Introduction au C++

Guillaume Charpiat*

December 7, 2006

Avant-propos

Ce cours d'introduction au C++ est, comme son nom l'indique, un cours d'introduction: il ne prétend pas donner une liste exhaustive des possibilités offertes par le C++ mais seulement un avant-goût. D'autres cours/polys disponibles en ligne sont plus adaptés au perfectionnement des connaissances en C++.

Ce cours est par ailleurs pour l'instant incomplet: la toute fin, constitués de chapitres moins indispensables, n'a pas encore été rédigée.

Bonne lecture !

1 La mémoire

Avoir une idée du fonctionnement d'un ordinateur aide beaucoup à programmer correctement. C'est un passage obligé pour tous ceux qui désirent comprendre ce qu'est un pointeur.

En simplifiant, un ordinateur peut se résumer à une partie *mémoire* où sont stockées les informations et à son *processeur* qui se charge de modifier la mémoire selon les instructions reçues.

La programmation est donc l'art de donner des instructions efficaces qui vont gérer la mémoire de façon optimale, de façon lisible et pratique, en s'appuyant sur des outils construits par d'autres programmeurs.

Dans ce chapitre sont présentées les notions fondamentales de *variable*, *pointeur* et *référence*.

*guillaume.charpiat arrob@s ens.fr

1.1 Variables, adresses et contenus

La mémoire de l'ordinateur peut être vue comme un très grand espace découpé en cases. Pour s'y retrouver, les différentes cases de cet espace ont été numérotées, chaque case porte ainsi une adresse.

Lorsque l'on connaît l'adresse d'une case, on peut demander d'aller consulter le contenu de la case en question. On peut également bien entendu modifier le contenu de cette case.

Une variable dans le programme représente en quelque sorte une case de la mémoire. Toute variable porte un nom qui permet de l'identifier dans le programme, et possède une adresse qui désigne la case qui lui est associée. Pour regarder la *valeur* d'une variable, c'est-à-dire son *contenu*, il faut donc aller dans la mémoire jusqu'à la case désignée par l'adresse de cette variable, puis lire le contenu de cette case. De même pour modifier la valeur de la variable.

La quantité de mémoire réservée pour une variable, c'est-à-dire la taille de la case associée, ainsi que la façon de lire et d'écrire dans cette case, dépend du type de contenu que l'on s'attend à y trouver. En effet en pratique une case est formée d'un certain nombre de *bits* qui peuvent prendre les valeurs 0 et 1. Il faut donc que lorsque l'on accède à cette case on connaisse le nombre de *bits* (c-à-d la taille de la case) et que l'on sache interpréter correctement ce qui a été codé dans cette suite de 0 et de 1. Par exemple les nombres entiers positifs, les nombres avec virgule, les nombres négatifs, les chaînes de caractères, etc., sont tous représentés au final par des 0 et des 1, mais bien évidemment la façon de coder ces informations n'est pas la même.

Le type de contenu d'une variable, duquel dépend la façon de stocker la valeur de cette variable, est donc primordial. On l'appelle le *type* de la variable.

Avant d'utiliser une variable il faut la *déclarer*, c'est-à-dire préciser de quel *type* est cette variable. Cette *déclaration* permet à l'ordinateur de réserver une case dans la mémoire ayant les bonnes propriétés (la bonne taille) et d'en retenir l'adresse, désormais associée au nom de la variable.

Par exemple, si l'on souhaite utiliser une variable censée représenter un nombre entier, il faut tout d'abord lui donner un nom, disons `x`, ainsi que préciser son type. Le type le plus courant pour désigner un nombre entier est le type `int`. Une liste des types courants sera donnée plus tard. Dans le programme il faudra ainsi écrire:

```
int x;
```

avant la toute première utilisation de `x` afin que l'ordinateur sache ce que représente `x`. Dès ce moment-là, on peut accéder à l'adresse de `x` grâce à

l'opérateur "prendre l'adresse de" nommé `&`. L'adresse de `x` est ainsi `&x`.

Connaissant l'adresse `a` d'une variable, par exemple `a = &x`, on peut accéder à son contenu grâce à l'opérateur "prendre le contenu de" nommé `*`. Le contenu de la case numérotée `a` est ainsi `*a`, ou encore `*&x`.

