être exprimés avec n'importe quelle expression valide ne faisant pas intervenir de variable locale :

```
const float epsilon=1e-10 ;
float f(int x, int y, float dx=sqrt(epsilon)) ;      // ok
```

Les paramètres eux-mêmes ne peuvent pas intervenir dans l'expression d'une valeur par défaut :

```
// pas de x et y déclarés ici
void f(int x, int y, int z=x*y) ;                  // illégal
```

Un argument par défaut ne peut pas être redéfini, même si sa valeur est la même :

```
void f(int=0) ;                                  // prototype

void f(int i=0)                                     // erreur
{
  /* ... */
}
```

Ici, la redéfinition de l'argument par défaut pour `i` est illégale.

Par contre, la liste des paramètres dotés d'une valeur par défaut peut être étendue incrémentalement au fur et à mesure des prototypes/en-tête rencontrés :

```
void f(int, char=' ') ;                  // premier prototype
void f(int=0, char) ;                     // second prototype
```

ces deux prototypes étant équivalents à :

```
void f(int=0, char=' ') ;
```

Dans l'exemple ci-dessous, la fonction `Simpson` est munie progressivement d'arguments par défaut :

```
typedef float (*F)(float) ;

                // premier prototype sans valeur par défaut
float Simpson(F f, float a, float b, float dx) ;

extern float epsilon ;

                 // définition : dx a une valeur par défaut
float Simpson(F f, float a, float b, float dx=epsilon)
{
  /* ... */
}

extern float A, B ;
F f ;

float aire1=Simpson(f) ;     // erreur, arguments manquants
float aire2=Simpson(f,A,B) ;                          // ok

  // second prototype : a, b, dx ont une valeur par défaut
float Simpson(F f, float a=A, float b=B, float dx) ;

float aire3=Simpson(f) ;                              // ok
```

Mais pour ajouter des arguments par défaut à une fonction, il faut que les déclarations soient situées dans une même région déclarative. Ainsi, le prototype de la fonction `f` locale à `g` suivant :

```
void f(int, int) ;
void f(int, int=1) ;

void g()
{
  void f(int=0, int) ;                          // erreur
}

void f(int=0, int) ;                                  // ok
```

est illégal, car il ne complète pas la liste des arguments par défaut spécifiés par les prototypes globaux : les prototypes ne sont pas dans la même région déclarative.

Les arguments par défaut ne font pas partie du type de la fonction :

```
void f(int) ;
void g(int=4321) ;
```

```
f(2.71828,10,"Anaïs") ;          // "Anaïs" est optionnel,
                                 // 10 n'est pas optionnel
f(2.71828,10) ;                     // 10 est optionnel
f(2.71828) ;              // 2.71828 n'est pas optionnel
```

Un argument peut être omis s'il est optionnel :

```
void f (float, int=2, char * ="");

f(2.71828,10) ;                                          // ok
f(2.71828,"Anaïs") ;                                  // illégal
f(2.71828) ;                                             // ok
f() ;                                                 // illégal
```

Pour un appel donné, un paramètre avec valeur par défaut est initialisé avec sa valeur par défaut lorsque l'argument correspondant a été omis :

```
void f (float, int=2, char * ="");

f(2.71828,10,"Anaïs") ;                  // appel classique
f(2.71828,10) ;          // identique à : f(2.71828,10,"");
f(2.71828) ;              // identique à : f(2.71828,2,"");
```

Les arguments par défaut peuvent être exprimés avec n'importe quelle expression valide ne faisant pas intervenir de variable locale :

```
const float epsilon=1e-10 ;
float f(int x, int y, float dx=sqrt(epsilon)) ;       // ok
```

Les paramètres eux-mêmes ne peuvent pas intervenir dans l'expression d'une valeur par défaut :

```
// pas de x et y déclarés ici
void f(int x, int y, int z=x*y) ;                  // illégal
```

Un argument par défaut ne peut pas être redéfini, même si sa valeur est la même :

```
void f(int=0) ;                                  // prototype

void f(int i=0)                                     // erreur
{
  /* ... */
}
```

Ici, la redéfinition de l'argument par défaut pour `i` est illégale.

Par contre, la liste des paramètres dotés d'une valeur par défaut peut être étendue incrémentalement au fur et à mesure des prototypes/en-tête rencontrés :

```
void f(int, char=' ') ;                  // premier prototype
void f(int=0, char) ;                     // second prototype
```

ces deux prototypes étant équivalents à :

```
void f(int=0, char=' ') ;
```

Dans l'exemple ci-dessous, la fonction `Simpson` est munie progressivement d'arguments par défaut :

```
typedef float (*F)(float) ;

                // premier prototype sans valeur par défaut
float Simpson(F f, float a, float b, float dx) ;

extern float epsilon ;

                 // définition : dx a une valeur par défaut
float Simpson(F f, float a, float b, float dx=epsilon)
{
  /* ... */
}

extern float A, B ;
F f ;

float aire1=Simpson(f) ;    // erreur, arguments manquants
float aire2=Simpson(f,A,B) ;                          // ok

  // second prototype : a, b, dx ont une valeur par défaut
float Simpson(F f, float a=A, float b=B, float dx) ;

float aire3=Simpson(f) ;                              // ok
```

Mais pour ajouter des arguments par défaut à une fonction, il faut que les déclarations soient situées dans une même région déclarative. Ainsi, le prototype de la fonction `f` locale à `g` suivant :

```
void f(int, int) ;
void f(int, int=1) ;

void g()
{
  void f(int=0, int) ;                          // erreur
}

void f(int=0, int) ;                                  // ok
```

est illégal, car il ne complète pas la liste des arguments par défaut spécifiés par les prototypes globaux : les prototypes ne sont pas dans la même région déclarative.

Les arguments par défaut ne font pas partie du type de la fonction :

```
void f(int) ;
void g(int=4321) ;
```

```
void main()
{
  void (*ptr)(int) ;
  ptr = f ;                                          // oui
  ptr = g ;                                          // oui
}
```

`f` et `g` sont donc de même type.

Les arguments par défaut sont évalués au point de déclaration, et non pas au moment de l'appel. Ainsi :

```
int i=123 ;
int f(int x=i) { cout << x ; }

void main()
{
  int i=0 ;
  f() ;                                              // f(::i)
}
```

écrit `123`, et non pas `0`.

v. Fonctions en ligne

C++ introduit la notion de fonctions *en ligne*. Ce sont des fonctions expansées à chaque utilisation : le compilateur incorpore, à chaque appel, les instructions du corps de la fonction, sans générer les séquences habituelles d'entrée/sortie de fonctions. `inline` est le mot clé utilisé pour définir une fonction en ligne.

Les fonctions en ligne sont intéressantes dans le cas de petites fonctions, pour lesquelles le coût de l'appel (en temps et en volume de code) n'est pas négligeable face au coût du traitement.

Les fonctions en ligne de C++ sont à rapprocher des macros de C, mais la sémantique de l'appel d'une fonction en ligne est identique à celle d'une fonction classique. Les fonctions en ligne ne sont donc pas affectées par les effets de bord bien connus qui perturbent l'utilisation des macros [CTP][FLR].

Il n'y a cependant aucune garantie d'expansion d'une fonction en ligne : le compilateur peut, s'il le juge nécessaire, l'implémenter comme une fonction classique. Ce peut être le cas si la fonction est récursive, si elle est trop volumineuse ou si son adresse est utilisée dans le programme.

La comparaison suivante montre les différences de comportement, au niveau du typage et des effets de bord, entre les macros et les fonctions en ligne :

en C	en C++
`#define ABS(x) (x)>0?(x):-(x)`	`inline int abs(int x) { return x>0 ? x : -x ; }`
`int i=1, j;`	`int i=1, j;`
`float z1, z2=-2.1;`	`float z1, z2=-2.1;`
`j=ABS(i++);  /*j vaut 2*/`	`j=abs(i++);  //j vaut 1`
`z1=ABS(z2);  /*z1 vaut 2.1*/`	`z1=abs(z2);  //z1 vaut 2.0`

L'utilisation des fonctions en ligne doit être préférée à celles des macros : elles sont mieux intégrées au langage, et respecte le système de type. Cependant, dans certains cas, les macros restent irremplaçables. Gérées par le préprocesseur, elles offrent en effet des possibilités que n'offrent pas les fonctions en ligne :

```
#define MALLOC(nb,type) malloc(nb*sizeof(type))
/* ... */
tab = MALLOC(10,int) ;
```

Les fonctions en ligne étant expansées par le compilateur lui-même, elles ne peuvent pas traiter les mots clés comme de simples littéraux, comme le fait la macro ALLOC ci-dessus avec le mot clé `int`.

De même, la définition d'une macro d'édition d'un message d'erreur :

```
#define ERREUR(msg) printf("erreur en ligne %d : %s",
                                                __LINE__,msg)
```

est intéressante, car le message indique le numéro de la ligne où a eu lieu l'erreur, alors que la fonction en ligne `erreur` :

```
inline void erreur (char *msg)
  { printf("erreur en ligne %d : %s",__LINE__,msg) ; }
```

n'a aucun intérêt, car `__LINE__` représente le numéro de la ligne de la fonction `erreur` contenant l'appel à `printf` .

La portée d'une fonction en ligne est l'unité de compilation dans laquelle elle est définie. Par conséquent, la définition de cette fonction doit se trouver dans chaque unité de compilation qui l'utilise. Pour la partager entre plusieurs fichiers sources, on la définit dans un fichier en-tête commun.

vi. Surcharge des noms de fonctions

C++ permet la surcharge des noms de fonctions[21], c'est-à-dire qu'il offre la possibilité d'avoir plusieurs fonctions de même nom. Ces fonctions doivent se différencier par la liste de leurs paramètres, c'est-à-dire par leur nombre de paramètres, ou par le type de ceux-ci :

---

21 Appelée, par abus de langage *surcharge de fonctions*.

en C

```
int ipower(int x, int y)
{
  /* x^y = x.x.x ... .x */
}

double fpower(double x,
                      double y)
{
  /* x^y = e^(y.Log(x)) */
}

i = ipower(2,16);
z = fpower(3.14,1.1);
```

en C++

```
int power(int x, int y)
{
  /* x^y = x.x.x ... .x */
}

double power(double x,
                      double y)
{
  /* x^y = e^(y.Log(x)) */
}

i = power(2,16);
z = power(3.14,1.1);
```

Les deux fonctions `ipower` et `fpower` peuvent s'appeler toutes deux `power` en C++.

Des alias de type ne sont pas considérés comme des types distincts :

```
typedef int entier ;
void f(int i) { /* ... */ }
void f(entier e) { /* ... */ }    // erreur : redéfinition
```

Ici, le compilateur diagnostique : redéfinition de `f`. De même :

```
void f(char *str) { /* .. */ }
void f(char s[10]) { /* .. */ }   // erreur : redéfinition
```

est illégal, car `char *` et `char []` sont considérés comme un même type.

Pour un appel donné, le compilateur sélectionne la fonction à appeler en comparant le type des paramètres avec ceux des arguments, et en choisissant la *meilleure* correspondance. La résolution des surcharges peut être résumée par les règles suivantes[22] :

— s'il existe une fonction assurant une correspondance exacte ou triviale[23], cette fonction est appelée ;

— sinon, s'il existe une fonction assurant une correspondance à partir d'une promotion en entier[24] ou d'une promotion `float` vers `double`, cette fonction est appelée ;

— sinon, si, en utilisant les autres conversions standard, une correspondance est trouvée, cette fonction est appelée ;

— sinon les conversions définies par l'utilisateur sont mises en œuvre[25].

---

[22] L'algorithme de la résolution des surcharges est détaillé au chapitre *L'algorithme de résolution des surcharges* page 301.

[23] Une conversion triviale est une conversion du type tableau vers pointeur, fonction vers pointeur de fonction ou type *T* vers `const` *T*.

[24] Une promotion en entier est une conversion d'un type entier ou booléen en un type entier de taille suffisante pour qu'aucune altération n'affecte les valeurs converties.

La *signature* d'une fonction est l'ensemble des informations qui participent à la résolution de la surcharge de la fonction. Le nom de la fonction, le type de ses paramètres font partie de sa signature. Par contre, son type de retour n'en fait pas partie. Par conséquent, il est interdit d'avoir deux fonctions de même nom si elles se différencient uniquement par leur type de retour :

```
void f(int) ;
int f(int) ;                                    // illégal

void g(int) ;
int g(void) ;                                   // ok
```

Si plusieurs fonctions résolvent la surcharge, il y a ambiguïté et l'appel est rejeté par le compilateur :

```
int f(char) ;
int f(double) ;
int i = f(1) ;          // ambigu : f(char) ou f(double) ?
int j = f(1.) ;               // d'accord, c'est f(double)

int g() ;
int g(int=0) ;
int i = g() ;          // erreur, ambigu : g() ou g(int) ?
int j = g(1) ;                    // d'accord, c'est g(int)
```

vii. Déclaration de fonctions C

Le mécanisme de surcharge de fonctions est géré par un *substantypage* au niveau des noms de fonctions. Le compilateur génère pour chaque fonction un nom (celui sous lequel la fonction est connue de l'éditeur de liens) à partir de sa signature. Le nom de la fonction :

```
void f(int i) ;
```

ressemble à quelque chose comme `@f$qi`. Si son paramètre avait été de type `char`, son nom aurait été différent, par exemple `@f$qzc`. De même, si elle avait eu deux paramètres de type `int`, son nom aurait été encore différent, par exemple `@f$qii`.

Ainsi, le problème de l'identification de la bonne fonction à appeler est-il résolu à la compilation et passe inaperçu à l'édition des liens : à une fonction surchargée deux fois correspondra deux fonctions de noms différents, qui seront traitées comme deux fonctions distinctes par l'éditeur de liens.

Un problème se pose alors lorsqu'une fonction C (c'est-à-dire ayant été compilée par un compilateur C) est appelée dans un programme C++. En effet, le substantypage n'existe pas en C, et une fonction :

---

[25] Voir *Conversions de l'utilisateur* page 99.

```
void f(int i) ;
```

est simplement nommée `f` ou `_f`. Mais comme C++ l'appelle `@f$qi`, il n'y a pas concordance de nom, ce qui conduit à une erreur à l'édition des liens.

Afin de résoudre ce problème, les fonctions C doivent être déclarées dans un bloc `extern "C"`, de la façon suivante :

```
extern "C" void f(int) ;
```

ou :

```
extern "C"
{
  void f(int) ;
  int g(char) ;
}
```

Cela signifie que les fonctions `f` et `g` ne doivent pas être substantypées par le compilateur C++ : elles auront pour nom le même que celui qui leur aurait été assigné par un compilateur C. En contrepartie, les fonctions `f` et `g` n'acceptent pas d'être surchargées.

Cette ouverture est particulièrement intéressante pour utiliser les fonctions de la bibliothèque standard C dans un programme C++ :

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

Cette construction déclare les fonctions d'entrée/sortie de la bibliothèque standard C dans un programme C++. Ces fonctions peuvent être ensuite appelées normalement[26].

Au même titre que les symboles spéciaux `__LINE__` et `__FILE__` gérés par le préprocesseur, il existe un symbole `__cplusplus` défini par tout compilateur C++ (et non défini bien sûr par les compilateurs C). Il facilite l'écriture de code commun aux deux langages :

```
#ifdef __cplusplus
  extern "C" {
#endif
#include <stdio.h>
#ifdef __cplusplus
  }
#endif
```

[26] Une autre solution serait de recompiler la bibliothèque C avec un compilateur C++. Mais la bibliothèque obtenue ne pourrait plus être utilisée par les programmes C !

Cet extrait déclare les fonctions de `stdio.h` dans un fichier source pouvant être compilé en C comme en C++. Cela est particulièrement intéressant pour réaliser des fichiers en-tête communs à C et C++.

7 - Les types

i. Le type `bool`

C++ dispose d'un type booléen, le type `bool`, dont les valeurs sont `true` et `false`.

Sa taille dépend de l'implémentation. Une seule certitude : une valeur booléenne peut toujours être stockée dans un champ de bits.

Les conditions du `if`, `while`, `do`, `for` et de l'expression conditionnelle `? :` sont des expressions de type `bool`.

Les opérateurs relationnels[27] ont un résultat booléen. Les opérateurs logiques[28] ont leurs opérandes et leur résultat de type `bool`.

Il existe une conversion implicite du type `bool` vers les types arithmétiques[29] : `false` est converti en `0` et `true` en `1`.

Réciproquement, il existe une conversion implicite des types arithmétiques vers le type `bool` : `0` est convertie en `false`, et toute valeur non nulle est convertie en `true`.

Introduites pour des raisons de compatibilité avec le langage C, l'utilisation de ces conversions est à proscrire. Elles nuisent à la clarté des programmes, et reflètent de toute façon un style de programmation assez pauvre. Elles sont vouées à devenir obsolètes et doivent être réservées à des cas exceptionnels[30] :

```
{
  int i, j, k ;
  bool b ;
  b = i>j ;          // affectation de deux booléens : bien
  k = i<j ;              // conversion bool->int : à éviter
  while ( b ) /* ... */                          // bien
  if ( k ) /* ... */    // conversion int->bool : à éviter
}
```

---

27 ==, !=, <, <=, > et >=.

28 &&, || et !.

29 Les types arithmétiques sont `char`, `short`, `int`, `long` et les types qui s'en déduisent avec `const`, `volatile`, `signed` et `unsigned`.

30 Il existe aussi une conversion du type pointeur vers le type `bool`, dans laquelle un pointeur `NULL` est convertit en `false` et un pointeur différent de `NULL` en `true`. Dans le même registre, on peut citer l'opérateur ++ qui, appliqué à une variable de type `bool`, lui donne la valeur `true` (par contre, l'opérateur `--` n'est pas défini !). Tout cela a été introduit pour des raisons de compatibilité avec C, et l'utilisation en est fortement déconseillée (elle est d'ores et déjà obsolète).

ii. Les références

Un nouveau type dérivé a été introduit en C++ : le type référence.

Si *T* est un type, *T* `&` est le type référence vers *T*.

L'objectif est de fournir une alternative aux pointeurs : une valeur de type référence est une adresse. La différence entre référence et pointeur réside dans leur utilisation respective : hormis l'initialisation, toute opération effectuée sur la référence agit sur l'objet référencé et non pas, comme pour un pointeur, sur l'adresse. Il est même impossible de modifier la valeur d'une référence après son initialisation. On peut voir les références comme des pointeurs *auto-déréférencés*.

Cependant, le type `void &` n'est pas valide, et il est impossible :

— de créer un pointeur sur une référence,

— de créer une référence sur une référence,

— de créer un tableau de références.

Nommage et synonymes

Une référence peut être utilisée pour définir un synonyme d'un identificateur, ou pour donner un nom à un objet qui n'en a pas :

```
int &elem = tab[i] ;
```

`elem` est un nom attribué à l'élément `i` du tableau `tab`. C'est avantageux si `tab[i]` doit être utilisé plusieurs fois.

Une référence doit obligatoirement être initialisée lors de sa déclaration, car c'est le seul moyen de lui donner une valeur.

Dans l'exemple suivant, `tab` est un tableau d'entiers dont le premier élément, `tab[0]`, contient le nombre d'éléments effectifs du tableau. Un nom, `nb`, est donné à `tab[0]` :

en C

```
int tab[100]={5,1,2,3,4,5};
int *nb = &tab[0];
    /*nb pointe vers tab[0]*/

(*nb)++;          /*tab[0]++*/
tab[*nb] = 6;     /*tab[6]=6 /
```

en C++

```
int tab[100]={5,1,2,3,4,5};
int &nb = tab[0];
 //nb est synonyme de tab[0]

nb++;                //tab[0]++
tab[nb] = 6;         //tab[6]=6
```

Paramètres de type référence

Les paramètres de type référence permettent de réaliser un passage par adresse sans utiliser explicitement de pointeur, comme cela doit se faire nécessairement en C[31] :

---

[31] Les paramètres de type référence donnent un équivalent aux paramètres précédés du mot clé `var`, en Pascal.

en C

```
void swap(int *x, int *y)
{
  int z;

  z = *x;
  *x = *y;
  *y = z;
}

int main()
{
  int a=1, b=2;
  swap(&a,&b);
  /*...*/
}
```

en C++

```
void swap(int &x, int &y)
{
  int z;

  z = x;
  x = y;
  y = z;
}

int main()
{
  int a=1, b=2;
  swap(a,b);
  /*...*/
}
```

Le gain au niveau de la lisibilité et de la facilité d'utilisation est évident. Néanmoins, il n'est plus possible, comme en C, de déterminer, en regardant uniquement l'appel d'une fonction, si les arguments vont être modifiés ou non par la fonction.

Références et pointeurs peuvent se combiner. L'exemple suivant montre l'utilisation d'une référence sur un pointeur :

en C

```
void ouvrir(FILE **f,
                    char *nom)
{
  *f = fopen(nom,"r");
  assert(*f!=NULL);
}

int main()
{
  FILE *fic;
  ouvrir(&fic,"fichier");
  /*...*/
}
```

en C++

```
void ouvrir(FILE *&f,
                      char *nom)
{
  f = fopen(nom,"r");
  assert(f!=NULL);
}

int main()
{
  FILE *fic;
  ouvrir(fic,"fichier");
  /*...*/
}
```

La variable `fic` est passée par adresse afin que la fonction `ouvrir` puisse l'initialiser.

Retour de fonction de type référence

Les références permettent de créer des fonctions dont la valeur de retour est une *lvalue*. Dans l'exemple suivant, la fonction `f` renvoie, suivant la valeur de son paramètre, l'adresse de l'une des deux variables `x` ou `y` :

en C

```
int x, y;

int *f(int b)
{
  return b ? &x : &y;
}

*f(0) = 5;    //range 5 dans y
*f(1) = 2;    //range 2 dans x
```

en C++

```
int x, y;

int &f(int b)
{
  return b ? x : y;
}

f(0) = 5;    //range 5 dans y
f(1) = 2;    //range 2 dans x
```

Le gain au niveau de la lisibilité et de la facilité d'utilisation est évident.

iii. Les types enum, struct, union et class

Utilisation des types enum, struct et union

En C++, le nom d'une énumération, d'une structure ou d'une union est, par lui-même, suffisant pour identifier le type. L'ajout des mots clés `enum`, `struct` et `union` est facultatif. Cela allège l'écriture des définitions, déclarations et transtypages :

en C

```
enum E {/*...*/};
struct S {/*...*/};
union U {/*...*/};

enum E e;
struct S s, *p;
union U u;


p=malloc(sizeof(struct S)) ;
```

en C++

```
enum E {/*...*/};
union U {/*...*/};
struct S {/*...*/};

E e;
S s, *p;
U u;
union U v;   //encore valable

p=(S *) malloc(sizeof(S)) ;
```

Cependant, l'utilisation des mots clés `enum`, `struct` et `union` reste nécessaire pour accéder à un type masqué par un objet, une fonction ou un énumérateur :

```
struct S { /* ... */ } ;

void f()
{
  int S=0 ;
  struct S s ;     // impossible de se passer de struct ici
  /* ... */
}
```

Un type enum, struct ou union ne peut pas avoir la même portée qu'un typedef de même nom :

```
struct S { /* ... */ } ;
typedef int S ;              // valide en C, incorrect en C++
```

Un type enum, struct ou union défini localement masque les identificateurs (variables, constantes, fonctions, types, etc.) déclarés dans les blocs englobants. C'est contraire à ce qui se passe en C, où un type local ne masque jamais une variable ou une fonction déclarée dans une région déclarative englobante :

```
int s[10], b ;

void f()
{
  struct s { /* ... */ } ;
  printf("%d ", sizeof(s)==sizeof(int[10])) ;
  printf("%d ", sizeof(s)==sizeof(struct s)) ;
}
```

f écrit 1 0 en C, alors qu'elle écrit 0 1 en C++.

Restrictions sur l'utilisation du type enum

Chaque énumération est un type particulier, différent du type int. Un type énumération ne peut donc prendre ses valeurs que parmi celles énumérées dans sa définition. En particulier, il ne peut pas prendre une valeur arithmétique :

```
enum jour { lun, mar, mer, jeu, ven } ;
enum signe { neg, nul, pos } ;

enum jour j=1 ;              // légal en C, illégal en C++
enum signe s=lun ;           // légal en C, illégal en C++
```

Il existe par contre une conversion implicite de tout type enum vers int :

```
int i=mer ;                                          // i vaut 2
```

Le type d'un énumérateur est son énumération en C++, alors que c'est int en C :

```
enum jour { lun, mar, mer, jeu, ven } ;
/* ... */
printf("%d", sizeof(lun)==sizeof(int)) ;
```

écrit 1 en C, mais peut écrire 0 ou 1 en C++, suivant l'implémentation.

La discrimination entre les différents types d'énumérations d'une part, et entre les énumérations et les entiers d'autre part, peut être exploitée au niveau de la surcharge des fonctions :

```
enum jour { lundi, mardi, mercredi, jeudi, vendredi } ;
enum piece { pion, cavalier, fou, tour, reine, roi } ;

void f(jour) ;
void f(piece) ;
void f(int) ;
```

Ici, les trois fonctions `f` sont distinctes et peuvent coexister dans un même programme. Elles se surchargent, puisqu'elles ont des paramètres de type différent.

Le type `class` et les extensions des structures et des unions

Avec l'orientation objet de C++, l'importance des structures est devenue telle qu'un nouveau type a été créé : le type `class`.

Ce type est très proche du type `struct`. Il est approfondi au chapitre *C++ : un langage à base de classes*, page 43.

Les types structures et unions ont été du même coup enrichis. Ces extensions sont étudiées, au même chapitre, au travers du type `class`.

8 - Les entrées/sorties : alternative aux fonctions de `<stdio.h>`

Bien que toutes les fonctions de `<stdio.h>` soient utilisables en C++[32], la bibliothèque C++ offrent des objets flux impliquant une nouvelle écriture pour les opérations d'entrée/sortie. L'intérêt de ces objets provient d'une meilleure intégration dans le monde des objets, facilitant la personnalisation des entrées/sorties pour les objets créés par le programmeur[33].

Il existe quatre flux prédéfinis, déclarés dans l'en-tête `<iostream>` :

- `cin`, correspondant à `stdin` de `<stdio.h>`, le flux d'entrée standard,
- `cout`, correspondant à `stdout` de `<stdio.h>`, le flux de sortie standard,
- `cerr`, correspondant à `stderr` de `<stdio.h>`, le flux de sortie d'erreur non tamponné,
- `clog`, sans correspondant dans `<stdio.h>`, le flux de sortie d'erreur tamponné.

Des informations sont envoyées dans les flux de sorties à l'aide de l'opérateur `<<`, de la façon suivante :

---

[32] Voir *Déclaration de fonctions C* page 29.

[33] Voir *Surcharge des opérateurs d'entrée/sortie << et >>* page 360.

```
int i=12 ;
                    // envoie "i=12" sur la sortie standard
cout << "i=" << i << '\n' ;

                  // envoie un message sur la sortie d'erreur
cerr << "\a mémoire insuffisante" ;
```

Des informations sont lues sur le flux d'entrée à l'aide de l'opérateur >>, de la façon suivante :

```
float x, y ;

     // initialisation de x et y à partir du flux d'entrée
cin >> x >> y ;
```

Par rapport aux fonctions `printf` et `scanf`, on remarque l'absence de format. La reconnaissance du type des informations lues ou écrites est automatique :

```
cout << 65 ;                                    // écrit "65"
cout << 'A' ;                                    // écrit "A"
```

De même, les pointeurs de type `char *` sont traités différemment des autres pointeurs :

```
int i, *ptr1=&i ;
char *ptr2="Antonin" ;

cout << ptr1                        // écrit l'adresse de i...
     << ptr2 ;          // ... mais écrit la chaîne "Antonin"
```

Il existe néanmoins diverses possibilités de formatage, de changement de base[34], etc.

Les flux peuvent être assignés à des fichiers, ce qui permet d'y écrire avec << et d'y lire avec >>. Ils offrent une alternative aux fonctions `fprintf`, `fscanf`[35]...

Les flux peuvent également être assignés à des zones mémoire, ce qui offre une alternative aux fonctions `sprintf` et `sscanf`[36].

---

34 Voir *La classe ios_base* page 222, et *Manipulateurs* page 224.

35 Voir *La classe ifstream* page 232, et *La classe ofstream* page 233.

36 Voir *La classe istringstream* page 235, et *La classe ostringstream* page 235.

9 - Opérateurs de gestion dynamique de la mémoire

i. Nouvelle possibilité pour gérer dynamiquement la mémoire

C++ dispose de deux opérateurs de gestion de mémoire, `new` et `delete`, qui proposent une alternative à l'utilisation des fonctions `malloc` et `free` de la bibliothèque standard C.

La syntaxe d'une instruction `new` est :

```
new type
```

pour allouer un unique objet, ou bien :

```
new type[n]
```

pour allouer un tableau de *n* objets. *type* est un type quelconque et *n* est une expression arithmétique quelconque. `new` renvoie un pointeur de type *type* *. Dans la seconde expression, `new` alloue une zone pour mémoriser *n* éléments de type *type* et renvoie un pointeur de type *type* * vers le premier élément alloué. Dans les deux cas, si l'allocation échoue, une *exception*[37] `bad_alloc`[38] est lancée.

La syntaxe de l'utilisation de `delete` est :

```
delete adresse
```

pour libérer un unique objet créé par `new`, ou bien :

```
delete[] adresse
```

pour libérer un tableau. La destruction d'un objet avec `delete[]` et celle d'un tableau avec `delete` conduit à un résultat indéfini[39].

En utilisant ces opérateurs, les notations pour la gestion de la mémoire dynamique se trouvent allégées :

en C

```
int *ptr, *tab, n=100;

ptr = malloc(sizeof(int));
tab = malloc(n*sizeof(int));
/*...*/
free(tab);
free(ptr);
```

en C++

```
int *ptr, *tab, n=100;

ptr = new int;
tab = new int[n];
/*...*/
delete[] tab;
delete ptr;
```

---

[37] Voir *La gestion des exceptions* page 154.

[38] Voir *bad_alloc* page 192.

[39] Pour plus d'informations, voir *Allocation dynamique de tableaux* page 250.

La façon dont les opérateurs `new` prédéfinis obtiennent la mémoire dépend de l'implémentation. En particulier, il n'est pas spécifié si `new` fait appel ou non à la fonction `malloc`.

Le type des éléments d'un tableau créé par `new` peut être lui-même un tableau. Pour créer un tableau de `i` éléments, chacun de type *tableau de 10 entiers*, on écrit `new int[i][10]`, et le résultat est de type `int (*)[10]` :

```
int (*ptr)[10] = new int[i][10] ;
```

La taille des éléments alloués doit être connue à la compilation. Par conséquent, seule la première expression entre `[]` peut être variable. Si `i` et `j` sont des variables, seule la dernière expression est correcte :

```
new int[i][j] ;                                    // non
new int[10][j] ;                                   // non
new int[i][10] ;                                   // oui
```

De même :

```
new int [10][i][10] ;                              // non
new int [10][10][i] ;                              // non
new int [i][10][10] ;                              // oui
```

L'intérêt des opérateurs `new` et `delete`, par rapport aux fonctions `malloc` et `free` de la bibliothèque standard C, c'est leur intégration au système de type du langage. C'est d'ailleurs pour cette raison qu'ils ont été introduits, car l'utilisation de `malloc` et `free` pose problème en C++ :

```
int *ptr = malloc(sizeof(int)) ;                   //illégal
```

Cette instruction est illégale, car l'adresse renvoyée par `malloc` est de type `void *`, et ne peut donc pas être affectée à une variable de type `int *`. Un *cast* lourd et peu élégant est ici nécessaire :

```
int *ptr = (int *) malloc(sizeof(int)) ;
```

Il existe également deux *opérateurs de placement* prédéfinis, dont l'objectif est de permettre la création d'objets à des adresses connues. La syntaxe d'une expression de placement est :

```
new(adresse) type
```

et :

```
new(adresse) type[n]
```

Ces opérateurs ne font rien d'autre que de renvoyer l'adresse qui leur est fournie entre parenthèses. Exemple d'utilisation :

De façon générale, on appelle *opérateur de placement* une fonction `operator new` ou `operator new[]` munie d'un certain nombre de paramètres supplémentaires, de type quelconque.

Des opérateurs de placement peuvent être définis par l'utilisateur :

```
void *adr[100] ;

void *operator new(size_t, int i) { return adr[i] ; }
void *operator new[](size_t, int i) { return adr[i] ; }
```

Ils s'utilisent de la façon suivante :

```
int *ptr1, *ptr2;
ptr1 = new(1) int;              // operator new(sizeof(int),1)
ptr2 = new(2) int[9];    // operator new[](sizeof(int)*9,2)
```

Dans le premier cas, la fonction `operator new` reçoit `sizeof(int)` et la valeur `1` en paramètre. Dans le second cas, la fonction `operator new[]` reçoit `sizeof(int)*9` et la valeur `2` en paramètre.

Cependant, les deux opérateurs de placement prédéfinis :

```
void *operator new(size_t, void *) ;
void *operator new[](size_t, void *) ;
```

ne peuvent pas être redéfinis par l'utilisateur.

Cette possibilité d'ajouter de nouveaux paramètres n'existe pas pour les opérateurs `delete`, qui ne peuvent pas avoir plus d'un paramètre[41].

[41] Cette remarque ne s'applique qu'aux opérateurs `delete` globaux, car les opérateurs `delete` définis au niveau d'une classe peuvent avoir un second paramètre de type `void *` (voir *Les opérateurs new et delete* page 89).

II - C++ : UN LANGAGE A BASE DE CLASSES

Les classes et les structures étant deux types très voisins, ce chapitre ne traite que des classes. Sauf indications contraires, tout ce qui est dit sur les classes s'applique aux structures. Quant aux unions, le point est fait au paragraphe *Le point sur les unions*, page 78.

Sauf indications contraires[42], le terme *objet* désigne à partir de maintenant une lvalue dont le type est une classe ou une structure. Objet est donc synonyme d'*instance de classe* ou d'*instance de structure*.

1 - Les classes

Le type classe est une extension du type structure du langage C. Définir une classe, c'est spécifier ses membres. Une définition de classe est semblable à celle d'une structure, si ce n'est que le mot clé `struct` est remplacé par le mot clé `class` :

```
class Pixel
{
  short x, y ;
  unsigned color ;
} ;                          // définition d'une classe Pixel
```

Déclarer une classe, c'est annoncer l'existence de la classe, sans forcément spécifier son contenu. Une déclaration de classe est semblable à celle d'une structure, si ce n'est que le mot clé `struct` est remplacé par le mot clé `class` :

```
class Pixel ;               // déclaration d'une classe Pixel
```

Une classe déclarée et non définie peut être utilisée dans toute expression ne nécessitant la connaissance ni de ses membres, ni de sa taille :

```
class C ;
C c1 ;                      // non, la taille est nécessaire
extern C c2 ;                                   // d'accord
C *ptr ;                                         // correct
ptr = new C ;               // non, la taille est nécessaire
```

---

42 Comme dans l'expression *objet de type prédéfini*.

Comme pour accéder aux champs d'une structure, on accède aux membres[43] d'une classe à l'aide des opérateurs . et ->.

Les principales nouveautés des classes et des structures de C++ par rapport aux structures du langage C sont :

— la possibilité de définir des membres constants et des membres de type fonction (et non pas simplement de type *pointeur de fonction*),
— la possibilité de définir des membres partagés par tous les objets d'un même type,
— la possibilité de définir des types imbriqués,
— la possibilité de masquer une partie des membres aux utilisateurs du type.

La principale différence entre les classes et les structures de C++ se situe au niveau des droits d'accès implicite aux membres. Les membres d'un objet de type structure sont accessibles par tous ceux qui ont accès à l'objet : tout *utilisateur* de l'objet a accès à ses membres. Par contre, les membres d'un objet de type classe ne sont accessibles que de l'*intérieur* de l'objet, c'est-à-dire que par d'autres membres de l'objet. Autrement dit, seul le *concepteur* de la classe peut y accéder.

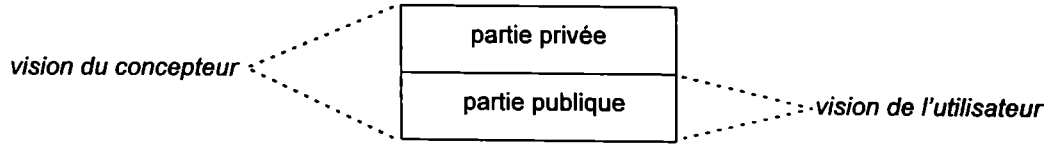
Les membres d'une structure sont dits *publics*, alors que ceux d'une classe sont dits *privés*.

Ainsi la classe `Pixel` ci-dessus n'a aucun intérêt, puisque l'utilisateur d'un pixel ne peut pas accéder à ses membres :

```
Pixel p ;
p.x=1 ;                                  // illégal, x est privé
```

Partie publique et partie privée

Le concepteur d'une classe peut modifier les droits d'accès aux membres et définir pour la classe une partie publique et une partie privée[44].



Cette modification de l'accès est faite à l'aide des mots clés `public` et `private` :

---

[43] On parle de *membre* de classe et de structure en C++, alors qu'on dit plutôt *champ* de structure en C.

[44] Il existe également une partie *protégée*, liée à la notion d'héritage (voir *Contrôle d'accès protégé* page 103).

```
class Chainon
{
    Chainon *precedent, *suivant ;        // privé implicite

  public :
    char info[80] ;                                  // public
} ;
```

Les membres situés après le mot clé `public` sont publics jusqu'à la fin de la classe ou jusqu'à la rencontre du mot clé `private`. De même, les membres situés après le mot clé `private` sont privés jusqu'à la fin de la classe ou jusqu'à la rencontre du mot clé `public`. Par conséquent, la définition précédente est équivalente à :

```
struct Chainon
{
    char info[80] ;                  // implicitement public

  private :
    Chainon *precedent, *suivant ;                   // privés
} ;
```

Là encore, les membres `precedent` et `suivant` ne peuvent être utilisés par personne et ne sont donc d'aucune utilité.

Le contrôle d'accès s'effectue entre classes et non entre objets : deux objets de même type ne peuvent rien se cacher.

2 - Fonctions membres

Les fonctions membres[45] d'un objet ont accès à tous les membres de l'objet, qu'ils soient publics ou privés.

Dans une définition de classe, une fonction membre peut être soit complètement définie, auquel cas elle est considérée en ligne[46] :

---

45 On parle de *fonction membre* en C++, alors qu'on rencontre, dans d'autres langages, le terme de *méthode*.

46 Ce sont généralement, pour des raisons de lisibilité, de petites fonctions, ce qui justifie parfaitement le fait qu'elles soient en ligne.

```
class Pixel
{
    short x, y ;
    unsigned color ;

  public :
    short &X() { return x ; }              // X est inline
    short &Y() { return y ; }              // Y est inline
    unsigned &Color()                    // Color est inline
      { return color ; }
} ;
```

soit seulement déclarée :

```
class Pixel
{
    short x, y ;
    unsigned color ;

  public :
    short &X() ;                        // X n'est pas inline
    short &Y() ;                        // Y n'est pas inline
    unsigned &Color() ;               // Color n'est pas inline
} ;
```

auquel cas sa définition s'effectue après celle de la classe, en utilisant l'opérateur de résolution de portée `::` pour préciser à quelle classe appartient la fonction :

```
short &Pixel::X() { return x ; }
short &Pixel::Y() { return y ; }
unsigned &Pixel::Color() { return color ; }
```

L'accès aux fonctions membres se fait suivant la même syntaxe que l'accès aux variables membres :

```
Pixel p ;
p.X() = p.Y() = 1 ;
p.color = 8 ;                      // erreur, color est privé
p.Color() = 16 ;                                  // d'accord
```

Le corps des fonctions membres est dans la portée de tous les membres. Par conséquent, les fonctions membres peuvent utiliser tous les membres de la classe, quel que soit l'ordre des déclarations :

```
class Pixel
{
  public :
    short &X() { return x ; }                    // correct
    short &Y() { return y ; }                    // correct
    unsigned &Color() { return color ; }         // correct

  private :
    short x, y ;
    unsigned color ;
} ;
```

Ici, les membres `x`, `y` et `color` sont utilisés dans les fonctions `X`, `Y` et `Color` avant même d'être déclarés.

La classe (et non pas seulement des pointeurs et des références vers la classe courante) peut être utilisée dans l'expression même de sa définition. C'est souvent le cas pour les paramètres et le type de retour des fonctions membres. Il est cependant interdit de définir une classe récursive :

```
class Pixel
{
    Pixel symetrique ;      // erreur, définition récursive
    Pixel *conjugue ;                          // d'accord

  public :
    void Init(Pixel) ;                           // correct
    Pixel Conjugue() ;                    // correct aussi
} ;
```

Le pointeur `this`

Chaque fonction membre dispose d'un paramètre caché dont la valeur est l'adresse de l'objet pour lequel cette fonction a été appelée. Il permet à la fonction d'accéder aux membres de l'objet pour lequel elle a été appelée. Ce pointeur peut être explicitement désigné par le mot clé `this`. `this` est un pointeur constant dont la valeur est l'adresse de l'objet lui-même.

```
class Complexe
{
    double x, y ;

  public :
    double &X() { return x ; }
    double &Y() { return y ; }
} ;
```

est équivalent à :

```
class Complexe
{
    double x, y ;

  public :
    double &X() { return this->x ; }
    double &Y() { return this->y ; }
} ;
```

`this` est ici de type `Complexe * const`.

`this` ne peut pas être explicitement déclaré. Etant de type `* const`, sa valeur ne peut pas être modifiée.

3 - Constructeurs

Un constructeur est une fonction membre spéciale qui est appelée automatiquement juste après toute création d'objet. Il porte le même nom que la classe, et n'a pas de type de retour.

Un constructeur sert à initialiser l'objet au moment de sa création :

```
enum Colors { VIOLET, BLEU, VERT, JAUNE, ORANGE, ROUGE } ;

class Pixel
{
    int x, y ;
    Colors color ;

  public :
    Pixel(int xx, int yy, Colors c=ROUGE)
                              { x = xx; y = yy; color = c; }
  /* ... */
} ;
```

Lors de la définition de l'objet, les valeurs d'initialisation, qui sont passées en arguments au constructeur, sont mentionnées entre parenthèses après le nom de l'objet[47] :

```
Pixel p1(1,2,JAUNE) ;
Pixel p2(1,1) ;                          // p2 vaut (1,1,ROUGE)
```

On peut aussi utiliser le signe = suivi explicitement de l'appel du constructeur[48] :

---

[47] Cette syntaxe est valable aussi pour les objets de type prédéfini. Par exemple :

```
int i(2) ;
```

est équivalent à :

```
int i=2 ;
```

[48] Cette syntaxe est valable, là encore, pour les objets de type prédéfini :

```
Pixel p1 = Pixel(1,2,JAUNE) ;        // Pixel p1(1,2,JAUNE)
Pixel p2 = Pixel(1,1) ;     // équivalent à : Pixel p2(1,1)
```

Une classe peut avoir plusieurs constructeurs qui doivent alors se différencier, comme n'importe quelles fonctions surchargées, par la liste de leurs paramètres :

```
class Pixel
{
  public :
    Pixel(Colors c) { x = y = 0 ; color = c ; }
    Pixel(int xx, int yy, Colors c=ROUGE)
                          { x = xx ; y = yy ; color = c ; }
  /* ... */
} ;
```

Dans le cas où une seule valeur d'initialisation est mentionnée, il est alors possible d'omettre l'appel explicite du constructeur, et on retrouve l'initialisation classique des objets de type prédéfini[49] :

```
Pixel p=VERT ;                  // similaire à Pixel p(VERT)
```

Même principe pour la création d'objets dynamiques : les arguments sont fournis au constructeur dans l'expression `new`. La syntaxe est la suivante :

```
Pixel *p1 = new Pixel(1,2) ;        // *p1 vaut (1,2,ROUGE)
Pixel *p2 = new Pixel(VERT) ;        // *p2 vaut (0,0,VERT)
```

Cette notation est aussi valable pour les types prédéfinis :

```
int *ptr = new int(3) ;
```

crée un entier et y range la valeur `3`.

Un constructeur peut être utilisé pour créer explicitement un objet temporaire sans nom :

```
int i=int(2) ;
```
est équivalent à :
```
int i=2 ;
```

[49] Cependant,
```
Pixel p = VERT ;
```
et
```
Pixel p(VERT) ;
```
n'ont pas exactement la même sémantique. Voir *Constructeurs de conversion* page 99.

```
void placer(Pixel) ;

int main()
{
  /* ... */
  for (i=10,j=50 ; i<80 ; i++,j--)
                                    placer(Pixel(i,j,BLEU)) ;
  /* ... */
}
```

ce qui est similaire à :

```
int main()
{
  /* ... */
  for (i=10, j=50 ; i<80 ; i++, j--)
  {
    Pixel p(i,j,BLEU) ;
    placer(p) ;
  }
  /* ... */
}
```

i. Constructeur par défaut

Un constructeur qui peut être appelé sans argument est dénommé *constructeur par défaut*. C'est donc soit un constructeur qui n'a pas de paramètre, soit un constructeur dont tous les paramètres possèdent une valeur par défaut. Le constructeur par défaut joue un rôle particulier, car il est sollicité automatiquement dans différentes situations[50].

Voici un exemple de constructeur par défaut :

```
class Pixel
{
    int x, y ;
    Colors color ;

  public :
    Pixel(int xx=0, int yy=0, Colors c=JAUNE) ;
} ;
```

Toute création d'objet fait appel à un constructeur. Si la classe n'a pas défini de constructeur, tout ce passe comme si un constructeur public était *synthétisé* par le compilateur. Pour une classe `C`, l'en-tête de ce constructeur à la forme `C()`.

---

[50] Voir *Objets membres* page 73, *Tableaux d'objets* page 77 et *Processus de création et de destruction d'objets* page 111.

Le constructeur par défaut synthétisé pour la classe `Pixel` ci-dessous ne réalise aucun traitement. Il autorise simplement la création d'objets sans valeur d'initialisation. Un tel constructeur par défaut s'appelle un *constructeur trivial* :

```
class Pixel
{
    int x, y ;
    Colors color ;

  public :
    int &X() ;
    int &Y() ;
    Colors &Color() ;
} ;
```

Ici, un constructeur `Pixel() { }` est synthétisé, ce qui permet d'écrire :

```
Pixel p ;          // ok, utilise le constructeur synthétisé
Pixel *ptr ;
ptr = new Pixel ;     // utilise le constructeur synthétisé
```

Les valeurs des membres de `p` et de `*ptr` sont donc indéfinies. Le constructeur synthétisé ici est trivial, car `Pixel` est une classe *élémentaire*[51]. Cependant, dans le général, le constructeur par défaut synthétisé n'est pas toujours trivial[52].

Si un ou plusieurs constructeurs sont définis dans la classe, le constructeur par défaut n'est pas synthétisé. Dans ce cas, si aucun constructeur par défaut n'est explicitement défini, toute création d'objets requerra obligatoirement des valeurs d'initialisation :

```
class Pixel
{
    int x, y ;
    Colors color ;

  public :
    Pixel(int, int, Colors) ;
    int &X() ;
    int &Y() ;
    int &Color() ;
} ;
```

---

51 Une classe élémentaire est une classe sans objet membre, sans fonction virtuelle et sans classe de base.

52 Voir des exemples de synthétisations de constructeurs par défaut non triviaux aux paragraphes *Objets membres* page 73, et *Processus de création et de destruction d'objets* page 111.

```
Pixel p1 ;      // illégal, pas de constructeur par défaut
Pixel *ptr1 = new Pixel ;         // illégal (même raison)
Pixel p2(10,-10,BLEU) ;                              // bien
Pixel *ptr2 = new Pixel(10,-10,BLEU) ;                 // ok
```

ii. Constructeur de recopie

Il existe un autre constructeur particulier appelé *constructeur de recopie*. Le constructeur de recopie d'une classe `C` est un constructeur dont le premier paramètre est de type `C&` ou `const C&`[53], et les suivants sont tous munis d'une valeur par défaut.

Le plus souvent, son en-tête est simplement de la forme `C(const C &)`, mais `C(C &)` ou `C(const C &, int =0)` sont également des en-têtes de constructeurs de recopie. Par contre `C(const C &, int)` n'en n'est pas un.

Ce constructeur sert à créer des clones d'objets. Il est appelé implicitement à chaque fois qu'un objet est initialisé avec un objet de même type :

```
class Pixel
{
  public :
    Pixel(int, int, Colors) ;
    Pixel(const Pixel &) ;      // constructeur de recopie
    /* ... */
} ;

Pixel p1(1,2,JAUNE) ;
Pixel p2(p1) ;              // Pixel(const Pixel &) appelé
Pixel p3=p1 ;               // Pixel(const Pixel &) appelé
```

C'est le cas en particulier lorsqu'un objet est passé en argument à une fonction. Le constructeur de recopie est utilisé pour recopier l'argument dans le paramètre :

```
float distance(Pixel p) ;

Pixel p(1,2,JAUNE) ;
float x = distance(p);      // Pixel(const Pixel &) appelé
```

Le constructeur de recopie est appelé ici pour transmettre `p`.

Des objets temporaires peuvent être créés par le compilateur[54]. Dans ce cas, ces objets temporaires sont initialisés par le constructeur de recopie. Dans l'exemple suivant :

---

[53] Il est illégal de déclarer un constructeur ayant un premier paramètre de type `C` ou `const C` et dont tous les autres paramètres sont munis d'une valeur par défaut.

[54] Les règles de créations des objets temporaires dépendent de l'implémentation.

```
Pixel symetrique(int x, int y) ;

Pixel p ;
p = symetrique(2,3) ;        // Pixel(const Pixel &) appelé
```

un objet temporaire est créé pour mémoriser la valeur de retour de `symetrique`, avant de l'affecter à `p`. Le constructeur de recopie de `Pixel` est donc sollicité[55].

Si la classe n'a pas défini de constructeur de recopie, alors tout se passe comme si un constructeur de recopie était synthétisé par le compilateur. Ce constructeur réalise une copie membre à membre de l'objet dans le clone. Pour une classe élémentaire, un tel constructeur de recopie est appelé *constructeur de recopie trivial.*

Dans la plupart des cas, le constructeur de recopie synthétisé a un comportement satisfaisant, ce qui évite au programmeur de prendre en charge l'implémentation du clonage. Il existe cependant des classes pour lesquelles le clonage membre à membre réalisé par le constructeur de recopie est inacceptable. Dans ce cas, il est impératif de définir explicitement un constructeur de recopie correct, puisque l'opération de clonage peut être sollicitée sans avertissement par le système. Ce point important est discuté au paragraphe *Quand et pourquoi doit-on implémenter une duplication personnalisée ?* page 272.

Pour la classe suivante :

```
class Pixel
{
    int x, y ;
    Colors color ;
} ;
```

le constructeur de recopie est synthétisé à la forme :

```
Pixel::Pixel(const Pixel &p)
{
  x = p.x ; y = p.y ; color = p.color ;
}
```

4 - Destructeurs

Un destructeur est une fonction membre spéciale appelée automatiquement juste avant la destruction d'un objet. Il porte le même nom que la classe, précédé du signe ~. Il n'a pas de paramètre ni de type de retour.

Il donne au programmeur une dernière occasion de libérer les ressources acquises par l'objet.

---

55 Il se peut que le compilateur soit suffisamment intelligent pour compiler cette instruction sans générer un appel au constructeur de recopie. Mais en tout état de cause, le renvoi d'un objet par une fonction exige que cet objet ait un constructeur de recopie accessible, même si en définitive ce constructeur n'est pas appelé.

Une classe ne peut avoir qu'un seul destructeur. La classe `String` suivante modélise le type chaîne de caractères :

```
class String
{
    char *str ;

  public :
    String(const char *s)
      {
        str = new char[strlen(s)+1];
        strcpy(str,s);
      }
    ~String() { delete[] str ; }
    /* ... */
} ;
```

Le destructeur `~String` libère la mémoire allouée dans le constructeur.

Si la classe ne définit pas de destructeur, alors tout se passe comme si un destructeur était synthétisé par le compilateur :

```
class Pixel
{
    int x, y ;
    Colors color ;

  public :
    int &X() ;
    int &Y() ;
    Colors &Color() ;
} ;
```

Dans le cas de cette classe élémentaire, le destructeur synthétisé ne réalise aucun traitement. Un tel destructeur s'appelle un *destructeur trivial*. Cependant, en général, le destructeur synthétisé n'est pas trivial[56].

Un destructeur est rarement appelé explicitement. Un tel appel a pour effet de nettoyer l'objet sans récupérer la mémoire qu'il occupe :

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

void *operator new(size_t, void *addr) { return addr ; }
void *get_addr_screen() ;
```

[56] Voir des exemples de destructeurs synthétisés non triviaux aux paragraphes *Objets membres* page 73, et *Processus de création et de destruction d'objets* page 111.

```
void f()
{
  Screen *screen = new(get_addr_screen()) Screen ;
  /* ... */
  screen->~Screen() ;
}
```

Ici, l'opérateur de placement `new` n'alloue pas de mémoire, mais renvoie simplement l'adresse qui lui est passée en argument. Aucun `delete` ne doit donc être effectué sur un pointeur initialisé de cette façon. Le destructeur est toutefois explicitement appelé pour restaurer les ressources acquises par l'objet.

En guise de conclusion provisoire sur les synthétisations, étant donnée la classe :

```
class C
{
  int a ;
} ;
```

les synthétisations du constructeur par défaut, du constructeur de recopie et du destructeur donnent finalement une classe équivalente à :

```
class C
{
    int a ;

  public :
    C() {}                  // constructeur par défaut trivial
    C(const C &c)           // constructeur de recopie trivial
      { a = c.a ; }
    ~C() {}                             // destructeur trivial
} ;
```

5 - Membres constants

i. Constantes membres

Une constante membre se définit à l'aide du mot clé `const`. L'initialisation doit se mentionner sur l'en-tête de chaque constructeur de la classe, de la façon suivante :

```
class Complexe
{
    const bool reel ;
    double x, y ;

  public :
    Complexe(double xx, double yy) : reel(false)
                                     { x = xx ; y = yy ; }
    Complexe(double xx) : reel(true) { x = xx ; y = 0 ; }
  /* ... */
} ;
```

Ici, la constante booléenne `reel` est initialisée tantôt à `true`, tantôt à `false`, suivant que le complexe est réel ou non. La constante prend sa valeur au moment de la création de l'objet et ensuite ne peut plus être modifiée.

La valeur d'une constante membre est inconnue lors de la compilation. Une constante membre ne peut par conséquent pas intervenir dans une expression constante[57] :

```
class C
{
    const int max ;
    int tab[max] ;                                  // illégal

  public :
    C() : max(100) {}
} ;
```

Ici, le tableau ne peut pas être dimensionné avec la constante `max`[58].

ii. Fonctions constantes

Parmi les fonctions membres, mis à part les constructeurs et les destructeurs, on peut distinguer d'une part les fonctions de consultation de l'état de l'objet, et celles qui modifient l'état de l'objet d'autre part. Le mot clé `const`, placé en fin d'en-tête et de prototype, signifie que la fonction en question est une fonction de consultation et qu'elle ne modifie en aucun cas l'état de l'objet :

---

[57] Au sens d'expression constante du langage C, c'est-à-dire d'expression évaluable à la compilation.

[58] Voir une solution de remplacement au paragraphe *Constantes membres statiques* page 59.

```
class Complexe
{
    double x, y ;

  public :
    Complexe(double, double) ;
    void Annule() { x = y = 0 ; }
    double X() const { return x ; }
    double Y() const { return y ; }
} ;
```

L'introduction de cette sémantique supplémentaire permet au compilateur d'autoriser certaines manipulations sur les objets constants :

```
const Complexe c(1,2) ;
double x = c.X() ;                                    // ok
double y = c.Y() ;                                    // ok
```

L'appel des fonctions `X` et `Y` de la constante `c` ne pose pas de problème, puisque ces fonctions sont spécifiées constantes. Par contre, toute exécution d'une fonction ordinaire est illégale :

```
const Complexe c(1,2) ;
c.Annule() ;                                          // erreur
```

Ici, en effet, `c` est une constante et `Annule` est susceptible de la modifier : le compilateur refuse l'appel.

Plus précisément, les fonctions membres constantes et non constantes peuvent être appelées pour un objet non constant, alors que seules les fonctions constantes peuvent être appelées pour un objet constant.

Dans une fonction constante, `this` est de type *pointeur constant vers objet constant.* Ainsi, dans les fonctions `X` et `Y` ci-dessus, `this` est de type `const Complexe * const`.

Une fonction membre non constante peut être surchargée par une fonction constante de même nom, avec la même liste de paramètres : la spécification `const` fait en effet partie de la signature d'une fonction membre. Dans ce cas, pour les objets non constants, c'est la fonction non constante qui est appelée, et pour les objets constants c'est la fonction constante qui est appelée :

```
class Complexe
{
    double x, y ;

  public :
    Complexe () { x = y = 0 ; }
    double &X() { return x ; }
    double &Y() { return y ; }
    double  X() const { return x ; }
    double  Y() const { return y ; }
} ;
```

```
void main()
{
  Complexe c ;
  const Complexe cc ;
  double x ;
  x = c.X() ;           // d'accord, double &Complexe::X()
  x = cc.X() ;         // bien, double Complexe::X() const
  c.X() = x ;           // parfait, double &Complexe::X()
  cc.X() = x ;              // non, modification illégale
}
```

Cette dernière instruction est illégale (et heureusement), puisque :

— la fonction appelée est `Complexe::X() const`, et

— celle-ci ne renvoie pas de lvalue.

Les constructeurs et destructeurs ne peuvent pas être spécifiés constants : ce sont les seules fonctions membres non constantes pouvant être appelées pour des objets constants.

Membres mutables

La notion de constance peut correspondre à deux concepts différents [STR]. Il peut s'agir d'une constance physique, telle que tout membre de l'objet conserve la valeur qui lui a été attribuée par le constructeur, jusqu'à l'appel du destructeur. Mais il peut s'agir aussi d'une constance logique, dans laquelle seule la perception qu'ont les utilisateurs de l'objet qu'il manipule reste invariable. L'objet peut dans ce cas modifier son état interne (mise à jour de compteurs, de drapeaux, de statistiques, etc.) tant qu'il apparaît constant à ses utilisateurs.

Pour implémenter ce concept de constante logique, il est nécessaire de permettre à une fonction constante de modifier certaines variables de l'objet. Cela est possible grâce au mot clé `mutable` : les membres déclarés `mutable` peuvent être modifiés par toute fonction membre, constante et non constante.

```
class Complexe
{
    double x, y ;
    mutable int nb_acces ;

  public :
    Complexe () { x = y = nb_acces = 0 ; }
    double &X() { nb_acces++ ; return x ; }
    double &Y() { nb_acces++ ; return y ; }
    double  X() const { nb_acces++ ; return x ; }   // oui
    double  Y() const { nb_acces++ ; return y ; }   // oui
} ;
```

Ici, le membre `nb_acces` sert à comptabiliser le nombre d'accès à l'objet courant : il est incrémenté à chaque consultation et à chaque modification de la

valeur du complexe. Le membre `nb_acces` est déclaré `mutable`, ce qui autorise sa modification par les fonctions constantes `x` et `y`.

6 - Membres statiques

Il peut être intéressant de définir des membres partagés par toutes les instances d'une même classe. Il suffit pour cela de spécifier les membres `static`.

i. Variables membres statiques

```
class Complexe
{
    static int nb_objet ;              // simple déclaration
    double x, y ;

  public :
    Complexe(double xx, double yy)
                              { x = xx; y = yy; nb_objet++; }
    ~Complexe() { nb_objet-- ; }
  /* ... */
} ;
```

La variable `nb_objet` est ici partagée par tous les objets de type `Complexe`[59]. Elle sert à compter le nombre d'objets `Complexe` en cours d'utilisation.

L'initialisation de cette variable est réalisée hors de la classe, et hors de toute fonction, au niveau global[60], de la façon suivante :

```
int Complexe::nb_objet=0 ;                    // définition
```

Cette initialisation sera prise en compte, comme pour toute variable globale, au début de l'exécution du programme, avant d'entrer dans `main`.

ii. Constantes membres statiques

On peut de la même façon déclarer une constante `static` :

---

[59] On dit que `nb_objet` est une variable de classe, alors que `x` et `y` sont des variables d'instance.

[60] Même si cette variable n'est pas publique.

```
class Complexe
{
    double x, y ;

  public :
    static const double PI ;
    Complexe(double xx, double yy) ;
  /* ... */
} ;
```

et définir sa valeur au niveau global[61], de la façon suivante :

```
const double Complexe::PI=3.14159265358979323385 ;
```

Un membre statique peut s'utiliser comme un membre non statique :

```
Complexe z(1,2) ;
int K = z.PI/4 ;
```

mais peut aussi s'utiliser indépendamment de tout objet[62], de la façon suivante :

```
int K = Complexe::PI/4 ;
```

Un membre (variable ou constante) `static` ne peut pas être `mutable`.

Les constantes membres statiques de type arithmétique peuvent être initialisées dans la classe, et peuvent de ce fait intervenir dans les expressions constantes (au sens *expression évaluable à la compilation*) utilisées dans la classe :

```
class C
{
    static const int max=100 ;
    int tab[max] ;                                    // ok

  public :
    C() {}
} ;

const int C::max ;                    // nécessaire quand-même
```

Une énumération peut être utilisée pour définir d'un coup plusieurs constantes membres statiques :

```
class C
{
  enum { max1=100, max2=1000 } ;
  int tab1[max1], tab2[max2] ;                        // ok
  /* ... */
} ;
```

---

61 Même si cette constante n'est pas publique.

62 A condition bien sûr qu'il soit public.

iii. Fonctions membres statiques

Des fonctions statiques se définissent de la même façon. Ces fonctions ont une existence indépendante de tout objet. Elles ne disposent pas du pointeur `this`, et ne peuvent accéder qu'aux membres statiques de la classe :

```
class Complexe
{
    static int nb_objet ;
    double x, y ;
    static void IncObjet() { nb_objet++ ; }
    static void DecObjet() { nb_objet-- ; }

  public :
    Complexe(double xx, double yy)
                          { x = xx; y = yy; IncObjet(); }
    ~Complexe() { DecObjet() ; }
    static int NbObjet() { return nb_objet ; }
  /* ... */
} ;
```

Les fonctions `IncObjet`, `DecObjet` et `NbObjet` sont statiques. Comme pour les variables et constantes statiques, les fonctions statiques publiques peuvent être utilisées indépendamment de tout objet :

```
nb = Complexe::NbObjet() ;
```

Une fonction membre statique ne peut pas surcharger une fonction membre de même nom munie de la même liste de paramètres :

```
class C
{
  static void f(int) ;
  void f(int) ;                                    // erreur
} ;
```

7 - Pointeurs vers membres

Pour définir un pointeur vers un membre de type `T` appartenant à une classe `C`, il est nécessaire de préciser d'une part le type du membre, `T`, et d'autre part la classe, `C`. Autrement dit, ce pointeur est dédié à la classe `C`. Cela se déclare par :

```
int C::*ptr ;
```

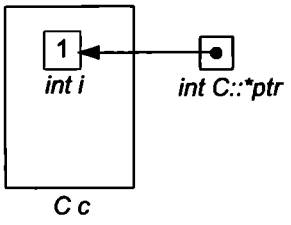
`ptr` est un pointeur susceptible de pointer vers un membre de type `int` de la classe `C`. Mais `ptr` n'est pas a priori un membre de la classe `C`.

Accéder au membre pointé par `ptr` d'un objet `c` se fait par un nouvel opérateur `.*`. Si la classe `C` est définie comme :

```
class C
{
  public :
    int i ;
    void f(int) ;
} ;
```

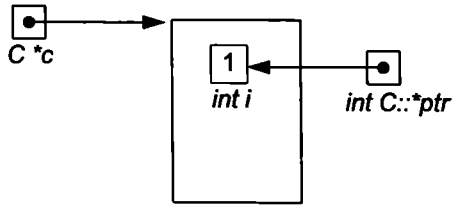
on peut écrire :

```
int C::*ptr ;                    // ptr est un pointeur vers
                                 // un membre de C de type int
C c ;
ptr = &C::i ;                    // ptr pointe maintenant vers
                                 // le membre i de la classe C
c.*ptr = 1 ;          // le membre de c pointé par ptr vaut 1
```



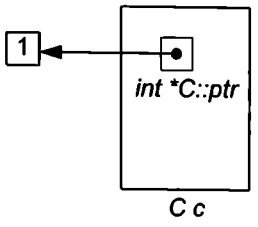
De même, accéder au membre pointé par `ptr` d'une instance pointée par `c` se fait par un nouvel opérateur `->*` :

```
int C::*ptr ;                    // ptr est un pointeur vers
                                 // un membre de C de type int
C *c ;
c = new C ;
ptr = &C::i ;                    // ptr pointe maintenant vers
                                 // le membre i de la classe C
c->*ptr = 1 ;       // le membre pointé par ptr pour l'objet
                    // pointé par c est initialisé à 1
```



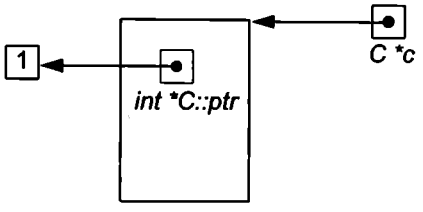
Cela ne doit pas être confondu avec les expressions plus habituelles :

```
C c ;
*c.ptr = 1 ;          // valeur de la zone pointée par ptr,
                                 // membre de c
```



ni avec :

```
C *c ;
c = new C ;
*c->ptr = 1 ;          // valeur de la zone pointée par ptr,
                          // membre de l'objet pointé par c
```



dans lesquels `ptr` est un membre de la classe `C` de type `int *`.

Un pointeur vers une constante entière membre d'une classe `C` se déclare :

```
int C::* const ptr ;
```

Un pointeur de fonction membre se définit de la même façon :

```
void (C::*pf)(int) ;
```

`pf` est un pointeur susceptible de pointer vers une fonction membre (fonction à un paramètre entier et ne renvoyant rien) de la classe `C`. Là encore, `pf` n'est pas a priori un membre de `C`.

Exécuter la fonction pointée par `pf` d'un objet `c` se fait par :

```
void (C::*pf)(int) ;                    // ptr : pointeur vers
                                     // une fonction membre de C
C c ;
pf = &C::f ;                        // pf pointe maintenant vers
                          // la fonction membre f de la classe C
(c.*pf)(1) ;                 // la fonction membre de c pointée
                                        // par pf est exécutée
```

Exécuter la fonction pointée par `pf` d'un objet pointé par `c` se fait par :

```
void (C::*pf)(int) ;
C *c ;
c = new C ;
pf = &C::f ;                    // pf pointe maintenant vers
                        // la fonction membre f de la classe C
(c->*pf)(1) ;                   // pour l'objet pointé par c,
                    // la fonction pointée par pf est exécutée
```

Un pointeur vers une fonction constante membre d'une classe `C`, avec un paramètre et un type de retour entier, se déclare :

```
int (C::*pf)(int) const ;
```

Un pointeur vers membre ne peut pas pointer sur un membre statique, ni sur un membre de type référence. Inversement, le type *référence vers membre* n'existe pas.

L'exemple suivant utilise les pointeurs de membres pour implémenter de façon élémentaire un mécanisme de retardement d'exécution de tâches. Une classe `Fenetre` possède deux services `Rafraichir` et `Imprimer` dont l'exécution peut être programmée pour être exécutée plus tard. Ces fonctions `Rafraichir` et `Imprimer` sont mémorisées par un objet `Ordonnanceur` qui les déclenche lorsque le système le lui demande. L'ordonnanceur possède un tableau de pointeurs vers les fonctions qui ont été programmées, et un tableau de pointeurs vers les objets propriétaires de ces fonctions :

```
class Fenetre
{
    /* ... */

  public :
    void Rafraichir() ;
    void Imprimer() ;
    /* ... */
} ;

const int MAX=100 ;

class Ordonnanceur
{
  typedef void (Fenetre::*Fonction)() ;
  Fonction tabf[MAX] ;              // fonctions programmées
  Fenetre *tabo[MAX] ;                      // propriétaires
  int itab, otab ;

  public :
    Ordonnanceur() : itab(0), otab(-1) {}
    void Programmer( Fenetre &, Fonction) ;
    void Declencher() ;
} ;
```

```
void Ordonnanceur::Programmer( Fenetre &f,
                                                Fonction action)
{
  tabo[itab] = &f ;
  tabf[itab] = action ;
  itab = (itab+1)%MAX ;
}

void Ordonnanceur::Declencher()
{
  if (otab<itab)
  {
    otab = (otab+1)%MAX ;
   (tabo[otab]->*tabf[otab])() ;
  }
}

void main()
{
  Ordonnanceur o ;
  Fenetre f1, f2 ;
  o.Programmer(f1, &Fenetre::Rafraichir) ;
  o.Programmer(f2, &Fenetre::Imprimer) ;
  o.Programmer(f1, &Fenetre::Rafraichir) ;
  o.Programmer(f2, &Fenetre::Imprimer) ;
  /* ... */
  o.Declencher() ;
  /* ... */
}
```

8 - Types imbriqués

Des types peuvent être définis dans une classe :

```
class Pixel
{
  public :
    typedef unsigned Colors ;

  private :
    enum Status { in, out } ;
    long p ;
    Colors color ;
```

```
  public :
    void Init(short xx, short yy, Colors c) ;
    short GetX() ;
    short GetY() ;
    Colors GetColor() ;
} ;
```

Ils sont alors locaux à la classe. Ici, le type `Colors` est local à la classe `Pixel`. Les types imbriqués ne peuvent être utilisés en dehors de la classe englobante sans une qualification de portée explicite[63] :

```
Pixel::Colors c=1 ;              // qualification nécessaire
Pixel p ;
p.Init(10,10,c) ;
```

L'accès aux types imbriqués suit les mêmes contrôles d'accès que ceux des membres :

```
Pixel::Status status ;          // illégal, Status est privé
```

L'exemple suivant définit une énumération dans la classe `Pixel` :

```
class Pixel
{
  public :
    enum Colors
            { VIOLET, BLEU, VERT, JAUNE, ORANGE, ROUGE } ;

  private :
    long p ;
    Colors color ;

  public :
    void Init(short xx, short yy, Colors c) ;
    short GetX() ;
    short GetY() ;
    Colors GetColor() ;
} ;
```

ce qui peut s'utiliser comme :

```
Pixel p ;
p.Init(10,10,Pixel::BLEU) ;
```

---

[63] C'est contrairement à ce qui se passe en C, pour lequel un type imbriqué est dans la région déclarative globale :

```
struct S1
{
  struct S2 { /* ... */ } s ;
} ;
struct S2 s ; // valide en C, illégal en C++
struct S1::S2 s ; // valide en C++
```

Le corps de chaque fonction membre est dans la portée des types imbriqués. Cela permet aux types locaux d'être utilisés dans toute fonction membre, quel que soit l'ordre des déclarations. On peut écrire :

```
class C
{
  int f() { T i ; }
  typedef int T ;
} ;
```

mais on ne peut pas écrire :

```
class C
{
  T f() ;                          // erreur : T hors de portée
  typedef int T ;
} ;
```

car `T` est défini après `f`. Il faut écrire :

```
class C
{
  typedef int T ;
  T f() ;                                                  // oui
} ;
```

En particulier, les classes peuvent être imbriquées. Dans ce cas, la classe englobante n'a aucun droit particulier sur la classe englobée, et inversement :

```
class A
{
  int x ;

  class B
  {
    int y ;
    int f(A a)
      { return a.x ; }          // illégal, A::x est privé
  }

  int g(B b) { return b.y ; }   // illégal, B::y est privé
} ;
```

Le seul privilège de la classe imbriquée est de pouvoir utiliser les types, énumérateurs et membres statiques de la classe englobante sans qualification de portée explicite :

```
class Segment
{
  public :
    typedef long Coord ;

    class Pixel
    {
      Coord x, y ;          // implicitement Segment::Coord
      /* ... */
    } *a, *b ;
  /* ... */
} ;
```

L'intérêt de cette construction est de limiter les conflits de noms de portée globale. De plus, si `Pixel` avait été définie en partie privée, elle aurait été invisible hors de la classe `Segment`.

La classe englobée peut n'être que déclarée dans la classe englobante. Elle doit être alors définie ensuite à l'extérieur, en utilisant l'opérateur `::`. Dans ce cas, les types, énumérateurs et membres statiques de la classe englobante doivent être qualifiés si nécessaire.

L'exemple suivant est équivalent au précédent :

```
class Segment
{
  public :
    typedef long Coord ;

    class Pixel ;
    Pixel *a, *b ;
  /* ... */
} ;

class Segment::Pixel
{
  Segment::Coord x, y ;     // qualification nécessaire ici
  /* ... */
} ;
```

9 - Amis

Le concept d'ami est en rapport direct et exclusif avec les contrôles d'accès aux membres de classe. Il introduit une entorse aux règles simples et claires de

protection des membres. Ce concept n'est pas fondamental, et il est clair qu'une programmation propre doit limiter son utilisation au strict nécessaire[64].

i. Fonction amie

Une fonction amie d'une classe est une fonction qui, bien que n'appartenant pas à la classe, est autorisée à accéder à tous ses membres. Une fonction amie se déclare ou se définit dans la classe, indifféremment en partie privée ou en partie publique. On utilise à cet effet le mot clé `friend` :

```
class Tableau
{
    friend void Print(const Tableau &) ;

    int *tab, taille ;

  public :
    Tableau(int n) { tab = new int[taille=n] ; }
    int &Element(int i)
                  { assert (i>=taille) ; return tab[i] ; }
  /* ... */
} ;

void Print(const Tableau &t)
{
  for (int i=0 ; i<t.taille ; i++)
    printf("%d ", t.tab[i]) ;
}
```

La fonction `Print` a besoin d'accéder, pour des raisons de performances, directement au pointeur `tab` plutôt que de passer par la fonction `Element`. Or `tab` est privé et `Print` n'appartient pas à la classe `Tableau`. `Print` est donc déclarée amie de `Tableau`.

Comme les fonctions membres, les fonctions amies définies dans la classe sont en ligne. Une fonction amie peut aussi être membre d'une autre classe :

---

64 C'est sûrement pour faciliter l'écriture des opérateurs sous forme de fonctions globales, notamment les opérateurs d'entrée/sortie << et >>, qu'a été introduit le concept d'ami dans le langage.

Le problème des opérateurs d'entrée/sortie, c'est qu'ils ne peuvent pas être définis comme fonctions membres, puisque les classes streams sont prédéfinies dans la bibliothèque standard. Ils sont donc définis par l'utilisateur sous forme de fonctions globales. Or ces opérateurs ont généralement besoin d'accéder à la totalité des informations composant l'objet. Donc très souvent, ces opérateurs sont des fonctions amies (voir *Surcharge des opérateurs d'entrée/sortie << et >>* page 360).

