1. Introduction

Ce cours est orienté sur un outil, le compilateur C++, et un ensemble de concepts gravitant autour. Ce document se focalise sur la syntaxe et les concepts de base, l'utilisation poussée du C++ faisant l'objet d'un autre volume sur l'*Etude de paradigmes informatiques*. Les concepts abordés sont les suivants :

- un langage,
- un compilateur,
- un préprocesseur,
- une méthodologie,
- une algorithmique.

La base sémantique est composée de l'ensemble des fonctionnalités du C++ et de choix techniques tirés de ce langage.

Beaucoup de terminologies peuvent se faire à partir des *paradigmes* [1] du C++. On pourrait caractériser le C++ par les trois points suivants :

- c'est un langage orienté objet et non pas un langage objet au sens pur du terme,
- il offre un domaine d'utilisation très large car il reste en partie un langage procédural et séquentiel,
- il offre des outils nouveaux par rapport au Ç et répond à des concepts "modernes".

On peut résumer les fonctionnalités du C++ par :

- c'est un langage séquentiel amélioré,
- intégrant l'*encapsulation*[2] (au même titre que les langages Ada [Booch,1987a] , Simula-67 [Dahl,1970]),
- permettant la programmation objet (mais au prix de quelques efforts),
- permettant la programmation par acteurs, et autres modèles, avec plus de facilité que des langages ordinaires. De façon générale, c'est une base d'implantation pour d'autres paradigmes.

Il faut bien sûr entourer l'étude de ce langage d'une partie méthodologique :

- il apporte des concepts nouveaux (protection des données - encapsulage, arborescence "invisible" de ressources - héritage, mutation dynamique du code - polymorphisme, types dynamiques - template);
- il intègre le concept des références;
- une connaissance du langage C facilite son utilisation pour l'aspect grammatical, mais il demande un apprentissage particulier du langage pour son aspect *objet*.

La complexité se retrouve dans l'algorithmique utilisée :

demande d'une bonne structuration préalable au programme,

- nécessité d'une algorithmique nouvelle (recherche des relations inter-objets), différente de l'algorithmique séquentielle appliquée habituellement au langage C.

1.1 Contenu du cours

Le cours est découpé en plusieurs parties :

1. une présentation du langage :
    - il faut mettre le C++ dans un cadre historique, en faire la genèse,
    - il n'existe pas de spécifications fixées,
    - présentation de qui l'a fait, comment il a été fait, et dans quels buts,
        — bouleversements à venir,
        — les 3 grandes générations du C++,
        — le langage continue d'évoluer.
    - insertion dans le parc des langages (dans quelle famille le classer),
2. le langage :
    - analyse sémantique,
    - exemples et exercices,
    - le langage de programmation proprement dit,
    - développement et programmation en C++, utilisations du C++.

2. Présentation du langage C++

En premier lieu, le C++ [Stroustrup,1991] est un C [Ritchie,1978] évolué. Une première partie concerne les adjonctions apportées au C (C+), avant d'aborder la section traitant plus particulièrement de l'orientation objet (C++).

2.1 Historique

Ce paragraphe aborde l'historique et la genèse du Ç sans lesquels le C++ ne serait pas.

Tout commence vers 1960. Dans les années 60, avant l'ère informatique, les mathématiciens ont un problème de taille à résoudre : un problème de communication des informations.

Ce problème rentre dans la foulée des travaux de linguistique (Aho, Chomsky), dans un courant de travaux de grammairiens, notamment sur l'étude des grammaires formelles (qui ont donné lex [Lesk,1975] et yacc [Johnson,1975] en ligne directe). Il en résulte la conception d'un langage pseudo-algorithmique : **algol** (celui des mathématiciens) :

- c'est un langage proche du langage humain,
- relativement formalisé,
- permettant d'écrire en phrases des choses relativement claires.

Il résulte de cette formalisation la création d'un langage nouveau permettant de décrire avec finesse des concepts et procédures hors du langage humain.

Ce langage est encore utilisé dans les livres de mathématique ou d'algorithmique (cfr. par exemple le livre "*Graphes et Algorithmes*" [Gondran,1979]).

Le **Projet Algol** dans les années 60 consiste à transformer ce langage mathématique en un langage informatique. Ce projet fait intervenir tout un ensemble de personnes, tels le M.I.T., I.B.M., Honeywell, les Bell Laboratories, ainsi qu'une base internationale, tous travaillant en commissions dans les différents pays concernés. Le but de ce projet est d'arriver à réaliser le *langage parfait*, qui permettrait de tout faire : gestion, scientifique, embarqué. Pour ce faire, on a bénéficié :

- des travaux de recherche théoriques
  - — notamment sur les langages à pile (*working stack*[4]), travail théorique jusqu'alors inexploité,
  - — introduction de la récursivité,
  - — introduction de la notion de langage à bloç avec variables locales,
  - — création dynamique de nouveaux types (au lieu des traditionnels *records*[5]).
- des travaux sur la théorie de la compilation :
  - — techniques d'optimisation : décomposition d'un programme en arborescence de noeuds (décomposition des instructions sous forme d'arbre), algorithmes simples, non polynomiaux, d'optimisation des affectations de registres;
  - — optimiseur dans les passes de compilation (le langage C est le premier langage informatique à le proposer).

Ce projet a demandé 10 années de spécifications avant de donner la définition du langage Algol [Naur,1960; Naur,1963] . Il en résulte que :

- Algol est le premier langage à implémenter le concept de bloç
- il propose une structuration de données d'une souplesse extrême : on peut définir une structure au bit près,
- il inclue le concept de la récursivité,
- et détermine une analyse des relations entre les constituants d'une variable dans un programme : Algol dégage 4 termes constituants.

Constituants d'une variable Algol :

Quels sont les constituants d'une variable pour Algol ? Lorsqu'on a des expressions du type :

```
I = 3
J = I
```

plusieurs concepts sont utilisés. Dans la première expression, le `3` représente une valeur (*right-value*[6]), le `I` représente une *left-value*[7] (un récipient). Dans la seconde expression, le `J` représente la *left-value*, mais le `I` utilisé est-il le même que celui de la première expression ? C'est le représentant d'une valeur. Le `I` est donç dans le programme, une *left-value* (la chose en mémoire) et une *right-value* (la valeur qui est dedans). On en conclut que :

- le premier constituant d'une variable est le récipient : une variable est quelque chose qui peut contenir une valeur. Ce récipient est intuitivement désigné par la *left-value*, mais sans l'être vraiment : le récipient est désigné par le nom `I`;
- le second constituant d'une variable est sa valeur (assimilable à la *right-value*);
- le troisième constituant est défini par l'adresse : l'endroit où se trouve la variable, et son mode d'accès. *Nota bene* : en Ç l'instruction `*I` intègre deux concepts distincts : mettre dans le récipient, et mettre dans la valeur. L'instruction `&I` permet de retrouver l'adresse de la variable nommée `I`;
- le quatrième constituant concerne le nommage : le nom de la variable. Il n'est associé ni à une adresse, ni à une valeur. Quand on parle de `I`, il faut le remplacer par "la valeur associée à la variable" dans le cas d'une *right-value*, et au "récipient" pour la *left-value*. A noter qu'il existe des variables non nommées :
  - — variables fantômes (temporaires, générées lors de la résolution partielle d'expressions : `I = I + 3 * J`)
  - — allocation dynamique (`p = malloc(expr);`). A noter que si on fait ensuite `p++`, cette instruction déréférence la variable `p`. Le pointeur n'est plus représentatif du récipient de départ.

On obtient schématiquement les relations de nommage suivantes :

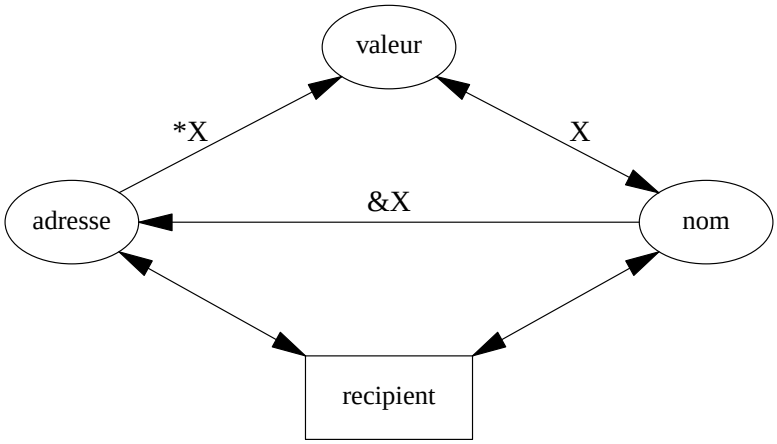


Figure 1. Relation de nommage

Algol permet de pouvoir manipuler tout par tout. Un nom peut changer de visée en cours de travail, via la référence. Le nom est alors considéré comme un des moyens d'accès à la variable au même titre que l'adresse, et par le nom on arrive à modifier la valeur de la variable (notion de déréférenciation).

Dans l'exemple ci-dessous,

```
I
ref J sur I
I = 3
```

si on se donne une variable `I`, et `J` comme étant une référence sur `I`; l'instruction `I = 3` modifie la variable `I`, mais `J` a aussi été modifié. *Remarque :* cet outil, non implémenté en Ç se retrouve dans le C++.

Après 11 ans de travaux, le M.I.T. a sorti une norme : le rapport Algol-68 [Wijngaarden,1969; Wijngaarden,1974; AFCET,1972] . Il représente l'ensemble de tout ce qui concerne les spécifications du compilateur. Mais c'est une norme complexe, sans étude de faisabilité. Les essais d'implantation d'un compilateur complet à l'époque échouèrent tous. Ce n'est que presque 20 ans après, en 1984, à l'université de New-York, que le premier *vrai* compilateur Algol est sorti, qui essaie d'implémenter toute la norme Algol-68 [AFCET,1975] .

Considérons le problème suivant :

```
f(int i)
{
        int t[i];
}
```

où un tableau local est créé sur une taille non connue à la compilation. Ce problème n'est pas résolu par le Ç car, en Ç la taille d'un tableau doit être connue à la compilation. Algol prévoit de générer dynamiquement les variables à l'exécution, en sus de celles reconnues à la compilation. Ada le propose, le C++ aussi.

Au bout d'un certain temps (une dizaine d'années), le projet Algol est déclaré *non-faisable*; progressivement les gens sont retirés du projet. Algol, en tant que langage de programmation, n'existe pas, mais influe sur les langages qui sont ensuite créés.

Le premier langage généré suite à Algol est le langage PL/1 [Berthet,1971] . Ce langage offre des structures de données plus importantes que le C. Puis viendront la création de Pascal [Wirth,1971; ,] , du Ç de Simula-67 [Dahl,1970; Masini,1989] .

A noter qu'à l'époque de la conception d'Algol, on "cache" la machine. Il existe des frontières fermées entre les langages assembleurs, les macro-assembleurs, et des langages dits évolués tels Fortran et Basic. La seule chose de bas-niveau laissée à l'utilisateur est l'indiciation. Le langage Algol réintroduit la notion d'adresse machine via les pointeurs.

On retrouve Algol dans le langage Algol/W [Sites,1972] et aux Bell Laboratories dans des outils tels **adb** (algol debugger, devenu par la suite assembler debugger).

Les travaux réalisés aux Bell Laboratories ont généré toute une famille de langages tirés d'Algol : le langage A, représentant d'un Algol *allégé*, puis les langages B, Ç ..., Y, Z. La technique de validation d'un langage est particulière : lorsqu'un langage est créé, il est diffusé auprès de plusieurs services des Bell Labs. Si le langage est mauvais, on le jette. Pour ne pas perdre de temps, aucune spécification préétablie n'est figée. Deux critères de choix sont mis en place :

- les travaux systèmes,
- les travaux de communications.

Il en résulte deux caractéristiques importantes :

- le langage doit aller vite, plus vite que l'assembleur (la première version du C était 15% plus rapide qu'un programme fait par un "assembleuriste" de haut niveau);
- le langage doit être proche autant que possible du processeur, pour utiliser toutes les instructions disponibles.

On doit maîtriser les temps du programme. Le compilateur de Ritchie permet de compter le temps *cpu*[8] : on est en *one-to-one instruction/delay*[9].

Le langage B est inspiré du BCPL [Richards,1980] (macro-assembleur). Il permet une association d'un langage de haut niveau avec un langage de type macro-assembleur. Ainsi, dans la lignée de production des Bell Labs, Algol et BCPL ont influé sur la conception du langage C.

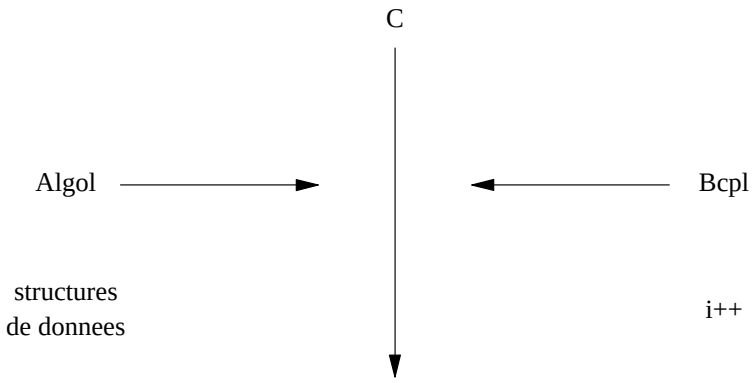


Figure 2. Génération du langage C

De tous ces langages créés aux Bell Labs, seuls resteront utilisés les langages C et Y. Aujourd'hui seul le compilateur C est utilisé universellement.

Comment ces langages sont-ils fait ? Quelles sont les spécifications de l'outil ? Réponse : par une technique de *hacking*[10], par génération successive de compilateurs, et sans spécification fixée à l'avance.

Il n'existe pas de spécification complète du langage C. Il s'agit de l'utilisation d'une grammaire

formelle dont on ne sait pas ce que ça fait. On ne peut pas définir intégralement une grammaire du C. Ce n'est pas un langage LALR(1). La grammaire fournie par Ritchie est émaillée de *raccourcis* qui ne sont pas de la grammaire proprement dite. C'est, du reste, ce qui caractérise la richesse du langage.

Dans le Ç un certain nombre de concepts Algol sont abandonnés :

- tableaux dynamiques,
- *encapsulage*[11] des données,
- redéfinition des opérateurs :
  - le compilateur apprend incrémentalement des définitions de signes,
  - redéfinition des priorités,
  - utilisation de signes arbitraires, avec introduction de nouveaux signes, dynamiquement.
- références.

Les travaux des Bell Labs, avec la création d'Unix [Ritchie,1974] apportent une universalité du langage Ç mais il s'agit d'un C intuitif, sans normes fixes.

Un certain nombre de personnes se penchent alors sur le langage, avec des idées fixes de normalisation : Bull, AT&T, le groupe ISO (comité international de normalisation), qui veulent des spécifications précises sur le langage, normalisées. C'est le projet X3J11 [X3-Secretariat,1980] plus communément connu sous le libellé : *Ansi-C*

- rupture entre le C de base et celui "ansi-ifié" (par exemple l'instruction `i++` en *implémentation*[12] de base est supprimée à terme, car "difficile" à implémenter...). Le langage est spécifié et modifié si nécessaire.
- améliorations (elles sont en déchaussement par rapport aux améliorations du C++). Introduction du prototypage.

Une autre source du langage C++, en sus des langages Algol, Bcpl, et Ç vient du langage Simula-67.

- introduction de la notion d'encapsulage :

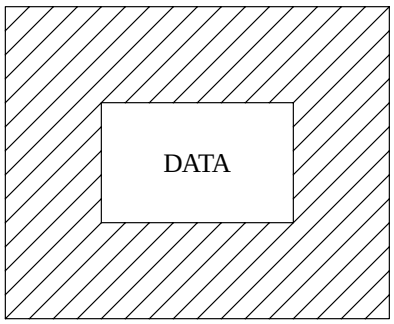


Figure 3. Encapsulage des données

- ce n'est pas un langage objet.

On part du principe que ce qui fait planter un programme ce sont les accès non contrôlés aux données. Simula-67 introduit donc une étanchéification des données, avec un code qui leur sera propre, circonscrivant les données.

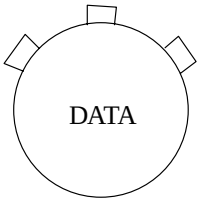


Figure 4. Protection et points d'accès aux données

Cette implémentation fait intervenir plusieurs concepts nouveaux :

- la notion de point d'accès, de "spécification fonctionnelle externe", de "services bien connus",
- la notion de fonctions internes d'implémentation,
- les opérations sur les données sont faites par du code interne,
- l'utilisateur ne connaît que les spécifications externes.

Cela demande une modification dans la conception du compilateur, car il doit rejeter le programme s'il y a un essai d'utilisation d'une spécification interne par un utilisateur externe.

La conception du programme revient à le voir comme un ensemble de données ayant des interactions entre elles. On arrive à la notion de *package*[13] et de *recouvrement*[14].

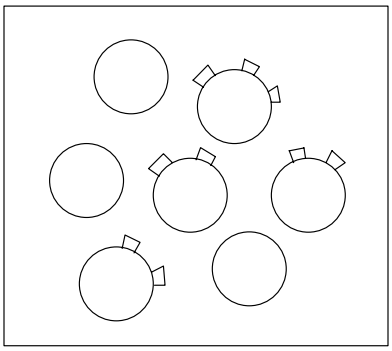


Figure 5. Package et recouvrement

Simula-67 introduit une nouvelle syntaxe permettant d'implémenter l'encapsulage et les services.

Le C++ est un langage des années 80. Il veut résoudre deux idées :

- fournir un complément au Ç pour se rapprocher le plus possible d'Algol ("algolisation" du C) :
  - — références,
  - — redéfinition des opérateurs,
  - — tableaux dynamiques,
  - — prototypage introduit pour un meilleur debugging,
  - — mais il reste séquentiel procédural.
- introduire des paradigmes divers, par influence des langages Simula-67, Smalltalk (Xerox Parc) [Goldberg,1976; Goldberg,1983; Wertz,1987; Lesbros,1989] , Occam [May,1984; Laurens,1987] , et des langages acteurs [Agha,1986] :

— peut-on faire un langage "ouvert" dans lequel il soit possible d'implémenter facilement des concepts nouveaux ?

— peut-on autoriser l'intégration de paradigmes de compilation ?

— que faut-il mettre dans le langage pour implémenter... *tout* ?

Si on considère le langage Ç il ne recouvre pas toutes les possibilités offertes par un langage Basic; il recouvre 98% du Fortran. Comment faire un compilateur qui intègre la notion d'objet, d'acteur, de *message-passing*[15], mais qui ne soit pas un monstre ? Si on prend par exemple, un objet défini par :

- un vecteur d'encapsulage,
- un héritage,
- une présentation,

On ne porte pas l'objet lui-même, mais les outils de base : encapsulage, héritage, et présentation, qui permettent de le définir. Que faut-il implémenter dans le C pour obtenir ce qui est attendu des autres langages ? C'est tout l'effort actuel du C++ d'apporter petit à petit une réponse à cette question.

2.2 Typologie

2.2.1 Compilateur par rapport à interpréteur

Comment déterminer un compilateur ? Il répond aux spécifications suivantes :

- demande à connaître les informations à la compilation qui ne seront pas modifiées à l'exécution;
- est opposé au concept d'exécution. Il n'existe pas de solutions dans certains langages. Par exemple :

  ```
  int tab[i]
  ```

  La valeur de `i` doit être connue à la compilation. Il faut fournir une constante résolue à la compilation.

  ```
  int tab[cte]
  ```

- tout doit être totalement spécifié pour le passage au processeur;
- passes de traduction : aboutissement à l'exécution de *code natif*[16] de la machine sur laquelle on tourne (binaire correspondant au processeur utilisé)

  Question : qu'en est il d'un processeur programmable (cfr. la machine Orion, et la possibilité de se définir dynamiquement de nouveaux opérateur, comme, par exemple l'opérateur `*p++`) ?.

Un interpréteur répond aux caractéristiques suivantes :

- analyse et exécution étroitement mêlés
- liberté de codage
- génère un langage intermédiaire (p-code), peu souvent du code machine
- souplesse d'utilisation
- déclaration dynamique de nouvelles données

- modification dynamique du code
- notion de *run-time*[17]

Le langage C est un langage compilé. Le cas du langage Basic est hybride. Tout se fait à l'exécution. Il s'agit d'un interpréteur. Smalltalk est un autre exemple de langage interprété. Il intègre :

- un mécanisme d'héritage et un langage d'objets très hiérarchisé,
- des objets existant dynamiquement dans le code,
- la réception de messages (le message passe au travers de l'ensemble des noeuds jusqu'à être traité, ou mis à la poubelle par le maître de la hiérarchie).

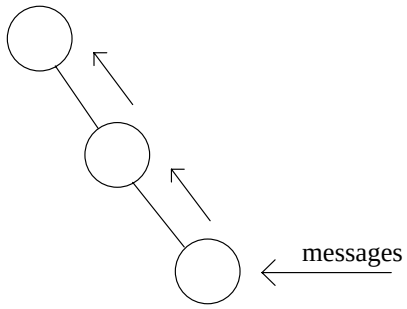


Figure 6. Transmission de messages

Chaque noeud n'a pas conscience de cette arborescence : qui émet le message ? Qui doit le réceptionner ? Ceci est typiquement interprété et n'est pas connu à l'exécution. Si le langage n'avait pas été un interpréteur, il aurait fallu implémenter "en dur" les liens de hiérarchie (héritage); or le lien n'est pas connu à l'avance, il ne peut donc être compilé.

2.2.1.1 Caractéristique technique du C++ :

- c'est un compilateur (au sens défini plus haut),
- mais il possède une partie interprétée (ce n'est pas un *full-compiler*[18]). En effet, le polymorphisme implique que tout ne soit pas défini, ni définissable à la compilation.

2.2.2 Stratégie par rapport à tactique

Il s'agit de distinguer un niveau élevé de programmation (stratégie) de la cuisine de bas niveau (tactique). On a besoin de moyens différents pour programmer ces niveaux (réflexion algorithmique). Aussi cela condamne les langages *univoques* (langages ayant un seul moyen d'expression pour la stratégie et la tactique).

- langage stratégique
  - — programmation descendante
  - — définition de structures de données et des fonctionnalités externes

  - Cfr. les annexes de "*Méthodes de programmation*" de B. Meyer et C. Baudoin [Meyer,1978] , ainsi que l'"*Introduction à la programmation orientée objet*" de Harald Wertz [Wertz,1988] .

- langage tactique
  - structures de contrôle
  - implantation des données et des fonctionnalités

La stratégie intervient dans l'encapsulage, la tactique dans les expressions de type `for( i= ...)`. Un langage tel que Lisp [Winston,1984; Steele,1984; Brooks,1985; Wertz,1989; Greussay,1986] convient à la stratégie. On peut tout faire en Lisp, mais le coût devient monstrueux.

Le langage Prolog [Clocksin,1981] est aussi un langage stratégique :

- on lui fournit des règles, des assertions proches du langage humain
- la programmation n'est pas spécifiée (interprétation)
- mais les petites choses à réaliser sont coûteuses
- unification des variables (`i = j` devient monstrueux à réaliser, car génère beaucoup d'instructions machines)
- aucun outil ne donne les moyens linguistiques pour faire des opérations simples (ce n'est pas du tout un langage tactique).

Le langage Smalltalk est un langage de haute stratégie, mais est peu fourni en outils tactiques.

Sous le critère tactique/stratégie, les langages forment un continuum :

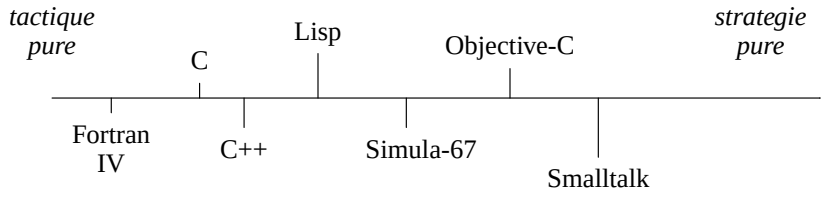


Figure 7. Tactique vs stratégie

Dans tout programme intervient toujours :

- une partie tactique (implémentation)
- une partie stratégique (conception)

Des outils spécifiques de compilation peuvent être utilisés pour chacune des deux parties.

- Franz-Lisp et C : *link*[19] dynamique des fichiers dits objets (radical `.o`)
- Simula-67 : hybridité du langage. Permet de travailler dans les deux sections, souplesse de la notion de programme (implémentation/utilisation)
- Objective-C : intègre la notion d'objet, de classe et d'héritage de Smalltalk, et utilise des outils de base d'implémentation du C
- C++ : séquentiel et rapide, admet un haut niveau de conception. Pondère les deux aspects (tactique/stratégie) sans les abandonner.

2.2.3 Langage procédural par rapport à langage non-procédural

2.2.3.1 Langage procédural (séquentiel)

— les instructions sont déroulées en séquence

```
i = 3;
j = 4;
```

En Ç par exemple, l'instruction `i = 3` est réalisée avant que ne soit faite l'instruction `j = 4`

— il faut prévoir les structures de contrôle permettant de se dérouter

— on peut empaqueter les séquences d'instructions dans un bloc réappelable ailleurs (fonctions)

```
f()
{
        i = 3;
        j = 4;
}

i = 3;
f();
j = 4;
```

Dans un langage procédural, il ne se passe que ce qu'on a effectivement demandé. En contre exemple, le langage neuronal Occam n'est pas un langage séquentiel. Les noeuds de jonction sont les points de synchronisation.

2.2.3.2 Langage non-procédural

— Il existe du code activé sans qu'on le dise.

Dans la séquence d'instructions suivante

```
int i = 3;
fichier += 3;
```

l'objet *fichier* (reconnu à l'exécution ou à la compilation) se dit "le dialogue avec un entier m'est inconnu".

— C'est au langage de faire le nécessaire pour permettre aux objets de se comprendre, de façon dynamique.

— Recherche d'un *XDR* (External Data Representation) [20] permettant de se comprendre (utilisation de librairies de traducteurs).

Dans l'instruction `fichier += 3`, le `fichier` est transformé en *XDR*, ainsi que le `3`, et l'instruction devient une addition entre objets de type *XDR*. Ceci n'est pas écrit explicitement par l'utilisateur dans le programme, mais est exécuté dynamiquement.

Dans un langage procédural, la notion de séquence est un concept fort. Mais on peut rompre, dans certains cas de figure, la séquentiation écrite. Ainsi, le couple `(Ç Unix)` permet, par la gestion des signaux, de faire une programmation par `trap`[21] avec déclenchement à l'exécution.

```
main()
{
        signal();
        pause();
}
```

Cet type de programmation permet de facilement implémenter en C un serveur de traitement.

En C++, du code peut être exécuté avant et après la fonction principale du programme. Dans l'exemple qui suit, la construction de `X` se fait avant l'entrée dans le `main()` (fonction principale, point de départ du programme utilisateur). Sa destruction se fait après la fin du `main()`. Le code rattaché est donc activé en dehors du "programme".

```
extern class x;
x X;

main()
{
        ;
}
```

Le code dépend des instructions mises dans les capsules des objets. Ces capsules sont définies en dehors du programme qui les utilise. A aucun moment on ne "voit" le code qui s'exécute.

2.3 Apport du C++

L'évolution du C vers le C++ vient d'une possibilité de fournir au langage les moyens permettant de réaliser les caractéristiques mentionnées précédemment.

Pour se faire, un certain nombre d'outils ont été rajoutés par rapport au C :

- le *prototypage*[22], permettant de différencier deux codes nommés identiquement,
- les références, permettant la synonymie entre les variables, leur localisation et leur dénomination,
- l'*overload*[23] de fonctions, donnant la possibilité de redéfinir des fonctions,
- l'*inlining*[24], permettant une "macroïsation"[25] des fonctions
- la fonctionalisation des opérateurs (et donc une possibilité d'overloading),
- la définition d'une nouvelle catégorie de type : les classes (`class`),
- leur fonctionnalité d'héritage,
- et de polymorphisme,
- le paramétrage des types `class`.

Ces trois derniers outils permettant en particulier l'implémentation objet (le `++` du C), les autres ne faisant apparaître que des concepts de base dans la lignée de l'évolution d'un langage (un `+` du C).

Au cours des prochains chapitres, nous verrons que les **classes** sont une nouvelle structure de données correspondant à l'encapsulage proposé par Simula-67, et permettant de décrire un ensemble composé de données et de code spécifique qui leur est rattaché (c'est une structure C en plus approfondi).

L'**héritage** est destiné à faire de l'économie de code ("`Y` ressemble à `X`" avec un delta en plus ou en moins). Cela permet de faire de la programmation par défaut, et une réutilisation du code existant en supprimant ce qu'il y a en trop, et en ajoutant ce qui manque.

 ISBN 2-909909-02-6

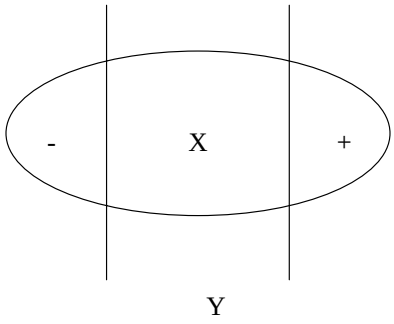


Figure 8. Héritage

On procède à un regroupement d'objets déjà existants, par un *héritage ascendant** (on demande aux parents de fournir des informations à leurs enfants).

Le **polymorphisme** introduit la partie interprétée du compilateur, et implémente un *héritage descendant*.

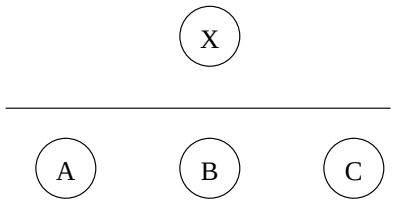


Figure 9. Polymorphisme

Cet héritage n'est connu qu'à l'exécution du programme; il est dynamique et interprété. Il procède à une recherche d'*instances de classe*[26] pour lesquelles on va se matérialiser le temps du travail. Par exemple, si le programme demande à `X` de s'attacher à `A`, il devient `A` à part entière lors de l'exécution, mais, à la compilation, `X` n'a aucune conception de `A`.

Un exemple typique est l'objet servant d'aiguillage pour une gestion d'imprimantes diverses pour un *spooler*[27]. C'est la notion de contexte, d'environnement, qui permet de se rattacher à qui de droit.

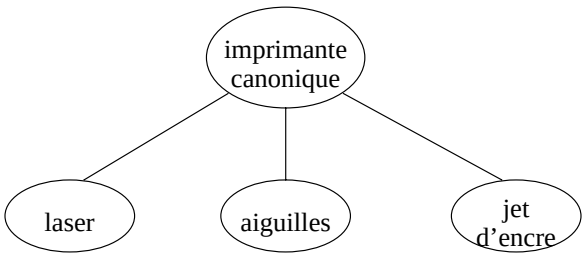


Figure 10. Exemple de polymorphisme

Les **template**s[28] apportent la possibilité de définir des nouveaux types, créés dynamiquement à la

* Cette notion d'*héritage ascendant/descendant* est une notion purement établie par les auteurs de ce document (C.J. Gerrebout et M.F. Detienne).

compilation. On décrit un type comme "base de génération", puis on déclare des objets "définitifs" utilisant cette définition pour exister. Il s'agit de "*types paramétrés*".

2.4 La notion d'objet

La notion d'**objet** est quelque chose de non fixée aujourd'hui, qui varie de langage à langage. Smalltalk, qui est présenté comme *le* langage objet par excellence, n'est pourtant pas un canon de langage objet.

Les caractéristiques principales d'un objet sont :

- l'**encapsulage** (protection des données par rapport aux agressions extérieures)
- l'**héritage** (réutilisation de code, programmation par défaut avec des plus et des moins), faisant l'objet d'une "descendance" (*héritage descendant*).

  Si on a un objet `O`, on dit que `X` hérite de `O`, et forme sa descendance, s'il est constitué des caractéristiques de `O` avec des formes en plus et/ou en moins.

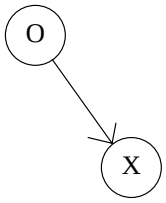


Figure 11. Héritage

- le **polymorphisme**, avec l'*héritage ascendant*, faisant intervenir une notion de *générique*[29].

  Un objet `O` va exister et s'instancier dans un autre objet : soient deux objets définis ailleurs `X` et `Y`. L'objet `O` va s'instancier dans l'un ou l'autre en fonction du contexte d'utilisation.

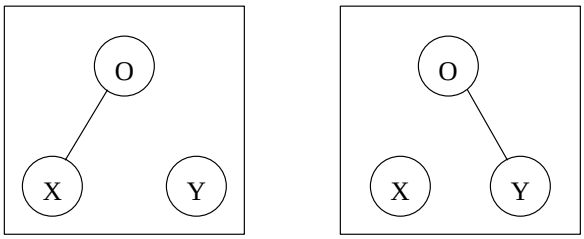


Figure 12. Polymorphisme

On opère sur une classe générique (`O`) qui sera instanciée dans une classe donnée (`X` ou `Y`).

Le concept d'objet fait aussi intervenir la notion de classe abstraite (classe vide, hors de toute définition de données). Cette notion n'est pas connue de tous les langages objets. Elle existe en Smalltalk, et est implémentée en C++.

En C++, l'*objet* est quelque chose d'intuitif. Il n'existe qu'un seul outil rajouté au C permettant de reconnaître la notion d'objet : le concept de `class`. C'est le seul mode de définition d'un *objet* C++.

2.5 Historique du C++

La première occurence du C++ vient en 1984 avec le *class preprocessor*, défini par Bjarne Stroustrup [Stroustrup,1983; Stroustrup,1984; Sinz,1984] , outil qui vient des Bell Laboratories, repris ensuite par AT&T sous Unix.

Il existe une ambiguïté pour cet outil :

- il convertit un programme écrit en C++ en un programme de sources C;
- tout ce qui est écrit en C++ peut être retranscrit en C "normal", permettant ainsi une utilisation des concepts du C++ quel que soit le compilateur "natif" de la machine;
- il s'agit d'une passe supplémentaire de compilation.

Le but est de fournir un outil facile à utiliser, et aisément portable. Le code C++ est transformé en code Ç et est ensuite compilé "sans problèmes" sur toutes les machines dotées d'un compilateur C.

On maintient une qualité du code :

- le préprocesseur est écrit en C++,
- le surcoût réalisé pour un programme ne dépasse pas 10%. Dans certains cas, on a un meilleur résultat (10 à 20% meilleur) par rapport à un programme C. Ceci concerne les cas de figure suivants :
  - — logiciels graphiques,
  - — logiciels embarqués;
- on l'utilise là où la rapidité est nécessaire (optimisation des algorithmes).

Il est intéressant de regarder le code généré par le préprocesseur C++. Mais il vaut mieux ne pas trop regarder les sources du préprocesseur lui-même car son code n'est pas le meilleur qui puisse être.

On pourra remarquer qu'aujourd'hui, personne ne sait ce qu'un compilateur C++ est capable de compiler (il en est de même avec le C d'ailleurs). Il vaut mieux se reporter pour cela au livre de Bjarne Stroustrup (l'auteur du C++), "*The C++ programming language*" [Stroustrup,1991] .

Le problème du préprocesseur des Bell Laboratories vient de sa lenteur pour la traduction du source C++ en source C. Une solution est partiellement apportée avec le C++ version 3.0 qui propose un compilateur, et non plus un simple préprocesseur.

Une demande en C++ existe aussi hors des Bell Laboratories. Une première réponse à ces besoins en C++ vient du monde DOS, avec la réalisation d'un *full-compiler* par **Zortech** :

- compilateur, permettant de générer directement du binaire à partir de sources C++
- très rapide
- la partie C est l'une des plus rapide, mais est spécialisée sur les processeurs Intel
- propose un debugger symbolique "objet"

Zortech a sorti aussi une version tournant sous Unix. Dans la même lignée, on peut trouver le compilateur C++ de **Borland** (sous Dos uniquement, moins performant), de même que le compilateur C++ de MicroSoft.

Une seconde réponse vient aussi du monde DOS, avec le class-preprocessor de **Glockenspiel** (tournant aussi sous Unix).

Une troisième réponse vient de la **Free Software Foundation**, avec le **g++** de Michael Tiemann, dans

la gamme des produits **Gnu**. C'est un *full-compiler*, basé sur le compilateur C de Richard Stallman (**gcc**), portable, mais supposant un environnement gnu**. Une version de ce compilateur existe aussi pour Dos. Avec cet outil et son environnement, les programmes sont *debuggables*[30] au niveau objet (class). S'il n'est pas totalement compatible avec la "norme" sortie par Stroustrup, les sources en sont par contre disponibles, et permettent toute modification (ce qui en fait tout l'attrait).

2.6 Aspect évolutif

Au départ, les gens qui on fait le C++ n'avaient pas une vision "objet" bien établie. En premier lieu, c'était une extension du Ç sans qu'un projet "objet" ne soit arrêté (version 1.2 du C++). Les possibilités du langages étaient assez limitées. Par exemple, on ne peut pas générer de nouveaux opérateurs dans le langage (on se limite à la liste des opérateurs du C).

Puis une nouvelle version est sortie, due aux échanges d'idées (version 2.0), apportant plus de souplesse. Le C++ intéresse une gamme importante de gens dans le monde. Les "fortranistes" ont enfin un intérêt à sortir du monde du Fortran. Les utilisateurs de Smalltalk ("smalltalkiens"), population venant d'un monde objet, sont attirés par l'aspect séquentiel du langage, et par sa rapidité d'exécution.

Ceci a amené une intégration totale des concepts objets, avec une incrémentation de l'aspect interprété du C++ (version 3.0), tout en fournissant de nouveaux outils "C-istes" (définition dynamique de type, détection d'erreurs ...).

Le principe de l'évolution du C++ suit la ligne directrice suivante :

1. Stroustrup, qui dirige le groupe de travail des Bell Labs sur ce chapitre, émet des propositions sous forme de *bêta-release*[31] au travers des *News*[32],
2. la communauté internationnale procède à une mise en oeuvre et à des essais,
3. et retourne acquiessements et/ou dénégations vers les Bell Labs,
4. ce qui produit une nouvelle *release*[33].

Le langage est donc construit sur une base pluraliste (l'essai de changement public/private entre la version 2.0 et 3.0 a été faite sous la pression des bans d'essais). On remarquera que cette procédure avait été la même, à plus petite échelle, pour la création du langage C.

On obtient parfois un résultat nocif : la non-compatibilité entre les différentes releases. C'est un choix volontaire de la part des utilisateurs, car sinon on ne pourrait jamais éliminer les mauvaises orientations du langage.

Il en résulte que le C++ est un langage qui évolue encore aujourd'hui : tout peut être remis en cause du jour au lendemain, et tout ce qui a été écrit devra peut-être être repris.

Remarque : toute proposition d'amélioration, de résolution de problèmes, de sujets nouveaux à intégrer, peut être émise sur les *news*, et sera analysée par l'équipe des Bell Laboratories.

On notera enfin les liens tumultueux qui existent entre le comité Ansi (appuyé par AT&T) et les Bell Labs (soutenues par les gens qui travaillent en C++). Un point critique est illustré par la notion de prototypage.

** Depuis sa version 2.2.2 (juillet 1992), gcc intègre le compilateur Ç le compilateur C++, ainsi que le compilateur Objective-C.

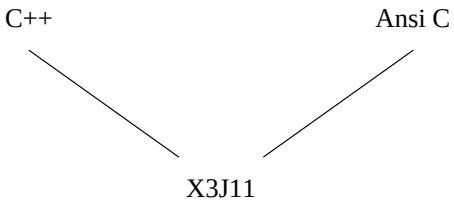


Figure 13. Variantes prototypage

Le prototypage proposé pour X3J11 n'intègre pas toutes les possibilité de reconnaissance attendues. Il n'intègre pas la notion d'*inlining*, ni les *default values*. Stroustrup le dit lui-même, "le C++ ne suit pas la norme Ansi, et là où le C++ doit diverger, il divergera" [Paris, 1991].

On a donc à disposition trois façon de réaliser des programmes

- en "old" C (Ritchie)
- en Ansi C (normalisé)
- en C++ (Stroustrup)

ou encore en **gcc**, intermédiaire entre l'Ansi-C et le C++ (cela peut être suffisant dans certains cas). Ces trois filières existent en divergence.

Un programme C "normal" ne peut pas tourner directement en C++; mais, sous réserve de quelques lignes à écrire, on peut réutiliser et/ou intégrer le code.

Pour plus d'informations à ce sujet, se reporter aux références bibliographiques suivantes : [Stroustrup,1987a; Stroustrup,1987b; Stroustrup,1987c] .

3. C+ : Extension du C

La programmation en C++, lorsqu'on se limite aux seules extensions du Ç reste "classique" (traditionnelle). On traite ici la partie orientation du C vers Algol. Seront abordés le prototypage et les nouveaux mots clef introduits dans le C++. Bjarne Stroustrup donne un résumé simple mais complet des apports du C++ par rapport au C dans un article publié dans les ACM de 1986 sur les langages orientés objet [Stroustrup,1986] .

3.1 Prototypage

Considérons le *synopsis*[34] de fonction suivant :

```
int func(l)
long l;
```

Cet en-tête de fonction peut être décrit comme suit :

- nom de fonction : `func`
- type de retour de la fonction : `int`
- liste des paramètres : `(l)`
- type de chacun des paramètres : `long`

En Ç le nombre de paramètres reste inconnu à la compilation. Rien n'interdit, en compilation séparée, de générer deux versions distinctes de cette fonction `func()`

```
/*------ Premier fichier ------*/
int func(l)
long l;
{
        return (int)(66 * l);
}

/*------ Second fichier ------*/
long func(i)
int i;
{
        return (long)(888888 * i);
}
```

Le nom est la seule référence à la fonction. Mais si on fait une édition de liens, on a un *clash*[35] sur la reconnaissance univoque de la fonction.

D'un point de vue informaticien, on a une différence dans les deux synopsis, et cela semble être représentatif de deux entités différentes.

Mais comment régler le choix de la fonction correspondante lorsqu'on utilise une instruction `func(3)` dans un programme ? Le compilateur peut introduire, tout seul, une opération de *cast* [36] lors de l'analyse de l'instruction, et réaliser par exemple `func((long)3)`. Dans ce cas, le compilateur sait de quoi il parle, et on ne peut plus "bidouiller" sur les passages de paramètres.

Le **prototypage** prend en considération, pour définir pleinement une fonction,

- le nom de la fonction
- la liste des paramètres, avec leur type associé

Remarque : il est à regretter que les compilateurs (de même que le préprocesseur d'AT&T) ne prennent pas aussi en compte le type de retour de la fonction.

Ainsi, les synopsis

```
int func(l)
long l;

int func(i)
int i;
```

représentent deux fonctions différentes. Le compilateur, au moment de la compilation du programme, va regarder quelle est la fonction qui correspond le mieux à l'appel :

`func(3)` : demande l'utilisation de la fonction définie par `int f(int)`

`func(3L)` demande l'utilisation de la fonction définie par `int f(long)`

Si on avait appelé la fonction en lui fournissant un flottant (`func(3.0)`), le compilateur n'aurait pu trouver directement une fonction correspondante. Alors, soit il le dit, soit, s'il existe une possibilité de passage de flottant en long, il fait une conversion automatique, et appelle la fonction correspondante.

On n'a pas toujours les sources de la fonction au moment où on compile (utilisation de librairies). Dans ce cas, lorsque le compilateur compile,

1. il y a génération d'un prototype par analyse de l'appel de fonction,
2. et détection de problèmes (dans ce cas, il jette).

3.1.1 Notation du prototypage

La notation est différente lorsqu'il s'agit

- de la définition de la fonction,
- ou de la déclaration de la fonction.

Le prototypage C++ se réalise en intégrant entre les parenthèses de la liste des paramètres les types de chacun de ces paramètres.

3.1.1.1 Prototypage pour une définition de fonction

Lors de la définition d'une fonction, on fournit le type de la fonction (type de la valeur de retour), ainsi que les paramètres, en en donnant un nom local à la fonction, et on précise le type de chacun de ces paramètres. Dans le cas du Ç les spécifications de type des paramètres se fait par une liste définie hors des parenthèses, et délimitée par l'accolade ouvrante spécifiant le début du code. En C++, les types des paramètres sont donnés immédiatement entre les parenthèses lorsqu'on nomme les paramètres.

```
int func(long l)
{
        ...
}
```

Le prototypage, dans sa forme formelle, dit tout ce qu'il faut savoir.

```
int func(long l, int i, char t[])
{
        ...
}
```

Il existe un cas d'espèce : si on passe en paramètre un pointeur sur fonction. Dans ce cas, il faut prototyper la fonction et aussi le pointeur sur fonction. Une fonction retournant un entier, et recevant en

argument un pointeur sur "fonction retournant un entier et n'ayant pas de paramètres" aura le prototype suivant :

```
int func(int (*p)())
{
        ...
}
```

Une fonction retournant un entier, et recevant en paramètre un pointeur sur "fonction retournant un entier et recevant en paramètre un entier et un double" aura le prototype suivant :

```
int func(int (*p)(int, double))
{
        ...
}
```

Toute fonction qui est utilisée doit avoir, au moment de la compilation, la déclaration intégrale de ses types.

Une fonction retournant un pointeur sur "fonction prenant un caractère en argument, et retournant un entier", et recevant en paramètres un entier et un double, se définira par :

```
int (*func(int (*p)(int,double)))(char)
{
        ...
}
```

3.1.1.2 Prototypage pour une déclaration de fonction

Les déclarations de fonctions se font, comme en Ç avec l'utilisation du mot clé `extern`, mais en indiquant en plus les types des paramètres.

```
extern int func(long, int, char *);
extern int func(long ceci, int cela, char * autre);
```

On peut aussi fournir la liste de paramètres aveç en sus du type, une dénomination (pseudo noms de variables). Cela permet, si on donne des noms lisibles, de donner des informations précises sur le mode d'utilisation de la fonction. Il n'existe pas d'interférences avec le programme.

Le prototypage est un prototypage "fort". Si on a le programme suivant

```
int func(int);

main()
{
        func(3);
}
```

le compilateur sort une erreur, car la fonction `func()` retourne un entier, et le programme n'utilise pas de *left-value* dans l'instruction d'appel. Cette non-utilisation de la valeur de retour de la fonction est considérée comme une erreur potentielle.

Une fonction déclarée sans paramètres ne peut en aucun cas être appelée avec des paramètres. Elle est considérée comme sans possibilité de recevoir des paramètres.

Il convient de faire attention lorsqu'on mélange des fichiers d'`include` C et C++ dans un programme.

3.1.2 Prototypage dynamique

C'est la stratégie utilisée par le class-preprocessor version 1.2. Le compilateur, sur base d'un appel, génère un prototype.

- Instruction d'appel :

  `func( 3L )`

- Prototype généré :

  `void func(long)`

Si, plus loin dans le programme, on a un autre type d'appel, par exemple

```
func( 'a' )
```

on a rejet sur ce second appel, avec un message de prototypage non conforme.

Le laxisme au niveau des données est interdit en C++. Il faut obligatoirement appliquer les conversions de types.

```
func( (long) 'a' )
```

Si maintenant on utilise la fonction de la façon suivante :

```
i = func( 2, 3)
```

cette troisième forme d'appel génère une erreur "fatale", car le nombre de paramètres n'est pas conforme (cela permet de trouver de vieux bugs bien cachés...).

Supposons que l'on ait perdu les sources de la fonction `f()` qui se trouve en librairie dans un fichier `.o`. Comment le compilateur peut-il connaître le prototype de cette fonction ?

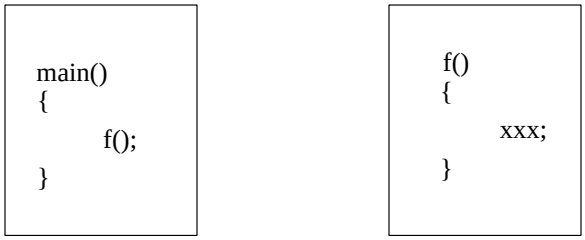


Figure 14. Exemple de programme

On associe un nom au niveau du binaire d'une fonction, du style

```
int funcc(int i, int j)
```

devient nommée, en librairie par quelque chose de la forme `I_f_II`.

Lors de l'analyse de l'instruction d'appel de fonction, il y a génération de nom "prototypé" par construction progressive et ensuite le compilateur compare les noms générés.

3.1.3 Prototypage statique

Ce mode de prototypage exige que la déclaration de la fonction soit donnée avant toute occurence d'utilisation de cette fonction dans le programme.

```
/* declaration de fonction */
```

```
int func(int);

/* programme faisant un appel */
main()
{
        func(3);
}
```

Le prototypage doit être déclaré avant la première occurence d'utilisation. A noter que la déclaration ne peut se faire qu'une seule fois dans un fichier. On ne peut fournir plusieurs prototype différents au même moment. Mais on peut avoir des prototypes différents pour des fichiers compilés séparément.

3.1.4 Prototypage par ellipsis

L'*ellipsis* (point de suspention) permet de ne pas fournir un prototype dans sa totalité.

Considérons le cas d'école du prototypage de la fonction `printf()`. De par sa spécification, cette fonction a un nombre indéterminé de paramètres (dépendant du format fourni en premier paramètre). Si on déclare

```
void printf(char * format)
```

cela signifie que la fonction ne peut prendre qu'un seul paramètre. Or le C++ exige que toute fonction soit totalement définie au niveau des paramètres.

La solution est fournie par la reconnaissance d'un nouveau symbole : `...` (ellipsis); ce symbole ne peut apparaître que dans les prototypes de déclaration de fonction. Il signifie "*un nombre indéterminé de paramètres de types inconnus*". Ainsi, la déclaration de la fonction `printf()` se fera de la façon suivante :

```
void printf(char * format ...);
```

Le nombre de paramètres est variable (mais il y en a au moins un), et les types des paramètres sont aussi variables.

Si on veut spécifier qu'une fonction a au moins deux paramètres, mais dont on ne connaît pas le type du second (ni des suivants) on peut la déclarer comme suit :

```
int func(int premier, ...);
```

L'opérateur de concaténation (,) montre bien l'existence d'une liste composée d'au moins deux éléments.

3.2 Valeurs par défaut

Il s'agit de la définition de valeurs par défaut fournies lors de la déclaration de la fonction.

La fonction supplée elle-même à une valeur pour un paramètre non déclaré à la compilation. A noter que cela demande une compilation séparée de la fonction et de la partie de programme qui en fait la déclaration et l'appel. Un premier fichier contient la définition de la fonction, avec le prototype qui lui correspond

```
int f(int i)
{
        i++;
}
```

Un second fichier contient la déclaration, avec la définition locale des valeurs par défaut, si l'appel ne fournit pas tous les paramètres attendus.

```
/* declaration avec valeur par defaut */
int f(int j = 3);

main()
{
        f(10);
        f();
}
```

Dans cet exemple, le premier appel à la fonction `f()` fournit toutes les informations nécessaires. La fonction est activée avec la valeur `10` en paramètre. Dans le second appel, on ne donne pas le paramètre attendu. La valeur par défaut fournie en définition du prototypage de déclaration va pourvoir à une donnée, et la fonction est activée avec la valeur `3` en paramètre. Ceci permet une programmation "floue".

Mais ce concept, tel qu'il est implémenté aujourd'hui, a des limites : on doit organiser les valeurs par défaut de la droite vers la gauche.

Supposons que l'on ait une fonction définie avec trois arguments :

```
int f(int i, int j, int k)
```

et que `i` et `j` puissent être non fournis par le programmeur, alors le paramètre `k` devra lui aussi être spécifié comme recevant une valeur par défaut. On aimerait pouvoir écrire `f(,,3)` en appel de fonction, ce qui renseignerait le paramètre `k` et laisserait le compilateur prendre les valeurs par défaut pour les premiers paramètres, mais cette notation n'est pas reconnue par le compilateur.

La définition de valeurs par défaut se fait lors de la déclaration de fonction. Si on utilise la compilation séparée, on peut donc fournir différentes valeurs par défaut suivant le cadre d'utilisation d'une même fonction dans des codes appelants différents (les prototypes de déclaration sont différents).

Contenu fichier `a.c` :

```
int f(int i = 10);

int a()
{
        int x;
        x = f();
        return x;
}
```

Contenu fichier `b.c`

```
int f(int i = 7);

int b()
{
        int x;
        x = f();
        return x;
}
```

Contenu fichier `f.c`

```
int f(int i)
{
        return i;
}
```

Dans cet exemple, la fonction `a()` appelle la fonction `f()` avec la valeur `10`, la fonction `b()` l'appelle avec la valeur `7`, la fonction `f()` étant définie comme recevant une valeur entière en

argument.

3.3 Void

Que signifie exactement

```
int f();
```

Cette fonction, dénommée `f()` retourne un entier, et n'a pas de paramètres. Lorsque, dans un programme C (old-C) on utilise une telle instruction, cela correspond uniquement à faire une déclaration de la fonction. On précise que cette fonction retourne un entier. Le C ne prend pas en compte la notion de prototypage des paramètres. En C++, une telle déclaration précise en sus que la fonction n'a pas de paramètres, et ne pourra jamais en avoir pour ce prototype. On aurait pu écrire, en C++ :

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

L'utilisation du mot clé `void` signifie *pas de type*. Donc dans ce cas, la liste de paramètres de la fonction est une liste de paramètres sans type. Or toute variable C++ doit avoir un type associé pour exister. Donc cela signifie que la liste de paramètres est vide (C.Q.F.D. !).

En C++, une fonction définie de type `void` spécifie une fonction qui ne retourne rien. Un contrôle strict sera alors fait sur l'utilisation de l'instruction `return` dans le code de la fonction.

La déclaration

```
void f(void);
```

est typique de la déclaration d'une fonction qui n'a pas de paramètres et ne retourne rien.

A noter que l'on ne peut déclarer une variable de type void (cela n'a aucun sens !).

```
void x; /* error */
```

Qu'en est-il de l'utilisation de `void` dans la définition d'un pointeur ?

```
void * x;
```

Il s'agit d'un pointeur sans taille associée, sans arithmétique (si on utilise l'expression `x++`, le compilateur ne sait pas de combien il doit incrémenter l'adresse). Cela renforce la notion de "*pointeur*", en tant que type particulier, notion différente de "*adresse de*" qui se rattache à une variable préalablement allouée, et donc typée.

Par exemple, si on prend le prototypage de l'appel système `write()`. En C (old-C), son prototype est :

```
int write(fd, buf, siz)
int fd;
char * buf;
unsigned int siz;
```

Le type du paramètre `buf` est défini comme "pointeur sur caractère", mais ceci uniquement pour spécifier "adresse mémoire". Comme l'unité de mesure de la mémoire se fait par le byte, dont la taille correspond à celle du caractère, ceci explique le choix de cette notation. En cas d'utilisation, pour être totalement conforme et portable, ce prototypage demande normalement un forçage de type lors de l'appel à la fonction (cast) :

```
int buffer[10];

write(fd, (char *)buffer, sizeof buffer);
```

Le C++ fait une différence fondamentale entre la notion de pointeur et celle de "adresse d'une variable". Pour le prototypage de la fonction `write()`, le C++ utilise la notion de "pointeur", sans lui

rattacher de type particulier. Le prototype C++ de l'appel `write()` système devient :

```
int write(int fd, void * buf, unsigned int siz)
```

Il n'y a pas besoin de *cast* lors de l'appel à la fonction, car le type du paramètre signifie *pointeur*, mais sans notion explicite du type de la donnée se trouvant à cette adresse.