Donnons un exemple simple. Soient `x` et `y` deux entiers, valant respectivement 4 et 5. On voudrait connaître leur somme et l'appeler `z`. La déclaration et l'instruction suivantes:

```
int x;
x = 4;
```

permettent de créer un entier nommé `x` (c-à-d de réserver une case qui aura un contenu de type `int`, et d'associer au nom `x` le numéro de cette case) puis de lui affecter la valeur 4 (c-à-d de remplir la case numérotée `&x` avec un code représentant le nombre 4). De même:

```
int y;
y = 5;
```

Et enfin, grâce à l'opérateur `+` qui est en fait une fonction prenant deux arguments (un devant lui et l'autre derrière) qui retourne la somme des deux entiers en question:

```
int z;
z = x + y;
```

Cette dernière instruction procède de la façon suivante: lire le contenu de la variable `x` et celui de la variable `y`, créer temporairement un autre entier (nommons-le `w`) dans lequel on écrit la somme des deux contenus, puis recopier la valeur de `w` (c-à-d le contenu de la case numéro `&w`) dans la case numéro `&z`, et enfin effacer la variable `w` qui n'est plus d'aucune utilité (c-à-d libérer la case numéro `&w`, ce qui n'a pas d'influence sur la case numéro `&z` puisque ce sont deux cases distinctes).

L'ordinateur exécutera les instructions dans l'ordre que l'on les lui a fournies. Du point de vue du programmeur, déclarer `x` avant `y` ou inversement n'a pas d'importance, il faut juste bien penser à déclarer les variables avant des les utiliser. Le morceau de programme ci-dessus peut donc être réécrit:

```
int x;
int y;
int z;
x = 4;
y = 5;
z = x + y;
```

ou encore, indifféremment pour le même résultat:

```
int z;
```

```
int x;
int y;
y = 5;
x = 4;
z = x + y;
```

La syntaxe des instructions et du programme est l'objet du prochain chapitre. Pour l'instant on se concentre sur les variables et la mémoire.

1.2 Les pointeurs

Un pointeur est une variable comme les autres, qui doit être déclarée, qui est associée à une case dans la mémoire et qui a donc une adresse, mais dont le contenu est particulier. En effet, au lieu de contenir une valeur comme un entier ou un caractère, un pointeur contient l'adresse d'une autre variable.

Par exemple si `x` est une variable de type `int`, dont l'adresse est `&x`, un pointeur `p` sur la variable `x` est une variable dont le contenu est `&x`. Bien évidemment, modifier le contenu de `p` (c-à-d modifier l'adresse qu'il stocke) ne change pas de place la variable `x`: dans ce cas, `p` pointera vers un autre endroit de la mémoire, c'est tout.

Dans le cas où `p` pointe sur `x`, pour connaître la valeur de `x` en ne connaissant que `p` il faudra donc faire `*p`, qui est de type `int` dans notre cas. Pour cette raison, le type de la variable `p` (qui est un pointeur vers un `int`) est `int*`. Pour déclarer `p` il faut donc taper:
`int* p;`
qui peut être comprise par le programmeur comme `(int *) p` (`p` est de type `int*`, c-à-d pointeur vers un `int`) ou `int (* p)` (le contenu de l'adresse stockée dans `p` est de type `int`). Puis pour lui affecter la valeur de `x`:
`p = &x;`
et enfin pour mettre le contenu de la case pointée par `p` dans un nouvel entier nommé `y`:

```
int y;
y = *p;
```

et ainsi les contenus de `x` et `y` seront similaires.

Attention: si l'on change maintenant la valeur de `x`, la valeur `*p` sera bien évidemment changée également puisque `p` pointe sur la case numéro `&x`. Par contre, cela ne changera pas la valeur de `y` puisque les cases numéro `&x` et `&y` sont deux cases distinctes qui n'ont aucun rapport sinon qu'il se trouve que leurs contenus sont (momentanément) identiques parce que l'on

a écrit la même chose dans les deux.