```
class Tableau ;

class Window
{
  void Print(const Tableau &) ;
  /* ... */
} ;

class Tableau
{
  friend void Window::Print(const Tableau &) ;
  /* ... */
} ;
```

Ici, la fonction `Print` de la classe `Window` est amie de la classe `Tableau`.

ii. Classe amie

Une classe amie d'une classe `C` est une classe qui est autorisée à accéder à tous les membres de `C`. Une classe amie se déclare ou se définit indifféremment en partie privée ou en partie publique :

```
class Chainon
{
    friend class Liste ;          // Liste amie de Chainon
    char *info ;
    Chainon *suivant ;
} ;

class Liste
{
    Chainon *tete ;

  public :
    Liste()
          { tete = new Chainon ; tete->suivant = NULL ; }
  /* ... */
} ;
```

La classe `Liste` peut accéder à tous les membres de `Chainon`. Et effectivement, le constructeur de `Liste` accède au membre privé `suivant` de `Chainon`. La classe `Chainon` ne peut d'ailleurs être utilisée que par la classe `Liste`, puisque tous ses membres sont privés.

```
class A
{
  enum { N=100 } ;
  friend class B ;
} ;
```

```
class B
{
  float tab[A::N] ;
} ;

class C
{
  int t[A::N] ;              //erreur, C n'est pas amis de A
} ;
```

Ici, la déclaration de B utilise la constante N de A.

La relation d'amitié n'est pas transitive :

```
class A
{
  friend class B ;
  int i ;
} ;

class B
{
  friend class C ;
} ;

class C
{
  void f(A a) { a.i++ ; }                          // illégal
} ;
```

10 - Processus de création/destruction des objets

i. Objets automatiques

Pour un objet automatique, le constructeur est appelé lors de la définition de l'objet, et le destructeur lors de la sortie de sa portée :

```
int main()
{
  /* ... */
  Pixel p(1,1,VIOLET) ;          // constructeur appelé ici

  /* ... */
}                                 // destructeur appelé ici
```

Pour avancer l'appel d'un destructeur, il est toujours possible de créer des blocs supplémentaires :

```
class Fichier
{
    FILE *f ;

  public :
    Fichier(char *nom) { f = fopen(nom,"r") ; }
    ~Fichier() { fclose(f) ; }
    /* ... */
} ;

int main()
{
  {
    Fichier f("donnees") ;
    /*
       lecture du fichier ici
       ...
    */
  }                          // f.~Fichier() est appelé ici
  // le fichier est fermé ici
  /*
     ...
  */
}
```

ii. Objets statiques

Pour un objet statique global, le constructeur est appelé en tout début d'exécution, et le destructeur est appelé en fin d'exécution :

```
Pixel p1(1,-2,ORANGE) ;

int main()
{
  /* ... */
}

Pixel p2(-10,2,VERT) ;

int f()
{
  /* ... */
}
```

Ici, le constructeur de `Pixel` est appelé deux fois avant d'entrer dans `main`, une fois pour `p1` et une fois pour `p2`. Le destructeur de `Pixel` est appelé deux fois après la sortie de `main`.

Pour un objet statique local, le constructeur est appelé au plus tard lors du premier appel, et le destructeur est appelé en fin d'exécution :

```
int f()
{
  static Pixel p(-10,2,VERT) ;
  /* ... */
}
```

iii. Objets dynamiques

Pour un objet dynamique, le constructeur est appelé juste après l'exécution de l'opérateur `new`, et le destructeur juste avant l'exécution de l'opérateur `delete` :

```
{
  Pixel *p = new Pixel(2,1,ROUGE) ;          /* allocation,
                              puis appel du constructeur */
  /* ... */
  delete p ;                          /* appel du destructeur,
                                         puis désallocation */
}
```

iv. Objets membres

Si un objet contient des objets, ceux-ci sont créés avant l'objet englobant, dans l'ordre de leur déclaration. Ils sont détruits après l'objet englobant, dans l'ordre inverse. Les constructeurs des objets membres sont donc appelés avant celui de l'objet englobant, et inversement leurs destructeurs sont appelés juste après celui de l'objet englobant.

Si les objets membres disposent d'un constructeur par défaut, c'est lui qui est appelé :

```
class Pixel
{
    int x, y ;
    Colors color ;

  public :
    Pixel(Colors c=VERT)          // constructeur par défaut
                          { x = y = 0; color = c; }
    Pixel(int xx, int yy, Colors c=ROUGE)
                          { x = xx; y = yy; color = c; }
    /* ... */
} ;
```

```
class Segment
{
    Pixel a, b ;

  public :
    Segment(int, int, int, int, Colors) ;
    /* ... */
} ;

Segment::Segment(int x1, int y1, int x2, int y2,
                                                Colors c)
{
  /*
    le constructeur par défaut a été appelé pour a et b
     a et b valent tous deux (0,0,VERT)
    ...
  */
}
```

Ici, avant d'entrer dans le constructeur de `Segment`, le constructeur par défaut de `Pixel` est appelé pour chacun des membres `a` et `b`.

S'il existe des objets membres ne possédant pas de constructeur par défaut, ou si l'on veut appeler, pour un objet membre un constructeur qui n'est pas le constructeur par défaut, une construction explicite doit être faite dans l'en-tête des constructeurs souhaités de la classe englobante :

```
class Segment
{
    Pixel a, b ;

  public :
    Segment(int, int, int, int, Colors) ;
    /* ... */
} ;

Segment::Segment(int x1, int y1, int x2, int y2, Colors c)
                : a(x1,y1,c), b(x2,y2,c)
{
  /*
     le constructeur Pixel(int, int, Colors)
     a été appelé pour a et b
     a vaut (x1,y1,c) et b vaut (x2,y2,c)
     ...
  */
}
```

Ici, avant d'entrer dans le constructeur de `Segment`, le constructeur `Pixel(int, int, Colors)` est appelé pour `a` et `b`.

Plus généralement, tout membre de classe exigeant des valeurs d'initialisation doit être initialisé sur l'en-tête de chaque constructeur de la classe englobante : c'est le cas pour les objets membres sans constructeur par défaut, mais aussi pour les constantes membres non statiques[65] et pour les membres de type référence :

```
class A ;

class B
{
  A &ra ;

  public :
    B(A &a) : ra(a) { /* ... */ }
} ;
```

Les synthétisations du constructeur par défaut, du constructeur de recopie et du destructeur pour une classe `B` ayant des objets membres :

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

class B
{
    A a ;
} ;
```

donnent une classe équivalente à :

```
class B
{
    A a ;

  public :
    B() : a() {}
    B(const B &b) : a(b.a) {}
    ~B() {}
} ;
```

avec `~B`, conformément à ce qui a été dit précédemment, appelant implicitement `a.~A`.

Si `A` possède un constructeur par défaut non trivial, le constructeur par défaut synthétisé pour `B` n'est par définition pas trivial. De même, si `A` possède un constructeur de recopie non trivial, le constructeur de recopie synthétisé pour `B` n'est pas trivial. Enfin, si `A` possède un destructeur non trivial, le destructeur synthétisé pour `B` n'est pas trivial.

---

65 Voir *Constantes membres* page 55.

v. Objets temporaires

Des objets temporaires peuvent se créer en cours d'exécution :

```
void f(Pixel) ;

int main()
{
  /* ... */
  f(Pixel(1,2)) ;          // création d'un objet temporaire
                           // l'objet temporaire est détruit ici
  /* ... */
}
```

Ici, un objet temporaire est créé (sauf optimisation particulière) pour l'argument de `f`. Dans ce cas, le constructeur est appelé lors de la définition de l'objet. Le destructeur est appelé à la fin de l'expression la plus englobante contenant la création de l'objet[66].

Le compilateur peut être amené lui-même à créer des objets temporaires. L'introduction de ces objets temporaires dépend de l'implémentation. Par exemple, l'instruction :

```
z = Add(z,z) ;
```

peut conduire le compilateur à créer des objets temporaires pour les arguments ou pour la valeur de retour, afin d'éviter des effets de bords.

11 - Affectation d'objets

En C++, la copie d'objets peut se faire soit par clonage, soit par affectation.

L'affectation d'un objet à un autre de même type est possible en utilisant l'opérateur d'affectation =[67] :

```
Complexe a, b ;
/* ... */
a = b ;
```

L'affectation implicite recopie membre à membre l'objet source dans l'objet destination.

Si ce traitement par défaut n'est pas satisfaisant, il est possible de redéfinir un opérateur d'affectation personnalisé[68], ou même d'interdire l'opération. Il existe en effet des situations où l'affectation membre à membre n'est pas acceptable. Ce point important est discuté au paragraphe *Quand et pourquoi doit-on implémenter une duplication personnalisée ?* page 272.

---

66 Pour plus de précisions, voir *Attention : objets temporaires* page 380.

67 De même qu'il est possible d'affecter une structure à une autre avec l'opérateur = en C.

68 Voir *L'opérateur* = page 87.

12 - Tableaux d'objets

La notion de tableau en C++ est la même qu'en C : c'est un concept de bas niveau permettant d'adresser un espace mémoire de la façon la plus efficace possible. Du fait du bas niveau, et en particulier de la conversion implicite d'un tableau en l'adresse de son premier élément, la manipulation des tableaux en C est intrinsèquement dangereuse.

L'optique C++ est la suivante [ARM] : *les tableaux du langage C doivent servir à construire des objets de plus haut niveau, offrant une souplesse accrue*[69] *et un niveau de sécurité plus élevé*[70].

Ceci dit, il est possible de construire un tableau (similaires à ceux que l'on rencontre dans le langage C) d'objets, à condition que les éléments possèdent un constructeur par défaut. Ces éléments sont alors initialisés avec ce constructeur :

```
class Complexe
{
  public :
    Complexe(double, double) ;       // constructeur unique
  /* ... */
} ;

Complexe tab[100] ;                                  // illégal
```

Il est impossible ici de créer un tableau de complexes, car la classe `Complexe` n'a pas de constructeur par défaut.

```
class Complexe
{
  public :
    Complexe(double=0, double=0) ;
  /* ... */
} ;

Complexe tab[100] ;                                  // d'accord
```

Ici, c'est possible et les éléments sont initialisés à `Complexe(0,0)`.

Pour chaque élément du tableau, les constructeurs sont appelés dans l'ordre des adresses croissantes. Les destructeurs sont appelés dans l'ordre inverse.

i. Cas des tableaux statiques et automatiques

Il est quand-même possible de créer un tableau statique ou automatique d'objets dépourvus de constructeur par défaut, à condition d'initialiser explicitement tous les éléments du tableau de la façon suivante :

---

69 Par exemple, en définissant la recopie de deux tableaux.

70 Par exemple en contrôlant l'indiçage.

```
class Complexe
{
  public :
    Complexe(double, double=0) ;
  /* ... */
} ;

Complexe tab[5] = { Complexe(1,2), 3, Complexe(2), 9,
                                          Complexe(2,3) } ;
```

Les éléments sont initialisés respectivement à `(1,2)`, `(3,0)`, `(2,0)`, `(9,0)`, `(2,3)`. On ferait de même avec des objets munis d'un constructeur par défaut, que l'on voudrait initialiser avec un autre constructeur.

Si les objets du tableau possèdent un constructeur par défaut, il est possible d'avoir une liste d'initialisations incomplète : dans ce cas, les derniers éléments sont initialisés avec ce constructeur.

S'il n'existe pas de constructeur par défaut, la liste des valeurs d'initialisation doit être complète.

ii. Cas des tableaux dynamiques et des tableaux membres

En ce qui concerne les tableaux dynamiques, les éléments du tableau doivent obligatoirement avoir un constructeur par défaut, car il est syntaxiquement impossible de fournir des valeurs d'initialisation.

```
Complexe *tab = new Complexe[100] ;
```

Ici, les `100` complexes sont initialisés avec le constructeur par défaut. Il ne peut en être autrement.

De même pour les tableaux membres d'une classe :

```
class Segment
{
    Pixel tab[1000] ;
  /* ... */
} ;
```

les éléments du tableau `tab` sont initialisés avec le constructeur par défaut de `Pixel`, et il ne peut en être autrement.

13 - Le point sur les unions

Les unions peuvent avoir des fonctions membres privées, publiques (et protégées[71]), y compris des constructeurs et un destructeur. Par contre, elles ne peuvent pas avoir d'objets membres possédant un constructeur non trivial, un

[71] Voir *Contrôle d'accès protégé* page 103.

destructeur non trivial, un constructeur de recopie non trivial ou un opérateur = non trivial[72]. Elles ne peuvent pas avoir non plus de membres statiques.

Union anonyme

Une union anonyme est une union de la forme :

```
union { /* ... */ } ;
```

Elle ne définit pas un type, mais un objet sans nom. Les noms des membres doivent alors être différents des noms déjà définis dans la région déclarative courante. Ces membres peuvent être utilisés directement sans qualification de portée :

```
{
  union { int i ; char *s ; } ;              //union anonyme
  i = 0 ;                              // membre i de l'union
  s = "Bouchemaine" ;                  // membre s de l'union
  /* ... */
}
```

Ici, `i` et `s` s'utilisent comme des variables ordinaires, si ce n'est qu'elles possèdent toutes deux la même adresse.

Une union anonyme ne peut pas avoir de membres privés ou protégés, ni de fonctions membres. Une union anonyme déclarée globale doit obligatoirement être spécifiée `static`. Une union utilisée pour déclarer des objets ou des pointeurs n'est pas une union anonyme :

```
{
  union { int i ; char *s ; } obj ;   // union non anonyme
  i = 0 ;                                        // illégal
  obj.i = 0 ;                                   // d'accord
  /* ... */
}
```

14 - Classes agrégats

Une classe agrégat est une classe n'ayant que des membres publics, que des constructeurs triviaux, et aucune classe de base[73].

Une classe agrégat peut être initialisée comme sont initialisées les structures du langage C :

---

72 Voir *L'opérateur* = page 87.

73 Voir *L'héritage* page 103.

```
class A                                          // classe agrégat
{
  public :
    int a, b ;
} ;

class B                                          // classe agrégat
{
  public :
    A a ;
    int b ;
} ;

B b1 = { {1,2}, 3 } ;
B b2 = { 2, 4, 6 } ;
```

Ici, `b1.a.a` vaut `1`, `b1.a.b` vaut `2` et `b1.b` vaut `3`. De même, `b2.a.a` vaut `2`, `b2.a.b` vaut `4` et `b2.b` vaut `6`.

Les membres statiques ne sont pas être initialisés dans une telle initialisation :

```
class A
{
  public :
    int a ;
    static int s ;
    int b ;
} ;

A a = { 1, 2 } ;
```

`a.a` vaut `1` et , `a.b` vaut `2`. `a.s` n'est pas initialisé.

15 - Exercices

Exercice 1

Concevoir et implémenter une classe `Rationnel` modélisant l'ensemble des nombres rationnels. L'interface de la classe contiendra un constructeur ainsi qu'une fonction `Val()` retournant la valeur (de type `float`) du rationnel.

Ajouter deux fonctions `Num()` et `Den()` permettant d'accéder au numérateur et au dénominateur.

Modifier la classe pour permettre à un utilisateur de modifier la valeur d'un rationnel.

Modifier la classe pour que tout rationnel soit représenté sous sa forme irréductible.

Exercice 2

Concevoir et implémenter une classe `InputFile` modélisant un fichier texte destiné à être lu. Ce fichier est supposé contenir une suite de lignes de caractères.

L'interface de la classe contiendra un constructeur auquel devra être fourni le nom du fichier à ouvrir, un destructeur, deux fonctions `GetChar` et `GetLine` renvoyant respectivement le caractère courant du fichier et la fin de la ligne courante (et avançant d'autant la position courante) et une fonction booléenne `Eof` indiquant si la fin de fichier est atteinte.

Exercice 3

Concevoir et implémenter une classe `OutputFile` modélisant un fichier texte destiné à l'écriture.

L'interface de la classe contiendra un constructeur auquel devra être fourni le nom du fichier à créer, un destructeur ainsi que deux fonctions `Put`, l'une envoyant un caractère dans le fichier et l'autre envoyant une chaîne de caractère.

Exercice 4

Se pencher sur le problème du clonage des classes `InputFile` et `OutputFile` réalisées aux exercices 2 et 3.

Exercice 5

Concevoir et implémenter une classe `String` modélisant les chaînes de caractères. Les objets de type `String` n'auront pas de taille maximale prédéfinie, l'allocation de l'espace mémoire se faisant dynamiquement et de façon transparente pour l'utilisateur.

L'interface de `String` contiendra au moins un constructeur, un destructeur si nécessaire, une fonction `Elem` permettant d'obtenir et de modifier un caractère de la chaîne, une fonction `Length` donnant le nombre de caractères de la chaîne, une fonction `Sub` renvoyant une sous-chaîne prise dans la `String` courante et une fonction `Val` renvoyant la valeur de la chaîne (de type `const char *`).

Vérifier le bon fonctionnement du clonage d'un objet `String`.

Peut-on utiliser l'affectation entre `String` ?

Peut-on utiliser des objets `String` constants ?

Exercice 6

Concevoir et implémenter une classe `Liste` modélisant une liste chaînée unidirectionnelle de réels. L'interface de cette classe contiendra les fonctions suivantes :

`float Courant()`, qui renvoie l'élément courant de la liste,

`void Debut()`, qui positionne l'élément courant sur le premier élément de la liste,

`void Suivant()`, qui positionne l'élément courant sur l'élément suivant,

`void Atteindre(int)`, qui positionne le pointeur courant sur le $n^{ième}$ l'élément de la liste,

`bool Valide()`, qui indique si l'élément courant est valide,

`void Chercher(float)`, qui positionne l'élément courant sur le premier élément égal à une valeur donnée,

`int Rang()`, qui fournit la position de l'élément courant,

`int Nombre()`, qui fournit le nombre d'élément de la liste,

`void InsererAvant(float)`, qui insère une valeur avant l'élément courant,

`void InsererAprès(float)`, qui insère une valeur après l'élément courant,

`void Supprimer()`, qui efface l'élément courant.

Exercice 7

La classe `Liste` de l'exercice 6 ne permet pas d'accéder à plusieurs éléments simultanément. Il serait par exemple laborieux de trier une telle liste...

Imaginer une solution à ce problème.

III - LES OPERATEURS

1 - Généralités

La sémantique des opérateurs du langage peut être étendue aux objets.

Un opérateur qui ne peut pas être par défaut appliqué aux objets, peut se voir doter d'une sémantique spécifique pour un type d'objet particulier. Il est possible par exemple de définir un sens à l'expression `z1==z2` dans laquelle `z1` et `z2` sont de type `Complexe`.

De même, un opérateur pouvant implicitement s'appliquer aux objets, peut se voir doter d'une sémantique différente pour un type d'objet particulier. Par exemple, il est possible de redéfinir le sens de l'expression `s1=s2` dans laquelle `s1` et `s2` sont de type `String`.

Tous les opérateurs C++ peuvent ainsi être étendus, exceptés les opérateurs `.` `.*` `::` `?:` et `sizeof`, ainsi que les deux symboles `#` et `##` gérés par le préprocesseur[74]. Il n'est pas possible de créer des opérateurs à partir de nouveaux symboles.

Surcharger un opérateur revient à créer une fonction. Par conséquent, toutes les règles (surcharge, résolution d'appel, etc.) relatives aux fonctions s'appliquent également aux opérateurs.

Les opérateurs ne peuvent pas avoir d'argument par défaut, excepté l'opérateur `()`.

Il est impossible de modifier ou de créer de nouvelles opérations sur les types prédéfinis : un opérateur surchargé doit avoir au moins un opérande qui n'est pas de type prédéfini.

La surcharge d'un opérateur conserve :

- sa pluralité : les opérateurs unaires restent unaires, les binaires restent binaires, et `-`, `+`, `*` et `&` peuvent être surchargés en tant qu'opérateurs unaires, binaires ou les deux ;
- sa priorité vis à vis des autres opérateurs : l'opérateur surchargé conserve la priorité qui est celle de son utilisation prédéfinie ;

---

74 Voir en annexe *Opérateurs surchargeables* page 385.

— son associativité : l'opérateur surchargé conserve l'associativité qui est celle de son utilisation prédéfinie.

Par contre, la surcharge d'un opérateur ne conserve pas :

— l'éventuelle commutativité : par exemple, définir la somme d'un complexe et d'un réel ne permet pas d'additionner un réel et un complexe ;

— les liens sémantiques avec les autres opérateurs : par exemple, la surcharge de l'opérateur + est indépendante de celle de ++ et de celle de +=.

2 - Surcharge d'un opérateur

Pour surcharger un opérateur @, il suffit de définir une fonction `operator @`.

i. Définition par une fonction membre

Si la fonction `operator @` est une fonction membre, elle comporte un nombre de paramètres égal à la pluralité de l'opérateur -1. Le premier opérande est l'objet propriétaire de la fonction[75]. Le deuxième opérande, s'il existe, est le paramètre de la fonction.

Une construction du type `a @ b` est interprétée comme un appel `a.operator @(b)`[76].

Excepté pour les opérateurs `new`, `new[]`, `delete` et `delete[]`, les fonctions membres `operator @` ne peuvent pas être statiques.

L'exemple suivant définit l'addition et le cumul de deux complexes par deux fonctions membres :

```
class Complexe
{
    double x, y ;

  public :
    Complexe (double=0, double=0) ;
    Complexe operator+(Complexe) ;
    void operator+=(Complexe) ;
  /* ... */
} ;
```

---

75 C'est-à-dire l'objet pointé par `this`.

76 Sauf pour `operator->` (voir *L'opérateur ->* page 93).

```
Complexe Complexe::operator+(Complexe z)
{
  return Complexe(x+z.x, y+z.y) ;
}

void Complexe::operator+=(Complexe z)
{
  x += z.x ;
  y += z.y ;
}
```

Cela peut s'utiliser comme :

```
Complexe z1, z2, z ;
/* ... */
z = z1+z2 ;
z1 += z2 ;
```

ce qui est interprété respectivement comme :

```
z = z1.operator+(z2) ;
```

et :

```
z1.operator+=(z2) ;
```

De même :

```
Complexe z1, z2, z3, z ;
/* ... */
z = z1+z2+z3 ;
```

est interprété, conformément à l'associativité de +, comme :

```
z = (z1.operator+(z2)).operator+(z3) ;
```

Le premier opérande d'un opérateur défini de cette façon doit obligatoirement être un objet. Ainsi, même si l'utilisateur a défini une conversion implicite d'un `double` en `Complexe`[77], il est impossible de calculer :

```
z1 = x+z2 ;
```

si `x` est de type `double`, puisque cela est interprété comme `z1=x.operator+(z2)`, ce qui n'a pas de sens.

ii. Définition par une fonction non membre

Si la fonction `operator @` n'est pas une fonction membre, elle comporte un nombre de paramètres égal à la pluralité de l'opérateur. Les opérandes sont simplement les paramètres de la fonction. Afin d'accéder à l'implémentation de l'objet, cette fonction est souvent amie de la classe.

---

[77] Voir *Conversions définies par l'utilisateur* page 294.

L'utilisation d'une fonction non membre permet de conserver une symétrie dans l'expression d'un opérateur binaire, ce qui s'avère naturel lorsque l'opérateur est commutatif.

Une construction du type `a @b` est interprétée comme un appel `operator @(a,b)`.

Cette formulation amène deux avantages par rapport à la précédente :

— le premier opérande n'est pas forcément un objet,

— il peut ne pas y avoir de correspondance exacte, entre le type du premier opérande et celui du premier paramètre de la fonction `operator`. Dans ce cas, le mécanisme habituel des conversions est mis en œuvre pour tenter d'obtenir une correspondance dégradée, et pour déterminer la meilleure fonction à appeler, si plusieurs d'entre elles sont candidates.

L'exemple suivant définit l'addition de deux complexes par une fonction globale :

```
class Complexe
{
    double x, y ;
    friend Complexe operator+(Complexe, Complexe) ;

  public :
    Complexe (double=0, double=0) ;
  /* ... */
} ;

Complexe operator+(Complexe a, Complexe b)
{
  return Complexe(a.x+b.x, a.y+b.y) ;
}
```

L'utilisation de l'opérateur est la même :

```
Complexe z1, z2, z ;
/* ... */
z = z1+z2 ;
```

mais cela est interprété ici comme :

```
z = operator+(z1,z2) ;
```

L'utilisation d'une fonction non membre permet de surcroît de définir un opérateur binaire dont le premier opérande est un type prédéfini, ce que ne permet pas une fonction membre (dans ce cas, il faut donc que le second opérande ne soit pas d'un type prédéfini) :

```
Complexe operator*(double x, Complexe z)
{
  return Complexe(x*z.x, x*z.y) ;
}
```

Cet opérateur définit la multiplication d'un réel et d'un complexe, ce qui s'utilise comme :

```
Complexe z ;
/* ... */
z = 2*z ;
```

3 - Les opérateurs prédéfinis pour les objets

i. L'opérateur =

L'opérateur = possède une sémantique prédéfinie pour tout couple d'objets de même type. On appelle *opérateur d'affectation de recopie* d'une classe `C` un opérateur = ayant un paramètre de type `C`, `C&` ou `const C&`.

Si une classe `C` ne définit pas explicitement l'affectation de recopie, alors tout ce passe comme si un opérateur = de recopie était synthétisé par le compilateur[78]. Celui-ci recopie membre à membre l'objet source (le paramètre) dans l'objet destinataire (l'objet courant).

L'opérateur d'affectation de recopie étant la quatrième et dernière fonction pouvant être synthétisée par le compilateur, on peut faire ici le bilan de toutes les synthétisations réalisables par le compilateur. Pour la classe `B` suivante :

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

class B
{
    A a ;
} ;
```

le compilateur synthétise une classe finalement équivalente à :

```
class B
{
    A a ;

  public :
    B() : a() {}
    B(const B &b) : a(b.a) {}
    B &operator=(const B &b) { a = b.a ; return *this ; }
    ~B() {}
} ;
```

---

78 Si la classe ne possède pas d'objet membre, de fonction virtuelle et de classe de base, l'opérateur d'affectation de recopie synthétisé est appelé *opérateur d'affectation trivial.* Si la classe possède un objet membre ayant un opérateur d'affectation non trivial, l'affectation synthétisée n'est pas triviale.

avec `~B` appelant implicitement `a.~A`.

Cependant, dans certaines situations, l'affectation synthétisée n'est pas acceptable et le programmeur doit surcharger l'opérateur `=`[79]. Dans ce cas, la fonction `operator=` doit être une fonction membre :

```
class Complexe
{
   double x, y ;

  public :
    Complexe operator=(const Complexe &z)
                        { x = z.x; y = z.y; return *this; }
   /* ... */
} ;
```

La fonction `operator=` peut être également utilisée pour définir l'affectation entre des objets de type différent. En effet, toute latitude est laissée quant au type du second opérande. On peut ainsi définir l'affectation d'un réel à un complexe :

```
class Complexe
{
   double x, y ;

  public :
    Complexe operator=(double) ;
} ;
```

Cela permet d'écrire :

```
Complexe z ;
/* ... */
z = 1.1 ;              // interprété comme z.operator=(1.1)
```

Dans ce cas, la définition de `Complexe::operator=(double)` ne supprime pas la synthétisation d'un opérateur d'affectation de recopie :

```
Complexe z1, z2 ;
/* ... */
z1 = z2 = 1.1 ;        // interprété z1=(z2.operator=(1.1))
```

le premier `=` étant l'affectation synthétisée et le second étant réalisé par `operator=(double)`.

ii. L'opérateur & unaire et l'opérateur ,

Les opérateurs `&` unaire et `,` ont eux aussi une signification prédéfinie pour tout type d'objet. Leur sémantique est la même que pour les types prédéfinis, à

---

[79] Voir *Quand et pourquoi doit-on implémenter une duplication personnalisée ?* page 272.

savoir & retourne l'adresse de son opérande et , évalue séquentiellement ses deux opérandes.

Ces opérateurs peuvent être surchargés.

4 - Cas particuliers d'opérateurs

i. Les opérateurs `new` et `delete`

Les opérateurs `new` et `delete` peuvent être surchargés au niveau d'une classe, indépendamment de leur surcharge au niveau global[80]. Dans ce cas, ils doivent être obligatoirement définis par des fonctions membres. `new` et `delete` sont implémentés sous forme de fonctions statiques et ne disposent donc ni des membres non statiques de l'objet, ni du pointeur `this`.

Si `new` et `delete` sont surchargés pour une classe, ils sont appelés à chaque création dynamique ou destruction dynamique d'objets de ce type. S'ils ne sont pas définis, ce sont les opérateurs `new` et `delete` globaux qui sont utilisés.

Allocation et désallocation d'objets uniques

La fonction membre `operator new` est appelée lors de l'évaluation d'une expression `new` avant le constructeur de l'objet. Elle est définie avec un paramètre de type `size_t` et un type de retour `void *`. Le paramètre est automatiquement initialisé avec la taille de la zone à allouer[81]. La valeur de retour doit être l'adresse de la zone allouée.

La fonction membre `operator delete` est appelée lors de l'évaluation d'une expression `delete` après le destructeur de l'objet. Elle est définie avec un paramètre de type `void *` et un type de retour `void`. Le paramètre reçoit automatiquement l'adresse de la zone à libérer.

Ce qui donne :

```
class C
{
  public :
    C(int) ;
    void *operator new(size_t size) ;
    void operator delete(void *ptr) ;
  /* ... */
} ;
```

ces opérateurs étant utilisés de façon transparente pour l'utilisateur :

---

80 Voir *Surcharge des opérateurs de gestion de mémoire* page 40.

81 Ce n'est pas forcément la taille de la classe courante : ce peut être la taille d'une classe dérivée dans le cas où une classe n'ayant pas défini d'`operator new` possède une classe de base l'ayant défini.

```
C *ptr = new C(1) ;                    // appel de operator new,
                                       // puis du constructeur
delete ptr ;                           // appel du destructeur,
                                       // puis de operator delete
```

La fonction `operator new` reçoit `sizeof(C)` en paramètre, et le constructeur reçoit `1`. La fonction `operator delete` reçoit la valeur de `ptr` en paramètre.

Des paramètres supplémentaires, de type quelconque, peuvent être fournis à la fonction `operator new` :

```
void *C::operator new(size_t,bool) ;
```

Ici, un paramètre booléen à été ajouté. L'utilisation de cet opérateur de placement devient :

```
C *ptr = new(true) C(1) ;
```

La fonction `operator new` reçoit `sizeof(C)` et `true` en paramètre, et le constructeur reçoit `1`.

Un paramètre supplémentaire de type `size_t` peut être affecté à la fonction `operator delete`[82] :

```
void C::operator delete(void *,size_t) ;
```

Dans ce cas, l'utilisation de `delete` est inchangée, mais la fonction `operator delete` reçoit en plus la taille de la zone à libérer.

Dans la définition de `new` et `delete`, il est possible de faire référence aux `new` et `delete` globaux, dénommés `::new` et `::delete`.

Allocation et désallocation de tableaux d'objets

C'est la fonction :

```
void *C::operator new[](size_t) ;
```

qui est appelée lors d'une expression de la forme :

```
C *ptr = new C[10] ;
```

Le paramètre est automatiquement initialisé avec la taille de la zone à allouer. La valeur de retour doit être l'adresse de la zone allouée. Le constructeur, appelé pour chaque objet du tableau, est toujours le constructeur par défaut de la classe. S'il n'existe pas, l'expression est illégale.

C'est la fonction :

```
void C::operator delete[](void *ptr) ;
```

qui est appelée lors d'une expression de la forme :

```
delete[] ptr ;
```

---

82 Ce qui n'est pas possible pour l'opérateur `delete` global.

Comme pour `operator new`, des arguments supplémentaires peuvent être transmis à la fonction `operator new[]`. Exemple :

```
class A
{
  public :
    void *operator new(size_t,void *) ;
} ;

void f(void *adr)
{
  A *pb1 = new(adr) A ;    // A::operator new(size_t,void*)
  A *pb2 = new A[10] ;          // ::operator new[](size_t)
  A *pb3 = new A ;                                  // erreur
}
```

La dernière instruction est invalide, car `::operator new(size_t)` est masqué.

De même, comme pour `operator delete`, un argument supplémentaire de type `size_t`, dont la valeur est la taille de la zone à libérer, peut être transmis implicitement à la fonction `operator delete[]`.

ii. L'opérateur [ ]

Si `o` est un objet, `o[i]` est interprété comme `o.operator[](i)`.

L'opérateur `[]` doit être défini comme une fonction membre.

```
class String
{
    char *str ;

  public :
    String(const char *) ;
    char &operator[](int i)
                { assert(i<strlen(str)) ; return str[i] ;}
  /* ... */
} ;

String s("Quarts de Chaume") ;
/* ... */
char c = s[3] ;      // équivalent à char c=s.operator[](3)
s[1] = 'i' ;            // équivalent à s.operator[](1)='i'
```

Le type de l'opérande, donc de l'indice, n'est pas forcément arithmétique : tout type (même `struct`, `class`, etc.) peut être utilisé.

L'opérateur ne peut être que binaire : il n'est pas possible de définir un opérateur `[]` à plusieurs indices[83] ;

Par contre, on peut définir un multi-indiçage, en définissant un opérateur `[]` qui renvoie un objet pouvant être lui-même indicé :

```
class mat
{
    class tab
    {
        int v[3] ;

      public :
        int &operator[](int i) { return v[i] ; }
    } t[3] ;

  public :
    tab &operator[](int i) { return t[i] ; }
} ;
```

La classe `mat` implémente une matrice 3×3 dont les éléments sont référencés par deux indices :

```
{
  mat m ;
  /* ... */
  int k = m[1][2] ;
}
```

L'interprétation de cette dernière instruction peut être décomposée successivement en :

```
int k = m.operator[](1).operator[](2) ;
int k = m.t[1].operator[](2) ;
int k = m.t[1].v[2] ;
```

iii. Les opérateurs ( )

L'opérateur `()` est l'appel de fonction. Il est *n*-aire.

Chacune des pluralités est implémentée, indépendamment des autres, par une fonction `operator()` spécifique. Ainsi :

- `operator()()` surcharge l'appel de fonction sans paramètre,
- `operator()(int)` surcharge l'appel de fonction avec un paramètre entier,
- `operator()(int,int)` surcharge l'appel de fonction avec deux paramètres entier,

---

[83] Quoique, en surchargeant à la fois les opérateurs `[]` et `,` , et avec un peu d'astuce, on finit par y arriver...

— `operator()(int,char)` surcharge l'appel de fonction avec deux paramètres entier et caractère,

et ainsi de suite...

Donc, si `o` est un objet,

— `o()` est interprété comme `o.operator()()`,

— `o(x)` est interprété comme `o.operator()(x)`,

— `o(x,y)` est interprété comme `o.operator()(x,y)`,

et ainsi de suite...

L'opérateur `()` doit être défini comme une fonction membre.

L'exemple suivant illustre la définition d'un opérateur `()` ternaire :

```
class String
{
  public :
    /* ... */
    String Sub(int, int) ;      // renvoi d'une sous-chaîne
    String operator()(int deb, int nb)
                                    { return Sub(deb,nb);}
  /* ... */
} ;
```

ce qui s'utilise comme :

```
String s1, s2 ;
/* ... */
s2 = s1(1,4) ;
```

`s1(1,4)` est interprété comme `s1.operator()(1,4)`.

Les opérateurs `()` permettent de définir des *objets-fonctions*[84].

iv. L'opérateur ->

L'opérateur `->` peut être surchargé, et il doit l'être par une fonction membre.

Si `o` est un objet (et non pas un pointeur !), `o->mbr` est interprété comme `(o.operator->())->mbr`.

Pour que cette expression ait un sens, l'opérateur `->` doit donc renvoyer :

a/ l'adresse d'une classe ou d'une structure, ou

b/ un objet (ou une référence vers un objet) ayant lui-même surchargé l'opérateur `->`.

Etudions d'abord le cas a/. Les membres publiques d'un objet `a` peuvent être référencés par l'intermédiaire d'un objet `b` si ce dernier possède un opérateur `->` renvoyant l'adresse de `a` :

---

84 Voir *Objets-fonctions page* 151.

```
class A
{
  public :
    int m ;
} ;

class B
{
    A a ;

  public :
    A *operator->() { return &a ; }
} ;
```

Si b est de type B, alors l'interprétation de l'instruction :

```
b->m = 1 ;
```

peut être décomposée en :

```
(b.operator->())->m = 1 ;
(&b.a)->m = 1 ;
b.a.m = 1 ;
```

Notons que l'opérateur -> appliqué à un pointeur conserve toujours sa signification prédéfinie[85], même si l'objet possède un opérateur -> :

```
B *b = new B ;
b->m ;                  // erreur : m n'est pas un membre de b
```

Voyons maintenant le cas b/. L'opérateur -> d'un objet b peut être appelé par l'intermédiaire d'un objet c si ce dernier possède un opérateur -> renvoyant b :

```
class A
{
  public :
    int m ;
} ;

class B
{
    A a ;

  public :
    A *operator->() { return &a ; }
} ;
```

[85] Puisque la définition d'opérateurs ne s'applique qu'aux objets et non pas aux types prédéfinis tel que les types pointeurs.

```
class C
{
    B b ;

  public :
    B operator->() { return b ; }
} ;
```

Si `c` est de type `C`, alors l'interprétation de l'instruction :

```
c->m = 1 ;
```

peut être décomposée en :

```
(c.operator->())->m = 1 ;
(c.b)->m = 1 ;
(c.b.operator->())->m = 1 ;
(&c.b.a)->m = 1 ;
c.b.a.m = 1 ;
```

L'opérateur `->` est utilisé pour déléguer la réalisation de services à un objet qui n'est pas le receveur de la demande.

Dans l'exemple suivant, `Tab` implémente un tableau possédant la notion d'élément courant. L'opérateur `->` permet, à partir d'un tableau, d'accéder aux membres de son élément courant :

```
class Tab
{
  public :

    class Elem
    {
      public :
        void f() ;
        int g(char) ;
      /* ... */
    } ;
    Elem *operator->() { return &t[courant] ; }

  private :
    Elem t[100] ;
    int courant ;
  /* ... */
} ;
```

L'utilisateur de la classe `Tab` peut accéder aux membres publics de `t[courant]`. En effet, si `tab` est de type `Tab`, alors :

```
tab->f() ;
```

est équivalent à :

```
tab.t[tab.courant].f() ;
```

ce qui exécute la fonction `f` de l'élément courant du tableau. De même :

```
int i = tab->g('A') ;
```

est équivalent à :

```
int i = tab.t[tab.courant].g('A') ;
```

D'autres exemples de surcharge de l'opérateur `->` sont fournis au paragraphe *Les modèles de classes* page 143.

v. Les opérateurs ++ et --

Les opérateurs `++` et `--` préfixés et postfixés peuvent être surchargés indépendamment les uns des autres.

Dans le cas de fonctions membres, `operator++()` et `operator--()` définissent les opérateurs `++` et `--` préfixés, alors que les fonctions `operator++(int)` et `operator--(int)` définissent les opérateurs `++` et `--` postfixés. L'argument entier supplémentaire ne sert à rien, sinon à différencier les deux familles de fonctions :

```
class Complexe
{
    float x, y ;

  public :
    Complexe operator++() { ++x ; return *this ; }
    Complexe operator++(int)
                  { Complexe c=*this ; x++ ; return c ; }
    Complexe operator--() { --x ; return *this ; }
    Complexe operator--(int)
                  { Complexe c=*this ; x-- ; return c ; }
    /* ... */
} ;
```

Dans le cas de fonctions non membres, un paramètre supplémentaire de type `int` permet de la même façon de discriminer les opérateurs préfixés des opérateurs postfixés :

```
Complexe operator++(Complexe &z) { ++z.x ; return z ; }
Complexe operator++(Complexe &z, int)
                    { Complexe c=z ; z.x++ ; return c ;}
Complexe operator--(Complexe &z) { --z.x ; return z ; }
Complexe operator--(Complexe &z, int)
                    { Complexe c=z ; z.x-- ; return c ;}
```

5 - Opérateurs et énumérations

Chaque énumération est, en C++, un type particulier, distinct de `int`. Il est par conséquent possible de munir les énumérations d'opérateurs :

```
enum jour { lun, mar, mer, jeu, ven, sam, dim } ;

jour operator++(jour &j)
{
  return j = jour( (j+1) % (dim+1) ) ;
}

void InitMois(jour mois[], jour premier, int nbjour)
{
  for (int i=0 ; i<nbjour ; i++)
  {
    mois[i] = premier ;
    ++premier ;
  }
}

void main()
{
  jour janvier[31] ;
  InitMois(janvier,dim,31) ;
  /* ... */
}
```

6 - Exercices

Exercice 8

Définir pour la classe `Rationnel` (exercice 1 page 80) les opérateurs `+`, `-`, `*`, `/` désignant les 4 opérations arithmétiques standard entre deux objets de cette classe, les opérateurs associés `+=`, `-=`, `*=`, `/=` ainsi que l'opérateur unaire `-`.

Définir les opérateurs relationnels `==`, `!=`, `<`, `<=`, `>`, `>=` .

Est-il nécessaire de redéfinir l'opérateur d'affectation de recopie ?

Exercice 9

Définir pour la classe `String` (exercice 5 page 81) les opérateurs `+`, `+=` et `[]` désignant les opérations de concaténation de chaîne, d'ajout en fin de chaîne et d'indiçage.

Peut-on réaliser des concaténations en série ?

Définir les opérateurs relationnels `==`, `!=`, `<`, `<=`, `>`, `>=` .
Est-il nécessaire de redéfinir l'opérateur d'affectation de recopie ?

Exercice 10

Concevoir et implémenter une classe `Monnaie` modélisant l'ensemble des nombres décimaux à deux décimales. Cette classe doit supporter les opérations arithmétiques de base sans propagation d'erreur. On réalisera l'addition et la soustraction d'un objet `Monnaie` à un autre de type `Monnaie`, ainsi que la multiplication et la division d'un objet `Monnaie` par un entier relatif.

Exercice 11

Concevoir et implémenter une classe `Durée` modélisant les durées, sous la forme *jour/heure:minute:seconde*.

L'interface de cette classe contiendra un ou plusieurs constructeurs, des fonctions de consultation `ObtenirJour`, `ObtenirHeure`, `ObtenirMinute` et `ObtenirSeconde`, des fonctions de modification `InitJour`, `InitHeure`, `InitMinute` et `InitSeconde`, et deux opérateurs d'addition et de soustraction de durées.

Exercice 12

Concevoir et implémenter une classe `Date` modélisant les dates sous la forme *jour/mois/année heure:minute:seconde*.

L'interface de cette classe contiendra un ou plusieurs constructeurs, des fonctions de consultation `ObtenirJour`, `ObtenirMois`, `ObtenirAn`, `ObtenirHeure`, `ObtenirMinute` et `ObtenirSeconde`, des fonctions de modification `InitJour`, `InitMois`, `InitAn`, `InitHeure`, `InitMinute` et `InitSeconde`, un opérateur d'addition d'une date et d'une durée (voir exercice 11), ce qui donne une date, et enfin un opérateur de soustraction de deux dates, ce qui donne une durée.

IV - CONVERSIONS DE L'UTILISATEUR

Des conversions de type peuvent être définies par le programmeur. Ces définitions se font en utilisant :

— les *constructeurs de conversion*, ou
— les opérateurs de conversion de type.

Une fois spécifiées, ces conversions sont mises en œuvre lors d'opérations de conversion explicite (*cast*). Elles sont également sollicitées automatiquement, lorsque le contexte l'exige (passage d'argument dont le type ne correspond pas exactement à celui du paramètre, retour de fonction d'un type différent de celui attendu, etc.), en plus des conversions standard[86].

1 - Constructeurs de conversion

Un constructeur de conversion est un constructeur qui peut être appelé avec un seul argument. C'est donc un constructeur disposant d'un paramètre unique, ou d'un seul paramètre sans valeur par défaut.

La spécification d'un constructeur de conversion `A(B)` définit une conversion de `B` vers `A`. Le constructeur de conversion de la classe `Complexe` suivante définit une conversion de `double` en `Complexe` :

```
class Complexe
{
    friend Complexe operator+ ( Complexe, Complexe ) ;

  public :
    Complexe(double a, double b=0) ;
  /* ... */
} ;

Complexe z1(2) ;                                    // oui
```

[86] Cependant, l'utilisation implicite des conversions de l'utilisateur est soumise à certaines règles (une seule conversion de l'utilisateur peut être appelée implicitement pour une valeur donnée, toute conversion ambigüe est invalide, etc.). Voir *Séquences de conversions implicites* page 295.

```
Complexe z2 = 4 ;                                    // oui
Complexe z3 = z1+Complexe(2.0) ;                     // oui
Complexe z4 = z1+(Complexe) 2.0 ;                    // oui
Complexe z5 = z1+2.0 ;                               // oui
```

Dans cet exemple, l'addition d'un complexe et d'un réel n'est pas définie. Pourtant la dernière instruction est correcte. En effet sont définies d'une part l'addition de deux complexes, et la conversion d'un réel en complexe d'autre part. L'instruction `z1+2.0` amène le compilateur à convertir, via le constructeur de conversion `Complexe(double, double=0)`, `2.0` en `Complexe`, puis à utiliser l'opérateur `operator+` pour additionner les deux complexes. Cette dernière instruction est donc interprétée comme `operator+(z1,Complexe(2.0))`.

Si l'on veut un constructeur à un paramètre, sans avoir la conversion correspondante, on utilise le mot clé `explicit` :

```
class Complexe
{
    friend Complexe operator+ ( Complexe, Complexe ) ;

  public :
    explicit Complexe(double a, double b=0) ;
  /* ... */
} ;
```

Le constructeur ci-dessus n'est plus un constructeur de conversion. Il doit maintenant être explicitement appelé :

```
Complexe z1(2) ;                                     // oui
Complexe z2 = 4 ;     // non, pas de conversion implicite
Complexe z3 = z1+Complexe(2.0) ;                     // oui
Complexe z4 = z1+(Complexe) 2.0 ;                    // oui
Complexe z4 = z1+2.0 ;                               // non
```

Les constructeurs de conversion ne permettent pas :

— de définir une conversion vers un type prédéfini (puisque le type *destination* doit posséder un constructeur),

— de définir une nouvelle conversion sans changer la définition du type *destination* (puisqu'il est nécessaire d'y rajouter un nouveau constructeur).

Pour définir une conversion vers un type non classe, ou pour définir une conversion sans changer la définition du type *destination*, on utilise les opérateurs de conversion.

2 - Opérateurs de conversion

La fonction membre `X::operator T()`, où *T* est un type[87], définit une conversion de `X` vers *T*.

```
class BigInt
{
    char v[NB_CHIFFRES] ;

  public :
    BigInt(int i) ;
    operator int () ;              // opérateur de conversion
  /* ... */
} ;
```

La classe `BigInt` modélise un ensemble de grands entiers. Chaque chiffre du nombre est mémorisé dans un tableau de caractères. Un `int` peut être converti en `BigInt` par l'intermédiaire du constructeur de conversion `BigInt(int)`, et réciproquement, un `BigInt` peut (tenter d') être converti en `int` par l'intermédiaire de l'opérateur `int()` :

```
BigInt i ;
i = 2 ;                            // conversion int->BigInt

int j ;
j = i ;                            // conversion BigInt->int
```

L'instruction `i=2` est interprétée comme `i=BigInt(2)`, et l'instruction `x=i` est interprétée comme `x=i.operator int()`.

Le mot clé `explicit` peut être utilisé pour indiquer que l'opérateur ne doit pas être utilisé pour effectuer des conversions implicites :

```
class C
{
  public :
    explicit operator int() ;
} ;

C c ;
int i = c ;                                    // illégal
int j = int(c) ;                                  // oui
```

87 N'importe quel type, prédéfini ou non prédéfini, sauf le type `void`.

3 - Exercices

Exercice 13

Etendre la classe `Rationnel` (exercice 8 page 97) pour permettre qu'un objet de cette classe puisse être utilisé dans une expression arithmétique quelconque faisant intervenir les types standard.

Que se passe-t-il si l'on ajoute la possibilité de construire un `Rationnel` à partir d'un entier ?

Exercice 14

Etendre la classe `String` (exercice 9 page 97) pour permettre qu'un objet de cette classe puisse être utilisé comme un objet de type `const char *` par les fonctions de la bibliothèque standard C.

V - L'HERITAGE

Le mécanisme d'*héritage* permet de définir une nouvelle classe à partir de classes existantes, dites *classes de base directes*. Cette nouvelle classe, dite *classe dérivée*, hérite des membres des classes qui ont servi à la construire, tout en ayant la possibilité de rajouter ou de redéfinir ses propres membres.

Une classe de base directe peut elle-même être une classe obtenue par héritage. On appelle *classe de base* d'une classe C soit une classe de base directe de C, soit une classe de base directe d'une classe de base de C.

Une *classe de base accessible* d'une classe C est une classe de base de C dont les membres publics sont accessibles par les utilisateurs de C[88].

L'héritage est *simple* si la classe dérivée ne possède qu'une seule classe de base directe, il est *multiple* sinon.

Il existe trois types d'héritage, qui se distinguent par les droits d'accès attribués aux membres hérités : l'héritage *privé*, l'héritage *protégé* et l'héritage *public*.

Il n'y a pas d'héritage possible avec les unions : une union ne peut pas avoir de classes de base, et réciproquement, une union ne peut pas servir de classe de base.

1 - Contrôle d'accès protégé

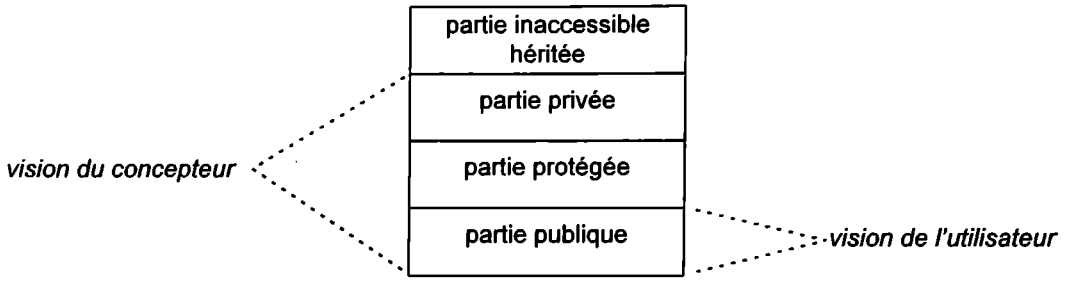
En plus des parties privées et publiques, une classe peut avoir une partie *protégée*. Le mot clé `protected` est utilisé à cet effet.

Les membres protégés sont accessibles au concepteur de la classe, comme le sont les membres privés et publics.

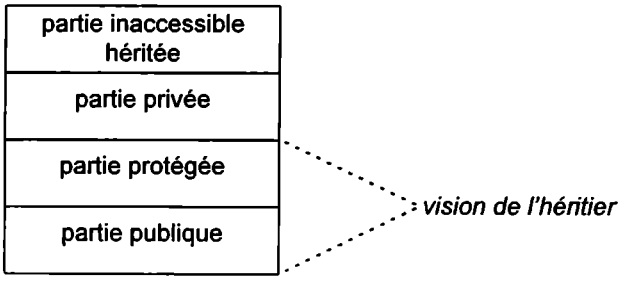
Pour un utilisateur de la classe, la partie protégée est équivalente à la partie privée : un membre protégé lui est inaccessible.

---

88 Dans le graphe d'héritage, le chemin reliant une classe à sa classe de base accessible est alors composé exclusivement d'arcs d'héritage public.



UN *héritier* d'une classe `A` est un concepteur d'une classe dérivée de `A`. Un héritier de `A` possède plus de droits d'accès qu'un utilisateur de `A`. En effet, les membres protégés de `A` lui sont accessibles. Il dispose donc des membres protégés et publics de `A`, mais n'a aucun accès aux membres privés.



Par exemple, si `B` est une classe dérivée de `A` :

```
class A
{
    int priv ;

  protected :
    int prot ;

  public :
    int pub ;
} ;

/* ... */

void B::f()
{
  priv = 1 ;                    // illégal
  prot = 1 ;                    // Ok
  pub = 1 ;                     // Ok
}
```

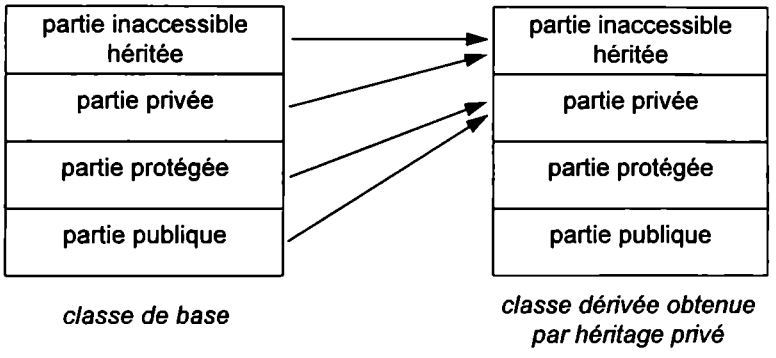
```
void main()
{
  A a ;

  a.priv = 1 ;                                    // illégal
  a.prot = 1 ;                                    // illégal
  a.pub = 1 ;                                        // ok
}
```

Les fonctions membres de B ont accès aux membres protégés et publics de A, mais pas aux membres privés. Les utilisateurs de A n'ont accès qu'aux membres publiques de A.

2 - Trois types d'héritage

i. L'héritage privé

C'est l'héritage qui restreint le plus l'accès aux membres hérités. Par une dérivation privée, les membres publics et protégés des classes de base directes deviennent des membres privés de la classe dérivée.



Aucun membre d'une classe A n'est donc accessible aux utilisateurs et aux héritiers d'une classe B obtenue par héritage privé à partir de A. Illustration, si C est une classe dérivée de B :

```
class A
{
    int priv ;

  protected :
    int prot ;

  public :
    int pub ;
} ;
```

```
class B : private A      // => A classe de base privée de B
{
} ;

/* ... */

void C::f()
{
  priv = 1 ;                                          // illégal
  prot = 1 ;                                          // illégal
  pub = 1 ;                                           // illégal
}

void main()
{
  B b ;

  b.priv = 1 ;                                        // illégal
  b.prot = 1 ;                                        // illégal
  b.pub = 1 ;                                         // illégal
}
```

La déclaration `using` permet d'introduire un identificateur d'une classe de base dans la classe courante :

```
class A
{
  public :
    int pub ;
  /* ... */
} ;

class B : private A
{
  public :
    using A::pub ;          // A::pub passe en public dans B
  /* ... */
} ;

void main()
{
  B b ;
  b.pub = 1 ;                                         // d'accord
}
```

En introduisant ici `A::pub` dans la partie publique de `B`, la déclaration `using` donne aux utilisateurs de `B` un accès particulier à ce membre.

L'héritage privé est l'héritage par défaut pour les classes, lorsque ni `public`, ni `protected`, ni `private` ne sont mentionnés.

Ce type d'héritage s'utilise lorsque, bien que la définition de B s'appuie sur celle de A, B n'est pas un cas particulier de A, c'est-à-dire lorsque B ne peut pas être utilisé comme *une sorte de* A. C'est le cas notamment lorsque B restreint les fonctionnalités de A.

A partir de la classe :

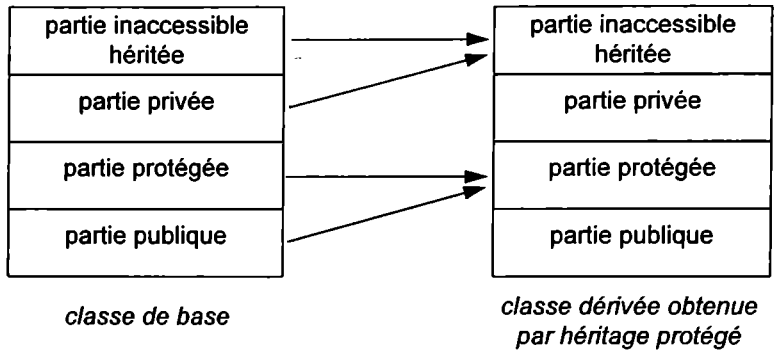
```
class Tableau
{
    int *tab, nb ;

  public :
    Tableau() ;
    Tableau(const Tableau &) ;
    ~Tableau() ;
    Tableau &operator=(const Tableau &) ;
    int &operator[](int) ;
} ;
```

il est possible de concevoir une classe `Pile`. Mais il est nécessaire de cacher à l'utilisateur de `Pile` la possibilité d'accéder à n'importe quel élément du tableau via l'opérateur `[]`. Pour cette raison, l'héritage utilisé est privé :

```
class Pile : Tableau        // => héritage privé par défaut
{
    int sommet ;

  public :
    Pile(int n) { sommet = -1 ; }
    void Empiler(int e) { (*this)[++sommet] = e ; }
    int Depiler() { return (*this)[sommet--] ; }
    bool Vide() { return sommet==-1 ; }
} ;

void main()
{
  Pile p(100) ;
  p.Empiler(1) ;                                    // ok
  p[1] = 2 ;            // illégal, operator[] inaccesible
}
```

ii. L'héritage protégé

C'est un héritage moins restrictif quant à l'accès des membres hérités. Par un héritage protégé, les membres publics et protégés des classes de base directes deviennent des membres protégés de la classe dérivée.



Comme pour l'héritage privé, aucun membre d'une classe A n'est donc accessible aux utilisateurs d'une classe B obtenue par héritage protégé à partir de A. Par contre, les membres protégés et publics de A sont accessibles aux héritiers de B. Illustration, si C est une classe dérivée de B :

```
class A
{
    int priv ;

  protected :
    int prot ;

  public :
    int pub ;
} ;

class B : protected A     // A classe de base protégée de B
{
} ;

/* ... */

void C::f()
{
  priv = 1 ;                                    // illégal
  prot = 1 ;                                        // ok
  pub = 1 ;                                         // ok
}

void main()
{
  B b ;

  b.priv = 1 ;                                  // illégal
  b.prot = 1 ;                                  // illégal
  b.pub = 1 ;                                   // illégal
}
```

Là encore, des accès à des membres non privés de A peuvent être donnés aux utilisateurs de B, à l'aide de déclarations `using`.

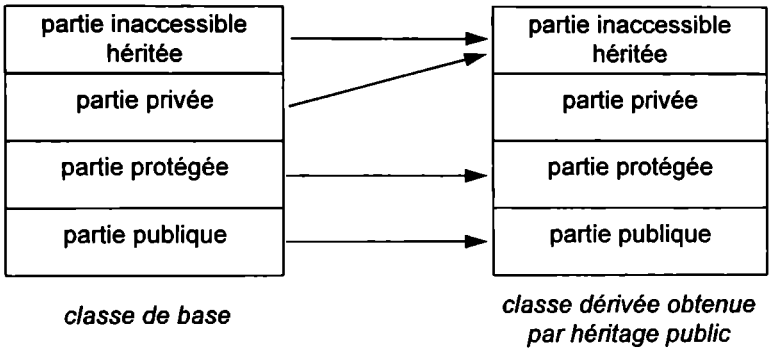
L'héritage protégé s'utilise lorsque l'on veut restreindre les possibilités d'accès pour les utilisateurs des classes dérivées, tout en conservant des droits accès maximum pour les héritiers de la classe.

iii. L'héritage public

C'est l'héritage le moins restrictif et le plus fréquemment utilisé. Par un héritage public :

— les membres publics des classes de base directes deviennent des membres publics de la classe dérivée,

— les membres protégés des classes de base directes deviennent des membres protégés de la classe dérivée.



Contrairement aux cas précédents, l'utilisateur d'une classe B obtenue par héritage privé à partir de A dispose des membres publics de A. Et comme pour l'héritage protégé, les membres protégés et publics de A sont accessibles aux héritiers de B. Illustration, si C est une classe dérivée de B :

```
class A
{
    int priv ;

  protected :
    int prot ;

  public :
    int pub ;
  /* ... */
} ;
```

```
/* ... */

void C::f()
{
  priv = 1 ;                                    // illégal
  prot = 1 ;                                         // ok
  pub = 1 ;                                          // ok
}

void main()
{
  B b ;

  b.priv = 1 ;                                  // illégal
  b.prot = 1 ;                                  // illégal
  b.pub = 1 ;                                        // ok
}
```

Là encore, des accès à des membres non privés de `A` peuvent être donnés aux utilisateurs de `B`, à l'aide de déclarations `using`.

L'héritage public est l'héritage par défaut pour les structures, lorsque ni `public`, ni `protected`, ni `private` ne sont mentionnés.

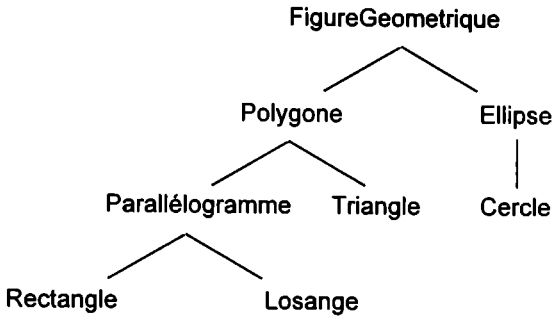
Ce type d'héritage s'utilise lorsque `B` peut être utilisé comme une sorte de `A`. Par exemple, à partir de la classe :

```
class FigureGeometrique
{
  public :
    void Placer(int x, int y) ;
    void Peindre(int couleur) ;
    void Tracer() ;
  /* ... */
} ;
```

on peut dériver publiquement :

```
class Polygone : public FigureGeometrique      { /*...*/ } ;
class Ellipse : public FigureGeometrique       { /*...*/ } ;
class Cercle : public Ellipse                  { /*...*/ } ;
class Triangle : public Polygone               { /*...*/ } ;
class Parallelogramme : public Polygone        { /*...*/ } ;
class Losange : public Parallelogramme         { /*...*/ } ;
class Rectangle : public Parallelogramme       { /*...*/ } ;
```

et ainsi de suite pour obtenir un arbre d'héritage tel que :



L'héritage public joue un rôle important, puisqu'il permet le *polymorphisme* : sous certaines conditions[89], un objet d'un type dérivé publiquement d'une classe C peut remplacer un objet de type C.

3 - Comportement des objets

i. Processus de création et de destruction d'objets

Lors de la création d'un objet, son constructeur et ceux de toutes ses classes de base sont exécutés.

Le processus d'appel des constructeurs est récursif : tout objet commence par construire ses composantes issues de ses classes de base directes avant d'exécuter les instructions de son propre constructeur.

Dans le cas particulier de l'héritage simple, les constructeurs sont donc exécutés en partant de la racine de l'arbre d'héritage et en descendant vers les classes dérivées. Pour les destructeurs, le mécanisme est le même, mais l'ordre est inversé :

```
{
  Losange l ;  /* exécution de :FigureGeometrique()
                                 Polygone()
                                 Parallelogramme()
                                 Losange() */
  /* ... */
}              /* exécution de :~Losange()
                                 ~Parallelogramme()
                                 ~Polygone()
                                 ~FigureGeometrique() */
```

Les constructeurs des classes de base sont exécutés avant ceux des objets membres.

Ce sont implicitement les constructeurs par défaut qui sont appelés. Si cela n'est pas souhaité ou pas possible, un appel explicite à un constructeur d'une classe

89 Voir *Polymorphisme* page 114.

de base directe peut être fait sur l'en-tête du constructeur de la classe dérivée, de la façon suivante :

```
class Base
{
  public :
    Base(int, char) ;
  /* ... */
} ;

class Deriv : public Base
{
  public :
    Deriv(int i) : Base(i,'a') { /* ... */ }
} ;
```

Les synthétisations réalisées par le compilateur pour la classe dérivée `B` :

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

class B : public A
{
} ;
```

donnent finalement une classe globalement équivalente à :

```
class B : public A
{
  public :
    B() : A() {}
    B(const B &b) : A(b) {}
    B &operator=(const B &b)
                    { A::operator=(b) ; return *this ; }
    ~B() {}
} ;
```

avec `~B` appelant implicitement `A::~A`.

