Introduction au C++

Kévin Santugini-Repiquet

Introduction au C++

Kévin Santugini

1er septembre 2011

Ce document constitue une “courte introduction” au langage C++, langage dérivé du `C` et conçu par Bjarne Stroustrup dans les années 80. Le `C++` a été standardisé officiellement en 1998. Un nouveau standard a été adopté en 2011. Dans ce polycopié, nous traitons le standard 1998, le standard 2011 est encore trop récent et les compilateurs ne sont pas encore prêts.

Il n'est pas strictement nécessaire de connaître le `C` pour apprendre le `C++`. Cependant, une certaine familiarité avec le `C` et en particulier avec sa syntaxe « déconcertante » de déclaration de types peut être utile. En effet, on ne programme pas en `C++` comme on programme en `C`. Le moindre programme en `C`, un tant soit peu complexe, réclame l'utilisation manuelle de l'allocation dynamique, même pour des opérations courantes comme la gestion de chaînes de caractères. Ceci peut être très souvent évité en `C++`.

Le langage `C++` tout en étant compatible à 99% avec le `C` ajoute énormément de fonctionnalités. Le `C` est avant tout un langage procédural. Le `C++` est un langage supportant.

1. La programmation procédurale.
2. La programmation modulaire.
3. La programmation orientée objet.
4. La programmation générique.

Certains principes de base ont guidé la conception du `C++` :

- Aussi compatible que possible avec le `C` mais pas plus.
- « zero-overhead principle » : les fonctionnalités non utilisées ne doivent pas induire de perte de performance.
- « le programmeur sait ce qu'il fait », principe hérité du `C` qui veut que l'on ne restreigne pas excessivement ce que le programmeur a le droit de faire. Aussi connu comme « Si le langage empêche le programmeur de faire quelque chose de stupide, il l'empêche aussi de faire quelque chose de très intelligent ».

Table des matières

1 Généralités sur la programmation **7**

1.1 Du fichier source à l'exécutable . . . 7
1.1.1 La compilation . . . 7
1.1.2 L'interprétation . . . 7
1.1.3 Langages interprétés et langages compilés . . . 7
1.2 Le typage . . . 8
1.2.1 Qu'est ce que le typage ? . . . 8
1.2.2 Langages non typés . . . 8
1.2.3 Typage statique et typage dynamique . . . 8
1.2.4 Typage explicite et typage implicite . . . 9
1.2.5 Typage et `C++` . . . 9
1.3 La compilation séparée et l'édition de liens . . . 9
1.4 Dans quelle catégorie se situe le `C++` . . . 9

2 Les bases en `C++` **11**

2.1 Mes premiers programmes en C++ . . . 11
2.1.1 Un programme qui ne fait rien ou presque . . . 11
2.1.2 Le programme Bonjour . . . 11
2.2 Types prédéfinis, variables . . . 12
2.2.1 Déclaration . . . 13
2.2.2 L'assignation . . . 14
2.3 Fonctions . . . 15
2.3.1 Définition et appel de fonctions . . . 15
2.3.2 Valeur de retour et **return** . . . 15
2.3.3 Le cas particulier de la fonction `main` . . . 16
2.3.4 Passage des arguments par valeurs ou par références . . . 17
2.3.5 Surdéfinition des fonctions . . . 18
2.3.6 Prototypes de fonctions . . . 18
2.4 Expressions . . . 19
2.4.1 Expressions arithmétiques . . . 19
2.4.2 Expressions relationnelles . . . 20
2.4.3 Expressions logiques . . . 20
2.4.4 Les assignations . . . 21
2.5 Les structures de contrôle . . . 21
2.5.1 L'instruction if . . . 21
2.5.2 Le switch . . . 23
2.5.3 La boucle **while** . . . 24

2.5.4 La boucle **do while** . . . 24
2.5.5 La boucle **for** . . . 25
2.6 Conclusion . . . 25

3 Les variables **26**
3.1 Les pointeurs . . . 26
3.2 Les références . . . 27
3.3 Les structures . . . 27
3.4 Pointeurs sur des fonctions . . . 28
3.5 Déclarations compliquées . . . 29
3.6 Les typedefs . . . 30
3.7 Durée de vie d'une variable . . . 30
3.7.1 Variables globales . . . 30
3.7.2 Variables locales . . . 31
3.7.3 Variables locales statiques . . . 32
3.7.4 Les variables dynamiques . . . 32

4 Les conteneurs en C++ : la STL **34**
4.1 Les tableaux . . . 34
4.2 Les listes . . . 36
4.3 Les arbres . . . 37
4.4 Conclusion . . . 37

5 Programmation modulaire et organisation d'un programme **38**
5.1 Un court exemple . . . 38
5.1.1 Un code désorganisé . . . 38
5.1.2 Séparation en fonctions . . . 39
5.1.3 Séparation en plusieurs fichiers . . . 40
5.2 Les headers . . . 41
5.2.1 Généralités sur les headers . . . 41
5.2.2 Création d'un header . . . 42
5.2.3 Que mettre dans un header ? . . . 43
5.3 Quelques règles de bonne conduite . . . 43
5.3.1 Placer le `main` à part . . . 44
5.3.2 Organiser vos fichiers . . . 44
5.4 Les namespaces . . . 44
5.4.1 Le namespace anonyme . . . 44
5.4.2 Les namespaces non anonymes . . . 44
5.5 Conclusion . . . 45

6 Les classes et la programmation orientée objet **46**
6.1 Les classes . . . 46
6.1.1 L'encapsulation . . . 46
6.1.2 Définition d'une classe . . . 47
6.1.3 Fonctions membres . . . 47

6.1.4 Le pointeur **this** . . . 49
6.1.5 Fonctions et classes amies . . . 49
6.1.6 Constructeurs . . . 51
6.1.7 Destructeurs . . . 54
6.1.8 Exemple : une classe de complexes . . . 55
6.1.9 Surdéfinition des opérateurs . . . 56
6.1.10 Assignation et constructeurs . . . 58
6.1.11 Conclusion sur les classes . . . 58
6.2 La programmation orientée objet . . . 58
6.2.1 L'héritage . . . 59
6.2.2 Pointeurs et conversions implicites ? . . . 60
6.2.3 Héritage public ou privé ? . . . 60
6.2.4 Constructeurs et héritage . . . 60
6.2.5 Les fonctions virtuelles . . . 61
6.2.6 Un exemple de programmation objet . . . 62
6.3 Conclusion . . . 63

7 Les templates et la programmation générique 64
7.1 Un exemple simple . . . 64
7.2 Les classes templates . . . 65
7.3 Spécialisation et surdéfinition des fonctions template . . . 66
7.4 La programmation générique : l'exemple de la STL . . . 67
7.5 Conclusion . . . 68

8 Conseils et conclusion 69

1 Généralités sur la programmation

Pour programmer, il est utile de connaître quelques généralités trop peu souvent expliquées car « allant de soi ». Leur méconnaissance peut donner lieu à de désagréables surprises.

1.1 Du fichier source à l'exécutable

Les programmes informatiques sont écrits dans un langage informatique. La plupart de ces langages ne sont pas compréhensible directement par la machine [1]. Pour pouvoir utiliser le programme, l'exécuter, il faut soit le compiler soit l'interpréter.

1.1.1 La compilation

La compilation est l'opération qui consiste à traduire un fichier source, *i.e.* un fichier contenant le programme dans un format humainement lisible, en un exécutable qui lui, contient le programme sous une forme directement utilisable par la machine. Cette opération est effectuée par un compilateur.

Une fois créé l'exécutable, le compilateur n'est plus nécessaire au déroulement du programme.

1.1.2 L'interprétation

Une autre possibilité d'exécuter un programme est de l'interpréter. Un interpréteur exécute le programme au moment même où il lit le source. Contrairement à ce qui se passe pour la compilation, l'interpréteur est nécessaire chaque fois que l'on souhaite utiliser le programme. Il est plus performant de compiler un programme que de l'interpréter.

1.1.3 Langages interprétés et langages compilés

Rien n'empêche en théorie de créer à la fois un interpréteur et un compilateur pour un même langage. Cependant, dans la pratique, la plupart des langages sont soit utilisés avec un compilateur soit avec un interpréteur. C'est pourquoi on parle le plus souvent de langages compilés et de langages interprétés.

Le C++ est un langage compilé. Nous nous concentrerons sur les langages compilés pour le reste de ce chapitre. En particulier, nous nous concentrerons sur cette question, primordiale en `C++` **qu'est-ce qui doit-être connu au moment de la compilation et qu'est ce qui peut attendre l'exécution ?**

1. Il est possible d'écrire un exécutable directement avec un éditeur hexadécimal mais ce n'est guère recommandé.

1.2 Le typage

1.2.1 Qu'est ce que le typage ?

Imaginons deux variables x et y flottants stockées sur 32 bits ainsi que deux variables entières m et n elles aussi stockées sur 32 bits. Supposons que les représentations binaires de x et de n soient identiques et que les représentations binaires de y et de m soient aussi identiques alors il n'est pas vrai en général que les représentations binaires de $x + y$ et de $m + n$ soient aussi identiques. L'addition flottante et l'addition entière sont des opérations différentes sur la machine. Cependant, dans la plupart des langages, ces opérations sont représentées par le même opérateur +. C'est au moment de la compilation, suivant le type de chaque variable que le compilateur décide de traduire l'opération + par l'opération machine appropriée.

Outre cette facilité de programmation, un autre intérêt du typage est de permettre au compilateur de repérer un certain nombre d'erreurs : un compilateur signalera des opérations entre des types incompatibles. Par exemple, un compilateur détectera et vous avertira si par mégarde vous avez programmer une opération n'ayant aucun sens comme la division d'un entier par une chaîne de caractères.

Des exemples courants de types sont

- booléen
- nombre entier
- nombre flottant
- caractère
- chaîne de caractères

La liste est loin d'être exhaustive. Il serait d'ailleurs illusoire d'essayer d'en établir une.

1.2.2 Langages non typés

Dans un langage non typé, c'est au programmeur de se rappeler ce que représente chaque variable, et même de combien d'octets une variable occupe. Parmi les langages non typés, on compte l'assembleur.

1.2.3 Typage statique et typage dynamique

Dans un langage compilé, si le typage est dynamique : chaque variable contient au moment de l'exécution son type de manière explicite. Cela signifie qu'il est possible de modifier son type en cours d'exécution et que le type d'une variable n'a pas à être connu au moment de la compilation.

Si le typage est statique, toute information sur le typage aura disparu après la compilation. À chaque ligne du code source, il est impératif que **le type d'une variable soit connu au moment de la compilation**. On ne peut pas changer le type d'une variable en cours d'exécution dans un langage statiquement typé.

Cette dernière condition apparemment anodine rend la programmation dans un langage statiquement typé plus contraignante. En contrepartie, un langage statiquement typé sera plus performant.

1.2.4 Typage explicite et typage implicite

Suivant le langage, le typage peut être explicite ou implicite. Dans un langage implicite, le type d'une variable peut être déduit lors de sa première apparition à gauche du signe assignation. C'est très courant pour les langages interprétés et beaucoup moins pour les langages langages compilés [2].

1.2.5 Typage et C++

Le C++ est un langage statiquement typé. Cela rend possible une meilleure performance lors de l'exécution mais en contrepartie les langages statiquement typés peuvent être plus délicats à programmer.

1.3 La compilation séparée et l'édition de liens

Un grand programme sera avantageusement écrit en plusieurs fichiers. Si un compilateur devait relire chaque fichier source avant de créer l'exécutable chaque fois que l'on modifie un seul fichier source, la **compilation** serait une opération prohibitivement coûteuse. Heureusement, certains langages supportent ce qu'on appelle la compilation séparée : chaque fichier source peut être compilé séparément en un fichier objet. Une seconde opération appelée **édition de liens** permet alors de regrouper tous les fichiers objets en un seul exécutable. Voir figure 1.1. L'édition de lien est une opération beaucoup moins coûteuse que la compilation. Il est donc beaucoup plus rapide d'éditer les liens de plusieurs fichiers objets que de compiler l'ensemble des fichiers sources d'un coup.

1.4 Dans quelle catégorie se situe le C++

Comme son prédecesseur le C, le C++ est un langage compilé statiquement typé. Le typage est explicite : chaque variable doit être préalablement déclarée avant toute utilisation. Le typage est cependant rendu plus souple par l'existence de nombreuses conversions implicites entre types. Il est par exemple possible d'ajouter un entier et un flottant sans convertir explicitemet l'entier en flottant.

2. En FORTRAN, le typage est implicite par défaut, je ne connais pas d'autres exemples de langages compilés avec un typage implicite.

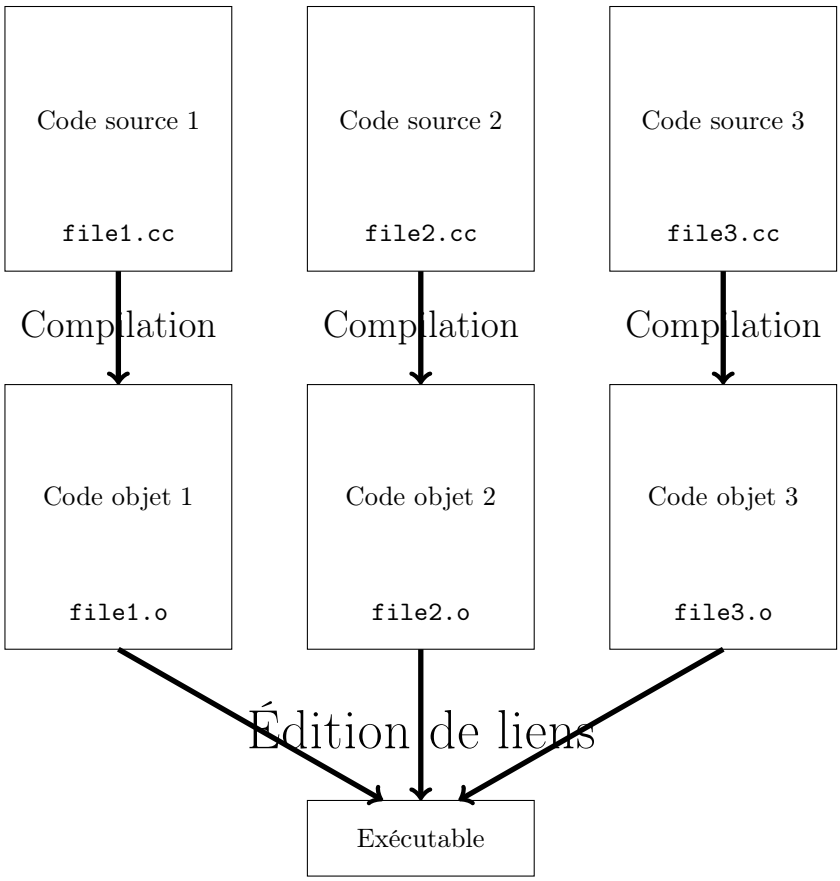

FIGURE 1.1: Du source à l'exécutable : compilation séparée

2 Les bases en C++

Le langage C++ est constitué de deux parties : le langage proprement dit « core » et sa bibliothèque standard. Un principe de départ du C++ était de ne pas alourdir le langage « core » et de déléguer au maximum les fonctionnalités à la bibliothèque standard. En particulier, le C++ ne définit pas un type « chaîne de caractères » [1] : mais sa bibliothèque standard en fournit une.