3.4 Overload de fonctions

Ce concept fait intervenir un nouveau mot clé : `overload`. Ce mot clé, appliqué à un nom de fonction, spécifie que plusieurs définitions seront fournies pour ce nom.

Le compilateur C++ est "intelligent". Il sait reconnaître des prototypes différents, et sait déterminer la différence entre

```
f("hello");     /* --> f(char*) */
f(3);           /* --> f(int) */
```

On veut pouvoir écrire autant de fonctions de même nom, mais ayant potentiellement des codes différents. Cela pose les problèmes :

- qu'en est-il lorsque le compilateur ne trouve pas son bonheur ?
- que faire si on ne veut pas de recouvrement de fonction (surcharge) ?

On va explicitement le spécifier :

- dans une déclaration
- dans une déclaration de liste de prototypes

3.4.1 Dans une déclaration de fonction

On spécifie un recouvrement possible dans le *scope*[37] de la compilation qui suit. On demande au compilateur d'être intelligent :

```
overload f;
int f(int);
int f(char *);

void a()
{
        ( f("hello") && f(3) && f(3.0) );
}
```

On peut ainsi donner plusieurs prototypes pour un même nom de fonction, ce dans un même fichier (ce qui n'est pas possible normalement). Dans ce programme, l'appel à la fonction `f()` avec une chaîne de caractères ou avec un entier est reconnue, par contre l'appel avec un nombre flottant va générer une erreur.

Ayant spécifié que la fonction `f()` était *overloadable*[38], on peut donner plusieurs définitions différentes de cette fonction dans le même fichier :

```
overload f;

int f(char * x)
{
        ...
}

int f(int i)
```

```
{
        ...
}
```

On a un seul *verbe* associé à des données de **types** différents. On approche de la notion de "*well-known service*"[39]. Suite à l'instruction de spécification d'overload sur une fonction `f()` donnée, toutes les fonctions `f()` apparaissant dans le programme sont overloadées. On a donc plusieurs définitions différentes, chacune correspondant à un des prototypes fournis, et univoquement reconnues par celui-ci.

3.4.2 Dans une déclaration de liste de prototypes

Ce mode de déclaration d'*overloading*[40] permet de limiter la possibilité d'overload à une liste précise de prototypes :

```
overload int f(int), f(char*);
```

Une fonction `f()` apparaissant dans le scope de l'overload est interdite (erreur détectée). Il y a limitation au scope de compilation locale. Mais cela n'interdit pas d'avoir des définitions différentes dans les librairies qui sont chargées lors de l'édition de liens.

Remarque

Les nouvelles versions du C++ ont fait disparaître l'obligation d'utilisation du mot clé `overload` appliqué à des fonctions. L'overload est devenu implicite.

```
int f(int);
int f(char);
```

3.5 Inline

Considérons l'utilisation du macroprocesseur `cpp` du C (première passe de compilation) :

```
#define f(x)    ((x)++ * ((x)++))

g()
{
        char *p;
        ...
        return( f(p) );
}
```

Cet exemple illustre un problème : l'occurence multiple du paramètre de la macro dans le code de définition de celle-ci. Si par exemple, on utilise la macro `f()` en lui passant en argument un appel de fonction

```
f( g(*p++) );
```

Cela revient à faire

```
return( g(*p++)++ * g(*p++)++ );
```

La variable `p` est modifiée deux fois, et on active deux fois la fonction `g()`. Ceci n'est pas visible de l'extérieur... Un bon conseil, valable aussi en C : ne jamais utiliser plus d'une fois les paramètres d'une macro dans le code associé.

Que faire si on veut utiliser une variable temporaire dans le code associé à la macro ? Le compilateur `gcc` permet de créer un bloc dans la section de définition de la macro,

```
#define f(x)    ({ int i; t = x; t++; ...})
```

mais cela n'est pas valable pour tous les préprocesseurs. Il ne faut pas oublier que le macroprocesseur reste un substituteur de texte. Les macros ne sont utilisées que pour éviter le temps de chargement d'une fonction. Dans certains cas, pour des codes petits, le temps d'appel de fonction (cfr. le nombre d'instructions machines nécessaires pour le réaliser) est supérieur à celui du code exécuté. On utilise donc les macros pour des raisons d'efficacité. Le code de la macro est directement injecté dans le code appelant, sans avoir à passer par le mécanisme de chargement de fonction.

Un nouveau mot clé du C++ permet de qualifier des fonctions en mode d'utilisation équivalent à celui des macros : `inline`.

- c'est un qualificatif d'allocation réservé aux fonctions (ne s'applique pas à des variables ou des données)
- c'est un qualificatif de fonction définie dans le programme (reconnu dans le scope de compilation du programme)

Si on regarde le code suivant :

```
inline int f(int i)
{
        return( i++ * i++ );
}
```

c'est une fonction au sens du C. Mais c'est une "macro" au sens C++, car le qualificatif `inline` demande à ce que le code soit injecté dans le code appelant, en place et lieu de l'appel de fonction. D'une façon générale, il n'y a pas de procédure d'appel de fonction pour les fonctions *inlinées*[41]. Elles n'apparaissent pas dans la table des symboles du programme compilé. Il est intéressant de produire des fonctions en les qualifiant d'inline de façon paramètrée à la compilation, pour permettre une utilisation fonctionnelle lors d'un *debug*[42] (on ne cherche alors pas l'efficacité d'exécution du code), mais avec possibilité de les utiliser sous forme de macros lors de la génération du binaire définitif.

```
#ifndef Inline
#define Inline
#else
#define Inline inline
#endif

Inline int f(int i)
{
        return( i++ * i++ );
}
```

Dans cet exemple, si on compile le source sans définir la macro `Inline` (qui, du reste, est une "vaie" macro), alors le code généré correspond à celui d'une fonction (car la macro `Inline` est remplacée par rien). Par contre, si on compile en spécifiant une définition de la macro

```
        gcc -DInline foo.c
```

la macro `Inline` est remplacée par le mot clé `inline` par le préprocesseur.

L'utilisation de l'*inlining* peut être dangereux. Ainsi, si dans une application, on a généré une centaine de fonctions définies avec le qualificatif `inline`, comme il y a automatiquement réinjection dans le code, le source final des programmes devient très gros. Les fonctions spécifiées `inline` sont donc un équivalent de macros "solides", mais il faut faire attention à leur utilisation. Le taux d'expension du programme source peut devenir très important sans que l'on s'en rende compte (substitution des appels "fonctions" par leur code). Le temps de compilation du source résultat ainsi que la taille du binaire final peuvent devenir incompatibles avec le contexte d'utilisation.

Face à des qualificatifs d'`inline`, le compilateur applique une certaine intelligence. Considérons le programme sous forme d'un arbre dont les noeuds sont les fonctions appelées. La racine de l'arbre est la fonction `main()`, et les sous-arbres sont composés par les appels de fonctions empilés. L'intelligence du compilateur intervient s'il s'agit d'une fonction feuille, ou si les variables locales peuvent être allouées sur des registres. Alors il ne construit pas de *frame*[43] spécifique pour la "fonction". Par contre si le code de la fonction fait intervenir trop de variables locales, ou s'il inclut un appel de fonction, alors le compilateur construit une frame spécifique dans la *stack*[44]. Ainsi, suivant évaluation, le `inline` peut devenir `{ f(); }` (inclusion dans un bloc).

Le compilateur peut refuser l'inlining s'il estime que le coût de la fonction est trop fort, ou que la complexité de l'inlining dépasse ses compétences (par exemple, le préprocesseur C++ de AT&T refuse d'inliner des fonctions qui incluent une boucle `for`). Mais cela reste transparent pour l'utilisateur.

Ayant intégré ce nouveau concept, que reste-t-il de l'utilisation du macro-processeur dans un code C++ ?

- toutes les macros de type fonction disparaissent

```
#define f(x)     (x++)
```

- on utilise le macro-processeur pour la définition de constantes

```
#define UN 1
```

Remarque : Pour que le concept d'`inline` soit pris en compte lors de la compilation d'un code faisant appel à des fonctions *inlinées*, il faut que le fichier contenant le code à compiler contienne aussi la définition de la fonction macroisée. Il en résulte que les définitions de fonctions *inline* sont mises dans des fichiers inclus dans les fichiers de sources les utilisant (fichier *header*).

foo.h : fichier définissant une fonction `inline` :

```
inline func(int * i)
{
        return( *i++ * *i++ );
}
```

bar.c : fichier de source faisant appel à la fonction :

```
#include "foo.h"
int a()
{
        int * p;
        return( func(p) );
}
```

En C++, les fonctions qui restent définies dans des fichiers `.c` sont des fonctions qui sont trop grosses pour être inlinées ou qui sont volontairement laissées sous forme de fonctions (utilisation de stack).

On a remarqué que le compilateur n'informe pas l'utilisateur s'il considère ne pas savoir inliner une fonction. Dans ce cas, on va avoir plusieurs définitions de la fonction pour l'ensemble du programme applicatif. Vu le conseil donné ci-dessus, avec l'utilisation d'un fichier *header* (fichier `.h`), il suffit de rajouter le qualificatif de *static fichier* pour éviter les redondances :

foo.h : fichier définissant une fonction `inline` :

```
static inline func(int * i)
{
        return( *i++ * *i++ );
}
```

bar.c : fichier de source faisant appel à la fonction :

```
#include "foo.h"
int a()
{
        int * p;
        return( func(p) );
}
```

3.6 Commentaires

Le C++ propose une nouvelle forme de commentaires par rapport à ceux existant en C : le commentaire ligne à ligne.

Les commentaires à la mode du old-C restent toujours utilisables

```
/* commentaire C */
```

Les commentaires multi-ligne du C demandaient une notation du style

```
/* pemiere ligne de commentaire C
/* deuxieme ligne de commentaire C
/* troisieme ligne de commentaire C
*/
```

ou

```
/*
** pemiere ligne de commentaire C
** deuxieme ligne de commentaire C
** troisieme ligne de commentaire C
*/
```

ou encore

```
/* pemiere ligne de commentaire C */
/* deuxieme ligne de commentaire C */
/* troisieme ligne de commentaire C */
```

Le commentaire C++, défini par le double *slash*[45] (`//`) correspond aux formes trouvées en PL/1, Smalltalk ou Lisp. Il travaille à la ligne. Sera pris comme du commentaire tout ce qui se trouve à droite du symbole de commentaire, jusqu'au premier *new-line*[46] rencontré.

```
// commentaire C++
// autre ligne de commentaire C++
int a;  // ce qui suit est en commentaire
```

Les commentaires en fin de ligne d'instruction sont à éviter, car on peut générer des erreurs difficilement visibles. Considérons le code source suivant, utilisant le commentaire dans la définition d'une macro :

```
#define f(a,b) (a + b) // addition de a et b

g()
{
        f(10,3);
        f(10,3) + 5;
}
```

Dans cet exemple, le commentaire est expansé avec la macro lors de l'utilisation de cette dernière. On obtient donc le code final suivant :

```
g()
{
        10 + 3 // addition de a et b;
        10 + 3 // addition de a et b + 5;
```

```
}
```

Le symbole de fin d'instruction est ainsi "inclus" dans le commentaire, et le compilateur rejette l'instruction.

3.7 Cast

La syntaxe du cast (conversion explicite de type) en C est donnée par :

```
(type) expression
```

avec `type` étant un type connu. Le *cast* du C est un opérateur hybride. Il peut être appliqué sur n'importe quel type, défini à la compilation : c'est un opérateur dynamique. C'est un opérateur unaire, placé à gauche de l'opérande sur lequel il porte :

```
int i;
i = (int) 3.0;
```

Cette forme reste valable en C++. Le C++ permet aussi une notation fonctionnelle :

```
type( expression )
```

Ainsi, en C++, on peut *caster*[47] une expression soit par une écriture "opérationnelle" soit par une écriture "fonctionnelle"

```
int i;
i = (int) 3.0;
i = int( 3.0 );
```

3.8 Volatile

Le mot clé `volatile` existe depuis très longtemps en C. Si on prend un compilateur des années 74/78, le mot clé existait, et était reconnu. Son utilisation devient importante aujourd'hui de par le fort taux d'optimisation des compilateurs de la nouvelle génération.

Le mot clé `volatile` est un qualificatif d'allocation. Il demande à ce que l'objet déclaré reste existant dans le scope de déclaration, même en cas d'optimisation possible.

Considérons un code définissant une variable globale entière, et une fonction `f()` qui l'utilise de la façon suivante :

```
int i = 1;

f()
{
        while( i )
        {
                ;
        }
}
```

La variable `i` peut être modifiée ailleurs dans le programme, même par un traitement non séquentiel (si on utilise l'appel système `signal()`, ou si `i` est attaché, via son adresse, à un registre processeur ou à un contrôleur, ou si on utilise la fonction `longjmp()`). Avec un vieux compilateur Ç même en phase d'optimisation, le `i` reste bien alloué. Sur un compilateur "moderne" (par exemple `rcc` sous Unix System V, ou `gcc` de Gnu), dans le code donné plus haut, la boucle disparaît, car considérée inutile. Au vu de l'instruction `i = 1`, et du `while()` utilisé comme une boucle sans fin, le compilateur ote la boucle et le test. il fait une économie d'instructions, mais le programme ne fait plus vraiment ce que l'on voulait !

Une solution serait de ne pas faire passer l'optimisation de code. Cette solution n'est pas satisfaisante, car on veut garder l'optimisation pour le reste de l'application (d'ailleurs certains compilateurs optimisent sans le dire). Pour garder les instructions qui manipulent la variable `i`, on va qualifier celle-ci de `volatile`.

```
volatile int i = 1;
        ...
if( !i )
{
        ...
}
```

Ceci demande à préserver toute utilisation de la variable dans le code. Les instructions qui utilisent cette variable vont rester, même après optimisation.

3.9 Unions anonymes

Les unions sont une structure de donnée introduite tardivement en C.

```
union machine {
        int i;
        long l;
};
```

Les unions anonymes interviennent dans la définition d'union en sous champ de structure.

```
struct foo {
        union machine value;
        int flag;
};
```

Nota Bene : passer une union par valeur en argument de fonction ne sert à rien quand elle est toute seule, car on ne sait pas ce sur quoi elle porte. On a nécessairement une seconde information à donner : quel est le champ de l'union qui est concerné.

Considérons les instructions suivantes :

```
struct foo a;
a.value.l = 10L;
```

On peut, par abus de langage, et par utilisation du concept d'union anonyme, écrire

```
a.l = 10L;
```

Ceci n'a de sens que pour une union champ de structure (ou, nous le verrons plus loin, pour un membre de classe).

Pour les premiers compilateurs Ç si le premier champ de structure était une union, il était possible de faire

```
struct foo {
        union machine value;
        ...
};

struct foo a;
a = 10L;
```

car l'adresse de `a` était utilisée implicitement, pour initialiser le premier champ de la structure. Ceci est maintenant refusé par les compilateurs.

A noter que dans le cas des unions anonymes, pour en utiliser le mécanisme, il ne faut pas que la structure ait des champs "directs" de même nom que ceux des champs de l'union, sinon le

compilateur ne sait pas discriminer de qui il s'agit lorsqu'on mentionne un champ par son nom : le champ direct ou le champ de l'union ?

3.10 Définitions locales diluées

Considérons une définition "classique" d'un bloc C :

```
f()
{
        int a;
        ...
        a = 10;
}
```

En Ç un bloc est défini par deux sections :

- la section déclaration
- la section exécution

la première instruction d'exécution terminant la section de déclaration (*Nota bene* : on peut avoir des initialisations dans la partie déclaration, mais l'initialisation n'est pas considérée comme une instruction d'exécution, car elle intervient lors de la construction de la variable déclarée).

Une grosse modification du compilateur a été apportée pour le C++, avec la possibilité de déclarer les variables quel que soit l'endroit dans le code.

```
f()
{
        int i;
        i = 3;

        char *p;
        p++;
}
```

Les variables "viennent" quand on en a envie. Une passe supplémentaire a été rajoutée au compilateur pour la reconnaissance des variables.

La notion de *scope de variable*[48] reste "classique" :

```
f()
{
        int i;

        {
                // scope local au sous bloc
                int i;
                ..
        }
}
```

La variable n'est connue qu'à partir du moment où elle a été déclarée. Elle est ensuite reconnue jusqu'à la fin du scope dans lequel elle a été déclarée.

```
f()
{
        int i;
        i = 3;

        p++;    // erreur : p non declare

        char *p;
```

```
        p++;    // ok : p a ete declare
}
```

Remarque : une déclaration de variable locale à une fonction ne devrait pas avoir le même nom que celui de l'un des arguments de la fonction, car la variable locale masquera le paramètre.

```
f(int i)
{
        int i;  // Erreur invisible
}
```

Le scope de reconnaissance des variables globales suit la même règle que celui des variables locales. On ne peut utiliser une variable globale que lorsqu'elle a été déclarée.

```
int i;

f()
{
        i++;
        j++;    // Erreur: variable globale non declaree
}

int j;

g()
{
        j++;    // Ok: variable declaree
}
```

Le scope de reconnaissance de la variable globale commence à partir du moment où elle a été déclarée, et non avant (ceci était déjà valable pour les compilateurs C usuels).

3.11 Boucle for

La définition d'une boucle `for` est donnée par

```
for( initialisation ; condition ; pas-suivant )
        bloc
```

illustrée par les instructions suivantes :

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

for( i = 0, j = 3 ; ... ; ... )
        ...
```

En C++, vu la modification apportée pour la déclaration des variables, on a une modification dans la syntaxe de la boucle `for`. La section *initialisation* devient aussi section de *déclaration*.

```
for( int i = 3; i < 10 ; i++ )
{
        ...
}
```

Le scope de la variable `i` est dans la boucle, mais aussi après la boucle.

```
for( int i = 3; i < 10 ; i++ )
{
        ;
}

int i;  // Erreur: redefinition de variable
```

Une évolution serait à apporter pour permettre de limiter la reconnaissance d'une variable

"localement" à la boucle `for`, comme si cette dernière se trouvait construite dans un bloc qui lui serait propre.

3.12 Initialisations

Considérons la séquence d'instructions suivante :

```
f()
{
        int i = 3;
        i++;
}
```

Ceci peut s'écrire, dans un autre style, mais ayant le même résultat :

```
f()
{
        int i;
        i = 3;
        i++;
}
```

En Ç il existe une totale équivalence, garantie, entre les deux écritures. L'opérateur `=` de l'initialisation est traité de la même façon que celui de l'affectation.

`i = 3`	affectation
`int i = 3`	abus de langage, initialisation

Si on considère maintenant le qualificatif `static` appliqué à une variable locale

```
f()
{
        static int i = 3;
        i++;
}
```

L'initialisation n'est faite qu'une fois, lors de la déclaration de la variable. Cela illustre qu'il y a quand même une différence entre une initialisation et une simple affectation.

Alors qu'en Ç la partie exécution de l'affectation reliée à l'initialisation est reportée plus loin, après toutes les déclarations (mais avant la section exécution proprement dite du bloc), le C++ fait une distinction très importante entre l'initialisation et l'affectation. La partie exécution de l'initialisation est réalisée au moment où elle apparaît. Dans l'exemple C++ suivant :

```
f()
{
        int i = 3;
        int j = 6;
        i += 6;
}
```

l'initialisation de la variable `i` est faite avant la création de la variable `j`. Il y a une création de stack pour chaque "variable" déclarée.

3.13 Allocateur dynamique intégré (new/delete)

Soit la fonction `ReadLine()` que l'on se crée pour lire sur l'entrée standard :

```
char * ReadLine()
{
        char buf[512];
```

```
        ...
}
```

Cette définition est mauvaise, car on limite la "taille" de la lecture. Que faire si on veut lire plus de `512` bytes ? Une solution consiste à utiliser une allocation dynamique.

```
char * ReadLine()
{
        char * p;
        p = malloc(BEAUCOUP);

        return p;
}
```

Mais dans ce cas, combien vaut `BEAUCOUP` ? On constate que `malloc()` et `BEAUCOUP` sont deux informations (taille de l'objet et quantité voulue) qui définissent l'objet contenant la ligne saisie sur l'entrée standard. Considérons l'instruction :

```
p = malloc( x * sizeof(type_voulu) )
```

Le buffer est alloué en mémoire centrale, à une adresse qui est stockée dans `p`. On peut vouloir libérer une partie de cet espace. Mais l'instruction

```
free(p + 10);
```

est une garantie pour un plantage assuré !!!

Nota bene : en utilisation des fonctions `malloc()`/`free()`, si on procède à une allocation par petits fragments, on a une perte drastique de place. Le temps de recherche d'un espace libre devient très lent. Même si on passe par une utilisation de *buddies*[49] (pour gens riches en mémoire centrale --BSD--), la consommation des données reste très importante.

Pour résoudre ce problème, le C++ a introduit deux opérateurs, avec deux mots clés correspondants. Il s'agit des opérateurs d'allocation :

`new`	allocation dynamique des entités
`delete`	libération dynamique

3.13.1 Opérateur new

Le compilateur va *type-checker*[50] les `new`. Cet opérateur est utilisé partout dans le code pour créer de nouvelles données.

```
f()
{
        char * p;
        p = new char[127]
}
```

L'utilisation de l'opérateur `new` suit la règle suivante :

```
        new type_existant
```

ou

```
        new type_existant [ quantité ]
```

la seconde forme permettant d'allouer dynamiquement un tableau d'objets de même type. L'opérateur `new` retourne l'adresse du début de la zone allouée.

Quand on exécute le `new`, on ne sait pas exactement ce qu'il fait; par défaut, il utilise la fonction `malloc()`, mais il peut aussi utiliser une fonction définie par l'utilisateur.

Cet opérateur permet d'implanter une création dynamique de tableaux, telle que le proposait Algol :

```
f(int i)
{
        int * t = new int[i];
        t[0] = 3;
}
```

Nota bene : le `new` peut planter. Si `malloc()` plante, comme par défaut c'est la fonction utilisée par `new`, celui-ci va aussi planter. A remarquer qu'on a une routine d'erreur qui peut être rattachée à l'opérateur `new`.

3.13.2 Opérateur delete

L'opérateur `delete` permet la libération d'objets alloués par `new`. Il permet éventuellement de libérer "une certaine quantité" d'objets alloués. Il suit la règle d'utilisation suivante :

```
delete adresse
```

ou

```
delete [ quantite ] adresse
```

Remarquë dans le cas d'une utilisation de la seconde forme, certains allocateurs exigent une quantité pour réaliser effectivement la destruction. Dans notre exemple, on aurait :

```
int * p = new int[127];
delete [127] p;
```

Dans le cas du `malloc()` rattaché à `new`, on n'a pas ce problème.

Si on fait

```
delete [10] (p + 20);
```

cela détruit `10` entités à partir de l'adresse `p + 20`. A noter que le type reste connu pour le `delete`.

Nota bene : ceci peut ne pas marcher si on utilise le `malloc()` de base, par contre si on a un allocateur "fait maison" qui reconnaît ce genre de travail, alors l'opérateur marchera de façon correcte.

Le C++ garantit que l'application multiple de l'opérateur `delete` sur une même variable ne causera pas d'effets désastreux (ce qui n'est pas le cas si on applique plusieurs fois la fonction `free()` sur un récipiendaire de résultat de `malloc()`).

3.13.3 Contrôle sur new

On a vu que l'opérateur `new` permet de faire de l'allocation dynamique. Que se passe-t-il s'il n'y a plus de place mémoire ? Le package de gestion d'allocation dynamique a prévu le cas, et permet d'utiliser une fonction activée au travers du pointeur sur fonction

```
void (* __new_handler)()
```

Sans modification, le pointeur sur fonction n'est pas initialisé, et en cas d'erreur d'allocation (par défaut en utilisant `malloc()`) il ne se passe rien de spécial.

On peut vouloir rattacher notre propre routine de contrôle d'erreur d'allocation (notament lorsqu'on se redéfinit les opérateurs `new` et `delete`). Pour cela, après s'être dotés d'une fonction d'erreur, on la rattache au pointeur `__new_handler()`.

On peut soit le faire directement, par affectation, soit utiliser un outil local offert par le package

d'allocation dynamique :

```
void (* set_new_handler( void (*) () )) ()
```

Donnons-nous par exemple une fonction qui affiche un message sur la sortie d'erreur, et qui arrête le programme. On rattache ensuite cette fonction au gestionnaire d'erreur d'allocation.

```
void ErrorInAllocation()
{
        perror("Local allocation fail0);
        exit(1);
}

typedef void (* PointerToFunctionVoid)();
extern PointerToFunctionVoid set_new_handler( PointerToFunctionVoid );

MyProgram()
{
        // for old value saving
        PointerToFunctionVoid MemoNewHandler;

        // set new value
        MemoNewHandler = set_new_handler(& ErrorInAllocation);
        ...

        // restore old value
        set_new_handler(MemoNewHandler);
}
```

La fonction `set_new_handler()` retourne la valeur précedente du pointeur `_new_handler` (i.e. l'adresse de la fonction de traitement d'erreur précédente).

Voici un exemple illustrant l'utilisation du mécanisme de détection d'erreur d'allocation. Nous nous sommes dotés d'un gestionnaire d'espace quelconque (le but ici n'est pas d'avoir quelque chose de "merveilleux", mais de montrer l'utilisation de `set_new_handler()`), affichant le nombre de bytes alloués lorsque l'allocation a pu se réaliser, et fait un routage sur la fonction de traitement des erreurs en cas d'allocation impossible :

```
#include <new.h>

#define ALOT    100

typedef struct st_Store_space {
        unsigned char Space [ALOT];
        unsigned char * Free;
} Store_space;

static Store_space StoreSpace ;

void InitStoreSpace()
{
        StoreSpace.Free = &StoreSpace.Space[0];
}

void * operator new(unsigned int size)
{
        unsigned char * ReturnValue = (unsigned char *)0;
        unsigned char * End = &StoreSpace.Space[ALOT];

        if( StoreSpace.Free < End  && (End - StoreSpace.Free) >= size )
        {
                printf("Allocating %d bytes0,size);
                ReturnValue = StoreSpace.Free;
```

```
                StoreSpace.Free = StoreSpace.Free + size;
        }
        else
        {
                (* __new_handler)();
        }
        return ReturnValue;
}

void operator delete(void * Space)
{
        if( (unsigned char *)Space < StoreSpace.Free )
        {
                StoreSpace.Free = (unsigned char *)Space;
        }
}

void OutOfStore()
{
        printf("Operator new fail0);
}

void main()
{
        // initialisation de l'allocateur
        InitStoreSpace();

        // Armement du controle sur allocation
        void (* PointerToFunction)();
        PointerToFunction = set_new_handler(& OutOfStore);

        // Allocation de donnees
        char * GoodData = (char *) new char[10];
        char * BadData = (char *) new char[ALOT];
}
```

L'activation de ce petit programme donne le résultat suivant :

```
Allocating 10 bytes
Operator new fail
```

En effet, alors que pour la première allocation d'espace (`GoodData`) on demande une allocation raisonnable, la seconde allocation génère une "sortie" de l'espace d'allocation; on passe alors par le code de traitement des erreurs inscrit dans le code de l'opérateur `new` dont nous nous sommes pourvus.

L'exemple suivant illustre le rattachement d'une fonction locale au gestionnaire d'alloecation dynamique offert en standard par le compilateur :

```
#include <new.h>

void OutOfStore()
{
        printf("Operator new fail0);
        exit(1);
}

#define TOOMUCH 100000000
void main()
{
        // Armement du controle sur allocation
        void (* PointerToFunction)();
        PointerToFunction = set_new_handler(& OutOfStore);
```

```
        // Allocation de donnees
        printf("Data allocation in range0);
        char * GoodData = (char *) new char[10];
        printf("Data allocation out of range0);
        char * BadData = (char *) new char[TOOMUCH];
}
```

L'activation de ce programme donne le résultat suivant :

```
Data allocation in range
Data allocation out of range
Operator new fail
```

Remarque : dans cet exemple, on force la sortie de programme dans la routine de traçage d'erreur d'allocation, car si on ne le fait pas, l'allocateur dynamique boucle à l'infini sur la fonction (en effet, on n'a rien implémenté pour modifier l'environnement de l'allocateur, et partant l'erreur n'est pas corrigée).

3.14 Constantes

Les constantes deviennent fortement qualifiées. Elles sont protégées de toute tentative de modification. Si on regarde un code classique de déclaration de chaîne constante :

```
char * str = "Hello folks";

main()
{
        ...
}
```

A priori, personne n'est censé venir modifier la chaîne de caractères dans les fonctions qui la manipulent. Mais en C (Old-C ou Ansi-C), les constantes ne sont pas spécialement protégées, car elles sont stockées dans le segment des données, au même titre que les variables.

```
char * str = "Hello folks";

main()
{
       /* affichage de la chaine avant modification */
       printf("str avant: %s0,str);

       /* modification de la chaine */
/*     *(str + 6) = 'F'; /* Error: protected area */

       /* affichage de la chaine apres modification */
       printf("str apres: %s0,str);
}
```

Le compilateur gcç lui, considère que ce qui est défini comme constant, et donc une "chaîne de caractères constante", doit être protégé; le programme précédent génère un `bus error` lors de la tentative de modification d'un des caractères de la chaîne. La modification des caractères composant notre "chaîne" ne pourra se faire que si on la transvase, même implicitement, dans un tableau de caractères (via l'allocation et initialisation dudit tableau par la chaîne voulue).

```
char str[] = "Hello folks";

main()
{
   /* affichage de la "chaine" avant modification */
   printf("str avant: %s0,str);

   /* modification de la "chaine" */
```

```
    *(str + 6) = 'F';

    /* affichage de la "chaine" apres modification */
    printf("str apres: %s0,str);
}
```

En C++, la protection explicite d'une constante se fait par l'utilisation du mot clé `const`. Il s'utilise à la déclaration des variables.

```
const char * str_string = "Hello folks";
const char str_array[] = "Hello folks";

main()
{

//      *(str_string + 6) = 'F'; // Error: const string
//      *(str_array + 6) = 'F';          // Error: const chars
}
```

L'utilisation du qualificatif `const` fait en sorte que la "variable" déclarée constante soit stockée dans le segment texte (read only). Toute tentative de modification génère alors une violation de segment.

Cette implantation est utilisée de façon standard en `gcc` pour toute constante (numérique et chaîne de caractères). Ce qui peut poser des problèmes de portabilité entre un programme compilé avec un compilateur "normal" et `gcc`. Un exemple flagrant est donné par l'utilisation de la fonction `mktemp()` qui permet de générer des noms, avec utilisation du *pid*[51] du process.

```
mktemp("fooxxx");
```

Ceci marchera pour un compilateur "normal". `Gcc` protégeant les chaînes de caractères constantes (mises dans le segment text, protégé en écriture), la fontion `mktemp()` ne peut modifier la chaîne fournie en paramètre. Il faut passer à la fonction une chaîne allouée dynamiquement.

```
char * p;
p = malloc(strlen("fooxxx")+1);
strcpy(p,"fooxxx");
mktemp(p);

char t[] = "fooxxx";
mktemp(t);
```

L'utilisation du qualificatif `const` favorise les optimisations de code. Dans le cas de figure suivant,

```
void f(const char * p)
{
        ...
}
```

le compilateur peut choisir de ne pas sauvegarder des registres dans l'appel de fonction.

L'utilisation du qualificatif `const` peut s'appliquer à tout type de variable, mais il faut ensuite savoir décrypter l'instruction de déclaration. La "variable" qualifiée de constante doit être initialisée à sa déclaration.

```
// char variable
char c;

// char constant
const char cc = 'a';

// pointeur variable sur char variable
char * pc;

// pointeur variable sur char constant
```

```
const char * pcc;

// pointeur constant sur char variable
char * const cpc = &c;

// pointeur constant sur char constant
const char * const cpcc = &cc;

// pointeur variable sur pointeur constant sur char variable
char * const * pcpc;

// pointeur variable sur pointeur constant sur char constant
const char * const * pcpcc;

// pointeur constant sur pointeur constant sur char constant
const char * const * const cpcpcc = & cpcc;
```

Ce qui autorise le code suivant :

```
c = cc;
*pcc = cc;
pcc = &cc;
pc = &c;
pc = cpc;
*pc = **pcpc;
```

Mais interdit les instructions suivantes :

```
cc = c;
pcc = &c;
pc = &cc;
pc = pcc;
pc = cpcc;
cpc = pc;
*cpc = *pc;
pc = *pcpc;
**pcpc = *pc;
```

3.15 Références

Reprenons la définition donnée par Algol pour une variable. Il existe une relation ternaire entre

- la valeur d'une variable
- l'adresse de cette variable
- le nom de cette variable

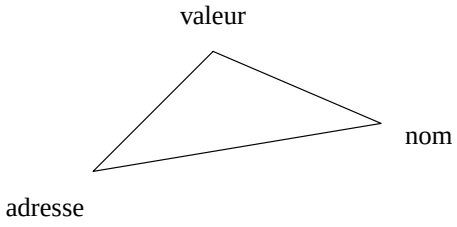


Figure 15. Références - nommage

Si on considère les instructions

```
i = 3;
j = i;
```

Pour la première, la variable `i` est une *left-value*. Le nom est directement associé à l'adresse (récipient). Dans la seconde instruction, la variable `i` est une *right-value*. Le nom est associé à la valeur de la variable.

Les pointeurs permettent d'associer une adresse à un nom. Du fait que l'on sait prendre l'adresse d'une entité, on sait pointer sur l'objet désigné. Mais si on fait "avancer" le pointeur, on réalise une **déréférenciation**.

```
int i = 3;       // entite
int * p; // pointeur
p = &i;          // association pointeur/adresse
p++;             // dereferenciation
```

On change, pour un nom donné (`p` en l'occurence), l'adresse maintenue associée au nom.

Autre exemple :

```
int t[10];
t++;
```

Ceci est tout à fait correct pour le compilateur, mais on a perdu la référence de l'objet `t[0]`.

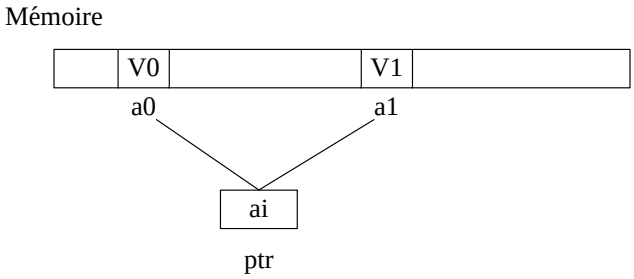


Figure 16. Déréférentiation

On sait changer le "pointage" de `p` de façon dynamique.

On va utiliser les adresses et les pointeurs pour faire des opérations sur des variables données.

```
f()
{
        int i;
        i = 3;

        int * p;
        p = &i;
        *p = 3;
}
```

Les instructions, dans ce contexte, `i = 3` et `*p = 3` ont le même effet. Mais lorsqu'on manipule le pointeur, il n'y a pas transparence du code : quand on manipule le pointeur, on reconnaît la notion de pointeur par l'utilisation explicite des opérateurs d'adressage.

Si on regarde le code suivant :

```
f(int i)
{
        i++;
}

main()
{
        int i = 0;
```

```
        while( i < 10 )
        {
                f(i);
        }
}
```

Les variables `i` des fonctions `main()` et `f()` sont chacunes locales à ces fonctions. Il y a passage par valeur lors de l'appel. En retour de la fonction `f()`, la valeur du `i` du `main()` n'a pas changé. Si on veut que le `main()` ait sa valeur modifiée par la fonction `f()`, il faut lui passer une adresse. Modifions la fonction `f()` pour qu'elle manipule un pointeur.

```
f(int * i)
{
        *i++;
}
```

On a perdu le caractère orthogonal de départ. Si maintenant la fonction `f()` marche par adresse, elle ne marche plus par valeur. Le programme appelant est lui aussi modifié : on doit utiliser explicitement l'opérateur d'adressage pour récupérer l'adresse de la variable `i` et la passer à la fonction.

```
main()
{
        int i = 0;
        while( i < 10 )
        {
                f(&i);
        }
}
```

La **référence** va prendre son sens dans une troisième forme d'écriture : passage des arguments par référence.

Syntaxe de la référence : utilisation de l'opérateur `&`.

`type &` *left-value* `=` *variable connue* `;`

La référence est une *left-value* (car son nom représente sa valeur et aussi son adresse). Elle est initialisée à la déclaration (utilisation de l'opérateur d'initialisation) avec une variable définie auparavant.

```
int j;
int & i = j;
i++;
j++;
```

On dispose alors de deux noms qui "pointent" sur une seule valeur (c'est la définition d'un *alias*); ils sont synonymes d'une même entité.

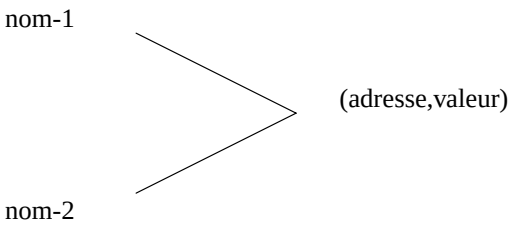


Figure 17. Références - Synonymes

L'instruction `i++` revient à faire `j++`. C'est la même variable qui est incrémentée.

Remarque : l'association des alias ne se fait qu'à la déclaration. On associe des noms, pas des

valeurs. Dans la séquence d'instructions suivante,

```
int j;
int & i = j;
i = 3;
printf("%d",j);
```

l'affichage de la variable `j` donne bien la valeur `3` affectée via l'utilisation de la référence `i` (`i` est une référence sur `j`).

Remarque : il est impossible d'initialiser une référence en dehors d'une déclaration :

```
f()
{
        int i;
        int & j; // reference anonyme
        j = i;   // Erreur
}
```

D'autre part, quand un alias a été réalisé, on ne peut pas le casser :

```
f()
{
        int j;
        int & i = j;

        int k;
        i = k;          // Erreur
}
```

On ne sait pas dé-référencer `j` et attacher le `i` à `k`.

Reprenons l'exemple manipulant un tableau :

```
int t[20];
int & i = t[6];
i++;
```

Dans cet exemple, on a changé la valeur de `t[6]` via la référence `i`. C'est équivalent à avoir fait `t[6]++`. On va pouvoir écrire le code avec une "simple variable" alors que l'on manipule un tableau.

Autre idée : on voudrait utiliser l'écriture de la fonction `f()` vue en premier abord :

```
f(int i)
{
        i++;
}
```

avec l'utilisation du résultat de la forme "par pointeur" dans le `main()` (modification pour le programme appelant de la variable "locale" passée en paramètre). La solution est apportée par les références. La fonction va spécifier que son argument est une référence.

```
void f(int & i)
{
        i++;
}
```

C'est la seule différence avec le premier code montré (passage des paramètres par copie). Au niveau du programme principal, on rappelle le prototype de la fonction utilisée, à savoir une fonction qui reçoit en paramètre une référence sur un entier.

```
void f(int &);

main()
```

```
{
        int i = 0;
        while( i < 10 )
        {
                f(i);
        }
}
```

La référence est donc équivalente, par le nommage, à l'objet désigné. Ce n'est pas une adresse au sens "pointeur". On n'utilise pas spécialement les opérateurs d'adressage sur une référence. Mais par la trilogie évoquée plus haut, c'est aussi l'adresse de l'objet, d'un point de vue localisation exacte du récipient. Dans l'exemple

```
void f(int &);

main()
{
        int t[20];
        f(t[3]);

        int *p = new int[3];
        f( *(p+2) );
}
```

Le premier appel à la fonction `f()` se fait en passant directement le 4-ième élément du tableau. Le second appel fournit en argument ce qui se trouve au bout du pointeur, car la fonction est définie comme recevant une référence sur un entier, et non un pointeur sur entier.

Ceci était prévu dans Algol. Une amnésie de 20 ans a ainsi été dévoilée par Stroustrup !.

On verra que tout ce qui peut se faire sur les classes (nouveau type du C++) se base sur la connaissance des références. A partir du moment où le concept est établi, on peut, en C++, changer l'algorithmique associée à la programmation.

Il n'existe que trois formes d'utilisation qui génèrent une création de variable :

— une déclaration de variable :

```
int & i = j;
```

— un appel de fonction avec paramètre :

```
f(int & i)
```

— un retour de fonction :

```
int & g()
```

Le premier cas de figure est illustré par :

```
{
        int j;
        int & i = j;     // alias

        j++;
        i++;
}
```

Les deux instructions d'incrémentation sont équivalentes. Elles portent sur le même objet.

Le second cas est illustré par :

```
void f(int & i)
{
        i++;
```

```
}

g()
{
        int k;
        f(k);           // passage par reference
}
```

Les deux fonctions travaillent sur le même objet, celui alloué dans la fonction `g()`.

Le troisième cas est illustré par :

```
int & f(int & i)
{
        i++;
        return i;
}
```

Le retour de la fonction est ambigu. On retourne une valeur. Mais `i` est aussi un entier, c'est donc aussi une variable.

Si à l'arrivée dans le code de la fonction `i` vaut `3`, on retourne la valeur `4`. Mais on retourne aussi l'endroit où réside cette valeur (son adresse !). On a donc deux informations qui sont retournées dans le cas d'un retour de fonction défini comme une référence :

— la valeur

— sa localisation

Ce qui permet d'écrire quelque chose d'impossible en C :

```
main()
{
        f() = 3;
}
```

En Ç on se fait jeter par le compilateur, avec un message du type "*left-value required*". En effet, une affectation exige un récipient en opérande gauche. Or une fonction C est une *right-value*. Si la fonction `f()` retourne un pointeur, adresse d'un récipient, alors on doit écrire en C :

```
*(f()) = 3;
```

Le C permet, par utilisation des pointeurs, de manipuler les adresses de récipients, mais l'écriture qui en découle n'est pas simple. L'exemple

```
int * f()
{
        static int i;
        return &i;
}
```

illustre la création d'une variable `i`, par appel de fonction. C'est une variable "privée" de la fonction, que l'on ne peut atteindre autrement que par la fonction (forme d'encapsulage). La fonction en fournit l'adresse en retour d'appel. On peut alors écrire le code du `main()` suivant :

```
main()
{
        int i;

        *f() = 3;
        i = *f();
}
```

La fonction `f()` a une fonctionnalité de *left-value* et de *right-value* si, et seulement si, on utilise la notion de pointeur. On a donc l'encapsulage (protection des accès directs aux données), même en C

"normal". Mais on doit systématiquement utiliser la notion de pointeur. Par contre, on sait écrire

```
*f() = *f();
```

Dans ce cas, la fonction `f()` a une définition non transparente. Cette transparence est permise en C++. On a envie d'écrire :

```
main()
{
        f() = f();
}
```

La fonction `f()` en tant que valeur de retour peut devenir une *left-value* (l'appel de fonction peut devenir un récipient, et non plus rester une simple *right-value*). On déclare la fonction `f()` comme retournant un récipient (retourne une référence sur un objet).

```
int & f()
{
        // declare une variable auto-initialisee
        static int i = 0;

        // retourne cette variable (et sa valeur)
        return i;
}
```

La fonction retourne `i`, mais celui-ci est commué en une référence. C'est le "mot" `f` qui est en fait une référence sur `i`. Dans le programme appelant, on trouve la notation suivante :

```
main()
{
        f() = f();
}
```

On a, pour l'instruction d'affectation,

- `f()` à gauche est une référence sur `i` (*left-value*). On a l'impression de travailler sur l'adresse et de faire l'équivalent du `*f()` dans la manipulation des pointeurs.
- `f()` à droite est la valeur de `i` (*right-value*)

Lorsqu'on manipule une référence, rien ne la distingue de la variable à laquelle elle correspond, ni de la valeur que celle-ci contient.

Axiome : toute utilisation d'une référence est totalement équivalente à l'utilisation d'une variable "normale".

Supposons que l'on ait écrit une fonction compliquée travaillant sur une variable

```
int & f(int & x)
{
        x = 3;
        return( x * 2 + ((&x - 1) << 3)[10] = 5);
}
```

L'utilisation de ce code ne nécessite aucune modification. La variable n'est pas transmise par adresse, ni par valeur, mais par référence.

La fonction qui était définie "par valeur", devient "par référence" par un simple changement du prototype. Tout reste écrit comme pour un travail "en local". Les pointeurs qui restent sont les "vrais" pointeurs (un baladeur sur une adresse différente), et non plus une manipulation sur des adresses pour faire "remonter" les modifications vers le code appelant.

Créons un tableau indicé "*smart*"[52] (les indices sont contrôlés). L'idée est de faire des tableaux encapsulés :

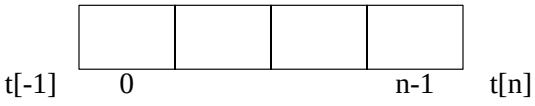


Figure 18. Tableau protégé aux bornes

On fonctionnalise la vue d'un tableau. Créons une fonction `t()` déclarée comme retournant une référence sur entier `int`, et recevant un indice en paramètre.

```
#define Beaucoup 100

int & t(int Index)
{
        static int table[Beaucoup];
        if( Index < 0 || Index > (Beaucoup - 1) )
        {
                Error();
        }
        return( table[Index] );
}
```

L'utilisation de cette fonction devient, par exemple :

```
main()
{
        t(3) = 5;
        if( t(6) == 8 )
        {
                t(6) = t(4);
        }
        t(1000) = 3;     // Error
        t(-1) = 2;       // Error
}
```

Si on utilise la fonction `t()` avec une valeur supérieur à `BCP`, il y a arrêt automatique du programme (si le code de la fonction `Error()` le prévoit).

A remarquer qu'une redéfinition de l'opérateur `[]` permet de revenir à une écriture de manipulation de tableau usuelle.

La transparence reste totale lorsqu'on manipule des objets de type composé tels des structures :

```
struct foo {
        int x;
        ...
};

struct foo & f()
{
        static struct foo a;
        return a;
}

struct foo & g()
{
        static struct foo * a = malloc(sizeof(struct foo));
        return *a;
}

main()
{
        f().x = 10;
        g().x = 33;
```

```
}
```

Remarque : on peut constater que ce genre de fonction ne marche que pour des objets rémanents.

3.16 Exercices

Exercice no. 1 :
Donner les déclarations correspondant aux spécifications suivantes (utilisation de `typedef` recommandée pour les deux dernières définitions) :

1. une fonction prenant en arguments un pointeur sur caractères, une référence sur un entier, et ne retournant rien;
2. un pointeur sur une fonction comme définie en (1);
3. une fonction ne retournant rien, et recevant en paramètre un pointeur comme défini en (2);
4. une fonction retournant un pointeur comme défini en (2)
5. une fonction recevant en paramètre un pointeur comme défini en (2) et retournant ce pointeur.

Exercice no. 2 :
Céer une fonction permettant s'afficher une chaîne de caractères; la tester en l'appelant avec la chaîne "Hello World", puis en l'appelant avec l'entier 10.

Exercice no. 3 :
Créer une fonction permettant d'afficher un entier et une chaîne de caractères; la tester en l'appelant en lui fournissant en arguments l'entier 10 et la chaîne "Hello World", puis en ne lui fournissant qu'un entier, puis qu'une chaîne de caractères.

Exercice no. 4 :
Créer une fonction `Print()` à plusieurs arguments et la mettre dans un fichier. Générer une fonction `main()` dans un autre fichier, en déclarant la fonction `Print()`. Utiliser la compilation séparée pour obtenir l'exécutable final.

Exercice no. 5 :
Créer une fonction `Print()` avec overload de code. Les différents codes permettent d'afficher un entier, un chaîne de caractères, et un entier et une chaîne de caractères.

Exercice no. 6 :
Traduire le programme C suivant en un programme C++ :

```
char * Alloc(x)
int x;
{
        return malloc(x);
}

int Free(x)
char * x;
{
        free(x);
}

main()
```

```
{
        char * ptr;

        ptr = Alloc(10);
        Free(ptr);
}
```

Exercice no. 7 :
Ecrire un programme définissant trois fonctions dont le but est d'ajouter la valeur `1` à l'entier fourni. La première utilise un passage de paramètre par copie, la seconde par adresse, et une troisième par référence. Vérifier les résultats d'utilisation de ces fonctions par un petit programme principal.

Exercice no. 8 :
Quelles sont les spécifications fonctionnelles d'une gestion de stack ? Donner une définition C+ des fonctions de traitement de stack.

 ISBN 2-909909-02-6

4. C++ : Langage orienté objet

Nous allons voir maintenant ce qui est du spécifique C++ pris en tant que langage orienté objet.

- les **classes** (encapsulage) : un nouvel agrégat, de style structure, permettant de définir des données avec du code associé

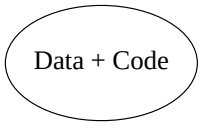


Figure 19. Classe : encapsulage

Les classes sont des définitions de types. Elles sont immatérielles, et font intervenir le concept d'**instanciation**, avec une notion de **collectif**. Une classe apparaît comme un groupe composé d'individus identiques.

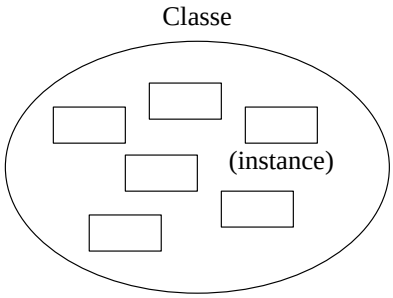
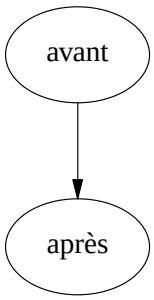


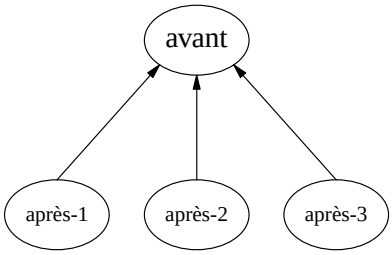
Figure 20. Classe et instance de classe

- l'**héritage** : fait intervenir une notion de parenté entre des instances de une à plusieurs classes. Deux formes d'héritage sont utilisées en C++ :
  - — *héritage descendant* (**héritage**) :



"après" intègre les capacités de "avant".

  - — *héritage ascendant* (**polymorphisme**) : fait intervenir le concept de classe *générique*.



"avant" utilise les capacités de ses "enfants" (après-1, après-2, après-3). Il y a mutation dynamique de "avant" dans l'un des "après".

Ces deux héritages se font avec le même outil.

4.1 Class

Le sens général du terme *objet* est défini par le fait

- qu'une notion d'encapsulage lui soit appliquée,
- qu'il intègre le mécanisme d'héritage,
- qu'il intègre le mécanisme de polymorphisme.

Pour définir un objet répondant à ces trois caractéristiques, le C++ définit un seul mot clé : `class`.

Le mot clé `class` s'utilise d'une façon similaire au mot clé `struct` du C.

```
struct foo {
        int x;
};

class bar {
        int x;
};
```

Les déclarations d'objets C++ se font plus simplement. Le nom associé à la définition d'une classe devient synonyme du type nouvellement défini. L'utilisation du `typedef` du Ç pour avoir des noms de type plus simples, devient superfétatoire.

```
/* C */
struct st_foo {
        int x;
};
typedef struct st_foo foo;
foo f;

// C++
class bar {
        int x;
};
bar b;
```

Lorsqu'on fait une déclaration d'un objet de type "classe", on parle d'instanciation de la classe correspondante, ou encore d'instance de classe.

Alors qu'en Ç on parle de **champs de structure**, on parlera de **membre de classe**. Les classes intègrent des membres données et des membres fonctions.

```
class Foo {
```

```
        int data;
        int fonction();
};
```

Un objet C++ est donc un individu ayant des données et du code associé. Le code est propre à la classe.

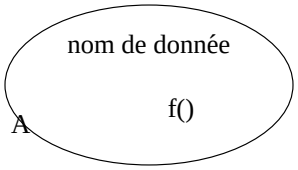


Figure 21. Un objet C++

L'accès aux membres d'une classe se fait (sauf protection) comme pour accéder aux champs d'une structure :

```
class Foo {
        int data;
        int fonction();
};

Foo a;
a.data = 10;
a.fonction();
```

Les informations ne sont accessibles qu'au travers d'une instance de classe (utilisable uniquement avec un objet effectivement présent). Le rôle de l'encapsulage, concernant la réelle visibilité des informations rattachées à une classe, sera vu plus loin.

Les fonctions membres d'une classe sont associées à cette classe. Leur code est partagé par toutes les instances de cette classe. Mais leurs données sont propres à chacune des instances.

4.2 Opérateur de domaine

Le C++ intègre un nouvel opérateur : l'opérateur de domaine, noté `::`. Cet opérateur suit la règle d'utilisation suivante :

```
        <type> :: <membre>
```

Il permet d'identifier totalement l'entité manipulée. Prenons par exemple une classe définie par un membre donnée et un membre fonction. On peut donner une définition de la fonction membre rattachée à la classe en utilisant l'opérateur de domaine :

```
class NomDeType {
        int MembreDonnee;
        int MembreFonction();
};

int NomDeType :: MembreFonction()
{
        int i = 10;
        int k = i + 100;
}
```

Lorsqu'on se trouve sur le code de définition de la fonction `MembreFonction()`, on spécifie que c'est la fonction rattachée à la classe `NomDeType`. On peut avoir plusieurs classes ayant des membres qui s'appellent de la même façon localement; ils seront totalement différenciés par leur

appartenance à une classe ou à une autre.

```
class Foo {
        int a;
        int func();
};

class Bar {
        int a;
        int func();
};

int Foö:func()
{
        // definition de la fonction attachee a la class Foo
}

int Bar::func()
{
        // definition de la fonction attachee a la class Bar
}
```

Le C++ permet de donner une définition de fonction membre de classe directement dans le code de définition de cette fonction.

```
class NomDeType {
        int MembreDonnee;
        int FonctionMembre () {
                int i = 10;
                int k = i + 100;
        };
};
```

On a donc le choix entre

- donner la définition de la fonction dans la définition de l'objet,
- donner une définition externe, en utilisant l'opérateur de domaine `::`.

Dans le cas d'une définition intégrée directement dans la définition de la classe, la fonction sera considérée `inline` (dans la mesure du possible --pas de boucle `for`--).

4.3 Encapsulage - Visibilité des membres de classe

Les classes sont définies dans un *scope* qui leur est propre (le domaine des classes), distinct de celui du programme. Il existe une frontière stricte et étanche entre ces deux scopes.

 ISBN 2-909909-02-6

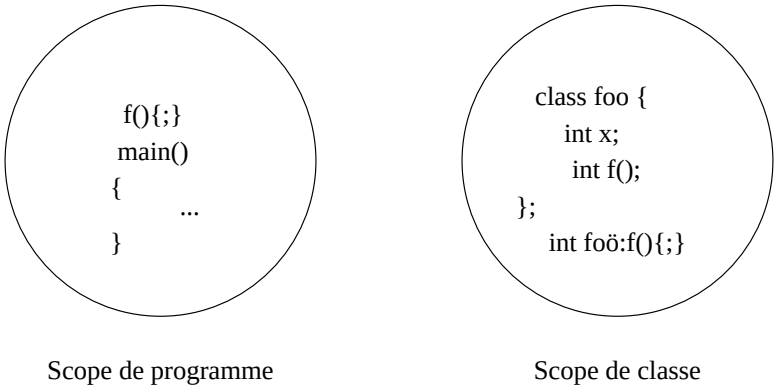


Figure 22. Scope programme vs scope classe

A priori, ces deux domaines cohabitent, sans se mélanger. Ils sont totalement étanches, sauf spécification explicite.

Les données globales (variables et fonctions) du domaine du programme, qui sont visibles à tous dans ce scope, pourront être "vues", suivant directive spécifique, par les individus du domaine des classes. On peut dire que les informations "globales" du scope du programme se trouvent sur la périphérie du scope. On retrouvera sur cette périphérie de domaine, toutes les informations dites "publiques" (`public access`[53]).

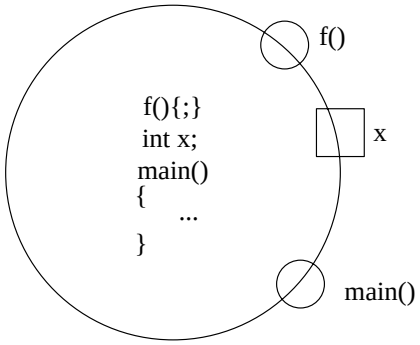


Figure 23. Scope programme

Lorsqu'une fonction du domaine des classes veut utiliser une fonction ou une variable globale du domaine du programme, elle spécifiera "*la fonction qui n'est rattachée à aucune classe*", en utilisant l'opérateur de domaine sans spécification de `type`.