Les parties héritées sont traitées comme des objets membres : le constructeur de recopie synthétisé appelle le constructeur de recopie des classes de base directes, et l'opérateur d'affectation synthétisé appelle les opérateurs d'affectation de recopie des classes de base directes.

Si `A` possède un constructeur par défaut non trivial, le constructeur par défaut synthétisé pour `B` n'est pas trivial, et même chose pour le constructeur de recopie, l'affectation de recopie et le destructeur synthétisés.

Les constructeurs et les destructeurs ne sont pas transmis par héritage : si une classe dérivée n'a pas défini de constructeur, elle n'hérite pas des constructeurs de sa classe de base directe et ne possède donc qu'un constructeur par défaut

synthétisé. Il est alors nécessaire que les classes de base directes aient elles aussi un constructeur par défaut[90].

ii. Redéfinition des membres

Il est possible de redéfinir un membre dans une classe dérivée. Le membre redéfini masque le membre issu de la classe de base. L'opérateur de résolution de portée `::` permet cependant d'accéder au membre caché :

```
int x ;

class A
{
  public :
    int x ;
  /* ... */
} ;

class B : public A
{
  public :
    int x ;
    void f ()
       {
         x++ ;                          // incrémente B::x
         A::x++ ;
         ::x++ ;                      // incrémente x global
       }
} ;
```

De la même façon, une fonction membre masque toutes les fonctions de même nom définies dans les classes de base. Ces fonctions ne se surchargent pas, car elles ne sont pas définies dans la même région déclarative :

```
class A
{
  public :
    void f(int) ;
    void f() ;
} ;

class B : public A
{
  public :
    void f() ;
} ;
```

90 Pour plus de détails et des exemples, voir *Constructeurs et héritage* page 262.

```
void main()
{
  B b ;
  b.f(1) ;                    // erreur : A::f(int) est masqué
  b.A::f(1) ;                                        // d'accord
  b.f() ;                          // d'accord, appelle B::f()
}
```

Ici, la fonction `B::f()` masque `A::f()` et `A::f(int)`.

4 - Polymorphisme

i. Compatibilité implicite entre classe de base et classe dérivée

Par un héritage public, une classe `B` dérivée de `A` est considérée comme une sorte de `A`. Quel que soit l'objet `a` de type `A` et l'objet `b` de type `B` dérivé publiquement de `A`, tous les services offerts par `a` sont aussi offerts par `b` : `b` peut donc remplacer `a`.

Plus généralement, si `A` est une classe de base accessible[91] de `B`, alors tous les services offerts par un objet `a` de type `A` sont aussi offerts par tout objet `b` de type `B` : `b` peut là encore remplacer `a`.

Cette propriété est exploitable au travers des pointeurs et des références : l'adresse d'un objet de type `B` peut être mémorisée dans un type pointeur ou référence vers une classe de base accessible de `B`, directe ou indirecte. L'objet référencé conserve en tout état de cause son type d'origine :

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

class B : public A
{
  /* ... */
} ;

A a, *pa ;
B b, *pb ;

pa = &b ;                        // ok, mais *pa reste de type B
pb = &a ;              // non, pas de conversion dans ce sens !
```

Ici, `A` est une classe de base accessible de `B` : il est donc possible de faire pointer le pointeur `pa`, de type `A *`, sur un objet de type `B`. L'objet pointé par le

---

91 Voir la définition de *classe de base accessible* page 103.

pointeur `pa` reste de type `B`. On dit que le *type statique* de `*pa` est `A` et que son *type dynamique* est `B`.

Par définition, le type statique d'une expression est le type obtenu à partir d'une analyse statique du code, sans aucune considération de sémantique d'exécution. Il est déterminé à la compilation. Le type dynamique d'une expression est déterminé par la valeur courante de l'expression, et peut changer en cours d'exécution.

Ainsi, tout objet pointé par un type `A *` est soit de type `A`, soit de type `B`, avec `A` classe de base accessible de `B`[92]. Et le type de l'objet pointé peut varier en cours d'exécution.

C'est la même chose avec des références :

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

class B : public A
{
  /* ... */
} ;

A a;
B b ;

A &ra = b ;                          // ok, c'est compatible
B &rb = a ;              // non, incompatible dans ce sens !
```

Le type statique de `ra` est `A`, mais son type dynamique est `B`. De manière générale, tout objet référencé par un type `A &` est soit de type `A`, soit de type `B`, avec `A` classe de base accessible de `B`. Et le type de l'objet référencé peut varier en cours d'exécution.

Grâce à ces conversions implicites de pointeurs et références, tout objet peut être traité comme un objet plus général. Cela permet de traiter collectivement un ensemble d'objets de types différents :

```
{
  FigureGeometrique *figure[50] ;
  figure[0] = new Triangle ;
  figure[1] = new Losange ;
  figure[2] = new Polygone ;
  figure[3] = new Ellipse ;
  figure[4] = new Cercle ;
  figure[5] = new FigureGeometrique ;
```

---

92 On ne peut donc plus affirmer, comme en C, qu'étant donné un pointeur `p` de type `T*`, `*p` est de type `T`.

```
    figure[6] = new Rectangle ;
    figure[7] = new Parallelogramme ;
    figure[0] = new Ellipse ;
}
```

Le type statique des objets `*figure[i]` est `FigureGeometrique`. Le type dynamique de `*figure[0]` est `Triangle` après exécution de la première instruction. Il devient `Ellipse` après exécution de la dernière.

Les *fonctions virtuelles*[93] offrent la possibilité d'exploiter automatiquement les services du type dynamique.

Cette conversion implicite n'existe pas si l'héritage est privé ou protégé[94].

ii. Liaison statique

Etant données les deux classes :

```
class A
{
  public :
    int a ;
} ;

class B : public A
{
  public :
    int a ;
} ;
```

on peut se demander quelle valeur reçoit `c` dans :

```
B b ;
A *ptr = &b ;
int c = ptr->a ;
```

ou dans :

```
B b ;
A &ref = b ;
int c = ref.a ;
```

---

[93] Voir *Liaison dynamique : les fonctions virtuelles* page 118.

[94] Si tel n'était pas le cas, le masquage des membres publics de la classe de base `A` mis en oeuvre lors de l'héritage vis-à-vis des utilisateurs de la classe dérivée `B` serait caduc, puisque un objet de type `B` pouvant jouer, par compatibilité, le rôle d'un objet de type `A`, l'utilisateur d'un objet `b` de type `B` pourrait accéder aux membres publics de `A` simplement en faisant jouer à `b` le rôle d'un objet de type `A`.

Est-ce la valeur du membre `a` de `A` ou celle du membre `a` de `B` ? Autrement dit, la sélection des membres est-elle fonction du type statique ou du type dynamique ?

La réponse peut sembler décevante : la sélection d'un membre est déterminé statiquement à la compilation, d'après le type du pointeur ou de la référence, et non pas en fonction du type de l'objet pointé[95]. Ainsi, dans l'exemple, `c` reçoit la valeur de `A::a`. De même, les fonctions membres sont par défaut liées statiquement à leurs appels :

```
class FigureGeometrique
{
  public :
    void Tracer() {}
} ;

class Polygone : public FigureGeometrique
{
  public :
    void Tracer() { /* ... */ }
} ;

class Triangle : public Polygone {} ;
class Parallelogramme : public Polygone {} ;
class Rectangle : public Parallelogramme {} ;
class Losange : public Parallelogramme {} ;

class Ellipse : public FigureGeometrique
{
  public :
    void Tracer() { /* ... */ }
} ;

class Cercle : public Ellipse {} ;

void Dessiner(FigureGeometrique *figure[], int nb)
{
  for (int i=0 ; i<nb ; i++)
    figure[i]->Tracer() ;                    // liaison statique
}

void main()
{
  FigureGeometrique *figure[100] ;

  figure[0] = new Triangle ;
  figure[1] = new Cercle ;
```

95 Cela se justifie par un souci d'efficacité et de compatibilité avec le langage C.

```
    figure[2] = new Losange ;
    figure[3] = new FigureGeometrique ;
    /* ... */
    figure[99] = new Rectangle ;
    Dessiner(figure,100) ;
}
```

L'appel `figure[i]->Tracer()` est lié à la fonction `Tracer` de la classe `FigureGeometrique`, puisque le type statique de `*figure[i]` est `FigureGeometrique`. Par conséquent, la fonction `Dessiner` ci-dessus ne fait rien (elle appelle `nb` fois une fonction qui ne fait rien), et n'a donc aucun intérêt.

iii. Liaison dynamique : les fonctions virtuelles

Dans l'exemple précédent, il aurait été souhaitable que la fonction appelée soit celle correspondant au type des objets pointés par les `figure[i]`. Mais ce type étant dynamique, il est impossible que la fonction soit déterminée à la compilation. Le compilateur met donc en place une liaison dynamique pour qu'à l'exécution, la fonction appelée soit en rapport avec le type de l'objet pointé au moment de l'appel. C'est ce mécanisme qui est mis en œuvre lors d'un appel à une fonction virtuelle.

Syntaxiquement, mettre en œuvre une liaison dynamique se résume simplement à ajouter le mot clé `virtual` lors de la déclaration de la fonction :

```
class FigureGeometrique
{
  public :
    virtual void Tracer() { erreur() ; }
} ;

class Polygone : public FigureGeometrique
{
  public :
    virtual void Tracer() { /* ... */ }
} ;

class Triangle : public Polygone {} ;
class Parallelogramme : public Polygone {} ;
class Rectangle : public Parallelogramme {} ;
class Losange : public Parallelogramme {} ;

class Ellipse : public FigureGeometrique
{
  public :
    void Tracer() { /* ... */ }
} ;
```

```
class Cercle : public Ellipse {} ;

void Dessiner(FigureGeometrique *figure[], int nb)
{
  for (int i=0 ; i<nb ; i++)
    figure[i]->Tracer() ;              // liaison dynamique
}

void main()
{
  FigureGeometrique *figure[100] ;

  figure[0] = new Triangle ;
  figure[1] = new Cercle ;
  figure[2] = new Losange ;
  figure[3] = new FigureGeometrique ;
  /* ... */
  figure[99] = new Rectangle ;
  Dessiner(figure,100) ;
}
```

Bien que `figure` soit déclaré comme un tableau de pointeur de `FigureGeometrique`, c'est maintenant la fonction `Tracer` de chacun des objets pointés qui est appelée à chaque fois.

Quelques remarques au sujet des fonctions virtuelles :

— les contrôles du compilateur restent statiques, même lorsqu'il s'agit d'une fonction virtuelle : dans l'exemple précédent, la définition de la fonction `FigureGeometrique::Tracer()` reste nécessaire pour que l'expression `figure[i]->Tracer()` soit légale ;

— tout comme une fonction membre ordinaire, une fonction virtuelle sert de fonction par défaut dans les classes dérivées ou elle n'a pas été redéfinie : dans l'exemple précédent, c'est la fonction `Tracer` de `Polygone` qui est appelée pour les triangles, parallélogrammes, rectangles et losanges, et la fonction `Tracer` d'`Ellipse` pour les cercles ;

— une fonction virtuelle reste virtuelle dans les classes dérivées, même si cela n'est pas spécifié : dans l'exemple, la fonction `Tracer` d'`Ellipse` est virtuelle ;

— les fonctions statiques ne peuvent pas être virtuelles ;

— l'implémentation des fonctions virtuelles s'appuie sur une indirection, donc l'appel consomme légèrement plus de temps que celui d'une fonction ordinaire[96] ;

— les unions ne supportent pas les fonctions virtuelles.

---

96 Voir *Implémentation des fonctions virtuelles* page 123.

Un constructeur ne peut pas être virtuel[97]. Un destructeur, par contre, peut être virtuel. C'est même obligatoire lorsque des objets de la classe sont susceptibles d'être créés dynamiquement par l'intermédiaire d'un pointeur ou d'une référence vers un type de base[98].

Des contraintes sont à respecter lors de la redéfinition d'une fonction virtuelle. Prenons une fonction virtuelle `A::f` redéfinie dans une classe dérivée `B`. La première contrainte concerne les paramètres : `A::f` et `B::f` doivent avoir la même liste de paramètre. Si ce n'est pas le cas, `B::f` ne redéfinit pas `A::f` :

```
class A
{
  public :
    virtual void f(int) ;
} ;

class B : public A
{
  public :
    virtual void f(char) ;
} ;

void main()
{
  A *ptr = new B ;
  ptr->f('A') ;                                    // ptr->f(int)
}
```

Ici, `B::f(char)` ne redéfinit pas `A::f(int)` : `ptr->f('A')` appelle `A::f(int)`, bien que le type dynamique de `*ptr` soit `B` et que l'argument soit de type `char`.

Et non seulement il n'y a pas de redéfinition, mais en plus il y a masquage. `B::f` masque `A::f` :

```
void main()
{
  B b ;
  b.f(1) ;                                         // b.B::f(char)
}
```

Ici, `b.f(1)` appelle `B::f(char)`, bien que l'argument soit de type `int` et qu'il existe une fonction `f(int)` dans la classe de base.

La seconde contrainte concerne les types de retour : ils doivent vérifier l'une des deux conditions suivantes :

— les types de retour de `A::f` et `B::f` sont égaux, ou

---

97 Voir des solutions de remplacement au chapitre *Constructeurs virtuels* page 322.

98 Voir *Destructeurs virtuels* page 324.

— les types de retour de `A::f` et `B::f` sont respectivement des pointeurs ou des références vers des classes `C` et `D`, avec `C` classe de base accessible de `D`.

Dans le cas général, cela veut dire :

```
class C {} ;
class D : public C {} ;

class A
{
  virtual C &f() ;
} ;

class B : public A
{
  virtual D &f() ;                          // redéfinit A::f()
} ;
```

Un cas particulier intéressant est celui où `C=A` et `D=B` :

```
class A
{
  virtual A &f() ;
} ;

class B : public A
{
  virtual B &f() ;                          // redéfinit A::f()
} ;
```

Si cette contrainte n'est pas vérifiée et si la première contrainte l'est, il y a erreur à la compilation :

```
class A
{
  public :
    virtual int f(int) ;
} ;

class B : public A
{
  public :
    void f(int) ;                                    // erreur
} ;
```

Une classe ayant au moins une fonction virtuelle (définie dans la classe ou obtenue par héritage) s'appelle une classe *polymorphe*.

Le concept de fonction virtuelle permet d'implémenter des mécanismes généraux dans les classes de base, avant même que les détails du traitement ne soient connus. Ces détails pourront être spécifiés ultérieurement dans les classes dérivées : ils seront aussitôt utilisés par la classe de base. Par exemple :

```
class FigureGeometrique
{
    Couleur couleur ;                    // couleur du tracé
    const static Couleur fond ;         // couleur de la page

  public :
    virtual void Tracer() { erreur() ; }
    void Effacer() {
                      Couleur c=couleur ;
                      couleur = fond ;
                      Tracer() ;
                      couleur = c ;
                    }
  /* ... */
} ;
```

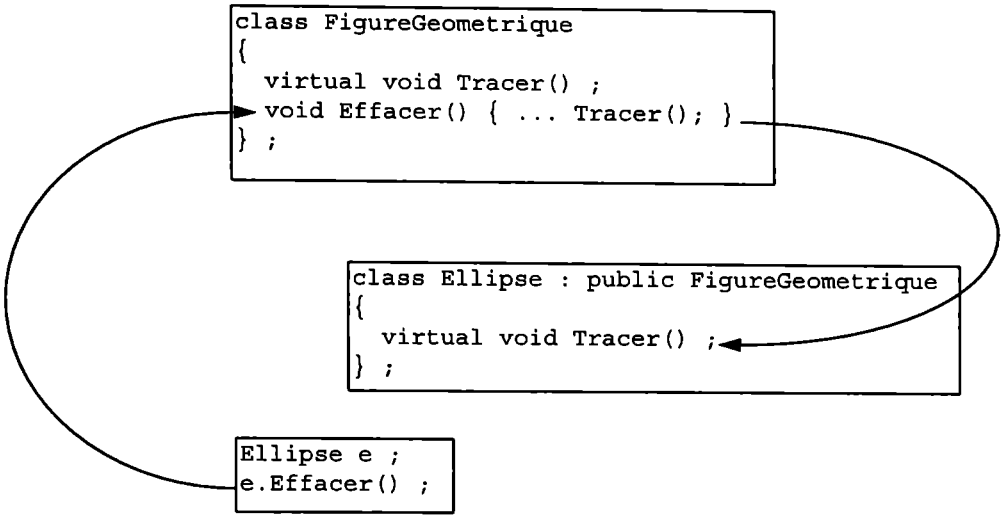
Bien que le service `Tracer` ne soit pas implémenté dans `FigureGeometrique`, la fonction `Effacer` l'utilise : on suppose ici qu'effacer une figure, c'est la retracer avec la couleur du fond. Ce qui est remarquable, c'est que cette fonction `Effacer` est déjà valable pour toutes les classes dérivées de `FigureGeometrique`, et n'aura pas besoin d'être redéfini par la suite :

```
class Ellipse : public FigureGeometrique
{
  public :
    virtual void Tracer()
            { /* implémentation du tracé de l'ellipse */ }
} ;
```

Il est inutile de redéfinir `Effacer` dans la classe `Ellipse`. Le simple fait de redéfinir `Tracer` fait fonctionner correctement la fonction `FigureGeometrique::Effacer` pour des ellipses :

```
Ellipse e ;
e.Effacer() ;                    // fonctionne correctement
```

Cette dernière instruction fait appel à la fonction `FigureGeometrique::Effacer()` qui fait appel à la fonction `Ellipse::Tracer()` :



En effet, l'appel `Tracer()`, dans la fonction `FigureGeometrique::Effacer`, est équivalent à `this->Tracer()`. Lorsque `Effacer` est appelée pour un objet `Ellipse`, `this` est de type `Ellipse *`. Par conséquent, `this->Tracer()` dans `FigureGeometrique::Effacer` appelle `Ellipse::Tracer`.

Il est remarquable que la fonction `FigureGeometrique::Effacer` fasse appel à une fonction qui n'existait probablement pas lors de sa compilation : la création de toute nouvelle classe dérivée de `FigureGeometrique` ne nécessite une recompilation ni de la fonction `FigureGeometrique::Effacer()`, ni de la classe `FigureGeometrique`. Cela peut paraître surprenant ! L'implémentation de ce mécanisme est détaillé au paragraphe suivant.

Les constructeurs par défaut, constructeur de recopie, destructeur et opérateur d'affectation synthétisés pour une classe polymorphe ne sont par définition pas triviaux.

iv. Implémentation des fonctions virtuelles

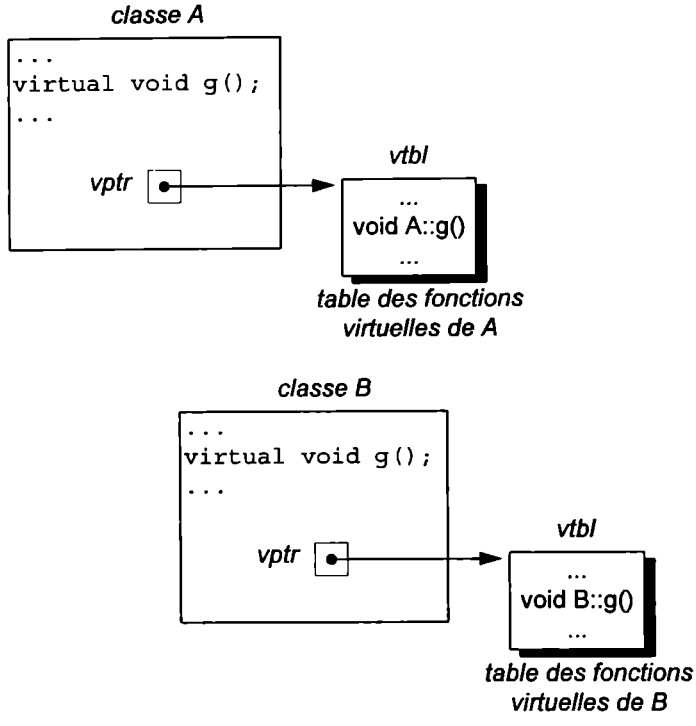
Toute classe polymorphe contient un pointeur supplémentaire en plus de ses membres. Ce pointeur `vptr`, fait référence à une table `vtbl` de pointeurs vers les fonctions virtuelles de la classe. Chaque appel à une fonction virtuelle se fait alors via un pointeur de cette table.

Le principe utilisé pour implémenter les liaisons dynamiques des deux classes :

```
class A
{
  public :
    virtual void g() ;
} ;
```

```
class B : public A
{
  public :
    virtual void g() ;
} ;
```

peut se schématiser de la façon suivante :



Les appels aux fonctions virtuelles se font alors par l'intermédiaire du pointeur `vptr`. Si `ptr` est de type `A*`, l'appel `ptr->g()` est interprété comme l'appel de l'une des fonctions de la table, plus précisément comme `(*(ptr->vptr[i]))()`, où `i` est l'indice de l'élément de la table contenant l'adresse de `g`.

Comme `vptr` se trouve dans l'objet lui-même, la table utilisée est celle correspondant au type de l'objet, et la fonction virtuelle appelée est donc dépendante du type de l'objet pointé.

Ainsi dans :

```
A *ptr = new A ;
ptr->g() ;
```

l'expression `(*(ptr->vptr[i]))()` appelle bien `A::g`, alors que dans :

```
A *ptr = new B ;
ptr->g() ;
```

la même expression `(*(ptr->vptr[i]))()` appelle `B::g`.

On obtiendrait exactement la même chose avec des références :

```
void f(A &a)
{
  a.g() ;
}
```

cette instruction est interprétée comme `(a.vptr[0])()`.

Lorsque l'héritage est multiple, l'implémentation de ce mécanisme est plus subtile...

v. Fonctions virtuelles pures

Certaines fonctions virtuelles sont trop générales pour pouvoir être implémentées. C'est le cas de la fonction `Tracer` de la classe `FigureGeometrique`. Cette fonction n'a jamais à être appelée, mais elle est quand-même nécessaire pour pouvoir écrire `figure[i]->Tracer()`.

Dans l'exemple précédent, `FigureGeometrique::Tracer` était simplement implémentée par une fonction envoyant un message d'erreur. Une meilleure solution est de déclarer la fonction `FigureGeometrique::Tracer` *virtuelle pure*, de la façon suivante :

```
class FigureGeometrique
{
    Couleur couleur ;                    // couleur du tracé
    const static Couleur fond ;         // couleur de la page

  public :
    virtual void Tracer()=0 ;              // virtuelle pure
    void Effacer() {
                      Couleur c=couleur ;
                      couleur = fond ;
                      Tracer() ;                        // oui
                      couleur = c ;
                    }
  /* ... */
} ;
```

Cela signifie que l'implémentation de `Tracer` est différée et déléguée aux classes dérivées. En clair : il n'existe pas de traitement par défaut pour tracer n'importe quelle figure géométrique.

Une classe ayant une fonction virtuelle pure ne peut pas être instanciée.

vi. Classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée. C'est en général une classe dont l'implémentation est insuffisante pour représenter

complètement un objet réel. Le rôle de la classe se réduit simplement à servir de moule pour dériver d'autres classes.

Une classe ayant tous ses constructeurs protégés et privés est évidemment une classe abstraite. Une classe déclarant une fonction virtuelle pure est aussi une classe abstraite.

Dans l'exemple précédent, l'instanciation d'une figure géométrique n'a pas de sens, puisque l'objet obtenu ne pourrait ni se tracer, ni s'effacer. `FigureGeometrique` est un cas typique de classe abstraite. On ne peut donc pas l'instancier :

```
class FigureGeometrique
{
  public :
    virtual void Tracer()=0 ;
  /* ... */
} ;

FigureGeometrique figure ;                    // illégal
FigureGeometrique *ptr ;                          // ok
ptr = new FigureGeometrique ;                 // illégal
```

Une classe qui hérite d'une fonction virtuelle pure et qui ne l'implémente pas est également une classe abstraite :

```
class Convexe : public FigureGeometrique
{
} ;
```

La classe `Convexe` est une classe abstraite puisqu'elle hérite de la fonction virtuelle pure `Tracer` de `FigureGeometrique`, et ne l'implémente pas.

5 - Identification dynamique de type

Un mécanisme d'identification de type est mis en œuvre à l'exécution pour les types polymorphes. Il permet d'exploiter, en cours d'exécution, des informations relatives au type des objets, de manière portable sur toute implémentation.

i. L'opérateur `dynamic_cast`

Cet opérateur réalise des transtypages sûrs. Sa syntaxe est :

```
dynamic_cast<T>(p)
```

où *T* représente un type pointeur et *p* un pointeur vers un type polymorphe[99]. Cet opérateur convertit son opérande *p* en *T* seulement si **p* est une sorte de *T**, c'est-à-dire si le type dynamique de **p* est soit *T**, soit un type dont *T** est une classe de base accessible. Sinon, l'expression vaut 0 :

```
class A
{
  virtual void f() {}                    // pour être polymorphe
} ;

class B : public A {} ;

class C : public B {} ;

void main()
{
  A *ab = new B ;
  A *ac = new C ;
  C *cc = dynamic_cast<C*>(ac) ;      // transtypage réussi
  B *bb = dynamic_cast<B*>(ac) ;      // transtypage réussi
     bb = dynamic_cast<B*>(ab) ;      // transtypage réussi
     cc = dynamic_cast<C*>(ab) ;       // échec : cc vaut 0
}
```

Cet opérateur réalise donc deux opérations : il vérifie que le transtypage est valide et si c'est le cas, le réalise. Pour convertir un pointeur vers *classe de base* en un pointeur vers *classe dérivée*, c'est donc un transtypage sûr, contrairement à celui réalisé par un *cast* habituel ou par un `static_cast`.

Pour une conversion *pointeur vers type dérivé* vers *pointeur vers type de base*, tous les transtypages ci-dessous donnent le même résultat :

```
class A
{
  virtual void f() ;                     // pour être polymorphe
} ;

class B : public A {} ;
```

99 Plus précisément, pour réaliser des conversions *pointeur vers type de base* vers *pointeur vers type dérivé* (cas le plus intéressant), *p* doit pointer vers un type polymorphe. Sinon, pour simplement réaliser des conversions *pointeur vers type dérivé* vers *pointeur vers type de base*, cette condition n'est pas nécessaire.

```
void main()
{
  B *b = new B ;
  A *a ;
  a = (A *) b ;
  a = static_cast<A *> (b) ;
  a = dynamic_cast<A *> (b) ;
  a = b ;
}
```

Par contre, pour une conversion *pointeur vers type de base* vers *pointeur vers type dérivé*, si l'objet pointé n'est pas du type dérivé, seul le `dynamic_cast` est utilisable :

```
A *a = new A;
B *b ;
b = (B *) a ;                          // résultat inutilisable
b = static_cast<B *> (a) ;             // résultat inutilisable
b = dynamic_cast<B *> (a) ;                          // b vaut 0
b = a ;                                // erreur de compilation
```

Les deux premiers *casts* forcent la conversion, et renvoient une valeur incorrecte. Par contre, le `dynamic_cast`, en renvoyant `0`, indique que l'objet ne peut pas être pointé par un type `B *`. Quant à la dernière instruction, elle est illégale, puisqu'il n'y a pas de conversion implicite dans ce sens.

Pour une conversion *pointeur vers type de base* vers *pointeur vers type dérivé*, lorsque l'objet pointé est du type dérivé, seul le `dynamic_cast` donne un résultat sûr :

```
A *a = new B;
B *b ;
b = (B *) a ;                    // résultat correct par chance
b = static_cast<B *> (a) ;       // résultat correct par chance
b = dynamic_cast<B *> (a) ;        // résultat correct et sûr
b = a ;                              // erreur de compilation
```

Les deux premiers *casts* forcent la conversion, et renvoient, par chance, une valeur correcte puisque `*a` est de type `B`. Par contre, le `dynamic_cast`, en renvoyant une valeur non nulle, indique que l'objet peut sans risque être pointé par un type `B *`. La dernière instruction, est toujours illégale, pour la même raison que précédemment.

Le `dynamic_cast` s'applique aussi aux références et s'utilise de la même façon pour convertir des références sur des classes de base en références sur des classes dérivées. La syntaxe est :

`dynamic_cast``<`*`T`*`>(`*`r`*`)`

où *T* représente un type référence et *r* une référence. Soit *T'* le type quel que *T*=*T'*`&`, cet opérateur convertit son opérande *r* en *T* seulement si *r* référence

réellement une sorte de `T'`, c'est-à-dire si le type dynamique de `r` est soit `T'`, soit un type dont `T'` est une classe de base accessible. Si tel n'est pas le cas, l'exception `bad_cast`[100] est lancée. Des exemples illustrent l'utilisation de cet opérateur aux paragraphes *Plus sur le dynamic_cast* page 312 et *Opérateurs virtuels* page 326.

ii. L'opérateur `typeid`

Le *cast* dynamique n'est qu'une des possibilités offertes par le mécanisme d'identification des types. L'opérateur `typeid` en est une autre. Il permet de réaliser des opérations plus élaborées. Sa syntaxe est :

`typeid``(`*`type`*`)`

ou

`typeid``(`*`expression`*`)`

et son résultat est de type `const type_info &`, où `type_info` est une classe de la bibliothèque standard C++[101]. `type_info` contient des informations de type relatives à l'opérande de `typeid`.

Lorsque l'opérande est d'un type polymorphe, les informations renvoyées concernent le type dynamique. Sinon, `typeid` renvoie des informations sur le type statique. `typeid` peut également être utilisé avec les types prédéfinis. Si son opérande est un pointeur `NULL` déréférencé, une exception `bad_typeid`[102] est lancée.

`type_info` dispose d'opérateurs `==` et `!=` permettant de déterminer si deux objets sont de même type :

```
class A
{
  virtual void f() {} ;              // pour être polymorphe
} ;

class B : public A
{} ;

void main()
{
  bool b ;
  A *pa = new B ;
```

[100] Voir *bad_cast* page 191.

[101] Voir *La classe type_info* page 188.

[102] Voir *bad_typeid* page 191.

```
    b = typeid(pa)==typeid(A *) ;                    // true
    b = typeid(pa)==typeid(B *) ;                    // false
    b = typeid(*pa)==typeid(A) ;                     // false
    b = typeid(*pa)==typeid(B) ;                     // true

    A &ra=*pa ;
    b = typeid(ra)==typeid(A) ;                      // false
    b = typeid(ra)==typeid(B) ;                      // true
}
```

La fonction `type_info::name` renvoie le nom complet du type, sous la forme d'une chaîne de caractères :

```
class A
{
  virtual void f() {} ;              // pour être polymorphe
} ;

class B : public A
{} ;

void main()
{
  A *pa = new B ;
  cout << "typeid(pa) : "
       << typeid(pa).name() << "\n" ;
  cout << "typeid(*pa) : "
       << typeid(*pa).name() << "\n" ;

  A &ra=*pa ;
  cout << "typeid(ra) : "
       << typeid(ra).name() << "\n" ;
}
```

ce qui donne :

```
typeid(pa) : A *
typeid(*pa) : B
typeid(ra) : B
```

6 - Particularités de l'héritage multiple

i. Généralités

Une classe peut avoir plus d'une classe de base directe :

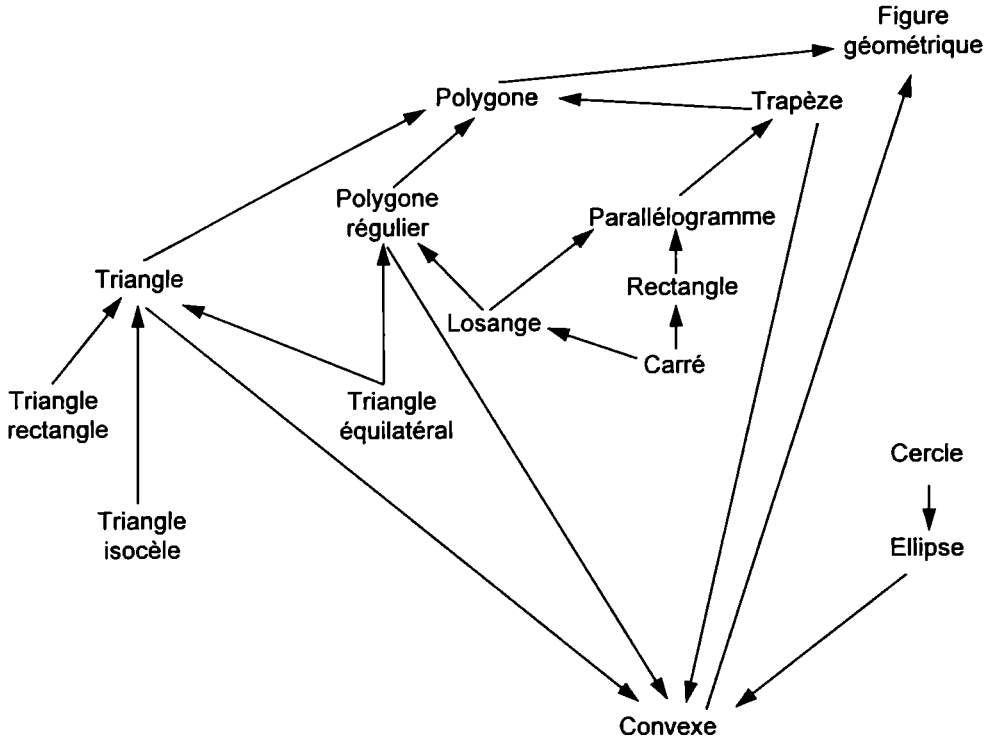
```
class FigureGeometrique
{
  public :
    enum Couleur { NOIR, ROUGE, BLEU, VERT, JAUNE,
                                                BLANC } ;
    FigureGeometrique(Couleur c) ;
    virtual void Tracer()=0 ;
    void Effacer () ;

  protected :
    Couleur couleur ;
} ;

class Complexe
{
  protected :
    double x, y ;

  public :
    Complexe (double=0, double=0) ;
    double Module() ;
    double Argument() ;
  /* ... */
} ;

class Point : private Complexe,
              public FigureGeometrique
{
  public :
    Point(int x, int y, Couleur c) :
                    FigureGeometrique(c), Complexe(x,y) {}
    void Tracer() { plot(x,y,couleur) ; }
    double Distance() { return Module() ; }
    double Azimut() { return Argument() ; }
} ;
```

Dans cet exemple, la classe `Point` est dérivée de `Complexe` et de `FigureGeometrique`. Un point possède donc deux coordonnées et une couleur. La classe `Point` renomme de façon plus explicite les membres de `Complexe` : `Module` est renommé `Distance`, et `Argument` est renommé `Azimut`.

Les constructeurs des classes de base directes sont appelés suivant l'ordre défini par la liste d'héritage, et non pas suivant celui des initialisations spécifiées sur l'en-tête du constructeur. Ainsi, dans l'exemple précédent, lors de la construction d'un point, le constructeur de `Complexe` est exécuté avant celui de `FigureGeometrique`.

Une classe ne peut pas apparaître en plusieurs exemplaires dans la liste des classes de base. Plus généralement, les classes liées par une relation d'héritage

forment un graphe orienté acyclique. Le graphe suivant représente la relation d'héritage sur un ensemble de figures géométriques :



ii. Conflits de noms

Plusieurs classes de base peuvent avoir des membres de même nom. Cela pose problème dans la classe dérivée s'ils sont utilisés sans qualification de portée :

```
class FigureGeometrique
{
  public :
  /* ... */
    void Imprimer(ostream &) ;
} ;

class Complexe
{
  public :
  /* ... */
    void Imprimer(ostream &) ;
} ;
```

```
class Point : public FigureGeometrique, public Complexe
{} ;

void f(Point p)
{
  p.Imprimer(cout) ;                    // illégal, car ambigu
}
```

L'appel `p.Imprimer(cout)` est ambigu : est-ce :

`p.Complexe::Imprimer(cout)`, ou

`p.FigureGeometrique::Imprimer(cout)` ?

Il est nécessaire de lever l'ambiguïté à chaque fois qu'elle apparaît :

```
void f(Point p)
{
  /* ... */
  p.Complexe::Imprimer(cout) ;                          // bien
  p.FigureGeometrique::Imprimer(cout) ;          // d'accord
}
```

Le mieux est de définir carrément une nouvelle fonction `Imprimer` dans `Point` :

```
class Point : public Complexe, public FigureGeometrique
{
  public :
    void Imprimer(ostream &c)
                { /*
                     en utilisant éventuellement
                     Complexe::Imprimer(c) ;
                           et/ou
                     FigureGeometrique::Imprimer(c) ;
                  */
                }
  /* ... */
} ;
```

Une telle ambiguïté apparaît notamment lorsque une classe se retrouve indirectement plusieurs fois classe de base :

```
class A
{
  public :
    int a ;
    static int s ;
    enum { e1, e2 } ;
    typedef float F ;
} ;
```

```
class B : public A
{} ;

class C : public A
{} ;

class D : public B, public C
{} ;
```

Ici, la classe `D` possède deux occurrences de la classe `A` dans ses classes de base. Son membre `a` ne peut pas être accédé sans qualification de portée explicite à partir d'un objet de type `D` :

```
void main()
{
  D d ;
  d.a = 1 ;                          // erreur car ambigu
  d.B::a = 1 ;                       // ok
}
```

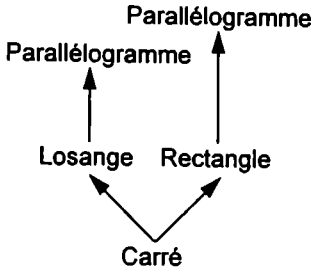
Cependant, les membres statiques, les types et les énumérateurs peuvent être référencés sans ambiguïté :

```
void main()
{
  D d ;
  int i=d.e1 ;                       // oui
  d.s++ ;                            // oui
  D::F x ;                           // oui
}
```

iii. Classe de base virtuelle

L'héritage multiple peut être utilisé pour associer des classes qui n'ont pas été a priori conçues pour travailler ensemble. Dans l'exemple précédent, la classe `Complexe` peut être issue d'une bibliothèque mathématique, tandis que `FigureGeometrique` peut provenir d'une bibliothèque graphique.

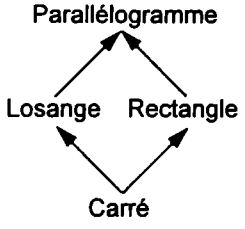
Mais, dans certains cas, l'héritage multiple est utilisé pour réunir des classes qui ont été conçues pour coopérer. Elles peuvent alors très bien dériver d'une même classe de base. Dans la hiérarchie de figures géométriques de la page 132, la classe `Carre` est issue de `Losange` et de `Rectangle`, elles-mêmes issues de `Parallelogramme`. Par un héritage ordinaire, les membres de `Parallelogramme` se retrouvent en double exemplaire dans `Carre` :



Pour coopérer, les classes `Losange` et `Rectangle` ont besoin de partager leurs membres issus de `Parallelogramme`. Pour réaliser cela, `Parallelogramme` doit être spécifié *classe de base virtuelle* pour `Losange`, `Rectangle` et `Carre`, de la façon suivante :

```
class Losange : public virtual Parallelogramme
{
  /* ... */
} ;

class Rectangle : public virtual Parallelogramme
{
  /* ... */
} ;

class Carre : public virtual Parallelogramme,
              public Losange, public Rectangle
{
  /* ... */
} ;
```

Toute instance de `Carre` contient maintenant une unique instance de `Parallelogramme` :



Les classes de base virtuelles sont toujours construites en premier, avant les non virtuelles, et indépendamment de l'ordre des déclarations des classes de base et des initialisations.

Ici, lors de la création d'un carré, le constructeur de `Parallelogramme` est appelé une seule fois. Il est exécuté avant ceux de `Losange` et de `Rectangle`. Et c'est l'inverse pour les destructeurs :

```
{
  Carre c ;     /* exécution de : Parallelogramme()
                                 Losange()
                                 Rectangle()
                                 Carre */
  /* ... */
}
              /* exécution de : ~Carre
                                ~Rectangle()
                                ~Losange()
                                ~Parallelogramme() */
```

S'il y a des arguments à transmettre, c'est donc au constructeur de `Carre` de les fournir au constructeur de `Parallelogramme` :

```
Carre::Carre(Point a, Point b, Point c) :
Parallelogramme(a,b,c), Losange(a,b,c), Rectangle(a,b,c) {}
```

iv. Exemple d'utilisation de l'héritage multiple

L'héritage multiple peut être utilisé pour implémenter un type abstrait de données à partir d'une structure de données fondamentale. Un type abstrait, comme la pile ou l'ensemble, peut être implémenté par différentes structures de données, comme un tableau ou une liste chaînée.

Le type abstrait est défini par ses propriétés. Par exemple, le type abstrait *ensemble* pourrait être spécifié de la façon suivante :

```
class EnsembleAbstrait
{
  public :
    virtual void Ajouter(int)=0 ;
    virtual void Enlever(int)=0 ;
    virtual bool Contient(int)=0 ;
} ;
```

La structure de données fondamentale `Tableau` suivante :

```
class Tableau
{
    int *tab ;

  public :
    Tableau(int n) { tab = new int[n] ; }
    ~Tableau() { delete[] tab ; }
    int &Element(int i) { return tab[i] ; }
  /* ... */
} ;
```

permet de réaliser une implémentation du type `EnsembleAbstrait` :

```
class TEnsemble : public EnsembleAbstrait, private Tableau
{
    int nb ;

  public :
    TEnsemble(int taille) : Tableau(taille) { nb = 0 ; }
    void Ajouter(int e)
              { if ( !Contient(e) ) Element(nb++) = e ; }
    void Enlever(int e) ;
    bool Contient(int e) ;
} ;

void TEnsemble::Enlever(int e)
{
  int i ;
  for (i=0 ; i<nb && Element(i)!=e ; i++) ;
  if (i<nb) Element(i) = Element(nb--) ;
}

bool TEnsemble::Contient(int e)
{
  int i ;
  for (i=0 ; i<nb && Element(i)!=e ; i++) ;
  return i<nb ;
}
```

Comme l'héritage de `Tableau` est privé, l'utilisateur de `TEnsemble` ne peut utiliser que les membres de `EnsembleAbstrait`. Par conséquent, l'utilisation d'un ensemble devient donc indépendante de la structure de données fondamentale sous-jacente :

```
void f(EnsembleAbstrait &e)
{
  e.Ajouter(2) ;
  /* ... */
  if (e.Contient(5)) { /* ... */ }
  /* ... */
  e.Enlever(1) ;
  /* ... */
}
```

Ici, par exemple, la fonction `f` utilise les services de l'ensemble `e` sans connaître son implémentation. Ainsi dans l'exemple :

```
TEnsemble e(100) ;
f(e) ;
```

`f` va faire appel aux membres de `Tableau`, alors que dans :

```
class LEnsemble : public EnsembleAbstrait, private Liste
{
  public :
    LEnsemble() ;
    void Ajouter(int) ;
    void Enlever(int) ;
    bool Contient(int) ;
  /* ... */
} ;

LEnsemble e ;
f(e) ;
```

la même fonction `f` fera appel cette fois aux membres de `Liste` pour réaliser les fonctionnalité `Ajouter`, `Enlever` et `Contient`.

A noter que la fonction `f` n'a même pas besoin d'être recompilée à chaque fois qu'une nouvelle implémentation d'`EnsembleAbstrait` lui est fournie en argument : elle est d'ores et déjà capable de traiter n'importe quel ensemble, quel que soit son implémentation sous-jacente.

7 - Exercices

Exercice 15

Factoriser le code commun aux classes `InputFile` et `OutputFile` (exercices 2, 3 et 4 page 81) dans une classe de base commune `File`.

Exercice 16

Concevoir une classe `Pile` d'entiers construite sur les bases de la classe `Liste` (exercices 6 et 7 page 82).

Exercice 17

Concevoir une hiérarchie de formes géométriques. Le sommet de la hiérarchie sera une classe abstraite `Forme`. Chaque forme possède un point d'ancrage et dispose, dans son interface, de fonctions `Tracer`, `Deplacer`, `SelectCouleur` et `Effacer` qui, respectivement, trace et déplace la forme à l'écran, change la couleur pour le prochain tracé, et efface la figure de l'écran. Effacer la figure consiste à la tracer avec la couleur de fond. Déplacer la figure consiste à l'effacer, à changer son point d'ancrage, et à la retracer.

Exercice 18

Concevoir une classe `Dessin` modélisant un ensemble de formes géométriques (voir exercice 17). L'interface de `Dessin` contient des fonctions `Ajouter`, `Tracer` et `Deplacer` qui respectivement ajoutent une nouvelle forme dans le dessin, tracent et déplacent toutes les formes du dessin.

Exercice 19

Implémenter le type `LEnsemble` à partir du type abstrait `EnsembleAbstrait` (voir page 136) et de la structure de données fondamentale `Liste` (exercices 6 et 7 page 82).

Exercice 20

Définir un type de données abstrait `PileAbstraite`.

Implémenter un type pile `TPile` à partir du type abstrait `PileAbstraite` et de la structure de données fondamentale `Tableau` (voir page 136).

Implémenter un type pile `LPile` à partir du type abstrait `PileAbstraite` et de la structure de données fondamentale `Liste` (exercices 6 et 7 page 82).

VI - LES MODELES

Les modèles permettent de décrire des familles génériques : familles de fonctions, familles de types, familles de membres, etc.

Un modèle est paramètré à la manière d'une fonction. Une particularité cependant : les *paramètres-types*. Ce sont des paramètres dont la valeur est un type. Ils sont déclarés à l'aide du mot clé `class`[103]. Les arguments correspondants s'appellent des *arguments-types*.

Les paramètres doivent rester constants : les valeurs des paramètres non types ne peuvent pas être modifiées dans le modèle, et les paramètres-types ne peuvent pas être redéfinis. Mais si le paramètre est un pointeur ou une référence, la valeur référencée est bien sûr modifiable.

Les arguments spécifiés lors de l'instanciation du modèle doivent être des expressions constantes[104], ou des expressions ne faisant intervenir que des adresses d'objets et de fonctions de portée externe.

1 - Modèles de fonctions

i. Définition

Un modèle de fonction se définit de la façon suivante :

```
template <class T>
void swap(T &a, T &b)
{
  T c ;
  c = a ;
  a = b ;
  b = c ;
}
```

[103] Il ne représente cependant pas forcément une classe. Il peut s'agir de tout type : type de base (`int`, `char`, etc.), type pointeur, type tableau, type structure, type fonction.... De ce fait, le choix du mot clé `class` n'est donc pas très heureux, son utilisation dans ce contexte pouvant prêter à confusion. Le choix du mot clé `typename` aurait sans aucun doute été préférable.

[104] Au sens *expression évaluable à la compilation*, c'est-à-dire des expressions composées de littéraux, constantes arithmétiques, valeurs d'énumérations et expressions `sizeof`.

Ici, `swap` est un modèle ayant un paramètre-type `T`. Sa valeur sera déterminée à chaque instanciation.

ii. Instanciation

Le compilateur exploite les modèles définis pour générer des instances particulières de fonctions :

```
int main()
{
  int i1=1, i2=2 ;
  char c1='A', c2='B' ;
  double x1=1.1, x2=2.2 ;

  swap(i1,i2) ;                          // swap(int,int)
  swap(c1,c2) ;                        // swap(char,char)
  swap(x1,x2) ;                  // swap(double,double)
  /* ... */
}
```

Le premier appel à `swap` conduit le compilateur à instancier une fonction `void swap(int,int)` à partir du modèle `swap` pour lequel le paramètre `T` a été instancié par `int`. Pour le second et troisième appel, `T` a été instancié respectivement par `char` et `double`.

Des erreurs peuvent apparaître lors de l'instanciation d'un modèle. Ici par exemple, si les arguments du modèle `swap` ne possèdent pas de constructeur par défaut, l'instanciation échoue, car la variable locale `c` ne peut pas être créée :

```
class Complexe
{
  public :
    Complexe(double,double) ;
} ;

int main()
{
  Complexe c1(1,2), c2(2,-1) ;

  swap(c1,c2) ;                    // erreur de compilation
}
```

De même, l'instanciation du modèle de fonction `min` suivant :

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

avec un type T non acceptable pour l'opérateur < échoue :

```
void main()
{
  int i, j ;
  char c, d ;
  struct S { /* ... */ } s, t ;
  /* ... */
  i = min(j,0) ;                          // int min(int,int)
  c = min(d,'A') ;                      // char min(char,char)
  s = min(s,t) ;                      // erreur de compilation
}
```

Si les arguments du modèle peuvent être déduits à partir de son appel, ils n'ont pas à être spécifiés explicitement à l'instanciation. C'est le cas pour `swap` et `min`. Sinon, ils doivent être énumérés. Dans le modèle suivant :

```
template <class U, class V> U conversion(V) ;
```

il est impossible de déterminer le paramètre U à partir d'un appel à `conversion`. Il est donc nécessaire de le mentionner explicitement :

```
int i ;
float x ;
i = conversion(x) ;  // erreur, impossible de déterminer U
i = conversion<int,float>(x) ;                    // parfait
```

Plus généralement, les derniers arguments d'un modèle peuvent ne pas être spécifiés à l'instanciation s'ils peuvent être déduits à partir de l'appel :

```
i = conversion<int>(x) ;      // ok : conversion<int,float>
```

Ici, le paramètre U est explicitement spécifié, et V est déduit du type de x.

iii. Spécialisation

Il est possible de spécialiser une ou plusieurs instances de modèle, pour des jeux de paramètres donnés :

```
template <class T>                          // modèle générique
T min(T a, T b)
{
  return a<b ? a : b ;
}
                              // spécialisation pour T=char *
char *min(char *a, char *b)
{
  return strcmp(a,b)<0 ? a : b ;
}
```

```
void main()
{
  int i, j ;
  char *s ;
  /* ... */
  i = min(j,0) ;                    // instanciation du modèle
  s = min("strou","strup") ;  // appelle la spécialisation
}
```

Ici, lorsque les arguments de `min` sont de type `char *`, la spécialisation est exploitée, sinon le modèle est instancié.

Le modèle ainsi que toutes ses spécialisations doivent être définis avant la première utilisation.

iv. Résolution des surcharges en présence de modèles

Le mécanisme de recherche d'une fonction surchargée en présence de modèles peut être résumé par les trois règles suivantes :

— si une correspondance exacte est trouvée entre les arguments et les paramètres d'une fonction, cette fonction est appelée ;

— sinon, s'il existe un modèle à partir duquel une fonction réalisant une correspondance exacte entre les arguments et ses paramètres peut être générée, alors cette fonction est générée et appelée ;

— sinon, les règles de résolution ordinaires sont appliquées.

Ce mécanisme est approfondi au chapitre *L'algorithme de résolution des surcharges* page 301.

2 - Les modèles de classes

i. Définition et déclaration

Un modèle de classe est un type paramètré. Il se définit de la façon suivante :

```
template<class A, class B>
class pair
{
  public :
    A a ;
    B b ;
    pair(A aa, B bb) : a(aa), b(bb) {}
    bool operator==(pair p) { return a==p.a && b==p.b ; }
} ;
```

Le modèle `pair` représente une paire de valeurs de n'importe quel type. Une instance est une classe qui s'utilise comme toute autre classe :

```
void main()
{
  pair<int,char> p1(1,'A') ;
  pair<float,bool> p2(2.,false) ;
  /* ... */
}
```

Le modèle `pair` se déclare de la façon suivante :

```
template<class A, class B> class pair ;      // déclaration
```

Un modèle déclaré et non défini peut être utilisé dans les mêmes conditions qu'une classe déclarée et non définie, à savoir dans des situations ne nécessitant pas plus d'informations que celles spécifiées dans la déclaration :

```
pair<float,int> *ptr ;                              // correct
pair<float,int> p1 ;  // illégal, la taille est nécessaire
extern pair<float,int> p2 ;                        // d'accord
```

L'exemple suivant définit un modèle `Test` dont l'objectif est de tester le bon fonctionnement d'une classe `T` quelconque. La classe `T` doit posséder une fonction membre `bool Invariant()` exprimant son invariant[105]. Un objet de type `Test` est construit par-dessus un objet de type `T`. Le premier intercepte les appels aux fonctions du second, et en profite pour vérifier à chaque fois que l'invariant de la classe `T` est respecté :

```
template <class T>
class Test
{
    T *t ;

  public :
    Test(T &tt) : t(&tt) {}
    T *operator->() { assert(t->Invariant()) ; return t ; }
} ;
```

Ainsi, étant donnée la classe :

```
class String
{
    char *str ;
    int len ;

  public :
    String Sub(int,int) ;       // renvoi d'une sous-chaîne
    bool Invariant() { return str!=NULL && len>=0 ; }
  /* ... */
} ;
```

[105] Un invariant de classe est une condition vraie à tout moment, pour tout objet de cette classe.

la classe `Test` peut servir à contrôler l'intégrité d'un objet `String` à chacune de ses utilisations :

```
Test<String> s1(String("Champigny")) ;
String s2 = s1->Sub(1,3) ;
```

Cette dernière instruction contrôle l'invariant de l'objet `String` sous-jacent à `s1`, puis appelle sa fonction `Sub`.

Les modèles de classes sont souvent utilisés pour modéliser des *conteneurs*, c'est-à-dire des objets contenant d'autres objets. Le modèle suivant généralise le type `Tableau` de la page 107 en paramètrant le type des éléments mémorisés :

```
template <class T>
class TableauGenerique
{
    T *tab ;
    int taille ;

    TableauGenerique(const TableauGenerique &) {}
    void operator=(const TableauGenerique &) {}

  public :
    TableauGenerique(int n) { tab = new T[taille=n] ; }
    ~TableauGenerique() { delete[] tab ; }
    T &operator[](int i)
                    { assert(i<taille) ; return tab[i] ; }
    const T &operator[](int i) const
                    { assert(i<taille) ; return tab[i] ; }
} ;
```

`TableauGenerique` s'utilise comme la classe `Tableau` :

```
TableauGenerique<double> tab1(100) ;
tab1[2] = 3.0 ;

TableauGenerique<Complexe> tab2(20) ;
tab2[7] = Complexe(3,1) ;
```

Fonctions membres d'un modèle de classe

Les fonctions membres d'un modèle de classe sont des modèles de fonctions munis des mêmes paramètres que le modèle de classe. Cela est implicite pour toutes les déclarations et les définitions situées dans le modèle. Par contre, cela doit être explicite pour les fonctions définies en dehors. Le modèle précédent peut s'écrire :

```
template <class T>
class TableauGenerique
{
    T *tab ;
    int taille ;

    TableauGenerique(const TableauGenerique &) {}
    void operator=(const TableauGenerique &) {}

  public :
    TableauGenerique(int) ;
    ~TableauGenerique() ;
    T &operator[](int) ;
    const T &operator[](int) const ;
} ;

template <class T>
TableauGenerique<T>::TableauGenerique(int n)
{ tab = new T[taille=n] ; }

template <class T>
TableauGenerique<T>::~TableauGenerique()
{ delete[] tab ; }

template <class T>
T &TableauGenerique<T>::operator[](int i)
                    { assert(i<taille) ; return tab[i] ; }

template <class T>
const T &TableauGenerique<T>::operator[](int i) const
                     { assert(i<taille) ; return tab[i] ;}
```

Paramètres de modèles de classes

Les modèles de classes peuvent avoir des arguments par défaut : les contraintes en ce qui les concerne restent les mêmes que celles imposées pour les arguments par défaut de fonctions[106] :

```
template <class T=int, int taille=100>
class TableauGenerique
{
    T tab[taille] ;

  public :
    T &operator[](int i)
                    { assert(i<taille) ; return tab[i] ; }
```

[106] Voir *Arguments par défaut* page 23.

```
    const T &operator[](int i) const
                    { assert(i<taille) ; return tab[i] ; }
} ;

TableauGenerique<float,10> tab ;                    // d'accord
TableauGenerique<char> tab ; // TableauGenerique<char,100>
TableauGenerique<> tab ;       // TableauGenerique<int,100>
TableauGenerique tab ;     // non, les <> sont obligatoires
```

Qualification des types

En général, la sémantique d'une expression en C++, comme en C, peut dépendre du contexte dans lequel elle se trouve. Par exemple, si `i` est une variable, alors :

```
i*j ;
```

est l'expression de multiplication, alors que si `i` est un type, l'expression est la déclaration d'un pointeur `j`.

Or, lors de la définition d'un modèle, la nature des identificateurs utilisés n'est pas toujours déterminée.

Pour éviter toute ambiguïté dans l'interprétation des expressions, les identificateurs utilisés dans la définition d'un modèle sont supposés ne pas être des types, sauf s'ils ont été déclarés explicitement comme tels, ou s'ils sont qualifiés avec le mot clé `typename` :

```
class C ;

template <class T>
class A
{
  class B ;
  void f() { C *c ;                    // c pointeur vers C
             T *t ;                    // t pointeur vers T
             A *a ;                    // a pointeur vers A
             B *b ;                    // b pointeur vers B

             E *e1 ;           // multiplication de E par e
             typename E *e2 ;         // e2 pointeur vers E

             T::D *d1 ;    // multiplication de T::D par d1
             typename T::D *d2 ;  // d2 pointeur vers T::D

             typedef typename T::D TD ;
             TD *d3 ;              // d3 pointeur vers T::D
           }
} ;
```

Ici, `E` n'a pas été déclaré auparavant. `E` n'est considéré comme un type que lorsqu'il est préfixé de `typename`. C'est la même chose pour `T::D`.

Par contre, `TD` est explicitement défini comme étant égal à `T::D`. `TD` sera ensuite toujours considéré comme un type.

ii. Instanciation

Deux instances de modèles représentent le même type si les modèles ont le même nom et si leurs arguments ont les mêmes valeurs.

```
typedef int T ;
TableauGenerique<int,100> t1 ;
TableauGenerique<T,10*10> t2 ;
```

`t1` et `t2` sont ici de même type. Par contre, `pair<int,char>` et `pair<float,char>` sont deux types distincts. Ainsi dans :

```
template<class A, class B>
class pair
{
  public :
    A a ;
    B b ;
    pair() {}
    pair(A aa, B bb) : a(aa), b(bb) {}
    bool operator==(pair p) { return a==p.a && b==p.b ; }
} ;

void main()
{
  pair<int,char> p1(1,'A') ;
  pair<float,char> p2(1,'A') ;
  if (p1==p2) /* ... */          // erreur de compilation
}
```

la comparaison est invalide, car les paires `p1` et `p2` ne sont pas de même type.

Pour pouvoir réaliser des comparaisons de paires de différents types, l'opérateur `==` peut être défini hors du modèle comme :

```
template<class A, class B, class C, class D>
bool operator==(pair<A,B> p1, pair<C,D> p2)
                    { return p1.a==p2.a && p1.b==p2.b ; }
```

La génération d'une instance, à partir d'un modèle, est réalisée automatiquement par le compilateur en cas de besoin :

```
pair<int,int> p1 ;              // instanciation nécessaire

pair<bool, char *> t2 ;         // instanciation nécessaire
```

```
pair<bool,char>* ptr ;      // instanciation non nécessaire
ptr = new pair<bool,char>(true,'A') ; // nécessaire ici
```

L'utilisation d'un modèle de fonction peut automatiser la détermination des arguments-types pour l'instanciation d'un modèle de classe, puisque les arguments d'un modèle de fonction sont déduits automatiquement lorsque cela est possible. Par exemple, le modèle de fonction :

```
template<class A, class B>
pair<A,B> make_pair(A a, B b) { return pair<A,B>(a,b) ; }
```

permet de créer des paires sans spécifier explicitement le type des deux composantes, ce qui simplifie l'écriture :

```
if ( p==make_pair(1,false) ) /* ... */
```

remplace avantageusement :

```
if ( p==pair<int,bool>(1,false) ) /* ... */
```

L'instanciation d'un modèle de classe peut provoquer une erreur de compilation. C'est le cas, en particulier, si les arguments-types ne vérifient pas les propriétés supposées. Par exemple, dans le modèle `TableauGenerique` de la page 146, l'instruction :

```
tab=new T[taille=n]
```

du constructeur sous-entend que si `T` est une classe, il possède un constructeur par défaut. Une instanciation du type :

```
TableauGenerique<Complexe,20> t ;
```

est donc illégale si la classe `Complexe` n'a pas de constructeur par défaut. D'autres cas peuvent conduire à une erreur d'instanciation :

```
template <class T> class M
{
  struct T t ;
  /* ... */
} ;

class C { /* ... */ } ;
struct S { /* ... */ } ;
union U { /* ... */ } ;

M<int> i ;            // illégal, int n'est pas une struct
M<C> c ;                                    // ça passe...
M<S> s ;                                       // parfait
M<U> u ;        // illégal, une union n'est pas une struct
```

Ici, le fait de précéder `T` de `struct` oblige le paramètre `T` à être soit une structure, soit une classe.

Un argument-type peut être lui-même une instance de modèle :

```
pair< int,pair<char,bool> >
                        p(1,pair<char,bool>('A',true)) ;
```

Ici, `p` est une variable égale à la paire `(1,('A',true))`. Là encore, la fonction `make_pair` permet une écriture plus légère :

```
pair< int,pair<char,bool> > p(1,make_pair('A',true)) ;
```

De même, on peut construire un `TableauGenerique` de `TableauGenerique`, à partir du moment où `TableauGenerique` respecte les contraintes qu'il suppose pour son propre paramètre-type.

```
template <class T, int taille>
class TableauGenerique
{
    T tab[taille] ;

  public :
    T &operator[](int i)
                    { assert(i>=taille) ; return tab[i] ; }
} ;
```

Ici, la déclaration du membre `tab` suppose que le type `T` dispose d'un constructeur par défaut. Or, `TableauGenerique` ayant bien lui-même un constructeur par défaut, on peut donc fabriquer :

```
TableauGenerique<TableauGenerique<int,100>, 100> mat ;
```

un tableau de `100` tableaux. On peut alors écrire des expressions de la forme :

```
mat[i][j] = 2 ;
```

dont l'interprétation peut être décomposée en :

```
(mat[i])[j]
(mat.tab[i])[j]
mat.tab[i].tab[j]
```

iii. Spécialisation

Des spécialisations peuvent être définies pour des jeux de paramètres donnés. Pour le modèle :

```
template<class A, class B> class pair ;
```

peut être définie la spécialisation :

```
class pair<bool,bool>
{
  /* ... */
} ;
```

Des spécialisations partielles peuvent aussi être définies. Par exemple, à partir du modèle :

```
template <class A, class B, int N> class C {} ;
```

les spécialisations suivantes peuvent être définies :

```
template <class A, class B, int N> class C<A,B*,N> {} ;
template <class T, int N> class C<T,T*,N> {} ;
template <class T> class C<T*,int,10> {} ;
```

A l'instanciation, une spécialisation sera préférée au modèle général. Et si plusieurs spécialisations sont candidates, la spécialisation choisie sera la plus spécialisée :

```
C<int,int,1> c ;
```

utilise le modèle, et instancie `A` et `B` à `int`, et `N` à `1`.

```
C<int,char *,1> c ;
```

utilise la première spécialisation, en instanciant `A` à `int`, `B` à `char` et `N` à `1`.

```
C<int,int *,1> c ;
```

utilise la seconde spécialisation, en instanciant `T` à `int`, et `N` à `1`. Enfin :

```
C<char *,int,10> c ;
```

utilise la dernière spécialisation, en instanciant `T` à `char`.

iv. Objets-fonctions

Un paramètre-type peut être instancié par un type fonction. C'est un moyen élégant de passer des fonctions en paramètre, sans déclarer de pointeur. Cela simplifie l'écriture, tout en augmentant la souplesse, la puissance et l'efficacité :

```
template <class Op, class T>
T appliquer(Op op, T x, T y) { return op(x,y) ; }
```

Ici, `Op` est visiblement le type fonction binaire. `appliquer` applique une fonction `op` binaire quelconque à `x` et à `y`. Aucune précision n'est donnée quant au type de retour et types des paramètres de la fonction `op`. Leur détermination sera faite automatiquement à l'instanciation.

Les deux fonctions suivantes :

```
float plus(float x, float y) { return x+y ; }
int fois(int x, int y) { return x*y ; }
```

dont les prototypes sont pourtant différents, peuvent être argument-type du modèle `appliquer` :

```
void main()
{
  float x=1, y=2 ;
  cout << appliquer(plus,x,y) << "\n" ;          // écrit 3
  cout << appliquer(fois,x,y) << "\n" ;          // écrit 2
}
```

Suivant le même principe, l'exemple suivant définit un modèle qui réalise la composition des appels de deux fonctions quelconques `op1` et `op2` :

```
template <class Op1, class Op2, class T>
T composer(Op1 op1, Op2 op2, T x) { return op1(op2(x)) ; }
```

Etant données deux fonctions quelconques :

```
float f(float x) { return  x*x ; }
float g(int x) { return sqrt(x) ; }
```

on peut alors calculer `(f•g)(x)` de la façon suivante :

```
cout << composer(f,g,x) << "\n" ;
```

Ici, cela redonne `x` (quand `x` est positif ou nul), puisque `f` et `g` sont réciproques.

Si on veut créer la fonction composée, c'est-à-dire ici la fonction `f•g`, c'est un peu plus compliqué. On utilise pour cela les *objets-fonctions*.

Un objet-fonction, c'est un objet qui implémente l'opération `()`.

L'idée pour réaliser la composition de fonctions, c'est de créer un objet qui mémorise les fonctions `f` et `g`, et qui compose leur appel à chaque exécution de son opérateur `()`. Cela donne :

```
template <class Op1, class Op2>
class Composition
{
    Op1 op1 ;
    Op2 op2 ;

  public :
    Composition(Op1 o1, Op2 o2) : op1(o1), op2(o2) {}
    float operator() (float x) { return op1(op2(x)) ; }
} ;
```

A partir de là, étant donnés, pour simplifier l'écriture, les deux types :

```
typedef float (*F)(float) ;
typedef float (*G)(int) ;
```

calculer `f•g` puis l'appliquer à `5`, s'écrit de la façon suivante :

```
Composition<F,G> fog(f,g) ;
cout << fog(5) << "\n" ;
```

Pour alléger l'écriture, et pour que les arguments du modèle soient déterminés automatiquement au moment de l'instanciation, on applique la même technique que pour la fonction `make_pair` de la page `149`, en définissant un modèle de fonction :

```
template <class Op1, class Op2>
Composition<Op1,Op2> composer(Op1 o1, Op2 o2)
                  { return Composition<Op1,Op2> (o1,o2) ; }
```

Ce modèle permet d'écrire naturellement :

```
cout << composer(f,g)(x) << "\n" ;
```

pour calculer la valeur de `f•g` en `x`.

Le concept d'objets-fonctions offre des possibilités intéressantes. Il est largement utilisé dans la bibliothèque standard de C++[107].

3 - Exercices

Exercice 21

A partir de la classe `Monnaie` (exercice 10 page 98) concevoir et implémenter une classe générique `Decimal` modélisant l'ensemble des décimaux à `n` décimales, cette classe étant paramètrée par `n` (entier positif).