1.3 Les références

Une référence permet au moment de la déclaration d'une nouvelle variable de préciser que cette nouvelle variable n'est qu'un nouveau nom pour une variable existant déjà et qu'en conséquence il ne faut pas lui associer une nouvelle case mémoire. Pour déclarer une variable comme référence, il suffit de rajouter un `&` derrière le type de la variable au moment de sa déclaration. Nécessairement à ce même moment il faut aussi donner le nom de la variable originale dont la nouvelle ne sera qu'un nouveau nom. Exemple:

```
int x;
x = 3;
int& y = x;
```

Ici, `y` n'est qu'un nouveau nom pour la variable `x`, il s'agit de la **même** variable, elles ont la même adresse et partagent la même case mémoire. Conséquence: modifier l'une des deux modifie l'autre.

Les références s'avèreront utiles plus tard, notamment pour passer des variables à des fonctions.

1.4 Les tableaux

La notion intuitive de tableau est un ensemble de cases qui se suivent, numérotées, et qui sont toutes du même type. En `C++`, les cases d'un tableau de taille `n` sont numérotées de 0 à `n-1` et se suivent dans la mémoire, de sorte qu'il existe une façon simple d'accéder à la case suivante en connaissant l'adresse de la case précédente, ou même tout simplement d'accéder à n'importe quelle case en connaissant que l'emplacement de la première. La taille d'une case dépend du type de son contenu et peut être connue par le programmeur via la fonction `sizeof()`. Ainsi `sizeof(int)` renvoie le nombre de *octets* nécessaires au stockage d'un objet de type `int`. Dans un tableau de `int`, l'accès à la case suivante se fait donc en ajoutant cette taille à l'adresse de la case courante. Compte tenu de ces remarques, un tableau en `C++` est simplement donné par l'adresse de sa première case et est donc un pointeur vers cette case.

Ainsi un tableau de `int` est de type `int*`.

Il existe une autre syntaxe pour les tableaux: au lieu de déclarer:

```
int* p
```

on peut écrire: `int p[]` dans le cas particulier des arguments d'une fonction. Je vous conseille d'oublier cette dernière.

1.5 Allouer avec `new`

Si `p` est un pointeur de type `int*`, a été initialisé n'importe comment et pointe donc vers une case quelconque d'adresse `p`, rien ne garantit que cette case d'adresse `p` est effectivement de type `int`, ni que les cases suivantes ont été effectivement réservées pour remplir un tableau. Il ne suffit pas de *déclarer* un pointeur pour avoir un tableau, il faut en effet lui réserver de la place mémoire pour stocker les cases supplémentaires. Pour cela on utilise le mot-clé `new` de la façon suivante:

```
int* p;
p = new int[42];
```

pour trouver un bloc de 42 cases consécutives de type `int` et conserver l'adresse de la première de ces cases dans `p`. La i-ème case du tableau est alors désignée par `p[i-1]` (attention, la première case est la numéro 0), de `p[0]` à `p[42-1]`. Pour mettre l'entier 100 dans la troisième case on peut ainsi faire:

```
p[2] = 100;
```

L'accès à une case hors du tableau (typiquement `p[42]` qui serait la 43-ième case) *peut* provoquer une erreur (de façon aléatoire). Il faut donc faire très attention à l'allocation des tableaux.

On peut également utiliser `new` sans les crochets afin d'allouer une seule variable et de retourner son adresse:

```
int* p;
p = new int;
```

La différence avec `p = new int[1];` est qu'il ne faudra pas appeler `delete[]` (décrit dans le paragraphe suivant) pour libérer l'espace mémoire; il faudra appeler `delete` à la place.

1.6 Désallouer avec `delete[]`

Lorsque l'on n'a plus l'utilité d'un tableau, on peut libérer la mémoire qu'il occupe. Cela est d'autant plus intéressant que le tableau est gros. Une règle standard de la programmation est de systématiquement libérer la mémoire des tableaux dès qu'ils ne sont plus utiles. Pour désallouer un tableau `p` créé par:

```
int* p;
p = new int[42];
```
il faut taper:
```
delete[] p;
```
Attention, appliquer `delete[]` à un pointeur qui n'a pas été alloué avec `new` auparavant provoquera une erreur.

1.7 Autres tableaux

Il existe une autre sorte de tableaux. La différence est que les tableaux statiquement alloués, à l'opposé des précédents, dynamiques, sont stockés dans une partie restreinte de la mémoire de l'ordinateur. Les gros tableaux doivent donc systématiquement être déclarés et alloués en tant que tableaux "dynamiques".