2.1 Mes premiers programmes en C++

Nous allons commencer par donner quelques exemples de programmes écrits en C++.

2.1.1 Un programme qui ne fait rien ou presque

```
/*Mon premier programme*/
int main ()
{
  return(0); //L'unique instruction du programme
}
```

Ceci définit un programme ne faisant rien et qui retourne 0. En voici l'analyse détaillée.

Tout se qui se trouve entre les caractères /* et */ est ignoré par le compilateur et constitue donc un commentaire [2]. Il en va de même pour tout ce qui se trouve entre un « // » et un retour à la ligne.

La ligne **int** `main` () déclare une fonction `main` retournant un entier (**int**) et ne prenant aucun argument : la liste des arguments qui se trouve entre la parenthèse ouvrante et la parenthèse fermante est vide. Le corps de la fonction `main` est située entre { et }. Il contient pour seule instruction **return**(0). L'instruction **return** est utilisée pour spécifier la valeur de retour, *i.e* le résultat, de la fonction.

La fonction `main` est toujours la première fonction appelée dans un programme C++. Dans un programme C++, il doit toujours y avoir une et une seule fonction appelée `main`. C'est ce qu'on appelle le point d'entrée du programme. *A contrario*, il ne doit jamais y avoir de fonction `main` dans une bibliothèque.

2.1.2 Le programme Bonjour

Donnons un exemple très légèrement plus compliqué.

1. Si on ne compte pas le **char*** hérité du C.

2. Pour la lisibilité d'un programme, il est important de commenter son code. Cependant, il ne faut pas prendre les commentaires de ce polycopié comme des exemples de bon commentaires pour un code réel. Ceux de ce polycopié ont pour but d'expliquer la signification d'une fonctionnalité du C++. Dans un code réel, il ne faut pas mettre de commentaires qui expliquent ce qui est évident en regardant le code. Par exemple, `x`=0; *//x vaut maintenant 0* ne servirait à rien dans un code réel.

```
/* Mon premier programme qui fait quelque chose */
#include <iostream>
int main ()
{
  std::cout << "Bonjour\n";
  return(0);
}
```

Ce programme sort « Bonjour » sur la sortie standard qui est la sortie à l'écran en mode texte.

À la première ligne du programme, on observe l'instruction #**include**<iostream>. Moralement [3], cette ligne sert à charger les fonctions d'entrée sorties de la bibliothèque standard : en effet, il n'y a pas d'instruction d'entrée sortie dans le langage cœur du C++.

Le signe « \n » est le code pour indiquer un retour à la ligne dans une chaîne de caractères. Le guillemet double " marque à la fois le début et la fin d'une chaîne de caractères. La variable std::cout est déclarée dans le fichier iostream [4] : elle correspond à la sortie standard. Le << est l'opérateur de sortie [5]. Par exemple,

```
/* Exemple d'utilisation de l'opérateur d'output */
#include <iostream>
int main ()
{
  std::cout << "2*3=" << 6 << "\n";
  return(0);
}
```

Lors de l'exécution, le programme sort :

2*3=6

Pour des programmes plus compliqués, nous aurons besoin d'utiliser des variables.

2.2 Types prédéfinis, variables

Comme dans presque tout langage, le C++ permet de définir des variables et de modifier leur valeur au cours du programme.

Voici un exemple simple de programme où intervient des variables :

```
#include <iostream>
int main()
{
  int d;
  std::cin >> d;
```

3. Cette directive #**include** est en réalité très primitive, il ne s'agit ni plus ni moins que d'une inclusion textuelle caractère par caractère du fichier iostream effectuée par le préprocesseur avant la compilation proprement dite. Un mécanisme moins primitif, dit de modules, est prévu dans un addendum au C++ 2011.

4. iostream est un header

5. il s'agit là d'une addition de la bibliothèque standard, dans le langage C++ « core », les opérateurs << et >> sont des opérateurs de « bit shifting ».

```
  std::cout << 2*d;
  return(0);
}
```

On déclare une variable entière puis on lit dans cette variable la valeur de l'entier entré au clavier (entrée standard). Enfin, on imprime à l'écran (sortie standard) le double de cette valeur. L'opérateur >> est symétrique de l'opérateur <<. Le premier est utilisé pour l'entrée des données à partir d'un flux (ici `std::cin`), le second représente l'envoi de donnée dans un flux (ici `std::cout`).

2.2.1 Déclaration

En C++, les variables doivent être explicitement déclarées avant leur première utilisation. Voici quelques déclarations simples de variables.

```
int n;                  // n : entier signé
long m;                 // m :  entier long signé
short s;                // s : entier court signé
unsigned int u;         // u : entier non signé
unsigned long ul;       // ul : entier long non signé
unsigned short us;      // us : entier court non signé

signed char sc          // sc : caractère signé
unsigned char sc        // sc : caractère non signé
char c                  // c  : caractère, on ne sait
                        // pas s'il est signé ou non.

float a;        // a : nombre flottant simple précision
double b;       // b : nombre flottant double précision
long double x;  // x : nombre flottant de précision
                //     au moins double

bool l; // l est un booléen
```

Il s'agit là de tous les types simples prédéfinis en C++.

On peut aussi initialiser les variables au moment de leur déclaration :

```
bool b1= true;
bool b2= false;

int a = 0;
int b = a;

char c ='a';    // 'a' est un caractère;
                // "a" est une chaîne de caractère.

float x=1.0;
float y= 1.4e-2;
```

true, **false**, 1.0, 0, 'a' et 1.4e−2 sont ce que l'on appelle des littéraux : *i.e.* une suite de caractères qui représente une valeur fixe d'un type donné. En C++98, un littéral représente forcément un type prédéfini.

Voici un autre programme légèrement plus utile

```
#include <iostream>
#include <string>
int main ()
{
  std::string mon_prenom;
  std::cout << "Entrez votre prénom\n";
  std::cin >> mon_prenom;
  std::cout << "Bonjour, " << mon_prenom << "\n";
}
```

Ici, nous introduisons un autre type std::string qui est le type de chaîne de caractères défini par la bibliothèque standard. Ce n'est pas un type prédéfini du C++. Le préfixe std:: indique que le type string à utiliser est celui défini dans la bibliothèque standard [6].

Vous vous demandez peut-être quel est le type du littéral chaîne de caractère "Bonjour". En effet, cela ne peut être std::string car ce dernier type n'est pas un type de base mais un type défini par la bibliothèque standard. Le type de "Bonjour" est un **char***, pointeur [7] sur un caractère. Mais peu importe car la conversion vers std::string est implicite. Et les opérations sur std::string sont beaucoup plus faciles à utiliser que les vieilles fonctions sur les **char*** héritées du C.

2.2.2 L'assignation

Après qu'une variable a été déclarée, on peut lui assigner d'autres valeurs. Le signe pour l'assignation en C++ est le =. Supposons que les variables a et b aient préalablement été déclarées comme entières.

```
a=0.0; // a vaut maintenant 0.0
b=1.0; // b vaut maintenant 1.0
b=a;   //b vaut maintenant 1.0
```

On peut assigner un entier à un flottant directement. La conversion est implicite (il ne s'agit pas d'une assignation bit par bit) :

```
float x=0.0;
int a = 2;
x=a; //x vaut maintenant 2.0
```

6. std est un namespace. C'est celui réservé pour la bibliothèque standard. Nous reparlerons des namespaces plus loin.

7. Ce pointeur pointe vers une région de la mémoire qui contient le premier caractère de la chaîne, les caractères suivants de la chaîne se trouvent derrière dans la mémoire. La fin d'une chaîne de caractères est indiqué par le caractère ayant la valeur 0 (et non '0').

2.3 Fonctions

Dans un programme réel, on ne met pas tout son code dans la fonction `main`, qui nous le rappelons constitue le point d'entrée du programme. Il est recommandé de séparer son code en fonctions.

2.3.1 Définition et appel de fonctions

Voici un exemple de fonction qui écrit « Bonjour » à l'écran :

```
void print_Bonjour(void)
{
  std::cout << "Bonjour\n";
}
```

Cette fonction ne retourne aucune valeur : c'est indiqué par le mot-clef **void** devant le nom de la fonction. Elle ne prend aussi aucun argument, c'est indiqué par le **void**[8] entre parenthèses.

On peut alors écrire une fonction `main` qui appelle cette fonction :

```
int main()
{
  print_Bonjour();// Appel de fonction indiqué par ()
  return(0);
}
```

La définition d'une fonction a la forme générale :

```
typeretour nomfonction (type1 arg1, type2 arg2)
{
  Corps de la fonction (instructions,...)
}
```

Le nombre d'arguments étant évidemment libre, ici 2 n'est qu'un exemple.

Pour appeler une fonction, il suffit d'écrire son nom puis de placer entre parenthèse les arguments :

```
/* A l'interieur d'une autre fonction */
  nomfonction(val1,val2); //nomfonction est appelé avec
                          //val1 et val2 comme arguments
/* Reste du code*/
```

Bien entendu, le compilateur va vérifier que `val1` et `val2` sont compatibles avec les types `type1` et `type2`.

2.3.2 Valeur de retour et return

Nous avons vu que la fonction `main` contenait une instruction **return**. À quoi correspond cette instruction ? C'est cette instruction qui spécifie la valeur de retour.

Certaines fonctions Voici un autre exemple de fonction prenant en argument un flottant double précision et retournant un double :

8. Le **void** entre les parenthèses n'est nullement nécessaire, **void** `print_Bonjour`() aurait eu la même signification en `C++` mais pas en `C`.

```
double square (double x)
{
  double machin = 1;
  machin = x*x;
  return(machin);
}
```

Le type situé sur la première lign avant le nom de la fonction est le type de retour, *i.e.*, le type du résultat de la fonction. Ce qu'il y a dans le **return** est la valeur de retour de la fonction. Ici, `machin` est la valeur retournée. Cette instruction est nécessaire pour que l'instruction

```
double y = square(x);
```

ait le résultat escompté. Nous aurions aussi pu définir la fonction de la manière suivante

```
double square (double x)
{
  return(x*x);
}
```

Les novices en programmation et en C++ confondent parfois deux opérations bien distinctes : il est primordial de bien distinguer le concept de retour d'une valeur et le concept d'affichage : le **return** n'affiche rien à l'écran, et les fonctions d'affichage de la section 2.1.2 ne retournent rien à la fonction appelante.

2.3.3 Le cas particulier de la fonction `main`

Par convention, la fonction `main` est le point d'entrée du programme. Cela signifie qu'au moment de l'exécution d'un programme, la fonction `main` est la première fonction à être appelée [9]. La fonction `main` doit avoir **int** comme type de retour. La fonction `main` peut prendre soit zéro argument, soit deux arguments. Nous ne considérerons [10] que le cas où la fonction `main` ne prend aucun argument en entrée. La fonction `main` a pour forme

```
int main()
{
/* Corps de la fonction */
return(0);
}
```

Vous vous demandez peut-être à quoi correspond la valeur de retour puisque la fonction `main` ne sera jamais appelé par une autre fonction. Quelle intérêt y a-t-il à retourner une valeur à la fonction appelante s'il n'y a pas de fonction appelante. La réponse est que cette valeur est retournée au système d'exploitation.

Pour la fonction `main` et **uniquement** pour la fonction `main`, la convention veut que la valeur de retour est un code d'erreur retournée par le programme au système d'exploitation. La valeur 0

9. En toute rigueur, les variables globales sont initialisées avant l'appel à la fonction `main` mais laissons ce détail de côté.

10. L'étudiant intéressé par l'autre cas pourra effectuer une recherche sur `argc` et `argv`.

signifiant que le programme s'est arrêtée normalement sans erreur. Rappelons le, cette convention ne vaut que pour la fonction `main` et pour aucune autre fonction.

2.3.4 Passage des arguments par valeurs ou par références

Par défaut, les arguments d'une fonction sont passés par valeur :

```
void valeur(int b)
{
  b=5;
}

int main()
{
  int a=3;
  valeur(a);
  std::cout << a <<'\n';
}
```

Compilons ce programme et exécutons le. Le nombre imprimé est 3 ! La fonction valeur n'a reçu en argument qu'une copie de `a` et ne change que la valeur de cette copie. Dans la fonction `main`, la valeur de `a` ne change pas.

Il est cependant possible de passer les arguments par référence. Pour cela, il suffit de rajouter & dans le type de l'argument

```
void reference(int& b)
{
  b=5;
}

int main()
{
  int a=3;
  reference(a);
  std::cout << a <<'\n';
}
```

imprime 5 à l'écran car le passage a eu lieu par référence.

Pour des considérations d'efficacité, on peut aussi passer par référence de gros objets tout en déclarant que leur valeur ne doit pas être modifiée. Pour cela on rajoute le mot-clef **const** devant le type de l'argument. Cela permet au compilateur de mieux vérifier le code et aide au débogage.

```
void const_reference(const int& b)
{
  b=5;
}
```

provoque une erreur et un diagnostic à la compilation. Cette forme est surtout utilisée pour passer un gros objet telle une matrice ou un conteneur : le passage par référence coûtera un petit nombre d'octets : 4 octets sur une machine 32 bits, 8 sur une 64 bits. En comparaison, le passage par valeur imposerait de recopier toute la matrice ou tout le conteneur soit potentiellement plusieurs milliers d'entiers.