Exercice 22

Généraliser la classe `Liste` (exercices 6 et 7 page 82) en concevant un modèle de classe `ListeGenerique` modélisant une liste chaînée unidirectionnelle d'objets de type `T` quelconque.

[107] Voir *Les classes-fonctions* page 198.

VII - LA GESTION DES EXCEPTIONS

1 - Introduction

Une certaine souplesse dans la gestion des erreurs d'exécution est indispensable pour réaliser des composants logiciels réutilisables. Cette souplesse passe par la définition d'un protocole entre le concepteur du composant et l'utilisateur.

En effet, l'utilisateur d'une fonction de bibliothèque, qui ne peut pas détecter une erreur provoquée par une instruction de la fonction, peut vouloir la traiter à sa façon, alors que le concepteur de la fonction, qui lui peut détecter une telle erreur, peut ne pas avoir les éléments pour la traiter.

Bien sûr, la solution simple consistant à interrompre l'exécution du programme est toujours envisageable :

```
struct PileInt { int *pile, sommet, taille ; } ;

void Empiler(PileInt &p, int x)
{
  assert(p.sommet+1<p.taille) ;       // interruption brutale
  p.pile[++p.sommet] = x ;
}

void Depiler(PileInt &p, int &x)
{
  assert(p.sommet>=0) ;               // interruption brutale
  x = p.pile[p.sommet--] ;
}
```

mais cette solution est rarement acceptable, puisqu'elle ne laisse aucune chance à l'utilisateur. Elle ne facilite pas non plus la réalisation de nouvelles fonctions réutilisables à partir de ces premières[108].

---

[108] Ceci dit, la macro `assert` reste un excellent moyen pour traquer les erreurs pendant la phase de mise au point.

D'autres solutions, moins radicales, existent : utiliser une variable globale pour signaler le type de l'erreur rencontrée (comme la variable `errno` pour les fonctions mathématiques de la bibliothèque standard C), ou appeler une fonction désignée au préalable par l'utilisateur comme fonction de traitement d'erreur (comme `new_handler`, `terminate` ou `unexpected`). Une autre solution classique, car souvent utilisée dans la bibliothèque C, consiste à renvoyer un code d'erreur :

```
bool Empiler(PileInt &p, int x)
{
  if (p.sommet+1>=p.taille) return false ;
  p.pile[++p.sommet] = x ;
  return true ;
}

bool Depiler(PileInt &p, int &x)
{
  if (p.sommet<0) return false ;
  x = p.pile[p.sommet--] ;
  return true ;
}
```

Cette solution nécessite un contrôle du code de retour à chaque appel de fonction, travail pénible qui a toutes les chances, dans la pratique, de n'être jamais totalement réalisé [FLR]. De plus, il n'est pas dit que l'erreur puisse être corrigée dans l'appelant direct. Le code d'erreur doit dans ce cas être propagé à toutes les fonctions en attente dans la pile des appels, jusqu'à la première pouvant rectifier la situation. Par exemple, pour ajouter une fonction `DepilerNFois` réutilisable qui dépile plusieurs éléments d'un coup :

```
bool DepilerNFois(PileInt &p, int n, int &x)
{
  int i ;
  bool ok ;

  for (i=1 ; i<=n ; i++)
  {
    ok = Depiler(p,x) ;
    if ( !ok )
      return false ;       // propagation du code de retour
  }
  return true ;
}
```

le code de retour de `Depiler` est transmis à l'appelant de `DepilerNFois`.

Un mécanisme de gestion des erreurs d'exécution a été introduit dans le langage lui-même. Cela standardise la technique de report des erreurs, évitant

ainsi au programmeur de s'adapter à une stratégie particulière pour chaque bibliothèque utilisée.

Le principe est le suivant : lorsqu'une fonction veut signaler une erreur, elle lance un objet erreur, et lorsque que l'utilisateur veut gérer l'erreur, il intercepte l'objet lancé. Une erreur non interceptée conduit à un traitement par défaut, qui est l'affichage d'un message d'erreur et l'arrêt de l'exécution.

Cette technique ressemble un peu à celle du renvoi de codes d'erreur, mais elle est supérieure en deux points :

— elle n'accapare pas la valeur de retour des fonctions., et
— elle peut signaler une erreur détectée dans un constructeur ou un destructeur, qui eux n'ont pas de type de retour.

Autres avantages : elle ne nécessite pas de paramètre supplémentaire, et ne fait pas intervenir de variables globales.

Mais surtout, le report d'erreur peut être ignoré en toute sécurité par l'utilisateur : par défaut, toute erreur non traitée provoque la fin de l'exécution.

2 - La structure de contrôle `throw/try/catch`

La structure de contrôle `throw/try/catch` offre au programme la possibilité de passer directement et automatiquement d'un niveau de détection d'erreur à un niveau disposant de toutes les informations permettant un traitement naturel de l'erreur.

Son champ d'action est même plus large que la gestion des erreurs, puisqu'elle peut être utilisée pour traiter toute situation exceptionnelle (comme une fin d'itération, une détection de fin de fichier, etc.)[109]. C'est pour cela qu'elle est appelée structure de contrôle de *gestion des exceptions*. Elle permet de transférer le contrôle du programme, ainsi que certaines informations, du point d'exécution courant jusqu'à un *gestionnaire* d'exception.

Plus précisément, elle prend en charge les trois principaux points suivants :

— la gestion de la remontée du signal de détection d'erreur de la fonction ayant décelé l'erreur vers les fonctions appelantes,
— la détection automatique de ce signal par la première fonction ayant prévu le traitement de ce type d'erreur,
— l'exécution automatique du traitement approprié.

i. Description statique

La mise en œuvre d'une gestion d'erreurs à l'aide de la structure `throw/try/catch` se fait en trois temps :

— signaler la présence de l'erreur (`throw`),

[109] Voir un exemple page 349.

— déclarer l'intention de prendre en charge le traitement des erreurs (`try`),
— définir le traitement de ces erreurs (`catch`).

ou, en termes plus généraux :

— *lancer* l'exception (`throw`),
— indiquer l'intention d'*intercepter* l'exception (`try`),
— définir les gestionnaires d'exceptions (`catch`).

La syntaxe du lancement de l'exception est :

```
throw expression_dont_la_valeur_est_lancée ;
```

La syntaxe de l'intention d'intercepter l'exception ainsi que celle des gestionnaires sont :

```
try
{
  /* ... */
  // instructions susceptibles de provoquer
  // un lancement d'exception
  /* ... */
}
catch(déclaration_paramètre1)
{
  /* ... */
  // instructions de traitement de l'exception
  // correspondant au type du parametre1
  /* ... */
}
catch(déclaration_paramètre2)
{
  /* ... */
  // instructions de traitement de l'exception
  // correspondant au type du parametre2
  /* ... */
}
/* ... */
```

Cette structure est non locale : les instructions `throw` et les blocs `try/catch` peuvent être séparés les uns des autres par un nombre quelconque d'instructions, se trouver dans des fonctions différentes, dans des fichiers sources différents et même dans des unités de compilation différentes.

L'expression `throw`, dont la syntaxe est :

```
throw exp
```

signale une erreur. L'expression *exp* est une information comparable à un code d'erreur. Elle est propagée aux fonctions en attente dans la pile des appels. On dit que cette instruction lance l'exception *exp*. `throw` *exp* est une expression de type `void`.

Le bloc `try`, dont la syntaxe est

```
try { /* ... */ }
```

signifie que les erreurs provoquées par l'exécution de ce bloc doivent être interceptées. Mais une erreur ne sera effectivement saisie que s'il existe un gestionnaire adéquat. On dit que ce bloc indique l'intention d'intercepter les exceptions.

Les exceptions sont traitées par un gestionnaire d'exception dont la syntaxe est :

```
catch (déclaration d'un paramètre)
{
  /* bloc d'instructions */
}
```

ou

```
catch (...)
{
  /* bloc d'instructions */
}
```

Un bloc `try` doit obligatoirement être suivi d'au moins un gestionnaire `catch`. Un gestionnaire `catch` ne peut se trouver qu'après un bloc `try` ou un autre gestionnaire `catch`. Seule l'instruction `throw` peut donner le contrôle à un gestionnaire d'exception. Mais elle ne peut le faire que si elle se trouve dans un bloc `try` ou dans une fonction appelée directement ou indirectement à partir d'un bloc `try`.

Le type du paramètre d'un gestionnaire est appelé *type du gestionnaire*.

ii. Description dynamique

A la rencontre d'une instruction `throw`, le contrôle est passé au gestionnaire *le plus près* (c'est-à-dire dont le bloc `try` a été le plus récemment exécuté par le flux de contrôle et dont l'exécution n'est pas encore terminée[110]) dont le type est *compatible*[111] avec le type de l'expression lancée par `throw`, sachant que pour un bloc `try` donné, les gestionnaires sont essayés séquentiellement suivant leur ordre de définition.

A l'entrée du gestionnaire, l'exception est supposée traitée. Lorsque l'exécution de ce gestionnaire se termine, le contrôle est passé à l'instruction située après le groupe de gestionnaires contenant le dernier gestionnaire exécuté. Les instructions non exécutées dans la fonction qui a lancé l'exception, dans toutes les fonctions rencontrées lors de la remontée de la pile des appels, et enfin dans le

[110] Il est trouvé en remontant la pile des appels.

[111] Voir *Sélection du gestionnaire* page 162.

bloc `try` de la fonction qui a intercepté l'exception, sont abandonnées. Les objets automatiques[112] créés dans ces portions de code sont détruits : les destructeurs de ces objets sont appelés.

La pile précédente peut gérer ses erreurs avec la structure `throw/try/catch` de la façon suivante :

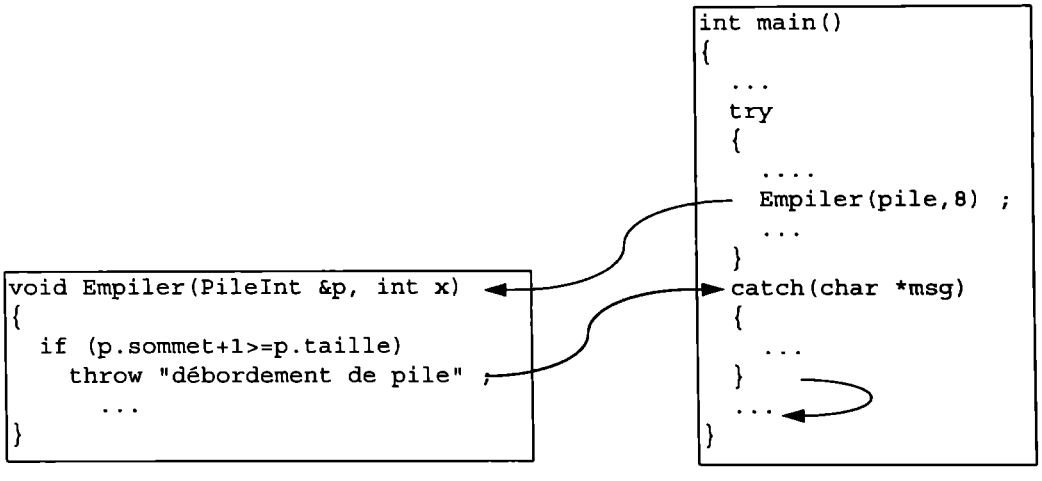
```
struct PileInt { int *pile, sommet, taille ; } ;

void Empiler(PileInt &p, int x)
{
  if (p.sommet+1>=p.taille) throw "débordement de pile" ;
  p.pile[++p.sommet] = x ;
}

void Depiler(PileInt &p, int &x)
{
  if (p.sommet<0) throw "pile vide" ;
  x = p.pile[p.sommet--] ;
}

void DepilerNFois(PileInt &p, int n, int &x)
{
  try
  {
    for (int i=1 ; i<=n ; i++) Depiler(p,x) ;
  }
  catch(char *msg)
  {
    cout << "fonction DepilerNFois : " << msg << "\n" ;
    exit(1) ;
  }
}
```

Les fonctions `Empiler` et `Depiler` lancent une exception en cas de détection d'erreur. La fonction `DepilerNFois` intercepte les exceptions pouvant être lancées par `Depiler` :

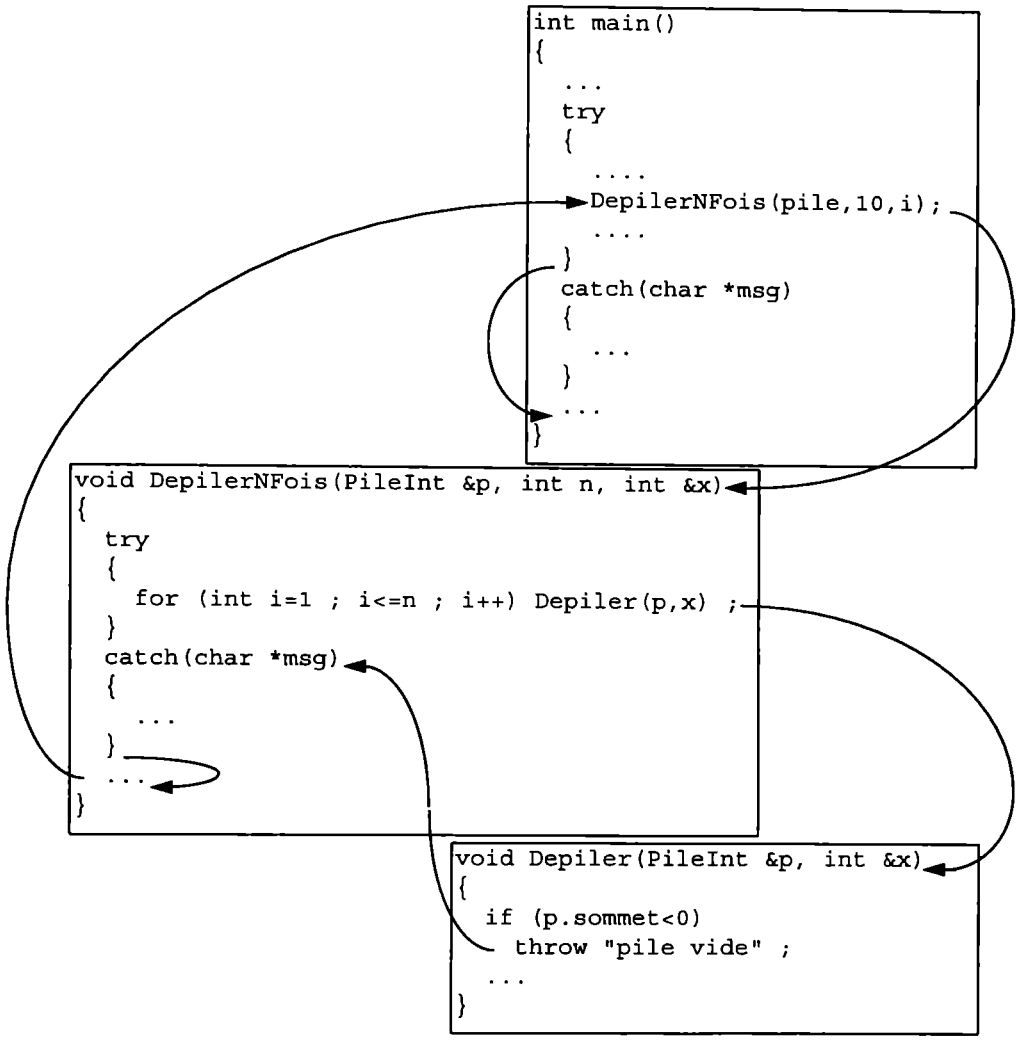
[112] C'est-à-dire les objets dont la classe d'allocation est `auto` : c'est la classe d'allocation par défaut pour les objets locaux et les paramètres de fonction, dont la durée de vie est fonction du code exécuté.

```
int main()
{
  PileInt pile ;
  int i, ret=0 ;

  Empiler(pile,1) ;
  /* ... */
  try
  {
    /* ... */
    Empiler(pile,8) ;
    /* ... */
    Depiler(pile,i) ;
    /* ... */
    DepilerNFois(pile,10,i) ;
    /* ... */
  }
  catch(char *msg)
  {
    cout << msg << "\n" ;
    ret = 1 ;
  }
  return ret ;
}
```

Ici, les erreurs provoquées par les appels `Empiler(pile,8)` et `Depiler(pile,i)` du bloc `try` de la fonction `main` seront traitées par le gestionnaire `catch(char *)` de `main`.



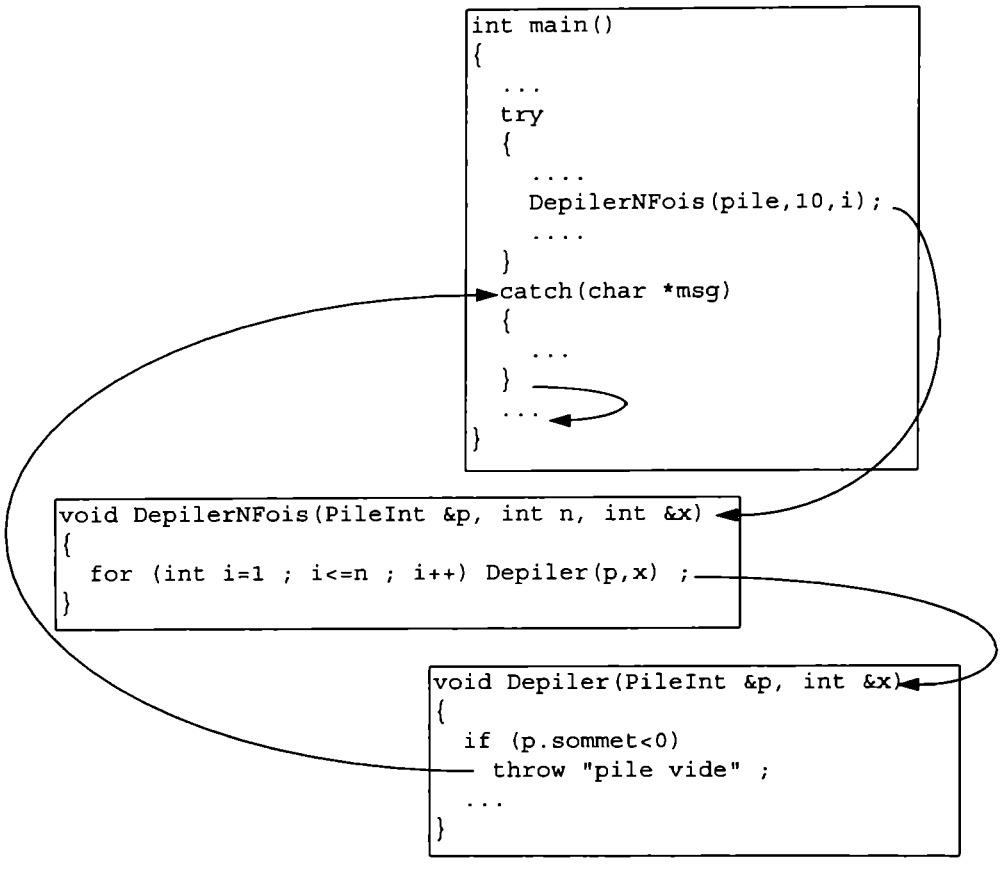
Par contre, les erreurs provoquées par l'appel `DepilerNFois` du bloc `try` de la fonction `main` seront interceptées et traitées par le gestionnaire `catch` de `DepilerNFois`.



Si la fonction `DepilerNFois` doit être réutilisable, il est alors plus judicieux qu'elle ne traite pas l'exception. Son expression se réduit alors à :

```
void DepilerNFois(PileInt &p, int n, int &x)
{
  for (int i=1 ; i<=n ; i++) Depiler(p,x) ;
}
```

Dans ce cas, les erreurs provoquées par l'appel `DepilerNFois(pile,10,i)` du bloc `try` de la fonction `main` seront traitées par le gestionnaire `catch(char *)` de `main`. En effet, le bloc `try` de `main` est le premier bloc `try` actif rencontré en remontant la pile des appels à partir de l'appel `Depiler(p,x)` disposant d'un gestionnaire `catch(char *)`.



iii. Sélection du gestionnaire

Comme mentionné précédemment, le gestionnaire sélectionné est le premier gestionnaire rencontré en remontant la pile des appels de fonctions possédant un type compatible avec celui de l'expression `throw`.

Un gestionnaire de type `T1`, `const T1`, `T1&` ou `const T1&` est compatible avec une expression `throw` de type `T2` si l'une des conditions suivantes est vérifiée :

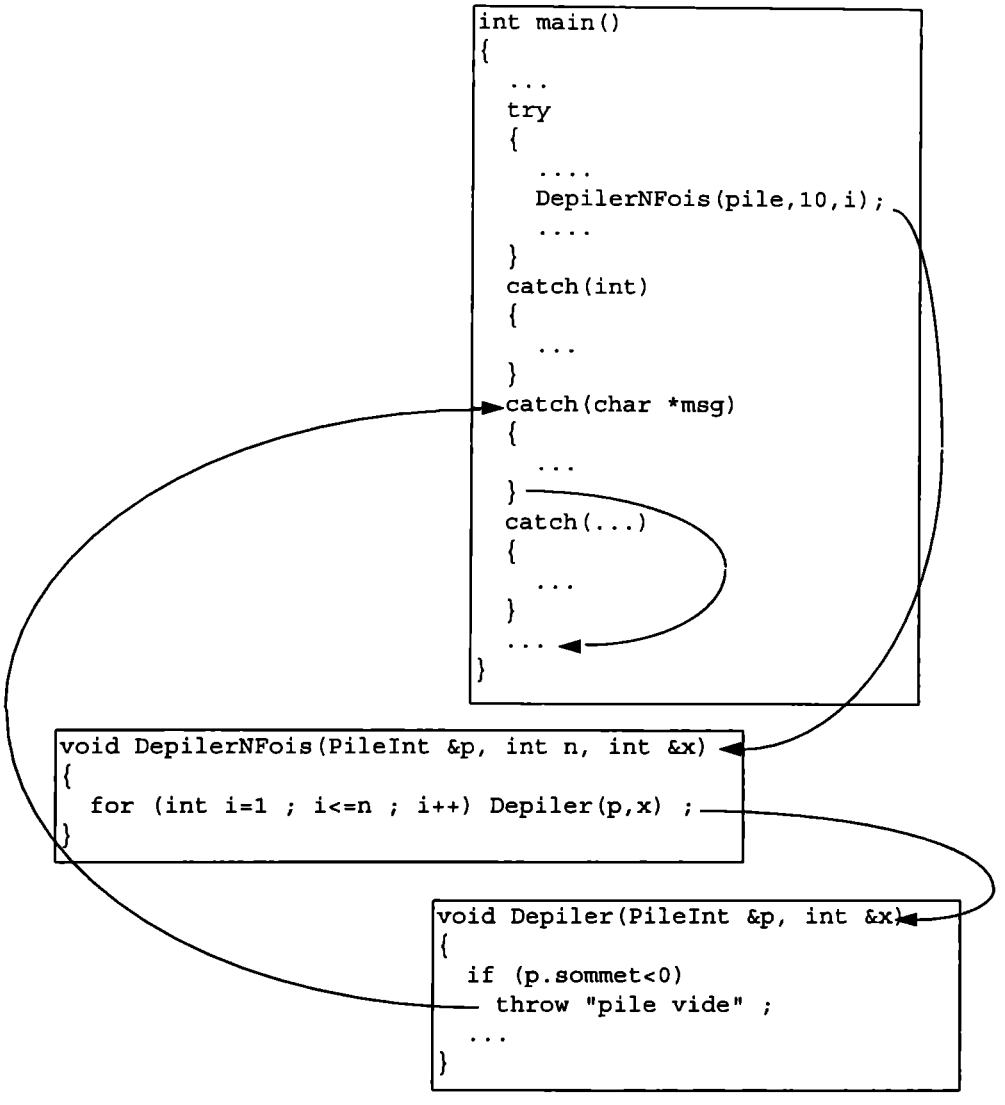
— `T1` et `T2` sont égaux, ou
— `T1` est une classe de base accessible de `T2`, ou
— `T1` et `T2` sont des pointeurs et `T2` peut être converti en `T1` par une conversion standard.

A noter donc que seules les conversions standards sur les pointeurs sont mises en oeuvre pour tenter de trouver une correspondance. Les conversions standards non relatives aux pointeurs ne sont pas tentées.

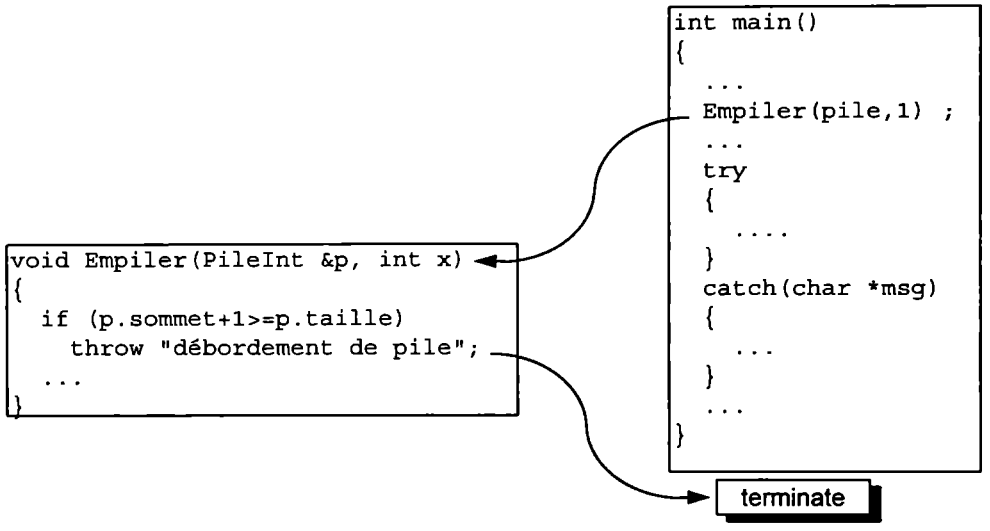
Le gestionnaire `catch(...)` intercepte tout type d'exception. Ainsi, puisque les gestionnaires sont essayés séquentiellement, `catch(...)` ne peut se trouver

qu'en dernière position (car si tel n'était pas le cas, les gestionnaires placés derrière lui n'intercepteraient jamais aucune exception) :

```
int main()
{
    /* ... */
  try
  {
    /* ... */
    DepilerNFois(pile,10,i) ;
    /* ... */
  }
  catch(int) { /* ... */ }
  catch(char *msg) { /* ... */ }
  catch(...) { /* ... */ }
  return 0 ;
}
```



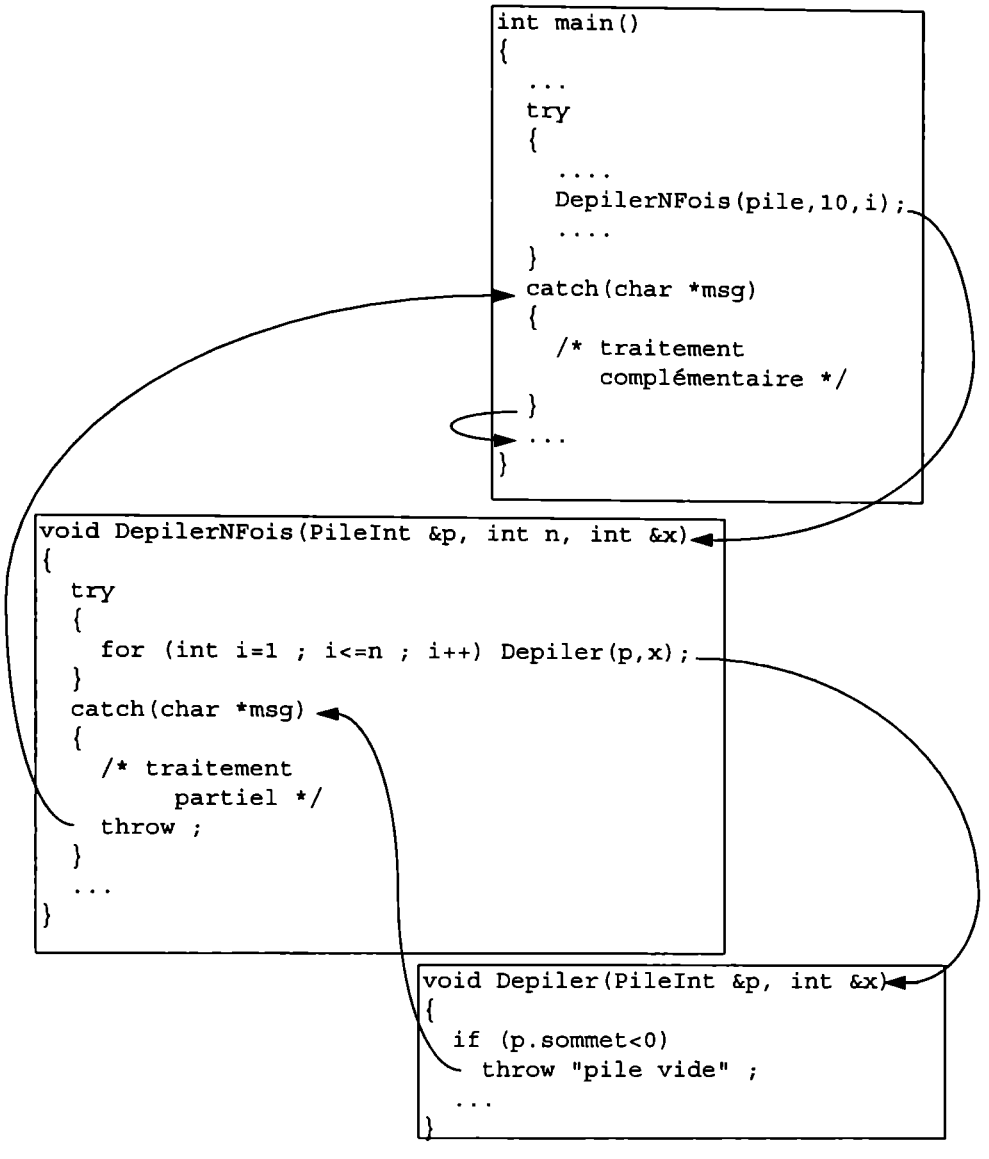
Si aucun gestionnaire adéquat n'est trouvé, c'est la fonction prédéfinie `terminate` qui est appelée. `terminate` est appelée aussi lorsqu'une exception est lancée hors d'un bloc `try`. Cette fonction, par défaut, appelle la fonction de la bibliothèque standard C `abort`, qui interrompt l'exécution du programme. Par exemple, l'appel `Empiler(pile,1)` de `main`, hors du bloc `try`, exécute `terminate` en cas d'erreur.



L'instruction `throw;` dans un gestionnaire signifie que l'exception reçue doit continuer sa remontée à travers la pile des appels. L'erreur peut ainsi être traitée partiellement, et relancée pour être traitée complètement à un niveau supérieur :

```
void DepilerNFois(PileInt &p, int &n, int &x)
{
  int i ;
  try
  {
    for (i=1 ; i<=n ; i++) Depiler(p,x) ;
  }
  catch(char *msg)
  {
    n = i-1 ;
    throw ;
  }
}
```

Ici, lorsque `Depiler` lance une exception, la valeur de `n` est ramenée à la plus grande valeur admissible, et l'exception est relancée :



`throw;` ne peut être utilisé que dans un gestionnaire d'exception ou dans une fonction appelée directement ou indirectement à partir d'un gestionnaire.

3 - Spécification des exceptions

Une fonction peut spécifier dans son en-tête et son prototype (les deux à la fois) les exceptions qu'elle est en mesure de lancer à son appelant, directement par `throw` ou indirectement par propagation :

```
void Depiler(PileInt &p, int &x) throw(char *)
{
  if (p.sommet<0) throw "pile vide" ;
  x = p.pile[p.sommet--] ;
}
```

`throw(char *)` en fin d'en-tête signifie que seules les exceptions `char *` peuvent être lancées par la fonction `Depiler`. De même :

```
struct PbMemoire {} ;

struct PbTaille {} ;

void Creer(PileInt &p, int t) throw(PbMemoire,PbTaille)
{
  if (t<0) throw PbTaille() ;
  if (t>MAX_TAILLE) throw PbMemoire() ;
  p.pile = new int [p.taille=t] ;
  p.sommet = -1 ;
}
```

signifie que seules les exceptions `PbMemoire` et `PbTaille` peuvent être lancées par la fonction `Creer`.

Plus généralement,

```
void f() throw(X) ;
```

signifie que la fonction `f` ne peut lancer que les exceptions de type `X` ou d'un type ayant `X` comme classe de base accessible. De même,

```
void f() throw(X*) ;
```

signifie que `f` ne peut lancer que les exceptions de type `X *` ou de type pointeur vers un type ayant `X` comme classe de base accessible[113] :

```
struct PbPile {} ;
struct PbDepiler : public PbPile {} ;

void Depiler(PileInt &p, int &x) throw(PbPile)
{
  /* ... */
  throw PbDepiler() ;       // d'accord, puisque PbDepiler
                    // est dérivée publiquement de PbPile
  /* ... */
}
```

Ici, le lancement de `PbDepiler` est autorisé, car `PbDepiler` est dérivée publiquement de `PbPile`.

Lorsque rien n'est mentionné, la fonction peut lancer n'importe quelle exception. Par contre, si `throw()` est mentionné, cela veut dire qu'aucune exception ne peut être lancée :

```
void Depiler(PileInt &p, int &x) throw(PbDepiler) ;
```

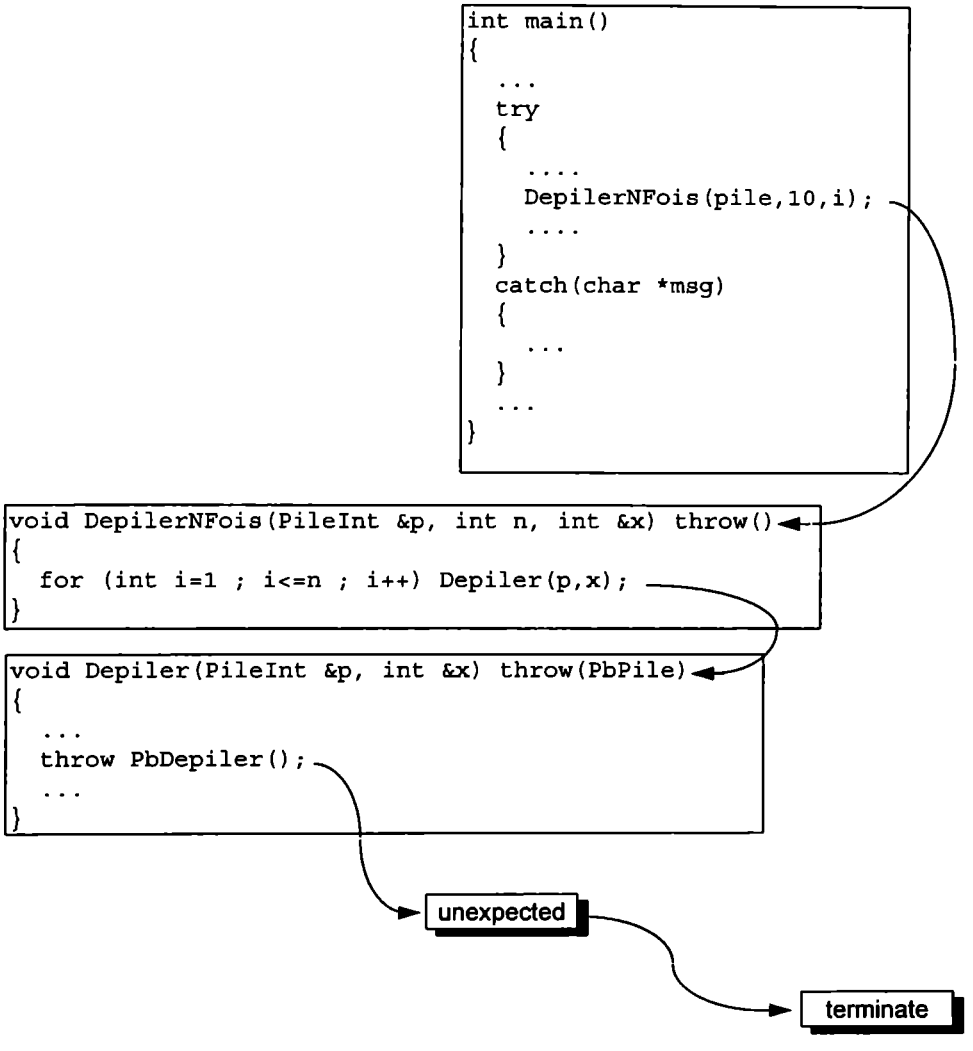
[113] Et même chose pour les références.

```
void DepilerNFois(PileInt &p, int n, int &x) throw()
{
  int i ;
  try { for (i=1 ; i<=n ; i++) Depiler(p,x) ; }
  catch(PbDepiler) {  /* ... */ }
}
```

Ici, pas de problème, l'erreur `PbDepiler` pouvant venir de `Depiler` est interceptée, donc `DepilerNFois` ne la transmettra pas. Par contre :

```
void DepilerNFois(PileInt &p, int &n, int &x) throw()
{
  for (int i=1 ; i<=n ; i++) Depiler(p,x) ;
}
```

va poser problème, puisque `Depiler` peut signaler une erreur et que celle-ci n'est pas interceptée dans la fonction. Lorsqu'un tel cas se présentera, la fonction prédéfinie `unexpected` sera appelée. Cette fonction, par défaut, appelle la fonction `terminate`.



Cette extension de la syntaxe des prototypes et en-têtes de fonctions apporte une normalisation dans l'écriture d'une information intéressante pour l'utilisateur (les prototypes sont toujours à la disposition des utilisateurs) qui, sinon, aurait tout intérêt à être portée en commentaire dans le programme. De plus, cela permet au compilateur de contrôler l'exactitude de cette information.

La spécification des exceptions ne fait pas partie du type de la fonction :

```
void f() ;
void g() throw() ;

void main()
{
  void (*pf1)() ;
  void (*pf2)() throw() ;
  pf1 = f ;                                    // oui
  pf1 = g ;                                    // oui
  pf2 = f ;                                    // oui
  pf2 = g ;                                    // oui
}
```

`f` et `g` sont donc de même type.

4 - `terminate` et `unexpected`

`terminate` et `unexpected` sont des pointeurs de fonctions.

Les types `terminate_handler` et `unexpected_handler`, définis dans `<exception>` comme :

```
typedef void (*terminate_handler)() ;
typedef void (*unexpected_handler)() ;
```

sont les types *fonction sans paramètre et ne renvoyant rien.*

Les fonctions prédéfinies `set_terminate` et `set_unexpected`, déclarées dans `<exception>` comme :

```
terminate_handler set_terminate(terminate_handler) ;
unexpected_handler set_unexpected(unexpected_handler) ;
```

permettent de changer respectivement les fonctions de terminaison `terminate` et `unexpected`. Les fonctions `set_terminate` et `set_unexpected` renvoient l'adresse des fonctions de terminaison actives juste avant ce changement :

```
void FinDeRemontee()
{
  cerr << "erreur non interceptée !" ;
  exit(255) ;
}
```

```
int f()
{
  terminate_handler old_terminate
                          = set_terminate(FinDeRemontee) ;
  /* la fonction de terminaison est ici FinDeRemontee */
  /* ... */
  set_terminate(old_terminate) ;          // restauration
}
```

`terminate` doit terminer l'exécution du programme : elle ne peut en aucun cas ni retourner à son appelant, ni lancer une exception.

`unexpected` ne doit pas retourner à son appelant, mais peut lancer une exception.

VIII - LES ESPACES DE NOMS

Utiliser plusieurs bibliothèques de fonctions qui n'ont pas été conçues pour fonctionner ensemble est souvent délicat. Des problèmes se posent dès que ces bibliothèques utilisent des identificateurs globaux identiques.

Les *espaces de noms* permettent de régler ce type de problème. Ils mettent en œuvre un mécanisme simple, qui permet :

— de lier (*linker*) plusieurs bibliothèques en évitant toute collision d'identificateurs,
— de sélectionner un identificateur donné, lorsque plusieurs identificateurs portent le même nom,
— de résoudre les collisions de noms sans avoir à modifier les instructions qui les provoquent, simplement en ajoutant des déclarations et des directives.

Les concepteurs de bibliothèques peuvent alors choisir sans contraintes leurs identificateurs globaux.

1 - Portée et visibilité

Une *région déclarative* est une portion de programme contiguë représentant l'étendue la plus grande dans laquelle les identificateurs qui y sont déclarés sont susceptibles d'être valides. Un bloc { }, une classe, une union sont des régions déclaratives.

Un identificateur est masqué par un identificateur de même nom déclaré :

— dans une région déclarative imbriquée, ou
— dans une classe dérivée.

La portée d'un identificateur est la partie du programme dans laquelle il est possible d'accéder à l'objet représenté par l'identificateur.

L'opérateur de résolution de portée `::` permet de référencer un objet par nommage explicite de sa portée sous la forme :

```
portée::identificateur
```

La notation :

```
::identificateur
```

fait référence à un identificateur global.

La visibilité d'un identificateur est la partie du programme dans laquelle il est possible d'accéder à l'identificateur.

La plupart du temps, portée et visibilité coïncident, sauf en cas de masquage de l'identificateur : dans ce cas, sa visibilité est incluse dans sa portée. L'opérateur `::` permet d'accéder aux identificateurs masqués :

```
int i ;                                          // de portée globale

void f()
{
  char c ;
  {
    // on est dans la portée de int i,
    // et int i est visible

    // on est dans la portée de char c,
    // et char c est visible

    float i ;

    // on est toujours dans la portée de int i,
    // mais int i est masqué

    i = i+1 ;                                          // float i
    ::i++ ;                                              // int i
    c = 'A' ;                                           // char c
    ::c = 'A' ;    // erreur : il n'existe pas de c global
  }

  // on est toujours dans la portée de int i,
  // et int i redevient visible

  i++ ;                                                  // int i
  c-- ;                                                 // char c
}

  // on est toujours dans la portée de int i,
// on est plus dans la portée de char c
```

Préfixer le nom d'une classe, d'une structure, d'une union ou d'une énumération permet parfois également d'y accéder lorsque son nom est masqué :

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

```
void main()
{
  int C ;

  C c ;                              // non, C n'est pas un type
  class C c1 ;                                              // ok
  ::C c2 ;                                                  // ok
}
```

2 - Espace de noms

Un espace de noms est une région déclarative dont le nom peut être utilisé pour accéder aux identificateurs qui y sont déclarés :

```
namespace N
{
  int i, j ;
  void f() { /* ... */ }
  class C { /* ... */ } ;
  /* ... */
}
```

Les identificateurs déclarés dans un espace sont appelés membres de l'espace. Ils sont visibles dans les régions déclaratives imbriquées tant qu'ils ne sont pas masqués :

```
int k ;

namespace N
{
  int i, j ;

  void f()
  {
    i = 0 ;                                              // N::i = 0
    j = 0 ;                                              // N::j = 0
  }

  class C
  {
      int j, l ;

    public :
```

```
        C()
        {
          j = i ;                                   // C::j=N::i
          l = k ;                                    // C::l=::k
        }
    } ;
}
```

Un identificateur déclaré hors de tout espace nommé, de tout bloc et de toute classe est dans l'espace global.

Les espaces de noms peuvent être imbriqués :

```
namespace N
{
  int i, j ;
  void f() { /* ... */ }
  class C { /* ... */ } ;

  namespace M
  {
    void g() ;
  }
}
```

Une qualification de portée explicite doit être utilisée pour accéder à un identificateur qui n'est ni dans la région déclarative courante, ni dans une région déclarative englobante :

```
namespace N
{
  int i, j ;
  void f() { /* ... */ }
  class C { /* ... */ } ;

  namespace M
  {
    void g() { f() ; }                                  // oui
  }
}

namespace O
{
  int j ;
```

```
  void  h()
  {
    j = 0 ;                                          // oui
    N::i = 2 ;                                       // bien
    f() ;                  // erreur, pas de f dans la région
                          // déclarative courante, ni dans une
                             // région déclarative englobante
    N::f() ;                                         // oui
  }
}

N::C c ;                                             // oui

void main()
{
  N::i = 0 ;                                         // oui
  N::f() ;                                           // oui
  N::g() ;                    // erreur, pas de N::g dans la
                              // région déclarative courante
  N::M::g() ;                                        // bien
}
```

Si la portée qualifiée ne commence pas par `::`, la résolution se fait en remontant, à partir de la région déclarative courante, les régions déclaratives englobantes. Sinon la résolution se fait à partir de la région déclarative globale :

```
namespace N
{
  int i ;

  namespace M
  {
    int j ;

    namespace N
    {
      int k ;

      void f()
      {
        cout << k ;                                  // oui
        cout << N::k ;                               // oui
        cout << M::N::k ;                            // oui
        cout << N::M::N::k ;                         // non
        cout << ::N::M::N::k ;                       // oui
```

```
            cout << j ;                                    // oui
            cout << N::j ;                                 // non
            cout << M::j ;                                 // oui
            cout << N::M::j ;                              // non
            cout << ::N::M::j ;                            // oui

            cout << i ;                                    // oui
            cout << N::i ;                                 // non
            cout << ::N::i ;                               // oui
        }
    }
  }
}
```

Les membres d'un espace peuvent être définis hors de l'espace, à l'aide d'une qualification de portée explicite :

```
namespace N
{
  int i, j ;
  void f() ;
  class C ;
}

void N::f() { /* ... */ }

class N::C
{
 /* ... */
} ;
```

Un membre n'est pas obligatoirement défini en global, mais doit l'être dans un espace englobant celui contenant sa déclaration :

```
namespace N
{
  int h() ;
  namespace M
  {
    void f() ;
    void g() ;
  }
  void M::f() { /* ... */ }                          // bien
}

void N::M::g() { /* ... */ }                         // bien

namespace O
{
  int N::h() { /* ... */ }     // erreur, O n'englobe pas N
}
```

Ce qui différentie principalement les espaces de noms des autres régions déclaratives (blocs, structures, classes), c'est leur possibilité d'être définis incrémentalement. Dans ce cas, l'espace est la concaténation de toutes les définitions partielles portant le même nom, et définie dans la même région déclarative :

```
namespace N
{
  int i, j ;
  void f() ;

  int g()
  {
    int i ;
    i = 0 ;
    ::N::i = 0 ;
    j = 0 ;
    k = 0 ;
    f() ;
  }
}
/* ... */

namespace N
{
  int i ;                          // erreur : redéfinition de i
  int k ;

  void f()     // ok, définition de f précédemment déclarée
  {
    j = 0 ;
    k = 0 ;
  }

  int g()                  // erreur : multiple définition de g
  { /* ... */ }
}
```

Ici, la variable `i` est définie deux fois dans la même région déclarative, ce qui est interdit. Et de même pour la fonction `g`. Par contre, pas de problème pour la fonction `f` déclarée dans la première partie de `N`, et définie dans la seconde.

Un espace ne peut être défini qu'au niveau global, ou qu'immédiatement imbriqué dans un autre espace.

3 - Déclaration `using`

La déclaration `using`, dont la syntaxe est :

```
using région::identificateur ;
```

introduit l'identificateur qualifié dans la région déclarative courante :

```
namespace N
{
  int i, j ;
  void f() ;
  class C { /* ... */ } ;
  /* ... */
}

using N::C ;     // introduit N::C dans le namespace global

C c ;                                    // c'est-à-dire N::C

void h()
{
  using N::f ;                       // introduit N::f dans h
  f() ;                                 // c'est-à-dire N::f

  i = 0 ;                                                // erreur
  using N::i ;
  i = 0 ;                                                   // oui
}
```

Comme lors d'une déclaration ordinaire, un identificateur introduit par une déclaration `using` masque les identificateurs de même nom déclarés dans les régions déclaratives englobantes :

```
namespace N
{
  int k ;
}

int k ;

void f()
{
  using N::k ;                              // N::k masque ::k

  k++ ;                                                  // N::k++
}
```

Ici, l'introduction de `N::k` dans la région déclarative courante provoque le masquage de tout identificateur `k` défini dans une région déclarative englobante.

L'introduction d'un nom de fonction introduit d'un coup toutes les surcharges de la fonction :

```
namespace N
{
  void f(int) ;
  void f(char) ;
}

void g()
{
  using N::f ;
  f(1) ;                                          // N::f(int)
  f('a') ;                                        // N::f(char)
}
```

Un identificateur ne peut être introduit par une directive `using` que s'il n'est pas déjà défini dans la région déclarative courante :

```
void f()
{
  int i=0 ;
  using N::i ;                    // erreur, redéfinition de i
  /* ... */
}
```

Les identificateurs introduits sont ceux connus au moment de la déclaration `using`. Les identificateurs déclarés après ne sont pas introduits :

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

using N::f ;

namespace N
{
  void f(char) ;
}

void g()
{
  f(1) ;                                          // N::f(int)
  f('a') ;                                        // N::f(int)
}
```

`f('a')` appelle donc `f(int)`, puisque `f(char)` n'a pas été introduite par la déclaration `using`.

4 - La directive `using`

La directive `using`, dont la syntaxe est :

`using namespace` *`nom_namespace`* `;`

introduit tous les identificateurs de l'espace dans la région déclarative courante. Ainsi, après cette directive, tous les identificateurs déclarés dans l'espace peuvent être utilisés sans qualification de portée :

```
namespace N
{
  int i ;
  void f() { /* ... */ }
}

using namespace N ;                    // introduit tous les
                                       // identificateurs de N
                          // dans la région déclarative globale

void main()
{
  i++ ;                                                  // N::i
  f() ;                                                 // N::f()
}
```

Si l'espace introduit par la directive `using` contient un identificateur déjà déclaré dans la région déclarative courante, alors l'utilisation de cet identificateur est ambiguë et illégale :

```
int j ;

namespace M
{
  int i, j ;
}

namespace N
{
  int i ;
  using namespace M ;        // introduction d'un autre i...
}

void f()
{
  N::i = 0 ;                                      // erreur ici...
                          // ... c'est ambigu : M::i ou N::i ?
```

```
    using namespace M ;

    j = 0 ;          // erreur : ambiguité entre ::j et M::j
    M::j = 0 ;                                          // ok
    ::j = 0 ;                                           // ok
}
```

La directive `using` est transitive : les identificateurs de `N1` introduits dans `N2` à l'aide d'une directive `using`, sont introduits par toute directive `using namespace N2` :

```
namespace N1
{
  int i ;
}

namespace N2
{
  using namespace N1 ;
  int j ;
  void f()
  {
    i = 0 ;                                             // N1::i
  }
}

namespace N3
{
  using namespace N2 ;
  void g()
  {
    i = 0 ;                                             // N1::i
    j = 1 ;                                             // N2::j
  }
}
```

Dans la pratique, les espaces peuvent être utilisés pour composer une interface à partir de noms de provenances multiples :

```
namespace MonEspace
{
  using namespace bibli1 ;
  using bibli2::String ;
  using namespace bibli3 ;
  typedef long CoOrd ;
  /* ... */
}
```

Ici, la définition du type `CoOrd` permet de lever l'ambiguïté lors de l'utilisation de `MonEspace`, si `CoOrd` existe dans `bibli1` et `bibli2`.

5 - Alias

L'instruction :

```
namespace alias = namespace_original ;
```

définit un alias pour l'espace original :

```
namespace N
{
  int i ;
  /* ... */
}

namespace M = N ;              // définit M comme alias de N
```

M est ici un alias pour N. L'alias s'utilise comme un espace ordinaire :

```
void f()
{
  M::i = 0 ;                        // équivalent à : N::i=0
  /* ... */
}

using namespace M ;     // équivalent à : using namespace N

namespace O = M ;  // définit O comme alias de M donc de N
```

Les alias permettent aux développeurs de bibliothèques d'utiliser des noms d'espace long, évitant ainsi toute collision avec d'autres noms d'espace dans d'autres bibliothèques du marché, sans contrainte supplémentaire pour l'utilisateur, qui peut les raccourcir à son gré :

```
namespace ObjectsWindowsLibrairies
{
  void main() ;
  /* ... */
}

namespace OWL = ObjectsWindowsLibrairies ;

void main()
{
  OWL::main() ;
}
```

Ils permettent également de s'affranchir du nom exact de la bibliothèque utilisée :

```
namespace window =
#if IHM==PM
  OWL
#else
  MFC
#endif
;

void main()
{
  window::main() ;
}
```

6 - Espace anonyme

Un espace anonyme est un espace défini de la façon suivante :

```
namespace
{
  int nb_erreur ;
  void erreur() { /* ... */ }
}
```

Aucun nom n'est spécifié. Cependant, le compilateur lui en attribue un quelconque, et ce nom est différent pour chaque unité de compilation (dans le cas par exemple où cet espace est dans un fichier en-tête inclus plusieurs fois). De plus, une directive `using` le concernant est insérée implicitement juste après sa définition. L'espace ci-dessus est interprété comme :

```
namespace unite_i
{
  int nb_erreur ;
  void erreur() { /* ... */ }
}

using namespace unite_i ;
```

avec *unite_i* spécifique à chaque unité de compilation.

Ainsi, même si les identificateurs d'un espace anonyme ont une portée externe, ils ne peuvent en aucun cas être utilisés à partir d'une autre unité de compilation (puisqu'ils sont qualifiés à chaque fois par un nom inconnu dépendant de l'unité de compilation courante).

Les espaces anonymes permettent ainsi de définir des identificateurs d'utilisation limitée à l'unité de compilation, au même titre que le mot clé `static` :

```
namespace
{
  int nb_erreur ;
  void erreur() { /* ... */ }
}
```

est une autre façon d'écrire :

```
static int nb_erreur ;
static void erreur() { /* ... */ }
```

Pour déclarer des objets locaux à l'unité de compilation, l'utilisation des espaces anonymes est préférable à celle du mot clé `static` chargé déjà plusieurs sémantiques (et dont l'utilisation dans ce contexte est vouée à devenir obsolète). `static` peut ainsi être réservé à la déclaration des membres de classe[114].

[114] Voir *Membres statiques* page 59.

— Seconde partie —

Composants prédéfinis

Une bibliothèque standard est associée au langage C++. Les composants de cette bibliothèque (fonctions, types, modèles, constantes...) peuvent coexister avec ceux de la bibliothèque C, qui reste donc utilisable par les programmes C++.

Cette bibliothèque est divisée en catégories. Une catégorie est un regroupement logique de composants. On trouve les catégories suivantes :

— support du langage
— diagnostic
— utilitaires généraux
— chaînes
— internationalisation
— conteneurs
— itérateurs
— algorithmes
— numériques
— entrées/sorties

Les composants de la bibliothèque sont membres d'un espace de noms appelé `std`. Si leur déclaration est livrée sous forme de fichiers en-tête, ce qui est le cas la plupart du temps, le nom de ces fichiers en-tête dépend de l'implémentation.

L'étude complète de cette bibliothèque ne fait pas partie des objectifs de ce livre : la description complète de la bibliothèque représenterait un volume d'information supérieur à celui correspondant à la référence du langage. Le but de cette partie est plus modestement de donner une vue d'ensemble du contenu de la bibliothèque et une idée de la philosophie générale. Seuls sont approfondis les éléments qu'il est indispensable de connaître, à savoir :

— la classe `type_info`, puisqu'elle est utilisée par l'opérateur `typeid`,
— la hiérarchie des classes d'exception, puisque certaines sont lancées par le système (`bad_cast`, `bad_alloc`, `bad_typeid`, etc.),
— la classe `string`, puisqu'elle est utilisée par les classes exceptions,
— une partie des classes `stream`.

Cette bibliothèque est dans son ensemble relativement complexe. Les définitions sont parfois lourdes et difficiles à décoder. Cela est dû en partie à l'utilisation massive des modèles. Afin de les rendre plus abordables, certaines définitions ont été un peu simplifiées. Par conséquent, ce qui suit ne doit pas être pris pour un guide de référence exact.

I - SUPPORT DU LANGAGE

Cette catégorie concerne d'une part les fonctions appelées implicitement pendant l'exécution des programmes, et d'autre part le type des objets créés implicitement par le système. On y trouve aussi la définition des types utilisés un peu partout dans la bibliothèque, les constantes numériques caractéristiques de l'implémentation (comme les limites des types prédéfinis), les fonctions relatives à l'initialisation et à la terminaison des programmes, les outils liés à la gestion dynamique de la mémoire, à l'identification dynamique de type...

C'est ici qu'est définie la classe `type_info`.

La classe `type_info`

Cette classe décrit les informations d'identification de type générées par le compilateur. Elle est déclarée dans `<type_info>`. Cette classe ne possède pas de constructeur public : l'utilisateur ne peut pas créer d'objet de ce type. Les objets de ce type sont créés par l'opérateur `typeid`[115]. La définition complète de cette classe dépend de l'implémentation, mais ce qui suit décrit l'interface minimum garantie.

```
class type_info
{
  public :
    const char *name() const ;
    bool operator==(const type_info &) const ;
    bool operator!=(const type_info &) const ;
    bool before(const type_info &) const ;

  private :
    type_info(const type_info &) ;
    type_info &operator=(const type_info &) ;
  /* ... */
} ;
```

[115] Voir *L'opérateur typeid* page 129.

```
const char *type_info::name() const ;
```

`name` renvoie une chaîne de caractères représentant le nom du type auquel se rapporte l'objet courant.

```
bool type_info::before(const type_info &) ;
```

`before` compare deux objets `type_info` suivant une relation d'ordre dépendante de l'implémentation.

Exemple

```
class A
{
  virtual void f() {} ;              // pour être polymorphe
} ;

class B : public A {} ;

void f(A a)
{ cout << typeid(a).name() << '\n' ; }

void g(A &a)
{ cout << typeid(a).name() << '\n' ; }

void h(A *a)
{
  cout << typeid(a).name() << ',' ;
  cout << typeid(*a).name() << '\n' ;
}

void main()
{
  A a ;
  B b ;
  f(a) ; f(b) ;
  g(a) ; g(b) ;
  h(&a) ; h(&b) ;
}
```

L'exécution de ce programme donne :

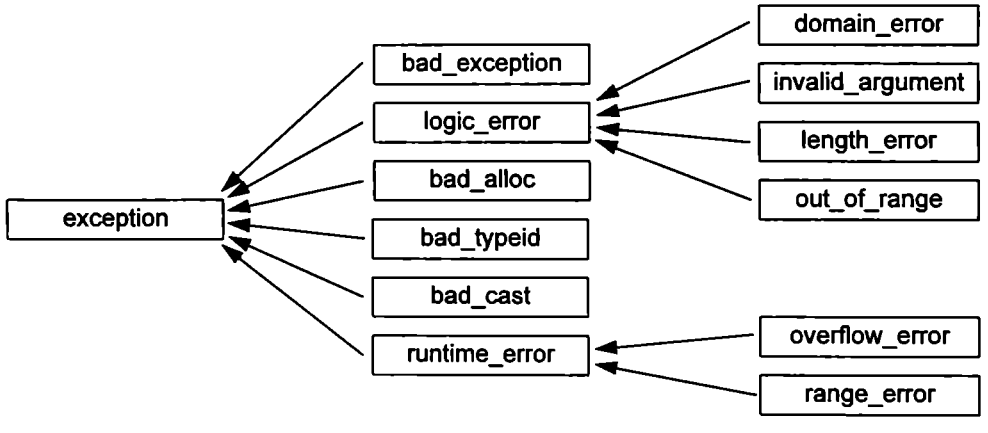
```
A
A
A
B
A *, A
A *, B
```

II - DIAGNOSTIC

Cette catégorie contient les composants utilisés pour détecter et signaler les erreurs d'exécution. Elle définit en particulier les classes exceptions.

Les classes exceptions

C'est un ensemble de classes polymorphes représentant le type des objets lancés en cas d'erreur pendant l'exécution des fonctions de la bibliothèque. Ces classes sont déclarées dans `<stdexcept>`, `<new>` et `<typeinfo>`. Le graphe d'héritage est le suivant :



Ces classes peuvent servir de classes de base pour les classes exceptions de l'utilisateur.

i. La classe `exception`

```
class exception
{
  public :
    exception() throw() ;
    exception(const exception &) throw() ;
    exception &operator=(const exception &) throw() ;
    virtual ~exception() throw() ;
    virtual const char *what() const throw() ;
  /* ... */
} ;
```

La classe `exception` est le sommet de la hiérarchie des classes exceptions.

```
virtual const char *exception::what() ;
```

`what` renvoie un message explicatif sur la nature de l'exception.

ii. La classe `bad_cast`

```
class bad_cast : public exception
{
  public :
    bad_cast() throw() ;
  /* ... */
} ;
```

La classe `bad_cast` est une exception rapportant l'exécution d'une expression `dynamic_cast` invalide.

iii. La classe `bad_typeid`

```
class bad_typeid : public exception
{
  public :
    bad_typeid() throw() ;
  /* ... */
} ;
```

La classe `bad_typeid` est une exception rapportant un pointeur `p` nul dans une expression `typeid(*p)`.

iv. La classe `bad_alloc`

```
class bad_alloc : public exception
{
  public :
    bad_alloc() throw() ;
  /* ... */
} ;
```

La classe `bad_alloc` est une exception rapportant un échec d'allocation mémoire.

v. La classe `bad_exception`

```
class bad_exception : public exception
{
  public :
    bad_exception() throw() ;
  /* ... */
} ;
```

La classe `bad_exception` est une exception rapportant une violation des spécifications d'exceptions (voir *Spécification des exceptions*, page 165). C'est typiquement une exception lancée par la fonction `unexpected`.

vi. La classe `logic_error`

```
class logic_error : public exception
{
  public :
    logic_error(const string &what) ;
  /* ... */
} ;
```

La classe `logic_error` est une exception rapportant une erreur présumée détectable avant l'exécution du programme (comme les violations des préconditions, postconditions ou invariants de classe).

vii. La classe `runtime_error`

```
class runtime_error : public exception
{
  public :
    runtime_error(const string &what) ;
  /* ... */
} ;
```

La classe `runtime_error` est une exception rapportant une erreur présumée détectable seulement lors de l'exécution du programme.

viii. La classe `domain_error`

```
class domain_error : public logic_error
{
  public :
    domain_error(const string &what) ;
  /* ... */
} ;
```

La classe `domain_error` est une exception rapportant une erreur de domaine.

ix. La classe `invalid_argument`

```
class invalid_argument : public logic_error
{
  public :
    invalid_argument(const string &what) ;
  /* ... */
} ;
```

La classe `invalid_argument` est une exception rapportant une erreur d'argument invalide.

x. La classe `length_error`

```
class lenght_error : public logic_error
{
  public :
    lenght_error(const string &what) ;
  /* ... */
} ;
```

La classe `lenght_error` est une exception rapportant une tentative de création d'un objet de taille supérieure ou égale à la taille maximale autorisée.

xi. La classe `out_of_range`

```
class out_of_range : public logic_error
{
  public :
    out_of_range(const string &what) ;
  /* ... */
} ;
```

La classe `out_of_range` est une exception rapportant une erreur d'argument hors limite.

xii. La classe `overflow_error`

```
class overflow_error : public runtime_error
{
  public :
    overflow_error(const string &what) ;
  /* ... */
} ;
```

La classe `overflow_error` est une exception rapportant une erreur de dépassement des limites arithmétiques.

xiii. La classe `range_error`

```
class range_error : public runtime_error
{
  public :
    range_error(const string &what) ;
  /* ... */
} ;
```

La classe `range_error` est une exception rapportant une erreur de rang.

Exemple

La fonction `main` suivante intercepte toutes les exceptions dérivées de `exception` et en écrit le type et la raison :

```
void main()
{
  try
  {
    /* ... */
  }
  catch(const exception &e)
  {
    cout << "exception " << typeid(e).name()
         << " : " << e.what() << '\n' ;
    terminate() ;
  }
  catch(...)
  {
    cout << "exception hors hiérarchie\n" ;
    terminate() ;
  }
}
```

L'exemple suivant contrôle le bon déroulement de l'allocation de mémoire :

```
int main()
{
  char *ptr ;
  int nb ;

  /* ... */
  try
  {
    ptr = new char[nb] ;
  }
  catch(bad_alloc &e)
  {
    cout << "mémoire insuffisante !\n"
         << e.what() ;
    terminate() ;
  }
  /* ... */
}
```

Notons la référence dans le type du gestionnaire, qui permet d'avoir une liaison dynamique dans l'expression `e.what()`.

III - UTILITAIRES GENERAUX

Cette catégorie concerne les composants utilisés par d'autres éléments de la bibliothèque. Ces composants peuvent aussi être utilisés directement par les programmeurs.

1 - Générateurs d'opérateurs

Ces générateurs permettent de déduire, pour n'importe quel type d'argument, l'expression de l'opérateur `!=` à partir de celle de l'opérateur `=`, ainsi que l'expression des opérateurs `>`, `>=`, `<=` à partir de celles de `=` et de `<`. Ils sont définis dans `<utility>` de la façon suivante :

```
template <class T>
bool operator!=(const T &x, const T &y)
{ return ! (x==y) ; }
```

```
template <class T>
bool operator>(const T &x, const T &y)
{ return y<x ; }
```

```
template <class T>
bool operator<=(const T &x, const T &y)
{ return ! (y<x) ; }
```

```
template <class T>
bool operator>=(const T &x, const T &y)
{ return ! (x<y) ; }
```

2 - La structure `pair`

Le modèle de structure `pair` représente une association de deux valeurs de types quelconques. Il est définie dans `<utility>` comme :

```
template <class T1, class T2>
struct pair
{
  T1 first ;
  T2 second ;
  pair(const T1 &x, const T2 &y) ;
};
```

Deux opérateurs relationnels y sont associés :

```
template <class T1, class T2>
bool operator==(const pair<T1,T2> &x, const pair<T1,T2>
&y)
{ return x.first==y.first && x.second==y.second ; }
```

```
template <class T1, class T2>
bool operator<(const pair<T1,T2> &x, const pair<T1,T2> &y)
{ return x.first<y.first ||
              (!(y.first<x.first) && x.second<y.second) ; }
```

Grâce aux générateurs d'opérateurs précédents, les opérateurs `!=`, `>`, `<=` et `>=` sont donc aussi définis pour deux valeurs de type `pair`. Ils permettent la comparaison de deux paires de même type.

Le modèle de fonction :

```
template <class T1, class T2>
pair<T1,T2> make_pair(const T1 &x, const T2 &y) ;
```

crée un objet `pair` à partir de deux valeurs. Il simplifie la création d'objets temporaires, comme dans l'expression :

```
if ( p < make_pair(a, make_pair(i,x)) ) /* ... */
```

qui compare `p` à la paire composée `(a,(i,x))`.

3 - Les classes-fonctions

Les classes-fonctions sont des classes qui définissent l'opération `()`[116]. Elles sont définies dans `<functional>`.

i. Structures de base

Les modèles de structures `unary_function` et `binary_function` suivants :

[116] Voir *Objets-fonctions*, page 151.