La déclaration et l'allocation des tableaux "alloués statiquement" se fait de la façon suivante:
```
int t[12];
```
et leur utilisation est semblable aux autres (`t[0] = 1;`, etc.). Ils ne doivent pas être désalloués avec `delete[]`.

On peut également les initialiser de façon plus pratique:
```
int t[4] = {25, 86, 902, 7};
```
même de façon incomplète:
```
int t[100] = {25, 12, 45};
```
(et dans ce cas les nombres non indiqués sont remplacés par 0).

2 Structure et syntaxe d'un programme simple

2.1 Les types de base

Chaque variable a un type. Parmis les types les plus courants l'on trouve:

- `int` : entier (standard, éventuellement négatif, compris entre deux bornes fixées)
- `short` : entier court (l'intervalle des valeurs possibles est beaucoup plus restreint)
- `long` : entier long (l'intervalle est plus grand que pour `int`)
- `float` : nombre (éventuellement négatif) avec une virgule (la précision est limitée à quelques chiffres significatifs)

- `double` : pareil que `float` mais avec une plus grande précision (c'est le type standard pour les nombres à virgules)
- `bool` : booléen (valant soit `true` soit `false`)
- `char` : un caractère (un seul), par exemple 'a' ou 'b' ou '9' ou '.'

Les entiers sont notés simplement (par exemple 2 ou 43) tandis que les nombres à virgule comme 3.14 ou 185.78342 doivent comporter un point (le symbole pour la virgule) pour être reconnus comme tels. Ainsi `4.` est un `double` tandis que 4 est un `int`. Les caractères sont notés entre apostrophes.

On peut bien entendu ajouter à cette liste les pointeurs vers de telles variables, par exemple:
`double*` : pour un pointeur vers un `double`, ou un tableau de `double`
`char*` : chaîne de caractères. Il y a une syntaxe particulière pour les chaînes de caractères: `char* chaine = "blabla";` (entre guillemets).

et de même:
`double**` : un pointeur vers un pointeur vers un `double` (c-à-d stockant l'adresse d'un pointeur vers un `double`), ou un pointeur vers un tableau de `double`, ou un tableau de tableaux de `double` (c-à-d un tableau de pointeurs vers des `double`)...

et ainsi de suite.

2.2 Déclaration

Comme on l'a dit précédemment, toute variable doit être déclarée avant d'être utilisée. Par exemple:

```
int x;
x = 2;
```

ou encore:

```
double* t;
t = new double[12];
t[0] = 33;
```

Les noms utilisées pour désigner une variable peuvent comporter des lettres (sans accent), majuscules comme minuscules, des chiffres, le symbole '_', mais ne doivent pas commencer par un chiffre. Les lettres minuscules et majuscules sont différenciées; ainsi `a` et `A` désignent deux variables différentes.

Il n'y a pas d'autre contrainte sur le nom d'une variable, excepté que sa longueur doit être raisonnable.

Il existe un raccourci pour déclarer plusieurs objets de même type:
`int x, y;`
est synonyme de:
`int x; int y;`

Il existe aussi un raccourci pour déclarer une variable et l'affecter au lieu de le faire en deux temps:
`int x = 2;`
est synonyme **dans le cas des types de base seulement** de:
`int x; x = 2;`
De même pour:
`double* p = new double[12];`
Attention pour les objets des classes ces deux syntaxes ont un sens différent !

Les références doivent être initialisées en même temps qu'elles sont déclarées:
```
int x;
int& y = x;
```

2.3 Variables constantes

Il existe un mot-clé qui sert à préciser, lors de sa déclaration et initialisation, qu'une variable ne doit jamais être modifiée par la suite. Evidemment dans ce cas la variable en question doit être initialisée en même temps qu'elle est déclarée, ce sans quoi elle ne pourrait jamais avoir de valeur par la suite. La syntaxe est:
`const int x = 12;`
Toute tentative du programmeur de changer la valeur de `x` sera soldée par un refus du compilateur.

Les pointeurs étant des variables comme les autres, ils peuvent eux aussi être constants:
```
double y;
double const* p = &y;
y = 3.4;
y = 0.1;
```
Ici `p` est un pointeur constant vers un `double`, il ne pourra pas pointer vers une autre variable.