2.3.5 Surdéfinition des fonctions

Il est possible en C++ de surdéfinir les fonctions avec des listes d'arguments différentes :

```
int max(int, int);
double max(double, double);
```

déclare deux fonctions `max`. Le choix de la bonne fonction sera effectué par le compilateur :

```
max(3.0,5.0); //max(double,double)
max(3,5);     //max(int,int)
max(3,5.0);   //max(double,double)
```

Le troisième choix a lieu en raison des règles de conversions implicites. Elles sont compliquées mais donnent en général ce que l'on souhaite.

2.3.6 Prototypes de fonctions

Le C++ est un langage typé statiquement. Pour améliorer la correction d'erreurs, le C++ impose que toute fonction soit déclarée avant son premier appel. Il n'impose pas qu'elle soit déjà définie. La définition peut apparaître plus tard ou même se trouver dans un autre fichier.

En particulier, le programme suivant est malformée

```
#include<iostream>
/* La fonction main est le point d'entrée du programme */
int main()
{
  int n= max(1.0,2.0);//Erreur fonction max non déclarée à ce point
  return(0);
}

/*Définition de la fonction max */
int max(int a, int b)
{
  if(a<b) {
    return b;
  } else {
    return a;
  }
}
```

Pour remédier à ce problème, on peut soit placer la définition de la fonction `max` avant celle de la fonction `main` ou déclarer sans la définir la fonction `max` avant le `main`. Voici comment déclarer la fonction `max` :

```
int max(int, int);// Noter le point virgule
```

C'est ce que l'on appelle un prototype de fonction. Il déclare une fonction sans la définir. Il déclare ici qu'il existe une fonction `max` qui prend deux arguments de type **int** et qui retourne un **int**. Pour rendre bien formé le programme ci-dessus, il suffit de rajouter le prototype de la fonction `max` avant le `main`.

2.4 Expressions

Une expression est une instruction de type

```
2*a+5;
2+3*4-2;
a<=5;
a==b;//comparaison d'égalité
```

2.4.1 Expressions arithmétiques

On compte en C++ plusieurs opérations arithmétique. Il y en a cinq binaires $+$, $-$, $*$, $/$, et le modulo $\%$. Il y a aussi deux opérateurs préfixes unitaires, la négation $-$ et le $+$ qui ne fait rien pour les types prédéfinis. Voici quelques exemples :

```
-3+2*5;
1*2*3*4*5*6*7*8*9; // factorielle de 9
1+(2*4-5)*5-12;
-4*5+3;
```

Les expressions peuvent contenir des parenthèses et modifier l'ordre de priorité des opérations. Pour les opérations arithmétiques, les priorités sont les priorités mathématiques usuelles : les $+$ et le $-$ (négation) **unitaires** sont prioritaires sur les opérateurs binaires $*$ (multiplication), $/$ (division) et $\%$ (modulo) qui sont eux même prioritaires sur les opérateurs binaires $+$ (addition) et $-$ (soustraction). L'assignation $=$ a heureusement une priorité plus basse que celle de tous les opérateurs arithmétiques.

Les expressions arithmétiques sont très courantes dans le membre droit d'une assignation. Voici l'exemple d'une itération de l'algorithme de Newton pour la fonction $x \mapsto x * x - 3$

```
x=x-(x*x-3)/(2*x);
```

Remarquez qu'il n'y a pas d'opérateurs arithmétiques pour la puissance en C++. Il faut utiliser la fonction `std::pow` de la bibliothèque standard [11].

11. Pour pouvoir l'utiliser, il faut d'abord inclure le header <`cmath`>.

2.4.2 Expressions relationnelles

Les expressions relationnelles sont plutôt utilisées dans les conditions d'arrêt de boucle et dans les instructions **if** dont nous parlerons ultérieurement. En effet, ces opérateurs retournent un booléen (type **bool**) qui vaut soit **true** soit **false**.

Il y a six opérateurs relationnels binaires. Le test d'égalité [12] ==, et le test d'inégalité != sont les deux premiers.

```
5==5;        //vrai
5==6;        //faux
5!=6;        //vrai
5!=5;        //faux
int a = 4;
a==4;        //vrai
a!=4;        //faux
```

Quatre autres opérateurs relationnels existent : le test inférieur strict <, le test supérieur strict <, le test inférieur ou égale <= et le test supérieur ou égale, opérateur >=. Voici quelques exemples d'utilisation :

```
int a=3;
int b=4;
a<b;              //vrai
a>b;              //faux
a<=b;             //vrai
a>=b;             //faux
a<3;              //faux
a<=3;             //vrai
a>=3;             //vrai
```

L'exemple suivant imprime à l'écran si le nombre `a` est pair.

```
int a;
...
std::cout << ((a%2)==0) << '\n';
```

2.4.3 Expressions logiques

Il y a 2 opérateurs logiques binaires : le OU logique || et le ET logique, opérateur &&. Et il y a un seul opérateur logique unitaire préfixe : le NON logique, opérateur ! ; ce dernier est préfixe : `!a` vaut **true** si `a` vaut **false** et inversement. Voici quelque exemples :

```
true||false;       //true
true ||true;       //true
true&&false;       //false
!true;             //false
!(false||false);   //true
```

12. Ne pas confondre avec l'assignation =. C'est l'erreur classique du débutant.

2.4.4 Les assignations

Nous avons déjà vu le symbole d'assignation =. Il y en a d'autres liés aux opérations arithmétiques :

```
int a = 2; // a vaut 2
a+=4; // a vaut 6
a-=1; // a vaut 5
a*=8; // a vaut 40
a/=5; // a vaut 8
a/3;  // a vaut 2
```

Et enfin, il y a aussi les opérations ++ et −− qui incrémentent et décrémentent respectivement la valeur d'une variable.

```
int i=0;
i++;      //i vaut 1
++i;      //i vaut 2
--i;      //i vaut 1
--i;      //i vaut 0
```

La différence entre les opérateurs −−, ++ préfixes et postfixes est que pour les opérateurs préfixes, l'incrémentation a lieu avant l'assignation alors que pour les opérateurs postfixes, elle a lieu après.

```
int i=0;
int b=4;
i=--b;    //i vaut 3, b vaut 3
i=b--;    //i vaut 3, b vaut 2
i=b++;    //i vaut 2, b vaut 3
i=++b;    //i vaut 4, b vaut 4
```

2.5 Les structures de contrôle

Comme dans tout langage de programmation [13], C++ propose des structures de contrôle.

2.5.1 L'instruction if

L'instruction **if** est la première structure de contrôle que nous étudierons en C++. Elle nous permettra de créer des programmes plus intéressants.

L'instruction **if** s'emploie en général avec des expressions relationnels et des expressions logiques. La forme générale de la structure **if** est :

```
if(condition) {
  //Ce qui se passe si condition est vrai.
}
```

Voici un exemple :

13. À l'exception de l'assembleur

```
void simplesort(double& a, double& b) {
  if(b<a) {
    double tmp=a;
    a=b;
    b=a;
  }
}
```

Une instruction **if** peut aussi contenir des **else**.

```
if(condition) {
  //Ce qui se passe si condition est vrai.
} else {
  // Ce qui se passe sinon
}
```

et comme exemple une étape de dichotomie pour le sinus :

```
double a=3;
double b=3.5;
double x=(a+b)/2;
if(sin(a)*sin(x)<0) {
  x=b;
}
else {
  x=a;
}
x=(a+b)/2;
```

Naturellement, pour avoir une dichotomie complète, il est nécessaire de rajouter une boucle autour de ce code.

Nous pouvons aussi avoir des **else if**

```
if(condition1) {
  ...
}
else if(condition2) {
  ...
}
else if(condition3) {
  ...
}
else {
  ...
}
```

2.5.2 Le switch

Le switch est une forme plus efficace que **if else if else if else if**. Voici ce que l'on pourrait coder :

```
double dynamic_operation (char c, double a, double b)
{
  double x;
  if(c=='+') {
    x=a+b;
  }
  else if (c=='-') {
    x=a-b;
  }
  else if (c=='/') {
    x=a/b;
  }
  else if (c=='*') {
    x=a*b;
  }
  else {
    ...// Erreur
  }
  return x;
}
```

Il est cependant plus efficace d'utiliser un **switch** :

```
double dynamic_operation (char c, double a, double b)
{
  switch(c) {
  case '+':
    x=a+b;
    break; // break est nécessaire ici
  case '-':
    x=a-b;
    break; // break est nécessaire ici
  case '/':
    x=a/b;
    break; // break est nécessaire ici
  case '*':
    x=a*b;
    break; // break est nécessaire ici
  default:
     ... //Erreur
  }
  return x;
```

}

S'il n'y avait pas de **break**, les instructions dans les **case** qui suivent celui atteint seraient elles aussi exécutées!! Faîtes attention!

Les switch sont plus rapides que les **if** mais présentent un certain nombre de contraintes :

1. Seul des tests d'égalité peuvent être exécutés.
2. Seul un type prédéfini peut être testé.
3. Les valeurs de test (derrière **case**) doivent être constantes.

C'est une volonté d'optimisation du code qui dicte ces contraintes, cela rend le switch plus rapide que la série de **if**, **else if**, **else if**, **else** équivalente.

2.5.3 La boucle while

La boucle **while** est la boucle la plus simple en C++, sa syntaxe est

```
while(condition) {
  // Instructions
}
```

Ce qui se trouve entre les accolades est appelée « Corps de la boucle ».

Voici une boucle qui imprime les nombres de 1 à 9 :

```
int i=1;
while(i<=9) {
  std::cout << i << "\n";
  i++;
}
```

et maintenant une boucle de dichotomie

```
double a=3;
double b=3.5;

while(abs(b-a)>1e-6) {
 if(sin(a)*sin(x)<0) {
  x=b;
 }
 else {
   x=a;
 }
 x=(a+b)/2;
}
```

2.5.4 La boucle do while

La boucle **do while** s'écrit

```
do {
  // Instructions
} while(condition)
```

Par rapport à une boucle **while**, la différence est que le corps de la boucle est assuré d'être exécuté au moins une fois.

2.5.5 La boucle for

La boucle **for** est plus complète que la boucle **while**. Sa forme générale est

```
for(initialisation; condition ; executionfin) {
  //Corps de la boucle
}
```

qui équivaut [14] à

```
initialisation
while(condition) {
  //Corps de la boucle
  Executionfin
}
```

Elle est surtout utilisée pour incrémenter des indices automatiquement

```
int i;
for(i=0; i<10 ; i++) {
   std::cout << i << '\n';
}
// La variable i existe toujours
```

imprime les nombres de 0 à 9 à l'écran, tout comme

```
for(int i=0; i<10 ; i++) {
   std::cout << i << '\n';
}
// La variable i n'existe plus
```

2.6 Conclusion

Dans ce chapitre, nous avons vu introduit les bases de syntaxe nécessaires à la programmation en C++. Nous aborderons des sujets plus avancés dans les chapitres ultérieurs.

14. En toute rigueur, ce n'est que presque vrai. Il y a une petite nuance technique due à l'existence de l'instruction **continue** que nous n'abordons pas dans ce polycopié.

3 Les variables

Dans ce chapitre, nous introduisons les types composés et définis par l'utilisateur. Ces types sont construit à partir des types de base. Nous choisissons délibérément de ne pas introduire les tableaux prédéfinies par le langage `C++` et héritées du `C` : utilisez `std::vector` de la STL à la place, voir le chapitre 4.

Nous nous attardons ensuite sur la notion de durée de vie d'une variable : où dans un programme une variable est-elle accessible ?

3.1 Les pointeurs

Les types peuvent être composés. En particulier, il est possible de déclarer des pointeurs

```
int* p;    // p est un pointeur sur un entier.
float* q; // q est un pointeur sur un
          // flottant simple précision.
```

Un pointeur est une variable contenant l'adresse d'une autre variable. Le contenu du pointeur `q` est alors accédé par `*q` et l'adresse d'une variable `x` est accédé par `&x` :

```
float  x=1.0; // x vaut 1.0
float  y=2.0; // y vaut 2.0
float *q;
q=&x;   // q pointe maintenant vers la variable x;
*q=3.0; // x vaut 3.0
q=&y;   // q pointe maintenant vers la variable y;
*q=4.0;   // y vaut maintenant 4.0
x=*q;   // x vaut maintenant 3.0
```

L'opérateur unitaire préfixe* est appelé opérateur de déréférencement.

Un pointeur ayant la valeur nulle ne pointe vers aucune zone mémoire et le déréférencement du pointeur nul donne toujours lieu à un « segmentation fault ». Le pointeur nul a pour nom `NULL` en `C`, 0 en `C++98` et `nullptr` en `C++2011`.

En `C` et en `C++`, il est possible de faire de l'arithmétique sur les pointeurs. Si `T` est un type quelconque alors les opérations suivante sont syntactiquement correctes.

```
T* p;
// Initialisation de p ...
*p;      // Contenu pointé par p
p[0];
*(p+1);  // Contenu à l'adresse mémoire p+taillememoire(T)
p[1];  // Contenu à l'adresse mémoire p+taillememoire(T)
```

Ces opérations prennent en compte la taille mémoire de l'objet `T`. Attention, c'est dangereux, il faut vraiment savoir ce que l'on fait pour utiliser l'arithmétique sur les pointeurs.

3.2 Les références

Les références ressemblent aux pointeurs.

```
int    x=1;
int    y=3;
int&   r=x; // r est une référence vers x
r=2;        // x vaut maintenant 2
r=y;        // x vaut maintenant 3
            // r reste une référence sur x
            // r ne devient pas une référence sur y
r=0;        // x vaut maintenant 0
            // y reste inchangé
```

Une façon de voir les références est de les imaginer comme des pointeurs constants pointant toujours vers la même variable et qui sont automatiquement déréférencés par le compilateur. Un autre façon de voir les références est de considérer qu'une référence n'est en réalité qu'un alias pour une autre variable. L'association d'une référence à une variable donnée ne peut avoir lieu que lors de l'initialisation de la référence. Les références sont principalement utilisées pour spécifier que le passage d'arguments à une fonction doit être fait par référence.

3.3 Les structures

Les structures sont l'ancêtre des classes, voir chapitre 6. Ce sont des types contenant plusieurs variables de types différent. Elles sont définis en utilisant le mot-clef **struct**.

```
struct point {
  double x; //x est le premier membre de point
  double y; //y est le deuxième membre de point
};
```

Les sous-variables d'une structure sont communément appelées ses membres. Ici, la structure `point` a deux membres : `x` et `y`. Cela signifie que chaque variable du type `point` contiendra exactement deux double.