```
template <class Arg, class Result>
struct unary_function
{
  typedef Arg argument_type ;
  typedef Result result_type ;
} ;
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function
{
  typedef Arg1 first_argument_type ;
  typedef Arg2 second_argument_type ;
  typedef Result result_type ;
} ;
```

modélisent les types *fonction unaire* et *fonction binaire*.

ii. Opérations arithmétiques, relationnelles et logiques

A partir de `unary_function` et de `binary_function` sont dérivées les classes-fonctions arithmétiques `plus`, `minus`, `times`, `divides`, `modulus` et `negate` qui modélisent respectivement l'addition, la différence, la multiplication, la division, le modulo et l'opposé. Elles sont toutes construites sur le même modèle. Par exemple, `plus` est définie comme :

```
template <class T>
struct plus : binary_function<T,T,T>
{
  T operator() (const T &x, const T &y) const
                                          { return x+y ; }
} ;
```

et :

```
template <class T>
struct negate : unary_function<T,T>
{
  bool operator() (const T &x) const { return -x ; }
} ;
```

De la même façon sont définies les classes-fonctions relationnelles `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal`. `equal_to` est définie comme :

```
template <class T>
struct equal_to : binary_function<T,T,bool>
{
  bool operator() (const T &x, const T &y) const
                                          { return x==y ; }
} ;
```

et les autres sont sur le même modèle.

Enfin, sont définies les classes-fonctions logiques `logical_and`, `logical_or` et `logical_not` comme :

```
template <class T>
struct logical_and : binary_function<T,T,bool>
{
  bool operator() (const T &x, const T &y) const
                                          { return x&&y ; }
} ;
```

et de même pour `logical_or`, et :

```
template <class T>
struct logical_not : unary_function<T,bool>
{
  bool operator() (const T &x) const { return !x ; }
} ;
```

Exemple

```
template <class Op, int raison, int nb>
class Suite
{
    int tab[nb] ;

  public :
    Suite(const Op &op) {
                          tab[0] = 1 ;
                          for (int i=1 ; i<nb ; i++)
                           tab[i] = op(tab[i-1],raison) ;
                        }
    void Ecrire()       {
                          cout << '(' << tab[0] ;
                          for (int i=1 ; i<nb ; i++)
                            cout << ',' << tab[i] ;
                          cout << ")\n" ;
                        }
} ;
```

Le modèle `Suite` représente une suite d'entiers. Les termes de la suite sont initialisés à l'aide d'un objet-fonction binaire.

Le premier argument du modèle est le type de l'opérateur à appliquer au $n^{ième}$ terme de la suite pour obtenir le $n+1^{ième}$ terme, l'opérateur lui-même étant passé en argument au constructeur `Suite()`. Le second argument du modèle est la raison de la suite, et le troisième est son nombre de termes.

Ce modèle s'utilise comme :

```
void main()
{
                                          // suite arithmétique
  Suite <plus<int>,1,10> s1(plus<int>()) ;
  s1.Ecrire() ;
                                           // suite géométrique
  Suite <times<int>,2,13> s2(times<int>()) ;
  s2.Ecrire() ;
                                             // suite alternée
  Suite <times<int>,-1,8> s3(times<int>()) ;
  s3.Ecrire() ;
}
```

ce qui donne :

```
(1,2,3,4,5,6,7,8,9,10)
(1,2,4,8,16,32,64,128,256,512,1024,2048,4096)
(1,-1,1,-1,1,-1,1,-1)
```

iii. Les binders

Deux modèles de fonctions `bind1st` et `bind2nd` prennent en argument un objet-fonction binaire `f` et une valeur `x` et renvoient un objet-fonction unaire construit à partir de `f` et de `x`, `x` jouant le rôle respectivement du premier ou du second argument pour `f`. `bind1st` et `bind2nd` sont définis comme :

```
template <class Op, class T>
binder1st<Op> bind1st(const Op &op, const T &x)
{ return binder1st<Op>(op, Op::first_argument_type(x)) ; }
```

```
template <class Op, class T>
binder2nd<Op> bind2nd(const Op &op, const T &x)
{ return binder2nd<Op>(op, Op::second_argument_type(x)) ; }
```

avec `binder1st` et `binder2nd` définis comme :

```
template <class Op>
class binder1st : public
unary_function<Op::second_argument_type, Op::result_type>
{
  protected :
    Op op ;
    Op::first_argument_type value ;

  public :
    binder1st(const Op &x, const Op::first_argument_type &y)
                                        : op(x), value(y) {}
    Op::result_type operator()(const argument_type &x) const
                                      { return op(value,x) ; }
} ;
```

```
template <class Op>
class binder2nd : public
unary_function<Op::first_argument_type, Op::result_type>
{
  protected :
    Op op ;
    Op::second_argument_type value ;

  public :
    binder2nd(const Op &x,
                          const Op::second_argument_type &y)
                                        : op(x), value(y) {}
    Op::result_type operator()(const argument_type &x) const
                                      { return op(x,value) ; }
} ;
```

`bind1st` et `bind2nd` facilitent la création d'objets-fonctions `binder1st` et `binder2nd`, suivant le même principe que la fonction `make_pair`[117].

Exemple

Etant donnés le modèle de classe :

[117] Voir *make_pair* page 198.

```
template <class T, int nb>
class Tab
{
    T t[nb] ;

  public :
    T &operator[](int i) { return t[i] ; }
    int Nb() { return nb ; }
} ;
```

et le modèle de fonction :

```
template <class Tab, class Op>
int chercher(Tab &tab, Op op)
{
  int i=0 ;
  while (i<tab.Nb())
    if (op(tab[i])) return i ; else i++ ;
  return -1 ;                  // renvoie -1 si rien trouvé
} ;
```

alors l'instruction :

```
chercher(t,bind2nd(greater<int>(),20)) ;
```

renvoie l'indice du premier élément du tableau `t` strictement supérieur à `20`, alors que :

```
chercher(t,bind1st(greater<int>(),20)) ;
```

renvoie l'indice du premier élément du tableau strictement inférieur à `20`.

IV - CHAINES

Des classes chaînes peuvent être générées à partir du modèle `basic_string`, défini dans `<string>`. Le type des éléments des chaînes est l'un des paramètres du modèle.

La position du premier élément est par convention 0. Les allocations et désallocations de mémoire sont automatiques. Les fonctions membres peuvent lancer des exceptions de type `length_error` et `out_of_range`. Ce modèle est défini dans `<string>`.

La classe `string`

La classe `string` est définie comme :

```
typedef basic_string<char> string ;
```

Pour simplifier l'écriture, seule la classe `string` obtenue par instanciation de `basic_string` est décrite ci-dessous. Il est facile à partir de là d'imaginer `basic_string` comme une généralisation de `string`.

Dans ce qui suit, `pos` désigne une position de caractère, `n` désigne un nombre de caractères, et `rep` désigne un facteur de répétition. `size_t` est un type entier non signé[118].

```
class string
{
  public :
    string(const string &str, size_t pos=0, size_t n=-1) ;
    string(const char *s, size_t n) ;
    string(const char *s) ;
    string(size_t rep, char c) ;

    ~string() ;
```

[118] `-1` en est donc la plus grande valeur, puisque par définition, l'opposé d'une quantité non signée de type `T` est calculé en ôtant sa valeur à $2^{k.\text{sizeof}(T)}$, où k est le nombre de bits du type `char`.

```
const char *c_str() const ;
const char *data() const ;

const char &at(size_t pos) const ;
char &at(size_t pos) ;

char operator[](size_t pos) const ;
char &operator[](size_t pos) ;

size_t length() const ;

string &assign(const string &str, size_t pos=0,
                                        size_t n=-1) ;
string &assign(size_t rep, char c) ;

string &operator=(const string &str) ;
string &operator=(char c) ;

size_t copy(char *s, size_t n, size_t pos=0) ;

string substr(size_t pos=0, size_t n=-1) const ;

int compare(size_t pos1=0, size_t n1=-1,
 const string &str, size_t pos2=0, size_t n2=-1) const ;

void swap(string &str) ;

string &append(const string &str, size_t pos=0,
                                        size_t n=-1) ;
string &append(size_t rep, char c) ;

string &operator+=(const string &str) ;
string &operator+=(char c) ;

string &insert(size_t pos1, const string &str,
                              size_t pos2=0, size_t n=-1) ;
string &insert(size_t pos, size_t rep, char c) ;

string &replace(size_t pos1, size_t n1,
       const string &str, size_t pos2=0, size_t n2=-1) ;
string &replace(size_t pos, size_t n1, size_t n2,
                                              char c) ;

string &remove(size_t pos=0, size_t n=-1) ;

bool empty() const ;

size_t find(const string &str, size_t pos=0) const ;
size_t find(char c, size_t pos=0) const ;
```

```
    size_t rfind(const string &str, size_t pos=-1) const ;
    size_t rfind(char c, size_t pos=-1) const ;

    size_t find_first_of(const string &str,
                                        size_t pos=0) const ;
    size_t find_first_of(char c, size_t pos=0) const ;

    size_t find_first_not_of(const string &str,
                                        size_t pos=0) const ;
    size_t find_first_not_of(char c, size_t pos=0) const ;

    size_t find_last_of(const string &str,
                                       size_t pos=-1) const ;
    size_t find_last_of(char c, size_t pos=-1) const ;

    size_t find_last_not_of(const string &str,
                                       size_t pos=-1) const ;
    size_t find_last_not_of(char c, size_t pos=-1) const ;
  /* ... */
} ;
```

i. Constructeurs et destructeur

```
string::string(const string &str, size_t pos=0,
                                                size_t n=-1) ;
```

Ce constructeur de recopie crée une chaîne contenant une copie des caractères de `str`. Seuls les caractères de rang supérieur ou égal à `pos` sont copiés, la copie ayant un nombre de caractères égal à `min(n,length()-pos)`. Il lance `out_of_range` si `pos` est supérieur à `length()`.

```
string::string(const char *s, size_t n) ;
```

Ce constructeur crée une chaîne contenant une copie des premiers caractères présents à l'adresse `s`. Seuls les caractères précédents un '`\0`' sont copiés, et la copie a au plus `n` caractères.

```
string::string(const char *s) ;
```

Ce constructeur crée une chaîne contenant une copie des premiers caractères présents à l'adresse `s`. Seuls les caractères précédents un '`\0`' sont copiés. Ce constructeur définit une conversion du type `char *` vers le type `string`.

```
string::string(size_t rep, char c) ;
```

Ce constructeur crée une chaîne contenant le caractère `c` répété `rep` fois.

```
string::~string() ;
```

Le destructeur libère la mémoire utilisée.

Exemple

```
string s1 ;                                    // chaîne vide
string s2("Aubance") ;
string s3(s2,2,4) ;                          // s3 vaut "banc"
string s4('A',10) ;                  // s4 vaut "AAAAAAAAAA"
```

ii. Fonctions membres

```
const char *string::c_str() const ;
const char *string::data() const ;
```

Les fonctions `data` et `c_str` renvoient l'adresse d'une chaîne de caractères, terminée par `'\0'`, contenant les caractères de la chaîne courante. Le contenu de cette chaîne ne doit pas être directement modifié.

```
const char &string::at(size_t pos) const ;
char &string::at(size_t pos) ;
char string::operator[](size_t pos) const ;
char &string::operator[](size_t pos) ;
```

Les fonctions `at` et `operator[]` renvoient le caractère situé à la position spécifiée. `at` lance `out_of_range` si la position est invalide.

```
size_t string::length() const ;
```

La fonction `length` renvoie le nombre de caractères de la chaîne courante (cette valeur est différente de `strlen(data)` s'il y a des `'\0'` dans la chaîne).

```
string &string::assign(const string &str, size_t pos=0,
                                              size_t n=-1) ;
string &string::assign(size_t rep, char c) ;
string & string::operator=(const string &str) ;
string & string::operator=(char c) ;
```

Les fonctions `assign` et les opérateurs = affectent les caractères spécifiés à la chaîne courante. Les fonctions `assign` lancent `out_of_range` si `pos` est supérieur à `length()`.

```
size_t string::copy(char *s, size_t n, size_t pos=0) ;
```

La fonction `copy` copie au plus `n` caractères de la chaîne courante pris à partir de la position `pos`, à l'adresse spécifiée. Elle lance `out_of_range` si `pos` est supérieur à `length()`.

```
string string::substr(size_t pos=0, size_t n=-1) const ;
```

La fonction `substr` crée une chaîne d'au plus `n` caractères, contenant une copie des caractères de la chaîne courante de rang supérieur ou égal à `pos`. Elle lance `out_of_range` si `pos` est supérieur à `length()`.

```
int string::compare(size_t pos1, size_t n1,
     const string &str, size_t pos2=0, size_t n2=-1) const ;
```

La fonction compare compare les caractères de la chaîne courante aux caractères spécifiés. Cette fonction renvoie une valeur <0, =0 ou >0 suivant que la chaîne courante est <, = ou > aux caractères spécifiés. Elle lance out_of_range si pos est supérieur à length().

```
void string::swap(string &str) ;
```

La fonction swap échange le contenu de str avec celui de la chaîne courante.

```
string &string::append(const string &str, size_t pos=0,
                                                size_t n=-1) ;
string &string::append(size_t rep, char c) ;
string &string::operator+=(const string &str) ;
string &string::operator+=(char c) ;
```

Les fonctions append et les opérateurs += ajoutent à la chaîne courante les caractères spécifiés. Les fonctions append lance out_of_range si pos est supérieur à length(). Ces fonctions lancent length_error si la somme des deux longueurs est supérieure à la plus grande valeur de type size_t.

```
string &string::insert(size_t pos1, const string &str,
                                size_t pos2=0, size_t n=-1) ;
string &string::insert(size_t pos, size_t rep, char c) ;
```

Les fonctions insert insèrent les caractères spécifiés dans la chaîne courante. Elles lancent out_of_range si pos ou pos1 est supérieur à length() ou si pos2 est supérieur à str.length(). Elles lancent length_error si la somme des deux longueurs est supérieure à la plus grande valeur de type size_t.

```
string &string::replace(size_t pos1, size_t n1,
          const string &str, size_t pos2=0, size_t n2=-1) ;
string &string::replace(size_t pos, size_t n1 size_t n2,
                                                char c) ;
```

Les fonctions replace remplacent dans la chaîne courante les caractères spécifiés en premier par les caractères spécifiés en second. Elles lancent out_of_range si pos ou pos1 est supérieur à length() ou si pos2 est supérieur à str.length(). Elles lancent length_error si la longueur résultante est supérieure à la plus grande valeur de type size_t.

```
string &string::remove(size_t pos=0, size_t n=-1) ;
```

La fonction remove supprime les caractères spécifiés de la chaîne courante. Elle lance out_of_range si pos est supérieur à length().

```
bool string::empty() const ;
```

La fonction empty vide la chaîne courante.

```
size_t string::find(const string &str, size_t pos=0) const ;
size_t string::find(char c, size_t pos=0) const ;
```

Les fonctions `find` recherchent dans la chaîne courante, à partir de la position `pos`, la première occurrence des caractères spécifiés. Elles renvoient la position si trouvée, `-1` sinon.

```
size_t string::rfind(const string &str,
                                        size_t pos=-1) const ;
size_t string::rfind(char c, size_t pos=-1) const ;
```

Les fonctions `rfind` recherchent dans la chaîne courante, à partir de la position `pos`, la dernière occurrence des caractères spécifiés. Elles renvoient la position si trouvée, `-1` sinon.

```
size_t string::find_first_of(const string &str,
                                         size_t pos=0) const ;
size_t string::find_first_of(char c, size_t pos=0) const ;
```

Les fonctions `find_first_of` recherchent dans la chaîne courante, à partir de la position `pos`, la première occurrence de l'un des caractères spécifiés. Elles renvoient la position si trouvée, `-1` sinon.

```
size_t string::find_first_not_of(const string &str,
                                         size_t pos=0) const ;
size_t string::find_first_not_of(char c,
                                         size_t pos=0) const ;
```

Les fonctions `find_first_not_of` recherchent dans la chaîne courante, à partir de la position `pos`, la première occurrence d'un caractère non spécifié. Elles renvoient la position si trouvée, `-1` sinon.

```
size_t string::find_last_of(const string &str,
                                        size_t pos=-1) const ;
size_t string::find_last_of(char c, size_t pos=-1) const ;
```

Les fonctions `find_last_of` recherchent dans la chaîne courante, à partir de la position `pos`, la dernière occurrence de l'un des caractères spécifiés. Elles renvoient la position si trouvée, `-1` sinon.

```
size_t string::find_last_not_of(const string &str,
                                        size_t pos=-1) const ;
size_t string::find_last_not_of(char c,
                                        size_t pos=-1) const ;
```

Les fonctions `find_last_not_of` recherchent dans la chaîne courante, à partir de la position `pos`, la dernière occurrence d'un caractère non spécifié. Elles renvoient la position si trouvée, `-1` sinon.

Exemple

```
string s("paté aux prunes.") ;

s.at(1,'â') ;                    // s vaut "pâté aux prunes."
for (i=0 ; i<s.length() ; i++) cout << s.at(i) ;

if (s.compare("c")>0) /* ... */  // la condition est vraie

printf("%s",s.data()) ;          // équivalent à cout << s

i = s.find(" ") ;                                  // i vaut 4
i = s.rfind(" ") ;                                 // i vaut 8
i = s.find_first_of(",. ") ;                       // i vaut 4
i = s.find_last_of(",. ") ;                        // i vaut 15
i = s.find_first_not_of(",. ") ;                   // i vaut 0
i = s.find_last_not_of(",. ") ;                    // i vaut 14


string s1(s.substr(9,6)) ;              // s1 vaut "prunes"

s.assign("pâté aux ") ;                 // s vaut "pâté aux "
s.append(s) ;                  // s vaut "pâté aux prunes"

char buf[11] ;
s.copy(buf,4,0) ;       // buf vaut "pâté" sans '\0' en fin

s.insert(0,"un bon ") ; // s vaut "un bon pâté aux prunes"

s.remove(3,4) ;              // s vaut "un pâté aux prunes"
s.replace(8,3,"sans") ;     // s vaut "un pâté sans prunes"

s = "pâte" ;                                    // s vaut "pâte"
s += " aux prunes" ;            // s vaut "pâte aux prunes"

for (int i=0 ; i<s.length() ; i++) cout << s[i] ;
s[3] = 'é' ;                    // s vaut "pâté aux prunes"
```

iii. Fonctions non membres apparentées

```
string operator+(const string &str1, const string &str2) ;
string operator+(char c, const string &str) ;
string operator+(const string &str, char c) ;
```

Ces opérateurs concatènent les caractères et créent une nouvelle chaîne (voir append).

```
bool operator==(const string &str1, const string &str2) ;
bool operator!=(const string &str1, const string &str2) ;
bool operator<(const string &str1, const string &str2) ;
bool operator>(const string &str1, const string &str2) ;
bool operator<=(const string &str1, const string &str2) ;
bool operator>=(const string &str1, const string &str2) ;
```

Ces opérateurs testent les caractères (voir `compare`).

```
istream &operator>>(istream &is, string &str) ;
```

Cet opérateur initialise la chaîne spécifiée avec les caractères extraits du flux.

```
ostream &operator<<(ostream &os, const string &str) ;
```

Cet opérateur insère la chaîne spécifiée dans le flux.

```
istream &getline(istream &is, string &str, char delim='\n') ;
```

`getline` initialise la chaîne spécifiée avec les caractères extraits du flux, jusqu'à la rencontre de `delim` ou de la dernière séquence d'entrée.

Exemple

```
string b("anguilles") ;

string a = "roulade d'"+b ;      // a="roulade d'anguilles"

if (a==b) /* ... */              // la condition est fausse
if (a!=b) /* ... */               // la condition est vraie

cin >> a ;  // initialise a avec la chaîne extraite de cin
cout << a ;                                 // envoie a sur cout

getline(cin,a) ;              // initialise a avec une ligne
                                              // extraite de cin
```

V - CONTENEURS

Les conteneurs sont des objets contenant d'autres objets. Ils prennent en charge l'allocation et la libération de ces objets à travers leurs constructeurs, destructeurs et leurs opérateurs d'insertion et de suppression. Parmi les diverses sortes de conteneurs, on distingue les *séquences*, *les types abstraits* et les *conteneurs associatifs*.

1 - Les séquences

Une séquence est un conteneur contenant un ensemble d'objets de type identique, mémorisés linéairement.

La catégorie des conteneurs propose trois séquences de base : `vector`, `list` et `deque`. Elle offre également une séquence `bitset`, représentant une séquence de bits de taille fixe, munie des opérations de manipulation de bits habituelles `&`, `|`, `^`, `>>`, `<<`, `&=`, `|=`, `^=`, `>>=`, `<<=`.

Les modèles de classes `vector`, `list` et `deque` se différentient entre eux par la complexité des algorithmes qui leur sont appliqués. C'est le critère que le programmeur doit utiliser pour choisir la séquence adéquate en fonction des traitements à réaliser.

Des *itérateurs*[119] sont associés à ces séquences et permettent d'accéder à leurs éléments de différentes façons (aléatoirement, séquentiellement, etc.). Des fonctions d'accès aux éléments sont également implémentées dans les séquences mêmes : ce sont les accès *naturels* particulièrement optimisés pour chacun des différents types de séquence.

i. Le modèle de classe `vector`

```
template <class T>
class vector ;
```

[119] Voir *Itérateurs* page 216.

Cette séquence supporte les itérateurs à accès direct. Elle dispose d'opérations d'insertion et de suppression en fin de séquence, dont la complexité est constante. Elle dispose d'opérations d'insertion et de suppression en milieu, dont la complexité est linéaire. `vector` est le type de séquence qui doit être utilisé par défaut.

ii. Le modèle de classe `list`

```
template <class T>
class list ;
```

Cette séquence supporte les itérateurs bidirectionnels. Elle dispose d'opérations d'insertion et de suppression en tout point, de complexité constante. Elle n'autorise pas un accès direct rapide. `list` doit être utilisé quand les insertions et les suppressions en milieu de séquence sont fréquentes.

iii. Le modèle de classe `deque`

```
template <class T>
class deque ;
```

Cette séquence supporte les itérateurs à accès direct. Elle dispose d'opérations d'insertion et de suppression en début et en fin de séquence, de complexité constante. Elle dispose d'opérations d'insertion et de suppression en milieu, de complexité linéaire. Etant particulièrement optimisée pour des manipulations en début et en fin de séquence, `deque` doit donc être utilisée lorsque la plupart des ajouts et des suppressions se font aux extrémités.

2 - Types abstraits

La catégorie des conteneurs propose également trois types abstraits : `stack`, `queue` et `priority_queue`.

Ces types abstraits se construisent par dessus un autre conteneur. Ils jouent le rôle d'interface entre le conteneur à partir duquel ils sont construits, et l'utilisateur. Ils transforment et limitent les opérations applicables au conteneur sous-jacent.

i. Le modèle de classe `queue`

```
template <class Container>
class queue ;
```

Ce type abstrait limite l'accès aux seuls éléments de tête et de queue du conteneur sous-jacent. `Container` est le container sous-jacent. Les insertions ont lieu en queue, et les suppressions en tête. Les traitements autorisés sont donc de type *file d'attente*.

Exemple

```
queue<list<int> > file ;
```

L'objet `file` est une queue édifiée sur une liste d'entiers. Seules des manipulations en début et en fin de liste sont autorisées sur `file`.

ii. Le modèle de classe `priority_queue`

```
template <class Container, class Compare>
class priority_queue ;
```

Ce type abstrait associe un ordre aux éléments mémorisés. `Container` est le container sous-jacent, et `Compare` est la relation d'ordre. Seules des manipulations en tête de séquence sont autorisées. Les insertions ont lieu en queue, et les suppressions en tête.

Exemple

```
priority_queue<vector<float>,greater_equal<float> > file ;
```

L'objet `file` est une queue ordonnée basée sur un tableau de réels. La relation utilisée pour ordonner les éléments est la classe-fonction `greater_equal<float>`[120]. Seules des manipulations en début et en fin de liste sont autorisées sur `file`.

iii. Le modèle de classe `stack`

```
template <class Container>
class stack ;
```

Ce type abstrait définit une structure de pile sur le conteneur sous-jacent. `Container` est le container sous-jacent. Seules des manipulations en sommet de pile sont autorisées. Les insertions et les suppressions ont lieu en sommet de pile.

[120] Voir *Les classes-fonctions* page 198.

Exemple

```
stack<deque<double> > pile ;
```

L'objet `pile` est une pile basée sur une `deque` de réels. Seules les opérations habituelles sur les piles sont autorisées sur `pile`.

3 - Les containers associatifs

Un conteneur associatif offre la possibilité de rechercher rapidement les éléments mémorisés, à partir d'une clé.

La bibliothèque offre quatre conteneurs associatifs : `set`, `multiset`, `map` et `multimap`.

Ces conteneurs sont paramètrés avec le type de la clé et une relation de comparaison. Celle-ci doit induire un ordre total sur les clés.

i. Les modèles de classes `set` *et* `multiset`

```
template <class Key, class Compare=less<Key> >
class set ;
```

```
template <class Key, class Compare=less<Key> >
class multiset ;
```

Pour `set` et `multiset`, la clé est la valeur même de l'élément. Chaque élément mémorisé est de type `Key`.

`set` exige que les valeurs des clés soient uniques. `multiset` accepte plusieurs occurrences d'une même valeur.

ii. Les modèles de classes `map` *et* `multimap`

```
template <class Key, class T, class Compare=less<Key> >
class map ;
```

```
template <class Key, class T, class Compare=less<Key> >
class multimap ;
```

Pour `map` et `multimap`, un élément est composé d'une clé et d'une valeur. Chaque élément mémorisé est de type `pair<const Key,T>`.

`map` exige que les valeurs des clés soient uniques. `multimap` accepte plusieurs occurrences d'une même valeur.

VI - ITERATEURS

Les itérateurs permettent d'exécuter des itérations sur les conteneurs. Ils permettent de travailler sur différentes structures de données de façon uniforme et générique.

On distingue en particulier les itérateurs séquentiels unidirectionnels, les itérateurs séquentiels bidirectionnels et les itérateurs à accès direct. Un itérateur séquentiel bidirectionnel peut être utilisé à la place d'un itérateur séquentiel unidirectionnel, et un itérateur à accès direct peut être utilisé à la place d'un itérateur séquentiel bidirectionnel.

Deux itérateurs de même type peuvent être comparés avec `==` et `!=`, et affectés avec `=`. L'opérateur unaire `*`, appliqué à un itérateur, renvoie la valeur de l'élément courant du conteneur sous-jacent pour cet itérateur. On peut voir les itérateurs comme une généralisation des pointeurs.

Les conteneurs peuvent créer des itérateurs. En particulier, les conteneurs `vector`, `list` et `deque` disposent de fonctions membres `begin` et `end` qui renvoient un itérateur positionné respectivement sur leur premier élément et sur leur marque de dépassement (dont la position se situe après celle du dernier élément). Un itérateur référençant la marque de dépassement ne doit jamais être déréférencé avec l'opérateur `*`.

1 - L'itérateur séquentiel unidirectionnel

```
template <class T>
struct forward_iterator ;
```

Cet itérateur possède des opérateurs `++` préfixé et postfixé, qui passent à l'élément suivant, et qui renvoient l'élément courant respectivement après et avant le déplacement :

Exemple

```
vector<int> v(100) ;
/* ... */
for (forward_iterator<int> it=v.begin() ;
                                        it<>v.end() ; it++)
  cout << *it ;
```

Cette boucle écrit tous les éléments du tableau v.

2 - L'itérateur séquentiel bidirectionnel

```
template <class T>
struct bidirectional_iterator ;
```

Cet itérateur possède en plus deux opérateurs `--` préfixé et postfixé, qui passent à l'élément précédent, et qui renvoient l'élément courant respectivement après et avant le déplacement :

Exemple

```
list<float> l(100) ;
/* ... */
for (bidirectional_iterator<float> it=l.end() ;
                                    it<>l.begin() ; it--)
  cout << *it ;
```

Cette boucle écrit tous les éléments de la liste v, du dernier au premier.

3 - L'itérateur à accès direct

```
template <class T>
struct random_access_iterator ;
```

Cet itérateur possède en plus un opérateur `[]` permettant un accès direct aux éléments du conteneur sous-jacent, des opérateurs `+` , `-` , `+=` et `-=` permettant un déplacement relatif, et des opérateurs relationnels `<`, `>`, `<=` et `>=` permettant de comparer la position courante de deux itérateurs basés sur le même conteneur :

Exemple

```
vector<int> v(100) ;
/* ... */
random_access_iterator<float> it=l.begin() ;
for (int i=0 ; i<v.size() ; i++)
  cout << it[i] ;
```

Cette boucle écrit tous les éléments du tableau `v`.

VII - ALGORITHMES

Cette catégorie contient des composants réalisant des traitements algorithmiques sur les conteneurs. Ces algorithmes travaillent indépendamment de l'implémentation des structures de données, et sont paramètrés par des types itérateurs. Ils sont déclarés dans `<algorithm>`.

```
template<class Iterator, class T>
Iterator find(Iterator first, Iterator last,
                                        const T &value) ;
```

L'algorithme `find` renvoie un itérateur sur le premier élément égal à `value`. Les éléments parcourus sont ceux du conteneur associé aux itérateurs `first` et `last`, dans l'intervalle [`first`,`last`].

```
template<class Iterator, class Predicate>
Iterator find_if(Iterator first, Iterator last,
                                        Predicate pred) ;
```

Lors de l'exécution de l'algorithme `find_if`, le prédicat[121] `pred` est appliqué sur chacune des valeurs de l'intervalle [`first`,`last`], et évalué. `find_if` renvoie le premier élément de l'intervalle qui vérifie le prédicat.

```
template<class Iterator, class Function>
void for_each(Iterator first, Iterator last, Function f) ;
```

L'algorithme `for_each` applique la fonction (ou l'objet-fonction) `f` à tous les éléments de l'intervalle [`first`,`last`].

```
template<class Iterator, class T, class Size>
void count(Iterator first, Iterator last, const T &value,
                                        Size &n) ;
```

L'algorithme `count` renvoie par `n` le nombre d'éléments compris dans l'intervalle [`first`,`last`] dont la valeur est égale à `value`.

---

[121] Un prédicat est une fonction ou un objet-fonction renvoyant une valeur de type `bool`.

```
template<class Iterator, class Predicate, class Size>
void count_if(Iterator first, Iterator last, Predicate pred,
                                                Size &n) ;
```

L'algorithme `count_if` renvoie par `n` le nombre d'éléments dans l'intervalle `[first,last]` vérifiant le prédicat `pred`.

```
template<class IIterator, class OIterator, class
UnaryOperation>
OIterator transform(IIterator first, IIterator last,
                    OIterator result, UnaryOperation op) ;
```

L'algorithme `transform` applique aux éléments de l'intervalle `[first,last]` l'opérateur `op`, et mémorise les valeurs obtenues à partir de `result`.

```
template<class Iterator, class T>
void replace(Iterator first, Iterator last, const T &old,
                                            const T &new) ;
```

L'algorithme `replace` initialise tous les éléments dans l'intervalle `[first,last]` ayant la valeur `old` à la valeur `new`.

```
template<class Iterator, class Predicate, class T>
void replace_if(Iterator first, Iterator last, Predicate
pred,
                                            const T &new) ;
```

L'algorithme `replace_if` initialise tous les éléments dans l'intervalle `[first,last]` vérifiant le prédicat `pred` à la valeur `new`.

Beaucoup d'autres algorithmes existent : diverses recherches (minimum, maximum, dichotomique, etc.), copies, remplissages, suppressions, permutations, rotations, partitionnements, tris, fusions, intersections, unions, différences symétriques... Leur utilisation est basée sur le même principe.

Exemple

Pour rechercher le premier élément du tableau `v` supérieur à `5`, on peut écrire :

```
vector<double> v(100) ;
/* ... */
find(v.begin(),v.end(),bind2nd(greater<int>(),5)) ;
```

L'exemple suivant multiplie par `2` tous les éléments de la liste `l`.

```
list<double> l ;
/* ... */
transform(l.begin(),l.end(),l.begin(),
                            bind1st(times<double>(),2)) ;
```

VIII - NUMERIQUES

Cette catégorie concerne des composants relatifs au calcul numérique. On y trouve en particulier les classes complexes, les vecteurs et des algorithmes numériques.

1 - Le modèle de classe `complex`

```
template <class T>
class complex ;
```

Ce modèle représente les nombres complexes. Il est défini dans `<complex>`. Il dispose des opérations arithmétiques, trigonométriques, exponentielles, logarithmiques, hyperboliques... Il dispose aussi des opérations spécifiques classiques (module, argument, conjugué, transformation polaire) et des opérations d'entrée/sortie. Trois instances sont spécialisées :

```
class complex<float> ;
class complex<double> ;
class complex<long double> ;
```

2 - Le modèle de classe `valarray`

```
template <class T>
class valarray ;
```

Ce modèle est défini dans `<valarray>`. Il représente un vecteur de composantes de type `T`, indicées à partir de `0`. Les opérations arithmétiques, relationnelles, logiques, de manipulation de bits, trigonométriques, trigonométriques inverses, exponentielles, logarithmiques, hyperboliques, etc. sont applicables globalement aux vecteurs. L'opération est dans ce cas appliquée à chacune des composantes du vecteur. Par exemple `v1+v2` renvoie un vecteur dont les composantes sont les sommes des composantes de `v1` et `v2`. De même, `-v` renvoie un vecteur dont les composantes sont les opposées des composantes de `v`.

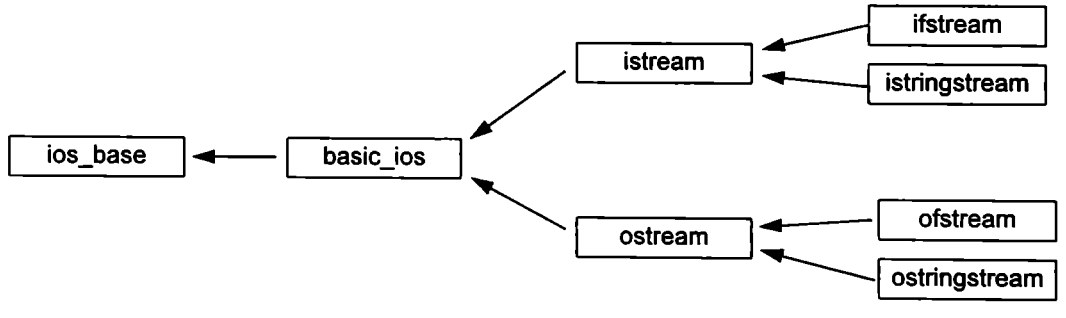
IX - ENTREE/SORTIE

Cette catégorie propose des classes et des modèles pour manipuler les flux d'entrée/sortie. Le principal avantage de ces classes, par rapport aux fonctions d'entrée/sortie du langage C, c'est leur possibilité d'être étendues aux types définis par l'utilisateur.

Beaucoup de classes et de modèles sont liées aux flux. Les principales à connaître sont :

— `istream` et `ostream` pour exploiter les services de `cin`, `cout`, `cerr` et `clog`,
— `ifstream` et `ofstream` pour utiliser les fichiers de données, et
— `istringstream` et `ostringstream` pour utiliser les tampons mémoire.

Cependant, pour pouvoir les exploiter pleinement, il est nécessaire de connaître en partie les classes et modèles `ios_base` et `basic_ios`. Le graphe d'héritage peut être résumé comme :



1 - La classe `ios_base`

`ios_base` est la classe de base pour les entrées/sorties. C'est une classe abstraite. Elle est définie dans `<ios>`.

```
class ios_base
{
  public :
    class failure
    {
      public :
      explicit failure(const string &what) ;
      /* ... */
    } ;

    enum openmode { app, ate, in, out, binary, trunc };

    enum seekdir { beg, cur, end } ;

    int width() const ;
    int width(int) ;

    int precision() const ;
    int precision(int) ;

  protected :
    ios_base() ;
  /* ... */
} ;
```

```
class ios_base::failure : public exception
{
  public :
  explicit failure(const string &what) ;
  /* ... */
} ;
```

La classe `failure` est la classe de base pour toutes les exceptions lancées lors d'erreurs sur les flux d'entrée/sortie.

```
enum ios_base::openmode
{
  app,                                  // en mode ajout à la fin
  ate,                                     // positionné à la fin
  in,                                          // en mode lecture
  out,                                        // en mode écriture
  binary,                                      // en mode binaire
  trunc                    // efface le contenu à l'ouverture
};
```

L'énumération `openmode` spécifie les différents modes d'ouverture d'un flux.

```
enum ios_base::seekdir
{
  beg,                                  // relativement au début
  cur,                  // relativement à la position courante
  end                                   // relativement à la fin
} ;
```

L'énumération `seekdir` spécifie à quoi se rapporte le déplacement spécifié lors d'opération de navigation dans un flux.

```
int ios_base::width() const ;
int ios_base::width(int) ;
```

Ces fonctions permettent d'obtenir et de modifier la largeur du champ de sortie (voir aussi le manipulateur `setw`).

```
int ios_base::precision() const ;
int ios_base::precision(int) ;
```

Ces fonctions permettent d'obtenir et de modifier le nombre de décimales lors de la sortie des réels (voir aussi le manipulateur `setprecision`).

Manipulateurs

Les manipulateurs sont déclarés dans `<iomanip>`.

```
dec          hex          oct          setbase(int)
```

Ces manipulateurs permettent de modifier la base courante pour l'entrée et la sortie des entiers.

```
showbase   noshowbase
```

Ces manipulateurs spécifient si un indicateur de base doit être généré lors de la sortie des entiers.

```
fixed        scientific
```

Ces manipulateurs spécifient si les réels sont sortis en notation fixe ou scientifique.

```
left         right
```

Ces manipulateurs spécifient le cadrage des sorties.

```
showpoint  noshowpoint
```

Ces manipulateurs spécifient si un point doit être généré inconditionnellement lors de la sortie des réels.

```
showpos      noshowpos
```

Ces manipulateurs spécifient si un signe + doit être généré lors de la sortie d'un nombre positif.

```
uppercase  nouppercase
```

Ces manipulateurs spécifient si les minuscules doivent être remplacées par leur majuscule équivalente lors des sorties.

```
skipws     noskipws
```

Ces manipulateurs spécifient si les espaces de tête doivent être ignorés lors de certaines entrées.

```
setw(int)
```

Ce manipulateur permet d'obtenir et de modifier la largeur du champ de sortie (voir aussi les fonctions `width`).

```
setfill(int)
```

Ce manipulateur permet d'obtenir et de modifier le caractère de remplissage des sorties, c'est-à-dire le caractère utilisé pour obtenir la largeur de champ spécifiée (voir aussi les fonctions `fill`).

```
setprecision(int)
```

Ce manipulateur permet d'obtenir et de modifier le nombre de décimales lors de la sortie des réels (voir aussi les fonctions `precision`).

Des exemples d'utilisation de manipulateurs sont donnés aux pages 229 et 232.

2 - Le modèle de classe `basic_ios`

`basic_ios` ajoute des fonctionnalités à `ios_base`, dont elle hérite. Il est définie dans `<ios>`.

Le modèle `basic_ios` est paramètré avec le type des entités qui transitent dans le flux. Pour simplifier l'écriture, `basic_ios` est décrit ci-dessous comme si c'était une classe.

```
class basic_ios : public ios_base
{
  public :
    bool good() const ;
    bool eof() const ;
    bool fail() const ;
    bool bad() const ;
```

```
    int fill() const ;
    int fill(int) ;

    operator bool() const ;
    bool operator! () const ;

    void clear() ;
  /* ... */
} ;
```

```
bool basic_ios::good() const ;
bool basic_ios::eof() const ;
bool basic_ios::fail() const ;
bool basic_ios::bad() const ;
```

Ces fonctions renvoient des indications sur l'état du flux. `good` renvoie `true` si aucun indicateur d'erreur n'est positionné. `eof` renvoie `true` si le flux atteint la fin de la séquence d'entrée. `fail` renvoie `true` si la dernière opération a échoué. `bad` renvoie `true` s'il y a eu perte d'intégrité des données.

```
int basic_ios::fill() const ;
int basic_ios::fill(int) ;
```

Ces fonctions permettent d'obtenir et de modifier le caractère de remplissage des sorties, c'est-à-dire le caractère utilisé pour obtenir la largeur de champ spécifiée (voir aussi le manipulateur `setfill`).

```
operator basic_ios::bool() const ;
bool basic_ios::operator! () const ;
```

L'opérateur `operator bool` renvoie `!fail()`, et l'opérateur `operator!` renvoie `fail()`. Ces deux opérateurs permettent d'utiliser un `basic_ios` dans une expression conditionnelle.

Lorsqu'un flux est en état d'erreur, toute opération échoue jusqu'à ce que l'erreur soit corrigée et que le bit correspondant soit remis à zéro.

```
void basic_ios::clear() ;
```

Cette fonction efface les indicateurs d'erreur.

3 - La classe `istream`

Cette classe permet d'extraire des informations formatées et non formatées d'un flux d'entrée. Elle hérite de `basic_ios`. Elle est définie dans `<istream>`.

En réalité, `istream` est une instance du modèle `basic_istream` :

```
typedef basic_istream<char> istream ;
```

et `basic_istream` dérive du modèle `basic_ios`.

Mais pour simplifier l'écriture, seule la classe `istream` obtenue par instanciation de `basic_istream` est décrite ci-dessous.

```
class istream : virtual public basic_ios
{
  public :
    istream &operator>> (char *s) ;
    istream &operator>> (char &c) ;
    istream &operator>> (bool &b) ;
    istream &operator>> (short &s) ;
    istream &operator>> (unsigned short &s) ;
    istream &operator>> (int &i) ;
    istream &operator>> (unsigned int &i) ;
    istream &operator>> (long &l) ;
    istream &operator>> (unsigned long &l) ;
    istream &operator>> (float &f) ;
    istream &operator>> (double &d) ;
    istream &operator>> (long double &d) ;
    istream &operator>> (void *&v) ;

    int get() ;
    istream &get(char &c) ;
    istream &get(char *s, int n, char delim='\n') ;
    istream &getline(char *s, int, char delim='\n') ;

    istream &read(char *s, int n) ;
    istream &ignore(int n=1, int delim=EOF) ;

    int gcount() const ;

    int peek() ;
    istream &putback(char c) ;

    long tellg() ;
    istream &seekg(long &position) ;
    istream &seekg(long &deplacement,
                                  ios_base::seekdir origine) ;
  /* ... */
} ;
```

```
istream &istream::istream::operator>> (char *s) ;
istream &istream::operator>> (char &c) ;
istream &istream::operator>> (bool &b) ;
istream &istream::operator>> (unsigned short &s) ;istream
&istream::operator>> (short &s) ;
istream &istream::operator>> (unsigned int &i) ;
istream &istream::operator>> (int &i) ;
```

```
istream &istream::operator>> (unsigned long &l) ;
istream &istream::operator>> (long &l) ;
istream &istream::operator>> (float &f) ;
istream &istream::operator>> (double &d) ;
istream &istream::operator>> (long double &d) ;
istream &istream::operator>> (void *&v) ;
```

L'opérateur `>>` est défini pour tous les types prédéfinis. Il permet d'extraire des informations formatées du flux d'entrée. Il renvoie le flux courant, ce qui permet l'emploi *en série* de cet opérateur.

Toutes les autres fonctions d'extraction travaillent sur des informations non formatées.

```
int istream::get() ;
```

Cette fonction extrait un caractère s'il y en a un de disponible ou renvoie `EOF` sinon.

```
istream &istream::get(char &c) ;
```

Cette fonction extrait un caractère s'il y en a un de disponible et renvoie `*this`.

```
istream &istream::get(char *s, int n, char delim='\n') ;
```

Cette fonction extrait des caractères et les range à l'adresse `s` jusqu'à ce que l'une des conditions suivantes soit vérifiée :

- `n-1` caractères sont stockés,
- la fin de la séquence d'entrée est rencontrée,
- le prochain caractère à extraire est égal à `delim` (et ce caractère n'est pas extrait).

Un caractère `'\0'` est ajouté à la fin. La fonction renvoie `*this`.

```
istream &istream::getline(char *s, int, char delim='\n') ;
```

Cette fonction est similaire à la fonction précédente, si ce n'est que lorsque le caractère `delim` est rencontré, il est extrait du flux (mais non stocké).

```
istream &istream::read(char *s, int n) ;
```

Cette fonction extrait `n` octets du flux et les stocke à l'adresse `s`.

```
int istream::gcount() const ;
```

Cette fonction fournit le nombre de caractères effectivement lus lors de la dernière opération.

```
istream &istream::ignore(int n=1, int delim=EOF) ;
```

Cette fonction saute par-dessus `n` caractères, sans toutefois dépasser `delim` (qui est extrait dans ce cas).

```
int istream::peek() ;
```

Cette fonction renvoie le prochain caractère sans l'extraire (ou EOF en fin de séquence d'entrée).

```
istream &istream::putback(char c) ;
```

Cette fonction remet un caractère dans le flux et renvoie *this.

```
long istream::tellg() ;
```

Cette fonction renvoie la position courante dans le flux et renvoie *this.

```
istream &istream::seekg(long &position) ;
istream &istream::seekg(long &deplacement,
                                    ios_base::seekdir origine)
```

Ces fonctions effectuent un déplacement dans le flux et renvoient *this.

Manipulateur

```
ws
```

Ce manipulateur saute tous les espaces, tabulations horizontales et verticales, saut de ligne et retour chariot à partir de la position courante, dans le flux concerné.

Exemple

```
int i ;
cin >> hex >> i ;  // entier hexadécimal lu et affecté à i
cin >> setbase(10)
    >> i ;              // entier décimal lu et affecté à i

char c;
c = cin.get() ;    // prochain caractère lu et rangé dans c
cin.get(c) ;       // prochain caractère lu et rangé dans c

char ligne[80] ;
cin.get(ligne,79) ;          // extrait la prochaine ligne

cin.ignore(INT_MAX,'\n') ;    // ignore la fin de la ligne

char tab[100] ;
cin.read(tab,sizeof(tab)) ;              // lit 100 octets
```

```
const int MAX=100 ;
int nb ;
char buf[MAX] ;

while (cin.getline(buf,MAX))      // tant qu'il y a à lire
{
  nb = cin.gcount() ;           // nombre de caractères lus
  /* ... */
}

if (cin.peek()==',') cin >> c ;    // extrait si c'est ','

cin >> c ;
if (c!=',') cin.putback(c) ;  // remet si ce n'est pas ','
```

4 - La classe `ostream`

Cette classe permet d'injecter des informations formatées et non formatées dans un flux de sortie. Elle hérite de `basic_ios`. Elle est définie dans `<ostream>`.

En réalité, `ostream` est une instance du modèle `basic_ostream` :

```
typedef basic_ostream<char> ostream ;
```

et `basic_ostream` dérive du modèle `basic_ios`.

Mais pour simplifier l'écriture, seule la classe `ostream` obtenue par instanciation de `basic_ostream` est décrite ci-dessous.

```
class ostream : virtual public basic_ios
{
  public :
    ostream &operator<< (const char *s) ;
    ostream &operator<< (char c) ;
    ostream &operator<< (bool b) ;
    ostream &operator<< (short s) ;
    ostream &operator<< (unsigned short s) ;
    ostream &operator<< (int i) ;
    ostream &operator<< (unsigned int i) ;
    ostream &operator<< (long l) ;
    ostream &operator<< (unsigned long l) ;
    ostream &operator<< (float f) ;
    ostream &operator<< (double d) ;
    ostream &operator<< (long double d) ;
    ostream &operator<< (void *v) ;

    int put(char c) ;
    ostream &write(const char *s, int n) ;
```

```
    ostream &flush() ;

    long tellp() ;
    ostream &seekp(long pos) ;
    ostream &seekp(long deplacement,
                                ios_base::seekdir origine) ;
  /* ... */
} ;
```

```
ostream &ostream::operator<< (const char *s) ;
ostream &ostream::operator<< (char c) ;
ostream &ostream::operator<< (bool b) ;
ostream &ostream::operator<< (short s) ;
ostream &ostream::operator<< (unsigned short s) ;
ostream &ostream::operator<< (int i) ;
ostream &ostream::operator<< (unsigned int i) ;
ostream &ostream::operator<< (long l) ;
ostream &ostream::operator<< (unsigned long l) ;
ostream &ostream::operator<< (float f) ;
ostream &ostream::operator<< (double d) ;
ostream &ostream::operator<< (long double d) ;
ostream &ostream::operator<< (void *v) ;
```

L'opérateur << est défini pour tous les types prédéfinis. Il insère une information formatée dans le flux de sortie. Il renvoie le flux courant, ce qui permet l'emploi en série de cet opérateur.

Toutes les autres fonctions d'injection travaillent sur des informations non formatées.

```
int ostream::put(char c) ;
```

Cette fonction insère un caractère dans le flux.

```
ostream &ostream::write(const char *s, int n) ;
```

Cette fonction insère n caractères dans le flux, et renvoie `*this`.

```
ostream &ostream::flush() ;
```

Cette fonction vide le tampon de sortie (voir aussi le manipulateur `flush`).

```
long ostream::tellp() ;
```

Cette fonction renvoie la position courante dans le flux et renvoie `*this`.

```
ostream &ostream::seekp(long pos) ;
ostream &ostream::seekp(long deplacement,
                                ios_base::seekdir origine) ;
```

Ces fonctions effectuent un déplacement dans le flux et renvoient `*this`.

Manipulateurs

flush

Ce manipulateur vide le tampon de sortie (voir aussi la fonction `flush`).

endl **ends**

Les manipulateurs `endl` et `ends` insèrent respectivement dans le flux une fin de ligne ('\n') et une fin de chaîne ('\0').

Exemple

```
cout << setw(5) << setfill('0') << 123 ;     // écrit 00123
cout << setprecision(2) << fixed
     << 2./3 << flush ;                       // écrit 0.67
cout << hex << 255 << ' '
     << dec << 255 << endl ;              // écrit "ff 255\n"

cout.width(cout.width()+1) ;
                              // augmente la largeur de sortie

cout.put('A') ;                  // équivalent à cout << 'A'

char s[]="cabernet" ;
cout.write(s,sizeof(s)) ;               // envoie s sur cout
```

5 - La classe `ifstream`

C'est un flux pouvant être connecté à un fichier et offrant des possibilités de lecture. Il hérite de `istream`. Il est défini dans `<fstream>`.

En réalité, `ifstream` est une instance du modèle `basic_ifstream` :

```
typedef basic_ifstream<char> ifstream ;
```

et `basic_ifstream` dérive du modèle `basic_istream`.

Pour simplifier l'écriture, seule la classe `ifstream` issue de l'instanciation de `basic_ifstream` est décrite ci-dessous.

```
class ifstream : public istream
{
  public :
    explicit ifstream() ;
    ifstream(const char *name,
                     ios_base::openmode mode=ios_base::in) ;
```

```
    bool is_open() ;

    void open(const char *name,
                    ios_base::openmode mode=ios_base::in) ;

    void close() ;
  /* ... */
} ;
```

```
ifstream::ifstream() ;
```

Ce constructeur crée un flux non connecté.

```
ifstream::ifstream(const char *name,
                    ios_base::openmode mode=ios_base::in) ;
```

Ce constructeur crée un flux connecté au fichier *name*.

```
bool ifstream::is_open() ;
```

Cette fonction renvoie `true` si le flux est connecté et si le fichier est ouvert.

```
void ifstream::open(const char *name,
                    ios_base::openmode mode=ios_base::in) ;
```

Cette fonction connecte le flux au fichier *name* et l'ouvre.

```
void ifstream::close() ;
```

Cette fonction ferme le fichier.

6 - La classe `ofstream`

C'est un flux pouvant être connecté à un fichier et offrant des possibilités d'écriture. Il hérite de `ostream`. Il est défini dans `<fstream>`.

En réalité, `ofstream` est une instance du modèle `basic_ofstream` :

```
typedef basic_ofstream<char> ofstream ;
```

et `basic_ofstream` dérive du modèle `basic_ostream`.

Pour simplifier l'écriture, seule la classe `ofstream` issue de l'instanciation de `basic_ofstream` est décrite ci-dessous.

```
class ofstream : public ostream
{
  public :
    explicit ofstream() ;
    ofstream(const char *name, ios_base::openmode mode
                    =ios_base::out|ios_base::trunc) ;
```

```
    bool is_open() ;

    void open(const char *name, ios_base::openmode mode
                          =ios_base::out|ios_base::trunc) ;

    void close() ;
  /* ... */
} ;
```

```
ofstream::ofstream() ;
```

Ce constructeur crée un flux non connecté à un fichier.

```
ofstream::ofstream(const char *name, ios_base::openmode mode
                                          =ios_base::out) ;
```

Ce constructeur crée un flux connecté au fichier `name` et l'ouvre en écriture par défaut.

```
bool ofstream::is_open() ;
```

Cette fonction renvoie `true` si le fichier est ouvert.

```
void ofstream::open(const char *name,
    ios_base::openmode mode=ios_base::out|ios_base::trunc) ;
```

Cette fonction connecte le flux au fichier `name` et l'ouvre.

```
void ofstream::close() ;
```

Cette fonction ferme le fichier.

Exemple

```
{
  ifstream f("profile") ;    // ouvre "profile" en lecture
  ofstream g("profile.bis") ;         // ouvre en écriture
  char buf[81] ;

  f.getline(buf,80) ;                      // lit une ligne
                        // '\n' est extrait et non stocké
  while ( f )                          // tant que ça marche
  {
    g << buf << endl ;                     // envoie dans g
    f.getline(buf,80) ;                  // ligne suivante
  }
}
```

7 - La classe `istringstream`

C'est un flux pouvant être connecté à une chaîne et offrant des possibilités d'extraction. Il hérite de `istream`. Il est défini dans `<sstream>`.

En réalité, `istringstream` est une instance du modèle de classe `basic_istringstream` :

```
typedef basic_istringstream<char> istringstream ;
```

et `basic_istringstream` dérive du modèle `basic_istream`.

Pour simplifier l'écriture, seule la classe `istringstream` issue de l'instanciation de `basic_istringstream` est décrite ci-dessous.

```
class istringstream : public istream
{
  public :
    explicit istringstream(
                      ios_base::openmode mode=ios_base::in) ;
    explicit istringstream(const string &str,
                      ios_base::openmode mode=ios_base::in) ;

    string str() const ;
    void str(const string &str) ;
  /* ... */
} ;
```

```
istringstream::istringstream(
                      ios_base::openmode mode=ios_base::in) ;
```

Ce constructeur crée un flux non connecté.

```
istringstream::istringstream(const string &str,
                      ios_base::openmode mode=ios_base::in) ;
```

Ce constructeur crée un flux connecté à la chaîne `str` en mode extraction par défaut.

```
string istringstream::str() const ;
```

Cette fonction renvoie la chaîne connectée au flux.

```
void istringstream::str(const string &str) ;
```

Cette fonction connecte le flux à la chaîne `str`.

8 - La classe `ostringstream`

C'est un flux pouvant être connecté à une chaîne et offrant des possibilités d'injections. Il hérite de `ostream`. Il est défini dans `<sstream>`.

En réalité, `ostringstream` est une instance du modèle de classe `basic_ostringstream` :

```
typedef basic_ostringstream<char> ostringstream ;
```

et `basic_ostringstream` dérive du modèle `basic_ostream`.

Pour simplifier l'écriture, seule la classe `ostringstream` issue de l'instanciation de `basic_ostringstream` est décrite ci-dessous.

```
class ostringstream : public ostream
{
  public :
    explicit ostringstream(ios_base::openmode mode
                                          =ios_base::out) ;
    explicit ostringstream(const string &str,
                    ios_base::openmode mode=ios_base::out) ;

    string str() const ;
    void str(const string &str) ;
  /* ... */
} ;
```

```
ostringstream::ostringstream(ios_base::openmode mode
                                          =ios_base::out) ;
```

Ce constructeur crée un flux non connecté.

```
ostringstream::ostringstream(const string &str,
                    ios_base::openmode mode=ios_base::out) ;
```

Ce constructeur crée un flux connecté à la chaîne `str` en mode injection par défaut.

```
string ostringstream::str() const
```

Cette fonction renvoie la chaîne connectée au flux.

```
void ostringstream::str(const string &str)
```

Cette fonction connecte le flux à la chaîne `str`.

Exemple

```
ostringstream o ;
o.fill('0') ;
o << 2718 << "e-3 " << 6 << '.' << setw(3)
  << 23 << "E23" ;

cout << o.str() << endl ;       // écrit "2718e-3 6.023E23"
```

```
istrstream i(o.str()) ;
float e, N ;

i >> e >> N ;
cout << e << " " << N ;          // écrit "2.718 6.023e+23"
```

— Troisième partie —

C++ en pratique

I - UTILISATION DES REFERENCES

1 - Paramètres de type référence

Considérons p un paramètre de fonction. Si p est destiné à passer une valeur de la fonction appelée à la fonction appelante, alors l'argument correspondant doit être passé par adresse. p est alors un pointeur :

```
void f(int *p) { /* ... */ }
```

ou une référence :

```
void f(int &p) { /* ... */ }
```

Si p n'est destiné qu'à passer une valeur de l'appelante vers l'appelée, alors l'argument peut être passé aussi bien par valeur que par adresse.

Si la taille de p est de l'ordre de celle d'un pointeur, et si le constructeur de recopie de p est trivial (ce qui veut dire que p peut se cloner bit à bit), le coût (mémoire et temps) d'un passage par adresse est comparable au coût d'un passage par valeur. Mais l'accès à la valeur de p sera plus rapide si l'argument est passé par valeur, puisqu'il se fera sans indirection.

Si la taille de p est nettement supérieure à celle d'un pointeur, ou si p se clone par un constructeur de recopie coûteux, alors il est préférable d'utiliser un passage par adresse.

A partir de la classe String et de la fonction f suivante :

```
class String
{
    char *str ;

  public :
```

```
    String(const char *s)
       {
         str = new char[strlen(s)+1] ;
         strcpy(str,s);
       }
    String(const String &s)
       {
         str = new char[strlen(s.str)+1] ;
         strcpy(str,s.str);
       }
    ~String(){ delete[] str ; }
  /* ... */
} ;

void f(String str) {}
```

l'appel :

```
String s("ceci est une chaîne vraiment très "
                                        "très longue...") ;
f(s) ;                                // clonage de s par le
                        // constructeur de recopie de String
```

utilise le constructeur de recopie de `String`, pour recopier l'argument `s` dans `str`. A la sortie de `f`, le destructeur de `String` est appelé pour le paramètre `str`.

Coût d'un appel à cette fonction qui ne fait rien : un appel au constructeur de recopie et un appel au destructeur de `String`, c'est-à-dire une allocation de mémoire, une recopie de tous les caractères et une libération de mémoire.

Un passage par référence est ici préférable :

```
void f(const String &str) {}

String s("ceci est une chaîne vraiment très "
                                        "très longue...") ;
f(s) ;                                // pas de clonage pour s
```

Dans ce cas, seule la référence est recopiée.

L'utilisation de `const`, qui précise que la valeur référencée par `str` est constante, permet d'assurer à l'appelant que la valeur transmise ne sera pas modifiée par la fonction `f`.

2 - Fonctions retournant une référence

i. Gain de temps

Les remarques précédentes s'appliquent de la même manière au retour de fonction. La transmission d'une valeur de retour se fait également par clonage[122]. Si celui-ci est coûteux, le renvoi d'une référence (ou d'un pointeur) peut être envisagé. Le retour de `f(s1)` dans l'exemple suivant :

```
String f(const String &str) { return str ; }

String s1("ceci est une chaîne vraiment très longue"),
       s2("") ;
s2 = f(s1) ;              // clonage de la valeur de retour
```

appelle le constructeur de recopie de `String` à la sortie de `f` pour recopier `str` dans l'objet (temporaire et sans nom) renvoyé par `f`. Puis le destructeur de `String` est appelé à la destruction de cet objet temporaire, une fois l'affectation réalisée.

Coût d'un appel à cette fonction *identité* : un appel au constructeur de recopie et un appel au destructeur de `String`, c'est-à-dire une allocation de mémoire, une recopie des caractères et une libération de mémoire.

Un renvoi par référence est ici préférable :

```
const String &f(const String &str) { return str ; }

String s1("ceci est une chaîne vraiment très longue"),
       s2("") ;
s2 = f(s1) ;                          // pas de clonage ici
```

Dans ce cas, seule la référence est recopiée.

ii. Fonctions à appels chaînables

Outre l'aspect gain de temps, le retour d'une référence peut être employé pour réaliser des fonctions à appels chaînables.

Les fonctions `SelectPolice`, `SelectTaille` et `SelectStyle` suivantes :

[122] Sauf optimisation particulière du compilateur.

```
class Texte
{
  /* ... */
  public :
    enum Style { normal, italique, gras, souligne } ;

    Texte(char *) ;
    Texte &SelectPolice(char *)
                              { /* ... */ return *this ; }
    Texte &SelectTaille(int t)
                              { /* ... */ return *this ; }
    Texte &SelectStyle(Style s)
                              { /* ... */ return *this ; }
    const char *Val() const ;
} ;
```

renvoient une référence sur l'objet courant. Leurs appels peuvent par conséquent être chaînés :

```
Texte t("Coteaux du Layon") ;
t.SelectPolice("Times").SelectTaille(10)
 .SelectStyle(Texte::normal) ;
```

Sans ce renvoi de référence, il aurait été nécessaire d'écrire :

```
t.SelectPolice("Times") ;
t.SelectTaille(10) ;
t.SelectStyle(Texte::normal) ;
```

Cette technique est souvent utilisée pour les opérateurs d'entrée/sortie. L'opérateur d'injection d'un `Texte` dans un flux :

```
ostream &operator<<(ostream &o, const Texte &t)
{
  return o << t.Val() ;
}
```

renvoie une référence sur le flux sollicité (la fonction membre `ostream::operator<<(char *)` renvoie en effet une référence sur l'objet courant). L'opérateur ci-dessus renvoie donc une référence sur le paramètre `o`, ce qui est équivalent à :

```
ostream &operator<<(ostream &o, Texte t)
{
  o << t.Val() ;
  return o ;
}
```

Cela permet l'écriture d'expressions comme :

```
cout << t1 << t2 << t3 << '\n' ;
```

évaluée comme :

```
(((cout << t1) << t2) << t3) << '\n' ;
```

où chaque appel à << renvoie une référence sur le premier opérande, c'est-à-dire ici le flux cout.

iii. Fonctions de lecture/écriture d'objets

Le retour d'une référence peut s'utiliser pour réaliser des fonctions de consultation/modification de l'état d'un objet. La fonction operator[] suivante :

```
template <class T, int t>
class Tab
{
    T tab[t] ;

  public :
    T &operator[](int i)
    { if (i>=t)
        throw out_of_range("indice trop grand") ;
      return tab[i] ;
    }
} ;
```

permet des accès en consultation et modification aux éléments du tableau :

```
Tab<int,100> tab ;
int i=tab[1] ;                    // accès en consultation
tab[1] = 1 ;                      // accès en modification
```

Le problème d'une telle fonction, c'est qu'elle ne peut pas être utilisée sur des objets constants, même pour des accès en consultation :

```
template <class T, int t>
void ecrire(const Tab<T,t> &tab)
{
  for (int i=0 ; i<t ; i++)
    cout << tab[i] ;            // illégal, mais injustifié
}
```

En effet, cette fonction n'est pas déclarée const, et ne peut d'ailleurs pas l'être, puisqu'elle offre la possibilité de modifier l'objet courant. Elle ne peut par conséquent pas être appelée à partir d'un objet constant.

Une amélioration consiste à différencier l'accès en lecture et l'accès en écriture, en utilisant le fait que le spécificateur const est significatif pour la résolution des surcharges :

```
template <class T, int t>
class Tab
{
    T tab[t] ;

  public :
    T &operator[](int i)
    { if (i>=t)
        throw out_of_range("indice trop grand") ;
      return tab[i] ;
    }
    const T &operator[](int i) const
    { if (i>=t)
        throw out_of_range("indice trop grand") ;
      return tab[i] ;
    }
} ;
```

Maintenant, si le tableau est constant, c'est la fonction `const T &operator[](int) const` qui est appelée. Seule une consultation est alors permise :

```
template <class T, int t>
void g(const Tab<T,t> &tab)
{
  cout << tab[0] ;          // ok, appel de operator[] const
  tab[0]++ ;                  // appel de operator[] const :
                                  // illégal, mais justifié
}
```

Si le tableau n'est pas constant, c'est la fonction `T &operator[](int)` qui est appelée, et une modification est permise :

```
template <class T, int t>
void f(Tab<T,t> &tab)
{
  cout << tab[0] ;     // ok, appel de operator[] non const
  tab[0]++ ;           // ok, appel de operator[] non const
}
```

iv. Déjouer les pièges du renvoi de références

La première précaution à prendre, en ce qui concerne le renvoi de références, c'est de ne jamais renvoyer une référence sur un objet automatique[123]. En effet,

[123] `auto` est la classe d'allocation par défaut pour les objets locaux et les paramètres de fonction, dont la durée de vie est fonction du code exécuté.

celui-ci est détruit à la sortie de la fonction. L'appelant récupère donc une référence sur un objet qui n'existe plus. Les conséquences sont indéterminées.

L'exemple suivant définit une classe matrice `t1×t2` d'éléments de type `T`, ainsi qu'un opérateur d'addition :

```
template <class T, int t1, int t2>
class Matrice
{
    T mat[t1][t2] ;
  /* ... */
} ;

template <class T, int t1, int t2>
Matrice<T,t1,t2> &operator+(Matrice<T,t1,t2> m1,
                            Matrice<T,t1,t2> m2)
{
  Matrice<T,t1,t2> m3 ;

  for (int i=0 ; i<t1 ; i++)
    for (int j=0 ; j<t2 ; j++)
      m3.mat[i][j] = m1.mat[i][j]+m2.mat[i][j] ;

  return m3 ;                                  // Houlàlà !
}
```

Le résultat de l'addition est contenu dans une variable `m3` automatique, la référence renvoyée est donc invalide. Un clonage doit être effectué lors du renvoi de la matrice. Et pour cela, il suffit de modifier le type de retour (en enlevant la référence) :

```
template <class T, int t1, int t2>
Matrice<T,t1,t2> operator+(Matrice<T,t1,t2> m1,
                           Matrice<T,t1,t2> m2)
{
  Matrice<T,t1,t2> m3 ;
  /* ... */
  return m3 ;                    // clonage nécessaire ici
}
```

Ce piège existe déjà en C sous une autre forme : le renvoi d'un pointeur sur une variable locale[124]. Cette forme subsiste toujours en C++.

Pour éviter ce piège, il suffit de s'assurer que l'objet référencé subsiste hors de la fonction. Par exemple, il n'y a pas de problème à renvoyer une référence sur un paramètre qui est lui-même une référence :

[124] Voir [FLR] et [CTP] sur les pièges du langage C.