```
// domaine programme
int i;

int f()
{
        ...
}

// domaine classe
class foo {
        int func();
```

```
};

int foö:func()
{
        ::i = 99;       // variable du scope programme
        ::f();  // appel a fonction du scope programme
}
```

Si la fonction membre `func()` de la classe `foo` ne spécifie pas que la fonction `f()` utilisée est hors scope (pas de type rattaché à la fonction), cela présuppose que cette fonction est aussi une fonction membre de la classe.

```
// domaine programme
int f()
{
        ...
}

// domaine classe
class foo {
        int func();
        int f();
};

int foö:func()
{
        ::f();  // appel a fonction du scope programme
        f();    // appel a fonction membre locale a scope classe
}
```

Lorsqu'une fonction du scope du programme déclare une instance de classe, il se donne un point d'accès vers le scope de cette classe. Mais cela ne suffit pas pour accéder aux membres de cette classe. Il faut encore que cette dernière ait spécifié que les informations étaient accessibles par un "étranger". Pour cela, lorsqu'on donne la définition d'une classe, on va utiliser le mot clé `public` permettant de définir une section "visible de l'extérieur" (mise en périphérie du scope de la classe).

```
class foo {
public :
        int data;
        int func();
};
```

Toute définition qui se trouve à la suite de ce qualificatif sera alors considérée comme étant sur la périphérie du domaine de la classe.

 ISBN 2-909909-02-6

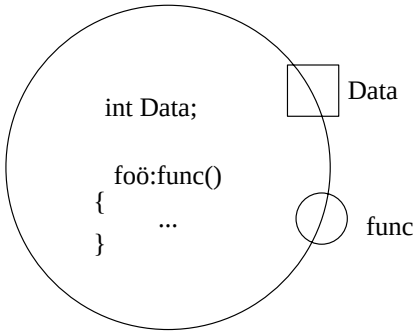


Figure 24. Scope de classe

L'entité ainsi définie sera accessible à partir du domaine "hors classe", donç dans notre exemple, à partir de l'instance de classe déclarée dans le scope du programme :

```
// scope de classe
class foo {
public :// mise en peripherie
        int data;
        void func();
};

void foo::func()
{
        data++;
}

// scope programme
void f()
{
        foo x;  // instance de classe

        x.func();
        x.data = 33;
}
```

Par défaut, quand on ne spécifie aucun qualificatif de visibilité, les informations contenues dans une classe sont protégées, non accessibles hors du scope de la classe (invisibles).

```
// scope de classe
class foo {
        int donnee;
public :// mise en peripherie
        int data;
        void func();
};

void foo::func()
{
        data++;
        donnee++;
}

// scope programme
void f()
{
        foo x;          // instance de classe
```

```
        // x.donnee = 999;       // Erreur : invisible
        x.func();                // Ok : public
        x.data = 33;     // Ok : public
}
```

On peut spécifier soi-même qu'une section de la classe est totalement inaccessible. Pour cela, on utilise le mot clé `private`. Toute information contenue entre une section déclarée `private` et un autre qualificatif de section sera alors totalement protégée.

```
// scope de classe
class foo {
private :       // protection totale
        int donnee;
        void f();
public :// mise en peripherie
        int data;
        void func();
};

void foo::func()
{
        donnee++;
        data++;
}

// scope programme
int f()
{
        foo x;          // instance de classe

        // x.donnee = 999;       // Erreur
        // x.f();                // Erreur

        x.func();                // Ok
        x.data = 33;     // Ok
}
```

Les définitions de section n'ont pas d'ordre précis. On peut avoir plusieurs sections `private` ou `public` dans une classe. C'est pour une question de facilité de lecture que l'on conseille de les regrouper entre elles.

```
// scope de classe
class foo {
private :       // protection totale
        int donnee;
        void f();
public :// mise en peripherie
        int data;
        int func();
private :       // protection totale
        int autre_donnee;
public :// mise en peripherie
        void g();
        long autre_data;
};
```

Pour utiliser les fonctionnalités d'une classe, il faut au moins un point d'entrée permettant au code du scope du programme de pouvoir "entrer" dans le scope de la classe. A partir du moment où on a pu entrer dans le scope de la classe, tout ce qui se trouve défini dedans est accessible, comme si on se trouvait dans un "simple programme" (mais local au scope). Ainsi, à partir du moment où on est entré dans le scope de la classe, les données et les fonctions définies dans le scope de cette classe, qu'elles soient qualifiées de `private` ou de `public`, sont visibles comme les données globales le sont pour

un programme classique. Dans l'exemple suivant,

```
// scope de la classe
class foo {
private :
        int i;
public :
        void g() {
                i = 3;
        };
};

// scope programme
void f()
{
        foo x;
        // x.i = 9;      // Erreur
        x.g();  // Ok
}
```

la donnée `i`, qualifiée de privée à la classe, n'est pas accessible par le programme. Par contre, dès que l'on entre dans le scope de la classe, par appel de la fonction `g()` attachée à la classe, cette dernière peut voir tous les composants de la classe. La donnée `i` est totalement protégée. L'accès à ce récipient ne peut se faire qu'au travers d'une fonction mise en libre accès de la classe. Pour de telles fonctions, on parlera de *service*.

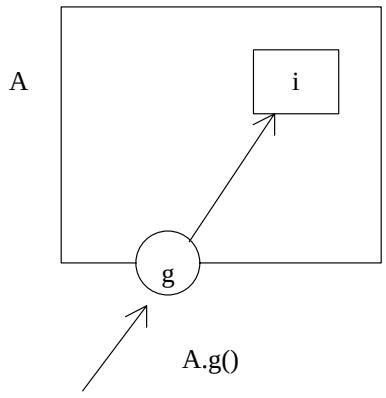


Figure 25. Services publics

Lorsqu'on est dans le scope de la classe, les données et fonctions définies comme membres de la classes sont visibles comme le sont les "globaux" pour un programme.

```
class foo {
private :
        int i;
        int f();// local classe
public :
        int j;
        int g() { i++; f(); };
};
```

Dans cet exemple, la fonction `g()` est accessible par le programme. Le code de la fonction fait appel à la fonction `f()`, fonction membre privée de la classe (non accessible par le programme).

Les règles de masquage, valables au niveau d'un programme, sont aussi appliquées, localement, au scope de la classe.

```
class foo {
        int i;  // membre prive
public :
        void g() { i = 3; };
        void f() { int i; i = 99; };
        void h(int i) { i = 88; };
        void p() { printf("i=%d0,i); };
};

main()
{
        foo x;

        x.g(); x.p();
        x.f(); x.p();
        x.h(10); x.p();
}
```

Si on active le programme donné ci-dessus, on constate que seule la fonction membre `g()` a modifié le membre donnée `i`. Pour la fonction membre `f()`, la variable locale de la fonction masque le membre donnée; de même pour la fonction membre `h()`, c'est le paramètre qui masque le membre donnée.

Si on veut qu'une fonction membre, qui aurait involontairement masqué un membre donnée, puisse accéder à ce membre donnée, il lui suffit de spécifier que c'est la variable "globale" de la classe. On utilise l'opérateur de domaine, appliqué à la classe.

```
class foo {
        int i;
public :
        void h(int i) { foo::i = i; };
        void p() { printf("i=%d0,i); };
};

main()
{
        foo x;

        x.h(10);
        x.p();
}
```

Dans cet exemple, la fonction `h()` copie la valeur de son paramètre dénommé `i` dans le membre donnée de la classe lui aussi dénommé `i`.

Si, dans notre exemple, on veut modifier une variable globale programme avec la valeur d'une variable de même nom du scope de la classe, on utilise aussi l'opérateur de domaine, permettant ainsi de localiser pleinement les variables manipulées.

```
// scope du programme
int i;

// scope de la classe
class foo {
        int i;
public :
        void SetGlob() { ::i = i; };
        void OtherSetGlob { ::i = foö:i; };
};
```

Considérons deux instances d'une même classe. Chacune a ses proprs données (définies dans le type classe). Elles sont dotées d'un vecteur composé des fonctions rattachées à la classe (il n'y a pas duplication de code).

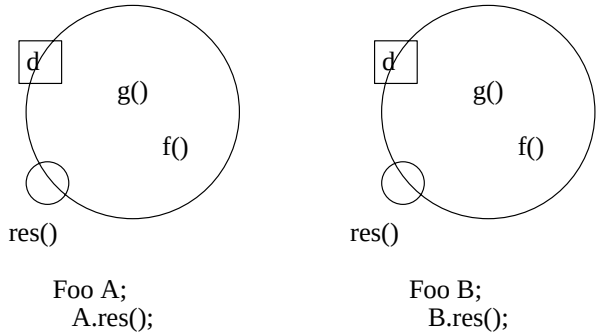


Figure 26. Deux instances de classe

Le domaine reconnu par les fonctions est celui de l'instance : les données "globales" du scope sont celles définies pour l'instance manipulée.

```
// definition de classe
class foo {
        int i;
public :
        void h(int i) { foö:i = i; };
};

main()
{
        foo x;  // instance de classe
        foo y;  // instance de classe

        x.h(10);
        y.h(33);
}
```

Dans cet exemple, l'activation de la fonction `h()`, définie pour la classe `foo` au travers de l'instance `x`, modifie le membre `i` de cette instance `x`. L'activation faite au travers de l'instance `y` modifie le membre `i` de l'instance `y`.

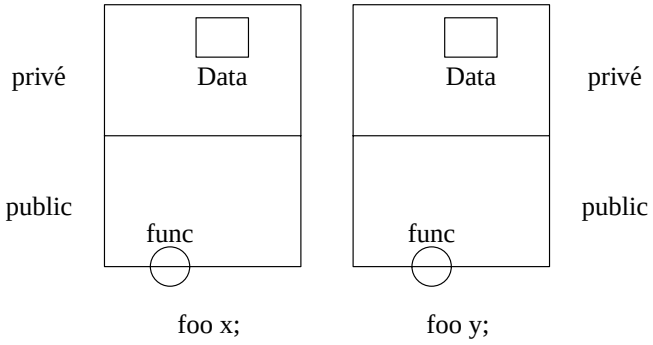


Figure 27. Deux instances de *class foo*

Une fonction membre peut, comme n'importe quelle fonction, recevoir en paramètre une référence sur une classe. Dans ce cas, elle a accès aux données définies pour l'objet référencé. S'il s'agit d'une classe de même type, alors elle a aussi accès aux données privées définies dans la classe.

```
class Foo {
        int nd;
        void f();
```

```
public :
        void g();
        void h(Foo &);
};

void Foö:h(Foo & a)
{
        a.nd++;
        a.f();
        a.g();
}
```

Lorsqu'on est dans la fonction membre de la classe, on peut accéder aux informations de la référence, qu'elles soient privées ou publiques (car la référence porte sur le même type que celui dont la fonction est membre). Ceci est illustré par l'exemple suivant :

```
class foo {
        int i;
public :
        void p() { printf("i=%d0,i); };
        void f( foo & x) { x.i = 888; x.p(); };
};

main()
{
        foo x;
        foo y;

        x.f(y);
}
```

Voici, en exemple d'utilisation des classes, plusieurs implémentations d'un "compteur" illustrant l'encapsulage des données. Un compteur est déterminé par une valeur que l'on peut

— initialiser,

— incrémenter,

— et consulter.

Première implémentation :

```
class compteur {
        int i;
public :
        int get() { return i; };
        void init() {  i = 0; };
        void incr();
};

void compteur::incr()
{
        i++;
}

main()
{
        compteur C;
        int tab[100];

        for( C.init(); C.get() < 100 ; C.incr() )
        {
                tab[C.get()];
        }
}
```

 ISBN 2-909909-02-6

L'intérêt de ce genre d'implémentation, est qu'il est facile d'inclure la limite du compteur dans le code de la fonction d'incrémentation. Dans ce cas, l'instruction conditionnelle de la boucle devient superfétatoire.

```
class compteur {
        int i;
public :
        int get() { return i; };
        void init() {  i = 0; };
        void incr();
};

void compteur::incr()
{
        if( i < 100 )
        {
                i++;
        }
        else
        {
                perror("limite");
                exit(1);
        }
}

main()
{
        compteur C;
        int tab[100];

        for( C.init(); ; C.incr() )
        {
                tab[C.get()] = 0;
        }
}
```

Seconde implémentation :

On peut aussi fournir une implémentation avec paramétrage de la routine d'incrémentation :

```
class compteur {
        int i;
public :
        int get() { return i; };
        void init() {  i = 0; };
        void incr(int);
};

void compteur::incr(int x)
{
        if( i < x )
        {
                i++;
        }
        else
        {
                perror("limite");
        }
}

main()
{
        compteur C;
        int tab[100];
```

```
        for( C.init(); ; C.incr(100) )
        {
                tab[C.get()] = 0;
        }
}
```

Troisième implémentation :

On peut implémenter la borne au moment de l'initialisation du compteur. Dans ce cas, c'est la fonction membre d'`init()` qui est à modifier :

```
class compteur {
        int i;
        int limite;
public :
        void init(int);
        void incr();
        int get() { return i; };
};

void compteur::init(int x)
{
        i = 0;
        limite = x;
}

void compteur::incr()
{
        if( i < limite )
        {
                i++;
        }
        else
        {
                perror("Limite");
        }
}

main()
{
        compteur C;
        int tab[100];

        for( C.init(100); ; C.incr() )
        {
                tab[C.get()] = 0;
        }
}
```

4.4 Les structures C++

Les structures C++ sont des classes pour lesquelles tout est défini dans une seule section `public`.

```
// style C
struct foo {
        int x;
        long y;
};

// C++
class foo {
public :
        int x;
```

```
        long y;
};
```

Tout comme pour les classes, on pourra associer du code à la structure, et avoir des champs fonctions. Mais ceci n'est plus du tout compatible avec une structure compilée en langage C. La compilation d'une structure en langage C et en langage C++ ne donne pas le même résultat d'un point de vue implémentation interne; on ne peut donc pas mélanger les deux codes.

4.5 Exercices

Exercice no. 9 :
Réaliser une classe de statistiques entières qui maintienne le maximum, le minimum, et la moyenne de l'ensemble des valeurs qui lui sont présentées. Cette classe doit répondre au programme suivant :

```
{
        Statistic s;

        for( int i = 0; i < 1000; i++ )
        {
                s.Set(rand());
        }
        printf("%d0,s.Average);
        printf("%d0,s.Minimum);
        printf("%d0,s.Maximum);
}
```

Exercice no. 10 :
Modifier la classe de l'exercice précédent, en renforçant les protections des données, et faire en sorte qu'elle réponde aux fonctionalités suivantes :

```
{
        Statistic s;

        for( int i = 0; i < 1000; i++ )
        {
                s.Set(rand());
        }
        printf("%d0,s.GetAverage());
        printf("%d0,s.GetMinimum());
        printf("%d0,s.GetMaximum());
}
```

5. Méthodologie de développement

Avant d'aborder l'utilisation du C++ en terme d'héritage et de polymorphisme, on va aborder la méthodologie à utiliser, différente de celle appliquée habituellement en C. C'est une méthodologie d'approche nouvelle, pour laquelle il faut perdre les réflexes usuels.

Une bonne formation au C aurait dû introduire dans les esprits une notion de modules (décisions de codage de base), avec le concept de "librairie de fonctions associées à un type particulier". Si c'est le cas, l'encapsulage ne devrait pas poser de problèmes de conception.

On va voir ici la conception d'objet, sans aborder les caractéristiques d'héritage. Il s'agit du *design* à appliquer dans la construction de solutions.

Il existe des méthodologies de conception (des méthodes de programmation, et des méthodes de conception de programmes). Des méthodes de programmation objet ont été mises en place, abandonnant les concepts procéduraux. Cela demande une analyse préliminaire des problèmes (cfr. Jackson, Merise version objet, ...). Les méthodes "classiques" ne fournissent pas de conception objet.

Une bonne introduction à la conception objet est présentée dans le livre *Object-Oriented Software Construction* de Bertrand Meyer [Meyer,1988] . D'autres méthodes sont présentées par Ian Graham dans son livre *Object Oriented Methods* [Graham,1991] . Influencé par la programmation Ada [Booch,1987b] G. Booch présente une autre méthode de conception objet dans son livre *Object Oriented Design with Applications* [Booch,1991] .

En ce qui concerne notre propos, on va se donner quelques éléments de base de conception objet : protection des données, réalisation de *packages*. Ceux qui programment "proprement" en C trouveront ce chapitre trivial.

5.1 Conception

5.1.1 L'encapsulage

L'encapsulage permet d'isoler les données dans un programme. Les "globales externes" sont prohibées. On bannit totalement les externes des programmes. En effet, à quoi servent les variables externes ? Elles servent :

- de *public access* à des fonctions,
- de faux paramètres entre des fonctions.

L'idée de l'encapsulage est que tout est isolé. Aucune donnée n'est accessible à aucun autre que celui qui s'en occupe. Il est, par nature, impossible d'accéder aux données autrement que par la personne chargée du traitement de cette donnée.

Ceci est implémenté par la section `private`/`public`, avec matérialisation du "service". Là où on a `b->a++` en Ç on transforme cela par une classe :

```
class foo {
        int a;
public :
        add() { a++; };
};

foo b;
b.add();
```

Cela force à matérialiser ce que l'on veut faire comme action sur la donnée. On type l'opération C. On dit explicitement ce que l'on fait.

Les services définissent exactement ce que l'on peut faire avec la classe. La partie donnée interne à la classe n'est pas la partie importante. C'est la partie spécification fonctionnelle externe qui détermine pleinement la classe.

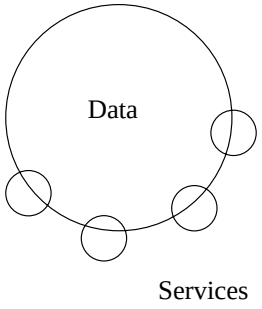


Figure 28. Données et services

5.1.2 La spécification fonctionnelle externe

C'est la partie la plus importante de la conception objet :

- pour faire une classe, on ne s'occupe pas des données. On évite de limiter le programme dans des structures de données figées;
- le programme est défini par les services.

Voici un mauvais exemple : on décide, dans un programme, d'utiliser un compteur. La mauvaise réflexion est de se dire : "j'ai besoin d'un entier, défini par un `int`".

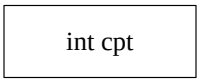


Figure 29. Un entier

et de ne penser qu'ensuite aux services attendus :

- incrémenter
- décrémenter
- remise à zéro (reset)

Ceci n'est pas bon, car l'objet compteur, défini par l'instruction `int cpt`, est figé.

Il faut raisonner à l'envers, et se dire : "un compteur, c'est un outil qui permet d'incrémenter, décrémenter, et resetter". On va se définir quelque chose dans le programme par un ensemble de services rendus. On peut dire qu'un compteur, c'est un individu qui sait incrémenter sa valeur de une unité, qui sait la décrémenter, et la remettre à zéro. On peut donc décrire un compteur comme suit :

```
compteur = { increment, decrement, reset }
```

Ce sont les services qui déterminent le "quelque chose" dont on vient de se doter. Si on formalise un objet compteur de cette façon, on est tout à fait opérationnel.

On définit une classe, sans se poser la question sur la "mémorisation" des données. Par contre, le

travail consiste à définir les prototypes des services.

```
class compteur {
public :
        void Increment();
        void Decrement();
        void Reset();
};
```

Si on prend, comme autre exemple, le concept "je dois faire une liste chaînée". C'est un mauvais départ de réflexion. "Je dois faire une liste" est quelque chose de plus important.

Il faut partir de la liste des services. Avec ça, tout le monde peut travailler ensemble (capacité de faire des développements énormes, avec beaucoup de gens, sans que ces derniers ne se rencontrent).

Après ça, on peut changer de "personnage" et devenir l'*implémenteur* de la classe. On peut faire alors plusieurs implantations, sans perturber le travail des autres. On a défini un objet dont l'écorce est totalement définie.

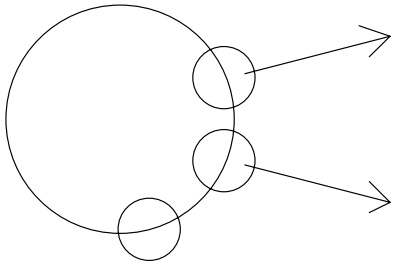


Figure 30. Ecorce (spécification externe)

Maintenant, on peut penser à l'implémentation.

5.1.3 Implémentation

On définit la matérialisation interne des services. A tout moment on peut modifier cette partie, sans changer la spécification externe (écorce).

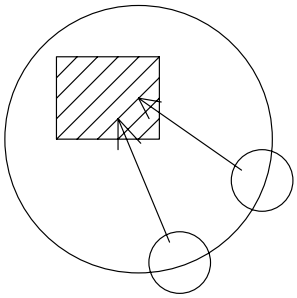


Figure 31. Implémentation (partie interne)

Implémenter, c'est

- définir les données,
- implémenter un sous-interface d'implantation,

 ISBN 2-909909-02-6

- implémenter les données nécessaires.

En Ç une "liste de compteurs" se traduit classiquement (en mauvaise programmation) par :

```
struct foo {
        struct foo * nxt;       /* suivant */
        int cpt;        /* donnee */
};
```

En C++, on ne peut en aucun cas imaginer qu'un compteur inclue une notion de "suivant". En effet, le fait d'avoir un suivant ne fait pas partie des caractèristiques spécifiques d'un compteur (un compteur ne sait que compter, comme son nom l'indique !).

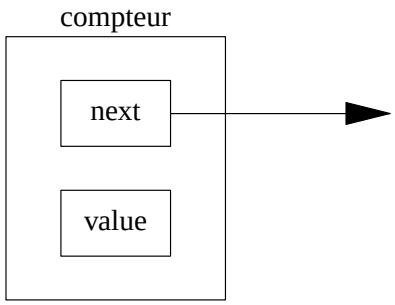


Figure 32. Compteur avec suivant (mauvais)

Regardons l'implantation d'un compteur, au vu de la première définition de classe :

```
class compteur {
        int i;  // implantation
public :
        void Increment() {      // conception
                i++;    // implantation
        };
        void Reset() {
                i = 0;
        };
};

main()
{
        compteur C;

        C.Reset();
        C.Increment();
}
```

A noter qu'avec la première définition, on pouvait déjà écrire un programme principal.

Transformons l'implémentation en disant que le compteur se trouve en fait stocké dans un fichier.

```
class compteur {
        int filedes;
public :
        void Increment();
};

void compteur::Increment()
{
        int i;

        read(filedes, &i, sizeof(i));
        lseek(filedes, 0L, 0);
```

```
        i++;
        write(filedes, &i, sizeof(i));
}
```

Le programme principal n'a pas changé. On fait un *hide*[54] de l'implémentation (on la cache). On ne fait plus :

```
main()
{
        i++;
}
```

On découpe donc totalement le travail. On applique une notion de modularité. Il faut s'exercer à coder séparément :

- les spécifications fonctionnelles,
- l'implémentation.

Il faut typer fortement les données que l'on manipule. Si on utilise un entier comme un flag, on peut le définir par

```
flag = { lever, baisser }
```

Ce qui est totalement différent de la notion de compteur telle qu'on l'a vue précédemment, même si, en Ç ces définitions se ressemblent.

5.2 Modularité de la programmation objet

Ceci fait intervenir deux nouveaux concepts :

- la généralisation,
- le *well-known service*.

L'idée est de ne jamais aborder un problème de manière complexe. Cela revient à dire : "si on a un gros problème, on le décompose en entités étanches (encapsulage)". La programmation finale est une mise en présence de modules qui vont opérer entre eux.

```
Module A
Module B
Module C

X = { relations entre A, B, C }
```

On va appliquer la technique de recouvrement.

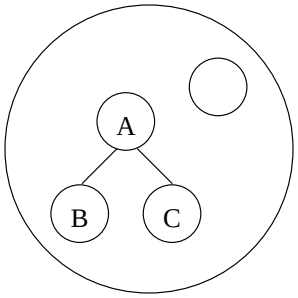


Figure 33. Découpage et regroupement

La programmation devient de la stratégie, et non plus de la tactique. On va "brasser" les services sans

 ISBN 2-909909-02-6

s’occuper de la cuisine d’implémentation.

Par exemple, si on doit réaliser un système d’exploitation, on découpe ce qu’il faut faire en différents objets :

— fichiers,

— mémoire centrale,

— gestion des processus,

— communication inter-processus,

— disques,

— ...

On fait une classification d’un point de vue fonctionnel, et non pas d’un point de vue données d’implémentation. Il s’agit de voir les interactions existantes dans le système.

Chaque individu ainsi déterminé est indépendant des autres, mis dans un ban de test qui lui est propre. L’objet doit être validé en tant qu’outil de production. Les objets sont mis ensuite en librairies d’objets, et ce n’est qu’à la fin que l’on "monte progressivement la mayonnaise" pour réaliser le système d’exploitation (intégration par technique d’assemblage d’objets validés, aveç si nécessaire, création de nouveaux objets pour recoller les morceaux).

Un autre point fort de la pogrammation objet est de faire de la programmation sans *bug*[55] (sans bug majeur). On valide l’objet en tant que tel.

Les seuls problèmes viennent ensuite de la cohabitation possible ou non entre les objets. Certains objets peuvent être incompatibles entre eux. Mais il s’agit là de problèmes de relations inter-objets. *Remarque* : il ne faut pas négliger cet aspect des choses, car certaines configurations deviennent impossibles du fait de la cohabitation des fonctionnalités externes des objets en présence. De telles situations peuvent être critiques.

Prenons comme exemple la réalisation d’une "batterie de compteurs". Il faut commencer par formaliser le problème, en créant éventuellement des mots, exprimant ce que l’on veut réaliser. La "batterie de compteurs" est quelque chose de "creux".

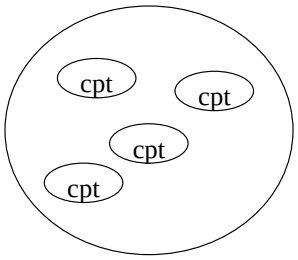


Figure 34. Ensemble de compteurs

C’est un "*bag*" au sens Smalltalk. Il n’y a aucune notion d’ordonnancement, ni de ce que sont les compteurs eux-mêmes.

La mauvaise approche serait de dire

- j’ai besoin d’une batterie,
- j’ai besoin de plusieurs compteurs,
- et il faut les coder ensemble.

L’orientation objet nous amène à penser suivant le schéma :

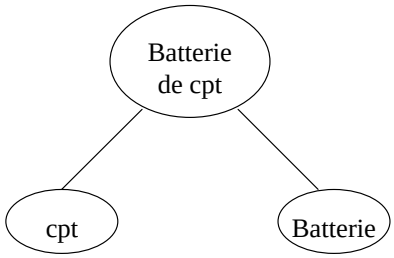


Figure 35. Batterie de compteurs

On postule l’existence des objets qui seront nécessaires. On réitère cette démarche jusqu’à arriver à la conception de classes très simples.

On travaille avec des concepts de plus en plus **génériques**. "Batterie de compteurs" est un cas beaucoup plus particulier que "batterie".

Cette conception pourra amener à utiliser des librairies d’objets simples, puis d’objets plus complexes. On cherche à faire des choses réutilisables.

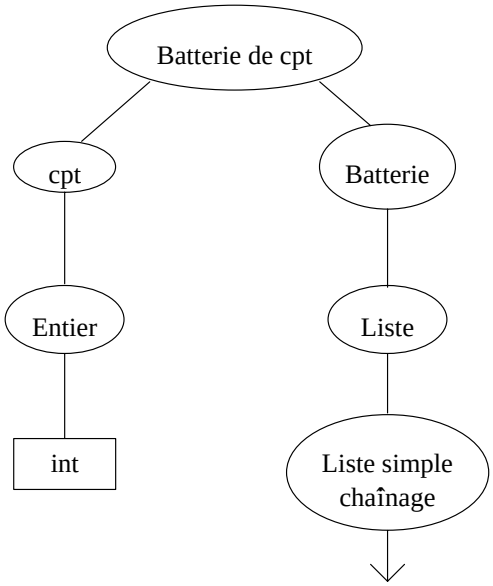


Figure 36. Regroupement des composants

On fait une projection de spécifications fonctionnelles vers le bas, de plus en plus génériques. Dans cette démarche, les services seront imposés par les couches du dessus. En programmation en C++, on se refuse toute complexité. Il faut arriver à rendre les choses les plus simples possibles.

Comment appliquer cette théorie à notre exemple ? On a besoin uniquement des spécifications

fonctionnelles de la couche du dessus. Les spécifications fonctionnelles d'une "batterie de compteurs" montrent qu'il faut :

- `GetNewCounter()` : permet d'avoir un identificateur de compteur

  `IdentC = GetNewCounter(Batterie)`
- `Flush()` : ViderBatterie
- `ListCounters()` (fait intervenir une notion d'itérateur)
- `PrintAllCounters()`
- `ApplyFunctionToAllCounter()`
- `GetCounterValue()`

  `GetCounterValue(IdentC)`

On a des services généraux et des services particuliers, et des actions liées à un compteur précis.

On passe ensuite à la généralisation des synopsis (prototypes)

```
CptIdent GetNewCounter();
void Flush()
...
void ApplyFunctionToAllCounter(void (*func)(compteur));
...
int GetCounterValue(IdentC)
...
```

Si on a ça, on a pratiquement la définition de la classe.

```
classe BatterieDeCompteurs {
public :
        CptIdent GetNewCounter();
        void Flush()
        ...
        void ApplyFunctionToAllCounter(void (*func)(compteur));
        ...
        int GetCounterValue(IdentC)
        ...
};
```

On peut maintenant passer à l'implémentation : on va implémenter les données au fur et à mesure. Une "batterie de compteurs", c'est

— une batterie (postulat),

— de compteurs;

On peut la schématiser comme suit :

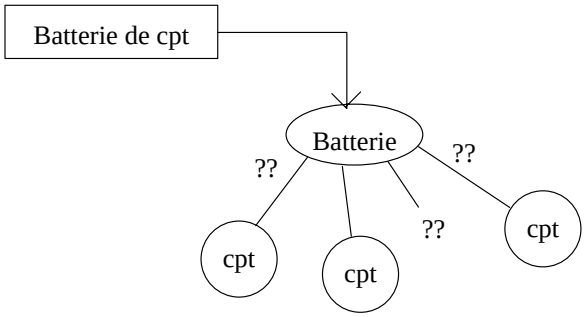


Figure 37. Batterie de compteurs

Remarque : en C++, il faut s'attacher à ce que, dans une classe, il n'y ait que le code spécifique à cette classe.

Par exemple, la notion de `GetCounterValue()` est connue au niveau de la "batterie de compteurs" et au niveau du compteur lui-même, mais n'est pas connue au niveau de la batterie (qui n'est qu'un *linker* de données). Définissons cette fonction :

```
int BatterieDeCompteurs::GetCounterValue(CptIdent Id)
{
        Compteur X;
        X = Batterie.GetData(DataIdent);
        return X.GetValue();
}
```

Il faut que le code tourne. Il faut projeter le résultat.

```
class BatterieDeCompteurs {
        Batterie * Batterie;    // suppose que Batterie existe
public :
        int GetCounterValue(CptIdent);
        ...
};
```

Cette implantation de `Batterie` dans `BatterieDeCompteurs` demande une rectification du code précédent :

```
typedef DataIdent CptIdent;

int BatterieDeCompteurs::GetCounterValue(CptIdent Id)
{
        Compteur * X;
        X = (Compteur *)Batterie->GetData((DataIdent)Id);
        return X->GetValue();
}
```

Nota Bene : une fonction C doit avoir une taille maximale de 2 pages et 5 variables au plus, sinon, disent les américains qui aiment faire des statistiques, c'est générateur d'erreurs à 75%.

Il faut maintenant faire la classe `Batterie`. On prend la feuille de notes contenant la liste des services demandés.

```
class Batterie {
        Array * Array;  // implantation
public :
        void * GetData(DataIdent I);
        ...
};
```

 ISBN 2-909909-02-6

```
void * Batterië:GetData(DataIdent I)
{
        ...
}
```

On décide d'implémenter la batterie sur un tableau. L'intérêt n'est pas de faire une "bonne" `Batterie`, mais d'avoir quelque chose de fini, un programme qui tourne. Les premiers codes sont toujours mauvais. Il faut simuler les outils de niveaux inférieurs.

On doit donc se doter de la définition d'un "tableau" (array). Qu'est-ce qu'un `Array` ? C'est une séquence de données contiguës. La `Batterie` sert à transmettre les informations vers la manipulation d'un tableau. Elle sert de relai entre la couche du dessus et les `Array`s.

On peut décider de ce que `Array` va coder pour `Batterie`.

```
void * Batterië:GetData(DataIdent I)
{
        return Array->GetData((int)I);
}
```

Un tableau est un ensemble de données ordonnées par leur index. Dans le cas donné ici (il s'agit d'uin exempl simple et rapidement implémentable), on a laissé tomber cette notion, et on a retenu uniquement la notion de *container*[56]. On a perdu la notion d'indice.

On doit maintenant construire la classe `Array`.

```
class Array {
        ...
public :
        void * GetData(int);
        ...
};
```

Nota Bene : cette méthodologie suit totalement le "Discours de la Méthode" de Descartes !

Il faut maintenant implémenter l'`Array`. Décidons qu'un `Array` contient un tableau de pointeurs sur des choses dont on ne connaît pas précisément le type.

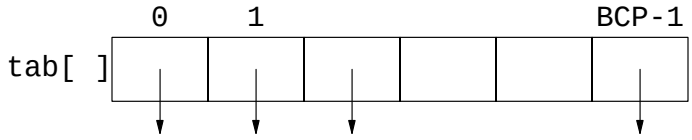


Figure 38. Tableau de pointeurs

On peut maintenant se donner une définition plus complète de la classe `Array` :

```
#define Beaucoup 100

class Array {
        void  * tab[Beaucoup];
        ...
public :
        ...
        void * GetData(int);
        ...
};

void * Array::GetData(int i)
{
        if( i < 0 || i >= Beaucoup )
        {
```

```
                Error.Display("Bleah!");
        }
        return tab[i];
}
```

Dans les exercices à réaliser, la démarche à suivre est la suivante :

1. faire des services sur des types de base : coder des objets qui peuvent vivre tout seuls (tout est objectivable, représentable sous forme d'une écorce avec des boutons poussoirs).

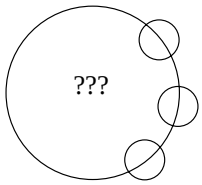


Figure 39. Réalisation des services

2. faire de l'assemblage de problèmes : se donner un agrégat, par exemple :
   - gestion de fichiers,
   - système d'exploitation,
   - fichier d'enregistrements;

   et procéder à la décomposition pour arriver à des objets de base (démarche de réitération).

5.3 Méthode générale

En premier lieu, l'analyse du cahier des charges doit permettre une mise à plat des problèmes. La conception par objet demande, de la part de l'analyse, de faire une schématisation des pôles de traitement.

Le *design* de l'application demande une vue d'ensemble aiguë de celle-ci, et une modélisation abstraite des échanges d'informations entre les différents pôles. Cette vue d'ensemble est primordiale. Il ne faut pas négliger le papier, la gomme et le crayon dans cette phase d'analyse, car de cette vue d'ensemble découleront les différents modules de développement.

Il faut commencer par détecter tous les acteurs de l'application, et faire, pour chacun d'eux, la liste de leurs relations avec le "monde extérieur". On ne s'occupe pas de leur implémentation dans le système. Il s'agit de savoir déterminer les différents domaines en interaction.

Lorsque cette étape a été franchie, se pose alors le problème de l'implémentation de ces acteurs. On peut alors procéder à une analyse plus fine de chacun des pôles, et regarder les différents composants. Ce découpage permet de faire émerger les intersections de fonctionalités, et partant de passer par des étapes successives de généralisation. De nouveaux objets, plus simples, répondant aux specifications "communes" vont prendre naissance. Ces objets deviendront des éléments génériques de l'applicatif. Ils permettent une programmation par regroupement d'"outils de base".

Une étape un peu longue consiste à se doter de ces "outils de base". Il s'agit de se donner un ensemble d'objets fiables, réutilisables dans des contextes différents. Plusieurs solutions peuvent se présenter au programmeur : soit il utilise des outils existants, mais cela demande une **très** bonne connaissance de ces derniers, soit il se les crée.

L'étape suivante consite à faire les regroupements, et à valider l'ensemble des fonctionalités inter-

domaines qui avaient été établies au départ.

La programmation par objet demande donc de faire le "yoyo" entre une analyse descendante et une analyse ascendante. Les shcématisations doivent rester toutjours cohérentes, et être validées à chaque étape. Les objets manipulés doivent intégrer la notion d'*objet parfait*.

Attention : Il ne faut à aucun moment perdre la vue d'ensemble de l'applicatif, ou du domaine traité.

5.4 Exercices

Exercice no. 11 :
Typer un compteur, en définissant ses services (proposer plusieurs implémentations)

Exercice no. 12 :
De la même façon, typer un flag (drapeau).

Exercice no. 13 :
Faire de même pour un sommateur (objet qui ne sait faire que des additions),

Exercice no. 14 :
Typer une string (chaîne de caractères), en définissant les services classiques des chaînes de caractères (concaténation, copie, ...).

Exercice no. 15 :
De la même façon, typer un nom de fichier Unix.

Exercice no. 16 :
Etendre cette même notion à un file-descriptor (descripteur de canal sur fichier unix).

6. Le ++ du C++

Nous allons voir maintenant les nouveautés introduites par le C++ concernant plus particulièrement l'aspect objet du langage.

6.1 Construction/Destruction

6.1.1 Constructeur

Nous avons vu qu'une classe peut maintenir du code associé permettant le traitement des données définies dans son scope. Lorsqu'on instancie une classe, cela génère l'allocation d'un espace pour les récipients de ces données. On peut activer du code spécifique, qui ne sera exécuté que lors de ces créations. C'est le code de ce que l'on appelle le **constructeur** de la classe.

Un constructeur est une fonction membre dont le nom est identique au nom de la classe. Le code définissant cette fonction sera activé une fois que l'allocation des données aura été réalisée. On peut donc mettre dans ce code des instructions d'initialisation spécifiques de ces données.

```
class foo {
        int x;
        int y;
public :
        foo();  // constructeur

        // services de la classe
        void setx(int a) { x = a; };
        void sety(int a) { y = a; };
        int getx() { return x; };
        int gety() { return y; };
};

// definition du constructeur de la classe
foo::foo()
{
        x = 0;
        y = 99;
}

foo a;
foo * pa = new foo;
```

Une construction de classe intervient soit par déclaration d'un objet de ce type, soit par allocation dynamique (utilisation de l'opérateur `new` appliqué au type de la classe).

Un constructeur est une fonction non-typée. Elle ne peut avoir d'instruction permettant un "*retour de fonction*" (i.e. pas de *return value* [57]).

Dans l'exemple donné ci-dessus, les étapes de construction de l'instance `a` de la classe `foo` sont :

- allocation de l'espace des données internes de la classe (les deux entiers `x` et `y`, variables privées),
- initialisation de ces deux entiers par activation du constructeur.

On peut avoir plusieurs fonctions contructeurs. Ceci permet de pouvoir utiliser différentes formes d'initialisation d'une instance. Les constructeurs sont alors reconnus entre eux par leurs prototypes. Cette reconnaissance se fait suivant la façon dont on déclare l'instance de la classe. Ainsi, si on se définit une classe comme suit :

```
class foo {
```

```
        int x;
        int y;
public :
        // premier constructeur
        foo() { x = 0; y = 0; };
        foo(int a) { x = a; y = 0; };
        foo(int a, int b) { x = a; y = b; };
};

foo a;
foo b(10);
foo c(10,20);
```

les instances de classes créées par les déclarations se font suivant les procédés

- `a` : utilisation du constructeur sans paramètres
- `b` : utilisation du constructeur ayant un entier en paramètre
- `c` : utilisation du constructeur ayant deux entiers en paramètres

On a un *overload* implicite des fonctions membres constructeurs de la classe, et reconnaissance dynamique par analyse de l'instruction de déclaration de la classe. Cette reconnaissance est aussi réalisée dans le cas d'une allocation dynamique. On utilise alors l'opérateur `new` en fournissant les paramètres nécessaires à la sélection du constructeur pour le type spécifié :

```
foo * pa;
pa = new foo;

foo * pb;
pb = new foo(10);

foo * pc;
pc = new foo(10,20);
```

Si on veut créer un tableau de classes, avec une initialisation particulière des objets de ce tableau, il est obligatoire d'avoir un constructeur sans paramètres.

```
class foo {
        int i;
public :
        foo() { i = -1; };
};

foo tab[10];
foo * p = new foo[10];
```

Le constructeur est invoqué lors de la génération de chacunes des instances de classe composant le tableau. Dans l'exemple donné ci-dessus, toutes les instances de classes composant le tableau auront leur membre `i` initialisé à `-1`. On ne peut appliquer des constructeurs différents pour la construction des éléments d'un tableau de classes. Le seul constructeur utilisable est un constructeur sans paramètres.

Le C++ permet de réaliser l'initialisation des membres données d'une classe, même lorsqu'il s'agit de types de base. On peut, lors de la définition du constructeur, fournir la liste des données que l'on veut initialiser automatiquement. Pour cela, on fait suivre les paramètres de la fonction constructeur du caractère `':'` et de la liste des variables à initialiser, avec pour chacune, entre parenthèse, la valeur de cette initialisation.

Si on reprend notre exemple de classe avec constructeur, on obtient la définition de classe suivante :

```
class foo {
        int x;
```

```
        int y;
public :
        // premier constructeur
        foo() : x(0), y(0) {;};
        foo(int a) : x(a), y(0) {;};
        foo(int a, int b) : x(a), y(b) {;};
};
```

Dans l'exemple donné ci-dessus, le code interne du constructeur se réduit alors à une instruction vide. Tout est réalisé dans la liste d'initialisation de membres de classe.

L'initialisation des membres données d'une classe, lorsqu'il s'agit de type de base, peut être réalisée avec le résultat de l'évaluation d'une expression. Prenons par exemple une classe contenant un entier. On veut initialiser cet entier avec une valeur particulière si la donnée fournie en paramètre ne correspond pas à un critère précis.

```
class foo {
        int x;
public :
        foo(int entier) : x( (entier < 3) ? 999 : entier ) {;};
        void pr() { printf("%d0,x); };
};

main()
{
        foo f(0);
        f.pr();

        foo g(33);
        g.pr();
}
```

Appliquons cette même théorie pour un membre de type chaîne de catactères. On obtient le code suivant :

```
class foo {
        char * chaine;
public :
        foo(const char * str) :
                chaine((char*)((strlen(str) < 3) ? "Hello" : str)) {;};
        void pr() { printf("%s0,chaine); };
};

main()
{
        foo f("hi");
        f.pr();

        foo g("Bonjour");
        g.pr();

        char * s = "Hello World";
        foo h(s);
        h.pr();
}
```

Remarque : le type "chaîne de caractères" du paramètre du constructeur doit être qualifié de `const` pour ne pas avoir de conflit de type lors de l'initialisation automatique.

6.1.2 Destructeur

On peut aussi associer du code à la destruction d'une instance de classe. C'est le code de ce qu'on appelle le **destructeur** de la classe. Le destructeur est une fonction membre dont le nom est composé du caractère '~' (tilde) accolé au nom de la classe :

```
class foo {
        int x;
        int y;
public :
        // constructeur
        foo();

        // destructeur
        ~foo();
};

// definition du constructeur
foo::foo()
{
        printf("Hello Folks");
}

// definition du destructeur
foo::~foo()
{
        printf("Good Bye");
}
```

Le destructeur est activé lorsque l'instance de la classe est détruite. Cette destruction intervient lorsqu'il y a désallocation de l'espace de donnée de la classe, soit parce qu'il y a destruction du scope courant (destruction implicite), soit par demande explicite de désallocation par utilisation de l'opérateur `delete` appliqué à un pointeur sur instance de classe.

```
void f()
{
        // allocation et construction
        foo a;

        ...
} // fin de scope de fonction : destruction implicite

void g()
{
        // appel a la fonction f()
        f();

        foo * pa;

        // allocation et construction
        pa = new foo;

        // destruction
        delete pa;
}
```

Le code de la fonction destructeur est activé juste avant la désallocation effective de l'espace mémoire réservé pour la variable.

Le destructeur s'applique directement sur l'instance courante. Il ne peut y avoir qu'une seule fonction destructeur pour une classe.

Remarque : les constructeurs et le destructeur attachés à une classe sont obligatoirement des fonctions

définies en section `public` de la classe. En effet, elles sont utilisées par accès externes à la classe.

Remarque : il ne faut pas confondre un constructeur avec une instruction d'initialisation à la déclaration d'une instance de classe. Le constructeur est activé dès que l'allocation est terminée et porte directement sur l'instance en cours de traitement. Une initialisation utilise l'opérateur d'affectation qui modifie totalement le contexte de l'instance. Si les types des opérandes de l'affectation ne sont pas identiques, on peut avoir de sérieuses surprises quant au résultat d'une initialisation sur instance de classe.

```
// construction instance
foo a;

// construction et "initialisation"
foo a = "hello"; // hum!
```

Sachant qu'un constructeur est activé à la création d'une instance et un destructeur à la destruction de cette instance, on peut réaliser une application qui active du code **avant** d'entrer dans le programme principal, et **après** être sorti du programme !

```
// definition de classe
class foo {
        int x;
public :
        // constructeur
        foo() {
                printf("Hello Folks");
        };

        // destructeur
        ~foo() {
                printf("Bye");
        };
};

// declaration d'un global scope programme
foo a;

// programme principal
main()
{
        ;
}
```

Dans cet exemple, l'allocation (et donc la construction) de la variable globale `a` de type `foo` se fait avant d'entrer dans le code du `main()`. La destruction de cette variable se fait en sortie de la fonction principale, donc après le code du `main()`. Vous pouvez le vérifier en changeant l'instruction vide du `main()` par une instruction d'affichage quelconque (argh!).

6.2 Protection par utilisation de const

Lorsqu'on déclare une classe, l'instance est allouée soit dans le segment `data`, soit dans le `bss`, soit dans la `stack`. D'une façon générale, l'objet est alloué dans un segment mémoire modifiable.

On peut vouloir protéger de tout accès modifiant certains membres données de la classe. Pour cela, il suffit de qualifier ces membres données de `const`.

```
class foo {
        const int a;    // protegee
        int b;
public :
```

```
        const int c;    // protegee
        int d;

        foo() : a(99), c(88) {;};
        foo(int x, int y) : a(x), c(y) {;};

        void print() { printf("a=%d, b=%d, c=%d, d=%d0,a,b,c,d); };
};

foo A;

main()
{
        A.print();

        foo B(10,11);
        B.print();
}
```

Dans ce cas, les membres données de la classe sont créés lors de l'instanciation, initialisés lors de la construction de l'objet, et ne seront ensuite plus modifiables par aucune fonction membre..

On peut vouloir protéger l'instance que l'on déclare. Il suffit, pour cela, de qualifier la classe de constante.

```
class foo {
        int a;
public:
        int b;

        foo() : a(99), b(88) {;} ;
        foo(int x, int y) : a(x), b(y) {;} ;

        void print() { printf("a=%d, b=%d0,a,b); };
};


main()
{
        // Declaration d'une classe constante
        const foo A;
        A.b++;          // erreur

        // Declaration d'une classe non-constante
        foo B;
        B.b++;          // ok
}
```

Dans ce cas, tous les membres données de la classe sont implicitement considérés comme constants, pour cette instance. Ils ne pourront donc pas être modifiés au cours de la vie de cet objet.

Remarque : l'initialisation d'un membre donnée constant (par qualificatif local au membre, ou par qualification de l'instance) se fait lors de la construction de l'instance de classe.

On peut protéger les données d'une classe lors de l'utilisation de l'une des fonctions membres de cette classe. La fonction membre doit alors être qualifiée de constante.

```
class foo {
        const int a;
        int b;
public :
        int c;

        foo() : a(99) {;};
```

```
        void Protection() const {
                b = 88;         // erreur
        };

        void Protegee() const;

        void Modifiante() {
                a++;            // erreur
                b = 66;         // ok
                c = 55;         // ok
        };
};

void foo::Protegee() const {
        c = 77;         // erreur
        a--;            // erreur
}
```

Le qualificatif de `const` se place après la liste des paramètres de la fonction, tant pour son prototype que pour sa définition. Dans notre exemple, les fonctions membres `Protection()` et `Protegee()` ne peuvent modifier aucun membre donnée de la classe `foo`. Par contre, la fonction `Modifiante()` peut modifier tout ce qui est modifiable (i.e. qui n'a pas été qualifié de `const` soit localement, soit par qualification d'instance).

Une fonction membre de classe peut retourner une valeur constante, la valeur de l'un des membres données par exemple, tout en étant empêchée de modifier ces membres.

```
class foo {
        int a;
public :
        int b;

        const Function() const { return a; };
};


main()
{
        foo A;
}
```

La position du qualificatif `const` permet de s'appliquer tant au retour de fonction que pour son comportement général vis à vis de la classe.

On peut appliquer le concept de `const` à tous les niveaux de la définition d'une fonction membre. Ainsi, la classe suivante :

```
class foo {
        int a;
public :
        const int Function(const int x) const {
                //...
        };
};
```

nous donne la définition d'une fonction membre recevant en paramètre une constante entière, retournant une constante entière, et ne modifiant pas les données de la classe.

Remarque : La liste des paramètres d'une fonction correspond à la liste des données fournies lors de l'appel à cette fonction. Lorsque les paramètres sont qualifiés de `const`, les données ne doivent pas obligatoirement avoir été qualifiées de `const` à leur définition. Cette qualification des paramètres de la fonction porte sur la qualification de constante dans le scope de la fonction.

6.3 Friend

On a vu, par utilisation des sections `public` et `private`, que le C++ intègre totalement l'encapsulage "fort" des données. On peut ainsi interdire tout accès direct sur les informations maintenues par une classe.

Un mécanisme permet de casser totalement cette protection : les *amis* (on devrait plutôt dire les faux-amis !). Ce concept d'amitié est implémenté par utilisation du mot clé `friend`. Ce qualificatif permet de déclarer qu'une entité étrangère à la classe (donc de provenance externe par rapport au scope de la classe) pourra avoir les autorisations de connaissance totale du scope de la classe courante.

Ce concept peut s'appliquer à une fonction globale programme, à une fonction membre particulière d'une autre classe, ou à une classe toute entière (i.e. pour toutes les fonctions membres de cette classe). Dans ce cas, si la fonction amie manipule une instance de cette classe, elle pourra modifier les données privées de cette classe.

Considérons la définition de classe suivante :

```
// definition de classe
class foo {
        int x;
public :
        ...
        // fonction globale programme amie
        friend void f();
};

// fonction globale programme
void f()
{
        foo a;

        // visibilite de section privee
        a.x = 88;
}
```

La classe `foo` spécifie que la fonction globale programme `f()` est une amie. Elle permet au code de cette fonction, si celle-ci utilise une variable de type `foo`, de pouvoir accéder aux données privées de la classe comme si on se trouvait dans le scope de celle-ci. On peut ainsi modifier le membre donnée `x` privé de l'instance `a` sans devoir passer par les services de la classe.

Attention aux effets de bord ! On a donné la clé de sa maison à son ami, et maintenant ce dernier s'approprie totalement notre domaine !

Considérons maintenant deux classes. Elles ont chacune leur scope, qui, à priori, sont totalement disjoints. Mais supposons que la seconde classe autorise l'une des fonctions membre de la première à s'introduire dans son propre scope :

```
// premiere classe
class foo {
        int x;

public :
        // ...
        void f();
        void g();
};

// seconde classe
```

```
class bar {
        int y;
        // ...

public :
        // ...

        // declaration d'amitie selective
        friend void foo::f();

};


void foo::f()
{
        bar a;

        // modification du membre prive de 'a'
        a.y = 888;
}

void foo::g()
{
        bar a;

        // a.y = 0;      // Error
}
```

On autorise la fonction membre d'une autre classe, donc d'un autre scope, à connaître la partie privée d'une instance de soi. Dans cet exemple, la fonction membre `f()` de la classe `foo` est déclarée être une amie de la classe `bar`. Elle peut donc manipuler directement les membres privés d'une instance de classe `bar`. Par contre, la fonction membre `g()` de la classe `foo` n'a pas été déclarée être une amie. Elle ne peut en aucun cas "violer" le scope de la classe `bar`.

Dans l'exemple suivant, le concept d'amitié est étendu à toutes les fonctions membres d'une classe. Dans ce cas, le qualificatif de `friend` s'applique au type de la classe.

```
// premiere classe
class foo {
        int x;
public :
        void f();
        void g();
};

// seconde classe
class bar {
        int y;
        // ...
public :
        // ...
        // declaration d'amitie totale
        friend class foo;
        // ...
};

void foo::f()
{
        bar a;
        a.y = 999;
}

void foo::g()
```

```
{
        bar a;
        a.y = 888;
}
```

Toutes les fonctions membres de la classe `foo` ont un libre accès sur les sections privées des instances de la classe `bar`. Ainsi, si on sait que toutes les fonctions membres d'une classe doivent "toucher" directement les données privées d'une autre classe, on n'est pas obligé de qualifier d'amie chacune de ces fonction.

Remarque : si on arrive à ce cas de figure, c'est peut-être à cause d'une mauvaise analyse du problème. Il vaut alors mieux se pencher sur l'aspect stratégique de l'analyse, plutôt que de rompre un des aspects forts du concept d'objet par une tactique de "sauve qui peut".

L'expérience montre qu'il est toujours dangereux de donner libre accès à des individus "étrangers" au scope d'une classe. Dans ce cas de figure, il devient parfois difficile de "traquer" les fauteurs de troubles, et de démasquer les générateurs d'erreurs de fonctionnement d'un programme.

6.4 Arborescence de classes

Le C permet d'imbriquer des structures (un champ de structure peut être de type structure). En restant dans la même philosophie, le C++ permet d'avoir des membres de classes de type `class`. La définition du type class du membre peut être donnée dans la définition de la classe "principale".

```
// definition classe principale
class foo {
        int x;
        // definition class pour membre
        class bar {
                int y;
        public :
                bar() { y = 999; };
                void pr() { printf("bar:y %d0,y); }
        } z;
public :
        foo() : z() { x = 888; };
        void pr() { printf("foo:x %d0,x); z.pr(); };
};

main()
{
        foo a;
        a.pr();
}
```

Les membres de la classe principale ne sont pas visibles des membres de la sous-classe (même si on les déclare publics); de même, les membres de la sous-classe ne sont pas accessibles directement par les membres de la classe principale.

Lorsque le type de la sous-classe a été défini, même s'il est inclus dans la définition de la classe principale, il est alors reconnu et utilisable en dehors du contexte de celle-ci.