Pour déclarer une variable de type `point`, on utilise la même syntaxe que pour les types prédéfinis :

```
point A; //A est un point
A.x=5.0;    //Le membre x de A vaut 5.0
A.y=3;    //Le membre y de A vaut 3.0
```

et on a construit le point A du plan de coordonnées $(5, 3)$. L'opérateur « . » est l'opérateur de sélection en `C++`. On peut aussi déclarer un pointeur sur une structure :

```
point A;
point* q;
q=&A;
*q.x=3.0;    // erreur: *(q.x) et non (*q).x
(*q).x=4.0;    // OK mais lourd en parenthèses
q->x=5;    // équivaut à (*q).x=5
```

Il faut alors soit mettre des parenthèses soit utiliser l'opérateur de sélection $->$ pour les pointeurs sur les structures.

Si on souhaite définir le type plus tard, on peut se contenter de déclarer la structure sans la définir :

```
struct point A; // Structure A déclarée mais non définie.
```

Dans une structure, on peut uniquement avoir des membres dont le type est déjà défini, y compris d'autres structures. Par exemple, une fois la structure `point` définie, on peut définir la structure `triangle` :

```
struct triangle {
  point A;
  point B;
  point C;
};
```

Par contre, une structure ne peut être récursive, on ne peut déclarer un membre dont le type n'est pas complètement défini. Cette contrainte provient du typage statique qui impose que le compilateur connaisse la taille en octets des types. Il est par contre possible de placer des pointeurs vers des structures qui ne sont que déclarées. En effet, la taille d'un pointeur ou d'une référence est fixe : 4 octets sur les machines 32 bits et 8 octets sur la machines 64 bits. Par exemple,

```
struct recursive {
  recursive next; //Illégal recursive n'est pas encore définie
  recursive* opt; //Légal recursive est déclarée
  recursive& up; //Légal recursive est déclarée
};
```

3.4 Pointeurs sur des fonctions

Il est possible en C++ de manipuler des pointeurs sur des fonctions. Cela peut être utile par exemple pour l'algorithme de la dichotomie que l'on ne souhaite pas réécrire chaque fois qu'on l'applique à une nouvelle fonction.

Voici une déclaration de pointeur sur une fonction.

```
double (*f) (double);
```

Et voici l'algorithme de la dichotomie

```
double dichotomie(double (*f)(double) , double a, double b)
{
```

```
  double x=(a+b)/2.0;

  while(abs(b-a)>1e-6) {
    if((*f)(a)*(*f)(x)<0) {
      b=x;
    } else {
      a=x;
    }
    x=(a+b)/2.0;
  }
}
```

et comment on appelle dichotomie depuis une autre fonction :

```
...
dichotomie(0,1, &fun);
...
```

où `fun` est une fonction qui prend un **double** en argument et qui retourne un **double**.

3.5 Déclarations compliquées

Le `C++` a un système de déclaration assez compliqué hérité du `C`. Voici quelques exemples de déclarations compliquées

```
/*pointeur sur un pointeur sur un char */
char** p;

/*
  pointeur sur une fonction prenant un entier
  et un flottant comme argument et retournant
  un pointeur sur un entier
*/
int* (*fun) (int, float);

/*
  fonction prenant un entier et un flottant comme
  argument et retournant un pointeur sur un pointeur
  sur un entier
*/
int** fun (int, float);

/* Pointeur sur une fonction retournant un caractère
  et prenant en argument un entier et un pointeur
  sur une fonction ne prenant aucun argument et
  retournant un caractère.
```

```
*/
char (*fun)( int , char (*)());
```

3.6 Les typedefs

La syntaxe compliquée de déclaration de type en C++ peut être simplifiée par l'utilisation du **typedef** qui permet de donner un alias, un synonyme à un type.

```
typedef int Int;
Int a; //a est un int;

typedef double (*fonctionreel)(double);

double dichotomie(double,double,fonctionreel);
```

Ce qui rend la définition de dichotomie plus lisible. Un **typedef** créé un synonyme pour un type, il ne créé pas de nouveau type :

```
typedef int Int;
int a=0;
Int b=0;
/* a et b ont exactement le même type */
a=b;//Légal
b=a;//Légal
```

3.7 Durée de vie d'une variable

La durée de vie d'une variable, aussi appelée étendue(scope en anglais) d'une variable est un concept très important. En effet, il existe plusieurs types de variables qui se distinguent principalement par leur étendues.

3.7.1 Variables globales

Une variable globale est une variable déclarée à l'extérieur du corps d'une fonction. Elle existe dans tout le programme, est accessible par toutes les fonctions et ne disparaît que lorsque le programme s'arrête. Il est extrêmement tentant de rendre toutes les variables globales car cela semble rendre plus facile la programmation. Cependant c'est trompeur : cela rend le programme difficile à lire [1] et pratiquement impossible à maintenir. Pour cette raison, il est en général préférable d'éviter l'utilisation de variables globales.

Pour définir une variable globale `xglobal` de type **int**, on écrit

```
int xglobal;
```

1. Avoir un programme lisible est important, on code autant pour les autres programmeurs que pour le compilateur

à l'extérieur de toute fonction. Cette définition ne doit apparaître qu'une seule fois dans le programme. En effet, sinon le compilateur réserverait plusieurs fois de la place mémoire pour la variable `xglobal`.

Pour que la variable soit accessible depuis les fonctions définies en dehors du fichier, on doit la déclarer :

```
extern int xglobal;
```

Le mot-clef **extern** indique qu'il s'agit là non d'une définition mais d'une simple déclaration. De préférence, cette déclaration aura lieu dans un header.

Nous voyons là la première nuance entre la notion de déclaration et la notion de définition. Une déclaration indique au compilateur qu'une variable ou un objet existe déjà mais ne demande pas au compilateur de réserver de la place mémoire pour cette variable. Elle permet au compilateur de vérifier que les opérations faites sur la variable sont bien licites. Une définition au contraire demande au compilateur de réserver de l'espace mémoire pour la variable. Aussi, une variable globale ne doit être dédinie qu'une seule fois par programme. Par contre, la même variable peut être déclarée autant de fois que l'on souhaite.

Une variable globale peut être déclarée **static**. Cela signifie qu'elle n'est accessible que par les fonctions définies dans le même fichier. C'est une fonctionnalité considérée obsolète et remplacée par les namespaces et notablement le namespace anonyme.

3.7.2 Variables locales

Une variable locale est une variable déclarée dans le corps d'une fonction. Elle n'est accessible que depuis cette fonction dès sa déclaration et disparaît une fois que l'on sort du plus petit bloc { } contenant la déclaration de la variable.

```
if(condition) {
  int x=0;
} else {
  int x=1;
}
int b=x; //OOPS, x n'existe plus.

int f (int)
{
  ...
  int x;
  {
    int y;
    ...
  }
// y cesse d'exister
// x existe toujours.
...
}
```

3.7.3 Variables locales statiques

Une variable déclarée dans le corps d'une fonction peut être déclarée statique :

```
void f ()
{
  static int x=0;
  std::cout << x <<'\n';
  x=x+1;
}
```

Dans ce cas, la variable n'est accessible que dans la fonction où elle est déclarée mais la valeur de la variable statique sera conservée entre les diverses exécutions de la fonction. Par exemple, l'instruction

```
int i;
for(i=0;i<4;i++) {
  f();
}
```

affichera à l'écran

```
0
1
2
3
```

Attention, la valeur de la variable `x` n'est conservée que parce qu'elle a été déclarée **static**.

3.7.4 Les variables dynamiques

Une variable dynamique est une variable explicitement allouée. Les variables dynamiques perdurent jusqu'à ce qu'elles soit explicitement desallouées par le programmeur.

Ces variables sont appelées dynamiques car l'espace mémoire nécessaire à la conservation est réservée au cours de l'exécution et non au cours de la compilation comme pour les autres variables (locales, globales et statiques). En particulier, cela permet de créer des tableaux ou des matrices dont la taille change à chaque exécution du programme ce qui n'est pas possible sans recompilation si on utilise des variables non dynamiques.

En `C++`, on alloue dynamiquement une variable avec l'opérateur **new** (qui remplace la fonction `malloc` du `C`). On désalloue dynamiquement avec l'opérateur **delete**

```
double* p;
double* q;
{
  double x;
  p = new double; // Alloue un seul double en mémoire dynamique.
  q = &x;
} // x cesse d'exister
*q = 1.0; // OOPS q pointe vers une variable qui a disparu
*p = 0.0; // OK
```

```
delete p;
*p = 1.0;  // OOPS *p a été effacé
```

On peut aussi allouer une grande quantité de mémoire d'un coup avec les opérateurs **new**[] et **delete**[] :

```
double* p;
p= new double[100]; // Alloue 100 double
P[0] = 1.0;  //le premier double alloué
p[5] = 4.0;
P[99] = 2.0; // le dernier double alloué
delete[] p;      // Rend au système la région de
                 // mémoire précédemment allouée
```

Attention, l'opérateurs **new**[] n'initialise pas la zone mémoire à 0.0 pour les types prédéfinis. Il n'y a pas besoin de se rappeler combien de double ont été allouées pour faire un **delete**[] de toute la région : la taille de la région est conservée en mémoire automatiquement.

Un bug classique et difficile à corriger lorsque l'on alloue de la mémoire est la "fuite de mémoire" : on alloue de la mémoire puis on oublie de la libérer(désallouer) quand on ne s'en sert plus . Plus le programme est destiné à tourner longtemps plus cette "fuite" s'aggrave et devient problématique. Un programme qui laisse fuire de la mémoire court le risque d'en allouer plus que le système ne peut en fournir et d'être arrêté d'office par le système (lors de l'arrêt d'un programme, la mémoire est de toute façon retournée au système).

L'allocation dynamique est à cantonner au code de bas-niveau [2]. En C++, on peut souvent éviter d'utiliser **directement** l'allocation dynamique en utilisant la bibliothèque standard, en particulier pour les chaînes de caractères et les conteneurs.

2. L'expression « bas-niveau » est à prendre au sens code proche de la machine. À opposer à l'expression « haut-niveau » qui signifie un code proche de l'abstraction idéaliséee de l'algorithme.

4 Les conteneurs en C++ : la STL

Il est très important dans les programmes réels d'avoir des conteneurs efficaces. Un conteneur est un type qui contient plusieurs variables d'un même type. Le conteneur de base du `C++` (hérité du C) est très primitif et nous choisissons de ne pas en parler dans ce document. En effet, la bibliothèque standard contient la STL [1], une bibliothèque de conteneurs très efficace.

4.1 Les tableaux

Un tableau est un type qui contient un certain nombre de variables du même type. Il est préférable d'utiliser les tableaux fournies par la bibliothèque standard à ceux héritées du `C`. Un tableau est simplement une région contiguë dans la mémoire de l'ordinateur qui contient plusieurs variables du même type, voir figure 4.1.

Voici un exemple d'utilisation :

```
std::vector<int> x(20);      // Un tableau de 20 entiers
for(int i=0;i<20;i++) {
  x[i]=i*i;
}
x.size() ; // retourne la taille du tableau x;
x.resize(40) ; // Le tableau contient 40 éléments.
x.resize(10) ; // Le tableau est tronquée
```

Attention l'accès aux valeurs d'un tableau par `x[i]` ne vérifie pas si on déborde ou non du tableau. Pour un accès moins rapide mais sûre qui vérifie les bornes du tableau, utiliser `x.at(i)`

```
std::vector<std::string> S; // un tableau vide de chaînes
                            // de caractères
std::string s;
do {
  std::cin >>s;    // Ligne d'entrée à l'écran dans s;
```

1. Standard Template Library

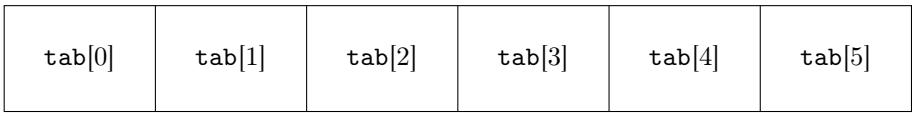

FIGURE 4.1: Stockage machine d'un tableau

```
  S.push_back(s); // Rajoute s en fin de tableau.
                  // La taille du tableau augmente de $1$.
} while (s!="\n");

sort(s.begin(),s.end());
                  // Chaînes triées par ordre alphabétique.

for(int i=0; i<S.size(); ++i) {
  std::cout << "Bonjour " << S[i] << '\n';
}
```

Lorsque l'on utilise `push_back`()il est possible que le tableau devienne trop grand et ne tienne plus dans la région mémoire allouée il sera déplacée automatiquement dans une région où il y a suffisamment de place. Pour un grand tableau, cela peut-être très coûteux.

En C++, on peut simplement déclarer un tableau de `T` par `std::vector<T>`

```
#include <vector>
std::vector<int>    tableaudentiers;
std::vector<float>  tableaudeflottants;
```

Et nous avons respectivement un tableau d'entiers et un tableau de flottants. On peut mettre presque n'importe quel type entre < et >. La clef d'un `std::vector` ne peut alors qu'être un entier.

```
std::vector<double> a;          // a a pour taille 0
std::vector<double> b(3);       // b a pour taille 3
                        // et chaque élément vaut 0.0
std::vector<double> c(4, 1.0);  // c a pour taille 4
                        // et chaque élément vaut 1.0

a.size(); // retourne la taille de a;
a[2]=5.0 ;   // assigne 5.0 au troisième élément de a
a[0]=5.0 ;   // assigne 5.0 au premier élément de a
a.at(2)=5.0; // pareil mais vérification que a.size()<=3.
a.push_back(3.0); // rajoute 3.0 à la fin de a
```

Quand un tableau devient trop grand (après des `push_back`()), il faut le déplacer. Cela est fait automatiquement mais cela prend du temps. En effet, il risque de ne pas y avoir de place dans la mémoire après le tableau pour les nouveaux éléments. Pour éviter cela, on peut réserver de la place :

```
a.reserve(100); // réserve 100 places
a.capacity(); // retourne la capacité du tableau a
```

Il ne faut pas confondre la capacité et la taille d'un tableau. Les tableaux sont efficaces pour l'accès aléatoire mais l'insertion coûte chère sauf en fin de tableau s'il y a assez de réserve.