```
int &f(int &x)
{
  return x ;                                    // pas de problème
}
```

En effet, `x` est une référence sur un objet externe à `f`, dont la durée de vie est par conséquent indépendante de `f`. Par contre, attention :

```
int &f(int x)
{
  return x ;                                    // eh non !
}
```

cette fonction pose problème, puisque `x` a la même durée de vie que `f`.

La seconde précaution à prendre, en ce qui concerne le renvoi de références, c'est de ne pas oublier que la valeur renvoyée est une lvalue. Par conséquent il est possible que l'appelant prenne l'adresse de l'objet renvoyé, et la conserve un certain temps. Il ne faut donc pas déplacer cet objet tant qu'il est susceptible d'être référencé.

Dans la classe `Stack` suivante :

```
template <class T>
class Stack
{
  public :
   T &Top() ;                  // consulter/modifier le sommet
   T Pop() ;          // dépiler et renvoyer la valeur dépilée
  /* ... */
} ;
```

la fonction `Top` accède en consultation/modification au sommet de la pile. `Pop` renvoie le sommet et l'enlève de la pile. Attention, cette classe est dangereuse ! Gare aux utilisations du style :

```
Stack<int> p ;
/* ... */
p.Top() = p.Pop()+1 ;                           // ?@#~^!
```

En effet, dans cette dernière instruction, si le terme de gauche de l'affectation est évalué en premier, c'est râpé : le terme de gauche est évalué, puis le terme de droite est évalué, mais cette dernière évaluation rend obsolète le résultat de la première évaluation ! L'affectation est désastreuse...

De manière générale, dans une classe conteneur, si les éléments sont susceptibles d'être déplacés, réagencés sans avertissement par des mécanismes automatiques d'optimisation, une réflexion s'impose avant tout renvoi de références sur les objets mémorisés.

3 - Conclusion

Même si quelques pièges gravitent autour de la manipulation des références, leur utilisation est moins dangereuse que celle des pointeurs. En particulier, contrairement aux pointeurs, les références sont toujours initialisées. Il n'y a pas non plus, pour les références, de valeur spéciale équivalente au pointeur `NULL`, source parfois de soucis.

Ceci dit, la manipulation des références mérite quand-même une certaine attention.

II - LA GESTION DYNAMIQUE DE LA MEMOIRE

1 - Allocation dynamique de tableaux

C++ dispose de deux mécanismes d'allocation de mémoire, l'un pour allouer des objets uniques, l'autre pour allouer des tableaux d'objets.

Le premier s'exprime avec les opérateurs `new` et `delete` :

```
MonType *ptr = new MonType ;
/* ... */
delete ptr ;
```

le second s'exprime avec les opérateurs `new[]` et `delete[]` :

```
MonType *ptr = new MonType[10] ;
/* ... */
delete[] ptr ;
```

Les deux expressions `new MonType` et `new MonType[10]` sont toutes les deux de même type, à savoir de type `MonType *` : dans le premier cas, on obtient un pointeur sur l'objet alloué, et dans le second, on obtient un pointeur sur le premier élément du tableau alloué.

Cela peut avoir des conséquences inattendues :

```
typedef MonType Tab[10] ;

Tab *tab = new Tab ;                    // non, erreur de type
```

Cette expression semble correcte, or elle pose un problème de type. En effet, `new Tab` étant équivalent à `new MonType[10]`, l'expression `new Tab` est de type `MonType *`. Il faut par conséquent écrire :

```
MonType *tab = new Tab ;                          // d'accord
```

et on obtient alors un pointeur sur le premier objet du tableau.

Rien ne garantit que les mécanismes mis en œuvre par les opérateurs `new` et `delete` d'une part, et par les opérateurs `new[]` et `delete[]` d'autre part, soient les mêmes : l'allocation d'un tableau peut nécessiter la mémorisation du nombre d'éléments alloués, information qui pourra être utile pour appeler le destructeur de

chacun des objets alloués lors de la libération du tableau. Par conséquent, attention à ne pas mélanger l'utilisation des opérateurs `delete` et `delete[]` :

```
MonType *ptr = new MonType[10] ;
/* ... */
delete ptr ;                                          // Aïe
```

Il aurait fallût écrire :

```
delete[] ptr ;
```

Les conséquences sont imprévisibles !

La règle consistant à dire : « tout ce qui est alloué par `new[]` doit être libéré par `delete[]` et tout ce qui est alloué par `new` doit être libéré par `delete` » est fausse. La réalité est plus subtile que cela :

```
MonType *tab = new Tab ;
/* ... */
delete tab ;
```

Ce code semble correct. Or il faut écrire :

```
MonType *tab = new Tab ;
/* ... */
delete[] tab ;
```

puisque `Tab` est un tableau. L'erreur est facile à commettre !

Le raisonnement est le même pour les tableaux multidimensionnels. On peut se demander quel doit être le type de `ptr` dans l'expression :

```
ptr = new MonType[20][10] ;
```

Cette instruction crée un tableau de `20` éléments de type `MonType[10]`, donc `ptr` est de type pointeur vers `MonType[10]`. `ptr` doit donc être déclaré comme :

```
MonType (*ptr)[10] = new MonType[20][10] ;
```

ce qui peut s'écrire, en utilisant le type `Tab` précédent :

```
Tab *ptr = new Tab[20] ;
```

2 - Détection des échecs d'allocations de mémoire

La première façon de gérer les erreurs d'allocations de mémoire est d'intercepter l'exception `bad_alloc`[125] :

[125] Voir *bad_alloc* page 192

```
void f(int nb)
{
  try
  {
    int *tab = new int[nb] ;
    /* ... */
  }
  catch(bad_alloc ex)
  {
    cout << ex.what() ;
    terminate() ;
  }
}
```

La seconde façon de gérer les erreurs d'allocations de mémoire est d'*installer* une fonction d'interception des échecs d'allocation. Une fois installée, cette fonction d'interception sera automatiquement appelée à chaque fois que new ne pourra pas satisfaire l'allocation demandée.

La fonction prédéfinie set_new_handler, déclarée dans <new> comme :

```
new_handler set_new_handler(new_handler) ;
```

avec new_handler défini comme :

```
typedef void (*new_handler)() ;
```

permet d'installer une fonction d'interception des erreurs d'allocation mémoire.

L'appel :

```
set_new_handler(0) ;
```

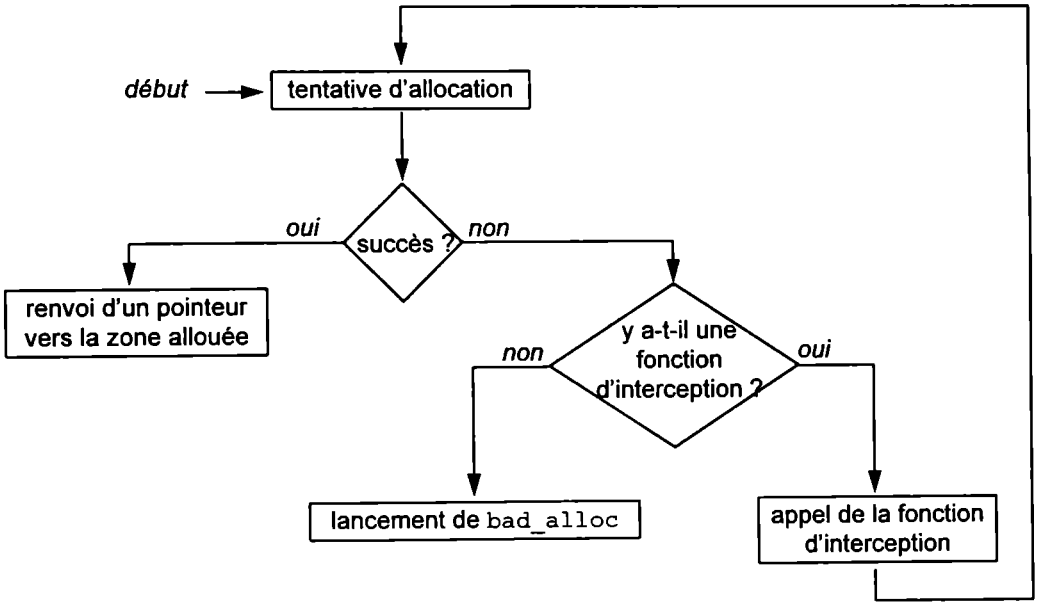
désinstalle la fonction d'interception courante, sans en installer de nouvelle.

set_new_handler renvoie un pointeur sur la fonction d'interception installée précédemment :

```
void MemoireSaturee()
{
  cerr << "mémoire insuffisante" ;
  terminate() ;
}

void f()
{
  new_handler ancien_handler ;
  ancien_handler = set_new_handler(MemoireSaturee) ;
  /* ... */
  // les échecs d'allocation sont gérés ici...
  // ... par la fonction MemoireSaturee
  /* ... */
  set_new_handler(ancien_handler) ;
}
```

A chaque fois qu'une allocation échoue dans la fonction `f`, le message `mémoire insuffisante` est affiché et l'exécution est interrompue. La fonction d'interception active à l'entrée de la fonction est restaurée à la sortie.

Dans l'exemple précédent, la fonction d'interception interrompt l'exécution du programme. Mais rien n'empêche d'envisager un traitement plus sophistiqué, visant à récupérer de la mémoire. Si la fonction d'interception rend la main à l'appelant, une nouvelle allocation est tentée, et cela jusqu'à ce que l'allocation réussisse, ou bien jusqu'à ce que la fonction d'interception interrompe le programme. L'algorithme réalisé peut être schématisé de la façon suivante :




```
void MemoireSaturee()
{
  static int nb_essai=0 ;

  switch (nb_essai++)
  {
    case 1 : cerr << "compactage du tas...\n" ;
             /* ... */
             break ;
    case 2 : cerr << "échange avec une page "
                                    " en mémoire...\n" ;
             /* ... */
             break ;
```

```
      case 3 : cerr << "échange avec une page "
                                          " sur disque...\n" ;
               /* ... */
               break ;
      case 4 : cerr << "mémoire insuffisante\n" ;
               terminate() ;
    }
  }

  void main()
  {
    set_new_handler(MemoireSaturee) ;
    int *ptr ;

    ptr = new int[1000] ;
    /* ... */
  }
```

Dans ce programme, si la mémoire disponible est insuffisante pour satisfaire l'allocation, plusieurs essais de récupération sont tentés. Et à chaque retour de `MemoireSature`, l'allocation est réessayée. Par exemple, si l'allocation `new int[1000]` n'aboutit pas du premier coup, toutes les stratégies prévues pour récupérer de la mémoire vont être tentées. Si aucune ne permet de satisfaire l'allocation, le programme s'arrête après avoir écrit :

```
compactage du tas...
échange avec une page en mémoire...
échange avec une page sur disque...
mémoire insuffisante
```

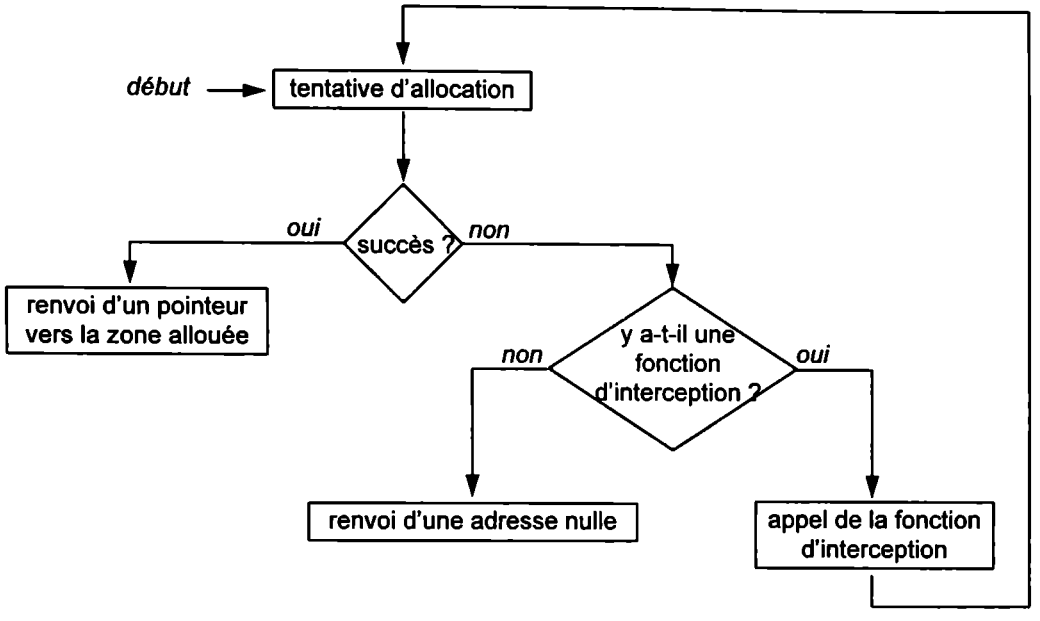
A signaler également l'existence de deux opérateurs `new` de placement :

```
void *operator new(size_t, const nothrow &) throw() ;
void *operator new[](size_t, const nothrow &) throw() ;
```

qui renvoient `0` en cas d'échec, plutôt que de lancer `bad_alloc` :

```
void f(int nb)
{
  int *tab = new(nothrow()) int[nb] ;
  if (tab==0)
  {
    /* ... */
    terminate() ;
  }
  /* ... */
}
```

Leur comportement peut se schématiser comme :


début
tentative d'allocation
succès ?
oui
non
renvoi d'un pointeur vers la zone allouée
y a-t-il une fonction d'interception ?
non
oui
renvoi d'une adresse nulle
appel de la fonction d'interception

III - CONCEPTION DE CLASSES

Les classes doivent être utilisées dans l'optique de la programmation par objets, notamment en ce qui concerne l'*encapsulation* et l'*abstraction des données*.

L'encapsulation est réalisée en regroupant dans une même classe les structures de données qui l'implémentent (matérialisées par les variables et constantes membres) et les fonctions qui les manipulent.

L'abstraction des données est obtenue en réalisant pour chaque classe une partie privée et une partie publique. La partie privée contient les structures de données et les fonctions à usage interne. La partie publique contient les fonctions, types, constantes offerts aux utilisateurs de la classe.

On appelle interface de la classe ce qui est accessible par tout le monde, et implémentation ce qui n'est accessible que par le concepteur de la classe.

1 - Isoler l'interface de l'implémentation

Plus l'indépendance est grande entre l'interface d'une classe et son implémentation, plus l'utilisation de la classe s'affranchit des détails de son organisation interne.

L'interface et l'implémentation de la classe :

```
class Pixel
{
    short x, y ;
    unsigned color ;

  public :
    short &X() { return x ; }
    short &Y() { return y ; }
    unsigned &Color() { return color ; }
} ;
```

sont fortement liées : si le concepteur est amené à modifier ses structures de données, l'utilisateur a de fortes chances de s'en trouver perturbé.

Supposons qu'il soit plus intéressant de mémoriser la position d'un pixel sous forme d'un entier égal à `x+1024xy`. Il est impossible d'apporter une telle modification dans l'implémentation sans une modification importante de

l'interface. Cela est dû à la présence des fonctions de consultation/modification `X` et `Y`, qui couplent fortement l'interface et l'implémentation.

Au contraire, dans la classe `Pixel` suivante :

```
class Pixel
{
    short x, y ;
    unsigned color ;

  public :
    void Init(short xx, short yy, unsigned c)
                              { x = xx; y = yy; color = c; }
    short GetX() { return x ; }
    short GetY() { return y ; }
    unsigned GetColor() { return color ; }
} ;
```

l'indépendance interface/implémentation est bien plus importante. Les fonctions de consultation/modification `X` et `Y` ont été remplacées par deux fonctions de consultation `GetX` et `GetY` d'une part, et une fonction de modification `Init` d'autre part.

Il est ici possible d'apporter la modification souhaitée de façon transparente pour l'utilisateur de la classe :

```
class Pixel
{
    long p ;
    unsigned color ;

  public :
    void Init(short xx, short yy, unsigned c)
                              { p = xx+1024*yy; color = c;}
    short GetX() { return p%1024 ; }
    short GetY() { return p/1024 ; }
    unsigned GetColor() { return color ; }
} ;
```

2 - Séparer physiquement interface et implémentation

La partie privée d'une classe n'est accessible que par l'auteur de la classe. Mais cette restriction d'accès peut être modifié par tout utilisateur ! En effet, l'utilisateur doit disposer de la définition de la classe pour pouvoir l'utiliser. Si cette définition est livrée sous forme d'un fichier en-tête, rien n'empêche l'utilisateur de modifier l'emplacement des mots clés `private`, `protected` et `public`.

De même, l'auteur de la classe peut cacher ses algorithmes : il suffit qu'il définisse les fonctions hors du fichier en-tête, et qu'il diffuse seulement le code compilé de ces fonctions. Mais il ne peut pas rendre confidentiel ses structures de données : la partie privée reste lisible par tout utilisateur.

A la lecture des structures de données de la classe `String` suivante :

```
class String
{
    char *str ;
    int len, nb_block ;
    const int size_block ;

  public :
    /* ... */
} ;
```

on devine que l'allocation de la mémoire se fait par blocs !

Pour rendre confidentielle l'implémentation d'une classe, il est nécessaire de la séparer physiquement de l'interface. L'auteur peut concevoir l'implémentation sous la forme d'une classe imbriquée, et la définir dans un fichier dont seul le code compilé sera diffusé. La seule donnée membre de la classe initiale se réduit alors à un pointeur sur une instance de son implémentation.

La définition de la classe `String` dans un fichier en-tête `string.hpp`, conçu pour être diffusé, a alors la forme suivante :

```
class String
{
    class Implementation ;
    Implementation *impl ;

  public :
    /* ... */
    int Len() ;
    String Substr(int pos, int len) ;
    /* ... */
} ;
```

La définition de la classe `Implementation` ainsi que les fonctions de `String` sont définies hors du fichier en-tête, dans un fichier dont seule la version compilée sera distribuée :

```
#include "string.hpp"

class String::Implementation
{
    char *str ;
    int len, nb_block ;
    const int size_block ;

  public :
    /* ... */
    int Len() { /* ... */ }
    String Substr(int pos, int len) { /* ... */ }
    /* ... */
} ;

int String::Len()
{
  return impl->Len() ;
}

String String::Substr(int pos, int len)
{
  return impl->Substr(pos,len) ;
}
```

Outre une confidentialité inviolable de l'implémentation, cette technique permet :

— de réduire les temps de compilation : une modification apportée à l'implémentation de la classe ne nécessite pas de recompilation des sources de l'utilisateur (puisque l'en-tête `string.hpp` n'est pas modifié) ;

— de pouvoir disposer de plusieurs versions de la bibliothèque, correspondant à différentes implémentations, et de sélectionner à l'édition des liens la version à utiliser en fonction de ses performances vis-à-vis du problème traité.

IV - COMPLEMENTS SUR LES CONSTRUCTEURS ET DESTRUCTEURS

1 - Constructeurs et performances

L'initialisation des objets membres peut être à l'origine d'une consommation excessive de temps d'exécution. A partir des classes `String` et `Personne` suivantes :

```
class String
{
    char *str ;

  public :
    String(const char *s="")
      {
        str = new char[strlen(s)+1];
        strcpy(str,s);
      }
    String(const String &s)
      {
        str = new char[strlen(s.str)+1];
        strcpy(str,s.str);
      }
    ~String() { delete[] str ; }
    String &operator=(const String &s)
      {
        if (&s!=this)
        {
          delete str ;
          str = new char[strlen(s.str)+1];
          strcpy(str,s.str);
        }
      }
    /* ... */
} ;
```

```
class Personne
{
    String nom, prenom ;

  public :
    Personne(const String &n, const String &p) ;
    Personne(const char *n, const char *p) ;
  /* ... */
} ;
```

on peut implémenter naïvement le premier constructeur `Personne` de la façon suivante :

```
Personne :: Personne(const String &n, const String &p)
{
  nom = n ;
  prenom = p ;
  /* ... */
}
```

Cette implémentation est tout à fait valide, car la classe `String` a un constructeur par défaut. Mais elle est très pénalisante. En effet, le compilateur va être amené à créer les deux membres `nom` et `prenom` à l'aide du constructeur par défaut de `String` avant d'entrer dans le constructeur de `Personne`, puisque les objets englobés sont construits avant l'objet englobant. Par conséquent, pour chacun des deux membres, une allocation de `1` caractère va être faite, suivie par une recopie du caractère `'\0'`. Ensuite, au début du constructeur de `Personne`, l'affectation va générer pour chacun des deux membres une libération de mémoire, une réallocation, suivie par une recopie des caractères de la `String` d'origine[126].

Coût total de la construction d'une instance de `Personne` : quatre allocations de mémoire, quatre recopies de chaînes de caractères et deux libérations de mémoire !

C'est pire pour le second constructeur, s'il est implémenté comme :

```
Personne :: Personne(const char *n, const char *p)
{
  nom = n ;
  prenom = p ;
  /* ... */
}
```

Cette implémentation valide est encore plus pénalisante. Ici, le compilateur va être amené, comme précédemment, à créer d'abord les deux membres `nom` et `prenom` à l'aide du constructeur par défaut de `String`. Ensuite, chacune des deux affectations va convertir son opérande de droite de type `char *` en `String`, ce qui crée deux objets temporaires à l'aide du constructeur de conversion

[126] Bien sûr, un compilateur assez futé peut corriger ce genre de maladresses.

`String(char*)`. Puis une affectation va être réalisée pour chacun des membres. Enfin, le destructeur va être appelé pour chacun des deux objets temporaires créés. Coût total de la construction : quatre appels au constructeur par défaut de `String`, deux appels à l'opérateur d'affectation et deux appels au destructeur de `String`, soit au total : six allocations de mémoire, six recopies de chaînes de caractères et quatre libérations de mémoire. Quel gâchis !

Pour éviter ce gaspillage, il faut implémenter les constructeurs de la façon suivante :

```
Personne :: Personne(const String &n, const String &p)
                                          : nom(n), prenom(p)
{
  /* ... */
}

Personne :: Personne(const char *n, const char *p)
                                          : nom(n), prenom(p)
{
  /* ... */
}
```

Le premier ne fait plus qu'un seul appel au constructeur de recopie de chacun des deux membres `nom` et `prenom`, soit deux allocations et deux recopies de chaînes de caractères.

Le second ne fait plus qu'un seul appel au constructeur par défaut de chacun des deux membres, soit là aussi deux allocations et deux recopies de chaînes de caractères.

Conclusion : l'acquisition de quelques réflexes, comme celui d'initialiser les objets membres sur l'en-tête du constructeur de l'objet englobant, technique qui d'ailleurs peut être généralisée à tout type d'objet, y compris ceux de types prédéfinis, peut devenir rapidement avantageuse.

2 - Constructeurs et héritage

Une classe dérivée n'hérite pas des constructeurs de sa ou ses classes de base directes. Cette règle doit être prise en compte lors de la conception de classes dérivées, sous peine de créer des classes inutilisables.

Dans la suite, on appelle *objet standard* une instance créée par le constructeur par défaut.

```
class Quadrilatere
{
    /* ... */
  public :
    Quadrilatere(Point p1, Point p2, Point p3, Point p4) ;
} ;
```

```
class Trapeze : public Quadrilatere
{
    // classe sans constructeur défini explicitement
} ;
```

Ici, il est impossible de créer des instances de `Trapeze`. En effet, la déclaration :

```
Trapeze t1 ;                                    // illégal
```

est illégale, car `Quadrilatere` n'a pas de constructeur par défaut (le constructeur par défaut de `Trapeze` cherche à appeler le constructeur par défaut de `Quadrilatere`, mais celui-ci n'existe pas). Et la déclaration :

```
typedef pair<int,int> Point ;
Point p1(1,2), p2(3,4), p3(5,6), p4(7,8) ;

Trapeze t2(p1,p2,p3,p4) ;                       // illégal
```

n'est pas plus légale, puisque `Trapeze` n'a pas de constructeur à quatre paramètres. La classe `Trapeze` est donc inutilisable.

Si on ajoute un constructeur par défaut dans `Quadrilatere` :

```
class Quadrilatere
{
    /* ... */
  public :
    Quadrilatere() ;
    Quadrilatere(Point p1, Point p2, Point p3,
                                          Point p4) ;
} ;

class Trapeze : public Quadrilatere
{
    // classe sans constructeur défini explicitement
} ;
```

des trapèzes standard peuvent alors être créés :

```
Trapeze t1 ;                                    // oui
```

mais il n'est toujours pas possible de créer des `Trapeze` non standard :

```
Trapeze t2(p1,p2,p3,p4) ;                       // non
```

Si maintenant chacune des classes `Quadrilatere` et `Trapeze` est munie d'un constructeur à quatre paramètres :

```
class Quadrilatere
{
  public :
    Quadrilatere(Point p1, Point p2, Point p3, Point p4) ;
} ;

class Trapeze : public Quadrilatere
{
  public :
    Trapeze(Point p1, Point p2, Point p3, Point p4)
                                                  {} //illégal
} ;
```

alors il redevient impossible de construire des trapèzes standard :

```
Trapeze t1 ;                                          // illégal
```

puisque `Trapeze` n'a plus de constructeur par défaut, et il n'est toujours pas possible de créer des `Trapeze` non standard :

```
Trapeze t2(p1,p2,p3,p4) ;                             // non
```

En effet, le constructeur de `Trapeze` cherche à appeler le constructeur par défaut de `Quadrilatere`, puisqu'aucun appel spécifique n'est explicitement spécifié sur l'en-tête de ce constructeur. Or ce constructeur par défaut n'existe pas. Là encore, `Trapèze` est inutilisable.

La solution consiste à appeler explicitement le constructeur de la classe de base :

```
class Quadrilatere
{
  public :
    Quadrilatere(Point p1, Point p2, Point p3,
                                                  Point p4) ;
} ;

class Trapeze : public Quadrilatere
{
  public :
    Trapeze(Point p1, Point p2, Point p3, Point p4)
                : Quadrilatere(p1,p2,p3,p4) {/* ... */} ;
} ;

Trapeze t2(p1,p2,p3,p4) ;                             // ok
```

Les destructeurs ne sont pas non plus transmis par héritage, mais ces problèmes ne se posent pas. En effet, seul un destructeur sans paramètre peut être défini par classe, et s'il n'y est pas, tout se passe comme si un destructeur sans action était synthétisé.

3 - Absence de synthétisation

Si une classe `C` ne définit aucun constructeur, alors tout se passe généralement comme si un constructeur par défaut était synthétisé par le compilateur. L'en-tête de ce constructeur est alors de la forme `C()`.

Mais cette synthétisation n'a pas lieu lorsque la classe possède :

— une constante membre, ou
— un membre non statique de type référence, ou
— un objet membre non statique avec un constructeur par défaut inaccessible, ou
— une classe de base avec un constructeur par défaut inaccessible.

De même, si une classe ne définit pas de destructeur, tout se passe généralement comme si un destructeur était synthétisé par le compilateur. Mais cette synthétisation n'a pas lieu lorsque la classe possède :

— un objet membre non statique avec un destructeur inaccessible, ou
— une classe de base avec un destructeur inaccessible.

Dans la classe `B` suivante, aucun destructeur n'est synthétisé :

```
class A
{
  ~A() {}
} ;

class B
{
  A a ;
} ;

void main()
{
  B b ;

  /* ... */
}                                                    // erreur
```

V - FONCTIONS MEMBRES : CONSTANTES OU PAS CONSTANTES ?

Une fonction membre constante est une fonction qui ne modifie pas l'état de l'objet pour lequel elle est appelée.

Le compilateur vérifie dans une certaine mesure que les instructions de la fonction respectent cet état de fait. Le pointeur `this`, par qui se fait l'accès aux membres de l'objet, est de type `const C * const` dans une fonction constante, ce qui interdit toute modification par son intermédiaire. Il est de type `C * const` dans une fonction non constante :

```
class C
{
  int x ;
  void f() { x++ ; }          // ok, this de type C * const
  void g() const { x++ ; }          // non, this de type
                                      // const C * const
} ;
```

Remarquons au passage que, dans tous les cas, `this` est un pointeur constant, ce qui interdit sa modification :

```
void C::f() { this++ ; }                // toujours incorrect
```

Bien que sans intérêt, ces contrôles peuvent être sciemment supprimés par un simple *cast* ou un `const_cast` :

```
void C::g() const { ((C *) this)->x++ ; }
void C::f() { ((C *)this)++ ; }          // forcément légal,
                                          // mais désastreux
```

Une fonction constante peut néanmoins modifier les variables mutables de l'objet :

```
class C
{
  mutable int x ;
  void g() const { x++ ; }                // ok, x est mutable
} ;
```

Une fonction membre, qui n'est ni un constructeur, ni un destructeur, peut être appelée pour un objet constant seulement si c'est une fonction constante. Cette restriction suffit en effet pour assurer que l'objet reste bien constant entre sa construction et sa destruction.

Les vérifications du compilateur peuvent être relativement fines, lorsqu'il s'agit de contrôler qu'une fonction déclarée constante l'est bien effectivement :

```
class C
{
    int x ;

  public :
    C() {}
    int &f1() const { return x ; }              // illégal
    int *f2() const { return &x ; }             // illégal
    const int &f3() const { return x ; }           // oui
    const int *f4() const { return &x ; }          // oui
} ;
```

En effet, par `f1`, il serait possible de faire des manipulations du type :

```
const C c ;
c.f1()++ ;
```

et par `f2` :

```
(*(c.f2()))++ ;
```

Par conséquent, il est interdit ici de déclarer `f1` et `f2` constantes.

Mais le compilateur ne peut pas tout vérifier, notamment lorsque l'état de l'objet n'est pas complètement stocké dans ses membres. Comment savoir si une zone de mémoire référencée par un pointeur membre implémente ou non l'état de l'objet :

```
class C
{
  char *s ;

  C() { s = new char[20] ; }
  void g() const { strcpy(s,"Plantagenêt") ; }    // légal
} ;
```

La fonction `g` ne modifie pas le membre `s` (le pointeur a la même valeur avant et après l'appel à `strcpy`), mais l'objet courant varie-t-il ? C'est très subjectif, et le compilateur ne peut rien imposer : `g` peut donc très bien être une fonction constante.

Par convention, l'absence de fonction constante dans une classe `C` signifie que celle-ci ne supporte pas les objets constants. Aucune opération n'est réalisable sur une constante de type `C` :

```
class C
{
  public :
    C() ;
    C(C &) ;                          // constructeur de recopie
    C operator=(C &) ;
    bool operator==(C &) ;
    int Etat() ;
} ;

const C c ;                        // ok pour créer une constante
```

Ici, bien que la création d'une constante de type `C` soit possible, aucune opération ne lui est applicable. Même sa duplication (clonage et affectation de recopie) est impossible :

```
const C ct ;
C c1 ;
c1 = ct ;                                                    // non
C c2(ct) ;                                                   // non
```

Inversement, la présence de fonctions constantes dans une classe signifie que son concepteur annonce son intention de supporter les objets constants. Mais dans ce cas, il a le devoir de supporter aussi les pointeurs vers les objets constants ainsi que les références sur les objets constants : si ce choix est pris, le support doit être complet. Ce n'est pas le cas de l'exemple suivant :

```
class C
{
  public :
    C() ;                             // constructeur par défaut
    C(C &) ;                          // constructeur de recopie
    C operator=(C &) ;
    bool operator==(C &) ;
    int Etat() const ;
} ;
```

Ici, la classe est censée supporter les objets constants, puisque la fonction `Etat` est déclarée constante. Mais le support est incomplet. En effet, il est toujours impossible de dupliquer un objet constant. Même la comparaison d'objets constants est impossible :

```
if (c1==ct) /* ... */                                        // illégal
```

Il est donc nécessaire de prendre en compte les constantes partout où elles peuvent apparaître, ce qui donne :

```
class C
{
  public :
    C() ;                          // constructeur par défaut
    C(const C &) ;                 // constructeur de recopie
    C operator=(const C &) ;
    bool operator==(const C &) ;
    int Etat() const ;
} ;
```

Cela permet d'écrire :

```
const C ct ;
C c1 ;
c1 = ct ;                                                  // oui
C c2(ct) ;                                                 // oui
if (c1==ct) /* ... */                                      // oui
```

Est-ce complet ? Non, c'est encore insuffisant ! La comparaison :

```
if (ct==c1) /* ... */     // toujours illégal dans ce sens
```

n'est toujours pas valide dans ce sens, car la fonction `operator==` n'est pas déclarée constante.

Il est donc également nécessaire de déterminer toutes les fonctions qui doivent être spécifiées constantes :

```
class C
{
  public :
    C() ;                          // constructeur par défaut
    C(const C &) ;                 // constructeur de recopie
    C operator=(const C &) ;
    bool operator==(const C &) const ;
    int Etat() const ;
} ;

if (ct==c1) /* ... */                                      // oui
```

Le choix de supporter ou non les objets constants pour une classe donnée détermine quasiment cette option pour les classes dérivées :

```
class A
{
  protected :
    int i ;
    int &Etat() { return i ; }   // fonction non constante
} ;
```

```
class B : public A
{
    int ConsulterEtat() const
      { return Etat() ; }                          // erreur
} ;
```

Ici, la fonction constante `ConsulterEtat` ne peut pas appeler la fonction non constante `Etat`. `ConsulterEtat` ne peut donc pas être déclarée constante. Donc, ou bien on laisse tomber l'idée de déclarer `ConsulterEtat` constante, ou bien on modifie la classe `A` en déclarant `Etat` constante. Mais telle quelle, la fonction `Etat` ne peut pas être déclarée constante, puisqu'elle renvoie une référence vers un entier non constant. Il est nécessaire de séparer l'opération de consultation et celle de modification. On obtient alors deux fonctions, dont l'une des deux est constante :

```
class A
{
  protected :
    int i ;
    int Etat() const { return i ; }
    int &Etat() { return i ; }
} ;

class B : public A
{
    int ConsulterEtat() const
      { return Etat() ; }                  // A::Etat() const
    int &ModifierEtat()
      { return Etat() ; }                        // A::Etat()
} ;
```

De même le choix de supporter ou non les objets constants pour une classe détermine quasiment cette option pour les classes englobantes :

```
class A
{
  public :
    int Etat() ;              /* fonction non constante */
} ;

class B
{
    A a ;

    int Etat() const
      { return a.Etat() ; }                          // non plus
} ;
```

Ici, la fonction constante `B::Etat` ne peut pas utiliser la fonction non constante `A::Etat` de son membre `a`. On résout le problème de la même façon que précédemment.

Le problème peut être résolu d'une autre façon, en déplaçant le membre de type `A` hors de la classe `B`, et en exploitant la remarque précédente sur la subjectivité de la notion d'état d'un objet :

```
class B
{
    A *a ;

    /* ... */
    int Etat() const { a->Etat() ; }                    // ok
} ;
```

Conclusion

Le concept de fonction membre constante est intéressant à utiliser, mais son utilisation n'est pas aussi simple qu'il y parait au premier abord. Définir une fonction constante signifie définir constantes toutes les fonctions de la classe qui le sont implicitement. Cela nécessite aussi d'adapter le type des paramètres de toutes les fonctions de la classe, ainsi que leur type de retour.

Une alternative est d'ignorer toute fonction constante : la classe ne supporte alors ni les objets constants, ni les références et pointeurs vers les objets constants. C'est restrictif, mais cela reste préférable à un support partiel et incomplet.

VI - DUPLICATION DES OBJETS

1 - Quand et pourquoi doit-on implémenter une duplication personnalisée ?

Au même titre que le processus de création et de destruction standard d'objets, mis en place implicitement par le système, et qui conduit à la synthétisation d'un constructeur par défaut et d'un destructeur, le processus de duplication d'objets est lui aussi mis en place implicitement par le système. Cela se traduit pour chaque classe par la synthétisation d'un constructeur de recopie et d'un opérateur d'affectation de recopie.

Mais, dans certaine situation, le comportement prédéfini du processus de duplication est inacceptable. Le concepteur d'une classe doit alors l'implémenter lui-même. Cela passe par la définition d'un constructeur de recopie et d'un opérateur d'affectation de recopie personnalisés.

Le constructeur de recopie synthétisé clone l'objet initial en clonant chacun de ses membres. Mais lorsqu'une partie de l'objet est externe à l'objet, celle-ci n'est pas clonée.

De même, l'opérateur d'affectation synthétisé duplique l'objet initial en dupliquant chacun de ses membres. Mais lorsqu'une partie de l'objet est externe à l'objet, celle-ci n'est pas dupliquée.

Cela se produit en particulier lorsque l'objet possède une partie dynamique. Dans ce cas, bien que les pointeurs membres soient dupliqués, les zones pointées ne le sont pas. Le constructeur de recopie et l'opérateur d'affectation synthétisés fabriquent donc une double référence sur chaque zone dynamique de l'objet initial, ce qui s'avère source de sérieux problèmes au moment de la libération de la mémoire, c'est-à-dire généralement dans le destructeur. Dans la classe :

```
class String
{
    char *str ;

  public :
    String(const char *s)
    {
      str = new char[strlen(s)+1];
      strcpy(str,s);
    }
    ~String() { delete[] str ; }
    const char *data() const { return str ; }
} ;
```

une allocation de mémoire est effectuée dans le constructeur, et la libération a lieu dans le destructeur. Mais cette classe est incorrecte à cause du comportement du constructeur de recopie synthétisé. Par exemple, le programme suivant :

```
void f(String beta)
{
  /* ... */
}

void main()
{
  String alpha("Savennières") ;
  f(alpha) ;
  /* ... */
}
```

aura un comportement indéfini après l'exécution de la fonction `f`, lorsque le destructeur de `beta`, clone de `alpha`, aura été sollicité.

Cette classe est également incorrecte à cause du comportement de l'opérateur d'affectation synthétisé :

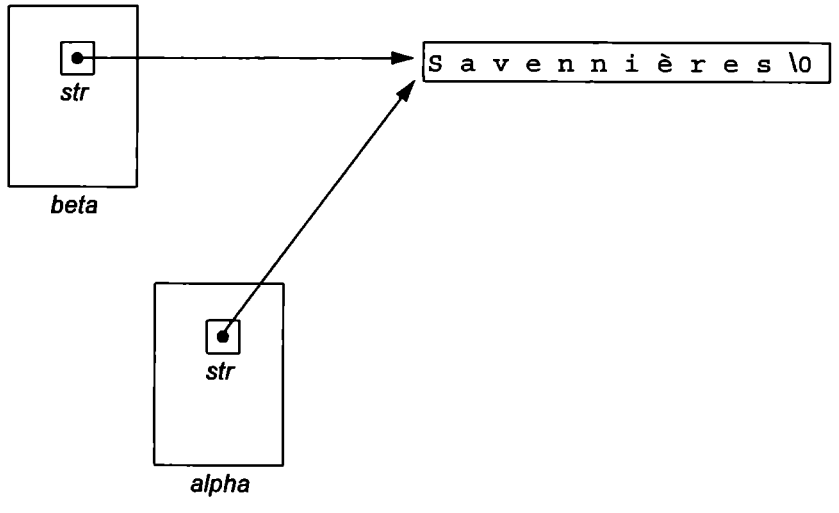
```
int main()
{
  String alpha("Savennières") ;
  /* ... */
  {
    String beta("") ;
    beta = alpha ;
    /* ... */
  }
  /* ... */
}
```

Ici, le comportement du programme sera indéfini après l'exécution du bloc le plus interne de la fonction `main`, lorsque le destructeur de `beta`, copie de `alpha`, aura été sollicité.

En effet, les synthétisations réalisées par le compilateur pour la classe `String` sont :

```
class String
{
    char *str ;

  public :
    /*...*/
    String(const String &s) : str(s.str) {}
    String &operator=(const String &s)
      { str = s.str ; return *this ; }
    /*...*/
} ;
```

Le problème vient du fait que, lors de la recopie de l'argument, le constructeur de recopie et l'opérateur d'affectation synthétisés recopient dans le membre `str` de `beta` la valeur du membre `str` de `alpha`. Les deux pointeurs pointent donc vers la même zone :



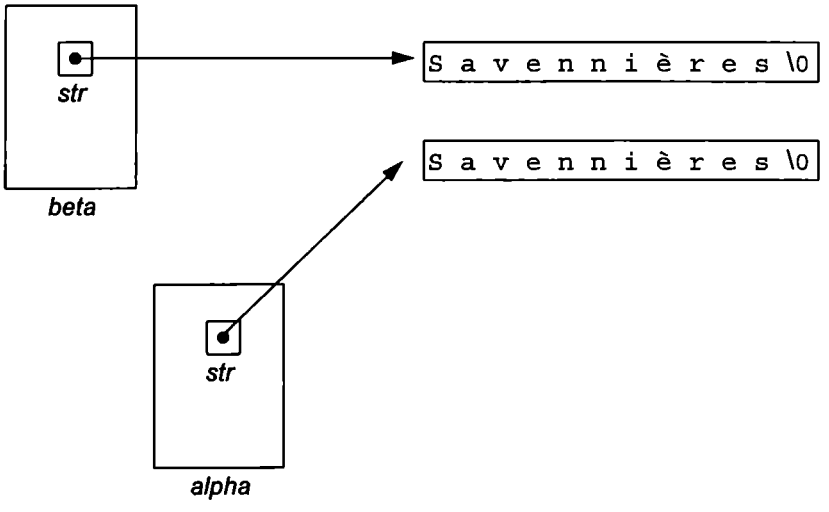
Lors de la destruction de `beta`, le destructeur est appelé, la zone référencée par `str` dans `beta` est donc libérée, ce qui libère également la zone référencée dans `alpha`, puisque c'est la même.

Les solutions à ce problème passent toutes par la redéfinition du constructeur de recopie, de l'opérateur d'affectation de recopie, et éventuellement du destructeur.

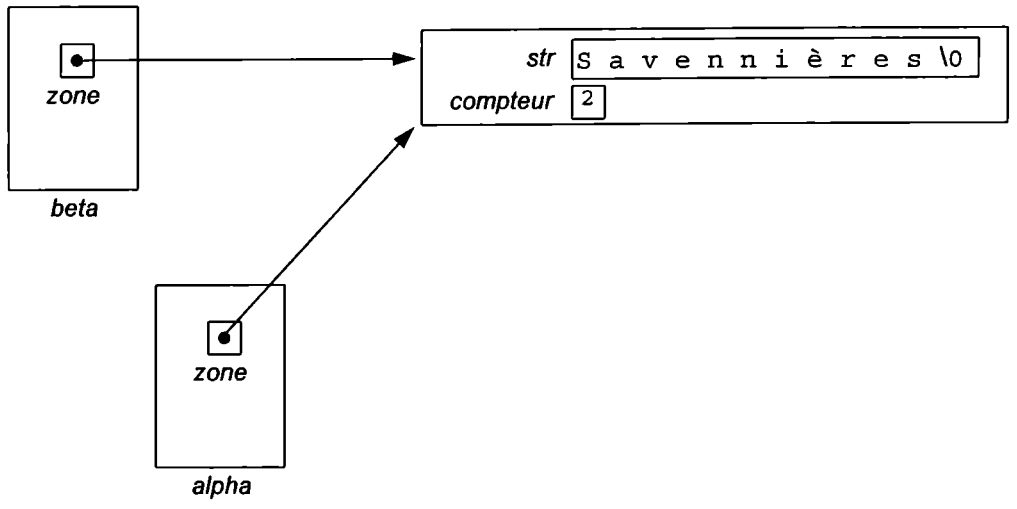
Une première solution consiste à recopier la zone dynamique de l'objet à chaque fois qu'on le duplique, ce qui supprime la double référence :

```
class String
{
    char *str ;

  public :
    String(const char *s)
      {
        str = new char[strlen(s)+1] ;
        strcpy(str,s);
      }
    String(const String &s)
      {
        str = new char[strlen(s.str)+1] ;
        strcpy(str,s.str);
      }
    ~String(){ delete[] str ; }
    String &operator=(const String &s)
      {
        if (&s!=this)
        {
          delete[] str ;
          str = new char[strlen(s.str)+1] ;
          strcpy(str,s.str) ;
        }
        return *this ;
      }
    const char *data() const { return str ; }
} ;
```

Si `beta` est une copie de `alpha`, on obtient maintenant la situation suivante :



Une autre solution consiste à associer à chaque partie dynamique allouée un compteur de référence qui indique combien d'objets la référencent :



Chaque duplication doit alors incrémenter le compteur : ce traitement est fait dans le constructeur de recopie et dans l'opérateur d'affectation de recopie. Le destructeur, quant à lui, ne doit libérer la zone que s'il s'agit de la destruction de la dernière référence. Ce qui donne :

```
class String
{
    struct Zone
    {
      int compteur ;
      char *str ;

      Zone(const char *s) : compteur(1)
      {
        str = new char[strlen(s)+1] ;
        strcpy(str,s) ;
      }

      ~Zone() { delete[] str ; }
    } *zone ;

  public :
    String() : zone(new Zone("")) {}
    String(const char *s) : zone(new Zone(s)) {}
    String(const String &s) : zone(s.zone)
      { zone->compteur++ ; }
    ~String()
      { if (--zone->compteur<=0) delete zone ; }
```

```
    String &operator=(const String &s)
      {
        s.zone->compteur++ ;
        if (--zone->compteur<=0) delete zone ;
        zone = s.zone ;
        return *this ;
      }
    const char &operator[](int i) const
      { return zone->str[i] ; }
    bool operator==(const String &s) const
      { return strcmp(zone->str,s.zone->str)==0 ; }
    bool operator!=(const String &s) const
      { return strcmp(zone->str,s.zone->str)!=0 ; }
} ;
```

Une dernière solution consiste, enfin, à interdire le clonage et l'affectation de recopie, en définissant un constructeur de recopie et un opérateur d'affectation de recopie privés :

```
class String
{
    char *str ;
    String(const String &) {}
    String &operator=(const String &) {}

  public :
    String(const char *s) ;
    ~String() ;
    const char *data() const { return str ; }
} ;
```

Le clonage devient alors illégal :

```
void f(String s)
{
  /* ... */
}

void main()
{
  String alpha("Bonnezeaux") ;
  f(alpha) ;                          // erreur de compilation
}
```

ainsi que l'affectation :

```
void main()
{
  String alpha("Bonnezeaux"), beta("") ;
  beta = alpha ;                      // erreur de compilation
}
```

Cette dernière solution, radicale il est vrai, est malgré tout intéressante :

— lorsque les opérations de clonage et d'affectation n'ont pas des sens pour la classe concernée[127],

— temporairement, lorsque le clonage et l'affectation de recopie ne sont pas encore au point, pour pouvoir tester partiellement la classe en toute sécurité,

— pour déceler les endroits où le compilateur utilise implicitement le constructeur de recopie, puisque chaque clonage implicite provoque une erreur de compilation.

Conclusion

En pratique, toute classe comportant une partie dynamique doit donc redéfinir son mécanisme de duplication. Cette redéfinition s'effectue via le constructeur de recopie et l'opérateur d'affectation de recopie.

Les classes concernées ne sont pas seulement celles qui allouent dans le constructeur une zone de mémoire avec `new`. Par exemple, une classe comportant un membre de type `FILE *` initialisé avec `fopen` est également concernée :

```
class Fichier
{
    FILE *f ;

  public :
    Fichier(char *nom, char *mode)
                                    { f = fopen(nom,mode) ; }
    ~Fichier() { fclose(f) ; }
} ;

void main()
{
  Fichier f1("donnees","r") ;
  {
    Fichier f2(f1) ;       // Attention ! Danger à venir...
    /* ... */
  }
  /* ... */                        // f1 est inutilisable ici
}
```

Lors de la destruction de `f2`, le fichier référencé par `f1.f` est fermé. Personnaliser la duplication est là aussi indispensable.

---

[127] On peut imaginer qu'une classe `Fichier` n'implémente pas les opérations de duplication, sous prétexte qu'une telle duplication se devrait de recopier physiquement le contenu du fichier, ce qui est peu envisageable, pour des raisons de performance.

2 - La duplication d'objets dérivés

Un objet original et sa copie ne sont pas forcément de même type : le premier peut être d'un type dérivé du second. Etant données les classes :

```
class Point
{
  protected :
    int x, y ;

  public :
    Point(int xx, int yy) : x(xx), y(yy) {}
    virtual void Print(ostream &o)
              { o << "Point(" << x << ',' << y << ')' ; }
} ;

class Pixel : public Point
{
  public :
    enum Couleur { NOIR, ROUGE, BLEU,
                                    VERT, JAUNE, BLANC } ;

    Pixel(int xx, int yy, Couleur c) : Point(xx,yy),
                                          couleur(c) {}
    virtual void Print(ostream &o)
                { o << "Pixel(" << x << ',' << y << ','
                                  << couleur << ')' ; }

  private :
    Couleur couleur ;
} ;
```

on peut créer un point clone d'un pixel :

```
Pixel px(1,2,Pixel::BLEU) ;
Point pt(px) ;                                          // légal
```

En effet, le constructeur de recopie synthétisé de `Point` étant de la forme `Point(const Point&)`, un `Pixel` peut lui être fourni en argument.

De même, on peut réaliser l'affectation :

```
Pixel px(1,2,Pixel::BLEU) ;
Point pt(0,0) ;
pt = px ;                                               // légal
```

En effet, l'opérateur d'affectation synthétisé de `Point` étant de la forme `operator=(const Point&)`, un `Pixel` peut lui être fourni en argument.

Ces opérations recopient dans `pt` la partie de `px` issue de la classe `Point`, à savoir `x` et `y`. Par conséquent, le membre `Pixel::couleur` de `px` n'est pas recopié, puisqu'il n'existe pas dans `Point`. `pt` vaut donc `Point(1,2)`.

Conséquence de cela, étant donnée la fonction `Print` :

```
void Print(ostream &o, Point p)
{
  p.Print(o) ;
}
```

l'appel :

```
Print(cout,Pixel(10,10,Pixel::NOIR)) ;
```

est légal, mais envoie sur `cout` la chaîne *Point(10,10)* et non la chaîne *Pixel(10,10,NOIR)*. En effet, le polymorphisme n'intervient pas ici : pour cela, il aurait fallu que le paramètre `p` soit de type `Point&`.

De même, le polymorphisme ne peut en aucun cas être mis en œuvre après une affectation comme :

```
pt = px ;
pt.Print(cout) ;                    // appelle Point::Print
```

`pt.Print` appellera toujours `Point::Print`.

Quant à l'opération inverse, à savoir :

```
Point pt(1,2) ;
Pixel px(pt) ;                                    // illégal
```

elle est illégale, sauf si un constructeur `Pixel::Pixel(const Point &)` est défini explicitement.

De même, l'affectation dans le sens :

```
px = pt ;                                         // illégal
```

est illégale, sauf bien sûr si un opérateur `Pixel::operator=(const Point&)` est définie explicitement.

Par conséquent, il est impossible de retrouver `px` à partir de `pt`.

3 - Synthétisations particulières

Si une classe `C` ne définit pas explicitement de constructeur de recopie, alors tout se passe comme si un constructeur de recopie était synthétisé par le compilateur. Généralement, son en-tête est de la forme `C(const C &)`. Dans ce cas, il peut cloner des objets constants et non constants de type `C`.

De la même façon, l'opérateur `=` de recopie possède une signification prédéfinie. Si une classe `C` ne définit pas explicitement l'affectation de recopie, alors tout ce passe comme si un opérateur `=` de recopie était synthétisé par le compilateur. Le plus souvent, son en-tête est de la forme `C& operator=(const C &)`, et dans ce cas l'opérateur peut dupliquer des objets constants et des objets non constants de type `C`.

Cependant, si la classe `C` contient un objet membre ne sachant pas cloner les constantes, alors le constructeur de recopie synthétisé de `C` ne pourra pas, lui non plus, cloner les constantes, et son en-tête sera alors `C(C&)` :

```
class A
{
  public :
    A() {}
    A(A&) {}
} ;

class C
{
    A a;
} ;

C c1 ;
const C c2 ;
C c3(c1) ;                                          // ok
C c4(c2) ;                                      // illégal
```

Ici, le constructeur de recopie synthétisé de `C` ne peut pas cloner la constante `c2`, car le constructeur de recopie défini dans `A` ne sait pas cloner les objets constants.

De même, si la classe `C` contient un objet membre de type `A` n'ayant qu'un opérateur d'affectation de recopie de la forme `operator=(A&)`, comme dans :

```
class A
{
  public :
    A &operator=(A&) ;
} ;

class C
{
    A a;
} ;
```

alors l'opérateur d'affectation synthétisé de `C` sera alors de la forme `C &operator=(C &)`.

C'est la même chose si la classe `C` dérive d'une classe ne sachant pas cloner les constantes. Le constructeur de recopie synthétisé de `C` ne pourra pas, lui non plus, cloner les objets constants, et l'en-tête sera `C(C&)` :

```
class A
{
  public :
    A() {}
    A(A&) {}
} ;

class C : public A
{
} ;

C c1 ;
const C c2 ;
C c3(c1) ;                                          // ok
C c4(c2) ;                                          // illégal
```

On obtient un résultat identique avec l'opérateur d'affectation synthétisé :

```
class A
{
  public :
    A &operator=(A&) ;
} ;

class C : public A
{
} ;
```

Ici, l'opérateur d'affectation synthétisé pour `C` est de la forme `operator=(C&)`.

Si une classe ne peut utiliser le constructeur de recopie de l'un de ses objets membres, alors aucun constructeur de recopie n'est synthétisé pour la classe englobante :

```
class A
{
    A(A&) {}

  public :
    A() {}
} ;

class B
{
    A a;
} ;

B b1 ;
B b2(b1) ;                                          // illégal
```

Ici, le constructeur de recopie de `A` étant inaccessible pour `B`, aucun constructeur de recopie n'est synthétisé pour `B`.

C'est la même chose pour l'opérateur d'affectation de recopie :

```
class A
{
    A &operator=(A&) ;
} ;

class B
{
    A a;
} ;

B b1, b2 ;
b1 = b2 ;                                    // illégal
```

Ici, l'opérateur d'affectation de recopie de `A` étant inaccessible pour `B`, aucun opérateur d'affectation n'est synthétisé pour `B`.

De même, si une classe ne peut utiliser le constructeur de recopie d'une de ses classes de base, alors aucun constructeur de recopie n'est synthétisé pour la classe dérivée :

```
class A
{
    A(A&) {}

  public :
    A() {}
} ;

class B : public A
{
} ;

B b1 ;
B b2(b1) ;                                   // illégal
```

et même chose pour l'opérateur d'affectation :

```
class A
{
    A &operator=(A&) ;
} ;

class B : public A
{
} ;
```

```
B b1, b2 ;
b1 = b2 ;                                    // illégal
```

Enfin, aucun opérateur d'affectation n'est synthétisé si la classe possède un membre constant, ou un membre de type référence :

```
class A
{
    const int k ;

  public :
    A() : k(0) {}
} ;

A a, b ;
b = a ;                                      // illégal
```

4 - Précisions à propos du clonage

i. Le constructeur de recopie et les objets membres

Sans indication particulière sur l'en-tête du constructeur d'une classe englobante, les objets membres d'une classe sont initialisés par leur constructeur par défaut. Cela est vrai pour tout constructeur de la classe englobante, y compris pour le constructeur de recopie.

Ainsi, lorsque le constructeur de recopie est défini sans spécification particulière quant à l'initialisation des objets membres, ces derniers sont initialisés par leur constructeur par défaut, et non pas par leur constructeur de recopie, comme on pourrait le penser :

```
class A
{
    char *ptrA ;

  public :
    A(){ ptrA = new char[100] ; }
    A(const A &a)
          { ptrA = new char[100] ; strcpy(ptrA,a.ptrA) ; }
    ~A(){ delete[] ptrA ; }
} ;
```

```
class B
{
    A a ;
    char *ptrB ;

  public :
    B() { ptrB = new char[10] ; }
    B(const B &b) ;
    ~B() { delete[] ptrB ; }
} ;

B::B(const B &b)
{
  ptrB = new char[10] ;
  strcpy(ptrB,b.ptrB) ;
}
```

Ici, le constructeur de recopie de `B` ne remplit pas sa fonction, car le membre `a` des clones obtenus est construit avec `A::A()` et non pas avec `A(const A&).a` n'aura donc pas la valeur de `b.a`.

Le constructeur de recopie suivant :

```
B::B(const B &b)
{
  ptrB = new char[strlen(b.ptrB)+1] ;
  strcpy(ptrB,b.ptrB) ;
  a = b.a ;          // non, l'affectation ne fonctionne pas
}
```

est également incorrect si, comme c'est le cas ici, l'affectation de recopie ne fonctionne pas correctement.

Le mieux, c'est d'initialiser les membres du clone sur l'en-tête du constructeur de recopie, par un appel explicite à leur propre constructeur de recopie :

```
B::B(const B &b) : a(b.a)
{
  ptrB = new char[strlen(b.ptrB)+1] ;
  strcpy(ptrB,b.ptrB) ;
}
```

ii. Le constructeur de recopie et l'héritage

Le constructeur de recopie, comme tout constructeur, n'est pas transmis par héritage. En particulier, si `B`, classe dérivée de `A`, ne définit pas de constructeur de recopie, les clones de `B` sont fabriqués par le constructeur de recopie synthétisé de `B`, et non pas par le constructeur de recopie défini pour `A`.

Le constructeur de recopie synthétisé fait une copie membre à membre des objets inclus. Récursivement, un appel au constructeur de recopie est effectué pour chacun d'eux.

Or, dans ce processus, la partie de `B` héritée de `A` est traitée comme un membre de type `A`. Par conséquent, le constructeur de recopie synthétisé de `B` fait appel au constructeur de recopie de `A`, ainsi qu'au constructeur de recopie de chacun des objets membres de `B`.

Mais attention, si `B` définit un constructeur de recopie, le constructeur de recopie de `A` n'est pas automatiquement appelé. Pour qu'il le soit, il faut que cela soit prévu explicitement dans l'en-tête du constructeur de recopie de `B`. Sinon, c'est le constructeur par défaut de `A` qui est appelé, s'il est accessible, pour construire la partie héritée. Si ce constructeur n'est pas accessible, il y a erreur de compilation :

```
class A
{
    char *ptrA ;

  public :
    A(){ ptrA = new char[100] ; }
    A(const A &a)
          { ptrA = new char[100] ; strcpy(ptrA,a.ptrA) ; }
    ~A(){ delete[] ptrA ; }
} ;

class B : public A
{
} ;
```

alors si `x` est de type `B`, l'initialisation :

```
B y(x) ;            // ok : appel implicite de A(const A &)
```

appelle bien le constructeur de recopie de `A` pour cloner la partie héritée. Par contre, étant donnée la classe :

```
class B : public A
{
  public :
    B(const B&) {}
} ;
```

alors, si `x` est de type `B`, l'initialisation :

```
B y(x) ;                          // A(const A &) non appelé
```

appelle le constructeur par défaut de `A`, et non pas le constructeur de recopie, pour cloner la partie héritée. Si celui-ci n'est pas accessible, cela entraîne une erreur de compilation.

Cet appel implicite au constructeur par défaut n'est en général pas satisfaisant. Pour cloner la partie héritée, il est nécessaire d'appeler explicitement le constructeur de recopie :

```
class B : public A
{
  public :
    B(const B &b) : A(b) {}
} ;
```

Maintenant, si `y` est de type `B`, l'expression :

```
B y(x) ;              // ok : appel explicite de A(const A &)
```

appelle bien le constructeur de recopie de `A` pour cloner la partie héritée.

5 - Précisions à propos de l'affectation

i. Ecrire un opérateur d'affectation

Une analogie peut être faite entre l'expression de l'opérateur d'affectation de recopie et celle du constructeur de recopie. Leur objectif est en effet le même : dupliquer l'objet courant.

Il existe pourtant une différence importante entre les deux traitements : lors d'un appel au constructeur de recopie, l'objet courant est *neuf*, alors que pour l'opérateur d'affectation de recopie, l'objet courant existe déjà.

Par conséquent, l'opérateur d'affectation doit nettoyer proprement l'objet courant avant de le réinitialiser, traitement qui n'a pas lieu d'être dans le constructeur de recopie. On pouvait remarquer cette différence dans le cas de la classe `String` de la page 275 :

```
String(const String &s)
{
  str = new char[strlen(s.str)+1] ;
  strcpy(str,s.str);
}

String &operator=(const String &s)
{
  /* ... */
```

où une instruction `delete` est présente dans l'expression de l'opérateur d'affectation et pas dans celle du constructeur de recopie.

De plus, il faut prévoir le cas, toujours possible quoique sans intérêt, de l'affectation d'un objet sur lui-même. En effet, rien n'empêche l'utilisateur de réaliser une telle affectation, dans laquelle l'argument de l'opérateur d'affectation est l'objet courant lui-même. Dans ce cas, l'opérateur d'affectation ne doit pas nettoyer la destination, car cela détruirait du même coup la source. Ce cas de figure n'existe ne peut pas exister avec le constructeur de recopie. On peut remarquer, là encore, cette différence dans le cas de la classe `String` précédente :

```
String &operator=(const String &s)
{
  if (&s!=this)                    // source==destination ?
  {
    delete[] str ;
    str = new char[strlen(s.str)+1] ;
    strcpy(str,s.str) ;
  }
  return *this ;
}
```

En ce qui concerne le type du paramètre de l'opérateur d'affectation de recopie, tout est permis : passage par valeur ou par référence, spécification `const` ou `volatile`... Là encore, un rapprochement peut être fait avec les remarques faites au sujet du constructeur de recopie.

L'opérateur :

```
String &operator=(String &) { /* ... */ }
```

autorise la modification de l'objet source, mais ne permet pas de dupliquer un objet constant.

Par contre, l'opérateur :

```
String &operator=(String) { /* ... */ }
```

provoque à chaque appel la création d'un objet temporaire pour cloner l'objet source. Cet objet consomme inutilement du temps et de l'espace mémoire.

En ce qui concerne le type de retour, là aussi, tout est permis. Pour réaliser des affectations en chaîne, l'opérateur d'affectation renvoie classiquement une référence sur l'objet destination (mais cela pourrait être aussi l'objet source) :

```
String &operator=(String &) { /* ... */ return *this ; }
```

Ainsi, si l'opérateur :

```
void String::operator+=(const String &s) ;
```

est l'opérateur de concaténation de l'opérande `s` à la chaîne courante, l'instruction :

```
(s1 = s2) += s3 ;
```

initialise `s1` avec la valeur de `s2`, puis concatène à `s1` la valeur de `s3`.

Avec l'opérateur suivant :

```
String operator=(const String &s)
                                { /* ... */ return *this ; }
```

la valeur d'une expression d'affectation n'est pas une lvalue : il y a clonage de l'objet renvoyé pour fabriquer le retour de la fonction. Outre la perte de temps et d'espace mémoire, les résultats obtenus peuvent différer de ceux obtenus précédemment. L'expression :

```
(s1 = s2) += s3 ;
```

initialise `s1` avec la valeur de `s2`, puis concatène `s3` au clone de `s1`, ce qui ne modifie ni la valeur de `s1` ni celle de `s2`.

Un opérateur tel que :

```
void operator=(const String &) ;
```

interdit toute expression de la forme :

```
s1 = s2 = s3 ;
```

Conclusion : bien qu'ils aient les mêmes desseins, l'implémentation de l'opérateur d'affectation de recopie est un peu plus délicate que celle du constructeur de recopie.

ii. Ne pas confondre affectation et initialisation

Affectation et initialisation ne doivent pas être confondues :

```
class Complexe
{
  public :
    Complexe(int=0, int=0) ;
    Complexe(const Complexe &) ;
    Complexe &operator=(const Complexe &) ;
  /* ... */
} ;

Complexe a ;
Complexe b(a) ;            // çà, c'est une initialisation
Complexe c=a ;                              // et çà aussi...
c = a ;                 // mais çà, c'est une affectation !
```

L'initialisation, même exprimée avec l'opérateur = n'est pas une affectation.

Affectation et initialisation sont deux opérations distinctes, la première étant réalisée par la fonction `operator=`, la seconde étant réalisée par le constructeur de recopie :

```
class Complexe
{
    Complexe &operator=(const Complexe &) ;
    /* ... */
  public :
    Complexe(int=0, int=0) ;
    Complexe(const Complexe &) ;
    /* ... */
} ;

Complexe a ;
Complexe b=a ;                                    // d'accord
b = a ;                                  // non, c'est privé !
```

Dans la classe suivante :

```
class Vecteur
{
    int *ptr, taille ;

  public :
    Vecteur(int t) { ptr = new int[taille=t] ; }
    Vecteur &operator=(int k)
       {
         for (int i=0 ; i<taille ; i++)
           ptr[i] = k ;
         return *this ;
       }
} ;
```

l'affectation d'un entier à un vecteur est définie comme l'affectation de l'entier à tous les éléments du vecteur. Ce qui permet d'écrire :

```
Vecteur v(20) ;
```

pour créer un vecteur de 20 éléments, et :

```
v = 10 ;
```

pour initialiser de tous ses éléments à 10.

Mais l'initialisation pouvant s'écrire aussi avec l'opérateur =, une confusion peut rapidement s'en suivre :

```
Vecteur v=10 ;
v = 10 ;
```

la déclaration ci-dessus pouvant être prise pour la création d'un vecteur dont tous les éléments sont initialisés à 10, ou bien inversement l'affectation pouvant être prise pour un redimensionnement du vecteur `v`. En fait, la première ligne crée toujours un vecteur de dix éléments, alors que la seconde initialise tous ses éléments à `10`.

Même la classe suivante, plus traditionnelle, peut prêter aussi à confusion :

```
class Vecteur
{
    int *ptr, taille ;

  public :
    Vecteur(int t) { ptr = new int[taille=t] ; }
    Vecteur &operator=(Vecteur &v)
       {
         if (&v!=this)
         {
           if (taille<v.taille)
           {
             delete ptr ;
             ptr = new int[taille=v.taille] ;
           }
           for (int i=0 ; i<taille ; i++)
             ptr[i] = v.ptr[i] ;
         }
         return *this ;
       }
} ;
```

Cette classe définit l'affectation de deux vecteurs comme l'affectation élément par élément. Elle permet des manipulations du type :

```
Vecteur v1(10), v2(20) ;
v1 = v2 ;              // recopie des élements de v2 dans v1
```

Là encore, les écritures peuvent prêter à confusion. Si on peut écrire :

```
Vecteur v=10 ;
```

pour créer un vecteur de 10 éléments, que penser de l'instruction :

```
v = 20 ;
```

Est-elle légale ? Et si oui, redimentionnelle-t-elle `v` ?

Bien que l'affectation d'un entier à un vecteur ne soit pas définie, cette affectation est tout à fait légale. A cause du constructeur de conversion, elle est interprétée comme :

```
v = Vecteur(20) ;
```

Elle ne redimensionne donc pas `v`. Elle recopie dans `v` les éléments d'un vecteur temporaire de `20` éléments non initialisés ! Ce n'est certainement pas ce qui était attendu.

Conclusion : bien qu'initialisation et affectation soient deux opérations distinctes, pour une bonne compréhensibilité de la classe, il est bon que leurs sémantiques ne soient pas trop éloignées.

iii. L'opérateur = et l'héritage

Une classe possède toujours un opérateur d'affectation de recopie : si elle ne le définit pas, le système le synthétise. Par conséquent, une classe ne peut jamais hériter d'un opérateur d'affectation, quel que soit le type du paramètre de cet opérateur : les opérateurs d'affectation des classes de base sont en effet toujours masqués (on retrouve le même phénomène avec les constructeurs et les destructeurs).

Premier cas de figure, une classe `A` possède une fonction `operator=(T)` qui n'est pas de recopie[128]. Si `B`, classe dérivée de `A`, ne redéfinit pas cet opérateur, alors toute affectation de `T` vers `B` est illicite :

```
class A
{
  public :
    void operator=(int) ;
} ;

class B : public A
{
} ;

B b ;
b = 1 ;          // erreur, operator=(int) n'est pas hérité
```

L'affectation d'un entier à `b` est illégale, car `B` n'a pas hérité de l'opérateur d'affectation.

Second cas de figure, `A` a un opérateur d'affectation de recopie. Si `B`, classe dérivée de `A`, ne redéfinit pas cet opérateur, toute affectation de `B` vers `B` est réalisée par l'opérateur d'affectation synthétisé pour `B`, et non pas par l'opérateur de la classe `A`.

Cela peut paraître surprenant, mais se justifie parfaitement par le fait qu'il n'y a aucune raison pour que l'opérateur d'affectation de la classe de base réalise une affectation correcte pour la classe dérivée : celle-ci possède en général des membres supplémentaires. Et de fait, les chances d'avoir un traitement correct sont plus nombreuses si celui-ci est réalisé par l'opérateur d'affectation synthétisé que s'il est réalisé par l'opérateur d'affectation de la classe de base.

En effet, l'opérateur = synthétisé réalise une affectation membre à membre de tous les objets composants l'objet. Mais pour une classe dérivée `B` d'une classe `A`, l'ensemble des membres de `B` hérités de `A` est traité comme un membre de type `A`. L'opérateur = synthétisé de `B` appelle donc, en plus des opérateurs d'affectation de tous les membres de `B`, l'opérateur = de recopie de `A` :

---

[128] C'est-à-dire que le type `T` de l'opérande est différent de `A` et `A&` (combinés éventuellement avec `const` et `volatile`).