Il ne faut pas condondre cet exemple avec un pointeur (normal) vers une variable qui elle est `const`:

```
const double y = 3.14;
const double * p;
p = &y;
const double z = 2.18;
p = &z;
```

2.4 Blocs et durée de vie des variables

Une instruction finit par un point-virgule. Elle peut servir à déclarer un objet, lui donner une valeur, appeler une fonction, appeler plusieurs fonctions de façon imbriquée, etc. Dire qu'une instruction finit par un point-virgule est sans doute une bonne ébauche de définition de l'instruction...

Un bloc d'instructions est délimité par des accolades. Toute instruction fait naturellement partie d'un bloc à un certain niveau (par exemple elle fait partie d'une boucle ou d'une fonction). Le bloc lui-même peut être considéré comme une grosse instruction. Il n'y a pas besoin d'ajouter un `;` après un bloc.

Une variable déclarée à l'intérieur un bloc est automatiquement détruite à la sortie du bloc. On ne peut alors plus la consulter. Si l'on veut que la variable continue d'exister après la fin de ce bloc, il suffit de la déclarer *avant* ce bloc. Exemple: dans

```
{ int x = 12; }
```

on ne peut plus consulter `x` après la fin du bloc, la variable a disparu de la mémoire. Mais l'on peut faire ainsi:

```
int y;
{ y = 12; }
int z = y + 1;
```

puisque dans ce cas la variable `y` existait déjà avant le bloc.

Dans le cas des tableaux, le problème se pose différemment, car effacer un pointeur ne signifie pas effacer la variable sur laquelle il pointe:

```
double* q;
{ double* p;
  p = new double[2];
```

```
  p[0] = 2.;
  p[1] = 4.5;
  q = &(p[0]);
}
double x = q[0] + q[1];
```

Ici, au sortir du bloc, `p` a été effacé, c-à-d que la variable `p` qui contenait l'adresse d'un tableau a été effacée, mais pas le tableau qui lui a toujours pour adresse la valeur qu'avait `p`. Heureusement, on a sauvé cette adresse dans un autre pointeur, `q`, ce qui permet d'accéder de nouveau au tableau. Le seul moyen d'effacer un tableau est d'appeler `delete[]` explicitement:
`delete[] q;`
ce qui libère le tableau qui a pour adresse la valeur de `q`.

Attention, si deux pointeurs `p` et `q` pointent vers un même tableau et que le tableau est effacé par `delete[] p;` alors `q` continue de pointer à la même adresse mais il n'y a plus d'espace alloué à cet endroit-là pour un tableau.

Notons également que l'on peut déclarer localement une variable portant un nom qui existe déjà. Dans ce cas, à l'intérieur du bloc, la variable qui sera considérée sera celle déclarée dans le bloc, et à la sortie du bloc l'ancienne sera à nouveau considérée alors que la variable locale aura disparu. Par exemple:

```
int a = 3;
int b;
{ int a = 4;
  b = a;
}
int c = a;
```

A la fin de cette série d'instructions, `b` vaudra 4 tandis que `c` vaudra 3.

2.5 Initialisation et affectation

On appelle *affectation* et *initialisation* le fait de donner une valeur à une variable précédemment déclarée ou de la construire. Dans la suite, quand l'on étudiera les classes, on verra que l'affectation et l'initialisation sont deux choses différentes, mais confondues pour les types de base comme `int`.

L'initialisation consiste à donner la valeur de la variable au moment même où on la déclare, ce qui permet éventuellement de *construire* cette

variable de façon différente en fonction de paramètres fournis à ce moment-là. Exemple:
`int a(2);`
s'écrit de façon équivalente:
`int a = 2;`
Pour des objets plus compliqués que des entiers, comportant des tableaux ou autres choses complexes, ce moment peut être propice à allouer des tableaux ou faire d'autres opérations utiles à la construction de l'objet. La déclaration et l'initialisation de l'objet peuvent ainsi avoir lieu en même temps afin de gérer au mieux l'allocation de la mémoire nécessaire pour stocker la nouvelle variable.