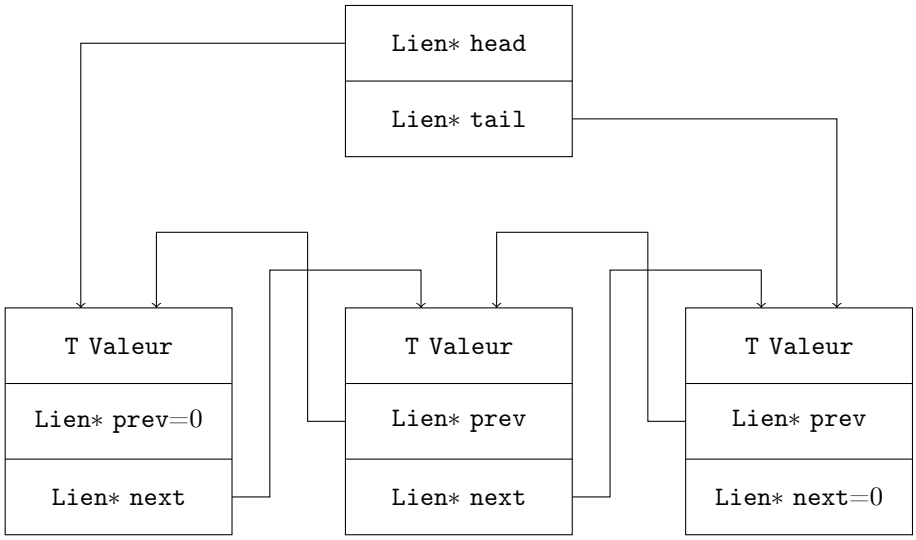

FIGURE 4.2: Stockage machine d'une liste

4.2 Les listes

Une liste est stockée de manière différente. Chaque élément de la liste contient en plus de sa valeur des liens [2] vers les éléments précédents et suivants, voir figure 4.2. La liste contient juste un accès au premier et au dernier élément. Pour déclarer une liste on utilise `std::list<T>` où T est un type :

```
std::list<int> listedentiers;
std::list<float> listedeflottants;
```

Comme pour un `vector`, on peut mettre presque n'importe quel type entre $<$ et $>$. Pour accéder aux éléments d'une liste, on emploie un itérateur :

```
std::list<int>::iterator p = listedentier.begin();
*p; // premier élément
p++;
p++;
*p; //troisième élément
p!=listedentier.end(); //Est-on en fin de la liste ?
```

Les listes sont très efficaces pour l'insertion et la suppression d'éléments mais la recherche est coûteuse.

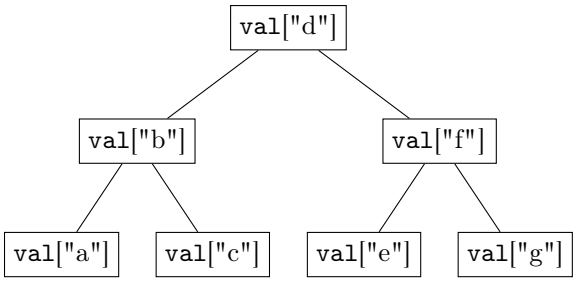

FIGURE 4.3: Un arbre equilibré indicé par des chaînes de caractères

4.3 Les arbres

Un arbre est une structure de données en informatique où chaque élément contient un lien vers un ascendant et deux liens (ou plus) vers des descendants, voir figure 4.3. Si l'arbre est équilibré, la profondeur de l'arbre est environ $\log_2(N)$ où N est le nombre de noeuds. Si le type qui servira de clef que l'on souhaite mettre dans le conteneur a un ordre, il est possible de placer les valeurs dans l'arbre de manière ordonnée. La recherche dans un arbre ordonnée est proportionneul à la profondeur de l'arbre qui s'il est équilibre est en $O(\log_2(N)$. Heureusement, il existe des algorithmes pour maintenir un arbre ordonné équilibré lors de l'insertion de nouveaux éléments.

La complexité de toutes les opérations sur un arbre sera de $O(\log_2(N)$. Des trois conteneurs vus ici, c'est le plus complexe à programmer. Heureusement, nous n'avons pas à le faire car d'autres l'ont fait pour nous. Il suffit d'utiliser les arbres fournis par la bibliothèque standard.

Pour déclarer une variable de type arbre on utilise `std::list<clef,T>` où `clef` est un type avec un ordre et `T` est un type :

```
std::map<std::string,int>   listedentiers;
std::map<std::string,float> listedeflottants;
listedentiers["ici"]=3;
int a= listedentiers["ici"];
```

4.4 Conclusion

Il faut bien se rappeler que la mémoire d'un ordinateur moderne est contiguë. Cela rend les performances des conteneurs différentes : aucun des conteneurs n'est optimal pour toutes les opérations. L'insertion dans une liste est rapide mais la recherche est en temps linéaire. La recherche dans un tableau triée est rapide mais l'insertion est coûteuse en temps linéaire. Pour un arbre, toutes les opérations sont en temps logarithmique.

Tous les conteneurs ne sont pas abordés dans ce chapitre. Nous avons présenté les trois les plus courant. Il y en a bien d'autres. Parmi eux, on peut citer les tableaux hashés [3].

2. Ce sont en général des pointeurs, nous en parlerons ultérieurement
3. Introduits dans la bibliothèque standard en C++2011 sous le nom `unordered_map`.

5 Programmation modulaire et organisation d'un programme

Maintenant que nous avons vu les structures de contrôle et les différents types de variables, nous allons pouvoir créer des programmes non triviaux.

La plupart des codes créés lors de l'apprentissage d'un langage sont courts et il n'y a pas besoin de les organiser pour qu'ils fonctionnent. Cependant, il est courant que les codes industriels atteignent 10000 lignes de codes et même 100000 lignes de code. Sans discipline et sans organisation, autant de lignes de codes ne peuvent être maintenues. Il est donc important de prendre de bonnes habitudes dès maintenant.

La programmation modulaire est une façon d'organiser un code. Dans la programmation modulaire, les fonctionnalités d'un programme sont séparées en fonctions et les fonctions apparentés sont regroupées dans un même fichier, dans un même **namespace**, voir section 5.4 ou dans un même module. Les différents modules sont alors aussi indépendant que possible les uns des autres. Pour cela, on utilise soit les namespaces, soit la séparation du code source en plusieurs fichiers.

Le support pour les modules eux-mêmes ne viendra en `C++` que dans un addendum technique au `C++2011`. Il faudra donc attendre encore quelques années. Il est cependant possible, dès à présent, de programmer modulairement avec de la discipline en utilisant les headers, les namespaces et la directive #**include**. Pour l'instant, nous devons donc fabriquer les headers à la main. Une fois l'addendum technique écrit et approuvé, cela sera heureusement inutile. Malheureusement, il n'est pas encore sorti à ce jour.

5.1 Un court exemple

Nous allons créer un court programme de calcul de la factorielle et montrer comment on peut l'organiser. C'est une fonction mathématique, elle ne posera pas beaucoup de problèmes informatiques. En effet, les fonctions mathématiques simples sont parmi les plus simples à concevoir : elles prennent un nombre fini et bien déterminé d'arguments, retourne un type bien déterminé, et surtout n'ont pratiquement jamais besoin d'accéder à une variable globale qui relate l'état d'un programme.

5.1.1 Un code désorganisé

Prenons l'exemple d'un programme qui calcule la factorielle d'un nombre entré au clavier. On peut facilement tout mettre dans la fonction `main`

Fichier `main.cc` : Version `main` fourre tout

```
int main ()
{
```

```
  unsigned int d;
  std::cout << "Entrez un entier : ";
  std::cin  >> d;

  unsigned int res=1;
  for(unsigned int i=1;i<=d;++i) {
    res*=i;
  }
 std::cout << res << "\n";
  return (0);
}
```

Comme le programme est court, cela ne pose pas de gros problèmes. Mais une telle organisation deviendra un cauchemar pour les codes grands de millions de lignes. C'est déjà beaucoup mieux.

5.1.2 Séparation en fonctions

Une meilleure organisation est de séparer les le calcul de la factorielle de la gestion des entrées sorties. Nous allons laisser la gestion des entrées sorties dans la fonction `main` et placer le calcul de la factorielle dans la fonction `factorielle`. Voici le code complet dans un seul fichier

Fichier `main.cc` : Version fonctions séparées

```
/* Implémente la fonction factorielle avec une boucle*/
#include <iostream>
unsigned int factorielle (unsigned int n)
{
  unsigned int res =1;
  for(int i=0;i<;i++) {
    res=res*i;
  }
  return(res);
}

int main ()
{
  unsigned int d;
  std::cout << "Entrez un entier : ";
  std::cin  >> d;

  unsigned int res=factoriel(d);
  std::cout << res << "\n";
  return (0);
}
```

Cependant, il est préférable de séparer les fonctions non apparentées en plusieurs fichiers.

5.1.3 Séparation en plusieurs fichiers

Le C++ est un langage à compilation séparée. Quand le compilateur lit un fichier source, il n'a aucune idée de ce qui se trouve dans les autres fichier sources. Aussi si dans `main.cc`, nous avons

Fichier `main.cc` : Version erronée avec fichiers séparés

```
#include<iostream>
/*Point d'entrée du programme*/
int main()
{
  unsigned int d;
  std::cout << "Entrez un entier : ";
  std::cin  >> d;

  unsigned int res=factorielle(d); // Erreur, le compilateur
                                   // ne sait pas que factorielle
                                   // est une fonction.
  std::cout << res << "\n";
  return (0);
}
```

et que la fonction `factorielle` est définie dans `maths.cc` :

Fichier `maths.cc` : Version fichiers séparés

```
unsigned int factorielle (unsigned int n)
{
  unsigned int res =1;
  for(int i=0;i<;i++) {
    res=res*i;
  }
  return(res);
}
```

alors nous obtenons une erreur. En effet, le compilateur compile séparément les fichiers `maths.cc` et `main.cc`. Quand il compile le fichier `main.cc`, le compilateur n'a aucune idée de ce qui se trouve dans `maths.cc`.

Pour résoudre le problème, il suffit de rajouter le prototype, voir section 2.3.6, de la fonction `factorielle` avant la fonction `main`. Voici le nouveau fichier `main.cc` :

Fichier `main.cc` : Version correcte avec fichiers séparés

```
#include<iostream>
/* Prototype de la fonction factorielle */
unsigned int factorielle(unsigned int);

/*Point d'entrée du programme*/
int main()
```

```
{
  unsigned int d;
  std::cout << "Entrez un entier : ";
  std::cin  >> d;

  unsigned int res=factorielle(d); // OK, grâce au prototype, le
                                   // compilateur a suffisamment
                                   // d'informations pour valider
                                   // cette instruction
  std::cout << res << "\n";
  return (0);
}
```

C'est déjà mieux que la précédente organisation.

Cependant, rajouter les prototypes à la main de toutes les fonctions appelées dans un fichier mais définies dans un autre fichier est non seulement fastidieux mais aussi une potentielle source d'erreurs. Pour remédier à ce problème, on utilise les headers.

5.2 Les headers

À chaque fichier source, correspond en général un header. Un header est juste un autre fichier qui contient du code souce et qui est destiné à être inclu par d'autres fichier de code source. Les headers ont par convention une extension en `.h` ou `.hh`. Les headers ne sont pas destinés à être compilés directement mais destinés à être inclu textuellement dans une étape qui précède la compilation. Il y a deux écoles pour les fichiers de headers : soit on créé un header unique fourre-tout pour tout le programme, soit on en créé un par fichier source. Dans un header, on place ce que le compilateur doit savoir quand il compile un autre fichier faisant appel aux fonctionnalité du (d'un des) fichier source associé au header. Nous donnerons plus loin les détails à la section 5.2.3. Les headers sont nécessaires car le C++ pratique la compilation séparée.

5.2.1 Généralités sur les headers

Nous avons déjà rencontré ces headers, <`iostream`> et <`cmath`> sont des headers. Ces headers interviennent par l'intermédiaire de la directive #**include**#. Ainsi au début d'un fichier source, on trouve courramment cette directive :

```
#include<iostream>
#include<cmath>
/* Reste du code */
```

Lors de la compilation, tout se passe comme si le contenu du fichier `iostream` et du fichier `cmath` était copié collé à la place de #**include**<iostream> et de #**include**<cmath>. Moralement, si on veut utiliser les fonctions d'entrées sorties de la bibliothèque standard, on rajoute la directive #**include**<iostream> en début de fichier et si on souhaite utiliser le fonctions mathématiques de la bibliothèque standard, on rajoute la la directive #**include**<cmath> en début de fichier. On peut le voir comme un mécanisme très primitif de chargement d'un module.

En plus des headers standard, le programmeur peut créer ses propres headers. À la différence des headers standard, les headers du programmeur doivent, dans les directives #**include**, être délimités par des guillemets doubles " et non des signes d'inégalités < >

```
#include <headerstandard>
#include "headernonstandard"
```

Ces headers sont nécessaires. En effet, le C++ compile séparément chacun des fichiers sources. Une fonction `main` définie dans un fichier `main.cc` peut avoir besoin d'appeler une autre fonction `print_result` définie dans un autre fichier `output.cc`. Pour pouvoir appeler cette fonction depuis `main.cc`, il faut que qu'elle ait été déclarée (prototypée) dans le fichier `main.cc` avant son premier appel. Pour cela, le mieux est d'inclure le header `output.h` où on aura préalablement prototypé toutes les fonctions définies dans `output.cc`.

Revenons maintenant à l'exemple de la section 5.1.

5.2.2 Création d'un header

Dans l'exemple de la section 5.1, nous en étions à chercher un moyen d'éviter de réécrire à la main le prototype de la fonction `factorielle` dans le fichier `main.cc`. Pour cela, nous devons créer un header nous même. Nous allons créer un header `maths.h` contenant les prototypes, voir section 2.3.6, de toutes les fonctions définies dans `maths.cc`. Ici, nous aurons dans `maths.h`, le prototype de la fonction `factorielle` :

Fichier `maths.h` : Version avec header

```
unsigned int factorielle(unsigned int);
```

Même si ce n'est pas toujours nécessaire, il est préférable d'inclure le header associé à un fichier source dans ce fichier source : cela permet au compilateur de repérer certaines erreurs et de vous en avertir.

Fichier `maths.cc` : Version avec header

```
#include "maths.h"
unsigned int factorielle (unsigned int n)
{
  unsigned int res =1;
  for(int i=0;i<;i++) {
    res=res*i;
  }
  return(res);
}
```

Enfin, le fichier `main.cc` est modifié pour utiliser un header et supprimer le prototype rajouté à la main :

Fichier `main.cc` : Version correcte avec header

```
#include<iostream>
```

```
#include "maths.h"

/*Point d'entrée du programme*/
int main()
{
  unsigned int d;
  std::cout << "Entrez un entier : ";
  std::cin  >> d;

  unsigned int res=factorielle(d); // OK, grâce au prototype, le
                                   // compilateur a suffisamment
                                   // d'informations pour valider
                                   // cette instruction
  std::cout << res << "\n";
  return (0);
}
```

Et nous avons maintenant organisé correctement le code d'un petit programme. Cela n'a pas l'air très utile pour un programme aussi court mais il est préférable de prendre les bonnes habitudes dès maintenant.