```
class A
{

  public :
    void operator=(const A &) ;
} ;

class B : public A
{
} ;
```

Si x et y sont de type B, l'instruction :

```
x = y ;              // ok : appel implicite de A::operator=
```

utilise l'opérateur d'affectation de recopie de A pour dupliquer la partie héritée. Par contre, étant donnée la classe :

```
class B : public A
{
  public :
    void operator=(const B &b) {}
} ;
```

l'instruction :

```
x = y ;                              // A::operator= non appelé
```

n'appelle pas l'opérateur d'affectation de recopie de A : la partie héritée n'est donc pas automatiquement dupliquée. Pour qu'elle le soit, il est nécessaire de le mentionner explicitement :

```
class B : public A
{
  public :
    void operator=(const B &b) { A::operator=(b) ; }
} ;
```

VII - LE POINT SUR LES CONVERSIONS IMPLICITES

On distingue deux types de conversions implicites :
- les conversions standard,
- les conversions définies par l'utilisateur.

1 - Conversions standard

Les conversions standard sont les conversions implicites définies pour les types prédéfinis. Font parties de cette catégorie :
- les conversions portant sur les spécificateurs `const` et `volatile`,
- les promotions entières, convertissant un type entier (`char`, `short`, `int`, `long`, combinés éventuellement avec `unsigned`) ou booléen en un type entier de taille au moins égale,
- les promotions flottantes : conversions d'un `float` en `double`,
- les conversions flottantes : conversions d'un entier en flottant,
- les conversions dégradantes : conversions entières dégradantes, entre type entier et booléen, et conversions flottantes dégradantes,
- les conversions *flottant* vers *entier*, et
- les conversions entre pointeurs.

2 - Conversions définies par l'utilisateur

Le programmeur peut définir des conversions de type de deux façons :
- en définissant pour une classe `C` un constructeur de conversion avec un paramètre de type `T` quelconque, ce qui définit une conversion de `T` vers `C`, ou
- en définissant pour une classe `C` un opérateur de conversion `operator T()`, ce qui définit une conversion de `C` vers `T`.

3 - Séquences de conversions implicites

Une séquence de conversions peut être implicitement appliquée à une expression dans les situations suivantes :

— lorsque l'expression est partie d'une expression englobante,
— lorsque l'expression est la condition d'un `if`, d'un `while`, etc., (conversion en `bool`),
— lorsque l'expression est l'expression d'un `switch` (conversion en entier),
— lorsque l'expression est une expression d'initialisation, ce qui comprend en particulier le passage d'argument et le retour de fonction.

Dans le cas le plus général, une séquence de conversions implicites se décompose successivement en :

— une première séquence de conversions standard,
— une conversion définie par l'utilisateur,
— une seconde séquence de conversions standard.

Toute séquence de conversions implicites C est donc de la forme $C = S_1 U S_2$, avec S_1 et S_2 des conversions standard et U une conversion de l'utilisateur. Par exemple, étant donné :

```
class C
{
  public :
    C(int) ;
} ;

void f(const C&) ;
```

l'appel `f('A')` déclenche une première conversion standard de `char` en `int`, puis la conversion de `int` en `C` définie par l'utilisateur, et enfin la conversion standard de `C` en `const C&`.

i. Utilisation des conversions implicites

Puisqu'une séquence de conversions implicites ne peut contenir plus d'une conversion de l'utilisateur, on en déduit que les conversions de l'utilisateur ne peuvent pas s'enchaîner implicitement. Lorsqu'une transformation nécessite plusieurs conversions de l'utilisateur, le programmeur doit les mentionner explicitement (sauf éventuellement une qui sera alors utilisée implicitement).

L'exemple suivant illustre cette règle dans le cas de conversions définies à l'aide des constructeurs :

```
class A
{
  public :
    A(int) ;                    // définit la conversion int->A
} ;

class B
{
  public :
    B(A) ;                      // définit la conversion A->B
} ;

class C
{
  public :
    C(B) ;                      // définit la conversion B->C
} ;

void main()
{
  // une conversion de l'utilisateur : bien
  A a = 1 ;                                        // A a=A(1)

  // une conversion : bien
  B b1 = B(1) ;                                    // B b1=B(A(1))

  // une conversion : bien
  B b2 = A(1) ;                                    // B b2=B(A(1))

  // deux conversions : c'est illégal
  B b3 = 1 ;     // erreur : n'effectue pas : B b3=B(A(1))

  // une conversion : bien
  C c1 = B(A(1)) ;                                 // C c1=C(B(A(1)))

  // une conversion : bien
  C c2 = C(A(1)) ;                                 // C c2=C(B(A(1)))

  // une conversion : bien
  C c3 = C(B(1)) ;                                 // C c3=C(B(A(1)))

  // deux conversions : c'est illégal
  C c4 = A(1) ;                 // erreur, n'effectue pas :
                                                   // C c4=C(B(A(1)))

  // deux fois 1 conversion : ça passe
  C c5 = B(1) ;                                    // C c5=C(B(A(1)))
```

```
  // deux conversions : c'est illégal
  C c6 = C(1) ;                    // erreur, n'effectue pas :
                                   // C c6=C(B(A(1)))

  // trois conversions : c'est illégal
  C c7 = 1 ;                                  // non évidemment
}
```

L'exemple suivant illustre cette règle dans le cas de conversions définies à l'aide des opérateurs de conversion :

```
class A
{
  public :
    operator int() ;          // définit la conversion A->int
} ;

class B
{
  public :
    operator A() ;              // définit la conversion B->A
} ;

void main()
{
  B b ;
  int i ;

  // deux conversions de l'utilisateur : c'est illégal
  i = b+1 ;                                  // n'effectue pas :
                        // i=(b.operator A()).operator int()+1

  // une conversion : bien
  i = A(b)+1 ;                                     // effectue :
                       // i=(b.operator A()).operator int()+1
}
```

Même chose si l'on combine les deux :

```
class A
{
  public :
    A(int) ;                  // définit la conversion int->A
} ;

class B
{
  public :
    operator int() ;          // définit la conversion B->int
} ;
```

```
void f(A) ;

void main()
{
  B b ;

  // deux conversions de l'utilisateur : c'est illégal
  f(b) ;        // n'effectue pas : f(A(b.operator int()))

  // une conversion : bien
  f(A(b)) ;          // effectue : f(A(b.operator int()))

  // une conversion : bien
  f(int(b)) ;        // effectue : f(A(b.operator int()))
}
```

ii. Limiter les conversions implicites

Une fois définies, ces conversions sont utilisées sans avertissement par le compilateur. Le cheminement menant aux résultats peut rapidement devenir obscur. Dans la classe suivante :

```
class Rationnel
{
    int n, d ;

  public :
    Rationnel() : n(0), d(1) {}
    Rationnel(int i) : n(i), d(1) {}
    Rationnel(int i, int j=1) : n(i), d(j) {}
    operator double() { return double(n)/d ; }
    Rationnel operator +(const Rationnel &r) { /* ... */ }
  /* ... */
} ;
```

la conversion d'un rationnel en double est réalisée par l'opérateur `double` qui effectue la division du numérateur par le dénominateur. La conversion d'un entier en rationnel est définie par le second constructeur : un entier est un rationnel avec un dénominateur égal à `1`. Enfin, l'addition de deux rationnels est définie par l'opérateur `+`.

Malgré sa simplicité, la classe `Rationnel` peut laisser l'utilisateur perplexe. En effet, que donne une expression comme `1+Rationnel(1,2)` ? Deux cheminements sont possibles :

— `Rationnel(1)+Rationnel(1,2)`, addition dans l'ensemble des rationnels, ce qui donne comme résultat `Rationnel(3,2)`, ou bien

— `1+Rationnel(1,2).operator double()`, addition dans l'ensemble des réels, ce qui donne `1.5`.

Et obtient-on le même résultat pour `Rationnel(1,2)+1` dans ce sens ? Et qu'obtient-on si l'opérateur d'addition n'est pas membre de la classe, mais une fonction globale ? Autant d'interrogations qui nuisent à la compréhensibilité du programme...

Par conséquent, il convient de bien réfléchir avant de définir de telles conversions, et de les limiter au strict nécessaire.

Mais de telles définitions se font parfois contre la volonté du concepteur. Un constructeur à un paramètre peut être intéressant, sans pour cela que la conversion associée soit souhaitée. Le mot clé `explicit` permet facilement de réduire le nombre de conversion :

```
class Rationnel
{
    int n, d ;

  public :
    Rationnel() : n(0), d(1) {}
    explicit Rationnel(int i) : n(i), d(1) {}
    Rationnel(int i, int j=1) : n(i), d(j) {}
    operator double() { return double(n)/d ; }
    Rationnel operator +(const Rationnel &r) { /* ... */ }
  /* ... */
} ;
```

Maintenant, l'expression `1+Rationnel(1,2)` est interprétée sans équivoque comme `1+Rationnel(1,2).operator double()` : c'est une addition dans l'ensemble des réels, qui donne `1.5`.

De même, pour la classe :

```
class Rationnel
{
    int n, d ;

  public :
    Rationnel() : n(0), d(1) {}
    Rationnel(int i) : n(i), d(1) {}
    Rationnel(int i, int j=1) : n(i), d(j) {}
    explicit operator double() { return double(n)/d ; }
    Rationnel operator +(const Rationnel &r) { /* ... */ }
  /* ... */
} ;
```

l'expression `1+Rationnel(1,2)` est interprétée comme `Rationnel(1)+Rationnel(1,2)`, ce qui donne `Rationnel(3,2)`.

4 - Hiérarchisation des conversions

Les conversions sont partiellement hiérarchisées entre elles : une conversion peut être *meilleure* qu'une autre (ou au contraire *plus mauvaise*), ou bien deux conversions peuvent être de même *qualité*.

Cette relation induit un ordre partiel sur l'ensemble des conversions. Cette hiérarchisation intervient notamment lors de la résolution des surcharges.

En ce qui concerne les conversions standard, on trouve, des meilleures au plus mauvaises :

— les conversions portant sur les spécificateurs `const` et `volatile`,
— les promotions diverses,
— les autres (conversions dégradantes, conversions flottantes, etc.).

Les conversions de l'utilisateur, par contre, ne sont pas hiérarchisées entre elles, mais une conversion standard est meilleure qu'une conversion de l'utilisateur.

Enfin, étant données deux séquences de conversions implicites C_1 et C_2, définies par :

$C_1 = S_{11}\, U_1\, S_{12}$ et $C_2 = S_{21}\, U_2\, S_{22}$,

avec les S_{ij} des conversions standard et les U_i des conversions de l'utilisateur, alors C_1 et meilleure que C_2 si $U_1=U_2$ et si S_{12} est meilleure que S_{22}.

Cela veut dire en particulier que seules les séquences basées sur la même conversion de l'utilisateur sont comparables.

5 - Ambiguïtés

Si n'existe pas de meilleure séquence de conversion pour passer d'un type `A` en un type `B`, ou au contraire s'il en existe plusieurs, alors une conversion explicite est nécessaire pour lever l'ambiguïté :

```
class A
{
  public :
    operator long() ;
    operator float() ;
} ;

void main()
{
  A a ;
  int i = 1+a ;              // ambigu : 1+a.operator long()
                             // ou 1+a.operator float() ?
  int j = 1+long(a) ;              // 1+a.operator long()
  int k = 1+float(a) ;             // 1+a.operator float()
}
```

VIII - L'ALGORITHME DE RESOLUTION DES SURCHARGES

La résolution des surcharges est le processus visant à sélectionner, pour un appel donné, la fonction à exécuter. Ce processus s'applique aux fonctions globales, aux fonctions membres et aux opérateurs.

Par exemple, la résolution détermine pour l'appel `f(t,t)` ci-dessous :

```
void f(int,int) ;
void f(float,float) ;

template <class T>
void h(T t)
{
  f(t,t) ;
}
```

si c'est la fonction `f(int,int)` qui sera appelée pour un `T` donné, si c'est au contraire `f(float,float)`, ou si l'appel est illégal.

Les critères de sélection intervenant lors des résolutions de surcharges sont :

— le nombre d'arguments spécifiés à l'appel et le nombre de paramètres des fonctions,

— la *distance* entre le type des arguments et celui des paramètres correspondants, c'est-à-dire le nombre et la qualité des conversions qu'il est nécessaire de faire subir à chacun des arguments pour l'amener au type du paramètre qui lui correspond,

— pour les fonctions membres, la distance entre le type de l'objet par qui la fonction est appelée et celui du propriétaire de la fonction.

Deux cas se présentent :

— la résolution aboutit, ce qui veut dire qu'il existe une *meilleure* fonction pour l'appel examiné, et c'est donc cette fonction qui sera retenue, ou bien

— la résolution n'aboutit pas, et dans ce cas l'appel est illégal.

La résolution n'aboutit pas :

— s'il n'existe aucune fonction pouvant répondre à l'appel, ou

— s'il existe plusieurs fonctions répondant à l'appel, et aucune d'entre elles n'est *meilleure* que les autres.

Lorsque la résolution aboutit, ce n'est pas pour autant que l'appel est légal. En effet, d'autres conditions (contrôle d'accès, fonction non virtuelle pure, etc.) doivent être ensuite vérifiées :

```
class C
{
    void f(int,int) ;

  public :
    void f(float,float) ;
} ;

C c ;
c.f(1,1) ;                                    // illégal
```

Ici, la résolution sélectionne `f(int,int)` pour l'appel `c.f(1,1)`, mais cette fonction est privée. L'appel est donc illégal.

La résolution des surcharges est un algorithme complexe, qui se déroule en trois phases successives :

— détermination des fonctions *candidates*,
— détermination des fonctions *viables*,
— détermination de la *meilleure fonction viable*.

1 - Fonctions candidates

La première phase consiste à construire l'ensemble des fonctions candidates. Pour l'appel d'une fonction non membre, une fonction candidate est une fonction de même nom que la fonction appelée, dont la déclaration est visible au point d'appel :

```
void f() ;
void f(int) ;
void f(double,double=0) ;
void f(char *,char *) ;

void g()
{
  f(1.2) ;
}
```

Ici, l'ensemble des fonctions candidates pour `f(1.2)` est { `f()`, `f(int)`, `f(double,double=0)`, `f(char *,char *)` }.

Une fonction déclarée dans une région déclarative donnée masque toutes les fonctions de même nom déclarées dans une région déclarative englobante. Ainsi, pour l'appel `f(1.2)` suivant :

```
void f(double) ;

void g()
{
  void f(int) ;
  f(1.2) ;
}
```

l'ensemble des fonctions candidates est { `f(int)` }. `f(double)` est masquée.

Pour l'appel d'une fonction membre, les fonctions candidates sont recherchées dans la portée de la classe dont l'objet est instance. Pour l'appel `c.f(1.2)` ci-dessous :

```
class C
{
  public :
    f(int) ;
} ;

void f(double) ;

C c ;
c.f(1.2) ;
```

l'ensemble des fonctions candidates est { `C::f(int)` }.

Une fonction déclarée dans une classe masque les fonctions de même nom déclarées dans les classes de base. Ainsi, pour l'appel `b.f(1.2)` ci-dessous :

```
f(double) ;

class A
{
  public :
    f(double) ;
} ;

class B : public A
{
  public :
    f(int) ;
} ;

B b ;
b.f(1.2) ;
```

l'ensemble des fonctions candidates est { `B::f(int)` }.

De façon générale, les régions déclaratives sont explorées de la plus englobée à la plus englobante, mais l'exploration s'arrête à la région déclarative contenant au moins une fonction candidate. Les fonctions candidates obtenues appartiennent donc toutes à la même région déclarative. Ainsi, dans l'exemple précédent, seule la classe `B` a été explorée pour construire l'ensemble des fonctions candidates. Mais si `B::f(int)` n'avait pas existé, la classe `A` aurait été explorée. De même, si `A::f(double)` n'avait pas existé, alors la région déclarative globale aurait été explorée.

La construction de l'ensemble des fonctions candidates pour un appel explicite à une fonction `operator` est identique. Mais lorsque la notation opérationnelle est utilisée, les règles sont différentes.

Tout d'abord, si les opérandes sont tous de type prédéfini, alors seul un opérateur prédéfini va pouvoir être appelé, puisque la sémantique des opérations sur les types prédéfinis ne peut être modifiée. Par exemple, l'addition suivante :

```
class String
{
  public :
    String(char *) ;
} ;

char *operator+(const String &, const String &) ;

void main()
{
  char *s = "un"+"deux" ;                          // erreur
}
```

est illégale : les opérandes étant tous deux de type `char *`, c'est forcément l'addition prédéfinie qui est choisie, et non pas l'opérateur + défini au-dessus. Mais l'addition de deux pointeurs n'étant pas définie, l'instruction est illégale.

Maintenant, s'il existe au moins un opérande de type objet ou énumération, et si la notation opérationnelle est utilisée, trois ensembles de fonctions `operator` candidates vont être construits : les fonctions membres candidates (si le premier opérande est un objet), les fonctions non membres candidates, et enfin les opérateurs prédéfinis candidats. Pour l'addition suivante :

```
class String
{
  public :
    String(char *) ;
    char *operator+(char *) ;
} ;

char *operator+(const String &, char *) ;
```

```
void main()
{
  String str("un") ;
  char *s = str+"deux" ;
}
```

les opérateurs candidats sont { `String::operator+(char *)`, `operator+(const string &,char *)` }.

Enfin, c'est au cours de cette phase que peuvent être générées de nouvelles fonctions, à partir des modèles de fonctions accessibles.

Si l'ensemble des fonctions candidates est vide, le processus s'arrête et l'appel est illégal. Sinon, le processus passe à la phase suivante.

2 - Fonctions viables

Pour un appel donné, les fonctions viables sont les fonctions candidates pouvant être appelées avec les arguments ou les opérandes spécifiés à l'appel.

Plus précisément, à partir de l'ensemble des fonctions candidates, la sélection des fonctions viables repose sur la comparaison entre le nombre et le type des arguments présents à l'appel, et le nombre et le type des paramètres des fonctions :

— premièrement, une fonction viable est une fonction candidate qui possède un nombre de paramètres adéquate pour accepter les arguments spécifiés, en tenant compte le cas échéant des arguments par défaut ;

— ensuite, pour chacun des arguments, il doit exister une séquence de conversions implicites pouvant le convertir dans le type du paramètre correspondant.

```
void f() ;
void f(int) ;
void f(double,double=0) ;
void f(char *,char *) ;

void g()
{
  f(1.2) ;
}
```

Ici, l'ensemble des fonctions viables est { `f(double,double=0)`, `f(int)` }.

Pour les fonctions membres, une transformation supplémentaire est appliquée. Les fonctions membres se voient dotées d'un paramètre supplémentaire, placé par convention en tête de liste, qui représente l'objet par qui la fonction est appelée :

```
class C
{
  void f(int) ;
  void g(int) const ;
} ;
```

Les deux fonctions ci-dessus sont considérées au cours de cette phase comme les fonctions :

```
void f(C&,int) ;
void g(const C&,int) ;
```

Parallèlement, la même transformation est appliquée aux arguments. L'appel :

```
C c ;
c.f(2) ;
```

est transformé en :

```
f(c,2) ;
```

Mais attention, ces transformations ne sont effectuées que pour résoudre la surcharge, et ne présument en rien de l'implémentation sous-jacente, qui est libre d'interpréter l'appel de fonction membre différemment.

Le paramètre et l'argument, après avoir été ajoutés, sont traités comme les autres paramètres et arguments, avec cependant une restriction supplémentaire quant aux conversions pouvant y être appliquées : les conversions définies par l'utilisateur n'ont pas le droit d'intervenir pour amener le type de cet argument à celui de ce paramètre :

```
class A
{
} ;

class B
{
  public :
    B(A&) ;
    void f() ;
} ;

A a ;
a.f() ;                                    // illégal
```

Ici, l'appel `a.f()` est illégal, car la conversion de `a` en `B` nécessite l'utilisation de la conversion définie par l'utilisateur à l'aide du constructeur de conversion.

A l'issue de cette étape, si l'ensemble des fonctions viables est vide, le processus s'arrête et l'appel est illégal. Sinon, le processus passe à la troisième et dernière phase.

3 - Meilleure fonction viable

La dernière phase consiste à choisir la meilleure des fonctions viables, en examinant la distance qui existe entre les types des arguments et ceux des paramètres correspondants.

Si `f1` et `f2` sont deux fonctions viables, `f1` est meilleure que `f2` s'il n'existe pas d'entier *i* pour lequel l'argument de rang *i* de `f1` nécessite une plus *mauvaise*[129] conversion que l'argument de rang *i* de `f2`, pour l'amener au type du paramètre correspondant, et si :

- il existe un entier *j* pour lequel l'argument de rang *j* de `f1` peut être amené au type du paramètre correspondant avec une meilleure conversion que l'argument de rang *j* de `f2`, ou
- `f1` n'est pas une instance de modèle et `f2` l'est, ou
- `f1` et `f2` sont des instances de modèle, mais `f1` est plus spécialisée que `f2`.

Ainsi, dans le contexte :

```
void f(int) ;
void f(float) ;
```

la meilleure fonction viable pour l'appel `f('A')` est `f(int)`. En effet, une promotion est meilleure qu'une conversion flottante.

De même, étant données :

```
void f(float,int) ;
void f(int,float) ;
```

il n'existe pas de meilleure fonction viable pour l'appel `f(1,1)`, ni pour `f(1.,1.)`.

Puisqu'une conversion standard, même dégradante, est meilleure qu'une conversion définie par l'utilisateur, dans le contexte :

```
class C
{
  public :
    C(int) ;
    operator int() ;
} ;

void f(float,int,int) ;
void f(int,C,int) ;
C c(1) ;
```

la meilleure fonction viable correspondant à l'appel `f(0.1,c,1)` est `f(int,C,int)`.

Par contre, dans le contexte :

---

[129] Voir *Hiérarchisation des conversions* page 300.

```
class A
{
  public :
    A(int) ;
} ;

class B
{
  public :
    B(int) ;
} ;

void f(A) ;
void f(B) ;
```

il n'y a pas de meilleure fonction viable pour l'appel `f(1)`, puisque les conversions de l'utilisateur ne sont pas hiérarchisées.

Dernier exemple, dans le contexte :

```
void f(int,long,int) ;
template<class T> void f(T,unsigned T,T) ;
```

la meilleure fonction viable pour l'appel `f(1,1,1)` est `f(int,long,int)`, alors que pour `f(1,1u,1)` la meilleure fonction viable est l'instance `f(T,unsigned T,T)` générée pour `T` valant `int`.

S'il existe une meilleure fonction viable, et si elle est unique, la résolution aboutit et produit cette fonction comme résultat. Sinon, la résolution échoue, et l'appel est illégal.

IX - L'HERITAGE DANS LA PRATIQUE

1 - Amitié et héritage

Ce paragraphe illustre, à l'aide de trois exemples, le comportement de la relation d'amitié face à l'héritage.

Etant donnée B une classe dérivée de A, une amie d'une classe B a accès aux membres de B hérités de A, comme y ont accès les membres de la classe B eux-mêmes :

```
class A
{
    int pri ;

  protected :
    int pro ;

  public :
    int pub ;
} ;

class B : private A
{
  friend void f() ;
} ;

void f()
{
  B b ;
  b.pri = 1 ;        // non, pri inaccessible à partir de B
  b.pro = 1 ;           // ok, pro accessible à partir de B
  b.pub = 1 ;           // ok, pub accessible à partir de B
}
```

La relation d'amitié n'est pas transmise par héritage : si une classe B est amie d'une classe A, alors toute classe dérivée de B n'est pas amie de A :

```
class A
{
    friend class B ;
    int i ;
} ;

class B
{
    f() {
          A a ;
          a.i = 1 ;                    // ok, B est amie de A
        }
} ;

class C : public B
{
    f() {
          A a ;
          a.i = 1 ;        // illégal, C n'est pas amie de A
        } ;
} ;
```

Par contre, une amie d'une classe `A` reste amie de la partie héritée dans toute classe dérivée de `A` :

```
class A
{
    friend void f() ;
    int i ;
} ;

class B : private A
{
    int j ;
} ;

void f()
{
  B b ;
  b.i = 1 ;          // ok, f est amie de la partie héritée
  b.j = 1 ;                              // erreur, j est privé
}
```

2 - Membres statiques et héritage

Les membres statiques, qui sont partagés par tous les objets d'une même classe, sont partagés également par les objets de type dérivé :

```
class A
{
    static int nb ;

  public :
    A() { nb++ ; }
    ~A() { nb-- ; }
} ;

int A::nb=0 ;

class B : public A
{
} ;

void main()
{
  A a ;                                    // A::nb vaut 1
  B b ;                                    // A::nb vaut 2
}
```

Après la construction de `b`, `A::nb` vaut 2.

3 - Pointeurs vers membres et héritage

Etant donnée une classe `D` et `B` une classe de base accessible de `D`, un pointeur vers un membre de `B` de type `T` peut être implicitement converti en un pointeur vers un membre de `D` de type `T`. Par exemple, à partir de :

```
class B
{
  public :
    int i ;
    void f() ;
} ;

class D : public B
{
  /* ... */
} ;
```

on peut écrire :

```
int D::*ptr ;
B b ;
ptr = &B::i ;                    // ok, conversion implicite
```

De même, on peut écrire :

```
void (D::*pfct)() ;
B b ;
pfct = &B::f ;                    // ok, conversion implicite
```

Bien que cette conversion *pointeur vers membres de classe de base* vers *pointeur vers membres de classe dérivée* soit logique (un membre de `B` est forcément dans `D`), elle semble inversée par rapport à la conversion des pointeurs vers les objets : *pointeur vers objets instances d'une classe dérivée* vers *pointeur vers objets instances d'une classe de base*[130].

4 - dynamic_cast

L'objectif du `dynamic_cast` est de pouvoir obtenir, à partir d'un pointeur de type `T*` ou d'une référence de type `T&` référençant un objet d'un type dérivé de `T`, des informations spécifiques à l'objet n'existant pas dans `T`.

Les fonctions virtuelles offrent déjà une telle possibilité, solution préférable et bien plus élégante : si le mécanisme des fonctions virtuelles peut être utilisé, c'est incontestablement un meilleur choix.

Si tel n'est pas le cas, l'opérateur `dynamic_cast` peut être intéressant. Supposons que l'on dispose d'une bibliothèque contenant les classes suivantes :

```
class Figure
{
  protected :
    int nbCote ;
    float lgCote ;

  public :
   float Perimetre() { return nbCote*lgCote ; }
  /* ... */
} ;

class TriangleEquilateral : public Figure
{
  public :
    TriangleEquilateral(int lgCote) ;
} ;

class Losange : public Figure
{
  public :
    Losange(int lgCote) ;
} ;
```

[130] Voir *Polymorphisme* page 114.

```
class Carre : public Losange
{
  public :
    Carre(int lgCote) ;
} ;

class Pentagone : public Figure
{
  public :
    Pentagone(int lgCote) ;
} ;
```

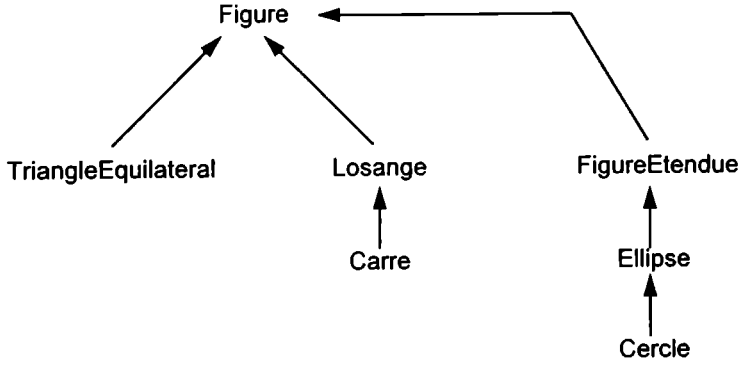
La fonction `Figure::Perimetre` n'est pas virtuelle, puisqu'elle est valable pour tout polygone régulier.

On veut maintenant créer une fonction calculant le périmètre d'une figure géométrique quelconque, pas forcément polygonale et régulière. On désire réutiliser au maximum ce qui existe déjà dans la bibliothèque. Mais on ne dispose pas des sources de cette bibliothèque.

On crée donc une hiérarchie de classes :

```
class FigureEtendue : public Figure
{
  public :
    virtual float Perimetre()=0 ;
  /* ... */
} ;

class Ellipse : public FigureEtendue
{
  public :
    Ellipse(int R, int r) ;
    float Perimetre() ;
} ;
/* ... */
```

que l'on rattache tant bien que mal à la hiérarchie de la bibliothèque à l'aide d'un lien d'héritage entre `Figure` et `FigureEtendue`.



Pour manipuler une figure quelconque, il est alors nécessaire de déterminer si celle-ci est une figure de la bibliothèque ou si c'est une figure rajoutée. La fonction calculant le périmètre d'une figure quelconque s'écrit alors :

```
float perimetre(Figure *f)
{
  FigureEtendue *e = dynamic_cast<FigureEtendue *>(f) ;
  return e ? e->Perimetre()      // f pas polygone régulier
           : f->Perimetre() ;       // f polygone régulier
}
```

ou bien :

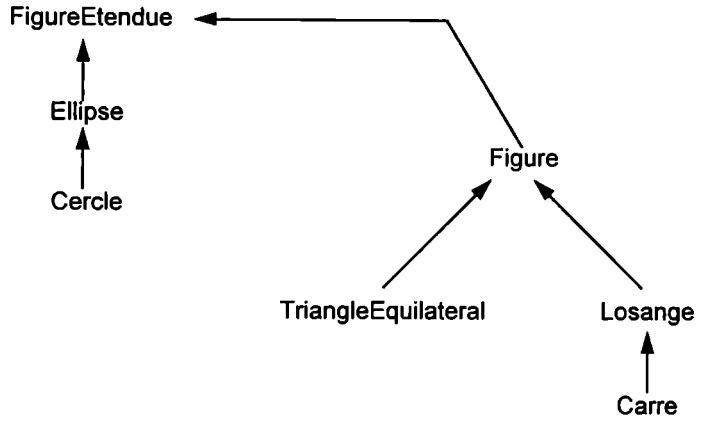
```
float perimetre(Figure &f)
{
  try
  {
    FigureEtendue &e = dynamic_cast<FigureEtendue &>(f) ;
    return e.Perimetre() ;       // f pas polygone régulier
  }
  catch (Bad_cast)
  {
    return f.Perimetre() ;           // f polygone régulier
  }
}
```

La fonction fait appel à `Figure::Perimetre` si son paramètre est une classe de la bibliothèque, ou sinon fait appel à l'une des fonctions virtuelles `Perimetre`.

Bien sûr, dans le cas où les sources de la bibliothèque sont disponibles, la meilleure solution consiste à dériver `Figure` de `FigureEtendue` et à rendre `FigureEtendue::Perimetre` virtuelle :

```
class FigureEtendue
{
  /* ... */
  public :
    virtual float Perimetre()=0 ;
    /* ... */
} ;

class Figure : public FigureEtendue
{
  protected :
    int nbCote ;
    float lgCote ;
```

```
  public :
   float Perimetre()
     { return nbCote*lgCote ; }        // devient virtuelle
  /* ... */
} ;

class TriangleEquilateral : public Figure
{
  public :
    TriangleEquilateral(int lgCote) ;
} ;
/* ... */

class Ellipse : public FigureEtendue
{
  public :
    Ellipse(int R, int r) ;
    float Perimetre() ;
} ;
/* ... */
```



la fonction `perimetre` s'écrivant alors trivialement comme :

```
float perimetre(FigureEtendue *f)
{ return f->Perimetre() ; }
```

Le `dynamic_cast` est intéressant à utiliser dans le cas où l'expression d'une fonction dépend du type dynamique de plusieurs objets à la fois.

Supposons qu'à partir des classes suivantes :

```
class Point : public pair<float,float>
{
  public :
    Point() : pair<float,float>(0,0) {} ;
} ;
```

```
class Figure
{
  virtual void Tracer() ;          // pour être polymorphe
} ;

class Cercle : public Figure
{
    float rayon ;
    Point centre ;

  public :
    Point &Centre() ;
    float &Rayon() ;
} ;

class Carre : public Figure
{
    float cote ;
    Point gravite ;

  public :
    Point &Centre() ;
    float &Cote() ;
} ;
```

on veut réaliser une fonction qui inscrit une figure dans une autre. Par exemple,

```
Cercle c1 ;
Carre c2 ;
Inscrire(&c1,&c2) ;
```

doit inscrire le cercle `c1` dans le carré `c2`.

L'expression de la fonction `Inscrire` va dépendre du type dynamique de ses deux paramètres. L'opérateur `dynamic_cast` peut être utilisé pour les déterminer :

```
void Inscrire(Figure *f1, Figure *f2)
{
  Cercle *c1 = dynamic_cast<Cercle *>(f1) ;
  if (c1)
  {
    Cercle *c2 = dynamic_cast<Cercle *>(f2) ;
    if (c2)
    {
      // inscription du cercle f1 dans le cercle f2
      c1->Rayon() = c2->Rayon() ;
      c1->Centre() = c2->Centre() ;
    }
```

```
    else
    {
      Carre *c2 = dynamic_cast<Carre *>(f2) ;
      if (c2)
      {
        // inscription du cercle f1 dans le carré f2
        c1->Rayon() = c2->Cote()/2 ;
        c1->Centre() = c2->Centre() ;
      }
    }
  }

  else

  {
    Carre *c1 = dynamic_cast<Carre *>(f1) ;
    if (c1)
    {
      Cercle *c2 = dynamic_cast<Cercle *>(f2) ;
      if (c2)
      {
        // inscription du carré f1 dans le cercle f2
        c1->Cote() = c2->Rayon()*sqrt(2) ;
        c1->Centre() = c2->Centre() ;
      }
      else
      {
        Carre *c2 = dynamic_cast<Carre *>(f2) ;
        if (c2)
        {
          // inscription du carre f1 dans le carre f2
          c1->Cote() = c2->Cote() ;
          c1->Centre() = c2->Centre() ;
        }
      }
    }
  }
}
```

`Inscrire` peut utiliser de la même façon des références, ce qui donne :

```
void Inscrire(Figure &f1, Figure &f2)
{
  try
  {
    Cercle &c1 = dynamic_cast<Cercle &>(f1) ;
```

```
    try
    {
      Cercle &c2 = dynamic_cast<Cercle &>(f2) ;
      // inscription du cercle f1 dans le cercle f2
      c1.Rayon() = c2.Rayon() ;
      c1.Centre() = c2.Centre() ;
    }
    catch(bad_cast)
    {
      try
      {
        Carre &c2 = dynamic_cast<Carre &>(f2) ;
        // inscription du cercle f1 dans le carré f2
        c1.Rayon() = c2.Cote()/2 ;
        c1.Centre() = c2.Centre() ;
      }
      catch(bad_cast)
      {
        // f2 n'est ni un cercle, ni un carré
      }
    }
  }
  catch(bad_cast)
  {
    try
    {
      Carre &c1 = dynamic_cast<Carre &>(f1) ;
      try
      {
        Cercle &c2 = dynamic_cast<Cercle &>(f2) ;
        // inscription du carré f1 dans le cercle f2
        c1.Cote() = c2.Rayon()*sqrt(2) ;
        c1.Centre() = c2.Centre() ;
      }
      catch(bad_cast)
      {
        try
        {
          Carre &c2 = dynamic_cast<Carre &>(f2) ;
          // inscription du carre f1 dans le carre f2
          c1.Cote() = c2.Cote() ;
          c1.Centre() = c2.Centre() ;
        }
```

```
        catch(bad_cast)
        {
          // f2 n'est ni un cercle, ni un carré
        }
      }
    }
    catch(bad_cast)
    {
      // f1 n'est ni un cercle, ni un carré
    }
  }
}
```

Un autre exemple d'utilisation du `dynamic_cast` est fourni au paragraphe *Opérateurs virtuels* page 326.

5 - Plus sur les fonctions virtuelles

i. Interprétation d'un appel de fonction virtuelle

L'accès à une fonction virtuelle est contrôlé statiquement par le compilateur :

```
class A
{
  public :
    virtual void f() ;
} ;

class B : public A
{
    void f() ;
} ;

void main()
{
  B b ;
  A *pa=&b ;
  B *pb=&b ;

  pa->f() ;                              // ok, pa->B::f()
  pb->f() ;                          // non B::f est privée
}
```

Ici, l'appel de la fonction `B::f` par l'instruction `pa->f()` est permis bien que cette fonction soit privée, car `pa` est de type `A*`. Le contrôle d'accès est réalisé statiquement par le compilateur d'après le type statique de `*pa`. Par contre, l'appel à partir d'un pointeur de type `B*` est bien sûr interdit.

C'est la même chose en ce qui concerne les arguments par défaut :

```
class A
{
  public :
    virtual void f(int=0) ;
} ;

class B : public A
{
  public :
    void f(int) ;
} ;

void main()
{
  B b ;
  A *pa=&b ;
  B *pb=&b ;

  pa->f() ;                                          // oui
  pb->f() ;                                        // erreur
}
```

Ici, bien que `pa->f()` appelle `B::f`, c'est l'argument par défaut de `A::f` qui est utilisé. Par contre, pour l'instruction `pb->f()` qui appelle aussi `B::f`, l'argument par défaut de `A::f` n'est pas utilisable. Les arguments par défaut sont donc ceux du prototype retenu, déterminé statiquement à la compilation.

C'est le même principe en ce qui concerne la résolution des surcharges :

```
class A
{
  public :
    virtual void f(int) ;
} ;

class B : public A
{
  public :
    virtual void f(char) ;
    virtual void f(int) ;
} ;

void main()
{
  B b ;
  A *pa=&b ;
  B *pb=&b ;
  pa->f('A') ;                          // appelle B::f(int)
  pb->f('A') ;                         // appelle B::f(char)
}
```

Ici, la résolution de la surcharge pour l'appel `pa->f('A')` est fait statiquement à la compilation, au sein de la classe `A`. Comme il n'y a qu'une fonction `f` dans `A`, la fonction sélectionnée est évidemment `f(int)`. Mais à l'exécution, la fonction `f(int)` appelée est celle de `B`, puisque `*pa` est de type `B` : c'est donc `B::f(int)` qui est appelée, et non pas `B::f(char)` comme on pourrait le croire.

ii. Quand faut-il déclarer une fonction virtuelle ?

La différence de comportement entre une fonction virtuelle et une fonction non virtuelle n'apparaît que lorsque la fonction en question est appelée pour un objet `o` via un pointeur ou une référence qui n'est pas du type de `o`. Si la fonction est virtuelle, le type de l'objet référencé est utilisé pour déterminer la fonction appelée. Sinon, c'est le type du pointeur ou de la référence qui est utilisé.

Concrètement, dans la pratique, la question qui se pose est : quand doit-on déclarer une fonction virtuelle ?

En fait, il n'y a aucun risque à rendre virtuelles des fonctions qui n'ont pas lieu de l'être. La seule conséquence est une petite surconsommation d'espace mémoire pour la classe et de temps pour l'appel. Par contre, oublier un `virtual` peut provoquer des dysfonctionnements.

Donc, pour ne pas prendre de risque, on peut déclarer `virtual` toutes les fonctions membres.

Plus finement, étant données une classe `A` et une fonction `A::f`, on peut se poser les questions suivantes :

1/- peut-il y avoir un intérêt à dériver de nouvelles classes à partir de cette classe `A` ? Si la réponse est non, les fonctions de la classe `A` n'ont pas de raison d'être virtuelles ;

2/- peut-il y avoir un intérêt à utiliser des pointeurs de type `A *` ou des références de type `A &` pour accéder à des objets dérivés de `A` ? Si la réponse est non, les fonctions de la classe `A` n'ont pas de raison d'être virtuelles ;

3/- peut-il y avoir un intérêt à redéfinir la fonction `A::f` dans une classe dérivée ? Autrement dit, le traitement réalisé par la fonction `f` est-il dépendant du type de l'objet courant ? Si la réponse est non, la fonction `A::f` n'a pas de raison d'être virtuelle.

Si la réponse aux trois questions est oui, la fonction `A::f` doit obligatoirement être virtuelle.

iii. Constructeurs virtuels

Un constructeur ne peut pas être virtuel, car le type d'un objet est déterminé lorsque celui-ci est complètement construit. Donc, pendant la séquence d'appels des constructeurs, le véritable type de l'objet ne lui est pas encore attribué.

Cela n'est pas sans conséquence pour le programmeur. Par exemple, lorsqu'une fonction virtuelle est appelée à partir d'un constructeur, la fonction appelée est celle définie dans la classe contenant le constructeur ou dans l'une de ses classes de base. Ce n'est jamais une fonction définie dans une classe dérivée :

```
class A
{
  public :
    A() { f() ; }
    virtual void f() ;
} ;

class B : public A
{
  public :
    virtual void f() ;
} ;

B b ;
```

Ici, c'est la fonction `A::f` qui est appelée lors de la construction de `b`, et non pas la fonction `B::f`, comme on serait tenté de le penser.

En fait, tout ce passe comme si le constructeur ne savait pas qu'il était en train de construire une partie d'un objet qui n'est pas de son propre type : dans le constructeur de `A` appelé par `B::B()`, l'objet pointé par `this` est de type `A`, et non pas, comme on pourrait s'y attendre, de type `B`. Dans un constructeur, le type dynamique de l'objet est toujours la classe de ce constructeur.

Cela a aussi des répercussions sur l'interprétation des expressions `typeid` et `dynamic_cast` au sein d'un constructeur :

```
class A
{
  public :
    A() { cout << typeid(*this).name() << '\n' ; }
    virtual void f() {}                    // A est polymorphe
} ;

class B : public A
{
} ;

B b ;
```

Bien que `A` soit polymorphe, c'est le message `"A"` qui sort à la construction de `b`, et non pas le message `"B"`.

Ces remarques s'appliquent de la même façon aux destructeurs :

```
class A
{
  public :
    ~A() { cout << typeid(*this).name() << '\n' ; }
    virtual void f() {}                    // A est polymorphe
} ;

class B : public A
{
} ;

B b ;
```

Là encore, c'est le message `"A"` qui sort à la destruction de `b`.

Il est donc clair qu'un constructeur ne peut pas être virtuel. Pourtant, dans certain cas, il est intéressant de pouvoir créer un objet sans connaître exactement son type. Comment faire ?

Une solution est de définir une fonction capable de créer un objet de type quelconque appartenant à une hiérarchie d'héritage donnée. C'est ce que fait la fonction `creer` ci-dessous, qui crée un nouvel objet de même type que celui pour lequel elle a été appelée :

```
class A
{
  public :
    virtual A *creer() { return new A ; }
} ;

class B : public A
{
  public :
    virtual B *creer() { return new B ; }
} ;
```

Cette fonction permet de créer des objets dont le type n'est pas connu à la compilation :

```
void f(A &a)
{
  A *nouveau = a.creer() ;              // quel que soit le
                                        // type dynamique de a
  /* ... */
}
```

Le nouvel objet pointé par nouveau est du même type que le type dynamique de a. On sait seulement qu'il est soit de type A, soit d'un type dérivé de A.

De même, la fonction cloner suivante :

```
class A
{
  public :
    virtual A *cloner() { return new A(*this) ; }
} ;

class B : public A
{
  public :
    virtual B *cloner() { return new B(*this) ; }
} ;

void f(A &a)
{
  A *nouveau = a.cloner() ;                    // quel que soit
                                          // le type dynamique de a
  /* ... */
}
```

crée un nouvel objet identique à celui pour lequel elle a été appelée. a.cloner clone donc a quel que soit son type, à partir du moment où celui-ci appartient à la hiérarchie d'héritage d'origine A.

iv. Destructeurs virtuels

Une classe peut être amenée à définir un destructeur virtuel, même si ce destructeur ne fait rien. Cela est obligatoire lorsque la classe peut servir de classe de base à d'autres classes, et que le polymorphisme est susceptible d'être utilisé. La classe abstraite suivante :

```
class Objet
{
  public :
    virtual ~Objet() {}
} ;
```

est destinée à être racine d'une hiérarchie d'héritage. La classe Objet possède un destructeur virtuel, ce qui est indispensable pour un fonctionnement correct du polymorphisme, même si celui-ci ne fait rien. En effet, dans l'exemple suivant :

```
class ObjetSpecial : public Objet
{
    char *s ;

  public :
    ObjetSpecial() { s = new char[80] ; }
    ~ObjetSpecial() { delete s ; }
    /* ... */
} ;

void main()
{
  Objet *ptr = new ObjetSpecial ;
  /* ... */
  delete ptr ;          // exécution de :~ObjetSpecial,
                        // puis de       ~Objet
}
```

si `Objet` n'avait pas eu de destructeur virtuel, l'instruction `delete ptr` n'aurait pas appelée `~ObjetSpecial`, mais seulement `~Objet`. Et si `Objet` n'avait pas eu de destructeur du tout, aucun destructeur n'aurait été appelé.

Un destructeur virtuel est donc obligatoire lorsqu'un objet est susceptible d'être détruit via un pointeur ou une référence dont le type est un type de base de celui de l'objet détruit.

A noter que si une classe dérive d'une classe possédant un destructeur virtuel, cette classe dérivée est dotée automatiquement d'un destructeur virtuel. La question du destructeur virtuel ne se pose donc que pour une classe racine d'un arbre d'héritage, et la déclaration d'un destructeur virtuel est suffisante pour toute la hiérarchie :

```
class ObjetTresSpecial : public ObjetSpecial
{
  public :
    ObjetTresSpecial() { /* ... */ }
    ~ObjetTresSpecial() { /* ... */ }
    /* ... */
} ;

void main()
{
  ObjetSpecial *ptr = new ObjetTresSpecial ;
  /* ... */
  delete ptr ;    // exécution de :   ~ObjetTresSpecial
                  // puis de :       ~ObjetSpecial
                  // et enfin de :   ~Objet
}
```

Ici, bien que la classe `ObjetSpecial` n'ait pas déclaré explicitement son destructeur `virtual`, les destructeurs sont appelés correctement.

v. Opérateurs virtuels

Les opérateurs peuvent être virtuels. Les cas d'utilisation sont les mêmes que pour les fonctions membres. La classe :

```
class Vecteur
{
    int *ptr ;

  public :
    Vecteur(int taille) { ptr = new int[taille] ; }
    ~Vecteur() { delete[] ptr ; }
    int &operator[](int i) { return ptr[i] ; }
} ;
```

implémente de façon classique un vecteur d'entiers, indicés de `0` à `taille-1`. Or un problème se pose si cette classe est employée comme classe de base et si le polymorphisme doit fonctionner :

```
class VecteurEvolue : public Vecteur
{
    int debut ;

  public :
    VecteurEvolue(int deb, int fin) :
                          Vecteur(fin-deb+1), debut(deb) {}
    int &operator[](int i)
                  { return Vecteur::operator[](i-debut) ; }
} ;
```

`VecteurEtendu` est un vecteur pour lequel l'indice du premier élément n'est pas forcément `0`. La fonction :

```
int somme(Vecteur &v, int a, int b)
{
  int s=0 ;

  for (int i=a ; i<=b ; i++) s += v[i] ;
}
```

pose problème. En effet, `v[i]` fait appel ici à la fonction `Vecteur::operator[]`, quel que soit le type dynamique de `v`, ce qui a toutes les chances de déborder lorsque le type dynamique de `v` est `VecteurEtendu`.

La solution est de définir virtuel l'opérateur `[]` de la classe `Vecteur` (et probablement aussi le destructeur) :

```
class Vecteur
{
    int *ptr ;

  public :
    Vecteur(int taille) { ptr = new int[taille] ; }
    virtual ~Vecteur() { delete[] ptr ; }
    virtual int &operator[](int i) { return ptr[i] ; }
} ;
```

Voici une autre illustration du même problème :

```
class A
{
  int ka ;
} ;

class B : public A
{
    int kb ;
} ;
```

Le résultat de l'affectation dans la fonction suivante :

```
void affect(A &a1, const A &a2)
{
  a1 = a2 ;                          // comportement incorrect
}
```

n'est pas satisfaisant. En effet, l'opérateur d'affectation synthétisé n'étant pas virtuel, l'instruction `a1=a2` appelle toujours l'opérateur d'affectation synthétisé de `A`. Si le type dynamique de `a1` et `a2` est `B`, le membre `a2.kb` n'est pas recopié dans `a1.kb`.

La solution consistant à définir pour chaque classe un opérateur d'affectation de recopie virtuel ne fonctionne pas :

```
class A
{
    int ka ;

  public :
    virtual A &operator=(const A &a) { ka = a.ka ; }
} ;
```

```
class B : public A
{
    int kb ;

  public :
    virtual B &operator=(const B &b)
                          { kb = b.kb ; A::operator=(b) ; }
} ;
```

En effet, le type du paramètre n'étant pas le même pour les deux opérateurs, `B::operator=` masque `A::operator=` au lieu de le surcharger. Par conséquent, l'affectation `a1=a2` appelle toujours dans ce cas `A::operator=`, puisque l'argument est de type `A`.

La solution est plus délicate. `B::operator=` doit avoir un paramètre de type `const A&`, et doit détecter si le type dynamique de ce paramètre est `B`. Si c'est le cas, il effectue la recopie de ses membres spécifiques puis recopie la partie héritée, sinon il appelle l'opérateur d'affectation de la classe de base :

```
class A
{
    int ka ;

  public :
    virtual A &operator=(const A &a)
                             { ka = a.ka ; return *this ; }
} ;

class B : public A
{
    int kb ;

  public :
    virtual B &operator=(const A &a)
      {
        try
        {
          B b = dynamic_cast<const B&>(a) ;
          kb = b.kb ;
        }
        catch(bad_cast) {}
        A::operator=(a) ;
        return *this ;
      }
} ;
```

Cela se généralise de la même façon à toute une hiérarchie (les écritures de messages insérées dans le code ci-dessous permettront de suivre le fonctionnement) :

```
class A
{
    int ka ;

  public :
    typedef A Self ;

    virtual Self &operator=(const Self &a)
       {
         cout << "  A::operator= : " ;
         cout << "membres propres à A recopiés\n" ;
         ka = a.ka ;
         return *this ;
       }
} ;

class B : public A
{
    int kb ;

  public :
    typedef B Self ;
    typedef A Super ;
    typedef A Base ;

    virtual Self &operator=(const Base &a)
       {
         cout << "  B::operator= : " ;
         try
         {
           Self b = dynamic_cast<const Self &>(a) ;
           kb = b.kb ;
           cout << "membres propres à B recopiés\n" ;
         }
         catch(bad_cast)
         {
           cout << "non concerné\n" ;
         }
         Super::operator=(a) ;
         return *this ;
       }
} ;
```

```
class C : public B
{
    float kc ;

  public :
    typedef C Self ;
    typedef B Super ;
    typedef B::Base Base ;

    virtual Self &operator=(const Base &a)
       {
         cout << "  C::operator= : " ;
         try
         {
           Self c = dynamic_cast<const Self &>(a) ;
           kc = c.kc ;
           cout << "membres propres à C recopiés\n" ;
         }
         catch(bad_cast)
         {
           cout << "non concerné\n" ;
         }
         Super::operator=(a) ;
         return *this ;
       }
} ;
```

L'exécution suivante :

```
void affect(A &a1, const A &a2)
{
  a1 = a2 ;
}

void main()
{
  A a ;
  B b ;
  C c ;

  cout << "a=a\n" ; affect(a,a) ;
  cout << "a=b\n" ; affect(a,b) ;
  cout << "a=c\n" ; affect(a,c) ;

  cout << "--------------------\n" ;
```

```
    cout << "b=a\n" ; affect(b,a) ;
    cout << "b=b\n" ; affect(b,b) ;
    cout << "b=c\n" ; affect(b,c) ;

    cout << "-------------------\n" ;

    cout << "c=a\n" ; affect(c,a) ;
    cout << "c=b\n" ; affect(c,b) ;
    cout << "c=c\n" ; affect(c,c) ;
}
```

donne :

```
a=a
  A::operator= : membres propres à A recopiés
a=b
  A::operator= : membres propres à A recopiés
a=c
  A::operator= : membres propres à A recopiés
-------------------

b=a
  B::operator= : non concerné
  A::operator= : membres propres à A recopiés
b=b
  B::operator= : membres propres à B recopiés
  A::operator= : membres propres à A recopiés
b=c
  B::operator= : membres propres à B recopiés
  A::operator= : membres propres à A recopiés
-------------------

c=a
  C::operator= : non concerné
  B::operator= : non concerné
  A::operator= : membres propres à A recopiés
c=b
  C::operator= : non concerné
  B::operator= : membres propres à B recopiés
  A::operator= : membres propres à A recopiés
c=c
  C::operator= : membres propres à C recopiés
  B::operator= : membres propres à B recopiés
  A::operator= : membres propres à A recopiés
```

Maintenant, l'affectation de recopie supporte le polymorphisme pour ces deux opérandes.

X - UTILISATION DES EXCEPTIONS

Le mécanisme de gestion des exceptions implémente un moyen de passer d'un point du programme où est détectée une situation *anormale* à un point de traitement de cette situation, passage accompagné d'un transfert d'informations. Ce saut d'exécution s'apparente à un appel de fonction avec passage d'un argument, à ceci près que la portion de code atteinte par ce saut est inconnue à la compilation.

1 - Créer des types d'exceptions

Il est intéressant de créer des types spécialement pour les exceptions. Cela augmente la compréhensibilité du programme. L'utilisation des structures ou des classes à ce propos est intéressante, puisque cela permet de remonter plusieurs informations au point de traitement de l'exception.

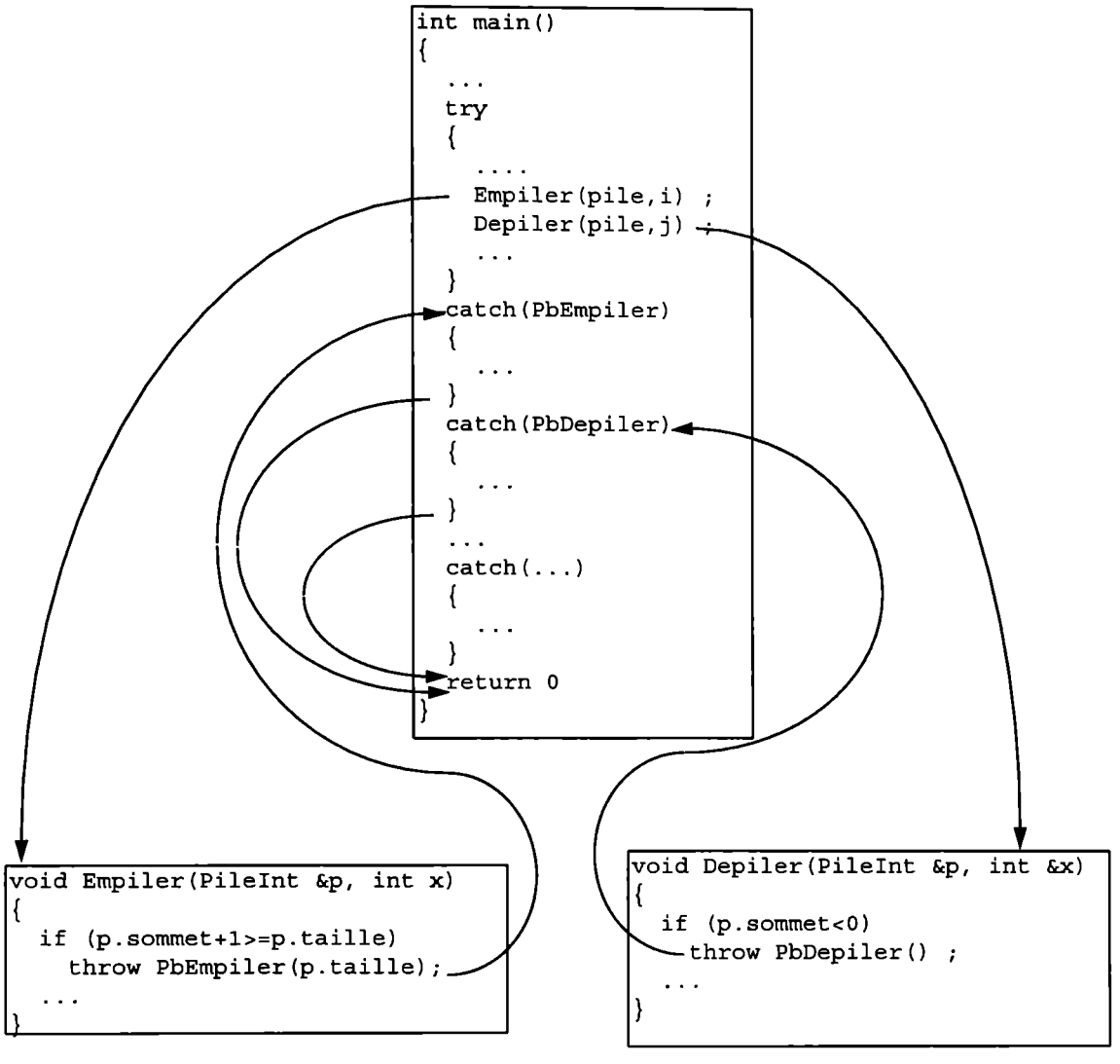
Finalisons l'exemple du paragraphe *Spécification des exceptions* page 165, en créant les types `PbEmpiler`, `PbDepiler` et `PbDepilerNFois`, tous dérivés de `PbPile`, et dédiés aux exceptions :

```
struct PileInt { int *pile, sommet, taille ; } ;

// --- types exceptions ----

struct PbPile : public logic_error
{
  PbPile() : logic_error("problème de pile") {}
} ;

struct PbEmpiler : public PbPile
{
  int taille ;
  PbEmpiler(int t) : taille(t) {}
} ;
```

```
struct PbDepiler : public PbPile {} ;

struct PbDepilerNFois : public PbPile
{
  int offre, demande ;
  PbDepilerNFois(int o, int d) : offre(o), demande(d) {}
} ;

// ---- implémentation des opérations sur la pile ----

void Empiler(PileInt &p, int x)
{
  if (p.sommet+1>=p.taille) throw PbEmpiler(p.taille) ;
  p.pile[++p.sommet] = x ;
}

void Depiler(PileInt &p, int &x)
{
  if (p.sommet<0) throw PbDepiler() ;
  x = p.pile[p.sommet--] ;
}

void DepilerNFois(PileInt &p, int n, int &x)
{
  int i ;

  try
  {
    for (i=1 ; i<=n ; i++) Depiler(p,x) ;
  }
  catch(PbDepiler)
  {
    throw PbDepilerNFois(i-1,n) ;
  }
}
```

Cette hiérarchie de classes d'exception autorise un contrôle fin des erreurs, où chaque type d'exception est traité spécifiquement :

```
int main()
{
  PileInt pile ;
  int i, j ;

  try
  {
    /* ... */
    Empiler(pile,i) ;
    /* ... */
    Depiler(pile,j) ;
    /* ... */
    DepilerNFois(pile,10,i) ;
    /* ... */
  }
  catch(PbDepiler pb)
  {
    cout << pb.what()
         << "\nproblème pour dépiler\n" ;
  }
  catch(PbEmpiler pb)
  {
    cout << pb.what() << "\nproblème pour empiler\n"
         << "taille de la pile:" << pb.taille << "\n" ;
  }
  catch(PbDepilerNFois pb)
  {
    cout << pb.what() << "\nproblème pour dépiler "
         << pb.demande << " fois:"
         << "il ne reste que " << pb.offre
         << "élément(s)\n";
  }
  catch(...)
  {
    cout << "erreur non prévue\n" ;
    terminate() ;
  }

  return 0 ;
}
```

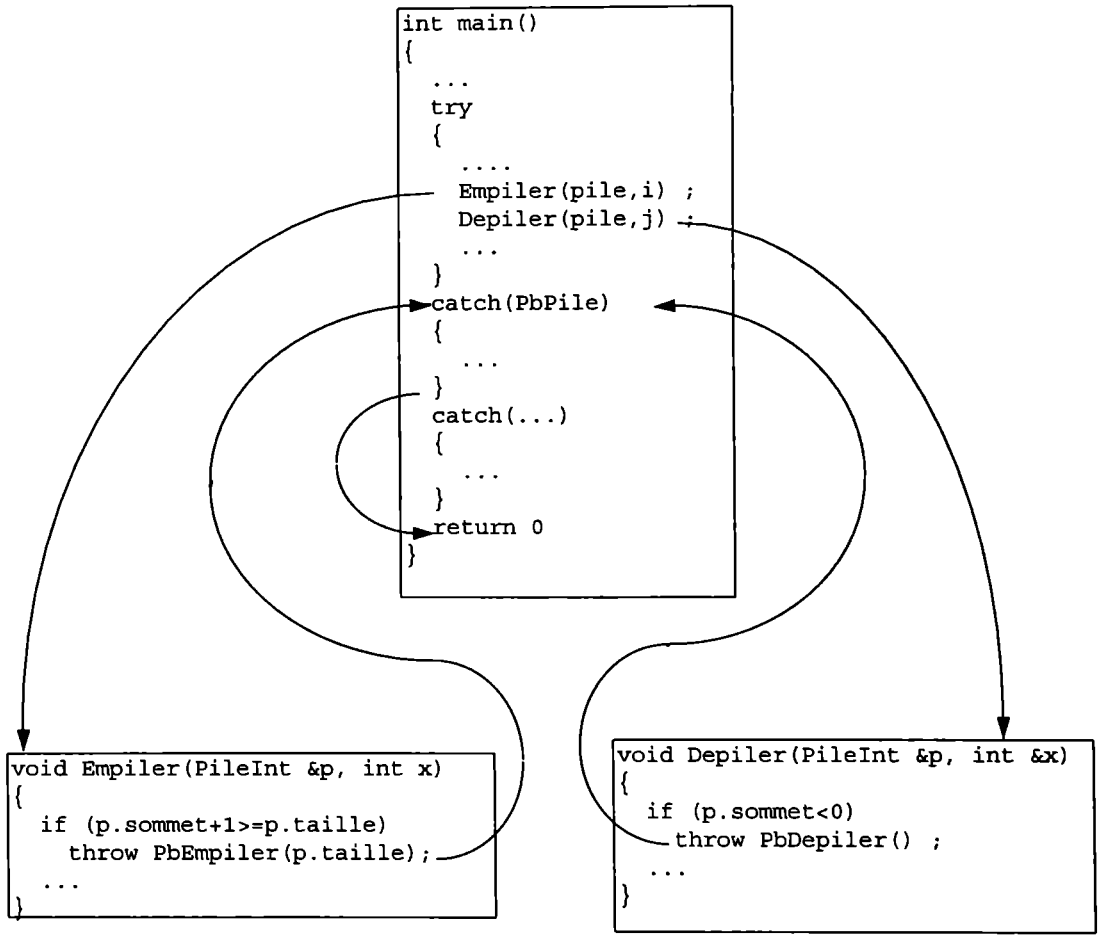


Elle autorise également un contrôle sommaire, où tous les types d'exceptions relatifs aux piles sont traités de la même façon :

```
int main()
{
  PileInt pile ;
  int i, j ;

  try
  {
    /* ... */ Empiler(pile,i) ;
    /* ... */ Depiler(pile,j) ;
    /* ... */ DepilerNFois(pile,10,i) ;
  }
```

```
  catch(PbPile pb)
  {
    cout << pb.what() ;
  }
  catch(...)
  {
    cout << "erreur non prévue\n" ;
    terminate() ;
  }
  return 0 ;
}
```

En effet, suivant les règles de sélection des gestionnaires d'exceptions[131], tout type ayant `PbPile` comme classe de base accessible sera intercepté par le gestionnaire `catch(PbPile)` :



[131] Voir *Sélection du gestionnaire* page 162.

2 - Exemples de gestionnaires

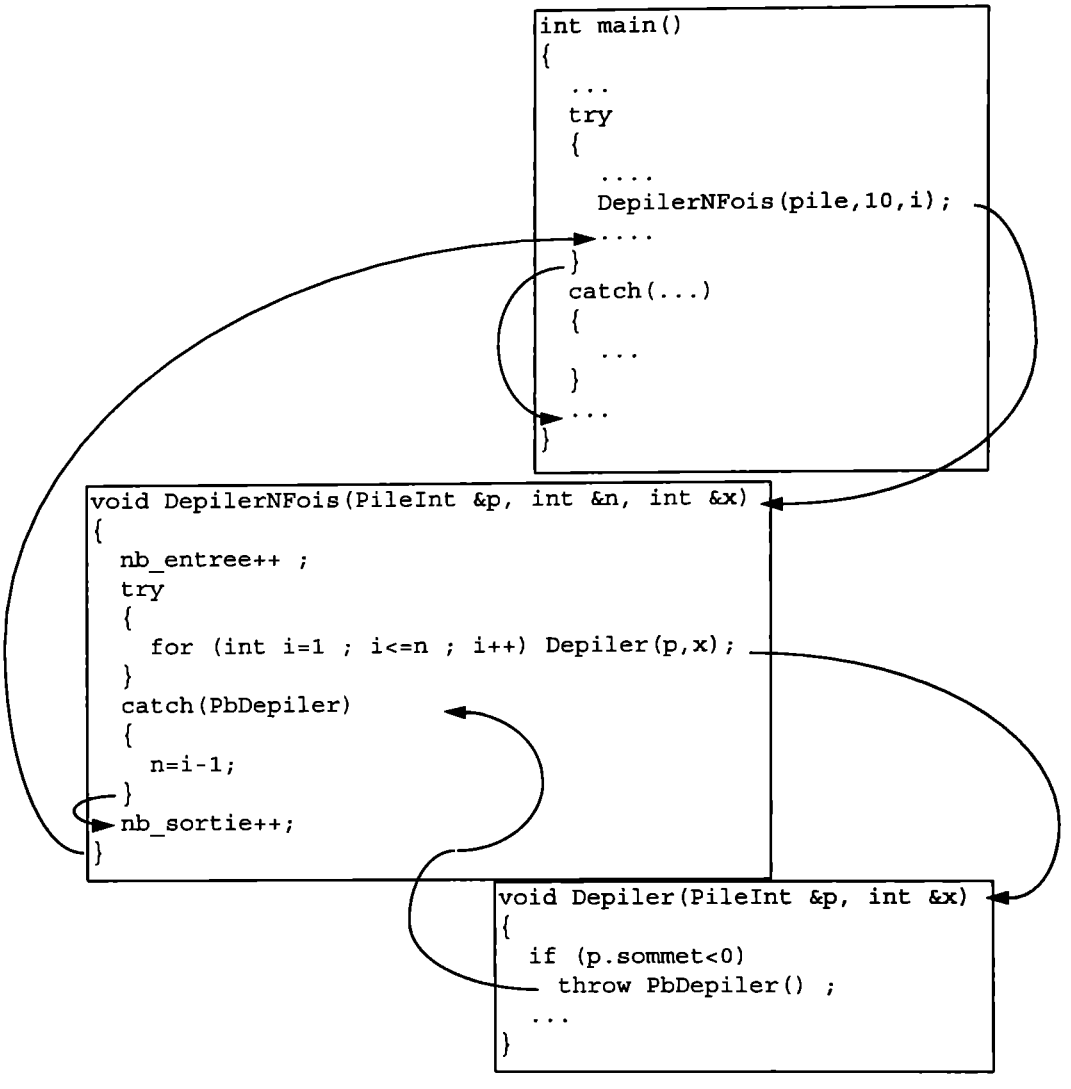
Le traitement d'une exception peut consister simplement à interrompre l'exécution du programme :

```
void f(PileInt &pile)
{
  int i ;

  try
  {
    /* ... */
    Empiler(pile,i) ;
    /* ... */
  }
  catch(PbPile)
  {
    cout << "problème de pile\n" ;
    terminate() ;                          // fin d'exécution
  }
}
```

Il peut consister à corriger l'erreur et à continuer avec l'instruction suivant la structure `try/catch` :

```
void DepilerNFois(PileInt &p, int &n, int &x)
{
  int i ;
  static int nb_entree=0 ;
  static int nb_sortie=0 ;

  nb_entree++ ;
  try
  {
    for (i=1 ; i<=n ; i++) Depiler(p,x) ;
  }
  catch(PbDepiler)
  {
    n = i-1 ;
  }
  nb_sortie++ ;
}
```

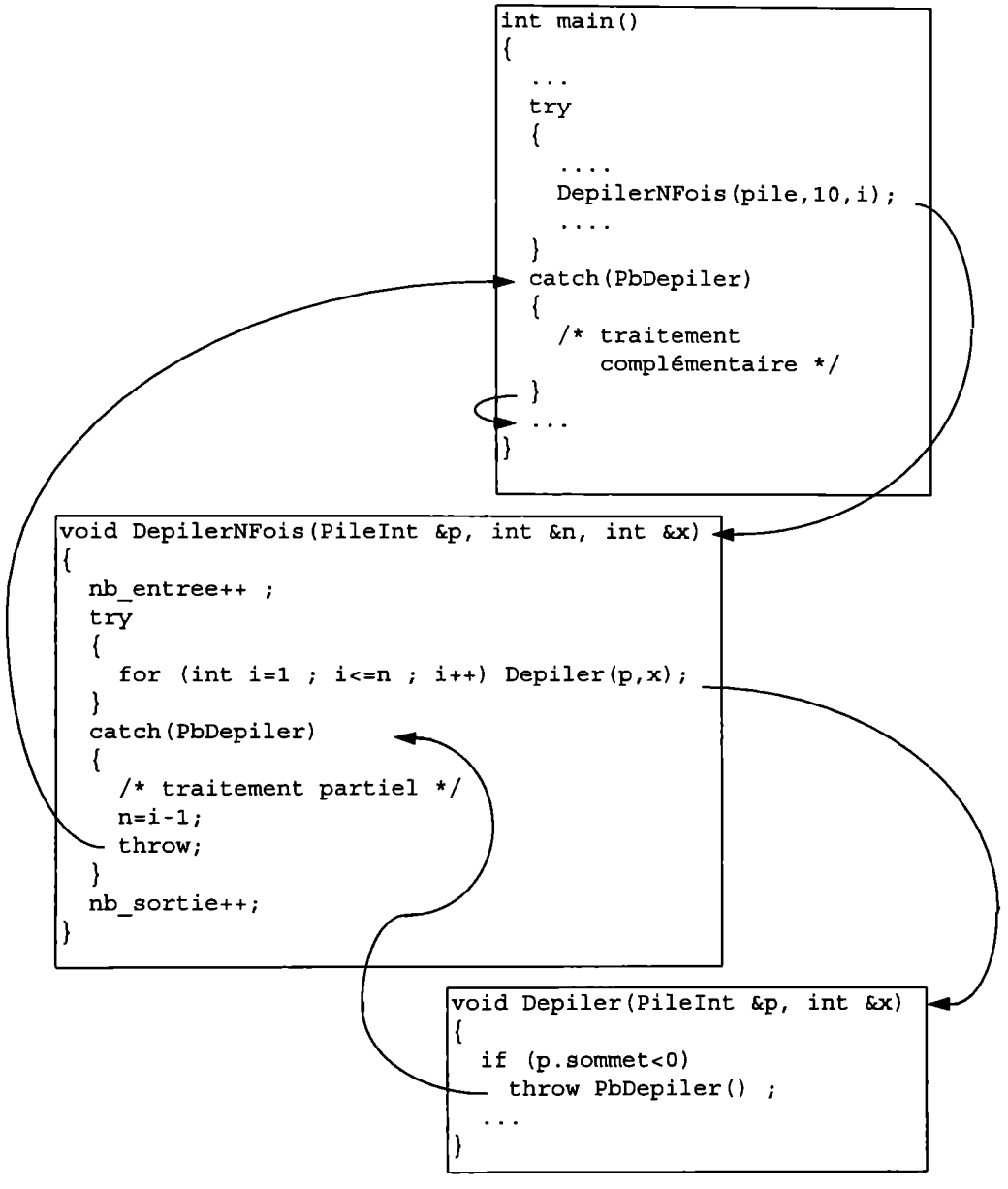
Ici, lorsque l'appelant fournit un second argument trop grand, ce dernier est ramené à la valeur la plus grande possible, `nb_sortie` est incrémenté normalement, et aucune erreur n'est vue de l'appelant. Dans cet exemple, `nb_entree` et `nb_sortie` ont donc toujours la même valeur hors de `DepilerNFois`.



On peut vouloir aussi corriger l'erreur, tout en la signalant à l'appelant :

```
void DepilerNFois(PileInt &p, int &n, int &x)
{
  int i ;
  static int nb_entree=0, nb_sortie=0 ;

  nb_entree++ ;
  try
  {
    for (i=1 ; i<=n ; i++) Depiler(p,x) ;
  }
  catch(PbDepiler)
  {
    n = i-1 ; throw ;
  }
  nb_sortie++ ;
}
```

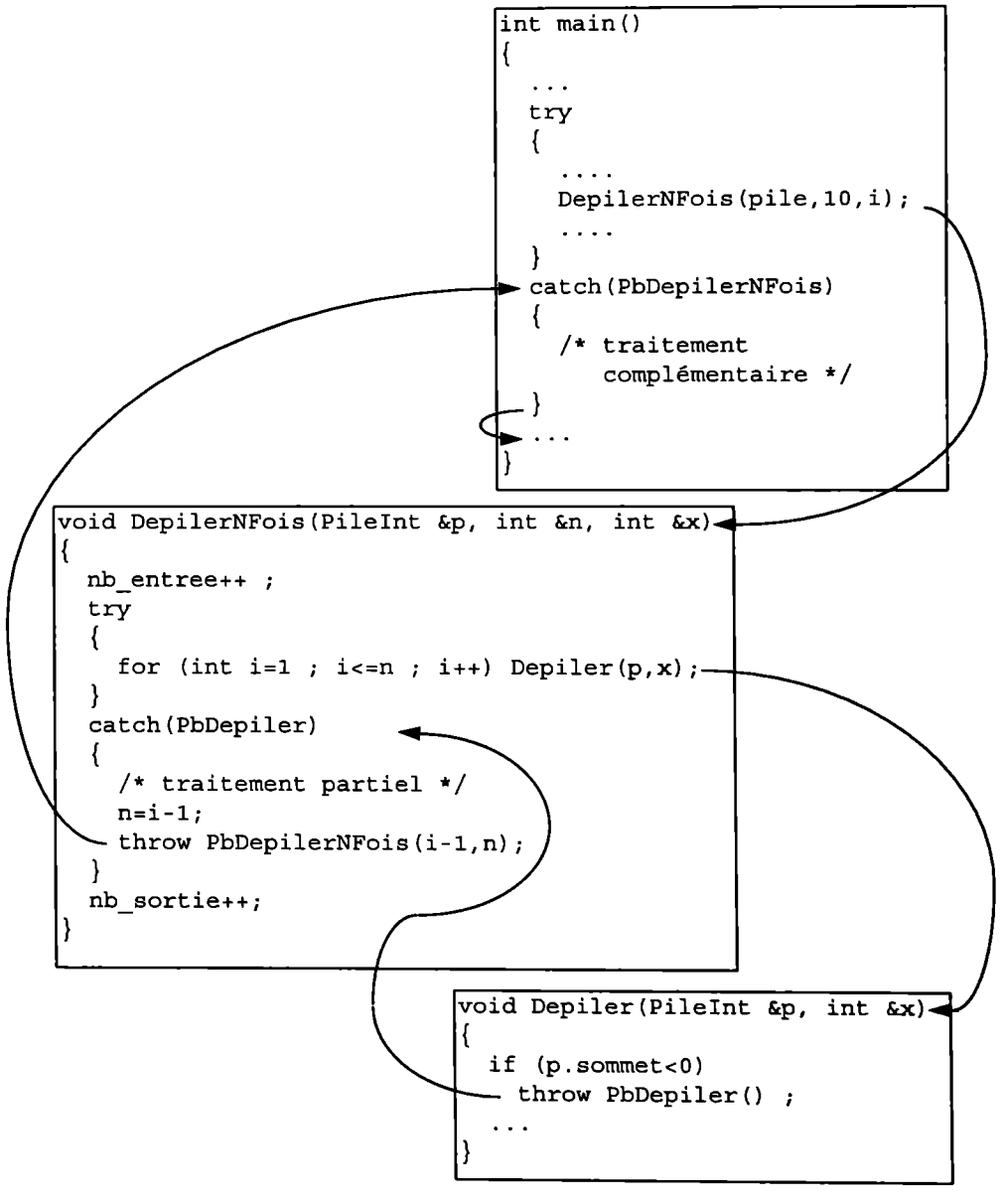
L'erreur est corrigée partiellement, et l'exception continue sa remontée à travers la pile des appels. Ici, `nb_entree` et `nb_sortie` n'ont donc pas forcément la même valeur hors de `DepilerNFois`, et la différence `nb_entree-nb_sortie` correspond au nombre d'interceptions de l'exception `PbDepiler` :



L'exception peut également se transformer en cours de route :

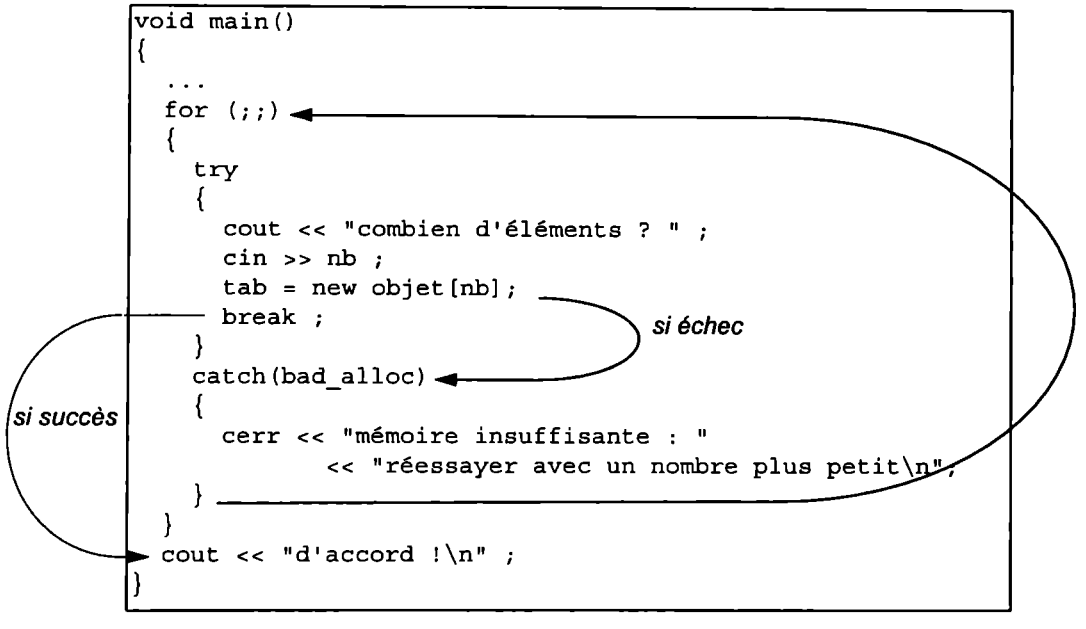
```
void DepilerNFois(PileInt &p, int &n, int &x)
{
  int i ; static int nb_entree=0, nb_sortie=0 ;

  nb_entree++ ;
  try { for (i=1 ; i<=n ; i++) Depiler(p,x) ; }
  catch(PbDepiler)
  { n = i-1 ; throw PbDepilerNFois(i-1,n) ; }
  nb_sortie++ ;
}
```

Là encore, `nb_entree` et `nb_sortie` n'ont donc pas forcément la même valeur hors de `DepilerNFois`, et la différence `nb_entree-nb_sortie` correspond au nombre d'interceptions de l'exception `PbDepiler` :



On peut enfin vouloir corriger quelque chose, puis réessayer. Dans l'exemple suivant, l'utilisateur doit fournir un nombre jusqu'à ce que l'allocation réussisse :

```
void main()
{
  Objet *tab ;
  int nb ;

  for (;;)
  {
    try
    {
      cout << "combien d'éléments ? " ;
      cin >> nb ;
      tab = new Objet[nb] ;
      break ;                          // pour sortir du for
    }
    catch(bad_alloc)
    {
      cerr << "mémoire insuffisante : "
           << "réessayer avec un nombre plus petit\n" ;
    }
  }
  cout << "d'accord !\n" ;
  /* ... */
}
```



3 - Libération de ressources

Le saut du point de détection d'une situation anormale au point de traitement de l'erreur entraîne des sorties intempestives d'un certain nombre de blocs. Ces sorties intempestives provoquent la destruction de tout objet automatique créé dans les blocs abandonnés : les destructeurs de ces objets sont appelés normalement.

Mais pour les autres objets, en particulier les objets alloués dynamiquement, la destruction n'a pas lieu :

```
{
  FILE *f = fopen("fichier","w") ;

  if (/* ... */) throw runtime_error(/* ... */) ;

  fclose(f) ;
}
```

Ici, si l'exception est lancée, on sort du bloc sans refermer le fichier.

Ainsi, de manière générale, des objets qui n'ont plus lieu d'exister vont subsister à chaque fois que leur destruction aura été court-circuitée par le déclenchement d'une exception, et consommer inutilement des ressources.

Dans ce problème, connu sous le nom de *problème d'acquisition/libération de ressources*, un traitement se décompose en trois phases : la première est l'acquisition des ressources nécessaires au traitement, la seconde est l'utilisation des ressources acquises, et la troisième est la libération de ces ressources.