```
// definition classe principale
class foo {
        int x;
        // definition class pour membre
        class bar {
                ...
        } z;
public :
```

```
        ...
};

main()
{
        // type 'bar' connu
        bar b;
        b.pr();
}
```

Remarque : il est plus "propre" de sortir les définitions de types hors de tout code, et partant d'écrire :

```
// definition classe pour membre
class bar {
        int y;
public :
        bar() { y = 999; };
        void pr() { printf("bar:y %d0,y); }
};

// definition classe principale
class foo {
        int x;
        // classe membre
        bar z;
public :
        foo() : z() { x = 888; };
        void pr() { printf("foo:x %d0,x); z.pr(); };
};

main()
{
        foo a;
        a.pr();
}
```

Il est encore plus souhaitable de mettre chaque définition de type dans des fichiers séparés, celui ayant besoin de l'autre pour exister l'intégrant par un `include`.

Dans le cas d'une classe membre, le constructeur de la classe "principale" doit obligatoirement donner le mode de construction de ce membre si la classe membre n'a elle-même pas de constructeur "sans paramètres". Pour se faire, on suit la règle de définition du contructeur suivante, utilisant le méta-opérateur ':' (deux-points) :

```
class membre {
        ...
public :
        // constructeur classe membre
        membre();
};

class principale {
        membre NomDeMembre;
        ...
public :
        // constructeur classe principale
        principale();
};

// definition constructeur classe principale
principalë:principale() : NomDeMembre() {
        ...
}
```

Lors de la définition du constructeur de la classe principale, en lui fournissant éventuellement une liste de paramètres, on donne aussi le mode de construction des membres de type classe, en les désignant par leur nom. Si on a plusieurs membres de types classe, on en donne la liste. Le mode de construction des membres classe peut faire intervenir des constructeurs paramétrés de ces classes. Dans ce cas, le choix du constructeur se fera suivant les paramètres fournis. On suit donc la règle suivante :

```
class membre {
        ...
public :
        // constructeur classe membre
        membre();
        membre(arguments);
};

class principale {
        membre Membre_1;
        membre Membre_2;
        ...
public :
        // constructeur classe principale
        principale();
        principale(arg_list);
};

// constructeur classe principale
principalë:principale() : Membre_1(), Membre_2() {
        ...
}

// autre constructeur classe principale
principalë:principale(arg_list) : Membre_1(arg), Membre_2(arg) {
        ...
}
```

Remarque : si la classe membre "présente" un constructeur sans paramètres, et que le constructeur de la classe "principale" ne mentionne pas de procédé explicite de construction de ses membres, c'est le constructeur "sans paramètres" qui sera utilisé pour la classe membre (constructeur "par défaut").

```
// definition de la classe "membre"
class bar {
        int a;
public :
        // constructeur "par defaut"
        bar() { a = 0; };

        // constructeur explicite
        bar(int x) : a(x) {;};
        void pr() { printf("%d0,a); };
};

// definition de la classe "principale"
class foo {
        bar b;
public :
        // construction sans precision sur la classe  membre
        foo(int entier) {;};
        void pr() { b.pr(); };
};

main()
{
        foo f(11);
        f.pr();
```

```
}
```

Le résultat de ce programme donne l'affichage de la valeur `0`.

Par contre, si on spécifie, dans la classe principale, le mode de construction de la classe membre, alors il y a recherche pour cette dernière, du constructeur qui lui correspond.

```
// definition de la classe membre
class bar {
        int a;
public :
        // constructeur "par defaut"
        bar() { a = 0; };

        // constructeur explicite
        bar(int x) : a(x) {;};

        void pr() { printf("%d0,a); };
};

// classe ayant un membre de type 'bar'
class foo {
        bar b;
public :
        // constructeur de foo
        // utilise la contruction parametree de 'bar'
        foo(int entier) : b(entier) {;};
        void pr() { b.pr(); };
};

main()
{
        foo f(11);
        f.pr();
}
```

Le résultat de ce programme donne l'affichage de la valeur `11`. En effet, la construction de l'instance de la classe `foo` demande la construction de son membre de type `bar` par utilisation du constructeur paramétré. On a ici transmission de la valeur du paramètre fourni à la construction de l'instance de la classe `foo`.

Un constructeur d'une classe principale peut fournir des valeurs arbitraires lors de la création de ses membres de type classe.

```
class bar {
        int a;
public :
        bar() { a = 0; };
        bar(int i) { a = i; };
};

class foo {
        bar fa;
        bar fb;
public :
        foo() : fa(), fb(10) {;};
        foo(int i) : fa(i), fb() {;};
};
```

Dans cet exemple, le choix du constructeur de `foo` donnera une "initialisation" de ses membres `fa` et `fb` avec éventuellement des valeurs fixées.

Les créations des membres se font dans l'ordre défini par le constructeur de la classe "principale". Ainsi, dans l'exemple suivant :

```
class bar {
        int a;
public :
        bar() { a = 0; printf("bar()0); };
        bar(int i) { a = i; printf("bar(%d)0,i); };
        void pr() { printf("%d0,a); };
};

class foo {
        bar fa;
        bar fb;
public :
        foo() : fa(10), fb() {;};
        foo(int i) : fb(i), fa() {;};
};

main()
{
        // Construction de foo sans parametres
        foo f;

        // Construction de foo avec parametres
        foo g(33);
}
```

la déclaration de l'objet `f` de type `foo`, par utilisation du constructeur sans paramètres commence par créer le membre `fa` puis le membre `fb`. Par contre, la construction de l'objet `g` fait intervenir le constructeur à paramètre entier, qui demande une construction du membre `fb` avant la construction du membre `fa`.

L'activation de ce programme donne les résultats suivants :

```
        bar(10)
        bar()
        bar(33)
        bar()
```

Vu le code des constructeurs, l'ordonnancement dans la création des membres classes n'a ici aucune interférence sur le résultat final (initialisation des membres privés des classes `bar`), mais cela pourraît en avoir lorsqu'il s'agit de code plus compliqués (initialisation de base de données, activation de serveurs, etc).

Si on intègre maintenant une fonction *destructeur* dans la classe `bar`, on peut constater que l'ordre des destructions des objets de types `foo` est en ordre inverse par rapport à celui des déclarations (créations); par contre, pour chacune des instances `foo` détruite, la destruction des classes membres suit l'ordre inverse des déclarations dans la définition des membres de la classe `foo`.

```
// definition du type des membres
class bar {
        int a;
public :
        // constructeur par defaut (sans parametres)
        bar() { a = 0; printf("bar()0); };

        // constructeur avec parametre
        bar(int i) { a = i; printf("bar(%d)0,i); };

        // destructeur (unique)
        ~bar() { printf("deleting value %d0,a); };
};

// definition de la classe principale
class foo {
```

```
        bar fa; // membre de type bar
        bar fb; // membre de type bar
public :
        // constructeur sans parametre
        foo() : fa(10), fb() {;};

        // constructeur avec parametre
        foo(int i) : fb(i), fa() {;};

        // destructeur "implicite"
};

main()
{
        // instanciation sans parametres
        foo f;

        // instanciation parametree
        foo g(33);

        // fin de programme, destruction automatique
}
```

L'activation de ce programme donne, pour les destructions, les messages suivants :

```
deleting value 33
deleting value 0
deleting value 0
deleting value 10
```

où les deux premiers messages correspondent à la destruction de l'instance `g` de la classe `foo`, avec destruction du membre `fb` avant celle du membre `fa`, et les deux seconds à la destruction de l'instance `f` de la classe `foo`.

La destruction des membres ne suit pas l'ordre de création fourni par le constructeur de la classe, mais l'ordre de déclaration dans la définition de la classe. De nouveau, l'ordre de déclaration des membres classe d'une classe peut avoir son importance dans le choix d'implantation des objets constituant l'applicatif.

Forts de tout ça, voyez ce que donne le programme suivant :

```
// definition de la classe "membre"
class bar {
        int a;
public :
        // constructeur "par defaut"
        bar() { a = 0; };

        // constructeur explicite
        bar(int x) : a(x) {;};

        // fonction d'affichage
        void pr() { printf("%d0,a); };
};

// definition de la classe "principale"
class foo {
        bar b;
public :
        // construction sans precision sur la classe membre
        foo(int entier) {;};

        // fonction d'affichage
        void pr() { b.pr(); };
```

 ISBN 2-909909-02-6

```
};

main()
{
        foo f(11);
        f.pr();
}
```

6.5 Operator

Le C++ introduit un nouveau mot clé, `operator`, permettant de redéfinir les opérateurs du C et de les adapter à la gestion d'un objet spécifique. Ce concept permet de garder la même syntaxe que le Ç mais en affectant des fonctionnalités nouvelles aux opérateurs.

Le C++ n'intégrant pas (encore) la possibilité de définir de nouveaux opérateurs en sus de ceux prévus à la base dans la grammaire du Ç l'idée est de dire que l'on utilise les opérateurs existants sur des objets complexes sans changer la forme d'un programme. On veut pouvoir écrire

```
void f()
{
        int i;
        i = 10;

        int j;
        int k;
        k = i + j;
}
```

et reprendre ce programme en ne modifiant que le type des objets utilisés, sans modification des instructions d'exécution (on veut prérserver la transparence du code) :

```
class foo {
public:
        int a;
};

void f()
{
        foo i;
        i = 10;

        foo j;
        foo k;
        k = i + j;
}
```

Prenons par exemple la manipulation de chaînes de caractères. On voudrait que l'"addition" de deux chaînes génère une nouvelle chaîne contenant la concaténation des deux opérandes. En Ç on crée une fonction :

```
/* Programme C */
char * add_chaine(a,b)
char * a;
char * b;
{
        char * ret;

        ret = (char *)malloc(strlen(a) + strlen(b) + 1);
        strcpy(ret,a);
        strcat(ret,b);
        return ret;
```

```
}

main()
{
        char * p;

        p = add_chaine("Hello ","Folks");
}
```

En C++, on va redéfinir l'opérateur d'addition. Cet opérateur va s'appliquer sur des objets C++ (des classes) permettant la manipulation de chaînes de caractères.

```
// definition de l'objet C++ de manipulation de chaines
class string {
        char * str;
public :
        string() { str = 0; };
        string(char * param) {
                str = new char[strlen(param) + 1];
                strcpy(str,param);
        };
        ~string() {
                if( str != 0 )
                {
                        delete str;
                }
        };
        int getsize() { return (str != 0) ? strlen(str) : 0 ; };
        char * getstr() { return str; };
        void pr() { printf("%s0,str); };

        // redefinition de l'operateur
        string & operator + (string & param) {
                char * ptr;
                ptr = new char[strlen(str) + param.getsize() + 1];
                strcpy(ptr,str);
                strcat(ptr,param.getstr());
                string * res = new string(ptr);
                delete ptr;
                return * res;
        };
};

main()
{
        string a("Hello ");
        string b("Folks");
        string c;

        // addition de chaines
        c = a + b;
        c.pr();
}
```

Cette notation apporte une totale transparence sur la manipulation des chaînes de caractères d'un point de vue utilisation des opérateurs.

Une seule restriction : on ne peut redéfinir les opérateurs que dans le scope d'une classe; on ne peut donc pas redéfinir les opérateurs pour des types de base.

La redéfinition des opérateurs est réalisée par une "fonctionnalisation" de l'opérateur concerné. Tous les opérateurs du C (ainsi que les opérateurs `new` et `delete` du C++) peuvent être redéfinis. Cette redéfinition suit la syntaxe suivante :

```
<type> operator <operateur> ( <liste de parametres> )
{
        <code associe>
}
```

La liste des paramètres correspond aux opérandes de l'opérateur concerné (cfr. opérateurs à une, deux, ou trois places du C), sachant que si la redéfinition est donnée en tant que fonction membre de la classe, le premier opérande (le plus "à gauche") est implicite, et correspond à la classe courante (celle sur qui portera l'opération).

```
class foo {
        int x;
public :
        foo(int a) { x = a; };
        int getx() { return x; };
        int operator + ( foo & param ) {
                return x + param.getx();
        };
};

int operator - (foo & OperandeGauche, foo & OperandeDroit)
{
        return OperandeGauche.getx() - OperandeDroit.getx();
}


main()
{
        foo a(10);
        foo b(5);

        int c;
        c = a + b;
        printf("%d0,c);

        c = a - b;
        printf("%d0,c);
}
```

Dans cet exemple, l'opérateur d'addition (`+`) de deux objets de type `foo` est défini en tant que fonction membre de la classe `foo` elle-même. L'opérande gauche n'est pas fourni dans la liste de paramètres, car il est représenté par la classe courante.

Par contre l'opérateur de soustraction (`-`), défini hors du scope de la classe (et n'est donc pas une fonction membre de la classe), mentionne la totalité des opérandes (il n'est rattaché à aucune classe spécifique).

L'ordre de priorité des opérateurs du C reste valable en C++, même en cas de redéfinition de l'action rattachée. Cet ordre de priorité ne peut pas être modifié. De même, le "sens" de lecture des instructions reste identique à celles du C (de la droite vers la gauche, ou de la gauche vers la droite, suivant les opérateurs).

6.5.1 Exemple d'utilisation de `operator`

Considérons le programme principal suivant, utilisant la classe `Integer`, le but étant de trouver la "meilleure" définition pour ce nouveau type :

```
main()
{
        Integer i;
        Integer j;
```

```
        for( i = 0; i < 100 ; i++ )
        {
                j = i;
                j = 2 + i * j;
                j = i + 2;
                printf("%d0,(int)j);
        }
}
```

Comment définir la classe `Integer` pour répondre, sans modification, aux spécifications de ce programme ? Il faut réaliser une classe qui réagisse comme le font les entiers (type de base). A priori, en regardant les instructions, et pour assurer la transparence, on doit définir les fonctionnalités suivantes :

- un constructeur sans argument (pour les déclarations d'objets),
- un opérateur relationnel '`<`', portant sur un `Integer` et un entier, implémentant la relation de vrai/faux,
- un opérateur d'incrémentation '`++`' portant sur un `Integer`,
- un opérateur d'addition '`+`', permettant d'additionner un entier avec un `Integer` et un `Integer` avec un entier,
- un opérateur de multiplication '`*`', permettant la multiplication entre deux `Integer`,
- un opérateur d'affectation '`=`' permettant d'affecter un entier à un `Integer`,
- un opérateur de `cast` permettant de traduire un `Integer` en un `int` pour l'appel de la fonction `printf()`.

Il faut prendre en compte la création de variables temporaires, générées lors d'une instruction multi-opératoire. En effet, l'instruction `2 + i * j` construit, dû au respect des priorités des opérateurs du Ç une première variable temporaire, anonyme, pour la multiplication des deux `Integer`, puis une seconde pour l'addition de celle-ci avec l'entier `2`.

Une première lecture du programme permet d'établir un premier jet de ce que sera la classe `Integer` (on regarde en premier lieu l'aspect prototype des fonctions) :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };
        Integer(int x) { i = x; };
        Integer operator = (Integer x);
        Integer operator + (Integer);
        Integer operator * (Integer);
        operator int();
};
```

6.5.2 Opérateur d'affectation

Regardons l'opérateur d'affectation. En Ç toute partie gauche peut devenir une partie droite d'un opérateur d'affectation se trouvant plus à droite : `c = y = z + 1`. On doit pouvoir garder cette spécificité au niveau du traitement des `Integer`; on doit la reporter au niveau de l'implémentation de la fonction.

```
Integer Integer::operator = (Integer x)
{
        i = x.i;
```

```
        return Integer(i);
}
```

Remarque : les variables temporaires anonymes sont automatiquement détruites en fin d'instruction.

6.5.3 Opérateur d'addition

Considérons l'instruction `j = i + 2`. On doit redéfinir l'opérateur d'addition permettant d'ajouter un entier à un `Integer`, en générant une variable temporaire de type `Integer` utilisée dans l'affectation.

```
Integer Integer::operator + (int x)
{
        return Integer(i + x);
}

Integer Integer::operator + (Integer x)
{
        return Integer(i + x.i);
}
```

La première définition n'est pas obligatoire, car l'addition d'un entier à un `Integer` peut être résolue directement par le compilateur si on a doté la classe d'un constructeur d'`Integer` recevant un entier en paramètre. Dans ce cas, le compilateur cherche dynamiquement à transformer l'entier en un `Integer` pour l'utilisation de l'opérateur d'addition entre deux `Integer`. Cela revient à faire

```
i + Integer(2)
```

6.5.4 Opérateur de multiplication

L'instruction `j = 2 + i * j` fait intervenir la multiplication de deux `Integer`

```
Integer Integer::operator * (Integer x)
{
        return Integer(i * x.i);
}
```

Pour cette instruction, il faut aussi résoudre le problème de l'addition d'un `Integer` à un entier. On ne peut redéfinir les opérateurs directement pour les types de base (entier). De plus, le compilateur n'autorise pas la commutativité automatique sur les opérandes (l'addition n'est plus commutative en C++). Il faut transformer cette expression pour permettre une compréhension par le compilateur. Ce dernier demande un *cast* (conversion de type). Soit on ramène l'`Integer` à un entier,

```
j = 2 + i * j
j = 2 + (int)(i * j)
```

soit on ramène l'entier à un `Integer`

```
j = 2 + i * j
j = (Integer)2 + i * j
```

Il faut donner une indication au compilateur, pour lui permettre de choisir, en fonction de ce que l'on veut effectivement réaliser. L'utilisation du `cast` appliqué à notre exemple est décrite un peu plus loin.

6.5.5 Opérateur relationnel

Considérons l'instruction `i < 100`. Ayant un constructeur permettant de générer un `Integer` à partir d'un entier, on peut définir l'opérateur relationnel comme travaillant sur deux opérandes de type `Integer`, et retournant une valeur entière répondant au critère du vrai/faux classique.

```
int Integer::operator < (Integer x)
{
        return i < x.i;
}
```

6.5.6 Opérateur d'incrémentation

L'opérateur d'incrémentation `++` utilisé dans la boucle `for()` est un opérateur modifiant, unaire, et qui doit suivre la règle du C d'une utilisation en post- ou pré-fixée. Jusqu'à la version 3.0 (non incluse) du C++, il n'y avait pas reconnaissance de cette règle. Les versions 3.0 et suivantes le permettent, mais par un *trick*[58] moyennement élégant...

Cet opérateur étant à une place, sa définition en tant que fonction membre ne devrait pas prendre d'argument, car il porte sur l'objet courant. Cette définition sera celle utilisée pour l'utilisation de l'opérateur en mode préfixé. Une définition avec un argument de type entier permettra de donner, pour ce même objet, une seconde définition de l'opérateur, et indiquera une utilisation en mode postfixé. L'entier défini en argument ne doit pas être utilisé dans le code de la fonction, car il ne sert que d'indicateur pour le compilateur. Dans notre exemple, l'opérateur d'incrémentation aura comme définitions possibles :

```
// definition pour utilisation en prefixe
Integer Integer::operator ++ ()
{
        // code relatif a l'incrementation prefixee
        return Integer(i);
}

// definition pour utilisation en postfixe
Integer Integer::operator ++ (int dummy)
{
        // code relatif a l'incrementation postfixee
        return Integer(i);
}
```

Une même solution est fournie par le C++ version 3.0 quant à l'opérateur de décrémentation. Dans notre exemple, l'opérateur de décrémentation aura comme définitions possibles :

```
// definition pour utilisation en prefixe
Integer Integer::operator -- ()
{
        // code relatif a la decrementation prefixee
        return Integer(i);
}

// definition pour utilisation en postfixe
Integer Integer::operator -- (int dummy)
{
        // code relatif a la decrementation postfixee
        return Integer(i);
}
```

Remarque: pour suivre les règles d'utilisation spécifiées par Bjarne Stroustrup sur les opéerateurs d'incrémentation (resp. décrémentation), l'opérateur d'incrémentation utilisé en pré-fixé doit pouvoir être utilisé en *left-value*. Comme on travaille sur l'objet qui a été incrémenté, on doit retourner cet

objet. Pour se faire, on va donc retourner une référence sur cet objet.

L'opérateur utilisé en post-fixé n'a pas obligation de représenter l'objet lui-même, car utilisé en *right-value* uniquement. L'opérateur va donc retourner une copie de la valeur avant incrémentation de l'objet, qui lui sera effactivement affecté.

Si on consid'ere le programme C suivant :

```
/* Evaluation Left-to-Right de la right-value */
main()
{
    int a = 1;
    int b = 0;
    printf("Start: %d %d0,a,b);

    a = 1;
    b = ++a + ++a;
    printf("prefixe:%d%d0,a,b);
    a = 1;
    b = a++ + a++;
    printf("postfixe:%d%d0,a,b);

    a = 1;
    b = ++a + 2 * ++a;
    printf("prefixe:%d%d0,a,b);
    a = 1;
    b = a++ + 2 * a++;
    printf("postfixe:%d%d0,a,b);
}
```

L'activation de ce programme donne les résultats suivants :

```
Start: 1 0
prefixë 3       6
postfixë        3       3
prefixë 3       9
postfixë        3       5
```

La définition d'un `Integer` correspondant aux mêmes spécifications que les entiers machines utilisés dans ce programme, sera, par exemple :

```
extern "C" {
  void printf();
}

class Integer {
private:
        int x;
        int y;
        void Initialize(int, int);
        void Incrementation();
        void Decrementation();

public:
        // constructors
        Integer(const int = 0, const int = 0);
        Integer(Integer &);

        // cast operator
        operator int();

        // Addition operator
        Integer operator + (Integer &);
```

```
        // Multiplication operator
        Integer operator * (Integer &);

        // Incrementation and decrementation operators
        Integer & operator ++ ();
        Integer operator ++ (int);
        Integer & operator -- ();
        Integer operator -- (int);
};

// Internal methods
void Integer::Initialize(const int a, const int b)
{
        x = a;
        y = b;
}

void Integer::Incrementation()
{
        x++;
        y++;
}

void Integer::Decrementation()
{
        x--;
        y--;
}

// Services (public domain methods)

// Constructors
Integer::Integer(const int a, const int b)
{
        Initialize(a,b);
}

Integer::Integer(Integer & a)
{
        Initialize(a.x, a.y);
}

// Cast into integer
Integer::operator int()
{
        return x;
}

// Addition
Integer Integer::operator + (Integer & a)
{
        return Integer( x + a.x, y + a.y);
}

// Multiplication
Integer Integer::operator * (Integer & a)
{
        return Integer( (x * a.x), (y * a.y));
}

// Prefixed incrementation
Integer & Integer::operator ++()
{
        Incrementation();
```

```
        return *this;
}

// Postfixed incrementation
Integer Integer::operator ++(int a)
{
        Integer Temporary (*this);
        Incrementation();
        return Temporary;
}

// Prefixed decrementation
Integer & Integer::operator --()
{
        Decrementation();
        return *this;
}

// Postfixed decrementation
Integer Integer::operator --(int a = 0)
{
        Integer Temporary (*this);
        Decrementation();
        return Temporary;
}

// Test program
main()
{
        Integer a(1);
        Integer b;
        printf("Start:a=%db=%d0,int(a),int(b));

        // prefixe
        a = 1;
        b = ++a + ++a;
        printf("prefixe:%d%d0,int(a),int(b));
        // postfixe
        a = 1;
        b = a++ + a++;
        printf("postfixe:%d%d0,int(a),int(b));

        // prefixe
        a = 1;
        b = ++a + ++a * 2;
        printf("prefixe:%d%d0,int(a),int(b));
        // postfixe
        a = 1;
        b = a++ + a++ * 2;
        printf("postfixe:%d%d0,int(a),int(b));
}
```

Les tests réalisé avec le compilateur `gcc-2.3.3` donnent les résultats suivants :

```
Start:  a=1    b=0
prefixë       3      6
postfixë      3      6
prefixë       3      9
postfixë      3      9
```

Ce qui n’est pas exactement le résultat attendu. En effet, si lexicalement les expressions utilisant la notation post-fixée sont reconnues, elles ne sont pas diffèerenciées de la notation préfixée d’un point de vue syntaxique. Une modification, qui enlève la transparence, permet d’accéder au code voulu :

```
extern "C" {
  void printf();
}

class Integer {
private:
        int x;
        int y;
        void Initialize(int, int);
        void Incrementation();
        void Decrementation();

public:
        // constructors
        Integer(const int = 0, const int = 0);
        Integer(Integer &);

        // cast operator
        operator int();

        // Addition operator
        Integer operator + (Integer &);

        // Multiplication operator
        Integer operator * (Integer &);

        // Incrementation and decrementation operators
        Integer & operator ++ ();
        Integer operator ++ (int);
        Integer & operator -- ();
        Integer operator -- (int);
};

// Internal methods
void Integer::Initialize(const int a, const int b)
{
        x = a;
        y = b;
}

void Integer::Incrementation()
{
        x++;
        y++;
}

void Integer::Decrementation()
{
        x--;
        y--;
}

// Services (public domain methods)

// Constructors
Integer::Integer(const int a, const int b)
{
        Initialize(a,b);
}

Integer::Integer(Integer & a)
{
        Initialize(a.x, a.y);
}
```

```
// Cast into integer
Integer::operator int()
{
        return x;
}

// Addition
Integer Integer::operator + (Integer & a)
{
        return Integer( x + a.x, y + a.y);
}

// Multiplication
Integer Integer::operator * (Integer & a)
{
        return Integer( (x * a.x), (y * a.y));
}

// Prefixed incrementation
Integer & Integer::operator ++()
{
        Incrementation();
        return *this;
}

// Postfixed incrementation
Integer Integer::operator ++(int a = 0)
{
        Integer Temporary (*this);
        Incrementation();
        return Temporary;
}

// Prefixed decrementation
Integer & Integer::operator --()
{
        Decrementation();
        return *this;
}

// Postfixed decrementation
Integer Integer::operator --(int a = 0)
{
        Integer Temporary (*this);
        Decrementation();
        return Temporary;
}

// Test program
main()
{
        Integer a(1);
        Integer b;
        printf("Start:a=%db=%d0,int(a),int(b));

        // prefixe
        a = 1;
        b = ++a + ++a;
        printf("prefixe:%d%d0,int(a),int(b));
        // postfixe
        a = 1;
        b = a.operator++(0) + a.operator++(0);
        printf("postfixe:%d%d0,int(a),int(b));
```

```
        // prefixe
        a = 1;
        b = ++a + ++a * 2;
        printf("prefixe:%d%d0,int(a),int(b));
        // postfixe
        a = 1;
        b = a.operator++(0) + a.operator++(0) * 2;
        printf("postfixe:%d%d0,int(a),int(b));
}
```

L'activation de ce code donne le résultat escompté précédemment :

```
Start:  a=1    b=0
prefixë       3      6
postfixë      3      3
prefixë       3      9
postfixë      3      5
```

6.5.7 Oper'rateur de `cast`

Revenons à notre premier exemple, et continuons le décryptage des instructions du programme principal. L'instruction `printf("%d",j)` demande l'utilisation d'un opérateur de `cast`, permettant de "traduire" l'objet de type `Integer` en un entier. La définition d'un opérateur de cast suit la syntaxe suivante :

```
<pas de type> operator <type connu> ( <pas d'arguments> )
```

L'opérateur de cast peut alors être utilisé sous la notation habituelle du C ou sous sa forme fonctionnelle C++ :

```
( <type connu> ) <objet>
<type connu> ( <objet> )
```

Dans notre exemple, nous avons besoin de fournir une représentation sous forme d'un entier (type de base) pour un `Integer`. La définition de l'opérateur permettant de résoudre ce problème est, par exemple,

```
Integer::operator int()
{
        return i;
}
```

Nous avons donc maintenant une définition complète du type `Integer` permettant de résoudre le programme `main()` sans modification de ce dernier :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };
        Integer(int x) { i = x; };
        Integer operator = (Integer x);
        Integer operator + (Integer);
        Integer operator * (Integer);
        int operator < (Integer);
        Integer operator ++ ();
        Integer operator ++ (int);
        operator int();
};
```

Regardons de plus près le problème général de conversion de type. Lorsqu'on a un objet, deux types d'opérations peuvent être concevables :

— on ramène tout à cet objet :

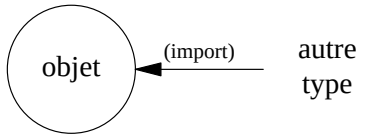


Figure 40. Import type

On est dans le cas où le cast importe des types "extérieurs" vers le type "intérieur" de l'objet. L'objet "étranger" ne doit pas être modifié ni recompilé. La solution consiste à se doter d'un constructeur de l'objet avec en paramètre le type "autre" :

```
Objet::Objet( <autre type> param )()
```

— on exporte notre vision vers les autres types :

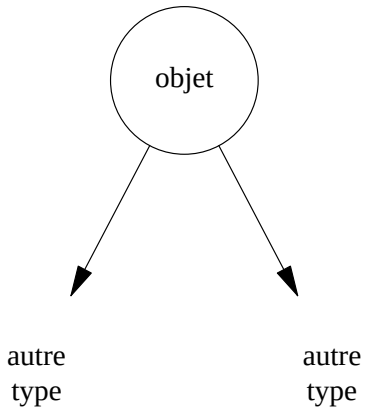


Figure 41. Export type

Les autres types ne sont pas à recoder. On fournit l'opérateur de cast correspondant :

```
Objet::operator <autre type> ()
```

Il faut considérer la stratégie qui économise les choix. En considérant cette formulation, on obtient une nouvelle définition possible de notre type `Integer`, plus simple que la précédente :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };

        // outil d'importation
        Integer(int x) { i = x; };

        // outil d'exportation
        int operator int() { return i; };

        // definition specifique
        void operator ++() { i++; };
};
```

Ceci donne un résultat équivalent à la version précédente si on ne change pas les spécifications des opérateurs.

La première version donne le code suivant :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };
        Integer(int x) { i = x; };
        Integer operator + (Integer);
        Integer operator * (Integer);
        int operator < (Integer);
        Integer operator ++ ();
        operator int();
};

Integer Integer::operator + (Integer x)
{
        return Integer(i + x.i);
}

Integer Integer::operator * (Integer x)
{
        return Integer(i * x.i);
}

int Integer::operator < (Integer x)
{
        return i < x.i;
}

Integer Integer::operator ++ ()
{
        i++;
        return Integer(i);
}

Integer::operator int()
{
        return i;
}

main()
{
        Integer i;
        Integer j;
        for( i = 0; i < 100 ; i++ )
        {
                j = i;
                j = 2 + i * j;
                j = i + 2;
                printf("%d0,j);
        }
}
```

La seconde version donne le code suivant :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };

        // outil d'importation
        Integer(int x) { i = x; };

        // outil d'exportation
        operator int() { return i; };
```

```
        void operator++() { i++; };
};

main()
{
        Integer i;
        Integer j;
        for( i = 0; i < 100 ; i++ )
        {
                j = i;
                j = 2 + i * j;
                j = i + 2;
                printf("%d0,j);
        }
}
```

Renforçons les conditions de travail sur ces `Integer`s. On manipule une donnée de façon lisible immédiatement :

```
main()
{
        Integer i;
        Integer j(3,1,1000);
        Integer k(5);

        for( i = 0; i < 100 ; i++ )
        {
                print(i);
        }
}
```

Par exemple,

- créons des entiers bornés (modèle de sécurisation que l'on a très souvent);
- on veut contrôler les objets, et vérifier que les valeurs fournies sur les opérations répondent à des spécifications internes.

On doit donc renforcer l'intelligence des objets, sans modification du code de base du programme principal.

Voici le "*look*"[59] de ce type de travail (qui maintient la transparence) :

```
extern void Error();

class Integer {
        int i;
        int n_use;
        int min;
        int max;
public :
        Integer(int x = 0, int y = 0, int z = 32000)
        {
                if( y > z ) { Error(); }
                if( x < y ) { Error(); }
                if( x > z ) { Error(); }
                i = x;
                n_use = 0;
                min = y;
                max = z;
        };

        Integer operator ++ ()
```

```
        {
                n_use++;
                if( (i+1) > max ) { Error(); }
                i++;
                return Integer(i,min,max);
        };

        //
        // on donne ici la definition de tous les autres operateurs
        //
};
```

On peut se donner une autre version de l'opérateur d'incrémentation

```
Integer Integer::operator ++ ()
{
        i = rand();
}
```

La boucle permet alors de faire un travail de balayage sur une boucle aléatoire.

Autre possibilité :

```
class Integer {
        int i;
        int step;
public :
        Integer(int s) { i = 0; step = s; };
        Integer operator ++() { i+= step; };
};

Integer i(3); // specification du saut d'incrementation
for( i = 0 ; i < 100 ; i++ )
{
        ...
}
```

La boucle du `main()` reste identique, mais les réactions des objets sont totalement différentes.

6.6 Static

L'utilisation du qualificatif `static` permet de spécifier la rémanence d'une variable. La reconnaissance de cette variable va dépendre du scope de définition de cette variable. Trois cas peuvent cohabiter :

- static fichier,
- static fonction,
- static classe.

6.6.1 Static fichier

Le qualificatif `static` appliqué à un élément connu globalement au scope du programme (information globale) spécifie que la définition de cet élément reste locale au fichier lors de la compilation. Les autres fichiers, compilés séparément, n'auront pas connaissance de cet élément (voire ils en auront une définition locale propre).

Dans le cas du scope fichier, on peut appliquer le qualificatif `static` aux éléments globaux suivants :

— à une variable,

— à une fonction.

Considérons que l'on ait un programme composé de plusieurs fichiers sources. La génération de l'exécutable correspondant nécessite une compilation séparée de ces fichiers. Considérons que l'on définisse une variable globale dans le premier fichier; à priori, elle est reconnue par toutes les fonctions, même dans les autres fichiers, par utilisation du qualificatif `extern` appliqué à la variable.

```
/*------------------- 1er fichier ---------------- */

int a;

main()
{
        a = 10;
        f2();
        f3();
}
/*------------------- 2eme fichier ---------------- */

extern int a;

int f2()
{
        a = 20;
}
/*------------------- 3eme fichier ---------------- */

extern int a;

int f3()
{
        a = 30;
        f2();
}
```

Dans cet exemple, les fonctions `f2()`, `f3()`, et `main()` utilisent la même variable dénommée `a`.

Faisons maintenant en sorte que la fonction `f2()` utilise une variable `a` globale, mais "locale au fichier".

```
/*------------------- 1er fichier ---------------- */

int a;

main()
{
        a = 10;
        f2();
        f3();
}
/*------------------- 2eme fichier ---------------- */

static int a;

int f2()
{
        a = 20;
}
/*------------------- 3eme fichier ---------------- */
```

```
extern int a;

int f3()
{
        a = 30;
        f2();
}
```

Localement au 2ème fichier, au lieu de qualifier la variable d'`extern`, signifiant qu'elle est définie (et allouée) ailleurs, on la qualifie de `static`. Toutes les fonctions définies dans le 2ème fichier vont alors manipuler une donnée qui ne sera connue que dans ce fichier. Les autres fichiers n'ont pas connaissance de cette variable. Tel qu'est écrit notre programme, on a une variable globale fichier `a` commune aux 1er et 3ème fichiers, mais le 2ème fichier manipule une variable `a` qui lui est propre, et qui masque l'existence de la variable commune aux fichiers.

Le qualificatif peut aussi être appliqué aux fonctions. La fonction sera manipulée avec sa définition locale dans le code du fichier.

```
/*------------------- 1er fichier ---------------- */

int a;

main()
{
        a = 10;
        f2();
        f3();
}
/*------------------- 2eme fichier ---------------- */

extern int a;

f2()
{
        a = 20;
}
/*------------------- 3eme fichier ---------------- */

extern int a;

static int f2()
{
        a = 99;
}

f3()
{
        a = 30;
        f2();
}
```

La fonction `main()`, qui appelle la fonction `f2()`, fait appel à la définition globale donnée dans le 2ème fichier. Par contre, la fonction `f3()` fait appel à la fonction `f2()` en utilisant le code défini localement au 3ème fichier pour celle-ci. La définition locale masque la définition globale.

6.6.2 Static fonction

Dans le cadre du scope d'une fonction, le qualificatif `static` s'applique à des données. La variable qualifiée de `static` va être rémanente. Elle est allouée, lors du premier appel à la fonction, non pas dans la stack (pile de travail de la fonction) mais dans le *bss*[60]. Sa référence est gardée jusqu'à la fin

de l'exécution du programme. Lors d'un appel ultérieur à la fonction, la variable existe déjà, sa dernière valeur est toujours accessible.

```
int a;
int f()
{
        static int a;
        a++;
}

main()
{
        a = 0;
        f();
        a = 99;
        f();
}
```

La déclaration de variable `static` dans le scope d'une fonction suit les règles de masquage des variables locales à une fonction : une variable globale (static ou non) sera masquée par la définition locale au scope de la fonction. Ainsi, dans notre exemple, la fonction `f()` manipule une variable `a` qui lui est locale, mais que l'on retrouve à tous les appels. Par contre la fonction `main()` manipule une variable `a` qui est une variable globale du programme.

6.6.3 Static classe

Le C++ introduit un nouveau scope : celui des classes. Lorsqu'on définit une classe, on crée un nouveau domaine : celui de la classe. On a vu que l'on peut créer des données et des fonctions, qui ne seront reconnues que dans ce domaine (les membres de la classe). On peut qualifier des données membres de classe de `static`

```
class Foo {
        static int a;
        int b;
        void f();
        ...
public :
        static int c;
        int d;
        void g();
        ...
};
```

Une fonction membre ne peut être qualifiée de `static` au niveau d'une définition de classe. Elles le sont à priori par défaut, car elles ne sont reconnues que dans le scope de la classe.

La qualification de `static` appliquée à une donnée de la classe a pour effet de rendre commune cette donnée à toutes les instances de cette classe. Ainsi, si on se définit la classe suivante :

```
class Foo {
        static int a;
        int b;
public :
        Foo();
        f(int i) { a = i;};
};

Foo A;
Foo B;
```

le membre `a` est commun aux instances de la classe. Les objets `A` et `B`, instances de la classe `Foo`

partagent cette donnée. Le récipient est commun, et l'accès au membre a par l'une des instances le manipule de façon transparente. Ceci est valable que ce soit pour une définition en section privée ou en section publique.

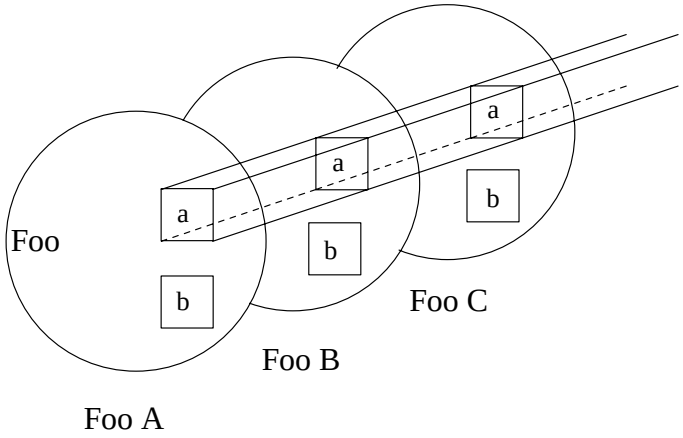


Figure 42. Membre static de classe

La déclaration de ce *common*[61] inter-instance de classe permet d'avoir une donnée partagée. Ceci reste valable dans le cas d'arborescences de classes. On peut construire une "mémoire partagée" entre objets de types différents, par le biais de l'arborescence par inclusion. Considérons les définitions de classes et leurs instanciations suivantes :

```
class foo {
        static int a;
public :
        foo();
};

class bar {
        foo a;
public :
        bar();
};

foo A;
bar B;
```

Ces instances peuvent se schématiser comme suit :

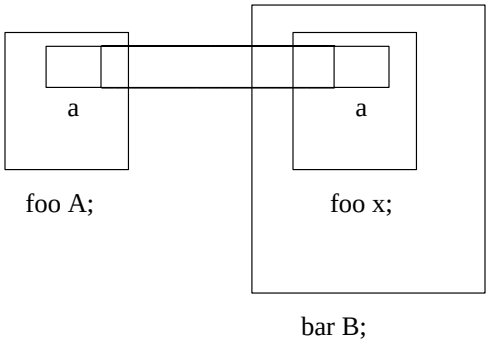


Figure 43. Membre static partagé entre instances

L'objet `A`, de type `foo`, par inclusion d'un membre de type `bar`, et l'objet `B` partagent la même variable `a`, membre de la classe `bar`.

Les permières versions du C++ autorisaient une initialisation directe des données qualifiées de `static` lorsqu'on définissait le type de la classe. Ceci n'est plus autorisé; l'initialisation d'un membre `static` est "garantie" être, par défaut, une initialisation à `0` par le compilateur.

Remarque : il est déconseillé de faire une initialisation de membre static dans le constructeur de la classe, car cela revient à modifier la donnée pour toutes les instances de cette classe à chaque nouvelle instanciation.

Les initialisations de membres static d'une classe peuvent se faire en dehors de tout scope de fonction du programme, sur le même "pied" que les fonctions (i.e. en dehors du `main()` et de toute autre fonction). Du fait de la staticité, les données sont allouées, même si elles ne sont pas accédées. L'espace est alors accessible. On l'accède non pas "au travers" d'une instance explicite de la classe, mais directement par le biais de la classe elle-même :

```
// Classe ayant un membre static
class Foo {
        static int y;
public :
        static int x;
        int Getx() { return x; };
        int Gety() { return y; };
};


// Initialisation des membres static
Foo::y = 88;
Foo::x = 99;


// Fonction du scope programme
main()
{
        // Regarde non autorise en Ansi C++
        printf("%d0,Foo::x);

        // Instanciation de la classe
        Foo A;

        // Verification des valeurs
        printf("%d0,A.Getx());
        printf("%d0,A.Gety());
}
```

L'exemple donné ici montre que le membre static `x` de la classe `Foo` a été initialisé avec la valeur `99`; de même,le membre static `y` a été initialisé avec la valeur `88`, et ce alors que nous n'avons pas encore manipulé une instance de cette classe. Les règles de l'Ansi C++ stipulent qu'il est interdit de "voire" la donnée autrement qu'au travers d'une instance de la classe (l'espace mémoire associé est "protégé" en mode lecture directe). Mais le C++ (utilisation ici de gcc-2.6.3) reste encore assez souple pour nous permettre d'y jeter un coup d'oeil. Il fera quand même un contrôle en ce qui concerne la localisation de la donnée statiç et interdit l'accès aux membres privés (seuls les données en section `public` autorisent un droit de regard).

Un cas typique d'utilisation de membre static d'une classe est celui d'un compteur d'instances.

```
class foo {
        static int cpt;
public :
```

```
        foo() {
                cpt++;
        }
        ~foo() {
                cpt--;
        }
        int Get() { return cpt; };
};

foo x;
foo y;

main()
{
        printf("x.cpt=%d0, x.Get());
        printf("y.cpt=%d0, y.Get());
}
```

Remarque : la version 2.3.3 de `gcc` amène des restrictions d'utilisation de membres statics dans des classes. Leur utilisation est garantie fonctionner correctement lorsqu'il s'agit d'instances allouées en section data (variables globales) ou bss (allocation dynamique), mais pas en section stck (variables locales).

```
class foo {
        static int cpt;
public :
        foo() { cpt++; };
        ~foo() { cpt--; };
        int Get() { return cpt; };
};

foo x;

main()
{
        printf("x.cpt=%d0,x.Get());

        foo * y;
        y = new foo;
        printf("y->cpt=%d0,y->Get());
        printf("x.cpt=%d0,x.Get());

        delete y;
        printf("x.cpt=%d0,x.Get());
}
```

Le programme ci-dessus montre le partage des données statiques entre les données globales et les données allouées dynamiquement par l'opérateur `new`. L'activation de ce programme donne le résultat suivant :

```
x.cpt=1
y->cpt=2
x.cpt=2
x.cpt=1
```

Par contre, le programme suivant montre une inconsistance des résultats lorsqu'il y a mélange de variables globales/dynamiques et locales lors de l'utilisation de la compilation séparée.

Considérons le premier fichier contenant la définition de la fonction principale

```
class foo {
        static int cpt;
public :
        int xx;
```

```
        foo() {
                cpt++;
        }
        ~foo() {
                cpt--;
        }
        int Get() { return cpt; };
        int Getx() { return xx; };
};

foo x;

extern void f();

main()
{
        printf("main:x.cpt=%d0, x.Get());

        foo y;
        printf("main:y.cpt=%d0, y.Get());
        printf("main:x.cpt=%d0, x.Get());

        foo z;
        printf("main:z.cpt=%d0, z.Get());
        printf("main:x.cpt=%d0, x.Get());

        printf("Call function f()0);
        f();

        foo * p;
        p = new foo;

        printf("main:p->cpt=%d0, p->Get());
        printf("main:x.cpt=%d0, x.Get());
}
```

et la définition de la fonction `f()` suivante

```
class foo {
        static int cpt;
public :
        foo() { cpt++; };
        ~foo() { cpt--; };
        int Get() { return cpt; };
};

extern foo x;
foo y;

void f()
{
        printf("f:x.cpt=%d0,x.Get());
        printf("f:y.cpt=%d0, y.Get());

        foo * p;
        p = new foo;
        printf("f:p->cpt=%d0, p->Get());
        printf("f:x.cpt=%d0, x.Get());

        foo z;
        printf("f:z.cpt=%d0, z.Get());
        printf("f:x.cpt=%d0, x.Get());
}
```

Le résultat montre un non partage de la donnée statique entre les variables globales et locales.

```
main:x.cpt=2
main:y.cpt=3
main:x.cpt=3
main:z.cpt=4
main:x.cpt=4
Call function f()
f:x.cpt=4
f:y.cpt=4
f:p->cpt=5
f:x.cpt=5
f:z.cpt=6
f:x.cpt=6
main:p->cpt=6
main:x.cpt=6
```

Remarque : lorsqu'on a défini une classe ayant un membre donnée statique, cette donnée peut être accédée sans qu'aucune instance de la classe n'ait été créée.

```
class foo {
        static int cpt;
public :
        foo() {
                cpt++;
        }
        ~foo() {
                cpt--;
        }
        int Get() { return cpt; };
        void Incr() { cpt += 2;};
};

main()
{
        printf("main:foo::cpt=%d0, foo::Get());

        printf("main: call foo::Incr()0);
        foo::Incr();
        printf("main:foo::cpt=%d0, foo::Get());

        foo * a = new foo;
        printf("main:a->cpt=%d0, a->Get());
        printf("main:foo::cpt=%d0, foo::Get());
}
```

Ceci permet de réaliser une initialisation de la donnée, avant toute instanciation. Le programme ci-dessus donne le résultat suivant

```
main:foö:cpt=0
main: call foö:Incr()
main:foö:cpt=2
main:a->cpt=3
main:foö:cpt=3
```

6.7 This

Le C++ introduit un nouveau mot clé : `this`, pour désigner le nom d'un pointeur pointant sur l'instance courante. Il n'est reconnu que dans le scope des fonctions membres d'une classe.

Considérons la définition suivante :

```
class foo {
        int a;
public :
```

```
        foo() { a = 0; };
        void f();
};

void foo::f()
{
        a = 999;
}

main()
{
        foo A;
        A.f();
}
```

Si on regarde l'instruction `A.f()`, on applique la fonction membre `f()` à l'instance `A` de la classe `foo`. Dans le scope de la fonction membre `f()`, l'objet sur lequel s'applique le code de la fonction est reconnu par un pointeur dénommé `this`. D'une façon générale, `this` est un pointeur de type

```
foo * const
```

où `foo` est le nom de la classe (type de l'instance sur laquelle on applique la fonction membre), sauf si la fonction membre est qualifiée de `const` ou `volatire`. Dans ce cas, le type du pointeur est :

```
const foo * const
volatile foo * const
```

Pour unne fonction membre déclarée `const` et `volatile`, le type du pointeur devient

```
const volatile foo * const
```

C'est un pointeur constant "pouvant pointer sur" un objet de type "celui de la classe courante", et initialisé pour pointer sur cette classe. On peut donc réécrire la fonction membre `f()` comme suit :

```
foö:f()
{
        this->a = 999;
}
```

Lorsqu'on est dans le scope de la fonction membre, le pointeur `this` permet de se voir soi-même.

Prenons un exemple concret. On a vu précédemment l'*overload* des opérateurs, par utilisation du mot clé `operator`. Considérons la redéfinition de l'opérateur d'incrémentation que nous avions donnée pour la classe `Integer`. Transformons son code pour que, en retour de fonction, on ait non pas une nouvelle instance de classe, mais une copie de l'objet sur lequel on travaillait.

```
class Integer {
        int i;
public :
        ...
        Integer operator++();
        ...
};

Integer Integer::operator++()
{
        i++;
        return *this;
}
```

La valeur de retour est une copie de l'instance désignée par le pointeur `this`, soit encore une copie de l'instance courante, modifiée par le code de l'opérateur d'incrémentation.

Pour ne pas avoir de copie, on utilise une référence en type de retour de fonction; il n'y a alors pas de

copie "temporaire".

```
class Integer {
        int i;
public :
        ...
        Integer & operator++();
        ...
};

Integer & Integer::operator++()
{
        i++;
        return *this;
}
```

Dans ce cas, il faut modifier le prototype de la fonction, et spécifier que le retour est une référence. En retour de fonction, on a bien l'objet sur lequel s'appliquait l'opérateur.

Bjarne Stroustrup mentionne, dans son livre *The C++ Programming Language*, au chapitre concernant les anachronismes, que le pointeur `this` peut aussi être utilisé dans le cas où on veut associer un allocateur dynamique spécifique pour une classe.

```
class foo {
        int a;
public :
        foo() {
                // specific allocation
                this = MyFavoriteAllocator(sizeof foo);
        };
        ~foo() {
                // deallocation
                MyFavoriteDeAllocator(this);

                // set this to 0
                this = 0;
        };
};
```

Ceci marche pour des allocations dynamiques de classes (utilisation des opérateurs `new` et `delete`). Dans ce cas, l'allocation de l'instance de classe se fait en utilisant l'allocateur spécifique `MyFavoriteAllocator()`. Les traitements particuliers sur les membres de la classe ne peuvent se faire qu'après allocation effective de l'espace.

Lorsqu'on rentre dans le constructeur, le pointeur `this` est non nul si une allocation a déjà eu lieu. C'est le cas lorsque des objets sont alloués dans la stack (variables en `auto`) ou dans le *bss* (variables en `static`), ou sont des membres d'une autre classe, par déclaration d'instance de classe; l'allocation est déjà réalisée lorsqu'on entre dans le constructeur. Sinon le pointeur `this` vaut zéro (allocation dynamique).

Les appels aux constructeurs pour une classe de base ou un membre classe ne se feront qu'après affectation du pointeur `this`. Si une classe de base modifie, par sa construction, le pointeur `this`, cette même valeur sera répercutée auprès des constructeurs des classes dérivées.

Dans le destructeur de la classe, on invoque la fonction de désallocation correspondante de l'allocateur dynamique, `MyFavoriteDeAllocator()`, puis on met le pointeur `this` à zéro. Cette mise à nul du pointeur interdit au "système" d'appliquer le désallocateur "standard" sur les instance de la classe allouées dynamiquement.

La mise à zéro du pointeur `this` supprime aussi les appels implicites aux destructeurs des classes de base et des membres classe de la classe.

 ISBN 2-909909-02-6

Le programme donné ci-dessous illustre ce schéma :

```
static char Buffer[51200];
static char * Last = Buffer;
static char * EndBuffer = Buffer + 51200;

void * MyFavoriteAllocator(int size)
{
        char * ReturnValue = 0;

        if( size < (EndBuffer - Last) )
        {
                ReturnValue = Last;
                Last += size;
                goto ReturnPoint;
        }
        printf("Out of Memory0);

ReturnPoint:
        printf("Alloc: %lu0,ReturnValue);
        return (void *)ReturnValue;
}

void MyFavoriteDeAllocator(void * WhatToDelete)
{
        printf("Del: %lu0,WhatToDelete);
        if( (char*)WhatToDelete < Buffer
                || (char *)WhatToDelete > EndBuffer )
        {
                printf("Out of range0);
        }
        else
        {
                printf("Ok for deallocation0);
        }
}

class foo {
        int a;
public:
        foo() {
                printf("foo/this: %lu0,this);
                if( this == 0 )
                {
                        this = (foo *)MyFavoriteAllocator(sizeof(foo));
                }
                printf("this=%lu0,this);
        };
        ~foo() {
                printf("~foo/this: %lu0,this);
                MyFavoriteDeAllocator(this);
                this = 0;
        };
};

main()
{
        // espace d'allocation dynamique
        printf("Buffer at %lu0,Buffer);
        printf("EndBuffer at %lu0,EndBuffer);

        // allocation auto
        foo x;
        printf("x at %lu0,(char*)(&x));
```

```
        // allocation dynamique
        foo * y;
        y = new foo;
        printf("y at %lu0,y);
        delete y;
}
```

Des tests avec la version gcc-2.2.2 montrent que le pointeur `this` n'est jamais à zéro, mais par contre, que si l'objet est en mode l'allocation `auto`, `static`, ou membre de classe, la modification du pointeur `this` dans le code du constructeur n'est pas prise en compte. On peut donc avoir le code suivant :

```
static char Buffer[51200];
static char * Last = Buffer;
static char * EndBuffer = Buffer + 51200;

void * MyFavoriteAllocator(int size)
{
        char * ReturnValue = 0;

        if( size < (EndBuffer - Last) )
        {
                ReturnValue = Last;
                Last += size;
                goto ReturnPoint;
        }
        printf("Out of Memory0);
ReturnPoint:
        printf("Alloc: %lu0,ReturnValue);
        return (void *)ReturnValue;
}

void MyFavoriteDeAllocator(void * WhatToDelete)
{
        printf("Del: %lu0,WhatToDelete);
        if( (char*)WhatToDelete < Buffer || (char *)WhatToDelete > EndBuffer )
        {
                printf("Out of range0);
        }
        else
        {
                printf("Ok for deallocation0);
        }
}

class foo {
        int a;
public:
        foo() {
                printf("foo/this: %lu0,this);
                this = (foo *)MyFavoriteAllocator(sizeof(foo));
                printf("this=%lu0,this);
        };
        ~foo() {
                printf("~foo/this: %lu0,this);
                MyFavoriteDeAllocator(this);
                this = 0;
        };
};

main()
{
        // espace d'allocation dynamique
        printf("Buffer at %lu0,Buffer);
```

```
            printf("EndBuffer at %lu0,EndBuffer);

            // allocation auto
            foo x;
            printf("x at %lu0,(char*)(&x));

            // allocation dynamique
            foo * y;
            y = new foo;
            printf("y at %lu0,y);
            delete y;
    }
```

L'activation de ce programme donne le résultat suivant :

```
            Buffer at 134664
            EndBuffer at 185864
            foo/this: 251657552
            Alloc: 134664
            this=134664
            x at 251657552
            foo/this: 195208
            Alloc: 134668
            this=134668
            y at 134668
            ~foo/this: 134668
            Del: 134668
            Ok for deallocation
            ~foo/this: 251657552
            Del: 251657552
            Out of range
```

Ceci illustre que le pointeur `this` n'est jamais nul, et que l'allocateur dynamique local n'est pris en compte que pour des objets alloués par utilisation explicite des opérateurs d'allocation.

Remarque : On peut constater aussi que cette gestion provoque de la perte de place, sauf si on faisait un contrôle plus fin sur la valeur reçue pour `this` (constater par exemple que l'adresse reçue correspond à la stack ou d'ailleurs). Toute modification de l'allocateur dynamique réalisée de cette façon n'est pas idéale; il faut éviter d'écrire un tel code...