5.2.3 Que mettre dans un header ?

Tout d'abord, signalons que si on a de la chance, l'addendum technique sur les modules au C++2011 sortira bientôt. Une fois sorti, la question sera obsolète car ce qui remplacera les headers sera automatiquement créé par le compilateur. Malheureusement, à ce jour, il n'est pas encore sorti. Nous somme donc, pour l'instant, obligés de créer les headers à la main et d'apprendre ce qui doit être et ce qui ne doit pas être mis dedans.

Dans un header, on place en général

1. Des prototypes de fonctions.
2. Des définitions de classe.
3. Des déclarations de variables global, *i.e.* avec le mot-clef **extern**.
4. Des template, c'est un sujet avancé, voir le chapitre 7.4.

Dans un header, on ne met généralement pas

1. Des définitions de variables globales, *i.e.* sans le mot-clef **extern**.
2. Des définitions de fonctions, *i.e.* avec un corps de fonction.
3. Des définitions de fonctions membres définies en dehors de la classe.

5.3 Quelques règles de bonne conduite

Certaines de ces règles peuvent probablement être ignorées si vous savez ce que vous faîtes. Cependant pour ignorer ces règles, il faut soit avoir une bonne raison, soit suivre d'autres règles.

5.3.1 Placer le `main` à part

Dans un programme, il ne peut y avoir qu'une seule fonction `main`. Il est préférable de placer la fonction `main` dans un fichier à part avec peut-être quelques fonctions auxiliaires qui sont propres au programme et n'ont aucun potentiel de réutilisation.

Attention, il ne doit **jamais** y avoir de fonction `main` dans une bibliothèque. En effet, supposer que vous programmiez une bibliothèque de résolution d'équation aux dérivées partielles. Si le fichier contenant la fonction `EulerExplicite` contient aussi une fonction `main`, la fonction `EulerExplicite` ne pourra jamais être utilisée par aucun programme

5.3.2 Organiser vos fichiers

Les fonctionnalités apparentées sont de préferences regroupées dans un même fichier ou dans un même namespace. Les fonctionnalités qui n'ont aucune parenté entre elles vont dans des fichier différents.

De plus, nommez vos fichiers avec soin. Mettez vous à la place de celui récupère votre code et qui trouve dans le répertoire de votre projet les fichiers `truc.cc`, `chose.cc`, `machin.cc`, `machinchose.cc`, `stuff.cc` ou encore `exo.cc`. Ce genre de nom de fichier est à bannir. Un nom de fichier doit informer sur son contenu.

5.4 Les namespaces

Les namespaces sont des régions de code qui peuvent contenir toutes les autres constructions du `C++`. Elles offrent une autre possibilité d'organiser son code.

5.4.1 Le namespace anonyme

La première fonctionnalité des namespaces est le **namespace** anonyme. les types, variables et fonctions définis dans le namespace anonyme ne peuvent être accéder que depuis le même fichier : cela remplace une des nombreuses utilisation du mot-clef **static**.

```
namespace {
  int a;
  int n;
}
```

5.4.2 Les namespaces non anonymes

Une autre fonctionnalité est de pouvoir faire coexister plusieurs types ayant le même nom dans un même programme. Imaginer que dans un programme coexiste deux bibliothèques de calcul symbolique et que chacune d'entre elle ait choisi l'identifiant `expr` pour le type expression. Si les bibliothèques sont déclarées dans la zone globale, il y aura conflit. Si chacune des bibliothèques est définie dans un namespace nommé différemment, il n'y aura aucun problème :

```
namespace mes_expressions {
  struct expression {
    //...
  }
  // Définitions de variables
  // Définitions de fonctions
}

namespace les_expressions_du_voisin {
  struct expression {
    //...
  }
  // Définitions de variables
  // Définitions de fonctions
}
```

Et on peut déclarer dans le programme des expressions de type différent avec

```
mes_expressions::expr e1;
les_expressions_du_voisin e2;
```

5.5 Conclusion

Organiser son code est primordial pour des programmes longs de plusieurs millions de lignes de code écrit en coopération par une équipe. Pour l'instant, en C++, cela signifie qu'il faut savoir créer des headers. Espérons que cela ne soit plus le cas à l'avenir.

Il est aussi très important d'être discipliné quandon programme en équipe. En effet, les morceaux disjoints d'un même programme écrit par des personnes différentes doivent être compatible. Pour assurer cette compatibilité, le seul moyen est de respecter à la lettre la spécification des interfaces données par le chef du projet : en particulier il faut respecter le nom des fonctions, ordre des arguments des fonctions, nom des classes, nom des fichiers headers, nom des fichiers sources chosies par le chef du projet. Ceci, même si vous n'aimez pas les choix du chef de projet. Sans cette discipline, les morceaux du programme codés par des personnes différentes ne s'imbriqueraient pas correctement.

6 Les classes et la programmation orientée objet

Par rapport au C, l'ajout le plus visible en C++ est la notion de classe. Les classes sont à la base de la programmation orientée objet mais il ne suffit pas de créer une classe pour faire de la programmation orientée objet.

6.1 Les classes

Nous avons introduit le concept de structure à la section 3.3. Les classes généralisent cette notion. La notion de classe permet aux programmeurs de définir de nouveaux types pouvant être utilisé avec les mêmes facilités que les types prédéfinies. Ces nouveaux types peuvent alors être dotés en plus d'un certain nombre d'opération et de fonctions membres. *I.E.* une classe n'est pas seulement définie par les sous-variables qui la constituent mais aussi par l'interface fournie à ses utilisateurs.

6.1.1 L'encapsulation

Dans la programmation modulaire, si l'on fait attention à diviser un programmes en plusieurs modules, toute les fonctions sont autorisées à modifier tous les membres d'un type et à accéder à toutes les variables, y compris les variables internes d'un autre module. Par convention, les programmeurs disciplinés s'abstiennent d'accéder directement aux données et appellent des fonctions spécialisées. En cas de bug, le nombre de fonctions à vérifier demeure alors beaucoup plus restreint, ce qui est primordial dans un grand programme. Cependant, le compilateur ne fait rien pour imposer cette contrainte et un programmeur peu attentif pourrait alors modifier directement un membre interne d'un type composé.

Ce n'est pas très grave pour un type simple comme un point de deux coordonnées :

```
struct point {
  int x;
  int y;
};
```

C'est beaucoup plus gênant pour une liste doublement chaînée.

```
struct lien {
  int val;
  lien* next;
  lien* prev;
};
```

```
struct liste {
  lien* tete;
  lien* fin;
}
```

Il est impératif pour qu'elle fonctionne que les liens `next` et `prev` soient cohérents. Il faut respecter un invariant : si `p` est un `lien*` et si `p->next` est non nul alors `(p->next)->prev` doit être égal à `p`. C'est ce que l'on appelle un invariant.

Si toutes les fonctions peuvent y accéder et qu'un programmeur décide d'accéder directement aux membres de `lien` et qu'il commet une erreur (ou que la bibliothèque de liste change son implémentation interne), on peut se retrouver avec un programme qui fera n'importe quoi et la cause de l'erreur risque d'être difficile à identifier.

6.1.2 Définition d'une classe

Pour cette raison, le C++ introduit les classes qui sont une généralisation des structures hérités du C. Une classe se définit comme une structure, voir section 3.3 mais on utilise le mot-clef **class** à la place de **struct**. Voici un exemple à vocation purement pédagogique

```
class point {
public: // Tous les membres déclarés après sont public
  double x; // x est un membre public
private: // Tous les membres déclarés après sont privés
  double y; // y est un membre privé
};
```

Nous remarquons ici les mots-clefs **private** et **public**. Ces mots clefs spécifient [1] si un membre d'une classe est un membre privé ou un membre public. Les membres publics sont directement accessibles par toute fonction tandis que les membres privés ne le sont qu'indirectement. Par défaut, en l'absence des mot-clefs **private** et **public**, les membres sont privés. Essayons maintenant d'accéder aux données

```
point A; //A est un point
A.x=5.0; //OK, x est public
A.y=3;   //Erreur, y est privé
```

Comment peut-on modifier le membre `y` ? Certaines fonctions ont le droit de modifier les membres privés d'une classe : les fonctions membres et les fonctions amies. Nous commençons par décrire ce qu'est une fonction membre.

6.1.3 Fonctions membres

Les classes peuvent aussi avoir des fonctions comme membre. On appelera fonctions membres ces fonctions. De même, on appellera données membres, les membres qui sont des variables. Reprenons la définition de la classe `point`

1. il existe une troisième spécification **protected** mais nous n'en parlerons pas dans ce document.

```
class point {
public:
  double x;
  double y;
private:
  double get_x() {return x;}  // déclarée et définie.
  double& get_y() {return y;}   // pareil et retourne une référence.
  point rotate(float angle) const;     // déclarée et non définie.
};
```

La classe `point` a deux données membres : `x` et `y`. Cette classe dispose aussi de trois fonctions membres : `get_x()`, `get_y()` et `rotate(float)`. Les fonctions membres d'une classe sont autorisées à modifier les données privées des variables de cette classe. De cette manière, on limite le nombre de fonctions autorisées à modifier la machinerie interne d'une variable d'une classe donnée.

Pour appeler une fonction membre, on utilise l'opérateur de sélection « . » :

```
point A;                // OK
A.x=2.5;                // OK
A.y=2.2;                // OK
double a=A.get_x();     // OK
double b=A.get_y();     // OK
A.get_x()=2.2;          // Erreur, get_x() ne renvoie pas une référence
A.get_y()=2.2;          // OK
```

L'instruction `A.get_x()=2.2` est une erreur car `get_x()` ne retourne pas une référence mais une valeur : la ligne a autant de sens que 1.0=2.2!!! Dans un code réel, la fonction membre `get_x()` renverrait elle aussi une référence.

Ici, et pour la plupart des classes que vous rencontrerez, les données membres sont privées et les fonctions membres sont publics. Rien n'interdit de déclarer privée une fonction membre, elle ne pourrait alors être appelée que depuis une autre fonction membre (ou depuis une fonction amie, notion non encore introduite). Cependant, cela n'est pas très courant.

Les fonctions membres `get_x()` et `get_y()` ont été définies à l'intérieur de la classe. Ce n'est pas le cas de la fonction `rotate(float)`. Pour le faire à l'extérieur de la classe, on procède de cette manière

```
/* Définition de rotate pour point */
point point::rotate(double angle) const
{
  point B;
  B.x=cos(angle)*x-sin(angle)*y;
  B.y=sin(angle)*x-cos(angle)*y;
  return B;
}
```

Le mot-clef **const** signifie que la fonction `rotate` ne modifie pas la variable de type `point` depuis laquelle elle est appelée. Le compilateur vérifiera que c'est bien le cas.

La syntaxe générale pour définir une fonction membre à l'extérieur de la classe est la suivante :

```
/* Définition de rotate pour point */
```

```
typederetour nomclasse::nomfonctionmembre(/*liste arguments*/) const
                                                    //const est optionnel
{
  /*Corps de la fonction*/
}
```

Remarquez que les données membres de la variable depuis laquelle la fonction membre `rotate`() est appelée sont accédées directement sans l'opérateur de sélection « . ». Les membres des autres variables du même type ou nom sont accédés de manière usuelle avec l'opérateur de sélection « . ». Par exemple, lors de l'appel `A.rotate`(3.14), le membre `A.x` est connu comme `x` sans aucun préfixe dans le corps de la fonction membre `rotate`(). Mais que fait-on quand on veut accéder à `A` lui même dans le corps de la fonction ? Nous répondons à cette question dans la section suivante.

6.1.4 Le pointeur this

Nous venons de voir dans les exemples précédents que les fonctions membres ont un accès direct aux membres de la variable depuis laquelle elles sont appelées. Pour accéder à la variable elle-même, il faut utiliser le pointeur **this** qui contient l'adresse de la variable elle-même. Voici un exemple de classe l'utilisant. Cette classe est inutile et a un but purement pédagogique

```
class exemple {
private: //inutile private par défaut
  int n;
public:
  exemple* adresse() {return this;}
  void setn_1(int m) { n=m;}
  void setn_2(int m) { this->n=m;}
  void setn_2(int m) { (*this).n=m;}
};
```

Alors, on peut l'utiliser de la manière suivante

```
exemple A;
exemple* p;
p=A.adresse(); // équivalent à p=&A;
A.setn_1(2);   // A.n vaut 2
A.setn_2(3);   // A.n vaut 3
A.setn_3(5);   // A.n vaut 5
```

Signalons enfin que le pointeur **this ne peut être accédé que** depuis une fonction membre.
Nous verrons des exemples d'utilisation de **this** moins artificiels plus loin dans ce polycopié.

6.1.5 Fonctions et classes amies

Considérons une classe de vecteurs mathématiques `mathvector` et d'une classe de matrices `matrice`. Pour effectuer le produit matrice vecteurs de ces deux objets, il faut pouvoir accéder aux données membres de ces deux classes. Mais il n'est pas possible pour une fontion d'être membre de deux classes différentes.

La solution est d'utiliser une fonction dite amie. Les fonctions amies sont des fonctions normales mais qui ont accès aux membres privés des classes dont elles sont amis. Nous avons besoin de la fonction suivante.

```
mathvector
produit_matrice_vecteur (const matrice& a, const mathvector& b)
{
  //Algorithme de produit matrice vecteur
}
```

Pour qu'elle ait accès aux membres privés des variables de type `matrice` et `mathvector`, il suffit de la déclarer ami dans les deux classes. Rien de plus facile, il suffit de rajouter le prototype de cette fonction précédé du mot-clef **friend** dans les deux classes.

```
class matrice {
private:
  //....
public:
  //....
  friend mathvector
  produit_matrice_vecteur (const matrice& a, const mathvector& b);
};

class mathvector {
private:
  //....
public:
  // ....
  friend mathvector
  produit_matrice_vecteur (const matrice& a, const mathvector& b);
};
```

Les fonctions amies ne sont pas des fonctions membres, elles sont appelées comme des fonctions normales :

```
matrice A;
mathvector x;
//...
mathvector b;
b=produit_matrice_vecteur(A,x);
```

Remarquez l'absence d'opérateur de sélection « . ».