```
{
  // acquisition ressource 1
  // acquisition ressource 2
  // acquisition ressource n

  // début de l'utilisation des ressources
    /* ... */
  // fin de l'utilisation des ressources

  // libération ressource n
  // libération ressource 2
  // libération ressource 1
}
```

La difficulté consiste alors à s'assurer que toute ressource acquise est libérée à la fin du traitement, quelle que soit l'origine de cette fin.

Une solution générale à ce problème consiste à intercepter toutes les exceptions pouvant être lancées, et à libérer dans le gestionnaire les ressources, avant de relancer l'exception :

```
{
  // acquisition ressource 1
  // acquisition ressource 2
  // acquisition ressource n

  try
  {
    // début de l'utilisation des ressources
    /* ... */
    // fin de l'utilisation des ressources
  }
  catch(...)
  {
    // libération ressource n
    // libération ressource 2
    // libération ressource 1
    throw ;
  }

  // libération ressource n
  // libération ressource 2
  // libération ressource 1
}
```

Elle donne sur l'exemple précédent :

```
{
  FILE *f = fopen("fichier","w") ;

  try
  {
    /* ... */
    if (/* ... */) throw runtime_error(/* ... */) ;
    /* ... */
  }
  catch (...)
  {
    fclose(f) ;
    throw ;
  }

  fclose(f) ;
}
```

Cette solution générale est néanmoins fastidieuse et peu élégante. Elle suppose en plus que toutes les ressources ont pu être acquises.

Une autre solution, plus fine, consiste à acquérir chaque ressource dans un constructeur, et à libérer la ressource dans le destructeur correspondant. Le destructeur étant appelé aussi bien en cas de sortie normale que de sortie sur

exception, cela assure une libération totale des ressources. De plus, cela résout le problème des exceptions survenant pendant la phase d'acquisition : si une exception est lancée en cours d'acquisition des ressources, les ressources acquises et elles seules sont libérées. Cette technique est appelée *acquisition de ressources par initialisation.* Elle donne sur l'exemple précédent :

```
class Fichier
{
    FILE *f ;

  public :
    Fichier(char *nom, char *mode)
                                    { f = fopen(nom,mode) ; }
    ~Fichier() { fclose(f) ; }
    /* ... */
} ;
/* ... */

void g()
{
  Fichier f("fichier","w") ;
  /* ... */
  if (/* ... */) throw runtime_error(" ... ") ;
  /* ... */
}
```

Ici, l'ouverture du fichier se fait dans le constructeur de `Fichier`, la fermeture dans le destructeur : lorsque l'exception est lancée, la variable automatique `f` est détruite, le destructeur est donc appelé et le fichier refermé.

Le problème est identique dans l'exemple suivant :

```
{
  Objet *o = new Objet ;
  /* ... */
  if (/* ... */) throw runtime_error("...") ;
  /* ... */
  delete o ;
}
```

Si l'exception est lancée, on sort du bloc sans que la mémoire allouée soit libérée. L'application de la technique précédente donne :

```
template <class T>
class Allocateur
{
    T *ptr ;
    Allocateur(const Allocateur &) {}
    void operator=(const Allocateur &) {}

  public :
    Allocateur(size_t n=1) { ptr = new T[n] ; }
    ~Allocateur() { delete[] ptr ; }
    operator T*() const { return ptr ; }
} ;
/* ... */

void f()
{
  Allocateur<Objet> zone ;
  /* ... */
  Objet *o=zone ;                    // appelle operator Objet*
  /* ... */
  if (/* ... */) throw runtime_error("...") ;
  /* ... */
}
```

La zone allouée est libérée sur sortie normale aussi bien que sur sortie exceptionnelle.

Acquisition de ressources dans un constructeur

Il est fréquent qu'un objet ait lui-même besoin d'acquérir des ressources pour pouvoir exister. Dans ce cas, l'acquisition a lieu dans le constructeur de l'objet :

```
class C
{
  public :
    C() {
          /* acquérir les ressources */
          /* initialiser l'objet */
        }
    ~C() { /* libérer les ressources */ }
} ;
```

Mais un problème se pose si une exception est lancée dans le constructeur. En effet, par définition, un objet est considéré construit seulement si son constructeur a été complètement exécuté. Et seuls les destructeurs des objets complètement construits sont appelés. Par conséquent, si une exception est lancée à partir du constructeur, le destructeur ne sera pas appelé, donc les ressources acquises dans le constructeur avant le lancement de l'exception ne seront pas libérées :

```
class A
{
    int *tab1 ;
    char *tab2 ;

  public :
    A(int i, int j)
        {
          tab1 = new int[i] ;
          tab2 = new char[j] ;

          /* initialisation */
        }
    ~A() {
          delete[] tab2 ;
          delete[] tab1 ;
        }
    /* ... */
} ;
```

Si une exception est lancée dans la partie initialisation du constructeur, l'objet n'est pas complètement construit, son destructeur ne sera donc pas appelé. Par conséquent la mémoire allouée pour `tab1` et `tab2` ne sera pas récupérée.

Même chose si une exception est lancée lors de l'acquisition des ressources : si l'allocation pour `tab2` échoue, la mémoire réservée pour `tab1` ne sera jamais récupérée.

La technique pour résoudre ce problème consiste à acquérir les ressources avant d'entrer dans le constructeur :

```
class C
{
  public :
    C() : /* acquérir les ressources */
        {
          /* initialiser l'objet */
        }
    ~C() { /* libérer les ressources */ }
} ;
```

Dans ce cas, si une exception est lancée à l'intérieur du constructeur, les ressources sont correctement libérées. De même, si l'acquisition d'une ressource échoue, les ressources acquises et elles seules sont libérées.

Cela donne sur l'exemple précédent :

```
class A
{
    Allocateur<int> tab1 ;
    Allocateur<char> tab2 ;

  public :
    A(int i, int j) : tab1(i), tab2(j)
                                  {  /* initialisation */ }
    ~A() { }
    /* ... */
} ;
```

Ici, si une exception est lancée au cours de l'initialisation, les destructeurs de `tab1` et `tab2` seront appelés (mais pas celui de `A`), libérant ainsi la mémoire allouée. De même, si la création de `tab2` échoue, le destructeur de `tab1` sera quand-même appelé.

Outre sa simplicité, cette solution a le mérite de décharger le programmeur de toute gestion explicite des exceptions dans le constructeur.

4 - Intégrité des objets

Une attention doit être accordée au respect de l'intégrité des objets dont le fonctionnement est susceptible d'être interrompu par un lancement d'exception : les objets doivent rester cohérents, quels que soient les points de lancement des exceptions. Reprenons la classe `String` :

```
class String
{
    char *str ;

  public :
    String(char *s) ;
    String &operator=(const String &s)
      {
        if (&s!=this)
        {
          delete[] str ;
          str = new char[strlen(s.str)+1] ;
                                    // bad_alloc lancée ??
          strcpy(str,s.str) ;
        }
        return *this ;
      }
      ~String() { delete[] str ; }
  /* ... */
} ;
```

Cet exemple semble correct. Pourtant, si l'instruction new échoue, l'exception bad_alloc est lancée et provoque la sortie de operator=. Dans ce cas, l'objet courant reste dans une situation incohérente : la zone pointée par son membre str est libérée, mais le pointeur conserve sa valeur et rien n'indique que l'objet est dans un état incorrect. Cela conduit inéluctablement à des dysfonctionnements :

```
{
  String s1(""), s2("Coulée de Serrant") ;

  try
  {
    s1 = s2 ;
  }
  catch(...)
  {
     cout << "échec d'affectation" ;
  }
  cin << s1 ;   // désastre, s1 est dans un état incorrect
}
```

s1 est dans un état incorrect si l'opérateur = lance bad_alloc. Par conséquent, toute utilisation de s1 après une affectation infructueuse est désastreuse.

Des précautions doivent être prises pour éviter de telles situations :

```
class String
{
    char *str ;

  public :
    String &operator=(const String &s)
       {
         if (&s!=this)
         {
           char *ptr ;
           try
           {
             ptr = new char[strlen(s.str)+1] ;
           }
           catch(bad_alloc)
           { throw ; }
           delete[] str ;
           str = ptr ;
           strcpy(str,s.str) ;
         }
         return *this ;
       }
  /* ... */
} ;
```

Maintenant, si l'affectation échoue, l'objet courant conserve son ancienne valeur et reste intègre.

5 - Conclusion

Le mécanisme de gestion des exceptions offre des possibilités intéressantes : c'est un moyen élégant de passer des informations d'un point de détection d'une anomalie à un niveau dans lequel la gestion de cette anomalie est naturelle.

L'objectif final est de définir des systèmes résistant aux pannes : chaque niveau traite les exceptions qu'il peut traiter, et laisse aux niveaux supérieurs le soin de gérer les autres. Si aucun niveau ne peut résoudre entièrement le problème, `terminate` est exécutée.

Mais l'utilisation de ce mécanisme implique, contrairement aux apparences, un travail non négligeable et une réflexion approfondie quant à la restauration systématique de l'environnement dans son état initial et quant au respect de l'intégrité des données.

La structure de contrôle `throw/try/catch` n'étant pas locale, elle nécessite la définition d'une stratégie globale : les différentes parties d'un programme doivent être d'accord sur la façon dont les exceptions sont utilisées.

Bien que la construction `throw/try/catch` puisse être considérée comme une structure de contrôle ordinaire, des utilisations sans rapport avec la gestion des erreurs, est à proscrire :

```
class Tab
{
    int v[100] ;

  public :
    int &operator[](int i)
       {
         if (i<0 || i>=100)
           throw runtime_error("hors limite") ;
         return v[i] ;
       }
} ;
```

```
void f()
{
  Tab t ;

  try                                  // remplissage du tableau
  {
    for (int i=0 ; ; i++) cin >> t[i] ;
  }
  catch(...) {}                          // fin du remplissage

  try                                  // parcours du tableau
  {
    for (int i=0 ; ; i++) cout << t[i] << ' ' ;
  }
  catch(...) {}                            // fin du parcours
}
```

Dans cette fonction `f`, la structure `throw/try/catch` est utilisée pour terminer l'itération. Une telle utilisation est déconseillée.

`throw/try/catch` provoque des sauts entre différents points du code, des sorties en milieu de blocs... Une utilisation massive de `throw/try/catch` déstructure donc sensiblement le programme. L'utilisation de `throw/try/catch` doit par conséquent être réservée, comme son nom l'indique, aux cas exceptionnels.

XI - LES MODELES DANS LEUR CONTEXTE

1 - Amies d'un modèle de classe

Une fonction amie d'un modèle de classe est amie de toutes les instances du modèle :

```
template <class T>
class A
{
  friend void f() ;
  int i ;
} ;
```

Ici, `f` est amie de toutes les instances de `A` :

```
void f()
{
  A<int> a ;
  A<float> b ;
  a.i = 1 ;                  // d'accord, f est amie de A<int>
  b.i = 1 ;             // bien, f est aussi amie de A<float>
  /* ... */
}
```

De même, une classe amie d'un modèle de classe est amie de toutes les instances du modèle.

De la même façon, une instance de modèle peut être amie d'une classe ou d'un modèle de classe :

```
template <class T> class B ;

template <class T>
class A
{
  friend class B<int> ;
  int i ;
} ;
```

Ici, l'instance `B<int>` est amie de toutes les instances du modèle `A` :

```
class B<int>
{
  A<float> x ;
  void f() { x.i = 1 ; }                    // pas de problème
} ;
```

mais les autres instances du modèle `B` ne sont pas amies du modèle `A` :

```
class B<float>
{
  A<bool> x ;
  void g() { x.i = 0 ; }                    // illégal
} ;
```

Un modèle peut être ami d'un autre modèle :

```
template <class T>
class A
{
  friend void g(A<T>) ;
  friend class B<T> ;
  int i ;
} ;
```

Ici, le modèle de fonction `g` et le modèle de classe `B` sont amis du modèle `A`. Chaque instance de `A` possède une fonction amie instanciée à partir de `g` avec la même valeur de `T` :

```
template <class T>
void g(A<T> a)
{
  A<float> b ;
  a.i = 1 ;                                 // correct
  b.i = 1 ;      // illégal quand T est différent de float
}
```

et chaque instance de la classe `A` possède également une classe amie instance de `B` :

```
template <class T>
class B
{
  void f()
  {
    A<T> a ;
    a.i = 1 ;                               // parfait
  }
```

```
    void g()
    {
      A<int> a ;
      a.i = 1 ;                    // erreur quand T n'est pas int
    }
} ;
```

2 - Membres statiques d'un modèle de classe

Chaque instance d'un modèle de classe a son propre jeu de membres statiques :

```
template <class T> class M
{
    static int nb_objet ;
    /* ... */
} ;
```

Ici, chaque instance du modèle `M` possède sa propre instance de membre `nb_objet`.

Toutes les instances d'un membre statique peuvent être initialisées de façon générique, à condition qu'elles soient de même type :

```
template <class T> int M<T>::nb_objet=0 ;
```

Cela n'est pas possible si ces instances ne sont pas de même type :

```
template <class T> class M
{
    static T t ;
    /* ... */
} ;
```

Chaque instance du membre `t` a ici un type particulier. Dans ce cas, les différentes instances de `t` doivent être initialisées spécifiquement pour chaque valeur de `T`. Cependant, un sous-ensemble de ces instances peut être initialisé de façon générique :

```
template <class T> T M<T>::t=0 ;
```

Cette initialisation est valable pour tout type `T` acceptant `0` pour valeur. Cela permet d'instancier les classes suivantes :

```
M<int> m1 ;                    // M<int>::t est initialisé à 0
M<float> m2 ;                  // M<float>::t est initialisé à 0.0
M<void *> m3 ;                 // M<void *>::t est initialisé à 0
```

Par contre, l'instanciation :

```
struct S {} ;
M<S> m4 ;                                        // non, illégal
```

est illégale, car `M<S>::t` ne peut pas être initialisé à `0`. Il est alors nécessaire de spécialiser l'initialisation de `M<S>::t` :

```
S s ;
S M<S>::t=s ;

M<S> m4 ;                                    // d'accord maintenant
```

3 - Modèle de classe et héritage

Une classe peut hériter d'une instance d'un modèle de classe. A partir de la liste générique suivante :

```
template<class T>
class List
{
  public :
    T &Current () ;

    void Insert ( const T &t ) ;
    void Delete () ;
  /* ... */
} ;
```

il est possible de créer une pile d'entiers :

```
class StackInt : private List<int>
{
  public :
    void Push(int k) { Insert(k) ; }
    int Pop() { int k = Current() ; Delete() ; return k ;
}
  /* ... */
} ;
```

Mais on peut conserver la généricité au travers de l'héritage : un modèle peut hériter d'un autre modèle. L'exemple suivant crée une pile générique à partir d'une liste générique :

```
template<class T>
class Stack : private List<T>
{
  public :
    void Push(T k) { Insert(k) ; }
    T Pop() { T k = Current() ; Delete() ; return k ; }
  /* ... */
} ;
```

Inversement, d'une classe peut être dérivé un modèle. Ce type d'héritage permet de factoriser du code, et d'éviter ainsi la génération de plusieurs exemplaires de code identique à chaque instanciation du modèle.

En effet, dans l'exemple précédent, chaque instanciation du modèle `List` génère une instance de chacune de ses fonctions membres. Cela peut être particulièrement coûteux, si les fonctions ont une taille importante :

```
List<int> l1 ;
List<float> l2 ;
List<char *> l3 ;
```

conduit à l'instanciation de trois fonctions `Current`, trois fonctions `Insert` et trois fonctions `Delete`.

On peut éviter cette redondance en regroupant dans une classe de base tous les membres du modèle qui sont indépendants des paramètres du modèle :

```
class BaseList
{
  public :
    void *Current () ;

    void Insert ( void * ) ;
    void Delete () ;
  /* ... */
} ;
```

La classe `BaseList` modélise une liste générique : elle ne manipule que des pointeurs génériques vers les éléments mémorisés. Mais cette classe est intrinsèquement dangereuse, car elle passe complètement au travers du système de type de C++. Et son utilisation nécessite de nombreux *casts* de la part de l'utilisateur.

Cette classe est néanmoins intéressante : son utilisation en tant que classe de base, pour des classes rétablissant un contrôle de type strict, va permettre de factoriser un maximum de code.

En effet, le modèle `List` précédent peut s'obtenir simplement par héritage à partir de `BaseList` de la façon suivante :

```
template <class T>
class List : private BaseList
{
  public :
    T &Current () { return * (T *) BaseList::Current() ; }
    void Insert ( T &t ) { BaseList::Insert(&t) ; }
    using BaseList::Delete ;
  /* ... */
} ;
```

Toutes les instances de `List` partagent maintenant un code commun : celui de `BaseList`. De plus, aucun code n'est généré, lors de l'instanciation du modèle

`List`, pour les fonctions membres en ligne et triviales, telles que `Current` et `Insert` (`BaseList::Delete` n'a même pas besoin d'être redéfinie, mais simplement repositionnée en public dans `List`).

4 - Méta-programmes

Grâce aux modèles, un compilateur C++ peut être utilisé comme un interpréteur. Des itérations et des tests peuvent être effectués à la compilation. Il est ainsi possible, à partir d'un sous-ensemble d'expressions C++, d'écrire des programmes qui sont partiellement interprétés à la compilation[132]. Cela peut améliorer de façon significative la vitesse d'exécution de certains programmes.

Le programme suivant utilise l'opération n^p :

```
long puissance(int n, int p)
{
  return p==0 ? 1 : n*puissance(n,p-1) ;
}

void main()
{
  const long k = puissance(2,10) ;
  /* ... */
}
```

Si les arguments de `puissance` sont, à chaque appel, des constantes, alors les n^p eux-mêmes sont des constantes. Leur valeur étant connue dès la compilation, autant les évaluer à ce moment.

Le préprocesseur n'est pas assez puissant pour réaliser ce type de calcul : les macros n'acceptent pas d'expressions récursives. Par contre, le compilateur peut réaliser ce genre de calcul :

```
template <int N,int P>
inline long Puissance()
{
  return Puissance<P==1?0:N , P-1>() * N ;
}

inline long Puissance<0,0>()
{
  return 1 ;
}
```

[132] Certains de ces programmes sont même complètement interprétés, et n'ont pas besoin d'être exécutés du tout : ils donnent leurs résultats à la compilation ! L'un d'eux circula au sein du comité ANSI/ISO : il donnait les nombres premiers à la compilation (la compilation générait des *warnings* contenant les nombres premiers).

```
void main()
{
  const long k = Puissance<2,10>() ;
  /* ... */
}
```

Ici, le calcul de 2^{10} est réalisé à la compilation. Pour instancier `Puissance<2,10>`, le compilateur à besoin de `Puissance<2,9>`, et ainsi de suite, jusqu'à `Puissance<2,1>`. L'instanciation de `Puissance<2,1>` nécessite l'expression de `Puissance<0,0>`, qui est donnée par la spécialisation, ce qui met fin au processus récursif. Le compilateur exécute donc une boucle pour évaluer 2^{10} à la compilation.

Le code généré pour le programme ci-dessus est strictement équivalent à ce qui aurait été généré pour le programme :

```
void main()
{
  const long k=1024 ;
  /* ... */
}
```

Cette technique peut être combinée avec du code C++ normal. On obtient alors un programme hybride composé d'un programme traditionnel évalué à l'exécution, et d'un *méta-programme* évalué à la compilation.

Modifions le programme précédent, afin d'autoriser un `n` non constant : seules les puissances `p` demeurent constantes. Les n^p ne sont plus des constantes, leur valeur est inconnue à la compilation. Par contre le nombre de multiplications à effectuer est connu. Par conséquent, l'itération peut être *déroulée* à la compilation :

```
template <int P>
inline long Puissance(int n)
{
  return Puissance<P-1>(n)*n ;
}

inline long Puissance<0>(int)
{
  return 1 ;
}

void f(int k)
{
  const long i = Puissance<10>(k) ;
  /* ... */
}
```

Le code généré pour le programme ci-dessus est équivalent à celui qui aurait été généré pour :

```
void f(int k)
{
  const long i = k*k*k*k*k*k*k*k*k*k ;
  /* ... */
}
```

Les multiplications sont effectuées à l'exécution, mais la récursivité est développée à la compilation.

Cette technique de génération de code permet, lorsque certains paramètres (comme la taille des données) sont constants, de *développer* certains algorithmes. Un algorithme de tri, par exemple, peut être développé de cette façon si le nombre d'éléments à trier est connu dès la compilation.

Des *méta-structures de contrôle* peuvent être implémentées à l'aide de cette technique. Les structures itératives sont implémentées récursivement, et les structures conditionnelles à l'aide de spécialisations.

Le méta-programme suivant annule tous les éléments d'un tableau possédant un indice paire, et oppose la valeur des autres :

```
template<bool> void IF(int tab[], int n) {}

inline void IF<true>(int tab[], int n)
{ tab[n] *= -1 ; }

inline void IF<false>(int tab[], int n)
{ tab[n] = 0 ; }

template<int N>
inline void FOR(int tab[])
{
  IF<N%2>(tab,N) ;
  FOR<N-1>(tab) ;
}

inline void FOR<-1>(int tab[]) {}

const int N=10 ;

void main()
{
  int *tab, i ;

  tab = new int[N] ;
  for (i=0 ; i<N ; i++) cin >> tab[i] ;
  FOR<N-1>(tab) ;
  for (i=0 ; i<N ; i++) cout << tab[i] << " " ;
}
```

Si l'on rentre :

```
1 2 3 4 5 6 7 8 9 10
```

on obtient :

```
0 -2 0 -4 0 -6 0 -8 0 -10
```

Cela ne fonctionne bien sûr que pour des tableaux dont la taille est connue à la compilation. Le code généré est équivalent à celui qui aurait été généré pour :

```
const int N=10 ;

void main()
{
  int *tab, i ;

  tab = new int[N] ;
  for (i=0 ; i<N ; i++) cin >> tab[i] ;
  tab[9] *= -1 ; tab[8] = 0 ;
  tab[7] *= -1 ; tab[6] = 0 ;
  tab[5] *= -1 ; tab[4] = 0 ;
  tab[3] *= -1 ; tab[2] = 0 ;
  tab[1] *= -1 ; tab[0] = 0 ;
  for (i=0 ; i<N ; i++) cout << tab[i] << " " ;
}
```

5 - Exercices

Exercice 23

Généraliser le type abstrait `EnsembleAbstrait` défini en page 136.

Généraliser le type `LEnsemble` (exercice 19 page 139), en l'implémentant à partir du type abstrait `EnsembleAbstrait` précédent et de la structure de données fondamentale `ListeGenerique` (exercice 22 page 153).

Exercice 24

Généraliser le type abstrait `PileAbstraite` de l'exercice 20 page 139. Implémenter un type pile `TPile` à partir du type abstrait `PileAbstraite` précédent et de la structure de données fondamentale `TableauGenerique` (voir page 146).

Implémenter un type pile `LPile` à partir du type abstrait `PileAbstraite` précédent et de la structure de données fondamentale `ListeGenerique` (exercice 22 page 153).

XII - EXPLOITATION DES STREAMS

1 - Surcharge des opérateurs d'entrée/sortie << et >>

La classe `ostream` définit l'opérateur `<<` pour écrire les valeurs de tous les types prédéfinis du langage, suivant un format adéquat.

Pour pouvoir écrire sur un flux un nouveau type `T` avec l'opérateur `<<`, il suffit de définir une fonction globale[133] ayant un prototype correspondant à l'une des formes suivantes :

```
ostream &operator<< (ostream &, T) ;
ostream &operator<< (ostream &, T&) ;
ostream &operator<< (ostream &, const T&) ;
```

cette fonction pouvant être éventuellement amie de la classe `T`. L'opérateur `<<` renvoie une référence sur le flux sollicité, ce qui permet de cumuler syntaxiquement les sorties :

```
cout << i << j << k ;
```

Partant de la classe :

```
class Complexe
{
    double x, y ;

  public :
    Complexe(double xx, double yy) : x(xx), y(yy) {}
  /* ... */
} ;
```

pour pouvoir envoyer des objets `Complexe` sur un `ostream`, il suffit de créer la fonction :

[133] Ce ne peut pas être une fonction membre de la classe `ostream`, puisque celle-ci est figée dans la bibliothèque standard.

```
ostream &operator<< (ostream &o, const Complexe &z)
{
  return o << z.x << '+' << z.y << 'i' ;
}
```

cette fonction étant déclarée amie de la classe `Complexe`, ce qui permet ensuite d'écrire :

```
Complexe z(2,3) ;
cout << z << '\n' ;                          // écrit 2+3i
```

Cette fonction est un peu trop simple pour réaliser des sorties acceptables. Toute amélioration est envisageable :

```
ostream &operator<< (ostream &o, const Complexe &z)
{
  if (z.x!=0)
  {
    o << z.x ;
    if (z.y>0) cout << '+' ;
  }
  if (z.y!=0) cout << z.y << 'i' ;
  return o  ;
}
```

Symétriquement, la classe `istream` définit l'opérateur `>>` pour lire toute valeur de type prédéfini du langage suivant un format adéquat.

Pour lire un nouveau type `T` sur un flux avec l'opérateur `>>`, il suffit de définir une fonction globale[134] ayant le prototype suivant :

```
istream &operator>> (istream &, T&) ;
```

cette fonction pouvant être éventuellement amie de la classe `T`. L'opérateur `>>` renvoie une référence sur le flux sollicité, ce qui permet de cumuler syntaxiquement les entrées :

```
cin >> i >> j >> k ;
```

Pour pouvoir extraire des objets `Complexe` d'un `istream`, il suffit de créer une fonction telle que :

[134] Ce ne peut pas être une fonction membre de la classe `istream`, puisque celle-ci est figée dans la bibliothèque standard.

```
istream &operator>> (istream &i, Complexe &z)
{
  char c ;

  i >> z.x ;
  i.get(c) ;                                  // caractère suivant
  switch(c)
  {
    case '+' : i >> z.y ;
               i.get(c) ;
               if (c!='i') throw PbFormat() ;
               break ;
    case '-' : i >> z.y ; z.y *= -1 ;
               i.get(c) ;
               if (c!='i') throw PbFormat() ;
               break ;
    case 'i' : z.y = z.x ; z.x = 0 ;
               break ;
    default  : z.y = 0 ;
               i.putback(c) ;
  }
  return i ;
}
```

avec `PbFormat` définie par exemple comme :

```
class PbFormat : public ios_base::failure
{
  public :
    PbFormat() : ios_base::failure("erreur de format") {}
} ;
```

La fonction `operator>>` étant déclarée amie de la classe `Complexe`, cela permet d'écrire :

```
Complexe z ;
/* ... */
cin >> z ;
```

2 - Comment écrire un manipulateur

Le manipulateur n'est pas un nouveau concept du langage. Il désigne simplement une syntaxe particulière d'appel de fonction. Il n'est pas lié particulièrement aux flux et peut s'utiliser avec n'importe quelle classe et n'importe quel opérateur binaire.

On distingue les manipulateurs sans paramètre, comme `dec`, `hex`, `endl`, et les manipulateurs paramètrés, comme `setw`, `setprecision` ou `setfill`.

i. Manipulateurs sans paramètre

Le principe des manipulateurs sans paramètre est le suivant : si une classe `C` désire accepter des manipulateurs sans paramètre pour un opérateur `@` binaire, elle définit une fonction membre

```
C &operator @ (C &(*pf)(C&))
{
  return (*pf)(*this) ;
} ;
```

Un manipulateur `M` est alors simplement une fonction dont le prototype est : `C& M(C&)`. En effet, si `c` est de type `C`, l'expression :

```
c @ M ;
```

équivalente à :

```
c.operator @ (M) ;
```

est interprétée, d'après la définition ci-dessus de l'opérateur `@`, comme :

```
M(c) ;
```

Et effectivement, la classe `istream` a bien une fonction membre :

```
istream &operator>> (istream &(*pf)(istream &)) ;
```

et la classe `ostream` a une fonction membre :

```
ostream &operator<< (ostream &(*pf)(ostream &)) ;
```

ce qui leur permet d'accepter les manipulateurs pour les opérateurs << et >>.

Pour écrire un nouveau manipulateur `tab` qui envoie `80` caractères `'*'` dans un `ostream`, il suffit de créer la fonction :

```
ostream &etoiles(ostream &os)
{
  for (int i=0 ; i<80 ; i++) os << '*' ;
  return os ;
}
```

Une expression telle que :

```
cout << etoiles ;
```

est alors interprétée comme :

```
etoiles(cout) ;
```

ii. Manipulateurs paramètrés

L'astuce précédente ne marche pas si le manipulateur a besoin d'arguments, car ceux-ci ne peuvent pas être passés à la fonction via l'opérateur.

Le principe des manipulateurs paramètrés est donc différent. L'idée est de créer un objet temporaire à chaque appel du manipulateur, puis d'envoyer cet objet anonyme à l'opérateur. La création d'un tel manipulateur `M` nécessite donc :

— la création d'une classe `M` ayant un constructeur muni des mêmes paramètres que le manipulateur souhaité,

— la création d'une fonction globale `operator @` ayant `C` pour premier paramètre et `M` pour second paramètre, et réalisant le traitement voulu.

Dans ce cas, une expression :

```
c @ M(arg) ;
```

est interprétée comme :

```
operator @(c, M(arg)) ;
```

Pour réaliser un manipulateur `blanc(int n)` qui envoie `n` espaces dans un `ostream`, il suffit de créer la classe et la fonction globale suivantes :

```
class blanc
{
    int n ;

  public :
    blanc(int i) : n(i) {}
    friend ostream &operator<<(ostream &o, blanc &m)
      {
        for (int i=0 ; i<m.n ; i++) o << ' ' ;
        return o ;
      }
} ;
```

Une expression telle que :

```
cout << blanc(10) ;
```

est alors interprétée comme :

```
operator<< (cout,blanc(10)) ;
```

iii. Manipulateurs implémentés à partir de modèles

Une autre implémentation des manipulateurs est possible à partir des modèles.

Le principe est le suivant : si une classe `C` désire accepter des manipulateurs pour un opérateur `@` binaire, elle définit un modèle de fonction :

```
template <class Manip>
C &operator @(C &c, Manip manip)
{
  manip(c) ;
  return c ;
}
```

Un manipulateur sans paramètre `M` est alors simplement une fonction dont le prototype est : `C &M(C &)`, et un manipulateur paramètré `MP` est une classe-fonction ayant un constructeur muni des mêmes paramètres que ceux du manipulateur, et une fonction `C &operator()(C &)`.

Dans ce cas,

```
c @ MP(arg) ;
```

équivalent à :

```
@(c,MP(arg)) ;
```

est interprétée, d'après le modèle de fonction ci-dessus, comme :

```
MP(arg)(c) ;
```

L'exemple suivant implémente une calculatrice fonctionnant en notation polonaise. Elle dispose d'une pile de quatre registres. Les opérations se font en sommet de pile. Elle dispose également de quatre mémoires. Les instructions sont passées à la calculatrice à l'aide de l'opérateur `<<`. Par exemple, `c << 9` empile la valeur `9` dans les registres de la calculatrice `c`.

Les opérations addition, soustraction, multiplication, division et opposé sont disponibles. Par exemple, `c << plus` effectue une somme en sommet de pile, et `c << opp` replace le sommet de pile par son opposé. Ces opérations sont implémentées comme des manipulateurs sans paramètre.

Des fonctions d'impression, d'effacement du sommet de pile et de vidage de pile sont implémentées aussi comme des manipulateurs sans paramètre. Par exemple, `c << print` écrit l'état de la calculatrice, `c << C` vide la pile et `c << CE` efface le sommet de pile.

Des fonctions de stockage et de déstockage sont disponibles. Par exemple `c << STO(0)` stocke le sommet de pile en mémoire `0`. De même, `c << RCL(0)` ramène le contenu de la mémoire `0` en sommet de pile. Ces fonctions sont implémentées sous forme de manipulateurs paramètrés.

Cela donne :

```
class Calc ;

                // définition des manipulateurs paramètrés
struct STO                                          // stockage
{
    int n ;
    STO(int nn) : n(nn) {}
    Calc &operator()(Calc &c) ;
} ;

struct RCL                                          // déstockage
{
    int n ;
    RCL(int nn) : n(nn) {}
    Calc &operator()(Calc &c) ;
} ;

class Calc                                          // calculatrice
{
    friend Calc &plus(Calc &c) ;
    friend Calc &moins(Calc &c) ;
    friend Calc &fois(Calc &c) ;
    friend Calc &div(Calc &c) ;
    friend Calc &opp(Calc &c) ;
    friend Calc &CE(Calc &c) ;
    friend Calc &C(Calc &c) ;
    friend Calc &print(Calc &c) ;
    friend Calc &STO::operator()(Calc &c) ;
    friend Calc &RCL::operator()(Calc &c) ;

    double reg[4] ;
    double memo[4] ;
    ostream &os ;

    void empiler()
      {
        for (int i=1 ; i<4 ; i++) reg[i-1] = reg[i] ;
      }
    void depiler()
      {
        for (int i=3 ; i>0 ; i--) reg[i] = reg[i-1] ;
      }
```

```
  public :
    Calc(ostream &o) : os(o)
      {
        for (int i=0 ; i<4 ; i++) reg[i] = memo[i] = 0 ;
      }
    Calc &operator<<(double x)
      {
        empiler() ;
        reg[3] = x ;
        return *this ;
      }
} ;

                    // implémentation des fonctions membres
                              // des manipulateurs paramètrés
Calc &STO::operator()(Calc &c)
{
  c.memo[n] = c.reg[3] ;
  return c ;
}

Calc &RCL::operator()(Calc &c)
{
  c.empiler() ;
  c.reg[3] = c.memo[n] ;
  return c ;
}

            // définition des manipulateurs sans paramètre
Calc &plus(Calc &c)                              // addition
{
  c.reg[2] += c.reg[3] ;
  c.depiler() ;
  return c ;
}

Calc &moins(Calc &c)                          // soustraction
{
  c.reg[2] -= c.reg[3] ;
  c.depiler() ;
  return c ;
}

Calc &fois(Calc &c)                          // multiplication
{
  c.reg[2] *= c.reg[3] ;
  c.depiler() ;
  return c ;
}
```

```
Calc &div(Calc &c)                                    // division
{
  c.reg[2] /= c.reg[3] ;
  c.depiler() ;
  return c ;
}

Calc &opp(Calc &c)                                    // opposé
{
  c.reg[3] *= -1 ;
  return c ;
}

Calc &C(Calc &c)                                  // effacement pile
{
  for (int i=0 ; i<4 ; i++) c.reg[i] = 0 ;
  return c ;
}

Calc &CE(Calc &c)                 // effacement sommet de pile
{
  c.reg[3] = 0 ;
  return c ;
}

Calc &print(Calc &c)                                  // édition
{
  c.os << "reg(" << c.reg[0] ;
  for (int i=1 ; i<4 ; i++) c.os << ',' << c.reg[i] ;
  c.os << ")\tmemo(" << c.memo[0] ;
  for (int i=1 ; i<4 ; i++) c.os << ',' << c.memo[i] ;
  c.os << ")\n" ;

  return c ;
}

                          // interprétation des manipulateurs
template <class Manip>
Calc &operator<<(Calc &c, Manip manip)
{
  manip(c) ;
  return c ;
}
```

Cette calculatrice s'utilise de la façon suivante :

```
void main()
{
  Calc calc(cout) ;

  calc
    << 1. << 2.
    << print                  // reg(0,0,1,2) memo(0,0,0,0)
    << plus
    << print                  // reg(0,0,0,3) memo(0,0,0,0)
    << 4.
    << print                  // reg(0,0,3,4) memo(0,0,0,0)
    << fois << STO(1)
    << print                 // reg(0,0,0,12) memo(0,12,0,0)
    << 1. << 2. << 3. << 4.
    << print                  // reg(1,2,3,4) memo(0,12,0,0)
    << plus << plus << plus << STO(2)
    << print                // reg(1,1,1,10) memo(0,12,10,0)
    << 2. << 3. << 4.
    << print                // reg(10,2,3,4) memo(0,12,10,0)
    << plus << fois << plus
    << print              // reg(10,10,10,24) memo(0,12,10,0)
    << RCL(1)
    << print              // reg(10,10,24,12) memo(0,12,10,0)
    << div << RCL(2)
    << print               // reg(10,10,2,10) memo(0,12,10,0)
    << opp
    << print              // reg(10,10,2,-10) memo(0,12,10,0)
    << plus
    << print              // reg(10,10,10,-8) memo(0,12,10,0)
    << CE
    << print               // reg(10,10,10,0) memo(0,12,10,0)
    << C
    << print                  // reg(0,0,0,0) memo(0,12,10,0)
    ;
}
```

3 - Exercices

Exercice 25

Etendre la classe `Rationnel` (exercice 13 page 102) pour permettre qu'un objet de cette classe puisse être injecté dans un `stream` et extrait d'un `stream`.

Exercice 26

Etendre la classe `String` (exercice 14 page 102) pour permettre qu'un objet de cette classe puisse être injecté dans un `stream` et extrait d'un `stream`.

Exercice 27

Etendre la classe `Duree` (exercice 11 page 98) pour permettre qu'un objet de cette classe puisse être injecté dans un `stream` et extrait d'un `stream`.

Exercice 28

Etendre la classe `Date` (exercice 12 page 98) pour permettre qu'un objet de cette classe puisse être injecté dans un `stream` et extrait d'un `stream`.

XIII - PIEGES EN VRAC

1 - Pièges lexicaux

Les pièges lexicaux du langage C existent toujours en C++. Ils sont bien connus et ont été inventoriés dans la littérature[135]. Nous n'y revenons pas. Par contre de nouveaux pièges lexicaux, propres à C++, sont apparus.

L'un d'eux se manifeste lors de l'instanciation d'un modèle, lorsque le dernier argument est lui-même une instance de modèle. Par exemple, si l'on crée, avec le modèle `template<class A, class B> struct pair` de la bibliothèque standard, des paires composées `(x,(y,z))` :

```
pair<int,pair<int,int>> p(/* ... */) ;          // illégal
```

on obtient une erreur de compilation, à cause des deux symboles > juxtaposés, qui sont interprétés comme une seule unité lexicale : l'opérateur de décalage à droite.

Cette définition est interprétée comme :

```
pair
<
int
,
pair
<
int
,
int
>>
p
/* ... */
```

ce qui est syntaxiquement incorrect. Le problème étant identifié, rien n'est plus facile de le corriger, en décomposant la définition :

```
typedef pair<int,int> pair_int ;
pair<int,pair_int> p(/* ... */) ;
```

---

[135] Voir des ouvrages comme [FLR] du même auteur, ou [CTP].

ou simplement en séparant les deux > par un espace :

```
pair<int,pair<int,int> > p(/* ... */) ;
```

L'erreur suivante est du même ordre :

```
void f(char *="") ;                                    // incorrect
```

C'est le prototype d'une fonction `f` ayant un paramètre de type `char *` muni d'une valeur par défaut. Ce prototype provoque une erreur de compilation, à cause des deux symboles `*` et `=` juxtaposés, et interprétés comme une seule unité lexicale : l'opérateur de modification par multiplication. Cela est décomposé en :

```
void
f
(
char
*=
"
"
)
;
```

ce qui est syntaxiquement incorrect. Le problème se résout de la même façon, en séparant ces deux caractères :

```
void f(char * ="") ;
```

La syntaxe du langage C++ étant relativement complexe, et devenant au fil des évolutions de plus en plus obscure, il est inéluctable que d'autres pièges lexicaux apparaîtront au cours des extensions à venir.

2 - Confusions malheureuses

La syntaxe ardue du langage ne facilite pas son apprentissage. Et tant qu'elle n'est pas assimilée, le programmeur, un oeil perdu dans le manuel de référence, l'autre rivé sur la liste des messages désobligeants du compilateur, a la désagréable impression de ne pas maîtriser entièrement la puissance du langage.

Quelques nuits blanches attendent donc le programmeur débutant qui, sans relâche, devra traquer la bogue à l'origine du comportement erratique de son programme, jusqu'à ce qu'il finisse par comprendre que l'on ne pas compter sur le compilateur pour signaler les anomalies.

Illustration :

```
{
  int *ptr = new int(100) ;

  for (int i=0 ; i<100 ; i++) cin >> ptr[i] ;
  /* ... */
}
```

Cet extrait se compile sans problème. Son exécution est pourtant désastreuse... Où est l'erreur ?

Voici le code correct :

```
{
  int *ptr = new int[100] ;

  for (int i=0 ; i<100 ; i++) cin >> ptr[i] ;
  /* ... */
}
```

Cherchez la différence...[136]

Prendre des parenthèses pour des crochets, quoi de plus innocent lorsque l'on découvre C++...

3 - Etourderies coûteuses

Des fautes de frappe peuvent passer inaperçues à la compilation et se révéler à l'origine de dysfonctionnements :

```
class MaClasse
{
    int i, tab[100] ;

  public :
    maClasse() { i = 0 ; }
    void Ajouter(int k) { tab[i++] = k ; }
    /* ... */
} ;

MaClasse x ;
x.Ajouter(20) ;                                    // Aïe !...
```

---

[136] Réponse : dans la première version, l'instruction

```
int *ptr = new int(100) ;
```

crée un entier et l'initialise à 100. La boucle `for` provoque par conséquent un débordement de zone.

Dans la seconde version, l'instruction :

```
int *ptr = new int[100] ;
```

crée bien les 100 entiers souhaités.

Ici, la faute de frappe qui s'est glissée dans le nom du présumé constructeur n'est évidemment pas décelée par le compilateur. Rien ne s'oppose à ce que la classe ait une fonction membre `maClasse`.

L'erreur ne sera pas décelée non plus par le programmeur, puisqu'un constructeur n'est jamais appelé explicitement. Donc, et contrairement aux apparences, il n'y a pas de constructeur dans la classe `MaClasse`, et par conséquent `i` n'est pas initialisé. L'appel `x.Ajouter(20)` est catastrophique.

Cette étourderie n'est pas toujours aussi facile à débusquer. Une solution pour éviter ce problème, est de privilégier les syntaxes spécifiques aux constructeurs :

```
class MaClasse
{
    enum { NB=10 } ;
    int i, tab[NB] ;

  public :
    maClasse() : i(0) {}          // erreur de compilation
    /* ... */
} ;
```

Maintenant, l'erreur est vue par le compilateur : la classe et son constructeur ne portent manifestement pas le même nom !

Autre distraction fâcheuse : faire une faute de frappe dans le nom d'une fonction virtuelle redéfinie :

```
class A
{
  public :
    virtual void NomExtrememementLong() { /* ... */ }
} ;

class B : public A
{
  public :
    void NomExtrememementLong() { /* ... */ }
} ;

class C : public A
{
  public :
    void NomExtrementLong() { /* ... */ }
} ;
```

Là encore, l'erreur qui s'est glissée dans le nom de la fonction virtuelle de la classe `C` n'est pas décelée par le compilateur : rien ne s'oppose à ce que la classe ait une fonction membre de ce nom. Mais le polymorphisme souhaité ne va évidemment pas avoir lieu :

```
int main()
{
  A *ptr1 = new B ;
  ptr1->NomExtrememementLong() ;  // B::NomExtrememementLong()
                                  // résultat conforme aux attentes
  A *ptr2 = new C ;
  ptr2->NomExtrememementLong() ;  // A::NomExtrememementLong()
                                  // résultat non conforme aux attentes
  /* ... */
}
```

Ici, le résultat obtenu n'est certainement pas celui attendu : la fonction appelée via le pointeur `ptr2` est `A::NomExtrememementLong()`, et non pas `C::NomExtrementLong()`.

Les informations d'identification de type peuvent aider à localiser l'erreur :

```
#define DEBUG \
  cout << "type de l'objet courant: " \
       << typeid(*this).name() << '\t' ; \
  cout << "type de this: " \
       << typeid(this).name() << '\n'  ;

class A
{
  public :
    virtual void NomExtrememementLong() { DEBUG /* ... */ }
} ;

class B : public A
{
  public :
    void NomExtrememementLong() { DEBUG /* ... */ }
} ;

class C : public A
{
  public :
    void NomExtrementLong() { DEBUG /* ... */ }
} ;
```

L'exécution de :

```
int main()
{
  A *ptr1 = new B ;
  ptr1->NomExtrememementLong() ;
```

```
    A *ptr2 = new C ;
    ptr2->NomExtrememementLong() ;
    /* ... */
  }
```

génère :

```
type de l'objet courant: B    type de this: B * const
type de l'objet courant: C    type de this: A * const
```

ce qui montre que c'est la fonction membre de `A` qui est appelée pour un objet de type `C`.

Le recours aux fonctions virtuelles pures, lorsque cela est possible, permet au compilateur de détecter ce type d'erreur :

```
class A
{
  public :
    virtual void NomExtrememementLong()=0 ;
} ;

class C : public A
{
  public :
    void NomExtrementLong() { /* ... */ }
} ;

int main()
{
  /* ... */
  C *ptr = new C ;          // erreur de compilation ici,
                            // C est une classe abstraite
  /* ... */
}
```

Ici, la création d'un objet de type `C` est illégale, puisque la classe `C`, qui n'implémente pas la fonction virtuelle pure héritée, est une classe abstraite.

4 - Compléments sur la portée des identificateurs

i. Déclarations dans l'instruction d'initialisation d'un `for`

L'instruction d'initialisation d'une boucle `for` peut contenir une déclaration de variable. La portée de cet identificateur s'étend à toute la boucle :

```
{
  const int MAX=100 ;
  int i=-1, x ;

  for (int i=0 ; i<MAX ; i++) { /* ... */ }
  x = i ;                                    // x vaut -1
}
```

Ici, la variable `i` déclarée dans la boucle n'existe que dans celle-ci[137]. La dernière affectation fait donc référence au `i` déclaré en début de bloc.

Par conséquent, le code suivant est incorrect :

```
for (int i=0 ; i<MAX ; i++)
{
  int i ;                          // erreur, i est redéfini
  /* ... */
}
```

Par contre, on peut écrire :

```
{
  for (int i=0 ; i<MAX ; i++) { /* ... */ }
  for (int i=0 ; i<MAX ; i++)      // un autre i est défini
  { /* ... */ }
}
```

puisque les deux variables `i` sont locales à leur `for` respectif.

ii. Déclarations dans une condition

Il est possible de déclarer un identificateur dans la condition des structures de contrôle `for`, `if`, `do` et `while`. La portée de l'identificateur s'étend à toute la structure de contrôle.

La déclaration est considérée alors comme une expression ayant pour valeur la valeur d'initialisation :

```
char *s ;
int i ;
/* ... */
if ( int c = s[i] )
  cout << c ;
else
  cout << '\n' ;

cout << c ;                    // erreur, c n'existe plus ici
```

[137] Cela diffère de ce qui se passait dans les premières versions de C++, dans lesquelles les variables déclarées dans l'instruction d'initialisation d'une boucle `for` étaient dans la région déclarative contenant la boucle.

Ici, la variable `c` déclarée dans l'expression du `if` conserve la valeur du caractère courant de la chaîne `s`. Sa portée est celle du `if` lui-même.

iii. Déclarations dans un `switch`

La portée d'un identificateur déclaré dans un `switch` suit la règle habituelle : l'identificateur est introduit dans la région déclarative courante.

```
void main()
{
  int i=100, k ;

  cin >> k ;
  switch (k)
  {
    case 1 : int i ;
             i = 0 ;
             cout << i;
             break ;
    case 2 : cout << i;                                    // @?!~#
  }
}
```

Ici, la portée de la variable `i` déclarée dans le `switch` est le bloc externe du `switch`. Elle est donc accessible à partir du point de définition, jusqu'à la fin du `switch`. Par conséquent, si `k` vaut `1`, le programme écrit `0`, mais si `k` vaut `2`, il écrit une valeur indéfinie : celle de la même variable `i` non initialisée. En aucun cas, il écrit `100`, la valeur du `i` défini en début de `main`. La confusion est facile à faire...

Mais si la variable `i` est initialisée, alors la construction devient illégale. En effet, par définition, un saut ne peut jamais court-circuiter l'initialisation d'une variable automatique :

```
void main()
{
  int i=100, k ;

  cin >> k ;
  switch (k)
  {
    case 1 : int i=0 ;
             cout << i;
             break ;
    case 2 : cout << i;        // erreur de compilation ici
  }
}
```

L'étiquette `2` est illégale, car elle permet de court-circuiter l'initialisation de `i`.

Pour que la seconde instruction d'écriture se rapporte au premier `i`, il est nécessaire d'introduire un nouveau bloc :

```
void main()
{
  int i=100, k ;

  cin >> k ;
  switch (k)
  {
    case 1 : {
               int i=0 ;
               cout << i;
               break ;
             }
    case 2 : cout << i;                          // écrit 100
  }
}
```

5 - Surcharge et espaces de noms

La surcharge des fonctions membres d'un espace s'opère suivant les règles habituelles de surcharge de fonctions : les fonctions de même nom définies dans la même région déclarative se surchargent. Ainsi, une directive `using` peut créer une surcharge sur une fonction déclarée dans la région déclarative courante :

```
void f(char) ;
void g(char) ;

namespace N
{
  void f(int) ;
}

using namespace N ;

void main()
{
  f('a') ;                                       // ::f(char)
  f(1) ;                                         // N::f(int)
  g(1) ;                                         // ::g(char)
}
```

Cette caractéristique n'est pas sans danger. Si l'utilisateur n'a pas pris soin d'examiner attentivement le contenu de l'espace `N` (notamment s'il n'en est pas l'auteur, et si `N` est volumineux), il va prendre `f(1)` pour un appel à sa fonction

`f(char)`. Or, une fonction `f(int)` étant définie dans `N`, c'est en réalité un appel à `N::f(int)`.

De même, la mise à jour de l'espace `N` nécessite une certaine prudence. En effet, si la nouvelle version de `N` contient une nouvelle fonction `g(int)`, alors `g(1)` va se transformer, après recompilation, en un appel à `N::g(int)`, alors que c'était auparavant un appel à `A::g(char)`.

Ces phénomènes existaient avant l'introduction des espaces de noms, mais ces derniers ont été introduits justement pour les résoudre. La directive `using`, en introduisant en bloc les identificateurs, les fait resurgir. Pour éviter cela, deux solutions : employer au cas par cas la déclaration `using`, ou utiliser à chaque fois une qualification de portée explicite :

```
void f(char) ;
void g(char) ;

namespace N
{
  void f(int) ;
}

void main()
{
  f('a') ;                                    // f(char)
  ::f(1) ;                                    // f(char)
  N::f('a') ;                                  // f(int)
  N::f(1) ;                                    // f(int)
}
```

6 - Attention : objets temporaires

Les objets temporaires ont, par définition, une durée de vie égale à celle de l'expression la plus englobante contenant l'origine de leur création.

En particulier, les objets temporaires créés lors de l'évaluation d'une simple expression :

```
exp ;
```

sont détruits avant l'instruction suivante. Les objets temporaires créés lors de l'évaluation de la condition d'un `if`, `swith`, `while` et `do` :

```
if (condition) instruction ;
```

sont détruits avant l'évaluation de l'instruction. Les objets temporaires créés lors de l'évaluation de l'une des trois expressions d'un `for` :

```
for (exp1 ; exp2 ; exp3) instruction ;
```

sont détruits en fin d'évaluation respectivement de `exp1`, `exp2` et `exp3`.

Une première exception concerne les objets temporaires créés lors d'une initialisation :

```
T t = exp ;
```

Ils sont détruits avant que `t` soit initialisé, sauf l'objet temporaire créé pour garder la valeur finale de `exp`.

La seconde exception concerne les objets temporaires qui ont initialisé une référence : ils persistent jusqu'à la destruction de la référence, ou jusqu'à la fin de la région déclarative dans laquelle le temporaire a été créé (le premier des deux cas rencontrés). Par exemple, les objets temporaires créés lors de l'évaluation d'un `return` :

```
return exp ;
```

sont détruits avant la sortie de la fonction, sauf pour l'objet temporaire créé pour garder la valeur retournée, qui persiste jusqu'au stockage de cette valeur de retour.

Quoique claire et précise, cette règle peut néanmoins être à l'origine de fâcheuses surprises.

Le danger apparaît, en effet, lorsque la valeur de l'objet temporaire est référencée par un pointeur de durée de vie plus longue que l'objet temporaire.

Prenons la classe `String` :

```
class String
{
    char *str ;

  public :
    String(const char *) ;
    operator const char*() { return str ; }
} ;

// concaténation :
String operator+(const String &, const String &) ;
```

Dans l'extrait suivant :

```
{
  String s1("beurre"), s2("blanc") ;
  const char *s = s1+s2 ;
            // l'objet temporaire (s1+s2) est détruit ici
  /* ... */
  cout << s ;                                    // Ouille !
}
```

les deux chaînes `s1` et `s2` sont concatènées, et `s` pointe vers le résultat. Or un objet temporaire de type `String` est créé pour la valeur renvoyée par l'opérateur de concaténation. D'après les règles énoncées précédemment, cet objet est détruit en fin d'instruction, juste après l'initialisation de `s`. Par conséquent, après son

initialisation, s pointe vers un objet qui n'existe plus : toute manipulation de s est catastrophique.

Une solution consiste à recopier immédiatement l'objet temporaire dans un objet de même durée de vie que s :

```
{
  String s1("beurre"), s2("blanc"), s3("") ;
  const char *s = s3 = s1+s2 ;
  /* ... */
  cout << s ;                                          // Oui
}
```

L'utilisation d'une référence est à cet égard moins risquée : lorsqu'une référence est initialisée avec un objet temporaire, celui-ci se voit attribuer la même durée que la référence :

```
{
  String s1("beurre"), s2("blanc") ;
  String &s3=s1+s2 ;                    // l'objet temporaire
                                       // n'est pas détruit ici
  /* ... */
  cout << s3 ;                                          // ok
}                    // l'objet temporaire est détruit ici
```

L'objet temporaire issu de la concaténation est conservé jusqu'à la fin du bloc.

Les temporaires créés dans une condition existent jusqu'à la fin de la condition. Dans l'exemple suivant :

```
const char *s ;
String s1("Gros"), s2("plant") ;
if (strlen(s=s1+s2)>0 && strlen(s)<100) { /* ... */ }
```

la condition est correcte et fonctionne comme souhaité : l'objet `String` temporaire créé lors de l'évaluation du premier opérande de l'opérateur `&&`, qui conserve le résultat de la concaténation `s1+s2`, existe toujours lors de l'évaluation du second opérande de l'opérateur `&&`.

De même, les temporaires créés dans l'une des expressions de l'opérateur conditionnel `?:` ne sont pas détruits avant la fin de l'évaluation de l'expression complète. L'exemple :

```
int f()
{
  /* ... */
  const char *s ;
  String s1("Gros"), s2("plant") ;
  return strlen(s=s1+s2)>0 ? strlen(s) : -1 ;          // ok
}
```

fonctionne correctement : le temporaire créé lors de l'évaluation du premier opérande de l'opérateur `?:` existe toujours lors de l'évaluation du second opérande.

Annexes

A1 - MOTS RESERVES

Les mots réservés du langage C++ sont :

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
compl	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	explicit
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	not	not_eq	operator
or	or_eq	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	xor
xor_eq			

A2 - OPERATEURS SURCHARGEABLES

Les opérateurs surchargeables sont :

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

BIBLIOGRAPHIE

ACCREDITED STANDARDS COMMITTEE — *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*, September 1995.

[ARM] M.A. ELLIS & B. STROUSTRUP — *The Annoted C++ Reference Manual (ANSI Base Document)* — Addison-Wesley, 1990.

[STR] B. STROUSTRUP — *Le langage C++ , 2e* édition — Addison-Wesley, 1992.

S.B. LIPPMAN — *L'essentiel du C++, 2e* édition — Addison-Wesley.

BORLAND C++ 4.0 — *Guide du programmeur* et *Guide de référence.*

[AHC] B. STROUSTRUP — *A history of C++ : 1979/1991* — ACM SIGPLAN Notices, vol. 28, n° 3, Mars 1993.

T. CARGILL — *Constant member functions* — C++ Report, vol. 5, n° 8, October 1993.

J. LAJOIE — *The new cast notation and the bool data type* — C++ Report, vol. 6, n° 7, September 1994.

J. LAJOIE — *Standard update : type conversions* — C++ Report, vol. 6, n° 3, March/April 1995.

T. VELDHUIZEN — *Using C++ template metaprograms* — C++ Report, vol. 7, n° 4, May 1995.

J. LAJOIE — *Function overload resolution : composing the set of candidate and viable functions* — C++ Report, vol. 7, n° 6, July/August 1995.

B. MEYER — *Conception et programmation par objets : pour du logiciel de qualité* — InterEditions

[FLR] J. CHARBONNEL — *Langage C : les finesses d'un langage redoutable* — Armand Colin, 1992

[CTP] A. KOENIG — *C Traps and Pitfalls* — Addison-Wesley

[K&R] B. KERNIGHAN & D. RITCHIE — *Le langage C*, 2e édition — Masson, 1990.

INDEX

—D—

—E—

MASSON Éditeur
120, boulevard Saint-Germain
75280 Paris Cedex 06
Dépôt légal : mai 1996

SNEL S.A.
Rue Saint-Vincent 12 – 4020 Liège
avril 1996

Manuels Informatiques Masson

Langage C++

la proposition de standard ANSI / ISO expliquée

Le langage C++ est apparu au début des années 80. Destiné à améliorer le langage C, il en conserve les points forts tout en introduisant les concepts modernes de programmation par objets. Son succès fut spectaculaire. C++ est d'ailleurs aujourd'hui l'un des langages les plus utilisés.
Afin de contrôler cet engouement, un processus de normalisation fut initié en 1989 par quatre grands constructeurs informatiques. Objectif : garantir une définition unique et stable du langage. Ces travaux, menés sous l'égide des organismes ANSI et ISO, vont aboutir dans les mois à venir. Ce sont ces résultats que Jacquelin Charbonnel se propose d'analyser et de commenter. Par une approche pédagogique de ce futur standard, l'auteur expose les concepts du langage, en décrit la syntaxe et la sémantique, explique enfin les mécanismes sous-jacents. Mises en garde contre les pièges, conseils pour fiabiliser et optimiser les programmes, discussions sur les différentes stratégies de programmation : le lecteur, supposé maîtriser le langage C, est guidé dans une exploration des possibilités et des subtilités de C++.

Jacquelin Charbonnel est enseignant et ingénieur informaticien à l'École Supérieure de Physique et de Chimie Industrielles de la ville de Paris (ESPCI). Il participe depuis 1991 aux travaux de normalisation de C++.

ISBN : 2-225-85219-7