Remarque : plutôt que de jouer sur la modification du pointeur `this`, qui risque de disparaître, et n'est pas une caractéristique obligatoire du langage, et partant n'est pas obligatoirement portée, il vaut mieux utiliser l'overload direct des opérateurs `new` et `delete` pour la classe concernée. On a ainsi plusieurs avantages :

1. c'est plus visible,
2. on ne tombe pas dans le piège illustré ci-dessus de perte de place indésirable,
3. le contrôle des allocations est plus fiable.

Un test de ce même programme avec la version gcc-2.6.3 montre qu'il est maintenant interdit de vouloir modifier la valeur attribuée au pointeur `this` par le compilateur; il considère ce dernier comme n'étant pas un récipiendaire d'adresse réel modifiable par le programmeur ("non-lvalue in assignment"), mais est un outil propre à la gestion interne des données par le compilateur.

6.8 Extern

6.8.1 Variables et fonctions

Tout comme pour le Ç le mot clé `extern`, appliqué à un nom de variable ou à un nom de fonction signifie que la définition de l'entité désignée se trouve en dehors du scope courant. Appliqué à une variable globale, cela signifie que la définition est dans un autre fichier; appliqué à une entité déclarée dans le scope d'une fonction, cela signifie qu'il s'agit de quelque chose de global.

```
extern int a;
extern int f(int,char);

int b;

void g()
{
        extern int a;
        extern int b;
        extern int f(int,char);
        ...
}
```

Dans le cadre du C++, le mot clé `extern` a été, pendant un certain temps, étendu aux définitions de types. Ainsi, on pouvait écrire le code suivant :

```
extern class foo;

class bar {
        foo x;
        ...
};
```

Cela permettait d'utiliser le type désigné, sans en avoir encore donné la définition exacte (ni localement, ni par un `include`). A noter qu'on ne peut que faire une "référence" à ce type, et non pas manipuler un objet déclaré de ce type avant que la définition n'en soit effectivement fournie.

Voici un exemple d'utilisation du qualificatif `extern` appliqué à un type :

```
// type defini plus tard
extern class x;

class z {
        int m;
public :
        // mention du type
        void p(x & d);
};

// definition du type
class x {
        int i;
public :
        void p() { i = 20; print(i); };
        friend class y;
        friend void z::p(x&);
};

class y {
        int u;
public :
        void p(x& d) { d.i = 3; };
};

void z::p(x& d)
{
```

```
        d.i--;
}
```

On peut constater que la définition de la fonction membre `p()` de la classe `z` ne peut être donnée que lorsque le type `x` a été effectivement défini. Mais cela n'empêchait en rien de donner la définition de la class `z`. Dans ce cas, la fonction ne peut avoir une définition dans le corps de définition de la classe.

Cette particularité, bien pratique, n'est plus d'actualité dans les versions actuelles du C++ (cela a été retiré des spécifications lors de la normalisation du langage par le groupe de travail).

6.8.2 Langage

L'utilisation du mot clé `extern` a été étendu à la qualification du langage utilisé pour la définition de code. Ceci sert de qualificatif utilisé lors du *link* du programme. Cette utilisation suit la règle suivant :

```
extern "C" {
        declarations C
        definitions C
}

extern "C" declaration
```

Les déclarations sont alors conformes aux règles de grammaire du C. L'exemple ci-dessous

```
// declaration C++
foo & FonctionCplusplus(foo &);

// bloc de declarations C
extern "C" {
        struct bar * FonctionC();
        int tab[100];
}

// declaration "unique" d'instruction C
extern "C" int write();
```

illustre l'utilisation du qualificatif `extern` appliqué en tant que qualificatif de type de langage de définition de code.

Ceci permet d'inclure du code en langage C dans un programme écrit en C++. Lorsqu'on veut utiliser des fonctions d'une librairie Ç il suffit de mettre la liste des prototypes de ces fonctions dans un bloc qualifié de "langage C".

Remarque : cette fonctionnalité permet d'include aussi des codes relatifs aux langages Pascal, ObjectiveÇ et Fortran.

6.9 Exercices

Exercice no. 17 :
Reprendre l'exercice de "statistiques" précedent, et implanter le mécanisme d'initialisation automatique. Rappel: On doit répondre à l'énoncé suivant :

```
main()
{
   Statistic a;

   for(int i = 0; i < 100000; i++)
   {
```

```
            a = rand();
        }
        Print(a.Average);
        Print(a.Minimum);
        Print(a.Maximum);
    }
```

Exercice no. 18 :
Reprendre l'exercice précedent, et donner une solution présentant le mécanisme d'arborescence de classes simples. On doit répondre au même programme.

Exercice no. 19 :
Reprendre l'exercice de gestion de stack du chapitre concernant le C+, et la transformer pour manipuler du C++.

Exercice no. 20 :
Réaliser une classe fichier d'octets qui masque complètement les entrées/sorties sur le fichier (i.e. on ne voit pas l'utilisation des appels systèmes). Se baser sur le programme suivant :

```
#define Create 0666
main()
{
        // foo is the file name
        File f("foo");

        // write 'a' in foo at offset 10
        f[10] = 'a';

        // a mode is allowed
        File g("bar",Create);

        // partial copy from foo to bar
        g[3] = f[10];

        // full copy of foo into bar
        g = f;
}
```

Exercice no. 21 :
Créer la classe d'entiers répondant aux spécifications du programme principal suivant, sans modification du code de ce dernier :

```
main()
{
        Integer i;
        Integer j;

        for( i = 0; i < 100 ; i++ )
        {
                j = i;
                j = 2 + i * j;
                j = i + 2;
                printf("%d0,j);
        }
}
```

Exercice no. 22 :
Réaliser un classe "tableau d'entiers machine" permettant d'écrire le programme suivant :

```
main()
{
        ArrayOfInteger a(100);

        a[10] = a[22];
        a.print();      // affiche la totalite du tableau
        a.print("%03d");
        a[101] = 10;    // provoque une erreur
}
```

Exercice no. 23 :
Reprendre la même classe, et faire en sorte qu'on puisse créer le tableau sur n'importe quel intervalle d'indexs :

```
main()
{
   ExtendedArrayOfInteger a(-1000,1200);

   a[24] = 23;
   a[-200] = a[24];
   a[-999] = 100;
   a.Print("%03d");
}
```

Exercice no. 24 :
Reprendre le même exercice, mais en utilisant une arborescence de classes.

Exercice no. 25 :
Réaliser une classe "tableau d'entiers machine" dynamiquement extensible (on tronquable) après sa création. Le tableau reste une suite contigue en mémoire, comme tout tableau "normal". La définition doit répondre au programme suivant :

```
main()
{
   DynamicArrayOfInteger a(200);

   a[10] = 2;
   a.Print();

   a.Resize(400);
   a[300] = a[10];
   a.Print();

   a.Resize(20);
   a.Print();

   a[30] = 10; // should exit with error 'out of rangé
}
```

Exercice no. 26 :
Reprendre la même classe définition de classe, mais en montant un mécanisme autorisant une extension dynamique du tableau qui soit transparente si on le désire. On doit répondre aux fonctionalités du programme suivant :

```
main()
{
        // tableau avec extension
   DynamicArrayOfInteger a(200,AllowTransparentExtend);
```

```
    a[10] = 2;
    a[300] = a[10];
    a[-1000] = 4;
    a.Print();

        // tableau sans extension possible
    DynamicArrayOfInteger b(100);
    b[200] = 10; // should abort with error 'out of rangé
}
```

Exercice no. 27 :
Créer la classe `chaine` permettant de répondre au programme principal suivant :

```
main()
{
        chaine a("Hello");
        chaine b("Folks");
        chaine c;

        c = a+b; // concatene les chaines
        c.pr(); // affichage du contenu de la chaine

        c -= 'l';       // ote toute occurence de la lettre 'l'
        c.pr();

        c = a + 2;      // met 2 occurences de la chaine a dans c
        c.pr();
}
```

Exercice no. 28 :
Réaliser une classe `String` autorisant toutes les manipulations usuelles sous forme d'opérateurs connus (extrapoler ce qui a été vu dans l'exercice précédent).

Exercice no. 29 :
Créer une classe "pile d'entiers naturels" de taille fixée à la création, et permettant les opérations usuelles sur une pile :

```
main()
{
    StackOfInteger s(100); // 100 is stacks depth

    s.Push(10);
    s.Print();

    s.Push(20);
    s.Print();

    s.Push(30);
    s.Print();

    s.Pop();
    s.Print();

    s.Push(s.Pop());
    s.Print();

    if(s.NonEmpty())
    {
        s.Pop();
    }
```

```
        s.Print();

        s.Flush();
        s.Print();

        while(s.NonFull())
        {
             s.Push(rand());
        }
        s.Print();

        while(!s.Empty())
        {
             s.Pop();
        }

    }
```

Exercice no. 30 :
Réaliser la même chose, mais en utilisant le concept d'arborescence de classes.

Exercice no. 31 :
Reprendre le programme sur la pile d'entiers naturels, et essayer de trouver une notation au moyen d'opérateurs et non plus de fonctions explicitement nommées "push" ou "pop".

7. Héritage des objets

L'aspect objet du langage C++ est marqué par l'intégration de deux concepts forts qui caractérisent les langages objets

- l'héritage descendant, ou encore *héritage par dérivation*,
- l'héritage ascendant, ou encore *polymorphisme*.

Ce sont ces deux concepts que nous allons voir maintenant, sachant que tous les deux sont basés sur un même concept : la dérivation des classes.

7.1 Héritage par dérivation

7.1.1 Classe de base, classe dérivée

Considérons une collection de classes. Une analyse fine de situation montre que l'on a souvent besoin d'une nouvelle définition qui inclut une définition déjà existante, soit dans sa totalité, soit partiellement, avec des fonctionnalités en plus ou en moins.

L'héritage va nous permettre de construire un regroupement de fonctionnalités au travers d'une "arborescence" de classes (il ne s'agit pas ici d'une arborescence par inclusion de classe dans une autre, mais par héritage, au sens "familial" du terme). Le regroupement de classes existantes pour la création d'une nouvelle classe, par intégration de fonctionnalités, s'appelle la *dérivation*. On parle de *classe dérivée* pour le nouveau type construit, et de *classes de base* pour les classes servant à sa construction. La notation syntaxique de cette construction est la suivante :

```
class <base> {
        ...
public :
        // constructeur
        <base>();
};

class <autre base> {
        ...
public :
        // constructeur
        <autre base>();
};

class <derivee> : <base> , <autre base> {
        ...
public :
        // constructeur
        <derivee>();
};

// constructeur de classe derivee
<derivee> :: <derivee>() : <base>(), <autre base>()
{
        ...
}
```

La définition de la classe dérivée demande, par adjonction de la liste des noms de classes de base placée après le signe `:`, d'inclure les données et les fonctions des classes de base. Les définitions des classes de base doivent exister lorsqu'on procède à une dérivation. On dit alors que la classe dérivée **hérite** des classes de base.

Le constructeur de la classe dérivée doit donner le mode de construction des classes de base, en

fournissant la liste des appels aux constructeurs de celles-ci.

Donnons-nous une classe `foo`, et considérons que la classe `bar` inclut toutes les spécifications de la classe `foo`. On va faire dériver la classe `bar` de la classe `foo`.

```
class foo {
        int a;
public :
        foo() { a = 0; };
};

class bar : foo {
        int b;
public :
        bar() : foo() { b = -1; };
};
```

Lorsqu'une classe dérivée n'hérite que d'une seule classe de base, on parle d'héritage simple; si elle hérite de plusieurs classes de base, on parle d'héritage multiple :

```
        class A { ... }
        class B { ... }

// Heritage simple
        class C : A { ... };
        class D : B { ... };

// Heritage multiple
        class C : A, B { ... };
```

7.1.2 Visibilité

Considérons la définition d'une classe. Elle peut être dotée de différentes sections, privées ou publiques.

```
class foo {
        int a;
        void f();
public :
        foo();
        int b;
        void h();
};
```

Tout ce qui est en zone `public` est accessible par le monde extérieur au scope de la classe; par contre tout ce qui est en section `private` n'est visible que de l'intérieur du scope. On peut visualiser cette visibilité par le schéma suivant :

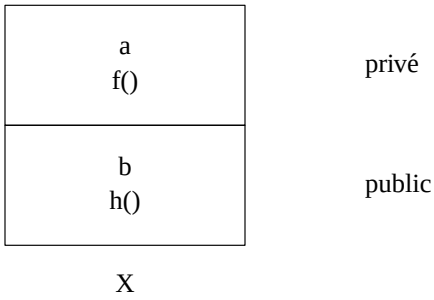


Figure 44. Class: sections internes

La notion de visibilité de scope de classe est étendue à celle d'héritage. On va pouvoir hériter de façon `private` et de façon `public`. Un nouveau qualificatif intervient aussi dans l'héritage : l'héritage en mode `protected`.

7.1.3 Héritage private

Considérons la classe `bar`, et définissons-la comme dérivant de façon privée de la classe `foo`.

```
class foo {
        int a;
        void f();
public :
        foo() {;};
        int b;
        void h();
};

class bar : private foo {
        int i;
        void l();
public :
        int j;
        bar() : foo() {;};
        void g();
};

bar B;
```

Portons notre attention sur l'objet `B`, instance de la classe `bar`. Ne sont visibles, à partir de `B`, que les informations en section `public` de `bar`. On ne peut donc accéder qu'aux membres `j`, `bar()`, et `g()` de la classe `bar`.

```
B.g();
B.j = 10;
```

La classe `bar` hérite de la classe `foo`. Cela signifie qu'elle inclut toutes les fonctionnalités et les données de cette dernière. Mais les accès à ces informations vont suivre la règle de visibilité. Ainsi, ne seront en accès direct, dans le scope de la classe `bar`, que les informations en section `public` de la classe `foo`.

D'autre part, on a spécifié que l'héritage était qualifié de `private`. Cela peut se voir comme si on héritait de la classe `foo` uniquement en section privée de la classe `bar`. Les informations de la classe `foo` sont donc invisibles au monde extérieur au scope de la classe `bar` (même ce qui est en section `public` de `foo` est invisible, car considéré comme défini en section `private` de `bar`).

```
B.g();  // OK
B.h();  // Error : private to bar
B.f();  // Error : private to foo
```

Par contre, à partir du moment où on est entré dans le scope de la classe `B`, via une fonction membre publique de celle-ci, on peut voir tout ce qui y est défini. Ainsi, si on entre dans le scope de la classe `bar`, on "verra" les fonctions et données propres à la classe, ainsi que tout ce qui se trouve défini en section `public` de la classe de base `foo`.

```
// fonction membre de la classe bar
void bar::g()
{
        // membre public de classe bar
        print(j);
        // membres prives de classe bar
        print(i);
        l();

        // membres public de classe foo
        print(b);
        h();

        // membre prive de classe foo
        //print(a);     // Error
}
```

Les membres privés de la classe `foo` ne restent accessibles que via des fonctions membres spécifiées `public`s de cette classe.

Notre "arborescence", et les accès autorisés, peuvent se schématiser comme suit :

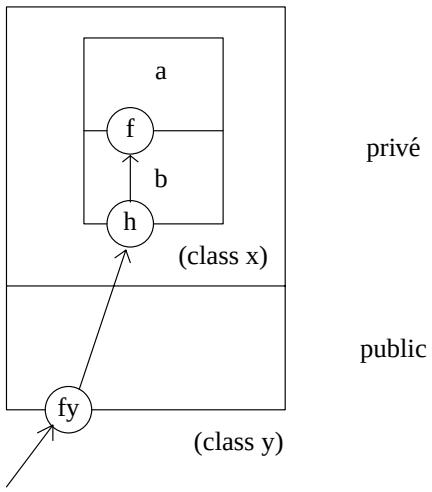


Figure 45. Accès dans héritage

On peut, par utilisation de l'opérateur de domaine `::` faire en sorte que des éléments qualifiés `public` d'une classe héritée en `private` puissent être rendus accessibles par la section `public` de la classe dérivée. Prenons par exemple le programme suivant :

```
class foo {
        int a;
public :
        int b;
        void SetA(int i) { a = i; };
```

```
        void SetB(int i) { b = i; };
        int GetA() { return a; };
        int GetB() { return b; };
};

class bar : private foo {
        int c;
public :
        int d;
        void SetC(int i) { c = i; };
        void SetD(int i) { d = i; };
        int GetC() { return c; };
        int GetD() { return d; };
};

main()
{
        bar x;

        printf("GetB(): %d0,x.GetB());
        x.SetB(10);
        printf("GetB(): %d0,x.GetB());
}
```

La compilation de ce programme génère des erreurs, car le programme principal, du scope programme, cherche à utiliser des fonctionnalités du domaine privé de la classe `bar` (héritage qualifié de `private`). Le compilateur nous renvoit les erreurs suivantes :

```
c13.1.cc: In function ‘int  main ()’:
c13.1.cc:8: the method int  foö:GetB () is private
c13.1.cc:27: within this context
c13.1.cc:6: the method void  foö:SetB (int) is private
c13.1.cc:28: within this context
c13.1.cc:8: the method int  foö:GetB () is private
c13.1.cc:29: within this context
```

Modifions la section `public` de la classe dérivée `bar` en spécifiant que l’on demande une transmission de reconnaissance de certaines fonctionnalités de la classe de base `foo` héritée en mode `private`. On utilise l’opérateur de domaine appliqué aux éléments que l’on veut rendre accessible à partir d’un point hors du scope de la classe `bar`.

```
class foo {
        int a;
public :
        int b;
        void SetA(int i) { a = i; };
        void SetB(int i) { b = i; };
        int GetA() { return a; };
        int GetB() { return b; };
};

class bar : private foo {
        int c;
public :
        int d;
        void SetC(int i) { c = i; };
        void SetD(int i) { d = i; };
        int GetC() { return c; };
        int GetD() { return d; };

        // mise en public d’une partie de l’heritage
        foo::SetB();
        foo::GetB();
};
```

```
main()
{
        bar x;

        printf("GetB(): %d0,x.GetB());
        x.SetB(10);
        printf("GetB(): %d0,x.GetB());
}
```

Lorsqu'on compile ce programme, le compilateur ne détecte plus d'erreurs, car les fonctions `SetB()` et `GetB()` de la classe `foo` ont été rendues du domaine public de la classe `bar`. Une activation de l'exécutable résultant donne les informations suivantes :

```
GetB(): 8820
GetB(): 10
```

La première valeur correspond à "ce qui se trouvait dans la stack", la seconde au résultat de la demande de modification du membre `b` de la classe `foo`.

On ne peut pas rendre public directement une fonction membre privée d'une classe héritée en mode `private`. On ne peut pas écrire le code suivant :

```
class foo {
        int a;
        void SetB(int i) { b = i; };
        int GetB() { return b; };
public :
        int b;
        void SetA(int i) { a = i; };
        int GetA() { return a; };
};

class bar : private foo {
        int c;
public :
        int d;
        void SetC(int i) { c = i; };
        void SetD(int i) { d = i; };
        int GetC() { return c; };
        int GetD() { return d; };

        // essai de mise en public d'une partie de l'heritage
        foo::SetB();
        foo::GetB();
};

main()
{
        bar x;

        printf("GetB(): %d0,x.GetB());
        x.SetB(10);
        printf("GetB(): %d0,x.GetB());
}
```

De nouveau, on tombe sur la détection de section privée de la classe de base. On pourrait "bidouiller" la classe de base en lui demandant de mettre en public une partie de sa section privée, pour que son héritier puisse faire de même... mais la solution est inélégante, et un peu *cludge*[62].

Remarque : cette possibilité de rendre publique une partie des membres publics d'une classe de base alors que celle-ci n'est pas héritée en mode public n'est plus autorisée avec le compilateur `gcc-2.6.3`. On peut par contre encore l'utiliser avec le compilateur de AT&T. Est-ce que cette restriction se généralisera, ou est-ce une erreur "temporaire" de la part du compilateur de Gnu, seul

l'avenir nous le dira.

7.1.4 Héritage public

Considérons maintenant une dérivation qualifiée de `public`.

```
class foo {
        int a;
        void f();
public :
        foo() {;};
        int b;
        void h();
};

class bar : public foo {
        int i;
        void l();
public :
        int j;
        bar() : foo() {;};
        void g();
};

bar B;
```

Dans ce cas, tout ce qui se trouve en section `public` de la class `foo` est considéré comme défini en section `public` de la classe `bar`. Les membres `public` de la classe `foo` seront donc accessibles hors du scope de la classe `bar`.

On peut schématiser cette situation comme suit :

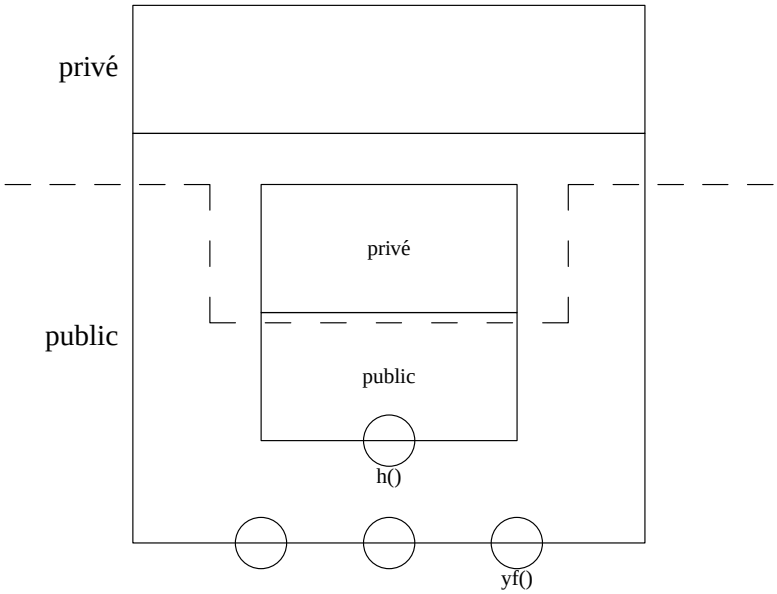


Figure 46. Héritage public

Les membres privés de la classe `bar` ne seront accessibles qu'au travers de fonctions membres de la classe `bar`, et les membres privés de la classe `foo` que par des fonctions membres de la classe `foo`.

```
// membres de classe bar
```

```
B.j = 0;// OK
B.g();  // OK

B.i = 0;// Error
B.l();  // Error

// membres de classe foo

B.b = 0;// OK
B.h();  // OK

B.a = 0;// Error
B.f();  // Error
```

A partir du moment où on est dans le scope de la classe `bar`, toute information visible est accessible. Une fonction membre privée de la classe `bar` peut utiliser une fonction membre `public` de la classe `foo` pour accéder aux informations privées de cette dernière. Par exemple, le code suivant

```
class foo {
        int a;
        void k() { a = 999; };
public :
        foo() { a = 0; };
        void f() { k(); };
        void pr() { printf("a=%d0,a); };
};

class bar : public foo {
        int b;
        void g() { f(); };
public :
        void prn() { printf("b=%d0,b); pr(); };
        void h() { g(); };
};

bar B;

main()
{
        B.prn();
        B.h();
        B.prn();
}
```

se schématise de la façon suivante :

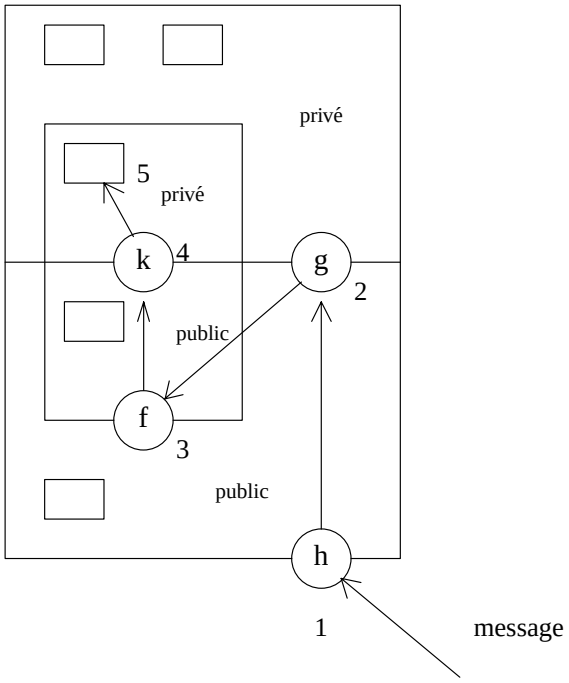


Figure 47. Transmission de message

Dans cet exemple, le point d'accès modifiant se fait à partir de la fonction public `h()` de la classe `bar` (1). Cette fonction fait appel à la fonction `g()` privée de la classe `bar` (2). La fonction `g()` invoque la fonction `f()` public de la classe `foo` (3), qui elle-même fait un appel à la fonction privée `k()` de `foo` (4) pour modifier une donnée privée de celle-ci (5).

7.1.5 Masquage des membres

L'héritage par dérivation permet de regrouper les fonctionnalités de types déjà existants. A priori, ces types forment une collection programmée par ailleurs, que l'on exploite localement. Dans l'exemple que nous avons vu précédemment, la classe `bar` rajoute des données et des fonctions au type `foo` préalablement créé.

L'héritage permet aussi de modifier un aspect des services publics d'un type. Considérons la classe `foo`, et supposons que le code que l'on veuille rattacher à la fonction `g()` ne soit pas celui déjà défini. On crée la classe `bar` dérivée de la classe `foo` de façon `public` (et donc qui présente à priori la fonction `g()` en service), mais on la dote aussi d'une fonction `g()`. La classe `bar` a donc deux définitions de fonction `g()` : celle héritée de la classe `foo`, et celle qui lui est propre. On peut schématiser ce cas de figure de la façon suivante :

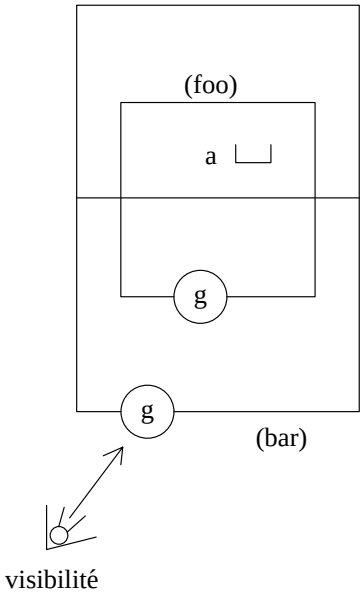


Figure 48. Visibilité en overload

Le code source correspondant à cette description est le suivant :

```
class foo {
        int a;
        void f() { a= 0; };
public :
        foo() { a = 0; };
        void g() { printf("foo:g()0); a = 999; };
        void pr() { printf("foo:a = %d0,a); };
};

class bar : public foo {
public :
        void g() { printf("bar:g()0); };
};

main()
{
        bar B;
        B.pr(); // pr() attachee a foo
        B.g();  // g() attachee a bar
        B.pr(); // pr() attachee a foo
}
```

Si on active le programme associé, on constate qu'il y a masquage de la fonction `g()` membre de la classe `foo` par la fonction `g()` membre direct de la classe `bar`.

```
foöa = 0
bar:g()
foöa = 0
```

L'utilisation de l'opérateur de domaine permet, lorsqu'on est dans le code d'une fonction membre de classe, d'invoquer une fonction membre héritée, en spécifiant le nom de la classe à laquelle appartient la fonction membre masquée.

Modifions le code précédent, et faisons en sorte que la fonction `g()` de la classe `bar` utilise la fonction `g()` de la classe `foo` héritée.

```
class foo {
        int a;
        void f() { a= 0; };
public :
        foo() { a = 0; };
        void g() { printf("foo:g()0); a = 999; };
        void pr() { printf("foo:a = %d0,a); };
};

class bar : public foo {
public :
        void g() {
                printf("bar:g()0);

                // activation de la fonction  g() attachee a foo
                foo::g();
        };
};

main()
{
        bar B;
        B.pr();
        B.g();
        B.pr();
}
```

Une activation de ce programme donne le résultat suivant :

```
foöa = 0
bar:g()
foög()
foöa = 999
```

Ces exemples montrent qu'une fonction définie en section `public` dans une classe de base héritée en `public` reste accessible par un élément extérieur au domaine de la classe, s'il n'est pas redéfini localement dans la classe dérivée. C'est le cas de la fonction `pr()` que nous avions défini dans la classe `foo`. Par contre, s'il y a une redéfinition, c'est celle de la classe dérivée qui est reconnue. C'est le cas de la fonction `g()`.

Si le programmeur connaît explicitement les codes manipulés, il peut toujours invoquer directement le code de la fonction `g()` de la classe `foo` associée publiquement à la classe `bar` en utilisant lui-même, dans son code, l'opérateur de domaine.

```
class foo {
        int a;
        void f() { a= 0; };
public :
        foo() { a = 0; };
        void g() { printf("foo:g()0); a = 999; };
        void pr() { printf("foo:a = %d0,a); };
};

class bar : public foo {
public :
        void g() { printf("bar:g()0); foo::g(); };
};

main()
{
        bar B;
        B.pr();
```

```
        // appel direct a la fonction de la classe foo
        B.foo::g();
        B.pr();
}
```

Le résultat de l'activation de ce programme est le suivant :

```
foöa = 0
foög()
foöa = 999
```

On ne passe alors pas par l'utilisation de la fonction `g()` de la classe dérivée.

7.1.6 Sous-dérivation

L'héritage par dérivation permet de créer une arborescence de classe. Une classe de base d'une classe dérivée peut être elle aussi dérivée d'une autre classe.

Considérons le cas d'une classe `foo` dérivant de façon privée d'une classe `bar`. Construisons une classe `foobar` dérivant de la classe `foo` de manière publique. On a le code suivant :

```
class bar {
        int alpha;
public :
        void func() { printf("bar:func()0); };
};

class foo : private bar {
        int a;
        void f() { printf("foo:f()0); };
public :
        int b;
        void h() { printf("foo:h()0); };
};

class foobar : public foo {
public :
        void fbf() { printf("foobar:fbf()0); };
};


main()
{
        foobar A;

        A.fbf();
        A.h();
//      A.func();       // Error: func() private to x
}
```

Ne sont visibles à partir du domaine externe de la classe `y` que les fonctions définies en sections `public` et héritées de façon `public`.

On peut schématiser les visibilité des sections de chaque classe composant cette arborescence comme suit :

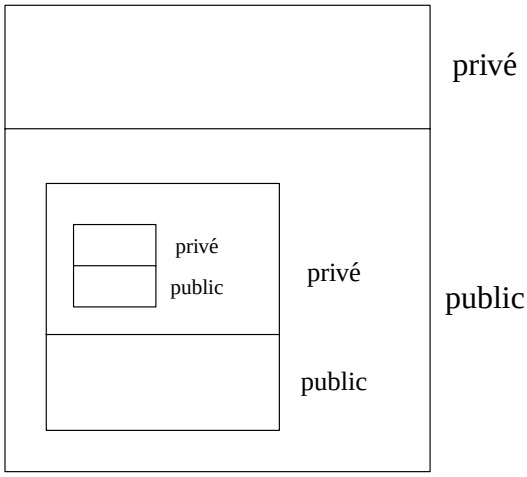


Figure 49. Héritage public/privé

Reprenons cet exemple et considérons maintenant que la classe `foo` dérive de façon `public` de la classe `bar`.

```
class bar {
        int alpha;
public :
        void func() { printf("bar:func()0); };
};

class foo : public bar {
        int a;
        void f() { printf("foo:f()0); };
public :
        int b;
        void h() { printf("foo:h()0); };
};

class foobar : public foo {
public :
        void fbf() { printf("foobar:fbf()0); };
};

main()
{
        foobar A;

        A.fbf();
        A.h();
        A.func();
}
```

L'activation de ce programme donne le résultat suivant :

```
        foobar:fbf();
        foöh();
        bar:fun();
```

Les accès aux différentes fonctions membres de l'arborescence sont autorisés en dehors du scope interne de la classe `foobar` (i.e. on n'est pas "obligé" de rentrer dans celui-ci par le biais d'une fonction membre de la section explicitement qualifiée de `public`), car chacune de ces fonction est définie en section publiç et les héritages se font en mode public.

On peut schématiser cette arborescence comme suit :

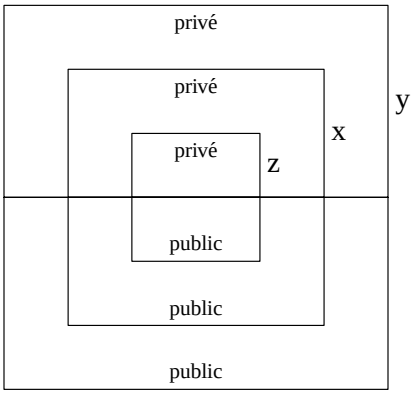


Figure 50. Héritage public/public (1)

Le schéma suivant donne une autre façon de visualiser cette arborescence :

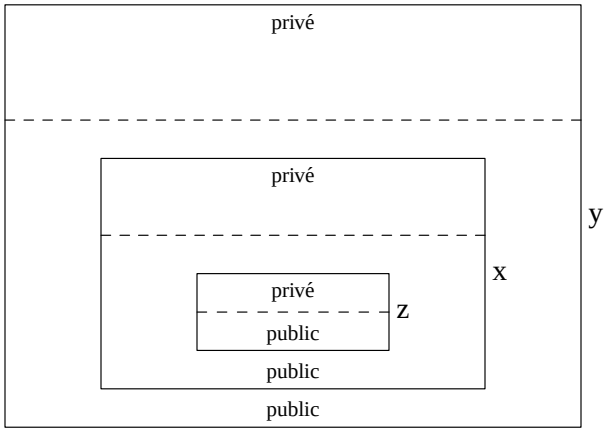


Figure 51. Héritage public/public (2)

Maintenant la fonction `func()` de la classe `z` est "visible" hors du domaine de la classe `y` car elle est définie en section `public`, et est héritée tout du long de façon `public`.

Remarque : les sections *private* restent toujours du domaine privé, et les membres qui y sont définis ne peuvent être accédés qu'au travers de fonctions membres de la section public de la classe concernée.

7.1.7 Héritage protected

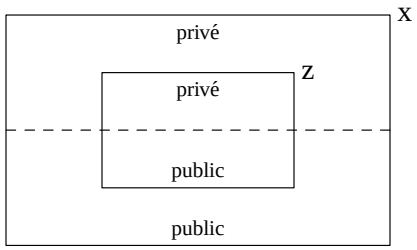
Un autre qualificatif intervient lorsqu'on fait un héritage par dérivation : `protected`. Ce qualificatif permet de limiter la vue `public` de l'héritage à la descendance directe. Il n'y a pas transmission à la sous-descendance.

Reprenons notre exemple, et qualifions l'héritage de `bar` par `foo` de `protected`.

```
class bar {
        int alpha;
```

```
public :
        void func() { printf("bar:func()0); };
};

class foo : protected bar {
        int a;
        void f() { printf("foo:f()0); };
public :
        int b;
        void h() { printf("foo:h()0); };
};


main()
{
        foo B;

        B.h();          // ok public

        // B.f();       // private
         B.func();      // protected
}
```

La classe `foo` est un descendant direct de la classe `bar`. La classe `bar` sera vue comme héritée publiquement par `x`. Cela revient, pour `foo`, à avoir fait un héritage qualifié de `public`. Mais cette vue `public` ne sera que interne au scope des classes, et non perceptible en dehors de celui-ci. On aura une erreur pour un accès à partir du scope programme.



Remarque : il faut bien faire la distinction entre le scope d'une classe, i.e. ce qui se passe dans la définition d'un type, et le scope programme, totalement distinc du scope des classes, qui manipule des instances de classes. Le scope programme ne "voit" les classes que par le biais de la "périphérie" des objets manipulés.

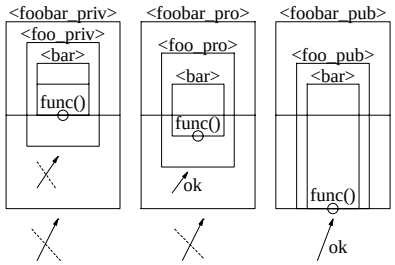
Regardons maintenant ce qui se passe pour une classe `foobar` dérivée de la classe `foo`. L'exemple suivant montre les différents modes d'héritage possible : `private`, `protected`, et `public`.

```
class bar {
public :
        void func() {
                printf("bar:func()0);
        };
};

class foo_priv : private bar {
public :
        void g() { printf("foo_priv::g()0); };
        void h() {
                printf("foo_priv: h() --> ");
                func();         // ok public of private
        };
};
```

```
class foo_pro : protected bar {
public :
        void g() { printf("foo_pro::g()0); };
        void h() {
                printf("foo_prot: h() --> ");
                func();         // ok public of protected
        };
};

class foo_pub : public bar {
public :
        void g() { printf("foo_pub::g()0); };
        void h() {
                printf("foo_pub: h() --> ");
                func();         // ok public of public
        };
};

class foobar_priv : public foo_priv {
public :
        void fbf() {
                printf("foobar:fbf() --> ");
                //func();       // Error: private foo

                printf("0);
        };
};

class foobar_pro : public foo_pro {
public :
        void fbf() {
                printf("foobar_pro: fbf() --> ");
                func();         // ok, protected foo
        };
};

class foobar_pub : public foo_pub {
public :
        void fbf() {
                printf("foobar_pub: fbf() --> ");
                func();         // ok, public foo
        };
};

main()
{
        printf("0sing foo_priv:0);
        foo_priv Bpriv;
        Bpriv.g();      // public foo_priv/prg
        Bpriv.h();      // public foo_priv
//      Bpriv.func();   // Error: private for foo_priv/prg

        printf("0sing foo_pro:0);
        foo_pro Bpro;
        Bpro.g();       // public foo_pro/prg
        Bpro.h();       // public foo_pro/prg with protected
//      Bpro.func();    // Error: protected for foo_pro/prg

        printf("0sing foo_pub:0);
        foo_pub Bpub;
        Bpub.g();       // public foo_pub/prg
        Bpub.h();       // public foo_pub/prg with public
        Bpub.func();    // public foo_pub/prg
```

```
        printf("0sing foobar_priv:0);
        foobar_priv Apriv;
        Apriv.fbf();    // public foobar_priv
//      Apriv.func();   // Error: private foo_priv

        printf("0sing foobar_pro:0);
        foobar_pro Apro;
        Apro.fbf();     // public foobar_pro
//      Apro.func();    // Error: protected foo_pro

        printf("0sing foobar_pub:0);
        foobar_pub Apub;
        Apub.fbf();     // public foobar_pro
        Apub.func();    // public foo_pub
}
```

Les différents héritages donnés dans ce programme peuvent s'illustrer comme suit :



Si on regarde plus spécifiquement chacune des classes `foo_xxx`, elles ont des comportements distincts, qui influent leurs instances, sur leurs classes dérivées, et sur les instances de celles-ci.

Considérons la classe `foo_priv`; elle hérite de la classe `bar` en mode `private`. Une fonction membre de la classe `foo_priv` peut voir, et donc activer, une fonction publique de la classe `bar`. Si on considère maintenant la classe `foobar_priv`, dérivée de la classe `foo_priv`, les membres publics de la classe `bar` non sont pas visibles, car protégés par l'héritage privé. On ne peut pas faire appel à ces fonctions dans le code associé à une fonction membre de la classe `foobar_priv`, et encore moins l'invoquer à partir du scope programme.

Considérons maintenant la classe `foo_pro`; elle hérite de la classe `bar` en mode `protected`. Une fonction membre de la classe `foo_pro` peut utiliser une fonction membre de la section `public` de la classe fAbar. De même, une fonction membre de la classe `foobar_pro`, dérivée de la classe `foo_pro`, peut utiliser cette fonction. On a donc visibilité des membres publics de la classe `bar` comme s'ils avaient été déclarés en section `public`. Par contre, ces membres ne sont pas visibles à partir du scope programme.

Si on considère maintenant la classe `foo_pub`, elle hérite de la classe `bar` en mode `public`. Une fonction membre de la classe `foo_pub` a accès aux membres publics de la classe `bar`. Il en est de même pour une fonction membre de la classe `foobar_pub`, dérivée de la classe `foo_pub`. On constate que ces membres publics sont aussi accessibles à partir du scope programme.

On peut résumer cet état de fait en disant qu'un héritage qualifié de `protected` revient à faire un héritage `public` tant qu'on reste dans le scopre des classes, mais correspond à un héritage `private` dès que l'on en sort et que l'on se trouve dans le scope programme. Ceci démontre la frontière existant entre ces deux domaines.

7.1.8 Section protected

Il s'agit ici de l'application du qualificatif `protected` localement à une section de classe. Les membres de la classe définis entre ce qualificatif et le qualificatif suivant seront considérés comme `protected`, à savoir seront reconnus par un descendant direct, donc comme s'ils étaient en `public`, mais ne le seront pas par un sous-descendant, comme s'ils avaient été définis en `private`. Le qualificatif `protected` appliqué à une section s'utilise de la façon suivante :

```
class foo {
private :
        int foo_i;
        void foo_f();
public :
        int foo_j
        void foo_g();
protected :
        int foo_k;
        void foo_h();
};
```

Les membres définis en section `protected` peuvent être des données et/ou des fonctions. Une vue des données et fonctions de la classe `foo` définie ci-dessus peut se schématiser par :

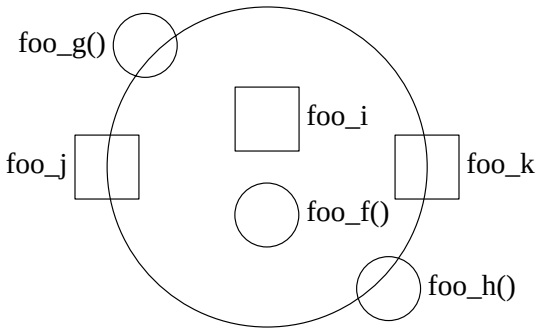


Figure 52. Sections private/protected/public

Considérons la classe `bar` dérivant de façon `public` de la classe `foo`.

```
// classe de base

class foo {
private :
        int foo_i;
        void foo_f() {};
public :
        int foo_j;
        void foo_g() {};
protected :
        int foo_k;
        void foo_h() {};
};


// classe derivee en mode public

class bar : public foo {
private :
        int bar_i;
        void bar_f() {};
```

```
public :
        int bar_j;
        void bar_g() {
                foo_k = 33;
        };

protected :
        int bar_k;
        void bar_h() {
                foo_k = 44;
        };
};


// programme de test
main()
{
        foo F;
        F.foo_j = 10;
//      F.foo_k = 99;   // Error: protected section

        bar B;
        B.bar_j = 20;
//      B.bar_k = 88;   // Error: protected section

        B.foo_j = 33;
//      B.foo_k = 44;   // Error: protected section
}
```

On peut schématiser une instance de la classe `bar` comme suit :

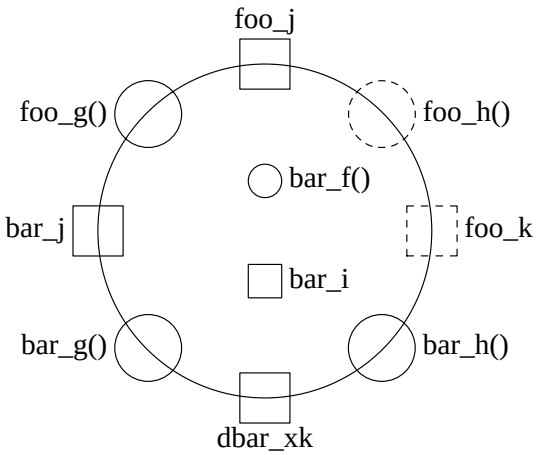


Figure 53. Héritage public et section protected

Si on se donne une instance `B` de la classe `bar`, ne pourront être accédés, à partir de cette instance, et donc hors du scope interne de la classe `bar`, car via une instruction au niveau du scope programme, que les membres qualifiés de `public` dans la classe de base `foo`. Par contre, pour une fonction membre de la classe `bar`, seront accessibles les membres qualifiés de `public` ainsi que les membres qualifiés de `protected`. Localement au scope de la classe dérivée, les membres protected de la classe de base sont vus comme s'ils avaient été qualifiés de `public`. Hors de ce scope, ils sont vus comme s'ils étaient qualifiés de `private`.

Remarque : ces membres restent inaccessibles à partir du scope programme. Les "visibilité" restent du domaine des classes.

Considérons maintenant la classe `foobar` qui dérive publiquement de la classe `bar`.

```
class foobar : public bar {
public :
        void foobar_f() { ... };
};
```

Une instance de la classe `foobar` se schématise comme suit :

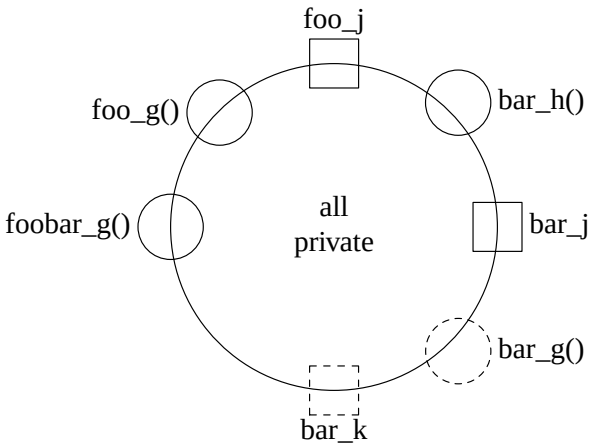


Figure 54. Héritage public/public avec sections protected (1)

On a donc une arborescence de classes ayant des sections `private`, `protected`, et `public`. Dans l'exemple suivant, les héritages sont tous qualifiés de `public` :

```
// classe de base

class foo {
private :
        int foo_i;
        void foo_f() {};
public :
        int foo_j;
        void foo_g() {};
protected :
        int foo_k;
        void foo_h() {};
};


// classe derivee en mode public

class bar : public foo {
private :
        int bar_i;
        void bar_f() {};

public :
        int bar_j;
        void bar_g() {
                foo_k = 33;
        };

protected :
        int bar_k;
        void bar_h() {
                foo_k = 44;
        };
};
```

```
// classe derivee en mode public de derivee

class foobar : public bar {
public :
        void foobar_f() {
//              bar_i = 11;     // Error: private
                bar_j = 22;
                bar_k = 33;

//              foo_i = 44;     // Error: private
                foo_j = 55;
                foo_k = 66;
        };
};


// programme de test
main()
{
        foo F;
        F.foo_j = 1;
//      F.foo_k = 2;    // Error: protected section

        bar B;
        B.foo_j = 3;
//      B.foo_k = 4;    // Error: protected section
        B.bar_j = 5;
//      B.bar_k = 6;    // Error: protected section

        foobar FB;
        FB.foo_j = 7;
//      FB.foo_k = 8;   // Error: protected section
        FB.bar_j = 9;
//      FB.bar_k = 10;  // Error: protected section

        FB.foobar_f();
}
```

Cette arborescence de classes se schématise, d'un point de vue "visibilité" des sections, de la façon suivante :

 ISBN 2-909909-02-6

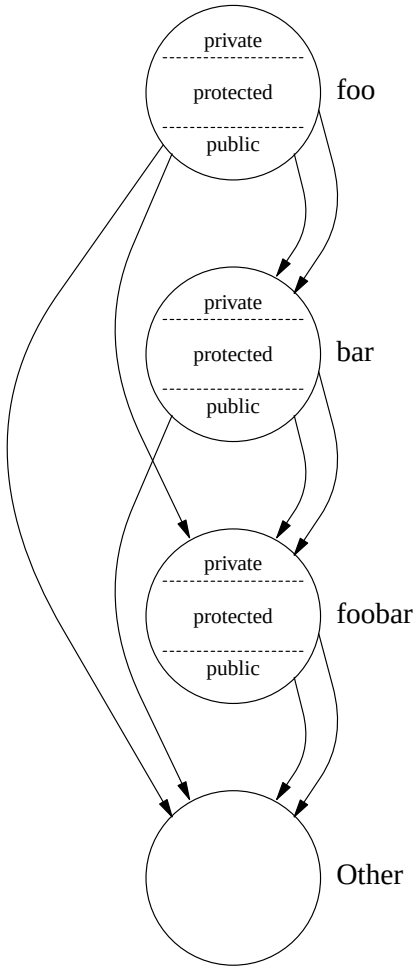


Figure 55. Héritage public/public avec sections protected (2)

La classe `foo` est une classe de base de la classe `bar`. Les membres `protected` de `foo` sont vus, dans le scope de `bar`, comme s'ils étaient qualifiés de `public`. La classe `bar` étant une classe de base de la classe `foobar`, la classe `foo` n'est pas une base directe de `foobar`. Elle n'est pas définie dans son scope de façon directe. Les membres `protected` de `foo` seront donc considérés comme `private` à partir de `foobar`. Par contre, les membres `protected` de `bar` sont accessibles, comme s'ils avaient été qualifiés de `public`.

7.1.9 Construction d'une arborescence de classes

Considérons une arborescence de classe par mixage d'inclusion et d'héritage descendant. On se donne une classe `foo` et une classe `bar`, indépendantes, ainsi qu'une classe `foobar` qui hérite de `foo` et dont l'un des membres est une classe `bar`. Le constructeur de la classe `foobar` doit fournir les moyens de générer les classe de base et classe membre.

```
// definition de classe foo
class foo {
        int foo_a;

public :
        foo(int a) {
```

```
                printf("C de foo0);
                foo_a = a;
        };

        ~foo() { printf("D de foo0); };

        void pr() { printf("%d0,foo_a); };
};

// definition de classe bar
class bar {
        int bar_a;

public :
        bar(int a) {
                printf("C de bar0);
                bar_a = a;
        };

        ~bar() { printf("D de bar0); };

        void pr() { printf("%d0,bar_a); };
};

// definition de classe foobar, derivee de foo
// l'un de ses membres est de type bar
class foobar : foo {
        bar foobar_a;
        int foobar_b;

public :
        foobar(int,int,int);
        ~foobar() { printf("D de foobar0); };

        void pr();
};

// constructeur de classe foobar
foobar::foobar(int a, int b, int c) : foo(c), foobar_a(a)
{
        printf("C de foobar0);
        foobar_b = b;
}

void foobar::pr()
{
        foobar_a.pr();
        foo::pr();
        printf("%d0,foobar_b);
}

main()
{
        foobar Y(10,20,30);

        Y.pr();
}
```

Lorsque l'on créée une instance de la classe `foobar`, l'ordre de construction des instances des classes membre et de base de cette classe est le suivant :

1. création des classes de base, dans leur ordre de dépendance,

2. création des classes membres,
3. création des données propres à la classe courante.

L'ordre des destructions est inverse; on a donc

1. destruction des données propres (foobar)
2. destruction des classes membres (bar)
3. destruction des classes de base (foo)

Le résultat de l'activation du programme donné ci-dessus est le suivant :

```
C de foo
C de bar
C de foobar

10      // valeur de ya (x)
30      // valeur de base (z)
20      // valeur de y (y)

D de foobar
D de bar
D de foo
```

7.1.10 Héritage multiple

On parle d'héritage multiple lorsqu'une classe dérive de plusieurs classes de base. En reprenant les classes `foo`, `bar`, et `foobar` définies précédemment, construisons une classe `Other` héritant de ces classes.

```
class Other : foo, bar, foobar {
        int Other_a;
public :
        Other(int,int,int,int);
        ~Other() { printf("D de Other0); };
        void pr();
};

Other::Other(int a = 10, int b = 20, int c = 30, int d = 40)
: foo(888), bar(999), foobar(a,b,c)
{
        printf("C de Other0);
        Other_a = d;
}
```

Une instance de la classe `Other` contient donc une instance de la classe `bar`, de la classe `foobar` et de la classe foo, sachant que la classe `foobar` se compose d'une instance de la classe `foo` et d'une instance membre de type `bar`.

On peut schématiser l'instance `T` de type `t` comme suit :

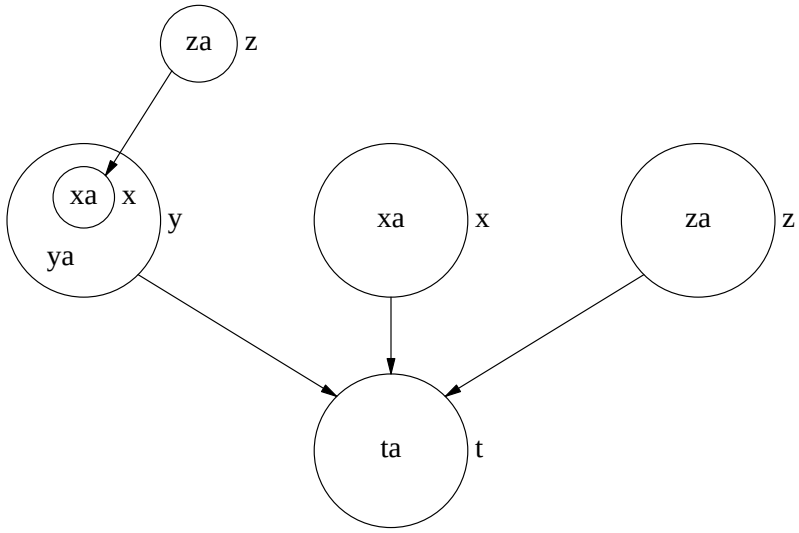


Figure 56. Héritage multiple

On constate la présence de plusieurs occurences d'instances de type `bar` et de type `foo` dans l'arborescence composant la classe `Other` : celles définies en direct, et celles résultant de la construction de la classe `foobar` héritée.

La fonction membre `pr()` de la classe `Other` permettant de visualiser les différentes données allouées peut se définir comme suit :

```
void Other::pr()
{
        printf("%d0,ta);
        foobar::pr();
        bar::pr();
        foö:pr();
}
```

A noter que le print de `foo_ae` est inaccessible à partir de `Other` directement, car c'est une donnée privée de la classe `foo`.

Remarque : la compilateur gcc-2.6.3 impose une restriction quand à la construction d'une telle arborescence. En effet, telle que nous l'avons définie, la construction de la classe `Other` nécessite la construction de deux classes `foo` ayant un seul niveau de différence : construction par héritage direct et construction d'une classe dérive de la classe `foo`. Le compilateur Gnu exige un double niveau de construction; nous avons donc introduit une classe intermédiaire `FooFoo` pour les besoins locaux de construction, cette classe servant de palier pour la classe `foo` "directe". On obtient donc le code suivant :

```
class foo {
        int foo_a;
public :
        foo(int a) {
                printf("C de foo0);
                foo_a = a;
        };
        ~foo() { printf("D de foo0); };
        void pr() { printf("%d0,foo_a); };
};

class bar {
```

```
        int bar_a;
public :
        bar(int a) {
                printf("C de bar0);
                bar_a = a;
        };
        ~bar() { printf("D de bar0); };
        void pr() { printf("%d0,bar_a); };
};

class foobar : foo {
        bar foobar_a;
        int foobar_b;
public :
        foobar(int,int,int);
        ~foobar() { printf("D de foobar0); };
        void pr();
};

// constructeur de classe foobar
foobar::foobar(int a, int b, int c) : foo(c), foobar_a(a)
{
        printf("C de foobar0);
        foobar_b = b;
}

void foobar::pr()
{
        foobar_a.pr();
        foo::pr();
        printf("%d0,foobar_b);
}

class FooFoo : public foo {
public :
        FooFoo(int a) : foo(a) {;};
};

// definition de classe Other a heritage multiple
class Other : FooFoo, bar, foobar {
        int Other_a;
public :
        Other(int, int, int, int);
        ~Other() { printf("D de Other0); };
        void pr();
};

Other::Other(int a = 10, int b = 20, int c = 30, int d = 40)
: FooFoo(888), bar(999), foobar(a,b,c)
{
        printf("C de Other0);
        Other_a = d;
}

void Other::pr()
{
        printf("%d0,Other_a);
        foobar::pr();
        bar::pr();
        FooFoo::pr();
}

main()
{
```

```
        // instance de classe Other
        Other T;
}
```

L'activation de ce programme donne le résultat suivant :

```
C de foo
C de bar
C de foo
C de bar
C de foobar
C de Other
D de Other
D de foobar
D de bar
D de foo
D de bar
D de foo
```

Lorsqu'on manipule l'héritage multiple, on peut qualifier cet héritage pour une partie (ou plusieurs parties) de l'ensemble des classes de base. Ainsi, on peut qualifier l'héritage de `public`, `private` ou `protected`, suivi de la liste des classes de base concernées.

```
// exemple de qualification de derivation
class Other : protected foobar, public bar, private foo {
        ...
};

// autre exemple de qualification de derivation
class Other : foobar, public bar , foo {
        ...
};
```

Nota bene : Le défaut de qualification d'héritage est le qualificatif `private` pour les compilateurs C++ jusqu'à la version 3.0 de AT&T. Vu que la plupart du temps on crée des héritages en mode `public`, il avait été proposé que le qualificatif "par défaut" du C++ 3.0 de AT&T devienne ce mode `public`, mais cette proposition n'a pas encore été agréée.