L'affectation en revanche peut avoir lieu à un moment quelconque du programme (pourvu que la variable ait déjà été déclarée et qu'elle ne soit pas `const`). Le symbole de l'affectation est `=`. On peut considérer l'affectation comme une fonction prenant deux paramètres (une variable à gauche et une valeur à droite).

```
int a;
a = 2;
```

2.6 Les fonctions

Une fonction est un bloc d'instructions séparé du reste du programme, identifié par un nom qui lui est propre (comme les variables), pouvant prendre des variables comme arguments (= paramètres) et pouvant éventuellement en retourner une.

Pour *déclarer* une fonction nommée `pourcent`, prenant en argument une variable de type `int` et renvoyant une autre de type `double`, la syntaxe est:
`double pourcent(int x);`
Il faut en effet donner un nom au paramètre afin de pouvoir l'utiliser ultérieurement dans la fonction.

Pour *définir* la fonction, c-à-d préciser quel bloc d'instructions est exécuté quand la fonction est appelée:

```
double pourcent(int x){
  double y;
  y = x / 100.;
  return y;
}
```

L'instruction `return y;` précise que quand la fonction sera appelée il faudra retourner la valeur de `y`. Evidemment le type de la variable derrière `return` doit être le même type que celui annoncé au dans la déclaration devant le nom de la fonction (ici `double`).

Pour être plus précis: supposons que la fonction ci-dessus soit appelée à un certain moment dans le programme, dans le bloc:

```
double b;
int a = 36;
b = pourcent(a);
```

Cette dernière ligne est équivalente au bloc suivant:

```
{
  int x = a;
  double y;
  y = x / 100.;
  b = y;
}
```

Les variables `x` et `y` sont donc propres au bloc correspondant à la fonction `pourcent` et n'existent plus à la sortie de ce bloc. Comme ces variables sont locales à ce bloc, il se peut qu'elles portent le même nom que d'autres variables définies précédemment dans le programme; il n'y aura pas d'interactions entre ces variables synonymes considérées comme différentes. Une autre chose à savoir est que l'on sort de la fonction quand on rencontre le mot-clé `return`, c-à-d que s'il y a des instructions dans la fonction après le retour de la valeur par `return`, elles ne seront pas exécutées.

Une fonction peut prendre plusieurs arguments, mais ne peut retourner qu'une seule variable. Exemple:

```
double moyenne(int a, int b){
  double c = 0.5 * (a + b);
  return c;
}
```

Le bout de code suivant:

```
int x = 0;
double m;
m = moyenne(x, 1);
```

est équivalent à:

```
int x = 0;
double m;
{
  int a = x;
  int b = 1;
  double c = 0.5 * (a + b);
  m = c;
}
```

Une fonction peut aussi ne rien retourner du tout. La convention est que le type de "rien" est `void`. Une fonction peut aussi ne prendre aucun argument. Exemple:

```
void riendutout(){
  int a = 12;
}
```

Cette dernière fonction est équivalente à:

```
void riendutout(){
  int a = 12;
  return;
}
```

2.7 Fonction `main`

Il existe une fonction particulière nommée `main`, ne prenant aucun argument (on verra plus tard qu'il est possible de donner des arguments à cette fonction mais pour l'instant on considère simplement qu'elle n'en a pas), et renvoyant un `int`. Dans tout programme il doit y avoir une telle fonction car c'est la fonction qui est appelée par l'ordinateur quand il exécute le programme. L'ordinateur exécute pas à pas les instructions de cette fonction (en faisant éventuellement appel à d'autres fonctions si ces instructions font

explicitement appel à ces fonctions) puis le programme est terminé quand la fin de la fonction `main` est atteinte.

Dans tout programme l'on trouve donc un passage de la forme:

```
int main() {

   // suite d'instructions

  return 0;
}
```

Les différentes fonctions sont déclarées et/ou définies les unes à la suite des autres dans le programme, dans n'importe quel ordre. La seule contrainte est que lorsqu'une fonction est appelée par une autre, elle doit avoir été déclarée avant (mais pas nécessairement définie, la définition pouvant venir après). C'est assez similaire au cas des variables, qui doivent avoir été déclarées avant d'être utilisées.