Le concept de classe amie existe aussi :

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

```
class B {
  //...
  friend class A; // Toutes les fonctions membres
                  // de A ont maintenant accès aux
                  // données privées de B;
};
```

Les fonctions et les classes amies sont une alternative aux fonctions membres quand il s'agit d'accéder aux données privées.

6.1.6 Constructeurs

Les constructeurs sont une notion primordiales en C++. Il est très important de comprendre ce qu'est un constructeur. Aussi, nous laissons leur syntaxe à plus tard. Un constructeur est une fonction membre d'une classe qui initialise la variable au moment même de la définition de cette variable.

Motivation

Pour justifier le concept de constructeur, observons comment on programmerait sans constructeurs. Supposons que nous ayons plusieurs classes, disons 3, `type1`, `type2`, `type3`. Les variables de ces types nécessitent une initialisation avant d'être utilisée. On programme alors des fonctions `init_type1`, `init_type2` et `init_type3`. Voici comment, nous les utiliserions

```
type1 var1;
init_type1(var1);
type2 var2;
init_type2(var2, arg21);
type3 var3;
init_type3(var3, arg31, arg32, arg33, arg34);
```

Mais cette manière de programmer est dangereuse : il est bien trop facile d'oublier une initialisation et pour certaines classes, particulièrement les classes utilisant l'allocation dynamique et pour les classes contenant des pointeurs, cela peut être dramatique. On peut se retrouver avec des variables dans un état indéfini et avoir un programme qui fait n'importe quoi. Par exemple, si un pointeur est mal défini, on peut avoir une erreur de segmentation. Ce genre d'erreur peut être difficile à repérer dans un long programme. La solution est d'utiliser les constructeurs, ainsi définition de la variable et initialisation auront lieu sur la même ligne et il n'y aura aucun risque d'oubli. Il serait préférable de pouvoir simplement écrire

```
type1 var1;
type2 var2(var2, arg21);
type3 var3(var3, arg31, arg32, arg33, arg34);
```

et que les variables `var1`, `var2` et `var3` soient initialisées comme si on avait appelé les fonctions `init_`. C'est exactement ce que font les constructeurs.

Syntaxe de déclaration des constructeurs

Les constructeurs sont reconnus comme constructeurs car ce sont des fonctions membres qui porte le même nom que la classe à laquelle ils appartiennent. Ils ne retournent rien, aussi c'est un des rare cas où une fonction est déclaré sans type de retour. La syntaxe générale pour déclarer un constructeur est

```
class nom_classe {
private:
  //Données
public:
  nom_classe (liste  arguments) ;
  //Autres fonctions membres
};
```

Puis, pour définir un constructeur en dehors de la classe

```
nom_classe::nom_classe (/*liste  arguments*/)
{
 //Corps de la fonction
}
```

Puis pour utiliser ce constructeur, il suffira d'écrire

```
nom_classe var(/*liste  arguments*/); //Initialisation faite par le constructeur
```

Il peut y avoir plusieurs constructeurs pour une même classe. Deux constructeurs sont particulièrement importants : le constructeur par défaut et le constructeur par copie.

Le constructeur par défaut

Si aucun autre constructeur n'est définie, le constructeur par défaut sera créé automatiquement par le compilateur. Le constructeur par défaut est celui qui est appelée lorsqu'on déclare une variable sans aucun argument. La syntaxe générale pour déclarer un constructeur par défaut est

```
class nom_classe {
private:
  //Données
public:
  nom_classe () ; // Constructeur par défaut
  //Autres fonctions membres
};
```

Pour initaliser une variable avec le constructeur par défaut, il suffit de la déclarer sans aucun argument

```
nom_classe var;
```

c'est le constructeur par défaut qui est appelé et qui initialise `var`.

Le constructeur par copie

Le constructeur par copie permet d'initialiser une variable d'un type par une variable du même type. Ce constructeur prend une référence constante sur la classe dont il est fonction membre comme argument. En voici la syntaxe générale :

```
class nom_classe {
private:
  //Données
public:
  nom_classe (const nom_classe&) ; // Constructeur par défaut
  //Autres fonctions membres
};
```

En particulier, si `var` est une variable préexistante de type `nom_classe`, alors lors des déclarations

```
nom_classe var1(var);
nom_classe var2=var;
```

c'est le constructeur par copie qui est appelé et qui initialise `var1` et `var2`.

Exemple

Prenons la classe `matrice`, cette classe a besoin d'allouer dynamiquement de la mémoire. En effet, on ne connait pas forcément lors de la compilation la taille des variables de type `matrice` et donc la quantité de mémoire dont elles auront besoin Voici comment créer des constructeurs pour cette classe :

```
class matrice
{
private:
  int m;
  int n;
  double *q;
public:
  matrice(int _m, int _n) {
    m=_m;
    n=_n;
    q= new double[m*n]; // allocation de mémoire
  }
  matrice(int _m, int _n, double val) {
    m=_m;
    n=_n;
    q= new double[m*n]; // allocation de mémoire
    for(int i=0;i<m*n;++i)
      q[i]=val;
  }
  matrice(const matrice& b) {
```

```
    m=b.m;
    n=b.n;
    /* Attention, on ne peut assigner b.q à q */
    q= new double[m*n]; // allocation de mémoire
    for(int i=0;i<m*n;++i) {
      q[i]=b.q[i];
    }
 }
  // Destructeur, voir section sur les destructeurs
  //autre fonctions membres
};
```

6.1.7 Destructeurs

Les destructeurs sont le contraire des constructeurs. Ils servent à nettoyer une variable quand celle-ci cesse d'exister. En effet, certaines classes aquièrent des ressources lors de leur construction. Ces ressources peuvent être de l'espace mémoire, des accès réseaux, des ouvertures de fichiers ou encore des locks. Le plus souvent, c'est de l'espace mémoire. Quand la variable atteint sa fin de vie, elle doit libérer ces ressources. Cela est fait avec un destructeur. Les destructeurs sont appelés automatiquement quand une variable atteint la fin de son existence. Comme les constructeurs, les destructeurs ne renvoient aucun argument. Contrairement aux constructeurs, une classe ne peut avoir qu'un unique destructeur et il ne prend aucun argument. Le destructeur se déclare comme un constructeur mais précédé de ~. Voici la syntaxe générale

```
class nom_classe {
private:
  //Données
public:
  ~nom_classe () ;
  //Autres fonctions membres
};
```

Puis pour définir un constructeur en dehors de la classe

```
nom_classe::~nom_classe (liste  arguments)
{
 //Corps de la fonction
}
```

Reprenons l'exemple de la classe matrice qui alloue de la mémoire dynamique lors de la construction. Il va falloir libérer cette mémoire dans le destructeur.

```
class matrice
{
private:
  int m;
  int n;
```

```
  double *q;
public:
  // Constructeurs, voir section précédente
  ~matrice() {delete[] q;} //libération de la mémoire
  //autre fonctions membres
};
```

Nous rappelons que les variables allouées dynamiquement existent jusqu'à ce qu'elles aient été explicitement détruites. C'est cette allocation dynamique qui justifie l'importance des destructeurs

```
{
  matrice m(4,4);

  // ...

}//~matrice() est appelé automatiquement avant le }
```

Le destructeur d'une variable est toujours appelé automatiquement lorsque la variable arrive en fin de vie. Il ne faut donc **jamais appeler explicitement un destructeur**. Ce principe vaut même pour les variables dynamique. En effet les opérateurs **delete** et **delete**[] appelent eux aussi automatiquement [2] les destructeurs. Ce principe de ne jamais appeler explicitement un destructeur ne souffre qu'une unique exception que nous n'aborderons pas dans ce polycopié.

6.1.8 Exemple : une classe de complexes

Nous allons maintenant regarder un exemple de classe.

```
class complexe {
private: //Non nécessaire, privé par défaut
  double re;
  double im
public:
  complexe() ; // Un constructeur
  complexe(double _re); // Un constructeur
  complexe(double _re, double _im); // Un constructeur
  complexe(const complexe&); // Le constructor par copie
  double& real();
  double& imag();
  complexe operator*(complexe b);
  complexe operator=(const complexe&); // L'assignation par copie
}
```

est un exemple de classe. Nous rappelons que les fonctions non membres ne peuvent accéder aux membres privés. Le but est de fabriquer une classe complexe qui puisse être utilisée de manière naturelle.

2. Et même au cas où vous surdéfinisseriez ces opérateurs **delete** et **delete**[], l'appel au destructeur est automatiquement rajouté et n'a pas à être ajouté manuellement dans le corps de fonction de ces opérateurs.

```
complexe a;
a.re=2; //erreur re est privé
double c= a.re; //erreur re est privé
```

Voici comment on souhaite pouvoir utiliser les constructeurs de la classe `complexe`. Rappelons que les constructeurs servent à initialiser une variable au moment même de sa création. Par exemple, lorsque l'on déclare :

```
complexe z; // complexe::complexe() est appelée
complexe z2(1.0); // complexe::complexe(double) est appelée
complexe z2=1.0); // complexe::complexe(double) est appelée
complexe z3(1.0,2.0); // complexe::complexe(double,double) est appelée

complexe z4(z);   // Constructeur par copie
                  // complexe::complexe(const complexe&)

/*Attention */
complexe z5=z;   // Constructeur par copie
                 // complexe::complexe(const complexe&)

z2=z;            // Assignation par copie
                 // complexe::operator=(const complexe&)
```

Ne pas confondre le constructeur par copie et l' assignation par copie..

6.1.9 Surdéfinition des opérateurs

En `C++`, il est possible de surdéfinir les opérateurs pour des classes définies par l'utilisateur. Cela signifie qu'il est possible de spécifier la signification des opérateurs arithmétiques $+$, $-$, $*$, $/$, des opérateurs de comparaisons, et de presque tous les opérateurs pour les classes définies par l'utilisateur. Il n'est pas possible de changer la priorité des opérateurs. Il n'est pas possible de changer le sens d'un opérateur pour les types prédéfinis. Voici comment on surdéfinit les opérateurs d'assignation $=$ et de multiplication $*$ pour la classe `complexe`. On souhaite que les instructions suivantes soient licites :

```
complexe a;
complexe b;
...
complexe z=a*b;
complexe z2;
z2=z;
```

Autrement dit, on souhaite pouvoir utiliser les complexes avec les notations mathématiques usuelles. Pour cela, il faut surdéfinir les opérateurs arithmétiques :

```
class complexe {
  double re, im;
public:
```

```
  //...
  complexe operator* (const complexe& b) const;
  complexe& operator= (const complexe& b);
}
```

Ici, si vous suivez, vous devez vous demandez pourquoi ces opérateurs ne prennent qu'un seul argument et non deux. La réponse est qu'il s'agit ici de fonctions membres. Le premier argument est comme pour toute fonction membre passée implicitement. Ainsi, si `z1` et `z2` sont des variables de classe `complexe`, l'instruction `z1=z2` est équivalente à `z1.operator=(z2)`. De même, l'instruction `z1*z2` est équivalente à `z1.operator*(z2)`. Ces deux formes longues sont licites mais ne sont pas utilisées en pratique. Ainsi, dans les deux cas, les données membres de `z1` sont accessibles directement sans préfixe dans la définition de ces deux opérateurs membres :

```
complexe complexe::operator*(const complexe& b)  const
{
 return(complexe(re*b.re-im*b.im,re*b.im+im*b.re);
}

complexe& complexe::operator=(const complexe& b)
{
  re=b.re;
  im=b.im;
  return(*this);
}
```

Le fait que l'opérateur = retourne une référence sur un `complexe` et qu'il retourne **return**(***this**) est nécessaire. C'est le cas pour tous les opérateurs d'assignation et **seulement les opérateurs d'assignation**, *i.e.* pour =, +=, −=, *=, et /=. Nous n'expliquerons pas pourquoi dans ce polycopié.

Il est aussi possible de surdéfinir les opérateurs par des fonctions non membres

```
mathvector
operator*(const matrice& a, const mathvector& b)
{
  //..
}
```

Il suffira de déclarer cet opérateur **friend** dans les classes `mathvector` et `matrice` :

```
class matrice {
//
public:
  friend mathvector
  operator*(const matrice& a, const mathvector& b);
};

class mathvector {
//
public:
```

```
  friend mathvector
  operator*(const matrice& a, const mathvector& b);
};
```

Remarquer que lorsque un opérateur est définie comme une fonction normale et non comme une fonction membre, aucun argument n'est passé implicitement et le nombre d'arguments est celui donnée par l'intuition.

On peut surdéfinir presque tous les opérateurs. Les plus courants sont les opérateurs arithmétiques, le déréférencement (∗ préfixe) et l'assignation.

6.1.10 Assignation et constructeurs

L'assignation, effectuée par **operator**=, et la construction, effectuée par un constructeur, sont deux notions différentes. Il est très important de s'en rappeler. Pour les classes simples, c'est peu apparent. Regardons un exemple où c'est évident. Reprenons l'exemple de la classe de matrice de la section 6.1.6. Lors de la construction, on aquiert de la mémoire . Lors de l'assignation `A=B`, la matrice `A` a déjà acquise de la mémoire. Si on programmait l'assignation comme la construction, on aurait ce que l'on appelle une fuite de mémoire. Si on impose comme condition que l'assignatioon entre deux matrice n'est licite que si les tailles de `A` et de `B` coïncident, alors il n'est pas besoin d'acquérir de la mémoire lors de l'assignation et l'opérateur = peut être définie de la manière suivante :

```
matrice& matrice::operator=(const matrice& b)
{
  if(m==b.m && n==b.n) {
    for(int i=0;i<m*n;++i) {
      q[i]=b.q[i];
    }
  } else {
    //Gestion d'erreur
  }
  return (*this);// Nécessaire pour que A=B=C fonctionne
}
```

6.1.11 Conclusion sur les classes

Les classes constituent le principal ajout du `C++` par rapport au `C`, elles constituent un outil puissant qui permet d'isoler l'utilisateur d'une bibliothéque des détails internes à la classe. Il est primordial pour le programmeur `C++` de savoir programmer des classes concrètes.

6.2 La programmation orientée objet

La programmation orientée objet ne se limite pas à la simple création de classes et l'encapsulation. Dans la programmation orientée objet, on se concentre sur la création d'objet, sur la création de classes et leur **hiérarchisation**. On essaie de regrouper les éléments communs d'une classe dans une classe de base.