7.1.11 Héritage virtual

Un nouveau qualificatif intervient avec l'héritage : `virtual`. Ce qualificatif permet de ne pas avoir de multiples occurences d'une classe de base au travers de l'arborescence. Considérons l'exemple donné précédemment, et modifions-le en qualifiant l'héritage de la classe `foo` de `virtual` dans la définition des classes `foobar` et `Other`.

```
class foo {
        int foo_a;
public :
        foo(int);
        ~foo();
        void pr();
};

class bar {
        int bar_a;
public :
        bar(int);
        ~bar();
        void pr();
};

class foobar : virtual foo {
```

```
        bar foobar_a;
        int foobar_b;
public :
        foobar(int,int,int);
        ~foobar();
        void pr();
};

class Other: foobar, bar, virtual foo {
        int Other_a;
public :
        Other(int, int, int, int);
        void pr();
};
```

Toutes les "occurences" de classe `foo` qualifiées de `virtual` au travers de l'arborescence composant la classe `Other` ne vont être représentées que par une seule et même instance de classe `foo` (il n'y a pas plusieurs instanciations de la classe `foo` pour les qualificatifs `virtual`). On peut schématiser une instance de notre nouvelle classe `Other` comme suit :

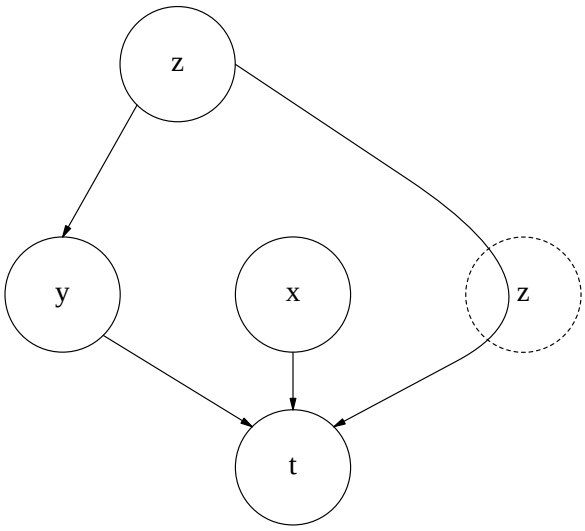


Figure 57. Héritage virtuel

Par contre il y a bien une instance distincte pour les classes de base non spécifiées `virtual`. Modifions l'exemple donné plus haut, et faisons dériver la classe `bar` de la classe `foo`, sans qualificatif particulier.

```
class foo {
        int foo_a;
public :
        foo(int a = 0) { foo_a = a; };
        ~foo() {};
        void pr(){};
};

class bar : foo {
        int bar_a;
public :
        bar(int a, int b) : foo(a) { bar_a = b; };
        ~bar() {};
        void pr(){};
};
```

```
class Intermed : virtual foo {
public :
        Intermed(int a) : foo(a) {};
        ~Intermed() {};
};

class foobar :  Intermed {
        bar foobar_a;
        int foobar_b;
public :
        foobar(int a,int b,int c) : Intermed(a), foobar_a(b,c) {
                foobar_b = c;
        };
        ~foobar() {};
        void pr(){};
};

class Other : foobar, bar, Intermed {
        int Other_a;
public :
        Other(int a, int b, int c, int d) :
                foobar(a,b,c), bar(c,d), Intermed(d) {};
        ~Other() {};
        void pr(){};
};

main()
{
        Other O(11,22,33,44);
}
```

On a donc trois occurences de dérivation à partir de la classe `foo` :

— `bar` dérive de `foo`,

— `foobar` dérive de `foo`,

— `Other` dérive de `foo`.

Du point de vue des instances effectivement allouées, les classes `foobar` et `Other` partagent la même, car les dérivations sont qualifiées de `virtual`, par contre on a une instance distincte pour la classe `foo`. On a donc deux instances réelles dans l'arborescence de notre programme.

Remarque : Il semblerait que le compilateur gcc-2.6.3 soit maniaque sur la reconnaissance des classes de base de types identiques dans un arborescence "proche". Ceci amène a devoir créer des classes intermédiaires. Ceci est peut-être dû à la gestion par stack des différents sous-arbres de l'héritage.

7.2 Polymorphisme

Le C++ n'implémente que deux formes d'héritages : la dérivation et le polymorphisme, basées sur un même mécanisme. Ces deux formes permettent d'implémenter tout le reste. Les héritages que les autres langages orientés objets et les langages objets proposent sont implémentables en compilé par le C++ par ces deux seuls outils.

7.2.1 Description

Le polymorphisme de base est un outil permettant d'implémenter les polymorphismes attendus des langages objets. Le mécanisme de polymorphisme utilise les deux formes d'héritage existant en C++ :

- l’héritage descendant : un enfant hérite de ses parents, i.e. un enfant va incorporer dans son domaine les fonctionnalités de ses parents :

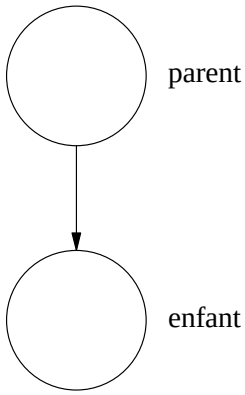


Figure 58. Héritage descendant

- l’héritage ascendant : un parent hérite de ses enfants, i.e. un parent peut faire "remonter" du code à partir de ses enfants :

Figure 59. Héritage ascendant

  Dans ce cas, l’arborescence de classe est vue au travers du parent; il en est le point d’entrée. Les services du parent vont pouvoir muter en fonction d’une décision d’orientation du "regard" du parent porté sur sa descendance.

Dans le cas du polymorphisme, les objets de base (parents) vont profiter des codes réalisés par les enfants. Les parents vont fournir le minimum d’informations pour que les enfants réalisent des choses, puis le parent pourra, sur choix spécifique, utiliser le code réalisé par l’enfant. Le parent est alors capable de trouver dans ses enfants une forme de lui-même, en "étendu". Le parent se verra doté, de par ses enfant, d’une éventuelle adjonction de données. Les enfants sont des héritiers (au sens "familial" du terme) de leurs parents. Le parent définit à l’avance la liste des fonctionnalités qu’il pourra trouver et invoquer chez ses enfants. Il définit les prototypes de ce qu’il veut retrouver chez ses enfants. Il peut même rendre le codage de ces fonctions obligatoires chez ses enfants. Il peut aussi, si aucun code n’a été prévu chez l’enfant, spécifier d’utiliser un code "par défaut" qu’il aura lui-même défini.

Un parent impose donc un certain nombre de choses sur sa descendance, de façon plus ou moins forte.

Un objet va ainsi pouvoir changer de représentation de façon fluxuante. On a une dynamique dans le temps de la représentation du parent. Cette dynamique est codée et compilée. Elle prend effet lors de l’exécution du programme.

Ce type de polymorphisme est une des caractéristiques principales des objets informatiques.

Les enfants sont compilés après la compilation des parents. Le compilateur ne peut plus résoudre à la compilation la dynamique de la personnalité. On a donç pour l’implémentation de cette mécanique, une partie "interprétée" permettant la reconnaissance du contexte lors de l’exécution.

Cette dynamique est donc implémentée par le **seul** mécanisme d’interprétation du C++. C’est le seul point où on ne sait pas résoudre le problème à la compilation.

```
Parent.f()   --->   (enfant).f()
```

A l'exécution, on a un mécanisme qui permet de trouver le *bon* `f()` à activer. On a une indirection sur le code. Il y a une recherche dynamique dans des tables à l'activation. On a un certain indéterminisme sur le parent lors de l'exécution.

On va coder au niveau parent des objets non finis, génériques. Le codage final se fait au niveau des enfants, qui englobent *un peu plus* que ce que les parents ont prévu. L'applicatif va manipuler les descriptifs génériques.

Le polymorphisme consiste à créer un héritage ascendant : à partir d'une classe de base, on va utiliser le code de fonctions définies dans des classes dérivées, et remappées[63] par le compilateur.

La classe de base va qualifier les fonctions remappables de `virtual`. Ce qualificatif ne s'applique qu'aux fonctions membres (pas aux données). Les classes de base ainsi définies deviennent des *classes génériques*[64].

Des classes génériques qui n'ont que des fonctions membres qualifiées de `virtual`, sans code associé, et sans membres données sont appelées des *classes abstraites*[65].

L'utilisation du polymorphisme intervient lorsqu'on désire qu'une classe générique prenne la forme d'un objet particulier de l'ensemble des objets définis, à un moment donné, puis une autre forme, à un autre moment, avec remapping dynamique lors de l'exécution du programme. Lors de la compilation, on ne sait pas que l'objet générique va se commuer en autre chose. Toute l'application est codée avec une utilisation de l'objet générique. Ce n'est qu'à un point particulier de l'applicatif que le domaine de l'objet sera déterminé, et que le code effectivement utilisé sera celui d'un objet particulier de la collection d'objets reconnus.

Considérons une classe `foo` définie comme étant une classe générique, donc ayant au moins une fonction membre qualifiée de `virtual` :

```
class foo {
        int i;
public :
        void f() { i++; };
        virtual void g() { i--; };
};
```

Le polymorphisme se définit au niveau du parent. Il est le seul à savoir qu'il va "parasiter" les services de ses enfants. Les enfants ne savent jamais qu'ils seront parasités par leurs parents.

Donnons-nous un enfant, héritier de notre classe de base :

```
class bar : public foo {
        int j;
public :
        void h() { j++; };
        void g() { j--; };
};
```

La fonction `h()` de l'enfant n'est pas déclarée dans le parent. Elle ne sera jamais utilisable par son parent, en cas de polymorphisme.

```
B b;
b.f();  // heritee de A
b.g();  // locale a B avec overload
b.h();  // locale a B
```

Modifions la classe `bar` en ajoutant une fonction `f()`.

```
class bar : public foo {
        int j;
public :
```

```
        void f() { j = 0; };
        void h() { j++; };
        void g() { j--; };
};
```

Si on déclare un objet de type `foo` (type de la classe de base) et un objet de type `bar`, chacun peut activer les fonctions membres du type qui lui correspond.

```
foo a;
a.f();  // f() de foo
a.g();  // g() de foo

bar b;
b.h();  // h() de bar
b.f();  // f() de bar par overload
b.g();  // g() de bar par overload
```

Donnons-nous maintenant des pointeurs

```
foo * pa;
pa = new foo;
p->f(); // f() de foo
p->g(); // g() de foo
```

Remarque : quand on a un pointeur, on a deux objets

- un pointeur (`pa`), récipiendaire d'adresse,
- un objet pointé (résultat du `new` ici), espace mémoire

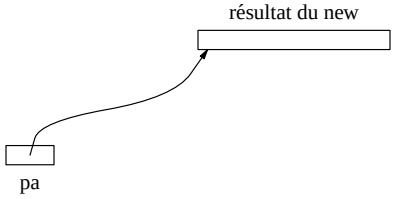


Figure 60. Pointeur et allocation dynamique

On a donc un type pointeur et un type pointé.

En C++, il est possible de faire pointer un pointeur sur un objet de type différent du type attendu par le pointeur, pourvu que ce soit un type dérivé du type attendu.

```
foo * pa = (foo *) new bar;
```

On alloue un objet de type enfant et on l'attache à un pointeur de type parent. Il existe une différence entre le type pointé et le type pointeur. Mais le pointeur garde la mémoire de son type propre et de son type pointé. Le pointeur C++ garde une information supplémentaire par rapport au pointeur C : le type de l'objet pointé.

Ceci n'est pas inimaginable, car dans le type enfant, on a bien inclusion d'un type parent.

Considérons le pointeur `p` de type pointeur sur `foo` pointant sur un objet de type `bar`, et drRegardons les deux appels de fonctions.

```
foo * p = (foo *) new bar;
p->f();
p->g();
```

La fonction `f()` utilisée est celle du type `foo` car le pointeur est de type `foo`, sans qualificatif de `virtual`. Par contre, la fonction `g()` de `foo` étant qualifiée de `virtual`, c'est la fonction `g()` de `bar` qui est effectivement activée.

Le pointeur ayant une notion de l'objet pointé, lorsqu'il tombe sur le qualificatif de `virtual`, il sait qu'il doit aller chercher le code dans l'enfant (routage dynamique).

A noter qu'à la compilation, rien ne donne d'indications sur ce routage vers une autre adresse. On a la création d'une table interne de routage.

Le pointeur, pris tout seul, ne peut pas marcher. Il faut le faire pointer sur un objet explicitement alloué.

On peut implémenter ce mécanisme avec un deuxième outil qui, lui aussi, n'alloue pas de données : les références. Considérons l'instruction suivante :

```
foo & r = * new foo;
```

L'expression `new foo` réalise une allocation, et retourne l'adresse de l'objet alloué. L'expression `* new foo` correspond à "ce qu'il y a à l'adresse", donc à l'objet lui-même. L'expression `foo & r` correspond à déclarer une référence sur un objet de type `foo`. L'objet référencé doit exister. La référence doit être initialisée à la déclaration. L'instruction `foo & r = * new foo;` déclare donc une référence sur un objet de type `foo` alloué dynamiquement. On peut, par la référence, accéder aux fonctions de l'objet référencé :

```
r.f();          // f de foo
r.g();          // g de foo
```

Considérons maintenant une référence de type `foo` référençant un objet de type `bar` (on utilise le *cast* permettant la correspondance de types).

```
foo & r = *((foo *)new bar);
r.f():          // f de foo
r.g();          // g de bar
```

Le polymorphisme s'applique par utilisation de la table de routage appliquée aux fonctions qualifiées de `virtual`.

Cette forme d'écriture est plus *innée* que la forme utilisant les pointeurs. Sans une vue générale du programme, on ne peut dire si c'est un "vrai" `foo` ou une référence sur un autre type qui est utilisé lorsqu'on manipule la référence. Le polymorphisme est ici totalement transparent.

Donnons-nous une autre classe, héritant aussi de la classe de base `foo` :

```
class other : public foo {
        int k;
public :
        h() { k++; };
        g() { k--; };
};
```

Donnons-nous un pointeur de type `foo`, et des objets de type `foo`, `bar` et `other` :

```
foo * p;
foo a;
bar b;
other c;

p = &a;
p->g();         // g de foo
p = (foo *)(&b);
p->g():         // g fr bar
p = (foo *)(&c);
p->g();         // g de other
```

Le pointeur `p` est "de type `foo`", mais au cours de l'exécution du programme, il pointe sur des objets différents, ayant une même base `foo` : le type attendu du pointeur. Le code exécuté sera différent

suivant les étapes de l'exécution du programme.

Remarque : on ne peut pas faire ce type de modification en utilisant des références, car la définition d'une référence ne peut changer en cours d'exécution. Il n'existe pas de dynamique polymorphe sur les références.

Soit `p` un pointeur sur un objet de type `foo`. Considérons les codes de boucles suivants :

```
foo * p;

for( ;; )
{
        p = SelectChild();
        p->g();
}

for( ;; )
{
        foo & r = SelectChildRef();
        r.g();
}
```

On ne peut dire, à la vue de ce code, de quelle fonction `g()` il s'agit.

Remarque : le polymorphisme est souvent implanté dans les couches basses du code.

Une fonction `SelectChild()` va être une fonction de la forme :

```
foo * SelectChild()
{
        int i;

        i = rand();     // fonction aleatoire
        if( !(i % 2) )
        {
                return((foo *)new foo);
        }
        else if( !(i % 3) )
        {
                return((foo *)new bar);
        }
        else
        {
                return((foo *)new other);
        }
}
```

Le code de la boucle `for()` n'a aucune connaissance des allocations différentes possibles réalisées dans la fonction `SelectChild()`. Dans la programmation concrète, on va manipuler des objets qui ne savent pas ce qu'ils sont au moment du codage.

7.2.2 Exemple : les shapes

Quand on fait un dessin, on le définit comme un ensemble de figures. Une figure peut être une figure de base ou un dessin. On pourrait se donner une définition de type *yacc* comme suit :

```
dessin : figure
figure : figure_de_base
       | dessin
figure_de_base : point
       | ligne
       | spline
       | arc
```

```
| cercle
| rectangle
| triangle
```

On peut tout faire dans le shape dessin. Le moindre point, la moindre ligne, doivent pouvoir réagir en fonction des demandes faites par l'utilisateur. Ainsi, si dans le dessin suivant :

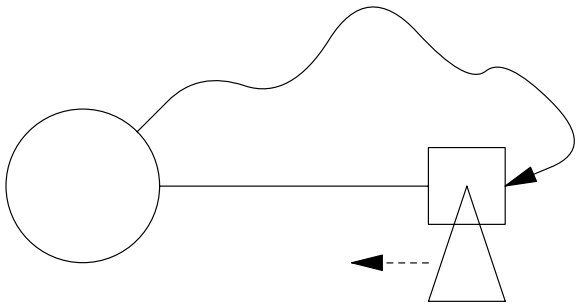


Figure 61. Mouvement de shape

on demande un déplacement du triangle, la figure "triangle" doit savoir réagir, et les autres éléments du dessin aussi, car leur environnement a peut-être été modifié.

D'un point de vue implémentation de shapes, on va utiliser une classe `figure`; elle sera une base de toutes les figures pouvant constituer un dessin. Elle va maintenir les informations communes à toute description d'objet. Elle proposera un ensemble de services communs aux différentes figures.

```
class Figure {
        coord lieu;
public :
        virtual move();
        virtual bigger();
};
```

Selon que les enfants sont difficiles à implémenter ou non, s'ils sont des figures de base, ils vont implémenter des définitions particulières des fonctions, sinon utiliser les définitions globales.

Prenons en exemple la définition de la classe `triangle` :

```
class triangle : public Figure {
        Figure a, b, c;
};

Figure & f = triangle;
f.move();       // code de Figure (classe de base)

triangle t;
t.move();       // heritage de la fonction
```

Dans le cas de l'utilisation de la fonction `move()` à partir d'un objet de type `triangle`, le code de la fonction est celui donné dans la définition de la fonction de la classe de base. En effet, la classe `triangle` ne définit pas de fonction qui lui soit propre. On utilise celle héritée par dérivation.

Si on veut forcer une classe dérivée à donner une définition d'une fonction réutilisable par polymorphisme, on ne va pas donner de code de définition de la fonction dans la classe de base, mais lui affecter la valeur `0`.

```
class Figure {
        coord lieu;
public :
```

```
        virtual move();
        virtual bigger();
        virtual ncotes() = 0;
};
```

La classe `triangle` devra implémenter sa propre version de la fonction.

Selon qu'il y a un code ou non dans la classe de base, la classe de base peut soit fournir du code *par défaut*, soit indiquer une obligation de définition de code pour les enfants.

Nota Bene : toutes les formes de définition d'objets permettent l'association du polymorphisme. Regardons ce qui existe en utilisation des références :

- définition déclarative d'une référence

```
foo & r = *(foo *)new bar;
```

- paramètres de fonction (passage par référence)

```
void f(foo & r)
{
        r.g();
}

bar b;
other c;

f(b);
f(c);
f(a);
```

- retour de fonction de type référence

```
foo & r = f();

foo & f()
{
        static foo a;
        static bar b;
        static other c;
        return( (rand() % 2) ? b : (rand() % 3) ? c : a;
}
```

La même chose s'écrit avec des pointeurs.

Considérons les définitions de classes suivantes :

```
// classe de base, pour polymorphisme
class foo {
public :
        // fonction polymorphable
        virtual void f() { printf("Generique foo.f()0); };

        // fonction non polymorphable
        void g() { printf("Generique foo.g()0); };
};

// classe derivee
class bar : public foo {
public :
        void f() { printf("Derivee bar.f()0); };
        void g() { printf("Derivee bar.g()0); };
};

// autre classe derivee
```

```
class other : public foo {
public :
        void f() { printf("Derivee other.f()0); };
        void g() { printf("Derivee othern.g()0); };
};
```

On a donc deux classes `bar` et `other` qui sont des dérivées d'une même classe `foo`. Pour que le mécanisme de polymorphisme soit pris en compte, les classes dérivées doivent donner une définition propre des fonctions membres définies avec le qualificatif de `virtual` dans la classe de base.

L'arborescence de classe donnée dans cet exemple se visualise sous la forme suivante, lorsqu'on la regarde sous l'aspect de polymorphisme :

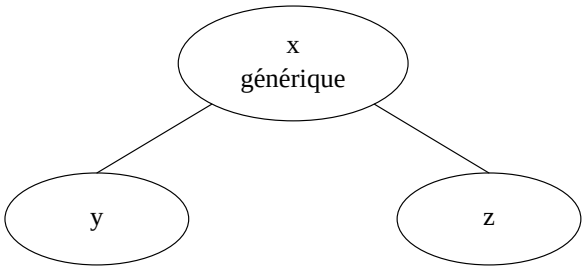


Figure 62. Arborescence par héritage

Si on se donne des instance de chacunes de ces classes, on peut activer les fonctions membres de celles-ci, directement à partir des objets alloués.

```
main()
{
        bar a;
        a.f();  // code de f() rattache a classe bar
        a.g();  // code de g() rattache a classe bar

        foo b;
        b.f();  // code de f() rattache a classe foo
        b.g();  // code de g() rattache a classe foo
}
```

L'activation de ce code a pour résultat :

```
Derivee bar.f()
Derivee bar.g()

Generique foo.f()
Generique foo.g()
```

Construisons un programme qui met en marche le mécanisme de polymorphisme. On se donne une instance de classe `bar` et de classe `other` et un pointeur sur classe `foo`. En faisant pointer le pointeur de "type" `foo` sur l'instance de classe `bar`, si on active la fonction qualifiée `virtual` à partir de ce pointeur (de type `foo`), alors c'est le code de la fonction rattachée à la classe `bar` qui est actvé. Par contre, pour les fonctions membres non qualifiées de `virtual`, c'est le code de la classe `foo` qui est activé.

```
main()
{
        // pointeur sur classe foo pour polymorphisme
        foo * p;
```

```
        // polymorphisme sur classe bar
        bar b;
        p = (foo *)(&b);
        p->f();
        p->g();

        // polymorphisme sur classe other
        other c;
        p = (foo *)(&c)
        p->f();
        p->g();
}
```

Le résultat de l'activation de ce programme illustre le polymorphisme réalisé sur les fonctions membres spécifiéss `virtual` :

```
Derivee bar.f()
Generique foo.g()
Derivee other.f()
Generique foo.g()
```

Remarque : pour que le polymorphisme puisse être résolu, l'héritage doit être publiç sinon le fils ne peut connaître les fonctionnalités de son père lors de l'overload dynamique.

Le polymorphysme peut se faire sur plusieurs niveaux; ainsi, dans l'exemple suivant, la "recherche" de code dans la descendance se fait "tant que l'on ne tombe pas sur une fonction membre non-virtuelle".

```
class foo {
public :
        virtual void pr() { printf("foo0); };
};

class bar : public foo {
public :
        virtual void pr() { printf("bar0); };
};

class foobar : public  bar {
public :
        void pr() { printf("foobar0); };
};

main()
{
        printf("Acces directs :0);

        foo * f = new foo;
        f->pr();

        bar * b = new bar;
        b->pr();

        foobar * fb = new foobar;
        fb->pr();

        printf("0cces indirects via pointeur sur base :0);

        foo * Pointer;

        Pointer = f;
        Pointer->pr();

        Pointer = b;
```

```
        Pointer->pr();

        Pointer = fb;
        Pointer->pr();
}
```

L'activation de ce programme donne le résultat suivant :

```
Acces directs :
foo
bar
foobar

Acces indirects via pointeur sur base :
foo
bar
foobar
```

7.2.3 Classes abstraites

On dit d'une classe que c'est une `classe abstraite` lorsqu'elle ne sert que de base de travail pour un mécanisme de polymorphisme "pur". Cette classe ne comporte aucune données (pas de membres données). Ses fonctions membres, toutes déclarée en `virtual`, ne proposent aucun code "par défaut".

```
class Abstraite {
public :
        virtual int foo() = 0;
};
```

De plus, on oblige les classe héritières à produire un code pour ces fonctions (affectation à `0` pour une initialisation du pointeur sur fonction utilisé par le routeur sur code du mécanisme de polymorphisme).

Remarque : L'utilisation des classes abstraites demande une connaissance parfaite de l'ensemble de l'arborescence de classes utilisée dans l'applicatif.

Une classe devient "abstraite" quand au moins l'une de ses fonctions membres est "initialisée" à zéro; i.e. quand on oblige la définition d'au moins une fonction membre par l'enfant directe de cette classe.

Du fait qu'aucun code n'est rattaché à l'une de ses fonction membre, le type d'une classe abstraite est considéré comme "incomplet. En effet, pour prendre effectivement forme, cette classe ne peut être utilisée que comme classe de base dans une arborescence. Ceci interdit donc l'instanciation d'une classe de base (on ne peut instancier qu'une de ses dérivées, si elle-même n'est pas abstraite).

Des classes abstraites peuvent être présentes sur plusieurs niveaux d'une même arborescence. La résolution du polymorphysme se fait en recherchant au travers de la totalité de l'arbre des héritages, et donc jusqu'à atteindre la classe de l'instance "pointée". On a alors soit la définition "par défaut" correspondant à cette classe, soit la recherche "plus bas dans la descendance" pour atteindre l'élément voulu.

La seule contrainte apparaissant avec les classes abstraite est l'obligation pour un descendant (direct ou d'un niveau inférieur) de définir la fonction membre.

7.3 Exercices

Exercice no. 32 :
Reprendre l'exercice sur le tableau étendu d'entiers, et donner une solution présentant le mécanisme d'héritage. Rappel: On doit répondre à l'énoncé suivant :

```
main()
{
   ExtendedArrayOfInteger a(-1000,1200);

   a[24] = 23;
   a[-200] = a[24];
   a[-999] = 100;
   a.Print("%03d");
}
```

Exercice no. 33 :
Reprendre l'exercice sur la gestion de pile d'entiers naturels, et donner un implantation utilisant le concept d'héritage. Garder le même programme de test.

8. Classes et fonctions paramétrées

Depuis longtemps il semblait nécessaire de mettre en place une implantation de types paramétrés, permettant une génération dynamique de types à partir de types déjà définis. Les outils à mettre en place recquièrent la possibilité de pouvoir rendre des classes génériques configurables suivant la demande de l'utilisateur. Bjarne Stroustup a présenté, au cours des conférences Usenix de 1988, les avancements de la recherche de solutions C++ pour ce type de problème [Stroustrup,1988] .

Basée sur ces travaux, la version 3.0 du C++ de AT&T introduit définitivement un outil permettant de générer dynamiquement des types. Un nouveau mot clé a été introduit à cet effet : `template` [66].

Depuis la version 2.2.2, le C++ de Gnu implémente aussi ce mécanisme avec le nouveau mot clé `template`.

La notion de paramétrage, implantée pour une configuration dynamique des types, a ensuite été étendue au concept de paramètres de configuration de fonctions. Le mot clé `template` s'applique donc pour les classes et pour les fonctions.

Remarque: Le concept de type dynamique était déjà en utilisation, dans les versions précédentes du C++, mais par l'utilisation d'un ensemble de macros interprêtées au niveau du préprocesseur (`cpp`).

8.1 Paramétrage par utilisation de macros

Avant la version 3.0 du C++ de AT&T (ou la version 2.2.2 de `gcc`), différentes subtilités ont été appliquées pour obtenir la génération dynamique de types. Sans modifications des compilateurs, l'utilisation approfondie du préprocesseur permet, par un jeu de macros, d'obtenir ce résultat.

L'une des premières utilisations des types dynamiques a été appliqué au concept de tableau. On voudrait créer un type générique *tableau*, et faire en sorte que ce type puisse automatiquement se commuer en *tableau d'entiers*, ou *tableau de classe foo*, dynamiquement, sur simple spécification d'environnement.

Ce chapitre présente la construction d'un tel type générique, qui, par un jeu de macros, va permettre de construire un nouveau type.

Pour créer le nom de notre nouveau type, généré dynamiquement, on joue sur le fait que l'opérateur ## du préprocesseur permet de réaliser la concaténation des paramètres :

```
#define Foo(Bar) Foo##Bar
```

génère, par concaténation, le mot

```
FooBar
```

Considérons, comme exemple, la création du type *tableau d'entier*. On peut considérer ce type comme un type composé du type *tableau* et du type *entier*. Le type *tableau* sait gérer des élements sous la forme d'un tableau, sans présupposé sur le type de ses éléments. C'est le type "générique". Le type *entier*, lui, correspond aux entiers machine (`int`). Il sert à construire le type "définitif". Il sera le "paramètre" de configuration du type générique.

On commence par se définir le type *tableau*. C'est une classe *générique* de gestion d'éléments, qui sera utilisée pour la génération des types définitifs.

```
// definition de la classe generique de gestion de tableau
class Array {
        int Length;
        int Size;
        void * Memory;
```

```
public :
        Array(int l, int s) {
                Length = l;
                Size = s;
                Memory = (void *) new char [ s * l];
        };
        ~Array() {
                delete Memory;
        };
        void * Address(int i) {
                if( i < 0 || i >= Length )
                {
                        Error("out of range");
                }
                return( Memory + (i * Size) );
        };
};
```

Elle intègre toutes les fonctionnalités attendues pour la gestion d'un tableau. A priori, elle n'est pas faite pour être utilisée directement. On doit, dans l'esprit qui est développé ici, lui fournir le type de "recouvrement" pour générer le type définitif. Elle n'a aucune consistence en elle-même.

On se dote maintenant d'une macro qui permettra de créer le nom du type définitif.

```
// definition de construction du nom du nouveau type
#define ArrayOf(Type) ArrayOf##Type
```

Puis on donne le mécanisme, toujours sous forme de macro, pour la construction "dynamique" du type. Le nouveau type est construit à la compilation (lors de la phase de pré-processing).

```
// definition de construction de nouveau type
#define BuildArrayOf(Type)      class ArrayOf(Type) : Array {    public :
```

Cet outil est alors utilisé pour la création du type définitif :

```
// Generation du nouveau type : tableau d'entiers
BuildArrayOf(int);
```

Le nouveau type que l'on vient de créer s'appelle `ArrayOf(int)`. On peut ensuite, dans le programme, déclarer des objets de ce nouveau type. Par exemple, ici, un tableau de 200 entiers.

```
// declaration d'un objet de type tableau d'entiers
ArrayOf(int) t(200);
```

Voici ce à quoi pourrait ressembler un programme utilisant ce nouveau type :

```
main()
{
        ArrayOf(int) t(200);
        t[10] = 10;
        t[5] = t[10];
}
```

Les outils rattachés au type `Array` (tableau de base) permettent une utilisation transparente de l'opérateur de tableau `[ ]`.

A partir du moment où le nouveau type a été construit, on peut déclarer et utiliser des objets de ce type dans le programme.

Supposons que la définition du type `Array`, et que les macros de construction dynamique de types `ArrayOf()` et `BuildArrayOf()` aient été mises dans le fichier d'include `DynamicArray.hh`. On peut créer le programme suivant :

```
// definitions de tableaux dynamiques
#include <DynamicArray.hh>
```

```
// affichage de valeurs (creation dynamique suivant le type voulu)
#define Print(Format,Type,Data) printf(Format,(##Type)Data)

// Generation du nouveau type : tableau d'entiers
BuildArrayOf(int);

// Generation du nouveau type : tableau de doubles
BuildArrayOf(double);

main()
{
        // declaration d'un objet de type tableau d'entiers
        ArrayOf(int) t(200);
        t[10] = 10;
        t[5] = t[10];
        Print("%d0,int,t[5]);

        // declaration d'un objet de type tableau de doubles
        ArrayOf(double) d(200);
        d[10] = 99.88;
        d[5] = d[10];
        Print("%f0,double,d[5]);
}
```

Ce programme manipule indifféremment des tableaux d'entiers et de flottants doubles précision, par construction dynamique des types se basant sur un objet générique de gestion de tableau.

Le type "définitif" est spécifié par

```
ArrayOf(int)
ArrayOf(double)
```

respectivement pour un tableau d'entiers ou un tableau de doubles. Il ne faut bien sûr pas oublier de "construire" ces types par utilisation de la macro `BuildArrayOf()` appliquée aux types `int` et `double`.

Remarque : L'invocation de cette macro, pour chacun des types définitifs voulus, ne doit apparaître qu'une seule fois dans le programme, sinon on se retrouve avec un message de définition multiple de la part du préprocesseur.

8.2 Utilisation du mot clé template

Les templates implémentent un mécanisme de paramétrage d'environnement. A la base, ils sont utilisés pour appliquer le concept de types paramétrés. Ils s'appliquent aux fonctions et aux classes.

Une *classe template* est une définition de classe particulière, générique, qui prendra sa réelle représentation lorsqu'on l'aura "initialisée" par un type préalablement défini. Cette initialisation se fait lors de l'instanciation de la classe.

Une *fonction template* est une fonction générique qui pourra manipuler indifféremment les types qu'on lui fournira lors de sa déclaration. Le code de la fonction correspondant aux types des paramètres d'appel est généré lors du premier appel de cette fonction pour ces types.

La définition d'un template se fait par utilisation du nouveau mot clé `template`. Le compilateur C++ 3.0 de AT&T a été le premier à l'implémenter. Les outils **Gnu** l'intègrent en partie avec la version 2.2.2 du compilateur `gcc` et totalement avec la version 2.3.3.

8.2.1 Définition de classe template (classe paramétrée)

Les templates sont utilisés pour définir des classes génériques récipiendaires, sans qu'un type définitif ne leur soit rattaché. Ce n'est que l'utilisateur final qui déterminera le type final de l'objet manipulé par l'application lors de l'instanciation de la classe.

Une classe générique suit les règles de définition et de déclaration suivantes :

```
template <class Type> class Generique {
        // membres prives de classe generique
        Type ...
public :
        // membres publics de classe generique
        Type ...
};

Generique <TypeVoulu> foo;
```

On définit une classe générique tout comme on définit une simple classe, mais on la qualifie de `template` et on spécifie entre '`< >`' les types "indéterminés" qui seront utilisés en utilisation finale (paramètres à fournir lors de la déclaration d'un objet définitif). Ces types, lors de la définition de la classe générique, doivent impérativement être précédés du mot clé `class`, même si, lors de la création du type définitif, il s'agit de types de base.

L'utilisation d'un objet répondant à cette classe générique se fait en fournissant les paramètres attendus par la définition de cette dernière (i.e. le nom du type "définitif").

Figure 63. Template

Remarque: Le type définitif n'est connu que lorsqu'on instancie la classe. Il faut donc compiler le source contenant cette déclaration avec tous les sources servant à définir le type générique.

Dans l'exemple donné ci-dessous, on spécifie, à la définition, que la classe générique recevra un type class en paramètre. Ce type est alors utilisable dans le scope de définition de la classe générique. On peut spécifier des données de ce type, ou définir des fonctions membres qui utilisent ce type.

```
template <class TypeParam> class Generique {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam x) {
                gb = new TypeParam [100];
                ga = x;
        };
        ~Generique() {
                delete [] gb;
        };
        TypeParam Get() {
                return ga;
        };
};

main()
{
        // type definitif : entier
        Generique <int> foo(10);
        int i = foo.Get();
        printf("%d0,i);

        // type definitif : pointeur sur flottant double precision
```

```
        double d = 99.88;
        Generique <double *> bar(&d);
        double *dd = bar.Get();
        printf("%f0,*dd);
}
```

Lors de l'utilisation de cette classe générique, pour la déclaration d'un objet, on fournit le type définitif à rattacher en place et lieu du type paramètre.

Ce type "définitif" doit être un type connu lors de son utilisation. On a ici déclaré deux objets qui ont les mêmes comportements, mais pour lesquels la classe générique s'est automatiquement cadrée pour le traitement d'entiers machine, ou de pointeurs sur flottants double précision.

Le résultat de l'activation de ce programme est le suivant :

```
10
99.880000
```

Le type "définitif" peut être une classe définie antérieurement. Dotons-nous de la définition d'un *entier borné*; on peut utiliser un objet de type classe `Generique` mappée sur un `EntierBorne`.

```
template <class TypeParam> class Generique {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam x) {
                gb = new TypeParam [100];
                ga = x;
        };
        ~Generique() {
                delete [] gb;
        };
        TypeParam Get() {
                return ga;
        };
};

class EntierBorne {
        int inf;
        int sup;
        int val;
public :
        EntierBorne(int v = 0, int i = 0, int s = 999) {
                inf = i;
                sup = s;
                if( v < inf || v > sup )
                {
                        perror("Hors Bornes");
                }
                val = v;
        };
        int Get() { return val; };
};

main()
{
        Generique <EntierBorne> foo(99);
        EntierBorne i = foo.Get();
        printf("%d0,i.Get());
}
```

L'applicatif ne "connaît" que les `EntierBorne`s comme type connu. Mais leur implantation se fait réellement par la classe `Generique`. Tous les objets qui seront déclarés par utilisation de cette classe générique auront le même comportement.

 ISBN 2-909909-02-6

8.2.2 Définition de fonction membre de template

Voyons comment on peut définir les fonctions membres de notre classe `template` hors du scope de définition de la classe. Il faut rattacher les fonctions à la classe. On utilise l'opérateur de domaine `::`. Mais il faut aussi spécifier que cette classe est qualifiée de `template`. On doit donc suivre la règle de paramétrisation, en qualifiant aussi la fonction membre de `template` et en spécifiant les paramètres nécessaires.

```
templace <class TypeParam> class Generique {
        // fonction membre
        TypeFonction Fonction();
};

template <class TypeParam> TypeFonction Generique <TypeParam> :: Fonction()
{
        // code fonction
        // return d'un objet de type TypeFonction
}
```

On peut l'écrire de façon plus lisible sur plusieurs lignes :

```
template <class TypeParam>
TypeFonction Generique <TypeParam> :: Fonction()
{
        // code fonction
        // return d'un objet de type TypeFonction
}
```

Il faut donc qualifier la fonction de `template <class TypeParam>`, mais il faut aussi rappeler que la classe `Generique` à laquelle est rattachée la fonction dépend d'un paramétrage de type : `Generique <TypeParam>`. Ceci ne facilite pas la lecture...

On peut, de plus, qualifier la fonction membre ainsi définie de `inline` pour continuer de bénéficier de l'optimisation de code :

```
template <class TypeParam>
inline TypeFonction Generique <TypeParam> :: Fonction()
{
        // code fonction
        // return d'un objet de type TypeFonction
}
```

Plus ça va, plus l'écriture multi-ligne devient nécessaire pour la lecture du programme...

Si on reprend notre exemple, avec une définition hors classe des fonctions membres de la classe générique, on obtient le code suivant :

```
template <class TypeParam> class Generique {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam x)
        {
                gb = new TypeParam [100];
                ga = x;
        };

        ~Generique() {
                delete [] gb;
        };

        TypeParam Get();
};
```

```
// definition de la fonction Get() qui retourne un TypeParam
template <class TypeParam>
TypeParam Generique <TypeParam> :: Get()
{
        return ga;
}

main()
{
        Generique <int> foo(99);
        int i = foo.Get();
        printf("%d0,i);
}
```

Par qualification de `inline` supplémentaire, on obtient le code suivant :

```
template <class TypeParam> class Generique {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam x)
        {
                gb = new TypeParam [100];
                ga = x;
        };

        ~Generique() {
                delete [] gb;
        };

        TypeParam Get();
};


// definition de la fonction Get() qui retourne un TypeParam
template <class TypeParam>
inline TypeParam Generique <TypeParam> :: Get()
{
        return ga;
}

main()
{
        Generique <int> foo(99);
        int i = foo.Get();
        printf("%d0,i);
}
```

Remarque: dans la version 2.3.3 de `gcc`, il y a possibilité de mettre les définitions des constructeurs et destructeurs de classes paramétrées en dehors de la définition de la classe elle-même. En cas de constructeurs simple, ne faisant pas appel à l'allocation dynamique, la fonction peut rester une vraie fonction. Si on doit faire des choses plus compliquées, on doit alors qualifier la fonction d'`inline`. Le destructeur, lui, doit systématiquement être qualifié de `inline` (sinon, on obtient une erreur de compilateur). La version gcc-2.2.2 était moins stricte à ce sujet.

```
template <class TypeParam> class Generique {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam,int);
```

```
        ~Generique();
        TypeParam Get();
};

template <class TypeParam>
inline
Generique <TypeParam> :: Generique (TypeParam x, int y)
{
        gb = new TypeParam [ y ];
        ga = x;
}

template <class TypeParam>
inline
Generique <TypeParam> :: ~Generique ()
{
        delete [] gb;
}

template <class TypeParam>
inline
TypeParam  Generique <TypeParam> :: Get()
{
        return ga;
}

main()
{
        Generique <int> foo(99, 200);
        int i = foo.Get();
        printf("%d0,i);
}
```

8.2.3 Classe à paramètres multiples

Une classe peut recevoir en paramètre plusieurs types qui permettront de la définir totalement. Par exemple, définissons la classe `Generique` comme dépendante des deux types `TypeParam` et `AutreType` à fournir en paramètres :

```
template <class TypeParam, class AutreType> class Generique {
        TypeParam ga;
        AutreType gb;
public :
        Generique(TypeParam, AutreType);
        TypeParam GetGa();
        AutreType GetGb();
};

template <class TypeParam, class AutreType>
Generique <TypeParam,AutreType>
        :: Generique (TypeParam x, AutreType y)
{
        ga = x;
        gb =y;
}

template <class TypeParam, class AutreType>
TypeParam  Generique <TypeParam,AutreType> :: GetGa()
{
        return ga;
}
```

```
template <class TypeParam, class AutreType>
AutreType  Generique <TypeParam,AutreType> :: GetGb()
{
        return gb;
}

main()
{
        Generique <int,double> foo(-4,99.88);
        int i = foo.GetGa();
        double d = foo.GetGb();
        printf("%d %f0,i,d);
}
```

Remarque: Lors de la définition des fonctions membres, il faut rappeler la totalité de la dépendance des paramètres, pour la classe.

On peut alors générer une instance de la classe en fournissant des "valeurs" aux paramètres attendus.

8.2.4 Classe template membre de classe

Le paramètre fourni à une classe peut être utilisé pour construire un membre de cette classe dont le type serait paramétré.

Considérons une première définition de classe paramétrée (`Cible`). Puis donnons-nous la définition d'une classe dont l'un des membre est du type défini en premier (`Generique`). Cette seconde définition devient rétroactivement générique, car elle maintient un membre lui-même de type générique.

L'exemple donné ici montre comment définir un tel membre et comment ensuite définir les constructeurs et fonctions qui manipulent ces informations.

```
// Definition d'une classe parametree
template < class Type> class Cible
{
 private:
  Type Data;

 public :
   Cible();
   Cible(Type &);
   Cible(Cible<Type> &);
   ~Cible();

  void SetData(Type &);
  Type & GetData();
};

template <class Type>
inline
Cible<Type>::Cible() : Data(0)
{
  ;
}

template <class Type>
inline
Cible<Type>::Cible(Type & Param) : Data(Param)
{
  ;
}

template <class Type>
```

```
inline
Cible<Type>::Cible(Cible<Type>& Param) : Data(Param.Data)
{
  ;
}

template <class Type>
inline
Cible<Type>::~Cible()
{
  ;
}
template <class Type>
inline
void Cible<Type>::SetData(Type & Param)
{
  Data = Param;
}

template <class Type>
inline
Type & Cible<Type>::GetData()
{
  return Data;
}


// Definition d'une autre classe parametree
// utilisant la premiere en type de membre
template <class Type> class Generique
{
private :
  Cible<Type> DataGenerique;

public:
  Generique();
  Generique(Type &);
  Generique(Cible<Type> & );
  ~Generique() { ; };

  Cible<Type> & SetDataGenerique(Cible<Type> &);
  Cible<Type> & GetDataGeneriqueReference();
  Cible<Type> * GetDataGeneriquePointer();

//  Generique<Type> & GetMyself();
};

template <class Type>
inline
Generique<Type>::Generique() : DataGenerique(0)
{
  ;
}

template <class Type>
inline
Generique<Type>::Generique(Type & Param) : DataGenerique(Param)
{
//  DataGenerique.SetData(Param);
  ;
}

template <class Type>
inline
```

```
Generique<Type>::Generique(Cible<Type> & Param) : DataGenerique(Param)
{
//  DataGenerique.SetData(Param.GetData());
  ;
}

template <class Type>
inline
Cible<Type> & Generique<Type>::SetDataGenerique(Cible<Type> & Param)
{
  DataGenerique = Param;
  return DataGenerique;
}

template <class Type>
inline
Cible<Type> & Generique<Type>::GetDataGeneriqueReference()
{
  return DataGenerique;
}

template <class Type>
inline
Cible<Type> * Generique<Type>::GetDataGeneriquePointer()
{
  return &DataGenerique;
}

/*
template <class Type>
inline
Generique<Type> & Generique<Type>::GetMyself()
{
  return (Generique<Type>)(*this);
}
*/
main()
{
  Cible<int> Entier(8);
  printf("Entier.Data = %d0,Entier.GetData());

  Generique<int> EntierGenerique((int)999);
  printf("EntierGenerique.Data = %d0,
         (EntierGenerique.GetDataGeneriqueReference()).GetData());

  (EntierGenerique.GetDataGeneriqueReference()).SetData(888);
  printf("EntierGenerique.Data = %d0,
         (EntierGenerique.GetDataGeneriqueReference()).GetData());

  Entier.SetData(777);
  printf("Entier.Data = %d0,Entier.GetData());

  EntierGenerique.SetDataGenerique(Entier);
  printf("EntierGenerique.Data = %d0,
         (EntierGenerique.GetDataGeneriqueReference()).GetData());
}
```

L’activation de ce programme donne les résultats suivants :

```
Entier.Data = 8
EntierGenerique.Data = 999
EntierGenerique.Data = 888
Entier.Data = 777
EntierGenerique.Data = 777
```

On a bien une propagation correcte des informations au travers des paramétrages respectifs.

8.2.5 Classe template à héritage

Une classe générique est une classe comme une autre. Rien n'interdit qu'elle hérite d'une autre classe. Considérons que notre classe `Generique` hérite en construction d'une classe `EntierBorne`. On a alors la définition suivante :

```
class EntierBorne {
        int inf;
        int sup;
        int val;
public :
        EntierBorne(int v = 0, int i = 0, int s = 999) {
                inf = i;
                sup = s;
                if( v < inf || v > sup ) {
                        perror("Hors Bornes");
                }
                val = v;
        };
        int Get() { return val; };
};

template <class TypeParam>
class Generique : public EntierBorne {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam,int);
        ~Generique();
        TypeParam Get();
};

template <class TypeParam>
inline
Generique <TypeParam> :: Generique (TypeParam x, int y)
        : EntierBorne(y)
{
        gb = new TypeParam [ EntierBorne::Get() ];
        ga = x;
}

template <class TypeParam>
inline
Generique <TypeParam> :: ~Generique ()
{
        delete [] gb;
}

template <class TypeParam>
TypeParam  Generique <TypeParam> :: Get()
{
        return ga;
}

main()
{
        Generique <int> foo(99, 200);
        int i = foo.Get();
        int j = foo.EntierBorne::Get();
        printf("%d %d0,i,j);
```

```
        Generique <int> bar(99, 1000);   // Error
}
```

A noter que le mécanisme d'héritage reste le même, avec le même type de qualification (`private`, `public`, `protected`, `virtual`).

L'héritage de la classe générique peut être multiple (plusieurs classes de base). Modifions notre classe `Generique` et faisons-la hériter d'un `EntierBorne` et d'un `Point` (que l'on se prédéfinit). On obtient la définition suivante :

```
class EntierBorne {
        int inf;
        int sup;
        int val;
public :
        EntierBorne(int v = 0, int i = 0, int s = 999) {
                inf = i;
                sup = s;
                if( v < inf || v > sup ) {
                        perror("Hors Bornes");
                }
                val = v;
        };
        int Get() { return val; };
};

class Point {
        int abs;
        int ord;
public :
        Point(int x, int y) {
                abs = x;
                ord = y;
        };
        int GetAbs() { return abs; };
        int GetOrd() { return ord; };
};

template <class TypeParam>
class Generique : public EntierBorne , private Point {
        TypeParam ga;
        TypeParam *gb;
public :
        Generique(TypeParam,int);
        ~Generique();
        TypeParam Get();
};

template <class TypeParam>
inline
Generique <TypeParam> :: Generique (TypeParam x, int y)
        : EntierBorne(y,0,999), Point(y,y)
{
        gb = new TypeParam [ EntierBorne::Get() ];
        ga = x;
}

template <class TypeParam>
inline Generique <TypeParam> :: ~Generique ()
{
        delete [] gb;
}

template <class TypeParam>
```

```
inline TypeParam  Generique <TypeParam> :: Get()
{
        return ga;
}

main()
{
        Generique <int> foo(99, 200);
        int i = foo.Get();
        int j = foo.EntierBorne::Get();
        printf("%d %d0,i,j);
}
```

Dans cet exemple, la classe `EntierBorne` est "accessible" par le programme car elle est du domaine publiç mais la classe `Point` est interne à la classe `Generique` car déclarée en `private`.

8.2.6 Héritage de templates avec transmission de paramètres

Considérons une classe de base générique, qualifiée de `template`, et configurable par un paramètre de type.

Construisons ensuite une autre classe, dérivée de cette classe template. Si on veut que cette classe dérivée transmette elle-même le paramétrage de sa classe de base, on doit aussi la qualifier de template, car elle sert de routage d'informations paramétrées vers l'une de ses classes de base.

```
// classe parametree
template <class TypeParam> class Generique {
        TypeParam ga;
public :
        Generique(TypeParam);
        TypeParam Get();
};

template <class TypeParam>
inline
Generique <TypeParam> :: Generique (TypeParam x)
{
        ga = x;
}

template <class TypeParam>
TypeParam Generique<TypeParam> :: Get()
{
        return ga;
}

// classe derivee routeur de parametre
template <class TypeParam>
class Derivee : public Generique <TypeParam> {
        int da;
public :
        Derivee(int x, TypeParam y) : Generique <TypeParam> (y) {
                da = x;
        };
        int Get() {
                return da;
        };
};

main()
{
        // classe parametree
```

```
        Generique <int> foo(99);
        int i = foo.Get();
        printf("%d0,i);

        // classe derivee fournissant le parametre
        Derivee <int> bar(11, 333);
        int j = bar.Get();
        printf("%d0,j);

        int k = bar.Generique<int>::Get();
        printf("%d0,k);
}
```

Si on se donne une instance de la classe de base (`foo`), on spécifie le type en paramètre. Pour une instance de la classe dérivée (`bar`), on doit aussi spécifier le type paramètre qui sert à construire sa classe de base.

On peut toujours, vu que l'on a une dérivation qualifiée de `public`, appeler une fonction de la classe de base à partir d'une instance de la classe dérivée. Mais il faut alors rappeler le paramètre utilisé par la construction de la classe de base lors de l'instanciation de la classe dérivée.

```
Derivee.Base<Type>::BaseFonction()
```

8.2.7 Héritage de template sans transmission de paramètres

On veut qu'une classe dérivée puisse hériter d'une classe de base qui est définie en template. Mais cette fois-çi, on ne veut pas que la classe dérivee soit une classe template (i.e. elle ne sert pas de routeur de paramaètres). Dans ce cas, elle ne peut hériter de la classe de base que si, et seulement si, elle fixe d'elle-même le paramètre type de la classe de base.

Reprenons notre classe de base template, comme définie plus haut, et donnons-nous une classe dérivée de cette classe de base. La classe dérivée n'est pas une classe template; on doit fixer le paramètre de la classe de base. Choisissons par exemple que la classe dérivée hérite de la classe de base avec le type "définitif" `double`. La définition de la classe dérivée devient :

```
// classe paramatree
template <class TypeParam> class Generique {
        TypeParam ga;
public :
        Generique(TypeParam);
        TypeParam Get();
};

template <class TypeParam>
Generique <TypeParam> :: Generique (TypeParam x)
{
        ga = x;
}

template <class TypeParam>
TypeParam Generique<TypeParam> :: Get()
{
        return ga;
}

// classe derivee fixant le parametre
class Derivee : public Generique<double> {
        int da;
public :
        Derivee(int,double);
```

```
        int Get() {
                return da;
        };
};

Derivee::Derivee(int x,double y ) : Generique<double> (y)
{
        da = x;
};

main()
{
        // classe parametree
        Generique <int> foo(99);
        int i = foo.Get();
        printf("%d0,i);

        // classe derivee fixant les parametres
        Derivee bar(11, 333);
        int j = bar.Get();
        printf("%d0,j);

        double k = bar.Generique<double>::Get();
        printf("%f0,k);
}
```

On peut alors utiliser la classe dérivée dans un programme en fournissant les valeurs nécessaires à la construction d'une instance conformément à la définition fournie.

8.2.8 Hiérarchie de classes à templates

Donnons nous une première classe `Generique` paramétrée par le type `TypeParam`.

Donnons-nous ensuite une seconde classe, elle aussi paramétrée, mais cette fois-ci par le type `TypeDerive`. Et décidons que cette seconde classe dérive de la première. Lors de la définition de la classe dérivée, on doit fournir toutes les informations concernant les paramètres, à savoir ceux qui concernent la classe dérivée proprement dite, et ceux qui concernent sa classe de base.

```
// classe parametree (base)
template <class TypeParam> class Generique {
        TypeParam ga;
public :
        Generique(TypeParam);
        TypeParam Get();
};

template <class TypeParam>
Generique <TypeParam> :: Generique (TypeParam x)
{
        ga = x;
}

template <class TypeParam>
TypeParam  Generique <TypeParam> :: Get()
{
        return ga;
}

// classe parametree (derivee)
template <class TypeDerive, class TypeParam>
class Derivee : public Generique <TypeParam> {
        TypeDerive da;
```

```
public :
        Derivee(TypeDerive x,TypeParam y):Generique<TypeParam>(y)
        {
                da = x;
        };
        TypeDerive Get() {
                return da;
        };
};

main()
{
        Generique <int> foo(-4);
        int i = foo.Get();
        printf("%d0,i);

        Derivee <double,int> bar(99.88, 333);
        double d = bar.Get();
        printf("%f0,d);

        int k = bar.Generique<int>::Get();
        printf("%d0,k);
}
```

Lorsqu'on crée une instance de la classe dérivée, il faut renseigner les paramètres concernant la classe dérivée (ici le type `double`) et ceux concernant sa classe de base (ici `int`).

Rappel: Les définitions des paramètres utilisent le mot clé `class` dans la syntaxe de `template`, même si les types définitifs ne seront que des types de base (et non pas des classes).