Ainsi deux fonctions `f` et `g` peuvent s'appeler mutuellement.

```
int f();

int g(int a){
  int b = f();
  return b+a;
}

int f() {
  return g(0);
}
```

3 Quelques instructions courantes

3.1 Opérateurs

On appelle *opérateurs* des fonctions à deux arguments donc le premier a la particularité de se trouver *devant* la fonction lorsqu'on l'appelle. Par exemple la fonction qui retourne la somme de deux entiers, comme dans:

```
int a = 3 + 4;
```

peut être vue comme une fonction `operator+()` dont la déclaration ressemblerait à:

```
int operator+(int x, int y);
```

Le symbole = de l'affectation désigne également une fonction. Par exemple dans:

```
int a;
a = 4;
```

l'affectation peut être vu comme un appel à la fonction:

```
int operator=(int& x, int y);
```

qui prendrait comme argument une référence sur la variable de gauche (pour pouvoir modifier son contenu) ainsi qu'une valeur de type `int`. La fonction renvoie un `int` (une copie du contenu de `x`) afin de pouvoir utiliser l'écriture condensée `b = a = 4;` si `b` est un autre `int` dont on veut qu'il vale également 4.

Les opérateurs courants sur les `int` sont:

- l'addition: `a + b`
- la soustraction: `a - b`
- la multiplication: `a * b`
- la division: `a / b` (attention cela renvoie un entier, donc tronque le résultat)
- l'affectation: `a = b;`
- l'incrémentation: `a += b;` (équivalente à `a = ( a + b );`)
- la décrémentation: `a -= b;`
- la multiplication: `a *= b;`
- la division: `a /= b;`
- prendre l'opposé: `-a` (appelé moins unitaire)
- l'incrémentation de 1: `a++;` (équivalente à `a += 1;`)
- la décrémentation de 1: `a--;` (équivalente à `a -= 1;`)
- le modulo: `a % b` retourne a modulo b.

Des opérateurs similaires existent sur les autres types (`double`, `float`, `long`, etc.). Dans le cas où un opérateur courant est appelé sur des variables de types différents (par exemple un `int` et un `double`), le compilateur choisit l'opérateur sur le type qui a le plus de précision (donc ici `double`), sauf bien entendu s'il s'agit d'un opérateur d'affectation auquel cas le membre de droite sera copié dans une variable temporaire du bon type. Ainsi:
`2 / 5.` appelle l'opérateur de division sur les `double`
`int a; a = 4.3;` convertit `4.3` en un `int` de valeur `4` et affecte `4` à `a`.

3.2 Test `if`

Parfois l'on souhaiterait exécuter une série d'instructions seulement si une certaine condition est vérifiée, et éventuellement une autre série dans le cas contraire, avant de continuer (d'autres instructions exécutées dans les deux cas).

```
bool vraioufaux = true;
if (vraioufaux){
  ... // bloc exécuté si vraioufaux est true
} else {
  ... // bloc exécuté si vraioufaux est false
}

... // le reste est exécuté ensuite dans les deux cas
```

Le deuxième bloc n'est pas obligatoire:

```
bool vraioufaux = false;
if (vraioufaux){
  ... // bloc exécuté si vraioufaux est true
}

... // le reste est exécuté ensuite dans les deux cas
```

Quelques exemples de booléens:

```
bool sontilsegaux = ( 12/4 == 3 );
bool estilplusgrand = (42 > 5);
```

```
bool estilplusgrandouegal = (42 >= 5);
bool sontilsdifferents = (42 != 5);
bool contraire = !vraioufaux;
```
Le symbole `!` est la négation (transforme `true` en `false` et réciproquement).
```
bool lesdeuxsontilsvrais = (sontilsdifferents && estilplusgrand);
bool lundesdeuxaumoinsestilvrai = (sontilsdifferents || estilplusgrand);
```

Attention, le test d'égalité entre deux variables de même type est `(a == b)` et non pas `(a = b)` qui est l'affection du contenu de `b` dans la variable `a`.

3.3 Boucle `while`

Pour répéter un bloc d'instruction tant qu'une certaine condition est vérifiée:

```
int a = 1;
bool continuer = true;
while (continuer) {
  a = 3 * a + 1;
  continuer = (a < 1000);
}
```

Attention, si une variable est déclarée à l'intérieur du bloc d'instructions de la boucle `while`, cette variable locale est effacée à chaque fois que la fin de la boucle est atteinte, et redéclarée à chaque fois que le bloc est exécuté.