Avant d'écrire les algorithmes, on créé des classes et on définit leur comportement : on commence d'abord par spécifier l'interface et par écrire les déclarations des fonctions membres. On implémente la classe ensuite.

6.2.1 L'héritage

Une classe peut être dérivé d'une autre et hériter tous ses membres. Voici un exemple,

```
class habitation {
  std::string nom;
  int pieces;
public:
habitation(std::string _nom, int _pieces);
void print();
};


class maison: public habitation {
  int nombreetages;
public:
maison(std::string _nom, int _pieces, int _etages);
void print(); // Redéclaration de print()
}

void habitation::print()
 {
  std::cout << "Habitation :" << nom << '\n'
            << '\t' << pieces << "pièces" << '\n'
}


void maison::print()
 {
   habitation::print();
   std::cout << '\t' << etages << "étages" << '\n';

   /*
      Illégale car maison ne peut accéder
      aux privés d'habitation
   */
   std::cout << '\t' << pieces/etages
             << "pieces par etages" << '\n';
}
```

La classe maison hérite de tous les membres de la classe `habitation` et est une classe hérité de `habitation`. Les fonctions membres de `maison` ne peuvent accéder aux membres privés de

habitation!!

6.2.2 Pointeurs et conversions implicites ?

Les pointeurs de type maison peuvent être convertis directement en pointeurs de type habitation sans conversion explicite. La réciproque n'est pas vraie.

```
maison a;
habitation b;

habitation*p=&a; // OK
maison*q=&b ;    // Erreur
```

Mais p−>print() va appelé le print d'habitation. Et le nombre d'étages ne sera pas affiché. En effet, le typage est statique et à la compilation, le compilateur ne peut deviner si p est non seulement une habitation mais aussi une maison! Pour cela nous introduisons les fonctions virtuelles.

6.2.3 Héritage public ou privé ?

Nous avons pour l'instant toujours déclare la classe de base **public**. Nous aurions aussi pu la déclarer **private** :

```
class A { public: void print() {std::cout << "n"}};
class B: public A {};
class C: private A {};
A a; a.print(); // OK
B b; b.print(); // OK print hérité de A et public.
C b; c.print(); // erreur la base A est privé
```

Les membres publics de C peuvent accéder aux membres publics de A mais l'interface de A dans C est maintenant privé .

6.2.4 Constructeurs et héritage

Nous introduisons ici une syntaxe un peu surprenante. En effet, comment construisons nous une classe dérivée. Il nous faut un constructeur.

```
class A {
private:
  A(); //Pas de constucteur par défaut.
public:
  A(double); // Un autre constructeur.
};

class B: public A {
private:
  B();
};
```

```
B::B()
{
    //?? Je veux appeler le constructeur A(1.0).
}
```

Nous ne pouvons pas appeler un constructeur de la classe de base depuis le corps du constructeur. En effet, une fois dans le corps de la fonctions, les classes de bases et les membres ont déjà été initialisées par défaut. Nous devons utiliser une autre syntaxe qui permet d'appeler les constructeurs.

```
B::B() : A(1.0)
{
// Autres instructions
}
```

Cette syntaxe est non seulement utilisé pour les classe de bases mais aussi pour les membres d'une classe. Cette syntaxe est d'ailleurs impérative pour les références qui doivent être initialisées.

```
class C: public A {
  float& r;
  double& x;
// ....
public:
  C (double a, float& b, double& x) : A(a), r(a), x(c){}
}
```

6.2.5 Les fonctions virtuelles

Supposons que l'on déclare

```
class habitation {
//...
public:
virtual print(); // Voir définition plus haut
}
class maison: public habitation {
//...
public:
virtual print(); // Voir définition plus haut
}
```

Alors dans ce cas

```
maison a;
habitation b;

habitation* p;
p=&a;
```

```
p->print();//

p=&b; // OK
p->print(); //
```

Cela vient que si une classe contient une fonction virtuelle, elle est considérée classe virtuelle et un champ est placée dans la représentation binaire de la classe par le compilateur qui précise le type exact de l'objet. Pour une classe virtuelle, il y a typage dynamique et la bonne fonction sera choisie lors de l'exécution. Cela ralentit l'exécution mais c'est nécessaire si on a besoin de polymorphisme dynamique où le type n'est pas prévisible lors de la compilation.

6.2.6 Un exemple de programmation objet

Supposons que l'on ait un programme géométrique qui dessine des figures. La façon objet de le programmer est d'avoir une classe abstraite `figure` et d'en dériver toutes les classes de figures particulières :

```
class figure {
public:
  virtual void dessiner() = 0 ;
  virtuel void rotation(double)=0 ;
}
```

Le =0 signifie que la fonction est virtuelle pure et qu'elle devra être redéclarée dans les classes dérivées. Une classe qui contient une fonction virtuelle pure est appelée classe abstraite et ne peut être instanciée :

```
figure A; // Erreur. A contient des foncions
          // virtuelles pures
```

On peut alors dériver cette classe :

```
class triangle : public figure {
  point A, B, C;
public:
  virtual void dessiner()       { /* code */ }
  virtuel void rotation (double angle) { /* code */ }
}

class cercle : public figure {
  point centre;
  double rayon;
public:
  virtual void dessiner()       { /* code */ }
  virtuel void rotation (double angle) { /* code */ }
}
```

Pour avoir un ensemble de figures, on utiliserait par exemple `std::list<figure*>`.

6.3 Conclusion

Dans la pratique, la très grande majorité des classes que l'on est amenées à programmer sont des classes concrètes autocontenues. Pour se lancer dans la programmation orientée objet au sens pure du terme, il faut d'abord bien maîtriser la programmation de ces classes concrètes. La programmation orientée objet trouve sa force quand on a besoin de polymophisme, *i.e.* de variables pouvant changer de type, à l'exécution.

7 Les templates et la programmation générique

Il est courant qu'un même algorithme soit programmé plusieurs fois pour des raisons purement informatique. Programmer un même algorithme plusieurs fois est source d'erreur. Il serait intéressant de pouvoir écrire un algorithme sous forme abstraite et laisser le compilateur l'adapter aux différents types. C'est possible en utilisant ce que l'on appelle la programmation générique. Ce type de programmation fait principalement appel à ce que l'on appelle les « templates ».

7.1 Un exemple simple

Beaucoup de fonctions portant le même nom implémente le même algorithme et ne diffèrent que par le type de leur arguments. Par exemple, regardons la fonction `max` :

```
int max(int a, int b)
{
  if (a>b)
  {
    return a;
  }
  else {
    return b;
  }
}

float max(float a, float b)
{
  if (a>b)
  {
    return a;
  }
  else {
    return b;
  }
}
```

Et il faudrait écrire toutes les versions pour tous les types prédéfinis et tous ceux définis par l'utilisateur alors que le corps de la fonction aura toujours la même structure. Peut-on réduire le nombre de copier-coller ? C'est possible avec les template.

```
template<typeneame LessThanComparable>

LessThanComparable
max(LessThanComparable a, LessThanComparable b)
{
  if(a>b) {
    return a;
  }
  else {
    return b;
  }
}
```

Maintenant tout appel à la fonction `max` avec deux arguments de même type va générer une nouvelle fonction `max` à partir du template. Supposons que l'on ait un type `mot` pour lequel on a défini une relation d'ordre alphabétique $<$ alors `max(mot1,mot2)` retourne le mot le plus à la fin du dictionnaire. La fonction `max(mot,mot)` est instanciée au premier appel de la fonction `max` pour le type `mot`.

7.2 Les classes templates

Une classe peut aussi bénéficier des templates. Essayons de programmer une liste pour un type arbitraire `T` :

```
template <typename T>
class ma_liste<T> {
  struct lien {
    lien* next;
    T val;
  };
  lien* tete;
  lien** fin;
  lien* current;
public:
  //Constructeurs
  //Acces
};
```

On peut alors déclarer des listes de types différents sans recopier 50 fois la même définition.

```
ma_liste<int> liste_entiers;
ma_liste<float> liste_flottants;
ma_liste<char> liste_caracteres;
```

Cette liste est très primitive et n'a qu'un usage pédagogique. Utilisez la liste de la STL `std::list` dans un vrai programme.

7.3 Spécialisation et surdéfinition des fonctions template

Cette section est technique et peut être sautée en première lecture.

Les fonctions en C++ peuvent partager le même nom. La résolution de la fonction à utiliser dépend uniquement du nombre et du type des arguments. Par exemple, on peut avoir décidé de spécialiser un template pour un type particulier.

```
template<typename T> fun (T a)
{
  // Corps de la fonction
}

template<typename T> fun(T* a)
{
  // Corps de la fonction
}

fun(int* a)
{
  // Corps de la fonction
}
```

La deuxième fonction `fun` est appelée une spécialisation partielle. La troisième est une spécialisation totale pour le type **int***. Les appels suivants sont résolus comme suit :

```
int n;
float* p;
int* q;
fun(n) ; // Utilise le template fun(T a)
fun(p) ; // Utilise le template fun(T* a)
fun(q) ; // Utilise le non template fun(int a)
```

Les règles de résolution sont extrêmement compliquées mais elles donnent en général ce que l'on souhaite : la fonction la plus spécifique est utilisée : bon nombre d'arguments et chaque argument a un ensemble de types acceptables inclus dans celui de toutes les autres fonctions. Par exemple si on a défini les template

```
template<typename T, typename U> fun (T a, U b)
{
  // Corps de la fonction
}

template<typename T, typename U> fun(T* a, U b)
{
  // Corps de la fonction
}

template<typename T, typename U> fun(T a, U* b)
```

```
{
  // Corps de la fonction
}
```

alors l'appel à `fun` dans

```
double* p;
double* q;
// ...
fun(p,q);
```

ne pourra être résolu : aucune des spécialisations n'est plus spécifique que l'autre pour **tous** les arguments : la deuxième spécialisation l'est plus pour le premier argument et la troisième plus pour le deuxième argument.

7.4 La programmation générique : l'exemple de la STL

Vous avez déjà vu sans le savoir un exemple de programmation générique : la STL qui est la bibliothèque standard de conteneurs en C++. La STL a été écrite et conçue par Alex Stepanov. La STL n'est pas orientée objet : il n'y a pas de classe de base abstraite dont hérite chacun des conteneurs. La STL est un exemple de programmation générique. Pour comprendre la STL, il faut avant tout comprendre la notion d'itérateurs.

Voyons d'abord comment on ferait une recherche linéaire sur un tableau contigû :

```
unsigned int find(tableau t, int u)
{
  unsigned int i;
  for(i=0; i<t.size()&& t[i]!=u;i++)
    ;// do nothing
  return(i);
}
```

et comment on ferait typiquement cette même recherche sur une liste simplement chaînée

```
link* find(liste l, int u)
{
  for(link* p=liste->head; p!=0 && p->val!=u;p=p->next)
    ;// do nothing
  return p;
}
```

La principale différence vient de la façon dont on itère sur une liste et de la façon dont on itère sur un tableau. On avance contigument dans la mémoire dans un tableau et on suit des liens dans une liste. Si on pouvait avoir une abstraction de l'itération, on pourrait utiliser les template pour programmer en une seule fois find pour tous les conteneurs. Cette abstraction est appelée itérateur et ce concept est la base du fonctionnement de la STL.

Un itérateur est une abstraction : tout type qui se comporte comme un itérateur est un itérateur. On peut voir un itérateur comme un pointeur qui se comporte intelligemment. Un itérateur est un

objet qui pointe vers un élément d'un conteneur. De plus, il existe un certain nombre d'opérations sur cet itérateur. Les plus utilisées sont les opérateurs $++$ et $--$ qui avancent et reculent respectivement l'itérateur. Les opérateurs de comparaison $==$ et $!=$ qui teste si deux itérateurs sont égaux. Enfin, s'inspirant de la notation de pointeur, l'opérateur unitaire préfixe $*$ retourne le contenu de l'élément vers lequel pointe l'itérateur. Pour obtenir un itérateur depuis un conteneur, on dispose des fonctions membres `begin`() qui pointe vers le premier élément du conteneur et `end`() qui pointe vers le non-élément qui se trouve derrière le dernier élément du conteneur.

Maintenant pour chaque conteneur de la STL, on définit un itérateur et en utilisant les template, on peut définir pratiquement n'importe quel algorithme sur les conteneurs en une seule fois pour tous les conteneurs !

```
template <class Iter, class ValT>
Iter find(Iter first, Iter last, ValT>
{
  Iter p;
  for(p=first;p!=last;p++) {
    if(*p==ValT)
      break;
  }
}
```

et maintenant on peut appeler find grâce à

```
std::vector<int> a;
std::list<float> b;
// Code
find(a.begin(),a.end(),0);
find(b.begin(),b.end(),0.0);
```

La plupart de ces algorithmes sont déjà dans la bibliothèque standard : utilisez-les ! Ce n'est pas peine de les reprogrammer (sauf à titre pédagogique pour apprendre le `C++`).

7.5 Conclusion

La programmation générique est un outil très puissant dont nous avons à peine effleurer la surface. Elle est cependant assez déroutante au début. Elle permet d'éviter beaucoup de copier coller sans perte de performance à l'exécution. Un de ses désavantages est qu'elle limite l'intérêt de la compilation séparée car les template, outil utilisé partout en programmation générique, doivent être placé dans les headers. La programmation générique est donc chère en temps de compilation.

8 Conseils et conclusion

Voici quelques conseils pour la suite. Limiter le nombre de variables globales. Cela rend la programmation plus simple au début mais la maintenance deviendra un cauchemar à terme. Séparer au maximum le code de haut-niveau et le code de bas-niveau. Créer des fonctions paramétrables et limiter le copier-coller. Maintenir le « single point of truth » : si je veux modifier mon programme, une petite modification devrait avoir à intervenir dans le moins de fichiers possibles et dans le moins d'endroits différents : si vous utilisez une constante quelque part, déclarez un paramètre **const int** `parametre`=10 et utiliser la constante `parametre`. Si vous modifiez ensuite cette valeur plus tard, vous n'aurez alors qu'à la modifier en un seul endroit.

Nous n'avons fait qu'effleurer le langage `C++` qui est un langage énorme. Le moindre livre d'introduction au `C++` fait facilement mille pages. Aussi, reste-t-il beaucoup de points qui n'ont pas pu être abordés ici. Parmi ces points, on trouve entre autres les exceptions qui sont un moyen de gérer les erreurs, l'héritage multiple, et les fonctions classes(qui peuvent remplacer les pointeurs sur les fonctions).