Attention : L'ordre de déclaration des paramètres dans les instructions de déclaration des objets est dépendant de celui de leur définition. Si on inverse la liste des paramètres dans l'instanciation de la classe dérivée, on obtient des types "définitifs" inversés par rapport à la programmation montrée ci-dessus. Les instructions de récupération des valeurs, et d'affichage de ces dernières, ne sont alors plus en conformité.

8.2.9 Dérivation de type paramétré

On a vu qu'une classe pouvait dériver d'une classe paramétrée qui prendra un type définitif lors de l'instanciation de la classe dérivée.

On peut aussi faire dériver une classe d'une autre classe dont on ne précisera le type que lors de l'instanciation de la classe dérivée. On dérive d'un template :

```
template <class TypeParam>
class Derivee : public TypeParam {
        ...
public :
        Derivee() : TypeParam () {
                ...
        };
        ...
};
```

La résolution de l'héritage ne sera effective que lors de l'instanciation, avec seulement alors la prise en compte du type de base spécifié en paramètre.

Remarque: Dans ce cas de figure, le paramètre doit impérativement être de type `class`. On ne peut pas faire dériver une classe d'un type de base.

Prenons, par exemple, l'instanciation d'une classe dérivant d'un paramètre appliqué à une classe de

base dont le type est totalement défini :

```
// Classe de base pour parametre
class Base {
        int ba;
public :
        Base(int x){
          ba = x;
        };
        int Get(){
          return ba;
        };
        void Print() {
          printf("Base::ba = %d0,ba);
        };
        void BasePrint() {
          printf("BasePrint::ba = %d0,ba);
        };
};

// classe derivee d'une classe passee en parametre
template <class ClassParam>
class Derivee : public ClassParam {
        int da;
public :
        Derivee(int x, int y) : ClassParam (y) {
                da = x;
        };
        int Get() {
                return da;
        };
        void Print() {
                printf("Derivee::da = %d0,da);
                ClassParam::Print();
        };
};


main()
{
        Base foo(99);
        int i = foo.Get();
        printf("%d0,i);

        Derivee <Base> bar(11, 333);
        int j = bar.Get();
        printf("%d0,j);

        int k = bar.Base::Get();
        printf("%d0,k);

        printf("Calling print() functions :0);
        foo.Print();
        bar.Print();
        bar.BasePrint();

        // Following is an error <int> is not a class
        // Derivee <int> Other(100,999);
        // Other.print();
}
```

Dans notre exemple, l'instance `bar` de la classe `Derivee` hérite de la classe `Base`, classe totalement définie. Lors de la déclaration des objet, on précise le paramètre, ainsi que les valeurs nécessaires à l'instanciation de chacune des deux classes concernées.

Remarque : on ne peut, dans ce cas de figure, donner en paramètre un type de base (`int`, `char`, ou un pointeur) car on a ici une demande d'héritage. On doit rester conforme à ce concept, et donc fournir un paramètre de type classe.

8.2.10 Héritage multiple de paramètres classe

On peut appliquer le concept d'héritage de paramètre au concept d'héritage multiple. Donnons-nous la définition de deux classes "de base", ainsi que la définiton d'une classe à héritage multiple paramétré.

```
// Classes de base pour parametre

class PremiereBase {
        int ba;
public :
        PremiereBase(int x){
          ba = x;
        };
        void Print() {
          printf("PremiereBase::ba = %d0,ba);
        };
};

class SecondeBase {
        int ba;
public :
        SecondeBase(int x){
          ba = x;
        };
        void Print() {
          printf("SecondeBase::ba = %d0,ba);
        };
};

// classe derivee de classes passees en parametres

template <class PremierParam, class SecondParam>
class Derivee : public PremierParam, SecondParam {
        int da;
public :
        Derivee(int x, int y, int z)
          : PremierParam (y), SecondParam(z) {
                da = x;
        };
        void Print() {
                printf("Derivee::da = %d0,da);
                PremierParam::Print();
                SecondParam::Print();
        };
};


main()
{
        Derivee <PremiereBase,SecondeBase> foo(666,777,888);
        foo.Print();
}
```

Le mécanisme d'héritage multiple est préservé. On a bien l'intégration totale des deux classes passées en définition de l'instance de classe `Derivee`.

8.2.11 Héritage de paramètres type et classe

On peut combiner le paramétrage en tant que type de membre de classe et en tant que type de classe de base dont on hérite. Dans l'exemple ci-dessous, la classe dérivée hérite d'une classe de base définie à l'instanciation; elle construit aussi un membre dont le type est paramétré pour cette instance.

```
// classe parametree pouvant servir de base
class Generique {
        int ga;
public :
        Generique(int x) {
          ga = x;
        };
        void Print() { printf("Generique: %d0,ga); };
};

// classe derivee d'un parametre classe de base
// et d'un parametre donnee
template <class ClasseParam, class TypeParam>
class Derivee : public ClasseParam {
        TypeParam da;
public :
        Derivee(TypeParam x, int y) : ClasseParam (y) {
                da = x;
        };
        void Print() { printf("Derivee: %d0,da);
                ClasseParam::Print();
        };
};

main()
{
        Derivee <Generique,int> foo(222, 333);
        foo.Print();

        // Error: <int> is not a class type
        // Derivee <int, Generique> bar(222, 333);
}
```

Les deux types fournis en paramètres sont appliqués lors de la construction de l'instance. Ces types peuvent donc varier d'une instance à l'autre.

Remarque : faire bien attention à l'ordre de définition des paramètres fournis lors de l'instanciation de la classe dérivée. Ici, le premier paramètre doit représenter une classe, car il correspond à la classe de base; le second peut être un type de base, car il correspond au type d'un membre de la classe dérivée.

8.2.12 Paramètre classe paramétrée

L'utilisation des `template` permet de transmettre en paramètre de classe une classe elle-même paramétrée.

```
// classe simple
class Simple
{
private:
  int Donnee;

public :
  Simple() {
    Donnee = 0;
  };
```

```
  Simple(int x) {
    Donnee = x;
  };

  ~Simple() {
    ;
  };

  int GetDonnee() {
    return Donnee;
  };
};

// classe de base parametree
template <class Type> class Base
{
private:
   Type SubClass;

public:
   Base() {
     ;
   };

   Base(Type & Elem) {
     SubClass = Elem;
   };

   ~Base() {
     ;
   };

   Type & GetSubClass() {
     return SubClass;
   };
};

// autre classe parametree
template <class TypeParam> class Autre
{
private:
  TypeParam Data;

public:
  Autre() : Data() {
    ;
  };

  ~Autre() {
    ;
  };

  TypeParam & GetData() {
    return Data;
  };

  void SetData(TypeParam x) {
    Data = x;
  };
};

// classe derivee de classe de base parametree
template <class ClassType> class Derivee : public Base<ClassType>
{
```

```
public:
   Derivee(ClassType& Param) : Base<ClassType>(Param) {
     ;
   };
};


main()
{
  // objet simple
  Simple simple(99);
  printf("simple.Donnee = %d0,simple.GetDonnee());

  Autre<Simple> autre;
  autre.SetData(simple);
  printf("autre.Data.Donnee = %d0,(autre.GetData()).GetDonnee());

  Base< Autre<Simple> > base(autre);
  printf("base.SubClass.Data.Donnee = %d0,
         ((base.GetSubClass()).GetData()).GetDonnee());

  Derivee< Autre<Simple> > derivee(autre);
  printf("derivee.base.SubClass.Data.Donnee = %d0,
         ((derivee.GetSubClass()).GetData()).GetDonnee());

}
```

L'activation de ce programme donne le résultat suivant :

```
simple.Donnee = 99
autre.Data.Donnee = 99
base.SubClass.Data.Donnee = 99
derivee.base.SubClass.Data.Donnee = 99
```

Dans cet exemple, la classe générique (`Base`) reçoit en paramètre une classe qui est elle-même une classe paramétrée (`Autre`).

On a poussé les choses un peu plus loin en définissant une classe dérivée de cette classe générique (`Derivee`). Cette classe dérivée devient aussi générique.

Remarque: En utilisation du compilateur `gcc-2.3.3`, vu que l'on a des définitions imbriquées des types définitifs lors de la demande d'instanciation de la classe paramétrée à double niveau, il faut impérativement laisser des espaces entre les signes '`>`', sinon le compilateur cherche à résoudre l'opérateur '`>>`', et génère une erreur de parsing. Ceci sera peut-être résolu dans les prochaines versions de `gcc`.

8.2.13 Paramètre classe de base paramétrée

Il serait intéressant de pouvoir passer en paramètre à une classe dérivée un paramétrage "total" sur la classe de base dont elle dépend :

```
// classe parametree pouvant servire de base
template <class TypeParam> class Generique {
        TypeParam ga;
public :
        Generique(TypeParam x) {
          ga = x;
        };
        void Print() { printf("Generique: %d0,ga); };
};

// classe derivee d'un parametre classe de base parametree
```

```
template <class ClasseParam <class TransmitParam> >
class Derivee : public ClasseParam <TransmitParam> {
        int da;
public :
        Derivee(int x, TransmitParam y) : ClasseParam<TransmitParam> (y) {
                da = x;
        };
        void Print() { printf("Derivee: %d0,da);
                ClasseParam<TransmitParam>::Print();
        };
};


main()
{
        Derivee < Generique<int> > bar(222, (int)333);
        bar.Print();
}
```

Ceci ne marche pas encore avec la version `gcc-2.3.3`. On tombe sur le message d'erreur suivant :

```
12:sorry, not implemented: aggregate template parameter types
12:parse error before '<'
19:syntax error before '{'
20:'TransmitParam' undeclared, outside of functions
20:Internal compiler error.
20:Please report this to 'bug-g++ prep.ai.mit.edu'.
```

Est-ce pousser les choses un peu trop loin ? On aimerait pourtant le faire !

8.3 Paramètres (template) autres que de types classe

La liste des paramètres d'une classe n'est pas limitée par des qualifications de types classe. On peut lui passer des chaîne de caractères, des expressions constantes, ou des noms de fonction.

Ce qui suit est valable pour les versions 3.2 de AT&T. Se reporter aux notes locales quand aux possibilités du compilateur **Gnu**. Les essais concernant ce dernier ont été réalisés avec la version 2.3.3 de `gcc`.

8.3.1 Paramètre constante entière

L'utilisation d'un paramètre de type constante entière permet (enfin) de construire des tableaux dynamiques tels qu'**Algol** les prévoyait.

```
template <int TypeParam> class Generique {
        int Array[TypeParam];
public :
        Generique() {
                for( int i = 0; i < TypeParam; i++ )
                {
                        Array[i] = i;
                }
        };
        void Print() {
                for( int i = 0; i < TypeParam; i++ )
                {
                        printf("Array[i] = %d0,Array[i]);
                }
        };
};
```

```
main()
{
  Generique <10> foo;
  foo.Print();
};
```

Ceci permet d'avoir des tableaux répondant réellement aux spécifications d'**Algol**, car lorsqu'on définit la classe, la taille du tableau est inconnue. Elle ne sera connue que lorsqu'on créera une instance de cette classe, avec en paramètre la taille effective du tableau.

Chaque instance de la classe pourra avoir des tableaux de tailles différentes. Ceci permet de ne plus devoir utiliser l'allocateur dynamique pour la création de membres tableau de tailles différentes entre les instances d'une classe.

8.3.2 Combinaison de paramètres classe et constante entière

On peut créer une classe ayant en paramètre le type des objets d'un tableau, ainsi que le nombre de ces objets, pour la définition de l'un de ses membres.

```
template <class ClassParam, int TypeParam> class Generique {
        ClassParam Array[TypeParam];
public :
        Generique() {
                for( int i = 0; i < TypeParam; i++ )
                {
                        Array[i].ClassParam::Set(i);
                }
        };
        void Print() {
                for( int i = 0; i < TypeParam; i++ )
                {
                        printf("Array[i] = %d0,
                                Array[i].ClassParam::Get());
                }
        };
};

class Integer {
        int Int;
public :
        void Set(int x) { Int = x; };
        int Get() { return Int; };
};

main()
{
  Generique <Integer,10> foo;
  foo.Print();
};
```

8.3.3 Paramètre constante chaîne de caractères

Bjarne Stroustrup nous dit dans son livre [Ellis,1991] , que l'on devrait pouvoir passer une chaîne constante de caractères en paramètre de classe, et donc pouvoir écrire le code suivant :

```
template <const char * TypeParam> class Generique {
        char * Chaine;
public :
        Generique() {
                char tmp = TypeParam;
```

```
                Chaine = new [strlen(tmp) + 1];
                strcpy(Chaine,tmp);
        };
        void Print() {
                printf("%s0,Chaine);
        };
};

main()
{
        Generique <"Hello"> foo;
        foo.Print();
}
```

Cela reste à être vérifié pour un compilateur 3.2 de AT&T. La version `gcc-2.3.3` ne le permet pas encore...

8.4 Fonctions paramétrées

Nous venons de voir l'utilisation du mot clé `template` dans la définition de classes paramétrées. Ce même mot clé permet aussi de définir des fonctions paramétrées. Dans ce cas, on spécifie les paramètres dont dépendra la fonction.

```
// Fonction parametree
template <class TypeParam>
TypeParam Function(TypeParam First, TypeParam Second)
{
        return First > Second ? First : Second;
}
```

Le paramétrage doit intervenir dans les types des arguments de la fonction; il peut alors être utilisé dans le corps de la fonction. Il peut aussi être utilisé pour le type de l'objet retourné par la fonction.

Le compilateur génère autant de fonctions, de codes différents, qu'il y a de types définitifs différents utilisé dans le programme. Ainsi, il y aura, sur analyse des appels à la fonction, génération automatique du code assembleur "le meilleur" pour le système (par exemple, utilisation du processeur flottant pour le traitement de paramètres réels).

Une petite restriction existe aujourd'hui avec la version 2.3.3 de `gcc`. Elle est due à un "léger" bug de génération des symboles rattachés aux noms de fonctions templates. En effet, le compilateur génère les mêmes noms pour les différentes fonction. L'assembleur n'est alors pas très content du résultat lorsqu'on a deux appels différents à la fonction paramétrée dans le même scope fonction. On peut contourner ce problème en englobant la fonction template dans une couche intermédiaire, et en utilisant ensuite la compilation séparée. Ce bug ne restera pas longtemps vivant (:-).

```
/*---------------- Fichier 1----------------*/
// Definition de fonction parametree
template <class TypeParam>
TypeParam Function(TypeParam First, TypeParam Second)
{
        return First > Second ? First : Second;
}

// definition de fonction d'utilisation
int FuncInt(int a, int b)
{
        return Function(a,b);
}

/*---------------- Fichier 2----------------*/
```

```
// Definition de fonction parametree
template <class TypeParam>
TypeParam Function(TypeParam First, TypeParam Second)
{
        return First > Second ? First : Second;
}

// Definition de fonction d'utilisation
double FuncDouble(double a, double b)
{
        return Function(a,b);
}

/*---------------- Fichier 2----------------*/
// Prototype des fonctions utilisees
extern int FuncInt(int , int );
extern double FuncDouble(double , double );

main()
{
        // Application aux entiers
        int a = 10;
        int b = 20;
        int c = FuncInt(a,b);
        printf("a=%d b=%d c=%d0,a,b,c);

        // Application aux flottants
        double d = 99.88;
        double e = 88.99;
        double f = FuncDouble(d,e);
        printf("d=%f e=%f f=%f0,d,e,f);
}
```

8.4.1 Classe paramétrée en argument de fonction

Donnons-nous une classe paramétrée. Donnons-nous aussi une fonction pouvant recevoir en argument une instance d'une telle classe. En C++, lorsqu'on donne la définition d'une fonction, il faut impérativement donner la liste des types des arguments, pour permettre un `overload` éventuel.

```
// Definition du type  parametre
template <class PremierParam, class SecondParam>
class Foo {
private:
  PremierParam PremiereData;
  SecondParam SecondeData;

public:
  Foo() : PremiereData(0) , SecondeData(0) {;};
  ~Foo() {;};
};

// Definition d'un type cible
class Bar {
private:
  int BarX;
  int BarY;

public:
  Bar(int a = 10, int b = 11) {
    BarX = a;
    BarY = b;
  };
```

```
  ~Bar() {;};
};

// Definition de fonction a argument parametre
template <class PremierParam, class SecondParam>
void Function(Foo <PremierParam,SecondParam> ArgumentParametre, int Compteur)
{
  printf("Code de Fonction -- Appel No. %d0,Compteur);
  return;
}

// Programme d'essai
main()
{
  Foo<Bar,int> Classe;
  Function(Classe,1);
}
```

 ISBN 2-909909-02-6

9. Entrées/sorties : stream.h

Le C++ est livré en package de base avec une librairie d'entrées/sorties définie sous le nom de `streams`. Il ne faut pas confondre ces streams avec les *streams System V* qui eux sont internes au noyau Unix. Les streams C++ correspondent au "flux" des entrées/sorties sur les canaux entrée standard, sortie standard, sortie d'erreur, ou tout autre canal de communication défini par l'utilisateur au niveau de son processus.

Les données utilisées par ce package sont définies dans le fichier d'*include* `stream.h`. On y trouve la définition des classes `istream` et `ostream`, correspondant respectivement à des canaux en entrée et en sortie. Trois "canaux" sont déclarés : `cin` correspondant à l'entrée standard, `cout` correspondant à la sortie standard, et `cerr` à la sortie d'erreur.

Les classes définissant les "canaux" d'entrée et de sortie redéfinissent respectivement les opérateurs `>>` et `<<`, en les appliquant uniquement aux types de base (caractères, entiers, entiers longs et courts, flottants simple et double précision, chaînes de caractères, et pointeurs). Ces opérateurs s'utilisent de la gauche vers la droite. Leurs définitions sont intégrés dans la librairie `g++`. Il faut donc inclure l'instruction `-lg++` lors de la compilation.

```
#include <stream.h>

void f()
{
        cout << "Hello World";
        cout << 10;
        cout << 'A' << " vaut, en valeur Ascii, " << int('A') << "0;

        int i;
        cin >> i;

        char tab[10];
        cin >> tab;
        cin >> tab[5];
}
```

Lorqu'on crée un nouveau type, si on veut utiliser les streams C++ comme mécanisme d'entrées/sorties, il faut doter ces types de la redéfinition des opérateurs de *shiftage*.

Considérons la définition de classe suivante :

```
class foo {
        int a;
public :
        friend istream & operator >> ( istream &, foo &);
        friend ostream & operator << ( ostream &, foo &);
};
```

Pour pouvoir accéder aux données internes, ces fonctions doivent être spécifiées comme des amies de la classe que l'on se définit. Elles ne doivent pas être des membres de la classe (paramètre 0 implicite). Elles ne sont pas non plus des membres de la classe [io]stream correspondante (car il faudrait reprendre la définition standard de ces dernières pour tout nouveau type que l'on se créerait). Elles sont définies au niveau du scope programme. Elles reçoivent en premier paramètre le "descripteur de canal" (entrée ou sortie) sur lequel elles portent.

```
// saisie des donnees internes de la classe foo
istream & operator >> ( istream & in, foo & f )
{
        cout << "Saisie de donnee interne : ";
        in >> f.a;
}
```

```
// affichage des donnees internes de la classe foo
ostream & operator << ( ostream & out, foo & f)
{
        out << "Donnee internë " << f.a;
}
```

Dans le cas d'une saisie d'informations sur le stream d'entrée, si on veut avoir un contrôle sur le nombre maximum de caratères que l'on peut recevoir dans le récipient (ceci est notamment utilisé pour des saisies de tableau dont on connaît la taille), on peut utiliser directement les fonctions `get()` de base de la classe `iostream`. La classe `istream` est dotée d'une fonction `get()` dont le synopsis est le suivant :

```
istream & get(char *, int, char)
```

Le premier paramètre correspond à l'adresse du récipient, le second au nombre de caractères à lire sur l'entrée, et le troisième à un délimiteur de fin de saisie.

Par exemple, la redéfinition de l'opérateur `>>` pour la classe `bar`

```
#include <stream.h>

class bar {
        char tab[10];
public :
        friend istream & operator >> (istream &, bar &);
};

istream & operator >> (istream & in, bar & b)
{
        in.get(b.tab,10,'0);
        return in;
}
```

permet une saisie "sécurisée" du tableau de caractères. En effet, un maximum de 9 caractères seront pris sur l'entrée, et la saisie s'arrêtera automatiquement lorsqu'on tombe sur le caractère de fin de ligne (`\n`).

Cette fonction est très utile lorsqu'on veut "saisir" des champs de "records" délimités par autre chose qu'un fin de ligne. Par exemple, pour un record de champs délimités par un deux-points, on utilisera la fonction avec les paramètres suivants :

```
in.get(b.tab,10,':');
```

 ISBN 2-909909-02-6

10. Solution des exercices

La plupart des exercices ayant été proposés par Marc Detienne, la majorité des solutions données ici sont de sa provenance.

Solution exercice no. 1 :

Les définitions suivantes répondent à cette question :

```
// fonction ne retournant rien, et recevant un
// pointeur sur caractere et une reference sur
// entier en parametres
void Function(char * Char, int & RefInt)
{
        ;
}

// pointeur sur une telle fonction
void (*PointerToFunction)(char *, int &);

// fonction ne retournant rien et recevant en
// parametre un tel pointeur
void OtherFunction( void (*)(char *, int&) )
{
        ;
}

// definition de synonyme entre TYPE et pointeur sur
// fonction ne retournant rien et recevant en parametres
// un pointeur sur charactere et une reference sur entier
typedef void (*TYPE)(char *, int&);

// fonction retournant un tel pointeur
TYPE FunctionReturnFunction()
{
        return PointerToFunction;
}

// notation sans synonyme
void (*(FunctionRetFun())) (char*,int&)
{
        return PointerToFunction;
}

// fonction retournant un tel pointeur, et
// recevant un tel pointeur en parametre
TYPE FunctioinParAndRet( TYPE Param )
{
        return Param;
}

// notation sans synonyme
void (*(FunctionParRetFun( void (*Param) (char *, int&) ))) (char*,int&)
{
        return PointerToFunction;
}
```

Solution exercice no. 2 :

La solution à cet exercice est donnée par le programme suivant :

```
int  Print(char * str)
{
        printf("%s0,str);
}

main()
{
        Print("Hello World");
        Print(10);
}
```

On notera un *warning* sur l'appel de fonction avec l'entier `10`, qui relève d'une erreur d'utilisation.

Solution exercice no. 3 :

La solution à cet exercice est donnée par le programme suivant :

```
int  Print(int i, char * str)
{
        printf("%d %s0,i,str);
}

main()
{
        Print(10,"Hello World");
        Print(10);
        Print("Hello World");
}
```

Ce programme bloque sur les deux derniers appels à la fonction `Print()`.

Solution exercice no. 4 :

Le premier fichier contient la définition de la fonction `Print()` :

```
int  Print(int i, char * str)
{
        printf("%d %s0,i,str);
}
```

Le second fichier contient la définition de la fonction `main()`, ainsi que la déclaration de la fonction `Print()` :

```
extern int  Print(int,char*);

main()
{
        Print(10,"Hello World");
}
```

Solution exercice no. 5 :

La solution au problème est donnée par le programme suivant :

```
overload Print;

int  Print(int i)
{
        printf("%d0,i);
}

int  Print(char * str)
{
        printf("%s0,str);
}

int  Print(int i, char * str)
{
        printf("%d %s0,i,str);
}

main()
{
        Print(10);
        Print("Hello World");
        Print(10,"Hello World");
}
```

Solution exercice no. 6 :

La traduction du programme C en C++ donne le code suivant :

```
void * Alloc(int x)
{
        return new char [x];
}

void Free(void * x)
{
        delete x;
}

main()
{
        char * ptr;

        ptr = Alloc(10);
        Free(ptr);
}
```

Solution exercice no. 7 :

Le programme suivant donne un exemple de fonctions à paramètres par valeur (copie), adresse (pointeurs), et référence.

```
void FunctionCopy(int x)
{
        x++;
}

void FunctionAddress(int * x)
{
        (*x)++;
}

void FunctionReference(int& x)
{
        x++;
}

main()
{
        int i = 10;
        FunctionCopy(i);
        printf("%d0,i);

        FunctionAddress(&i);
        printf("%d0,i);

        FunctionReference(i);
        printf("%d0,i);
}
```

L'appel à la fonction `FunctionCopy()` ne modifie pas la valeur de l'entier `i` (le code de la fonction travaille sur une copie de la variable d'appel); l'appel à la fonction `FunctionAddress()` modifie la variable d'appel, mais par utilisation explicite des opérateurs d'adressage; la fonction `FunctionReference()` utilise une référence sur la variable d'appel. Celle-ci est donc modifiée. D'un point de vue du programme appelant, on ne voit pas de différence entre passages de paramètres lors des appels aux fonctions `FunctionCopy()` et `FunctionReference()`. Les code de celles-ci ne sont pas différents non plus. La seule différence intervient au niveau de leur prototype.

Solution exercice no. 8 :

Les fonctionnalités caractérisant une stack (FIFO) sont les suivantes :

- create (création)
- erase (vider)
- push (empiler)

- pop (dépiler)
- top (sommet)

Le code C+ correspondant pourraît être le suivant :

```
void Error(char * msg)
{
        printf("%s0,msg);
}

typedef
struct _stack {
        int * StackSpace;
        int StackSize;
        int StackCount;
} stack;

stack * Stack;

void Create(int Size)
{
        Stack = new stack;
        Stack->StackSpace = new int [Size];
        Stack->StackSize = Size;
        Stack->StackCount = 0;
}

void Erase()
{
        delete Stack->StackSpace;
        Stack->StackSpace = new int [Stack->StackSize];
        Stack->StackCount = -1;
}

void Push(int Data)
{
        if(Stack->StackCount == Stack->StackSize)
        {
                Error("Stack overflow");
        }
        *(Stack->StackSpace + Stack->StackCount++) = Data;
}

int Pop()
{
        if(Stack->StackCount == 0)
        {
                Error("Stack underflow");
        }

        return *(Stack->StackSpace + --Stack->StackCount);
}

int Top()
{
        if(Stack->StackCount == 0)
        {
                Error("Empty Stack");
        }
        return *(Stack->StackSpace + (Stack->StackCount-1));
}

void Print()
{
        for(int i = 0; i < Stack->StackCount; i++)
        {
                printf("%d ",*(Stack->StackSpace+i));
        }
        printf("0);
}


main()
{
        Create(5);
        Push(1);
        Push(2);
        Print();
```

```
        int i;
        i = Pop();
        printf("i=%d0,i);
        Print();

        Push(3);
        Print();

        Erase();
        Print();
}
```

Solution exercice no. 9 :

Cet exercice demande à réaliser une classe maintenant trois données, le minimum, le maximum et la valeur moyenne, pour des valeurs entières fournies à une instance de cette classe. Ces données doivent être accessibles du programme utilisateur. Par contre, on doit maintenir un compteur et le total des valeurs fournies à la classe. Ces informations sont strictement internes à la classe.

La solution à cet exercice est donnée par le programme suivant :

```
class Statistic {
private :
        int Total;
        int Count;
        void UpdateMinimum(int);
        void UpdateMaximum(int);
        void UpdateAverage(int);
public:
        int Minimum;
        int Maximum;
        int Average;
        void Set(int);
};

void Statistic::UpdateMinimum(int x)
{
        if( x < Minimum )
        {
                Minimum = x;
        }
}

void Statistic::UpdateMaximum(int x)
{
        if( x > Maximum )
        {
                Maximum = x;
        }
}

void Statistic::UpdateAverage(int x)
{
        Total += x;
        Count++;
        Average = Total / Count;
}

void Statistic::Set(int a)
{
        UpdateMinimum(a);
        UpdateMaximum(a);
        UpdateAverage(a);
}
```

Vous remarquerez les services internes permettant la mise à jour des données de la classe. Ces fonctions ne sont pas connues en dehors du scope de la classe.

Solution exercice no. 10 :

La modification apportée par rapport à l'exercice précédent intervient au niveau de la visibilité des

donnée Minimum, Maximum, et Average. Elles ne sont plus directement accessibles. Il faut donc fournir les services qui permettent de les "voir" à partir du scope programme.

```
class Statistic {
private :
        int Total;
        int Count;
        int Minimum;
        int Maximum;
        int Average;
        void UpdateMinimum(int);
        void UpdateMaximum(int);
        void UpdateAverage(int);
public:
        void Set(int);
        int GetMinimum();
        int GetMaximum();
        int GetAverage();
};

void Statistic::UpdateMinimum(int x)
{
        if( x < Minimum )
        {
                Minimum = x;
        }
}

void Statistic::UpdateMaximum(int x)
{
        if( x > Maximum )
        {
                Maximum = x;
        }
}

void Statistic::UpdateAverage(int x)
{
        Total += x;
        Count++;
        Average = Total / Count;
}

void Statistic::Set(int a)
{
        UpdateMinimum(a);
        UpdateMaximum(a);
        UpdateAverage(a);
}

int Statistic::GetMinimum()
{
        return Minimum;
}

int Statistic::GetMaximum()
{
        return Maximum;
}

int Statistic::GetAverage()
{
        return Average;
}
```

Solution exercice no. 11 :

La liste des services rattaché à un compteur est totalement définie par les fonctionalités suivantes :

— incrémenter,

— décrémenter,

 ISBN 2-909909-02-6

— remettre à zéro

Il faut se doter d'un récipient, donnée interne d'implémentation, non visible hors du scope du compteur.

Une première implantation consiste à introduire des fonctions explicitement nommées :

```
class Compteur {
        int Value;
public :
        int Increment();
        int Decrement();
        int Reset();
        int GetValue();
};

int Compteur::Increment()
{
        return ++Value;
}

int Compteur::Decrement()
{
        return --Value;
}

int Compteur::Reset()
{
        return Value = 0;
}

int Compteur::GetValue()
{
        return Value;
}
```

Une seconde implantation propose l'utilisation des opérateurs d'incrémentation et de décrémentation :

```
class Compteur {
        int Value;
public :
        int operator ++ ();
        int operator -- ();
        int operator = (int);
};

int Compteur::operator ++ ()
{
        return ++Value;
}

int Compteur::operator -- ()
{
        return --Value;
}

int Compteur::operator = (int x)
{
        return Value = x;
}

int Compteur::GetValue()
{
        return Value;
}

main()
{
        Compteur a;
        a++;
        int x = a.GetValue();
        a--;
}
```

Solution exercice no. 12 :

Un drapeau est totalement défini par les fonctionnalités suivantes :

— lever le drapeau,

— baisser le drapeau.

On obtient la définition de type suivante :

```
class Drapeau {
        int Value;
public :
        void Lever() {
                Value = 1;
        };
        void Baisser() {
                Value = 0;
        };
        int GetValue() {
                return Value;
        };
};
```

Une seconde implémentation permet de d'utiliser le même programme principal que pour l'exercice précedent, mais en l'appliquant à un `drapeau` :

```
class Drapeau {
        int Value;
public :
        int operator ++ () {
                return Value = 1;
        };
        int operator -- () {
                return Value = 0;
        };
        int GetValue() {
                return Value;
        };
};
```

Solution exercice no. 13 :

Un sommateur ne sait faire que des additions. On lui fournit un paramètre, objet à "ajouter".

```
class Sommateur {
        int Value;
        void Add(int x) {
                Value += x;
        };
public :
        Sommateur & operator += (int val) {
                Sommateur * foo;
                foo = new Sommateur;
                foo->Value = Value;
                foo->Add(val);
                return *foo;
        };

        Sommateur & operator ++ () {
                Sommateur * foo;
                foo = new Sommateur;
                foo->Value = Value;
                foo->Add(1);
                return *foo;
        };
};
```

Solution exercice no. 14 :

On aimerait qu'une chaîne de caractères puisse avoir un récipient de taille dynamique variant suivant son utilisation. On doit pouvoir initialiser une chaîne de caractères avec un chaîne constante, copier une chaîne de caractères dans une autre, et pouvoir concaténer deux chaînes de caractères.

```
class String {
        char * Recipient;
        void ClearRecipient() {
                if( Recipient == (char *)0 )
                {
                        delete Recipient;
                }
        };
public :
        void operator = (char * foo ) {
                ClearRecipient();
                Recipient = new char[strlen(foo) + 1];
                strcpy(Recipient,foo);
        };
        void operator = (String & foo) {
                ClearRecipient();
                Recipient = new char[strlen(foo) + 1];
                strcpy(Recipient,foo);
        };
        void operator + (String & foo ) {
                char * Temp;
                Temp = new char[strlen(Recipient) + strlen(foo) + 1]
                strcpy(Recipient,Temp);
                strcat(foo,Temp);
                ClearRecipient();
                Recipient = Temp;
        };
};
```

Solution exercice no. 15 :

Un fichier Unix, sous Unix System V se limite, localement au directory courant, à 14 caractères

Solution exercice no. 16 :

Un numéro de canal est limité en valeur maximale par le système. Il ne peut être négatif. Il correspond à un fichier ouvert.

Solution exercice no. 17 :

Il s'agit ici d'utiliser le constructeur pour initialiser l'instance de la classe. On obtient le code suivant :

```
class Statistic
{
   int Total;
   int Count;

   // Internal Routines
   void UpdateAverage(int Value);
   void UpdateMinimum(int Value);
   void UpdateMaximum(int Value);

public:
   int Minimum;
   int Maximum;
   int Average;

   // Standard Services
   Statistic();
   ~Statistic();

   // Specific Services
   void Update(int Value);

   // Presentation Services
   Statistic& operator=(int Value);
};

// Standard Services
Inline Statistic::Statistic()
{
   Total = 0;
   Count = 0;
```

```
   Minimum = 0;
   Maximum = 0;
   Average = 0;
}

Inline Statistic::~Statistic()
{
   ;
}

// Specific Services
Inline void Statistic::Update(int Value)
{
   UpdateMinimum(Value);
   UpdateMaximum(Value);
   UpdateAverage(Value);
}

// Presentation Services
Inline Statistic& Statistic::operator=(int Value)
{
   Update(Value);
   return *this;
}

// Internal Routines
Inline void Statistic::UpdateMinimum(int Value)
{
   if(Value < Minimum)
   {
        Minimum = Value;
   }
}

Inline void Statistic::UpdateMaximum(int Value)
{
   if(Value > Maximum)
   {
        Maximum = Value;
   }
}

Inline void Statistic::UpdateAverage(int Value)
{
   Total += Value;
   Count++;
   Average = Total / Count;
}
```

Solution exercice no. 18 :

La solution proposé ici utilise une arborescence de classes de petites entités (Minimum, Maximum, Average, Total)

```
// ---------- Classe Total ----------
class Total
{
privatë
   int Value;

public:
   // Standard Services
   Total();
   Total(int StartValue);

   ~Total();

   // Specific Services
   void Update(int NewValue);

   // Presentation Services
   Total& operator=(int NewValue);
   operator int();
};

// Standard Services
Inline Total::Total()
{
```

```
   Total(0);
}

Inline Total::Total(int StartValue)
{
   Value = StartValue;
}

Inline Total::~Total()
{
   ;
}

// Specific Services
Inline void Total::Update(int NewValue)
{
   Value += NewValue;
}

// Presentation Services
Inline Total& Total::operator=(int NewValue)
{
   Update(NewValue);
   return *this;
}

Inline Total::operator int()
{
   return Value;
}

// ---------- Classe Average ----------
class Average
{
privatë
   Total Total;
   int Count;

public:
   // Standard Services
   Average();
   Average(int StartValue);

   ~Average();

   // Specific Services
   void Update(int NewValue);

   // Presentation Services
   Average& operator=(int NewValue);
   operator int();
};

// Standard Services
Inline Averagë:Average():Total(0)
{
   Average(0);
}

Inline Averagë:Average(int StartValue):Total(0)
{
   Count = 0;
}

Inline Averagë:~Average()
{
   ;
}

// Specific Services
Inline void Averagë:Update(int NewValue)
{
   Total = NewValue;
   Count++;
}

// Presentation Services
Inline Average& Averagë:operator=(int NewValue)
{
   Update(NewValue);
```

```
}

Inline Averagë:operator int()
{
   return (int(Total) / Count);
}

// ---------- Classe Minimum ----------
class Minimum
{
privatë
   int Value;

public:
   // Standard Services
   Minimum();
   Minimum(int StartValue);

   ~Minimum();

   // Specific Services
   void Update(int NewValue);

   // Presentation Services
   Minimum& operator=(int NewValue);
   operator int();
};

// Standard Services
Inline Minimum::Minimum()
{
   Minimum(0);
}

Inline Minimum::Minimum(int StartValue)
{
   Value = StartValue;
}

Inline Minimum::~Minimum()
{
   ;
}

// Specific Services
Inline void Minimum::Update(int NewValue)
{
   if(NewValue < Value)
   {
        Value = NewValue;
   }
}

// Presentation Services
Inline Minimum& Minimum::operator=(int NewValue)
{
   Update(NewValue);
   return *this;
}

Inline Minimum::operator int()
{
   return Value;
}

// ---------- Classe Maximum ----------
class Maximum
{
privatë
   int Value;

public:
   // Standard Services
   Maximum();
   Maximum(int StartValue);

   ~Maximum();

   // Specific Services
   void Update(int NewValue);
```

```
   // Presentation Services
   Maximum& operator=(int NewValue);
   operator int();
};

// Standard Services
Inline Maximum::Maximum()
{
   Maximum(0);
}

Inline Maximum::Maximum(int StartValue)
{
   Value = StartValue;
}

Inline Maximum::~Maximum()
{
   ;
}

// Specific Services
Inline void Maximum::Update(int NewValue)
{
   if(NewValue > Value)
   {
        Value = NewValue;
   }
}

// Presentation Services
Inline Maximum& Maximum::operator=(int NewValue)
{
   Update(NewValue);
   return *this;
}

Inline Maximum::operator int()
{
   return Value;
}

// ---------- Classe Statistic ----------
class Statistic
{
public:
   Minimum Minimum;
   Maximum Maximum;
   Average Average;

   // Standard Services
   Statistic();
   ~Statistic();

   // Specific Services
   void Update(int Value);

   // Presentation Services
   Statistic& operator=(int Value);
};

// Standard Services
Inline Statistic::Statistic():Minimum(0),Maximum(0),Average(0)
{
   ;
}

Inline Statistic::~Statistic()
{
   ;
}

// Specific Services
Inline void Statistic::Update(int Value)
{
   Minimum = Value;
   Maximum = Value;
   Average = Value;
}
```

```
// Presentation Services
Inline Statistic& Statistic::operator=(int Value)
{
   Update(Value);
   return *this;
}
```

Solution exercice no. 19 :

On reprend la définition fonctionnelle d'une stack, mais on l'implémente cette fois-ci avec les concepts de construction, de données protégées, de fonctions propres à l'implémentation, de services en accès publiç qui constituent les caractéristiques de base d'un objet C++.

```
class stack {
        int * StackSpace;
        int StackSize;
        int StackCount;
        void Error(char * msg) {
                printf("%s0,msg);
        };
public :
        stack(int);
        ~stack();
        void Erase();
        void Push(int);
        int Pop();
        int Top();
        void Print();
};

stack::stack(int Size)
{
        StackSpace = new int [Size];
        StackSize = Size;
        StackCount = 0;
}

stack::~stack(int Size)
{
        delete StackSpace;
}

void stack::Erase()
{
        delete StackSpace;
        StackSpace = new int [StackSize];
        StackCount = 0;
}

void stack::Push(int Data)
{
        if(StackCount == StackSize)
        {
                Error("Stack overflow");
        }
        *(StackSpace + StackCount++) = Data;
}

int stack::Pop()
{
        if(StackCount == 0)
        {
                Error("Stack underflow");
        }

        return *(StackSpace + --StackCount);
}

int stack::Top()
{
        if(StackCount == 0)
        {
                Error("Empty Stack");
        }
        return *(StackSpace + (StackCount-1));
}
```

```
void stack::Print()
{
        for(int i = 0; i < StackCount; i++)
        {
                printf("%d ",*(StackSpace+i));
        }
        printf("0);
}


stack Stack(5);

main()
{
        Stack.Push(1);
        Stack.Push(2);
        Stack.Print();

        int i;
        i = Stack.Pop();
        printf("i=%d0,i);
        Stack.Print();

        Stack.Push(3);
        Stack.Print();

        Stack.Erase();
        Stack.Print();
}
```

Solution exercice no. 20 :

Il s'agit ici de monter un interface de fichiers. Les fichiers doivent pouvoir être créés, sur spécification utilisateur. Les fichiers doivent être vus comme des tableaux de caractères.

```
#define BUFFER_SIZE 1024

class File {
        char * filename;
        int fd;
        long offset;
        void Error(char * msg) {
                printf("Error: [%s] %s0,filename,msg);
                exit(1);
        };
public :
        File(char *);
        File(char *, int);
        ~File();
        File& operator [] (int);
        File& operator = (char);
        File& operator = (File &);
};

// constructeur avec nom de fichier en parametre
Filë:File(char * name)
{
        filename = new char[strlen(name) + 1];
        strcpy(filename,name);

        if( (fd = open(name,O_RDWR|O_CREAT,0666)) < 0 )
        {
                Error("Cannot open file");
        }

        offset = 0L;
}

// constructeur avec nom de fichier et autorisations d'acces
Filë:File(char * name, int mode)
{
        filename = new char[strlen(name) + 1];
        strcpy(filename,name);

        if( (fd = creat(name,mode)) < 0 )
        {
                Error("Cannot create file");
```

```
        }
        close(fd);
        if( (fd = open(name,O_RDWR)) < 0 )
        {
                Error("Cannot open a created file");
        }

        offset = 0L;
}

// destructeur
Filë:~File()
{
        close(fd);
        delete filename;
}

// vue sous forme de tableau du fichier
File& Filë:operator [] (int i)
{
        if( i < 0 )
        {
                Error("Negative offset?");
        }
        offset = (long)i;
        return *this;
}

// ajout d'un caractere dans le fichier
File& Filë:operator = (char c)
{
        if( lseek(fd,offset,0) < 0 )
        {
                Error("Cannot lseek");
        }

        if( write(fd,&çsizeof c) != sizeof c )
        {
                Error("Cannot write a char");
        }

        offset = 0L;
        return *this;
}

// recopie de deux fichiers
File& Filë:operator = (File& from)
{
        if( lseek(fd,offset,0) < 0 )
        {
                Error("Cannot lseek for l-value File");
        }
        if( lseek(from.fd,from.offset,0) < 0 )
        {
                Error("Cannot lseek for r-value File");
        }

        char buffer[BUFFER_SIZE];
        int nb_bytes;
        int nb_bytes_to_copy;

        if( from.offset != 0L )
        {
                nb_bytes_to_copy = offset;
        }
        else
        {
                nb_bytes_to_copy = BUFFER_SIZE;
        }

        while((nb_bytes = read(from.fd,buffer,nb_bytes_to_copy)) > 0)
        {
                if( write(fd,buffer,nb_bytes) != nb_bytes )
                {
                        Error("Cannot write in l-value");
                }
        }

        offset = from.offset = 0L;
        return *this;
```

```
}
```

Solution exercice no. 21 :

On demande ici à définir une classe d'entiers qui réponde aux mêmes spécifications attendues des opérateurs que lorsqu'ils sont appliqués à des entiers machine habituels. Deux possibilités sont offertes : soit on redéfinit tout les opératuers utilisés et faisant intervenir les deux types (`Integer` et `int`), soit on opte pour une solution utilisant le `cast`.

Première solution :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };
        Integer(int x) { i = x; };
        Integer operator + (Integer);
        Integer operator * (Integer);
        int operator < (Integer);
        Integer operator ++ ();
        operator int();
};

Integer Integer::operator + (Integer x)
{
        return Integer(i + x.i);
}

Integer Integer::operator * (Integer x)
{
        return Integer(i * x.i);
}

int Integer::operator < (Integer x)
{
        return i < x.i;
}

Integer Integer::operator ++ ()
{
        i++;
        return Integer(i);
}

Integer::operator int()
{
        return i;
}
```

Deuxieme solution :

```
class Integer {
        int i;
public :
        Integer() { i = 0; };

        // outil d'importation
        Integer(int x) { i = x; };

        // outil d'exportation
        operator int() { return i; };

        operator++() { i++; };
};
```

Solution exercice no. 22 :

Le tableau demande une allocation dynamique à la construction. On doit pourvoir à une "vue" de tableau, avec utilisation transparente de l'opérateur `[ ]`. Un mécanisme de contrôle sur l'indice doit ête monté.

```
// definition des fonction d'erreur
```

```
Inline void Error(const char* Message)
{
   printf(Message);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument)
{
   printf(Message,Argument);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument,int AnotherOne)
{
   printf(Message,Argument,AnotherOne);
   printf("0);
   exit(1);
}

// definition de la classe tableau d'entiers
class ArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   int* AllocateArray(int Size);
   void FreeArray(int* Array);
   void SetUpArray(int Size,int Initializer);
   void InitializeArray(int Size,int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,
              int SecondArgument);
   int  CheckIndex(int Index);
   int  CheckSize(int RequestedSize);
   int  SetUpSize(int RequestedSize);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);

public:
   // Standard Services
   ArrayOfInteger(int RequestedSize);
   ~ArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);

};

// Standard Services
Inline ArrayOfInteger::ArrayOfInteger(int RequestedSize)
{
   SetUpSize(RequestedSize);
   SetUpArray(RequestedSize,0);
}

Inline ArrayOfInteger::~ArrayOfInteger()
{
   FreeArray(Array);
}

// Specific Services
Inline void ArrayOfInteger::Print()
{
   Print("%d");
}

Inline void ArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;
```

```
   for(Scanner = Array,End = Array + Size;
         Scanner < End;Scanner++)
   {
         PrintElement(Scanner,Format);
   }
}

Inline int& ArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
         Error("Index [%d] is out of allowed range [0,%d]",
                 Index,Size-1);
   }
   return *(Array + Index);
}

// Presentation Services
Inline int& ArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines
Inline int* ArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   if((Memory = new int[Size]) == 0)
   {
         Error("Unable to allocate [%d] int's in your program",Size);
   }
   return Memory;
}

Inline void ArrayOfInteger::FreeArray(int* Array)
{
   delete Array;
}

Inline void ArrayOfInteger::SetUpArray(int Size,int Initializer)
{
   Array = AllocateArray(Size);
   InitializeArray(Size,Initializer);
}

Inline void ArrayOfInteger::InitializeArray(int Size,int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
         InitializeElement(Scanner,Initializer);
   }
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,
                                  int Argument,
                                  int AnotherOne)
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= 0 && Index < Size);
}

Inline int ArrayOfInteger::CheckSize(int Size)
```

```
{
   return (Size > 0);
}

Inline int ArrayOfInteger::SetUpSize(int RequestedSize)
{
   if(!CheckSize(RequestedSize))
   {
         Error("Size [%d] is not a strictly positive integer",
                 RequestedSize);
   }
   Size = RequestedSize;
}

Inline
void ArrayOfInteger::InitializeElement(int* Element,const int Value)
{
   *Element = Value;
}

Inline void ArrayOfInteger::PrintElement(int* Element,const char* Format)
{
   printf(Format,*Element);
   printf("0);
}
```

Solution exercice no. 23 :

On doit prendre en compte le fait que l'on puisse donner des indexs d'initialisation et d'utilisation négatifs et positifs.

```
// Fonction d'erreur
Inline void Error(const char* Message)
{
   printf(Message);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument)
{
   printf(Message,Argument);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument,int AnotherOne)
{
   printf(Message,Argument,AnotherOne);
   printf("0);
   exit(1);
}

// Definition du tableau etendu d'entiers
class ExtendedArrayOfInteger
{
   int LowerIndex;
   int HigherIndex;
   int* Array;

   // Internal Routines
   int* AllocateArray();
   void FreeArray();
   void SetUpArray(int Initializer);
   void InitializeArray(int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,
                        int SecondArgument);
   int  CheckIndex(int Index);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);
   int Size();
   int MapVirtualIndexToPhysicalIndex(int Index);
   int MapPhysicalIndexToVirtualIndex(int Index);

public:
```

```
   // Standard Services
   ExtendedArrayOfInteger(int RequestedLowerIndex,
                  int RequestedHigherIndex);
   ~ExtendedArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);

};

// fonction locale au fichier de permutation
Inline static void Permute(int& First,int& Second)
{
   int Temporary;

   Temporary = First;
   First = Second;
   Second = Temporary;
}

// Standard Services
Inline ExtendedArrayOfInteger::ExtendedArrayOfInteger(
    int RequestedLowerIndex,
    int RequestedHigherIndex)
{
   if(RequestedHigherIndex < RequestedLowerIndex)
   {
        Permute(RequestedLowerIndex,RequestedHigherIndex);
   }
   LowerIndex = RequestedLowerIndex;
   HigherIndex = RequestedHigherIndex;
   SetUpArray(0);
}

Inline ExtendedArrayOfInteger::~ExtendedArrayOfInteger()
{
   FreeArray();
}

// Specific Services
Inline void ExtendedArrayOfInteger::Print()
{
   Print("%d");
}

Inline void ExtendedArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size();Scanner < End;Scanner++)
   {
        PrintElement(Scanner,Format);
   }
}

Inline int& ExtendedArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
        Error("Index [%d] is out of allowed range [%d,%d]",
            LowerIndex,HigherIndex);
   }
   return *(Array + MapVirtualIndexToPhysicalIndex(Index));
}

// Presentation Services
Inline int& ExtendedArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines
Inline int* ExtendedArrayOfInteger::AllocateArray()
{
```

```
   int* Memory;

   if((Memory = new int[Size()]) == 0)
   {
        Error("Unable to allocate [%d] int's in your program",Size());
   }
   return Memory;
}

Inline void ExtendedArrayOfInteger::FreeArray()
{
   delete Array;
}

Inline void ExtendedArrayOfInteger::SetUpArray(int Initializer)
{
   Array = AllocateArray();
   InitializeArray(Initializer);
}

Inline void ExtendedArrayOfInteger::InitializeArray(int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size();Scanner < End;Scanner++)
   {
        InitializeElement(Scanner,Initializer);
   }
}

Inline void ExtendedArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ExtendedArrayOfInteger::Error(const char* ErrorMessage,
                                          int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ExtendedArrayOfInteger::Error(const char* ErrorMessage,
                                          int Argument,
                                          int AnotherOne)
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ExtendedArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= LowerIndex && Index <= HigherIndex);
}

Inline void ExtendedArrayOfInteger::InitializeElement(int* Element,
                                                      const int Value)
{
   *Element = Value;
}

Inline void ExtendedArrayOfInteger::PrintElement(int* Element,
                                                 const char* Format)
{
   printf("[%d] = ",MapPhysicalIndexToVirtualIndex(Element - Array));
   printf(Format,*Element);
   printf("0);
}

Inline int ExtendedArrayOfInteger::Size()
{
   return (HigherIndex - LowerIndex + 1);
}

Inline
int ExtendedArrayOfInteger::MapVirtualIndexToPhysicalIndex(int Index)
{
   return(Index - LowerIndex);
}

Inline
```

```
int ExtendedArrayOfInteger::MapPhysicalIndexToVirtualIndex(int Index)
{
   return(Index + LowerIndex);
}
```

Solution exercice no. 24 :

Dans la solution présentée ici, la classe tableau étendu contient un membre de type tableau d'entiers.

```
// fonctions d'erreur
Inline void Error(const char* Message)
{
   printf(Message);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument)
{
   printf(Message,Argument);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument,int AnotherOne)
{
   printf(Message,Argument,AnotherOne);
   printf("0);
   exit(1);
}

// Definition de la classe tableau d'entiers
class ArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   int* AllocateArray(int Size);
   void FreeArray(int* Array);
   void SetUpArray(int Size,int Initializer);
   void InitializeArray(int Size,int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,int SecondArgument);
   int  CheckIndex(int Index);
   int  CheckSize(int RequestedSize);
   int  SetUpSize(int RequestedSize);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);

public:
   // Standard Services
   ArrayOfInteger(int RequestedSize);
   ~ArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);

};

// Standard Services
Inline ArrayOfInteger::ArrayOfInteger(int RequestedSize)
{
   SetUpSize(RequestedSize);
   SetUpArray(RequestedSize,0);
}

Inline ArrayOfInteger::~ArrayOfInteger()
{
   FreeArray(Array);
}
```

```
// Specific Services
Inline void ArrayOfInteger::Print()
{
   Print("%d");
}

Inline void ArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        PrintElement(Scanner,Format);
   }
}

Inline int& ArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
        Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
   }
   return *(Array + Index);
}

// Presentation Services
Inline int& ArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines

Inline int* ArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   if((Memory = new int[Size]) == 0)
   {
        Error("Unable to allocate [%d] int's in your program",Size);
   }
   return Memory;
}

Inline void ArrayOfInteger::FreeArray(int* Array)
{
   delete Array;
}

Inline void ArrayOfInteger::SetUpArray(int Size,int Initializer)
{
   Array = AllocateArray(Size);
   InitializeArray(Size,Initializer);
}

Inline void ArrayOfInteger::InitializeArray(int Size,int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        InitializeElement(Scanner,Initializer);
   }
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,
                                  int Argument,
                                  int AnotherOne)
```

```
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= 0 && Index < Size);
}

Inline int ArrayOfInteger::CheckSize(int Size)
{
   return (Size > 0);
}

Inline int ArrayOfInteger::SetUpSize(int RequestedSize)
{
   if(!CheckSize(RequestedSize))
   {
        Error("Size [%d] is not a strictly positive integer",RequestedSize);
   }
   Size = RequestedSize;
}

Inline void ArrayOfInteger::InitializeElement(int* Element,const int Value)
{
   *Element = Value;
}

Inline void ArrayOfInteger::PrintElement(int* Element,const char* Format)
{
   printf(Format,*Element);
   printf("0);
}

// Definition de la classe tableau etendu d'entiers
class ExtendedArrayOfInteger
{
   ArrayOfInteger Array;
   int LowerIndex;

   // Internal Services
   int MapVirtualIndexToPhysicalIndex(int Index);

public:
   // Standard Services
   ExtendedArrayOfInteger(int RequestedLowerIndex,
                  int RequestedHigherIndex);
   ~ExtendedArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);
};

// Standard Services
Inline ExtendedArrayOfInteger::ExtendedArrayOfInteger(
        int RequestedLowerIndex,int RequestedHigherIndex)
        :Array(RequestedHigherIndex-RequestedLowerIndex+1)
{
   LowerIndex = RequestedLowerIndex;
}

Inline ExtendedArrayOfInteger::~ExtendedArrayOfInteger()
{
   ;
}

// Specific Services
Inline void ExtendedArrayOfInteger::Print()
{
   Array.Print();
}

Inline void ExtendedArrayOfInteger::Print(const char* Format)
{
   Array.Print(Format);
```

```
}

Inline int& ExtendedArrayOfInteger::FetchElement(int Index)
{
   return Array.FetchElement(MapVirtualIndexToPhysicalIndex(Index));
}

// Presentation Services
Inline int& ExtendedArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Services
Inline int ExtendedArrayOfInteger::MapVirtualIndexToPhysicalIndex(int Index)
{
   return (Index - LowerIndex);
}
```

Solution exercice no. 25 :

Les fonctions de la classe doivent prévoir une réallocation des éléments du tableau déjà alloué. L'indexation doit prendre en compte les nouvelles "bornes" du tableau, en cas de modification de taille.

Deux solutions sont proposées pour cet exercice. La première solution est plus proche de la programmation "classique" en C. Les services font tout le travail.

```
 1 // Fonction d'erreur
 2 Inline void Error(const char* Message)
 3 {
 4     printf(Message);
 5     printf("0);
 6     exit(1);
 7 }

 8 Inline void Error(const char* Message,int Argument)
 9 {
10     printf(Message,Argument);
11     printf("0);
12     exit(1);
13 }

14 Inline void Error(const char* Message,int Argument,int AnotherOne)
15 {
16     printf(Message,Argument,AnotherOne);
17     printf("0);
18     exit(1);
19 }

20 // Definition de la classe tableau dynamique
21 class DynamicArrayOfInteger
22 {
23     int Size;
24     int* Array;

25 public:
26     // Standard Services
27     DynamicArrayOfInteger(int RequestedSize);
28     ~DynamicArrayOfInteger();

29     // Specific Services
30     void Print();
31     void Resize(int NewSize);

32     int& FetchElement(int Index);

33     // Presentation Services
34     int& operator[](int Index);
35 };

36 // Standard Services
37 Inline
38 DynamicArrayOfInteger::DynamicArrayOfInteger(int RequestedSize)
39 {
40     if(RequestedSize <= 0)
```

```
41      {
42          Error("Size [%d] must be a strictly positive Integer",
43                  RequestedSize);
44      }
45      Size = RequestedSize;
46      Array = new int[Size];
47      if(Array == 0)
48      {
49          Error("Cannot allocate [%d] int's in your program",Size);
50      }
51      for(int Index = 0; Index < Size; Index++)
52      {
53          Array[Index] = 0;
54      }
55  }

56  Inline DynamicArrayOfInteger::~DynamicArrayOfInteger()
57  {
58      delete Array;
59  }

60  // Specific Services
61  Inline void DynamicArrayOfInteger::Print()
62  {
63      for(int Index = 0; Index < Size; Index++)
64      {
65          printf("Array[%d] = %d0,Index,Array[Index]);
66      }
67      printf("0);
68  }

69  inline static int Minimum(int First,int Second)
70  {
71      return((First < Second)?First:Second);
72  }

73  Inline void DynamicArrayOfInteger::Resize(int NewSize)
74  {
75      int* NewArray;

76      if(NewSize < 0)
77      {
78          Error("Size [%d] must be a strictly positive Integer",Size);
79      }
80      if((NewArray = new int[NewSize]) == 0)
81      {
82          Error("Cannot allocate [%d] int's in your program",Size);
83      }
84      for(int Index = 0;Index < Minimum(NewSize,Size);Index++)
85      {
86          NewArray[Index] = Array[Index];
87      }
88      for(;Index < NewSize; Index++)
89      {
90          NewArray[Index] = 0;
91      }
92      delete Array;
93      Array = NewArray;
94      Size = NewSize;
95  }

96  Inline int& DynamicArrayOfInteger::FetchElement(int Index)
97  {
98      if(Index < 0 || Index >= Size)
99      {
100         Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
101     }

102     return(Array[Index]);
103 }


104 // Presentation Services
105 Inline int& DynamicArrayOfInteger::operator[](int Index)
106 {
107     return FetchElement(Index);
108 }
```

La seconde solution propose des fonctions de services allégées, mais utilisant une implantation plus "forte". La classe est dotée d'un ensemble conséquent de fonctions internes.

```
// Definition de la classe tableau dynamique d'entiers
class DynamicArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   void CheckSize(int ProposedSize);
   void SetSize(int ProposedSize);
   void BuildArray();
   void DeleteArray();
   void PrintElement(int Index);
   int* AllocateArray(int Size);
   void CopyPartialArray(int* Source,int* Destination,int Limit,int End);
   void SetArray(int* Array);
   void InitializeArray(int* ThatArray,int ThatSize);
   void Clear();

public:
   // Standard Services
   DynamicArrayOfInteger(int RequestedSize);
   ~DynamicArrayOfInteger();

   // Specific Services
   void Print();
   void Resize(int NewSize);

   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);
};

// Standard Services
Inline DynamicArrayOfInteger::DynamicArrayOfInteger(int RequestedSize)
{
   Clear();
   CheckSize(RequestedSize);
   SetSize(RequestedSize);
   BuildArray();
}

Inline DynamicArrayOfInteger::~DynamicArrayOfInteger()
{
   assert(Array != 0);

   DeleteArray();
}

// Specific Services
Inline void DynamicArrayOfInteger::Print()
{
   for(int Index = 0; Index < Size; Index++)
   {
        PrintElement(Index);
   }
   printf("0);
}

inline static int Minimum(int First,int Second)
{
   return((First < Second)?First:Second);
}

Inline void DynamicArrayOfInteger::Resize(int NewSize)
{
   int* NewArray;

   assert(NewSize > 0);

   CheckSize(NewSize);
   NewArray = AllocateArray(NewSize);
   InitializeArray(NewArray,NewSize);
   CopyPartialArray(Array,NewArray,Minimum(NewSize,Size),NewSize);
   DeleteArray();
   SetArray(NewArray);
   SetSize(NewSize);
}

Inline int& DynamicArrayOfInteger::FetchElement(int Index)
```

```
{
   if(Index < 0 || Index >= Size)
   {
        Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
   }

   return(Array[Index]);
}

// Presentation Services
Inline int& DynamicArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Services
Inline void DynamicArrayOfInteger::CheckSize(int ProposedSize)
{
   if(ProposedSize <= 0)
   {
        Error("Size [%d] must be a strictly positive Integer",
                ProposedSize);
   }
}

Inline void DynamicArrayOfInteger::SetSize(int ProposedSize)
{
   assert(ProposedSize > 0);

   Size = ProposedSize;
}

Inline void DynamicArrayOfInteger::BuildArray()
{
   SetArray(AllocateArray(Size));

   assert(Size > 0);

   InitializeArray(Array,Size);
}

Inline void DynamicArrayOfInteger::DeleteArray()
{
   assert(Array != 0);

   delete Array;
   Array = 0;
}

Inline void DynamicArrayOfInteger::PrintElement(int Index)
{
   assert(Index >= 0);

   printf("Array[%d] = %d0,Index,Array[Index]);
}

Inline int* DynamicArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   assert(Size > 0);

   Memory = new int[Size];
   if(Memory == 0)
   {
        Error("Cannot allocate [%d] int's in your program",Size);
   }
   return Memory;
}

Inline void DynamicArrayOfInteger::CopyPartialArray(int* Source,
   int* Destination, int Limit, int End)
{
   assert(Limit > 0);
   assert(End > 0);
   assert(Limit <= End);
   assert(Source != 0);
   assert(Destination != 0);

   for(int Index = 0;Index < Limit;Index++)
```

```
   {
        Destination[Index] = Source[Index];
   }
   for(;Index < End; Index++)
   {
        Destination[Index] = 0;
   }
}

Inline void DynamicArrayOfInteger::SetArray(int* NewArray)
{
   assert(Array == 0);
   assert(NewArray != 0);

   Array = NewArray;
}

Inline
void DynamicArrayOfInteger::InitializeArray(int* ThatArray,int ThatSize)
{
   for(int Index = 0; Index < ThatSize; Index++)
   {
        ThatArray[Index] = 0;
   }
}

Inline void DynamicArrayOfInteger::Clear()
{
   Array = 0;
   Size = 0;
}
```

Solution exercice no. 26 :

Le mécanisme consiste à monter un flag supplémentaire qualifiant le tableau d'extensible.

```
// valeurs possibles du flag
enum ExtendMode
{
   DenyTransparentExtend,
   AllowTransparentExtend
};

// Definition du tableau extensible
class DynamicArrayOfInteger
{
   ExtendMode Mode;
   int LowerIndex;
   int Size;
   int* Array;

   // Internal Routines
   void LowResize(int Extremum);
   void HighResize(int Extremum);

public:
   // Standard Services
   DynamicArrayOfInteger(int Size,
                ExtendMode Mode = DenyTransparentExtend);
   ~DynamicArrayOfInteger();

   // Specific Services
   void Print();

   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);
};

// Standard Services
Inline DynamicArrayOfInteger::DynamicArrayOfInteger(int StartSize,
        ExtendMode AskedMode)
{
   if(StartSize <= 0)
   {
        Error("Size [%d] must be a strictly positive Integer",StartSize);
   }
```

```
   Size = StartSize;
   Array = new int[Size];
   if(Array == 0)
   {
        Error("Cannot allocate [%d] int's in your program",Size);
   }
   for(int Index = 0; Index < Size; Index++)
   {
        Array[Index] = 0;
   }
   Mode = AskedMode;
   LowerIndex = 0;
}

Inline DynamicArrayOfInteger::~DynamicArrayOfInteger()
{
   delete Array;
}

// Specific Services
Inline void DynamicArrayOfInteger::Print()
{
   for(int Index = LowerIndex; Index < LowerIndex+Size; Index++)
   {
        printf("Array[%d] = %d0,Index,Array[Index-LowerIndex]);
   }
   printf("0);
}

Inline void DynamicArrayOfInteger::LowResize(int NewExtremum)
{
   int* NewArray;
   int NewSize;
   int Increment;

   Increment = -NewExtremum + LowerIndex;
   NewSize = Size + Increment;

   if((NewArray = new int[NewSize]) == 0)
   {
        Error("Cannot allocate [%d] int's in your program",NewSize);
   }
   for(int Index = 0;Index < Increment; Index++)
   {
        NewArray[Index] = 0;
   }
   for(;Index < NewSize;Index++)
   {
        NewArray[Index] = Array[Index-Increment];
   }
   delete Array;
   Array = NewArray;
   Size = NewSize;
   LowerIndex = NewExtremum;
}

Inline void DynamicArrayOfInteger::HighResize(int NewExtremum)
{
   int* NewArray;
   int NewSize;
   int Increment;

   Increment = NewExtremum - (LowerIndex + Size);
   NewSize = Size + Increment;

   if((NewArray = new int[NewSize]) == 0)
   {
        Error("Cannot allocate [%d] int's in your program",NewSize);
   }
   for(int Index = 0;Index < Size;Index++)
   {
        NewArray[Index] = Array[Index];
   }
   for(;Index < NewSize; Index++)
   {
        NewArray[Index] = 0;
   }
   delete Array;
   Array = NewArray;
   Size = NewSize;
```

```
   LowerIndex = NewExtremum;
}

Inline int& DynamicArrayOfInteger::FetchElement(int Index)
{
   if((Index < LowerIndex || Index >= LowerIndex+Size)
   && (Mode != AllowTransparentExtend))
   {
        Error("Extend at [%d] not allowed on this array",Index);
   }
   if(Index < LowerIndex)
   {
        LowResize(Index);
   }
   if(Index >= LowerIndex+Size)
   {
        HighResize(Index);
   }
   return(Array[Index-LowerIndex]);
}

// Presentation Services
Inline int& DynamicArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}
```

Solution exercice no. 27 :

Il s'agit ici de redéfinir les opérateurs appliqués aux `chaine`s. Une solution est donnée par la définition de type suivante :

```
class chaine {
private :
        char * str;
public :
        chaine();
        chaine(char * s);
        chaine operator + (const chaine & s);
        chaine operator + (const int i);
        chaine operator -= (const char c);
        void pr();
};

// constructeur de chaine sans parametre
chainë:chaine()
{
        str = new char [1];
}

// constructeur de chaine avec parametre
chainë:chaine(char * s)
{
        str = new char [strlen(s) + 1];
        strcpy(str,s);
}

// concatener deux chaines
chaine chainë:operator + (const chaine & s)
{
        char * ptr = new char[strlen(str) + strlen(s.str) + 1];
        strcpy(ptr,str);
        strcat(ptr,s.str);

        chaine ReturnValue(ptr);
        delete ptr;
        return ReturnValue;
}

// dupliquer une chaine
chaine chainë:operator + (const int i)
{
        char * ptr = new char[strlen(str) * 2 + 1];
        for( int j = 0; j < i ; j++ )
        {
                strcat(ptr,str);
        }
```

```
        chaine ReturnValue(ptr);
        delete ptr;
        return ReturnValue;
}

// oter toute occurence du parametre
chaine chainë:operator -= (const char c)
{
        char * ptr = new char[strlen(str) + 1];
        for( char *p = str, *q = ptr; *p ; p++ )
        {
                if( *p != c )
                {
                        *q++ = *p;
                }
        }
        *q = ' ';
        delete str;
        str = ptr;

        return chaine(str);
}

// affichage de chaine
void chainë:pr()
{
        printf("str=%s0,str);
}
```

Solution exercice no. 28 :

Voici une idée de programmation permettant un traitement "complet" de chaînes de caractères. Le programme propose l'utilisation de fonctionalités illustrées par le code suivant :

```
main()
{
   String a;
   String b("Hello");
   String c = "World";
   String d = b;
   String e('x');
   String f = 'c';
   char u;
   int i;

   // Copy from C string
   a = "Bonjour";
   a.Print();

   // Copy from String
   a = b;
   a.Print();


   // Catenate from C string
   a = b + "world";
   a.Print();

   // Catenate from String
   a = b + c;
   a.Print();

   // Catenate from String and C string
   a = b + " " + c;
   a.Print();

   // Catenate C string to myself
   a += " and Monde";
   a.Print();

   // Catenate String to myself
   a += b;
   a.Print();

   // Extract sub string
   a = b(2,3);
   a.Print();
```

```
    // Left Shift of String
    a = b << 3;
    a.Print();

    // Left Shift of a C string
    a = (String)"World" << 3;
    a.Print();

    // another way
    a = String("World") << 3;
    a.Print();

    // Self Left Shift
    c <<= 3;
    c.Print();

    // Right Shift
    a = b >> 3;
    a.Print();

    // Right Shift on a C string, note the Print
    (String("Salut Monde") >> 4).Print();

    // Self Shift Right
    a >>= 3;
    a.Print();

    // discrete access on a char as a left value
    a[3] = 'x';
    a.Print();

    // discrete access on a char as a right value, note the Print
    u = a[3];
    String(u).Print();

    // Compare Strings
    a = "Hello";
    if(a == b)
    {
         a.Print(); String(" == ").Print(); b.Print();
    }
    else if(a < b)
    {
         a.Print(); String(" < ").Print(); b.Print();
    }
    else
    {
         a.Print(); String(" > ").Print(); b.Print();
    }

    // compare Strings and C string
    if(a == "Hello")
    {
         a.Print(); String("== Hello").Print();
    }

    // but compare C string with Strings
    if((String)"Hello" == a)
    {
         String("Hello is equal to ").Print(); a.Print();
    }

    // Convert an integer to a String (like sprintf())
    a = 10;
    a.Print();

    // Convert a String to an integer (like atoi())
    i = a;
    printf("%d0,i);

    // Well ...
    (a += ((b << 3) + "Salut") + (String(((String)123456)[3]) >> 10)).Print();

}
```

Deux solutions sont proposées pour la définition de la classe `String` répondant à ces spécifications.

Première solution :

```
class String
{
   int Length;
   char* Text;

public:
   String();
   String(char Character);
   String(char* Initializer);
   String(String& Initializer);
   String(int Integer);
   ~String();

   String  operator+(String& Other);
   String  operator()(int ExtractOffset,int ExtractSize);
   String  operator<<(int ShiftSize);
   String  operator>>(int ShiftSize);

   String& operator=(String& Source);
   String& operator+=(String& Other);
   String& operator<<=(int ShiftSize);
   String& operator>>=(int ShiftSize);

   int operator<(String& Other);
   int operator<=(String& Other);
   int operator==(String& Other);
   int operator>=(String& Other);
   int operator>(String& Other);
   int operator!=(String& Other);

   operator int();

   char& operator[](int Index);

   void Print();
};

Inline String::String()
{
   Length = 0;
   Text = new char[Length+1];
}

Inline String::String(char Initializer)
{
   Length = 1;
   Text = new char[2];
   Text[0] = Initializer;
   Text[1] = ' ';
}

Inline String::String(char* Initializer)
{
   Length = strlen(Initializer);
   Text = new char[Length+1];
   strcpy(Text,Initializer);
}

Inline String::String(String& Initializer)
{
   Length = Initializer.Length;
   Text = new char[Length+1];
   strcpy(Text,Initializer.Text);
}

Inline String::String(int Integer)
{
   Text = new char[20];
   sprintf(Text,"%d",Integer);
   Length = strlen(Text);
}

Inline String::~String()
{
   delete Text;
}

Inline String& String::operator=(String& Source)
{
   delete Text;
```

```
   Length = Source.Length;
   Text = new char[Length+1];
   strcpy(Text,Source.Text);
   return *this;
}

Inline String String::operator+(String& Other)
{
   String NewString;

   delete NewString.Text;
   NewString.Length = Length + Other.Length;
   NewString.Text = new char[NewString.Length+1];
   strcpy(NewString.Text,Text);
   strcat(NewString.Text,Other.Text);
   return NewString;
}

Inline String& String::operator+=(String& Other)
{
   *this = *this + Other;
   return *this;
}

Inline String String::operator()(int ExtractOffset,int ExtractSize)
{
   String NewString;

   if(ExtractOffset >= Length)
   {
        NewString = "";
   }
   else
   {
        if((ExtractOffset + ExtractSize) >= Length)
        {
           ExtractSize = Length - ExtractOffset;
        }
        char* SubText = new char[ExtractSize+1];
        strncpy(SubText,Text + ExtractOffset,ExtractSize);
        SubText[ExtractSize] = ' ';
        NewString = SubText;
        delete SubText;
   }
   return NewString;
}

Inline String String::operator<<(int ShiftSize)
{
   String NewString;

   NewString.Length = Length - ShiftSize;
   if(NewString.Length < 0)
   {
        NewString.Length = 0;
   }
   delete NewString.Text;
   NewString.Text = new char[NewString.Length+1];
   strncpy(NewString.Text,Text+ShiftSize,NewString.Length);
   NewString.Text[NewString.Length] = ' ';
   return NewString;
}

Inline String& String::operator<<=(int ShiftSize)
{
   *this = *this << ShiftSize;
}

Inline String String::operator>>(int ShiftSize)
{
   String NewString;

   NewString.Length = Length + ShiftSize;
   delete NewString.Text;
   NewString.Text = new char[NewString.Length+1];
   for(char* Pointer = NewString.Text;Pointer < NewString.Text+ShiftSize;
        Pointer++)
   {
        *Pointer = ' ';
   }
```

```
   strcpy(NewString.Text+ShiftSize,Text);
   return NewString;
}

Inline String& String::operator>>=(int ShiftSize)
{
   *this = *this >> ShiftSize;
}

Inline char& String::operator[](int Index)
{
   if(Index < 0 || Index >= Length)
   {
        Error("Illegal String Index");
   }
   else
   {
        return Text[Index];
   }
}

Inline int String::operator<(String& Other)
{
   return (strcmp(Text,Other.Text) < 0);
}

Inline int String::operator<=(String& Other)
{
   return (strcmp(Text,Other.Text) <= 0);
}

Inline int String::operator==(String& Other)
{
   return (strcmp(Text,Other.Text) == 0);
}

Inline int String::operator>=(String& Other)
{
   return (strcmp(Text,Other.Text) >= 0);
}

Inline int String::operator>(String& Other)
{
   return (strcmp(Text,Other.Text) > 0);
}

Inline int String::operator!=(String& Other)
{
   return (strcmp(Text,Other.Text) != 0);
}

Inline String::operator int()
{
   return atoi(Text);
}

Inline void String::Print()
{
   printf("%s0,Text);
}
```

Seconde solution :

```
class String
{
   int Length;
   char* Text;

   // Specific Services
   String  Duplicate(String& Source);
   String  Catenate(String& Other);
   char&  FetchCharacter(int Index);
   String  LeftShift(int ShiftSize);
   String  RightShift(int ShiftSize);
   String  ExtractSubString(int ExtractOffset,int ExtractSize);
   String& SelfAffect(String& Source);
   String& SelfAppend(String& Other);
   String& SelfLeftShift(int ShiftSize);
   String& SelfRightShift(int ShiftSize);
   int    StrictlyBefore(String& Other);
```

```
   int    BeforeOrEqual(String& Other);
   int    Equal(String& Other);
   int    AfterOrEqual(String& Other);
   int    StrictlyAfter(String& Other);
   int    NotEqual(String& Other);
   int    ConvertToInteger();

public:
   // Standard Services
   String();
   String(char Character);
   String(char* Initializer);
   String(String& Initializer);
   String(int Integer);
   ~String();

   void   Print();

   // Presentation Services
   String  operator+(String& Other);
   String  operator()(int ExtractOffset,int ExtractSize);
   String  operator<<(int ShiftSize);
   String  operator>>(int ShiftSize);

   String& operator=(String& Source);
   String& operator+=(String& Other);
   String& operator<<=(int ShiftSize);
   String& operator>>=(int ShiftSize);

   int operator<(String& Other);
   int operator<=(String& Other);
   int operator==(String& Other);
   int operator>=(String& Other);
   int operator>(String& Other);
   int operator!=(String& Other);

   char& operator[](int Index);
   operator int();


};

// Standard Services
Inline String::String()
{
   Length = 0;
   Text = new char[Length+1];
}

Inline String::String(char Initializer)
{
   Length = 1;
   Text = new char[2];
   Text[0] = Initializer;
   Text[1] = ' ';
}

Inline String::String(char* Initializer)
{
   Length = strlen(Initializer);
   Text = new char[Length+1];
   strcpy(Text,Initializer);
}

Inline String::String(String& Initializer)
{
   Length = Initializer.Length;
   Text = new char[Length+1];
   strcpy(Text,Initializer.Text);
}

Inline String::String(int Integer)
{
   Text = new char[20];
   sprintf(Text,"%d",Integer);
   Length = strlen(Text);
}

Inline String::~String()
{
```

```
   delete Text;
}

// Specific Services
Inline String String::Duplicate(String& Source)
{
   String NewString(Source);
   return NewString;
}

Inline String String::Catenate(String& Other)
{
   String NewString;

   delete NewString.Text;
   NewString.Length = Length + Other.Length;
   NewString.Text = new char[NewString.Length+1];
   strcpy(NewString.Text,Text);
   strcat(NewString.Text,Other.Text);
   return NewString;
}

Inline char& String::FetchCharacter(int Index)
{
   if(Index < 0 || Index >= Length)
   {
        Error("Illegal String Index");
   }
   else
   {
        return Text[Index];
   }
}

Inline String String::LeftShift(int ShiftSize)
{
   String NewString;

   NewString.Length = Length - ShiftSize;
   if(NewString.Length < 0)
   {
        NewString.Length = 0;
   }
   delete NewString.Text;
   NewString.Text = new char[NewString.Length+1];
   strncpy(NewString.Text,Text+ShiftSize,NewString.Length);
   NewString.Text[NewString.Length] = ' ';
   return NewString;
}

Inline String String::RightShift(int ShiftSize)
{
   String NewString;

   NewString.Length = Length + ShiftSize;
   delete NewString.Text;
   NewString.Text = new char[NewString.Length+1];
   for(char* Pointer = NewString.Text;
        Pointer < NewString.Text+ShiftSize;
        Pointer++)
   {
        *Pointer = ' ';
   }
   strcpy(NewString.Text+ShiftSize,Text);
   return NewString;
}

Inline
String String::ExtractSubString(int ExtractOffset,int ExtractSize)
{
   String NewString;

   if(ExtractOffset >= Length)
   {
        NewString = "";
   }
   else
   {
        if((ExtractOffset + ExtractSize) >= Length)
        {
```

```
            ExtractSize = Length - ExtractOffset;
         }
         char* SubText = new char[ExtractSize+1];
         strncpy(SubText,Text + ExtractOffset,ExtractSize);
         SubText[ExtractSize] = ' ';
         NewString = SubText;
         delete SubText;
   }
   return NewString;
}

Inline String& String::SelfAffect(String& Source)
{
   delete Text;
   Length = Source.Length;
   Text = new char[Length+1];
   strcpy(Text,Source.Text);
   return *this;
}

Inline String& String::SelfAppend(String& Other)
{
   SelfAffect(Catenate(Other));
   return *this;
}

Inline String& String::SelfLeftShift(int ShiftSize)
{
   SelfAffect(LeftShift(ShiftSize));
   return *this;
}

Inline String& String::SelfRightShift(int ShiftSize)
{
   SelfAffect(RightShift(ShiftSize));
   return *this;
}

Inline int String::StrictlyBefore(String& Other)
{
   return (strcmp(Text,Other.Text) < 0);
}

Inline int String::BeforeOrEqual(String& Other)
{
   return (strcmp(Text,Other.Text) <= 0);
}

Inline int String::Equal(String& Other)
{
   return (strcmp(Text,Other.Text) == 0);
}

Inline int String::AfterOrEqual(String& Other)
{
   return (strcmp(Text,Other.Text) >= 0);
}

Inline int String::StrictlyAfter(String& Other)
{
   return (strcmp(Text,Other.Text) > 0);
}

Inline int String::NotEqual(String& Other)
{
   return (strcmp(Text,Other.Text) != 0);
}

Inline int String::ConvertToInteger()
{
   return atoi(Text);
}

Inline void String::Print()
{
   printf("%s0,Text);
}

// Presentation Services
Inline String String::operator+(String& Other)
```

```
{
   return Catenate(Other);
}

Inline String String::operator()(int ExtractOffset,int ExtractSize)
{
   return ExtractSubString(ExtractOffset,ExtractSize);
}

Inline String String::operator<<(int ShiftSize)
{
   return LeftShift(ShiftSize);
}

Inline String String::operator>>(int ShiftSize)
{
   return RightShift(ShiftSize);
}

Inline String& String::operator=(String& Source)
{
   return SelfAffect(Source);
}

Inline String& String::operator+=(String& Other)
{
   return SelfAppend(Other);
}

Inline String& String::operator<<=(int ShiftSize)
{
   return SelfLeftShift(ShiftSize);
}

Inline String& String::operator>>=(int ShiftSize)
{
   return SelfRightShift(ShiftSize);
}

Inline char& String::operator[](int Index)
{
   return FetchCharacter(Index);
}

Inline int String::operator<(String& Other)
{
   return StrictlyBefore(Other);
}

Inline int String::operator<=(String& Other)
{
   return BeforeOrEqual(Other);
}

Inline int String::operator==(String& Other)
{
   return Equal(Other);
}

Inline int String::operator>=(String& Other)
{
   return AfterOrEqual(Other);
}

Inline int String::operator>(String& Other)
{
   return StrictlyAfter(Other);
}

Inline int String::operator!=(String& Other)
{
   return NotEqual(Other);
}

Inline String::operator int()
{
   return ConvertToInteger();
}
```

Solution exercice no. 29 :

La pile est stockée dans un espace alloué dynamiquement lors de la déclaration. La classe doit présenter les services répondant aux fonction de pile.

```
// Definition de la classe pile d'entiers
class StackOfInteger
{
   int* Array;
   int* Top;
   int* Ceiling;

public:
   // Standard Services
   StackOfInteger(int RequestedDepth);
   ~StackOfInteger();

   // Specific Services
   int Push(int Value);
   int Pop();
   int Flush();
   int Empty();
   int NonEmpty();
   int Full();
   int NonFull();
   void Print();
};

// Standard Services
StackOfInteger::StackOfInteger(int Depth)
{
   Array = new int[Depth];
   if(Array == 0)
   {
        Error("Cannot allocate Stack array");
   }
   Ceiling = Array + Depth;
   Top = Array;
}

StackOfInteger::~StackOfInteger()
{
   delete Array;
}

// Specific Services
int StackOfInteger::Push(int Value)
{
   if(Full())
   {
        Error("Stack Full");
   }
   *Top++ = Value;
   return Value;
}

int StackOfInteger::Pop()
{
   if(Empty())
   {
        Error("Stack Empty");
   }
   return *--Top;
}

int StackOfInteger::Flush()
{
   Top = Array;
}

int StackOfInteger::Empty()
{
   return (Top == Array);
}

int StackOfInteger::NonEmpty()
{
   return !Empty();
}
```

```
int StackOfInteger::Full()
{
   return (Top >= Ceiling);
}

int StackOfInteger::NonFull()
{
   return !Full();
}

void StackOfInteger::Print()
{
   for(int* Scan = Array; Scan < Top; Scan++)
   {
        printf("%7d ",*Scan);
   }
   printf(".0);
}
```

Solution exercice no. 30 :

On utilise maintenant le concept d'arbre de classe. La classe pile est constituée d'un tableau d'entiers sur lequel on applique les fonctionalités de pile.

```
// Definition de la classe tableau d'entiers
class ArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   int* AllocateArray(int Size);
   void FreeArray(int* Array);
   void SetUpArray(int Size,int Initializer);
   void InitializeArray(int Size,int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,
                  int SecondArgument);
   int  CheckIndex(int Index);
   int  CheckSize(int RequestedSize);
   int  SetUpSize(int RequestedSize);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);

public:
   // Standard Services
   ArrayOfInteger(int RequestedSize);
   ~ArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);
   int GetSize();
};

// Standard Services
Inline ArrayOfInteger::ArrayOfInteger(int RequestedSize)
{
   SetUpSize(RequestedSize);
   SetUpArray(RequestedSize,0);
}

Inline ArrayOfInteger::~ArrayOfInteger()
{
   FreeArray(Array);
}

// Specific Services
Inline void ArrayOfInteger::Print()
{
   Print("%d");
}
```

```
Inline void ArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        PrintElement(Scanner,Format);
   }
}

Inline int& ArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
        Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
   }
   return *(Array + Index);
}

// Presentation Services
Inline int& ArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines
Inline int* ArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   if((Memory = new int[Size]) == 0)
   {
        Error("Unable to allocate [%d] int's in your program",Size);
   }
   return Memory;
}

Inline void ArrayOfInteger::FreeArray(int* Array)
{
   delete Array;
}

Inline void ArrayOfInteger::SetUpArray(int Size,int Initializer)
{
   Array = AllocateArray(Size);
   InitializeArray(Size,Initializer);
}

Inline void ArrayOfInteger::InitializeArray(int Size,int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        InitializeElement(Scanner,Initializer);
   }
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,
                                  int Argument,
                                  int AnotherOne)
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= 0 && Index < Size);
```

```
}

Inline int ArrayOfInteger::CheckSize(int Size)
{
   return (Size > 0);
}

Inline int ArrayOfInteger::SetUpSize(int RequestedSize)
{
   if(!CheckSize(RequestedSize))
   {
        Error("Size [%d] is not a strictly positive integer",
                RequestedSize);
   }
   Size = RequestedSize;
}

Inline
void ArrayOfInteger::InitializeElement(int* Element,const int Value)
{
   *Element = Value;
}

Inline
void ArrayOfInteger::PrintElement(int* Element,const char* Format)
{
   printf(Format,*Element);
   printf("0);
}

Inline int ArrayOfInteger::GetSize()
{
   return Size;
}

// Definition de la classe pile
class StackOfInteger
{
   ArrayOfInteger Array;
   int TopIndex;

public:
   // Standard Services
   StackOfInteger(int RequestedDepth);
   ~StackOfInteger();

   // Specific Services
   int Push(int Value);
   int Pop();
   int Flush();
   int Empty();
   int NonEmpty();
   int Full();
   int NonFull();
   void Print();
};

// Standard Services
StackOfInteger::StackOfInteger(int Depth):Array(Depth)
{
   TopIndex = 0;
}

StackOfInteger::~StackOfInteger()
{
   ;
}

// Specific Services
int StackOfInteger::Push(int Value)
{
   if(Full())
   {
        Error("Stack Full");
   }
   Array[TopIndex++] = Value;
   return Value;
}

int StackOfInteger::Pop()
```

```
{
   if(Empty())
   {
        Error("Stack Empty");
   }
   return Array[--TopIndex];
}

int StackOfInteger::Flush()
{
   TopIndex = 0;
}

int StackOfInteger::Empty()
{
   return (TopIndex == 0);
}

int StackOfInteger::NonEmpty()
{
   return !Empty();
}

int StackOfInteger::Full()
{
   return (TopIndex >= Array.GetSize());
}

int StackOfInteger::NonFull()
{
   return !Full();
}

void StackOfInteger::Print()
{
   Array.Print();
   printf(".0);
}
```

Solution exercice no. 31 :

La solution proposée redéfinit les opérateurs d'affectation etd'incrémentation, ainsi que de cast pour donner une utilisation plus "c-iste" de l'objet pile. La définition suivante :

```
// Definition de la classe Pile d'entiers
class StackOfInteger
{
   int* Array;
   int* Top;
   int* Ceiling;

public:
   // Standard Services
   StackOfInteger(int RequestedDepth);
   ~StackOfInteger();

   // Specific Services
   int Push(int Value);
   int Pop();
   int Flush();
   int Empty();
   int NonEmpty();
   int Full();
   int NonFull();
   void Print();

   // Presentation Services
   int  operator=(int Value){ return Push(Value); };
      operator int()    { return Pop(); };
   void operator--()     { Pop(); };
   int  operator*()      { return Full(); };
   int  operator!()      { return Empty(); };
   int  operator~()      { return Flush(); };
};

// Standard Services
StackOfInteger::StackOfInteger(int Depth)
```

```
{
   Array = new int[Depth];
   if(Array == 0)
   {
        Error("Cannot allocate Stack array");
   }
   Ceiling = Array + Depth;
   Top = Array;
}

StackOfInteger::~StackOfInteger()
{
   delete Array;
}

// Specific Services
int StackOfInteger::Push(int Value)
{
   if(Full())
   {
        Error("Stack Full");
   }
   *Top++ = Value;
   return Value;
}

int StackOfInteger::Pop()
{
   if(Empty())
   {
        Error("Stack Empty");
   }
   return *--Top;
}

int StackOfInteger::Flush()
{
   Top = Array;
}

int StackOfInteger::Empty()
{
   return (Top == Array);
}

int StackOfInteger::NonEmpty()
{
   return !Empty();
}

int StackOfInteger::Full()
{
   return (Top >= Ceiling);
}

int StackOfInteger::NonFull()
{
   return !Full();
}

void StackOfInteger::Print()
{
   for(int* Scan = Array; Scan < Top; Scan++)
   {
        printf("%7d ",*Scan);
   }
   printf(".0);
}
```

correspond au programme suivant :

```
main()
{
   StackOfInteger s(100); // 100 is stacks depth
   int i;

   // Push
   s = 10;
   s.Print();
```

```
   s = 20;
   s.Print();

   s = 30;
   s.Print();

   // Pop into an integer
   i = s;
   s.Print();

   // Pop without affect
   s--;

   // Push(Pop())
   s = (int)s;
   s.Print();

   // not Empty
   if(! !s)
   {
        --s;
   }
   s.Print();

   // Flush
   ~s;
   s.Print();

   while(! *s)
   {
        s = rand();
   }
   s.Print();

   while(!!s)
   {
        --s;
   }
   s.Print();
}
```

Solution exercice no. 32 :

La classe tableau étendu hérite de la classe tableau d'entiers.

```
// fonctions de traitement d'erreur
Inline void Error(const char* Message)
{
   printf(Message);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument)
{
   printf(Message,Argument);
   printf("0);
   exit(1);
}

Inline void Error(const char* Message,int Argument,int AnotherOne)
{
   printf(Message,Argument,AnotherOne);
   printf("0);
   exit(1);
}

// Definition de la classe de base (tableau d'entiers)
class ArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   int* AllocateArray(int Size);
   void FreeArray(int* Array);
```

```
   void SetUpArray(int Size,int Initializer);
   void InitializeArray(int Size,int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,
               int SecondArgument);
   int  CheckIndex(int Index);
   int  CheckSize(int RequestedSize);
   int  SetUpSize(int RequestedSize);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);

public:
   // Standard Services
   ArrayOfInteger(int RequestedSize);
   ~ArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);

};

// Standard Services
Inline ArrayOfInteger::ArrayOfInteger(int RequestedSize)
{
   SetUpSize(RequestedSize);
   SetUpArray(RequestedSize,0);
}

Inline ArrayOfInteger::~ArrayOfInteger()
{
   FreeArray(Array);
}

// Specific Services
Inline void ArrayOfInteger::Print()
{
   Print("%d");
}

Inline void ArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        PrintElement(Scanner,Format);
   }
}

Inline int& ArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
        Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
   }
   return *(Array + Index);
}

// Presentation Services
Inline int& ArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines
Inline int* ArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   if((Memory = new int[Size]) == 0)
   {
        Error("Unable to allocate [%d] int's in your program",Size);
   }
```

```
   return Memory;
}

Inline void ArrayOfInteger::FreeArray(int* Array)
{
   delete Array;
}

Inline void ArrayOfInteger::SetUpArray(int Size,int Initializer)
{
   Array = AllocateArray(Size);
   InitializeArray(Size,Initializer);
}

Inline void ArrayOfInteger::InitializeArray(int Size,int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        InitializeElement(Scanner,Initializer);
   }
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,
                                  int Argument,
                                  int AnotherOne)
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= 0 && Index < Size);
}

Inline int ArrayOfInteger::CheckSize(int Size)
{
   return (Size > 0);
}

Inline int ArrayOfInteger::SetUpSize(int RequestedSize)
{
   if(!CheckSize(RequestedSize))
   {
        Error("Size [%d] is not a strictly positive integer",
                RequestedSize);
   }
   Size = RequestedSize;
}

Inline
void ArrayOfInteger::InitializeElement(int* Element,const int Value)
{
   *Element = Value;
}

Inline
void ArrayOfInteger::PrintElement(int* Element,const char* Format)
{
   printf(Format,*Element);
   printf("0);
}

// Definition de la classe derivee (tableau etendu)
class ExtendedArrayOfInteger:public ArrayOfInteger
{
   int LowerIndex;
```

```
   // Internal Services
   int MapVirtualIndexToPhysicalIndex(int Index);

public:
   // Standard Services
   ExtendedArrayOfInteger(int RequestedLowerIndex,
                          int RequestedHigherIndex);
   ~ExtendedArrayOfInteger();

   // Presentation Services
   int& operator[](int Index);
};

// Standard Services
Inline ExtendedArrayOfInteger::ExtendedArrayOfInteger(
        int RequestedLowerIndex,int RequestedHigherIndex)
        :ArrayOfInteger(RequestedHigherIndex-RequestedLowerIndex+1)
{
   LowerIndex = RequestedLowerIndex;
}

Inline ExtendedArrayOfInteger::~ExtendedArrayOfInteger()
{
   ;
}

// Presentation Services
Inline int& ExtendedArrayOfInteger::operator[](int Index)
{
   return FetchElement(MapVirtualIndexToPhysicalIndex(Index));
}

// Internal Services
Inline int ExtendedArrayOfInteger::MapVirtualIndexToPhysicalIndex(int Index)
{
   return (Index - LowerIndex);
}
```

Solution exercice no. 33 :

La classe pile va hériter de la classe tableau d’entiers. Elle "rajoute" les fonctionalités spécifiques de traitement de pile.

```
// Definition de la classe de base (Tableau d’entiers)
class ArrayOfInteger
{
   int Size;
   int* Array;

   // Internal Routines
   int* AllocateArray(int Size);
   void FreeArray(int* Array);
   void SetUpArray(int Size,int Initializer);
   void InitializeArray(int Size,int Initializer);
   void Error(const char* ErrorMessage);
   void Error(const char* ErrorMessage,int FirstArgument);
   void Error(const char* ErrorMessage,int FirstArgument,
              int SecondArgument);
   int  CheckIndex(int Index);
   int  CheckSize(int RequestedSize);
   int  SetUpSize(int RequestedSize);
   void InitializeElement(int* Element,const int Value);
   void PrintElement(int* Element,const char* Format);

public:
   // Standard Services
   ArrayOfInteger(int RequestedSize);
   ~ArrayOfInteger();

   // Specific Services
   void Print();
   void Print(const char* Format);
   int& FetchElement(int Index);

   // Presentation Services
   int& operator[](int Index);
```

```
   // Added here for Stack from Exercice02 Version
   int GetSize();
};

// Standard Services
Inline ArrayOfInteger::ArrayOfInteger(int RequestedSize)
{
   SetUpSize(RequestedSize);
   SetUpArray(RequestedSize,0);
}

Inline ArrayOfInteger::~ArrayOfInteger()
{
   FreeArray(Array);
}

// Specific Services
Inline void ArrayOfInteger::Print()
{
   Print("%d");
}

Inline void ArrayOfInteger::Print(const char* Format)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        PrintElement(Scanner,Format);
   }
}

Inline int& ArrayOfInteger::FetchElement(int Index)
{
   if(!CheckIndex(Index))
   {
        Error("Index [%d] is out of allowed range [0,%d]",Index,Size-1);
   }
   return *(Array + Index);
}

// Presentation Services
Inline int& ArrayOfInteger::operator[](int Index)
{
   return FetchElement(Index);
}

// Internal Routines
Inline int* ArrayOfInteger::AllocateArray(int Size)
{
   int* Memory;

   if((Memory = new int[Size]) == 0)
   {
        Error("Unable to allocate [%d] int's in your program",Size);
   }
   return Memory;
}

Inline void ArrayOfInteger::FreeArray(int* Array)
{
   delete Array;
}

Inline void ArrayOfInteger::SetUpArray(int Size,int Initializer)
{
   Array = AllocateArray(Size);
   InitializeArray(Size,Initializer);
}

Inline void ArrayOfInteger::InitializeArray(int Size,int Initializer)
{
   int* Scanner;
   int* End;

   for(Scanner = Array,End = Array + Size;Scanner < End;Scanner++)
   {
        InitializeElement(Scanner,Initializer);
   }
```

```
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage)
{
   ::Error(ErrorMessage);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,int Argument)
{
   ::Error(ErrorMessage,Argument);
}

Inline void ArrayOfInteger::Error(const char* ErrorMessage,
                                  int Argument,
                                  int AnotherOne)
{
   ::Error(ErrorMessage,Argument,AnotherOne);
}

Inline int ArrayOfInteger::CheckIndex(int Index)
{
   return (Index >= 0 && Index < Size);
}

Inline int ArrayOfInteger::CheckSize(int Size)
{
   return (Size > 0);
}

Inline int ArrayOfInteger::SetUpSize(int RequestedSize)
{
   if(!CheckSize(RequestedSize))
   {
        Error("Size [%d] is not a strictly positive integer",
                RequestedSize);
   }
   Size = RequestedSize;
}

Inline void ArrayOfInteger::InitializeElement(int* Element,const int Value)
{
   *Element = Value;
}

Inline void ArrayOfInteger::PrintElement(int* Element,const char* Format)
{
   printf(Format,*Element);
   printf("0);
}

// Added here for Stack from Exercice02 Version
Inline int ArrayOfInteger::GetSize()
{
   return Size;
}

// Definition de la classe derivee (Pile)
class StackOfInteger : public ArrayOfInteger
{
   int TopIndex;

public:
   // Standard Services
   StackOfInteger(int RequestedDepth);
   ~StackOfInteger();

   // Specific Services
   int Push(int Value);
   int Pop();
   int Flush();
   int Empty();
   int NonEmpty();
   int Full();
   int NonFull();
};

// Standard Services
StackOfInteger::StackOfInteger(int Depth):ArrayOfInteger(Depth)
{
   TopIndex = 0;
```

```
}

StackOfInteger::~StackOfInteger()
{
   ;
}

// Specific Services
int StackOfInteger::Push(int Value)
{
   if(Full())
   {
        ::Error("Stack Full");
   }
   FetchElement(TopIndex++) = Value;
   return Value;
}

int StackOfInteger::Pop()
{
   if(Empty())
   {
        ::Error("Stack Empty");
   }
   return FetchElement(--TopIndex);
}

int StackOfInteger::Flush()
{
   TopIndex = 0;
}

int StackOfInteger::Empty()
{
   return (TopIndex == 0);
}

int StackOfInteger::NonEmpty()
{
   return !Empty();
}

int StackOfInteger::Full()
{
   return (TopIndex >= GetSize());
}

int StackOfInteger::NonFull()
{
   return !Full();
}
```

BIBLIOGRAPHIE

,

.

AFCET,1975
Groupe Algol de L' AFCET, P. Bacchus, J. Andre', C. Pair, *Manuel du langage algorithmique Algol 68*, Hermann, Paris, 1975.

AFCET,1972
Groupe Algol de l' AFCET, J. Buffet, P. Arnal, A. Que're', *De'finition du langage algorithmique Algol 68*, Hermann, Paris, 1972.

Agha,1986
G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems agha86*, MIT Press, 1986.

Berthet,1971
C. Berthet, *Le langage PL/1*, Dunod, Paris, 1971.

Booch,1987a
G. Booch, *Software Engineering with ADA*, Benjamin Cummings, Redwood City, California, 1987.

Booch,1987b
G. Booch, *Software Components with ADA*, Benjamin Cummings, Redwood City, California, 1987.

Booch,1991
G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, Redwood City, California, 1991.

Brooks,1985
R. A. Brooks, *Programming in Common Lisp*, John Wiley, New-York, 1985.

Clocksin,1981
W. Clocksin, C. Mellish, *Programming in Prolog*, Springer Verlag, New-York, 1981.

Dahl,1970
O. Dahl, B. Myrhaug, K. Nygaard, *Simula Common Base Language*, Norwegian Computing Center S-22, Oslo (Norway), 1970.

Ellis,1991
Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Man*, pp. 343-345, Addison-Wesley, 1991.

Goldberg,1983
A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Goldberg,1976
A. Goldberg, ed., A. Kay, ed., *Smalltalk-72 Instruction Manual*, Xerox Parc Technical Report, 1976.

Gondran,1979
M. Gondran, M. Minoux, *Graphes et algorithmes*, Collection de la Direction des Etudes et Recherches d'Electricite' de France, 37, Eyrolles, 1979.

Graham,1991
I. Graham, *Object oriented Methods*, Addison-Wesley, 1991.

Greussay,1986
P. Greussay, ‘‘Commandes de Vlisp-C’’, Rapport interne .], Departement informatique, Universite de Paris 8, 1986.

Johnson,1975
S. C. Johnson, ‘‘Yacc: Yet another compiler compiler’’, *Computing Science Technical Report*, no. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

Laurens,1987
R. Laurens, E. Hincapie, ‘‘Quelques proble‘mes en programmation concurrente et architectures systoliques, avec le langage Occam’’, Me’moire de Maiˆtrise Informatique .], Departement informatique, Universite de Paris 8, 1987.

Lesbros,1989
V. Lesbros, ‘‘Lexique Smalltalk’’, Rapport interne .], Departement informatique, Universite de Paris 8, 1989.

Lesk,1975
M.E. Lesk, ‘‘Lex - A lexical analyser generator’’, *Computing Science Technical Report*, no. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

Masini,1989
G. Masini, A. Napoli, D. Le’onard, D. Colnet, K. Tombre, *Les langages objets*, Inter Edition, Paris, 1989.

May,1984
M. D. May, R. J. Taylor, *Occam, microprocessor and microsystems* , 1984.

Meyer,1988
B. Meyer, *Object-oriented Software Construction*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Meyer,1978
B. Meyer, C. Baudoin, *Me’thodes de programmation*, Collection de la Direction des Etudes et Recherches d’Electricite’ de France, Eyrolles, Paris, 1978.

Naur,1960
P. Naur, ‘‘Rapport sur le langage algorithmique Algol 60 (traduction)’’, *Revue de l’ AFCET*, vol. Chiffres 3 .], pp. 1-44, 1960.

Naur,1963
P. Naur, ‘‘Revised report on the algorithmic language Algol 60’’, *Annual review in automatic programming*, vol. 4 .], pp. 217-599, 1963.

Richards,1980
M. Richards, C. Whitby-Strevens, *BCPL - The language and its compiler*, Cambridge University Press, 1980.

Ritchie,1978
D. M. Ritchie, B. W. Kernighan, *The C programming language*, Prentice-Hall, 1978.

Ritchie,1974
D. M. Ritchie, K. Thompson, ‘‘The Unix time-sharing system’’, *Communications of the ACM*, vol. 17, no. 7, pp. 365-, 1974.

Sinz,1984
P. M. Sinz, ‘‘Le class preprocesseur : implantation de facilite de manipulation de types abstraits pour le langage C’’, Rapport interne .], Departement informatique, Universite de Paris 8, 1984.

Sites,1972
R. L. Sites, ‘‘Algol W reference manual’’, Report STAN-CS-71-230, Computer Science Department, Stanford University (California), 1972.

 ISBN 2-909909-02-6

Steele,1984
G. L. Steele, *Common Lisp : The language*, Digital Press, Burlington, Massachusetts, 1984.

Stroustrup,1983
B. Stroustrup, ‘‘Adding classes to C : an exercise in language evolution’’, *Software practice & experience*, vol. 13 .], pp. 623-628, 1983.

Stroustrup,1984
B. Stroustrup, ‘‘Data abstraction in C’’, *AT&T Bell Laboratories Technical Journal*, vol. 63 .], no. 8, pp. 1701-1732, 1984.

Stroustrup,1986
Bjarne Stroustrup, ‘‘An Overview of ’’, *ACM Sigplan Notices on Programing Languages*, vol. 21, no. 10, pp. 7-18, 1986.

Stroustrup,1987a
Bjarne Stroustrup, ‘‘What is Object-Oriented Programming ?’’, in *Proc. 1rst European Conference on Object-Oriented Programming*, ed. Springer Verlag Lecture Notes in Computer Science, vol. 176, pp. 51, Paris, 1987.

Stroustrup,1987b
Bjarne Stroustrup, ‘‘Multiple Inheritance for C++’’, *Proc. EUUG Spring’87 Conference*, Helsinki, 1987.

Stroustrup,1987c
Bjarne Stroustrup, ‘‘The Evolution of C++ : 1985 to 1987’’, *Usenix Proceedings and Additional Papers C++ Workshop*, pp. 1, Santa-Fe, 1987.

Stroustrup,1988
Bjarne Stroustrup, ‘‘Parameterized Types for C++’’, *Usenix C++ Conference - Denver, October 1988*, pp. 1, Denver, 1988.

Stroustrup,1991
Bjarne Stroustrup, *The C++ programming language*, second edition, Addison-Wesley, 1991.

Wertz,1987
H. Wertz, ‘‘Smalltalk-80 : virtual Image version v12.2 Ref. Guide’’, Rapport interne .], Departement informatique, Universite de Paris 8, 1987.

Wertz,1988
H. Wertz, ‘‘Introduction a‘ la programmation oriente’e objet’’, Rapport interne .], Departement informatique, Universite de Paris 8, 1988.

Wertz,1989
H. Wertz, *Common Lisp: une introduction a‘ la programmation*, Masson, Paris, 1989.

Wijngaarden,1969
A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, *Report on the Algorithmic language Algol 68*, Springer Verlag, Berlin, 1969.

Wijngaarden,1974
A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, ‘‘Revised report on the algorithmic language Algol 68’’, in *Supplement to Algol Bulletin*, Department of Computer Science, University of Alberta, Edmonton, Alberta (Canada), 1974.

Winston,1984
P. H. Winston, B. K. P. Horn, *Lisp, second edition*, Addison-Wesley, Reading, Massachusetts, 1984.

Wirth,1971
N. Wirth, ‘‘The programming language Pascal’’, in *Acta Informatica*, *1, 1*, pp. 35, Springer Verlag, Berlin, 1971.

X3-Secretariat,1980
X3-Secretariat, ‘‘Standard - The C language’’, X3J11/90-013, Computer and Business Equipment Manufactures Association, Washington DC (USA), 1980.

 ISBN 2-909909-02-6

LEXIQUE

1. *paradigme* : du grec *paradeigma*; exemple, modèle, maquette, préfiguration, plan d'architecte
2. *encapsulation* : encapsulage, protection des données, limitation de domaine de reconnaissance
3. *design* : conception, schématisation, mise en forme, organisation, dessin
4. *working stack* : pile de travail
5. *record* : enregistrement
6. *right-value* : représentant d'une valeur, résultat de l'évaluation d'une expression
7. *left-value* : représentant d'un récipiant
8. *cpu* : Central Process Unit, unité centrale
9. *one-to-one instruction/delay* : une instruction par cycle
10. *hacking* : technique de programmation consistant à "cracher" le code, sans analyse écrite préalable. Si le résultat est bon, on le garde, sinon, on jette tout et on recommence à zéro.
11. *encapsulage* : protection des données
12. *implémentation* : implantation
13. *package* : module, ensemble de fonctions regroupées car afférentes à un même sujet
14. *recouvrement* : rechercher les solutions à partir d'outils déjà existants
15. *message-passing* : communication par transmission de messages
16. *code natif* : code machine, code binaire correspondant au processeur utilisé
17. *run-time* : programme compilé traduisant et exécutant du p-code
18. *full-compiler* : compilateur au sens complet du terme. Tout doit être défini à la compilation.
19. *link* : édition de liens
20. *XDR* : External Data Representation - modèle de représentation unique de données. Sert lors de la transmission de messages. Les informations sont traduites sous un format connu de l'émetteur et du récepteur. Chacun d'eux est alors à même de traduire ce format en un format local connu de lui seul.
21. *trap* : interruption software
22. *prototypage* : définition de l'entête de fonction
23. *overload* : redéfinition, "surcharge"
24. *inlining* : macroïsation de fonction, traduction sous forme de macro
25. *macroïsation* : mise sous forme de macro
26. *instance de classe* : représentant d'une classe; élément alloué ayant pour type celui défini par cette classe. Toute déclaration d'un objet de type classe génère une instance de cette classe.
27. *spooler* : gestion de file d'attente
28. *template* : type paramétré
29. *générique* : classe de base pour un polymorphisme
30. *debuggable* : que l'on peut debugger (en trouver les erreurs)
31. *bêta-release* : version en dernière phase de tests

32. *News* : réseau international de diffusion d'informations
33. *release* : version
34. *synopsis* : définition de l'en-tête d'une fonction, mode d'utilisation
35. *clash* : erreur, conflit
36. *cast* : conversion de type
37. *scope* : espace, domaine (mot qui n'a pas de traduction explicite en français !)
38. *overloadable* : que l'on peut overloader (redéfinir dynamiquement)
39. *well-known service* : service connu de tous, ressource, fonctionalité mise en commun.
40. *overloading* : action d'overload (redéfinission autorisée)
41. *inlinée* : vue sous forme de macro
42. *debug* : recherche des erreurs dans un programme
43. *frame* : espace de travail
44. *stack* : pile de travail
45. *slash* : barre de division
46. *new-line* : retour à la ligne
47. *caster* : faire un cast, appliquer une conversion de type
48. *scope de variable* : domaine de définition et de reconnaissance d'une variable, d'un objet, d'une fonction.
49. *buddies* : couple de jumeaux (!); intraduisible littéralement. Technique algorithmique particulière pour l'allocation d'espace mémoire
50. *type-checker* : procéder à un contrôle sur le type
51. *pid* : process identificator
52. *smart* : sympathique, agréable
53. *public access* : en accès publiç en libre accès; visible à partir d'un élément défini hors du domaine;
54. *hide* : caché, masqué, rendu invisible
55. *bug* : bogue (pas de chataîgne), anciennement "vermine"; erreur
56. *container* : récipient
57. *return value* : valeur de retour d'une fonction
58. *trick* : truç bidouille
59. *look* : aspect, forme
60. *bss* : blank storage section ("tas").
61. *common* : ce terme vient du Fortran, et spécifie une section commune de donnés (données partagées)
62. *cludge* : stupide, idiot, sans intelligence
63. *remapper* : mettre en vis-à-vis
64. *classe générique* : classe servant de base pour le polymorphisme (utilisation du qualificatif `virtual` pour les fonctions membres); qui sert de base de définition.

 ISBN 2-909909-02-6

65. *classe abstraite* : classe générique n’ayant que des fonctions membres qualifiées de `virtual`, sans code associé, et sans membres données.
66. *template* : mot clé du C++ permettant de définir une liste de paramètre dans la définition d’une classe générique ou d’une fonction générique (certaines traduction françaises parlent de classe "patron" et de fonction "patron")

TABLE DES MATIERES

Liste des Figures