3.4 Boucle `for`

Pour répéter un bloc un nombre de fois prédéterminé. Exemple qui calcule la somme des entiers de 3 à 77:

```
int a = 0;
for (int i = 3; i<78; i++){
  a = a + i;
}
```

Ce bloc est équivalent à:

```
int a = 0;
{
```

```
  int i = 3;
  while (i<78) {
    a = a + i;
    i++;
  }
}
```

L'instruction `i++;` signifie `i = i + 1;` qui peut aussi être abrégée en `i += 1;`. Attention, la variable `i` n'existe plus une fois qu'on est sorti de la boucle `for`.

3.5 Quitter ou continuer avec `break` et `continue`

L'on peut quitter à tout moment une boucle avec `break;`. L'on peut également à tout moment passer directement à la fin de la boucle avec `continue`.

```
int a = 0;
for (int i = 0; i<100; i++) {

   if (a < i)
      continue;

   a = 3 * a + i;

   if (a > 12000)
      break;
}
```

Dans cet exemple, si `(a < i)` alors l'on passe directement au tour suivant. Sinon, on modifie la valeur de `a` comme demandé et si `(a > 12000)` alors on quitte la boucle `for`.

Ces mot-clés fonctionnent également dans les boucles `while`. Malheureusement, si deux boucles sont imbriquées l'une dans l'autre, il n'y a pas de moyen de quitter les deux boucles en une seule fois. Un `break` dans la boucle à l'intérieur fera quitter cette boucle mais pas la boucle externe. Exemple de boucles imbriquées:

```
int a = 1;
for (int i = 0; i<10; i++)
   for (int j = 0; j<5; j++) {
```

```
    if (i < j)
      break;
    a += 1;
  }
```

3.6 Sélection `switch`

Une série de tests d'égalité portant sur une même variable peut être élégamment remplacée par un `switch`:

```
char uncaractere = 't';
int resultat;
switch (uncaractere) {
case 'a':
  resultat = 0;
  break;
case 'b':
  resultat = 1;
  break;
case 'c':
  resultat = 7;
case 'd':
  resultat = 3;
  break;
default:
  resultat = -1;
}
```

Les différentes possibilités sont énumérées avec `case`, le mot-clé `default:` désignant tous les cas non prévus. Le mot-clé `break` permet de quitter le `switch`, il est important. Par exemple son oubli dans le cas `'c'` fait que si `(uncaractere == 'c')`, alors les instructions `resultat = 7;` **et** `resultat = 3;` seront exécutées (toutes les instructions jusqu'au premier `break` ou la fin du `switch`).

4 Plus de détails sur les fonctions

4.1 Surcharge

Plusieurs fonctions différentes peuvent porter le même nom, à la condition que la liste des types de leurs arguments ne soit pas la même.

Ainsi les différentes fonctions suivantes peuvent être déclarées dans un même programme:

```
void f();
voif f(int a);
void f(double a);
void f(int a, double b);
double f(bool b);
```

Lorsqu'une fonction `f` sera appelée dans le programme, le compilateur choisira celle qui aura les arguments du bon type.

Attention, il n'est pas possible de déclarer deux fonctions ayant les mêmes arguments même si elles renvoient des types différents. Ainsi l'exemple suivant provoque une **erreur**:

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

4.2 Paramètres par défaut

Il est possible de préciser la valeur par défaut de l'argument d'une fonction, c-à-d que si un appel à la fonction "oublie" de préciser la valeur d'un argument, une valeur pré-fixée lui sera assignée.

On ne peut donner une valeur par défaut qu'aux derniers arguments d'une fonction, c-à-d qu'un argument ne peut pas avoir de valeur par défaut si le suivant n'en a pas. De même, lors de l'appel de la fonction, si on souhaite ne pas préciser l'un des arguments, on ne pourra pas préciser les suivants.

Syntaxe:

```
void f(int a, double b, bool c = true);

void g(double x = 3.14, double y = 2.78);
```

Dans cet exemple, un appel à `f(0, 0.1)` sera équivalent à un appel à `f(0, 0.1, true)`. `Un appel à \verb`g(0.)+ sera équivalent à `g(0., 2.78)` et un appel à `g()` à `g(3.14, 2.78)`. On ne peut pas préciser la valeur de `y` sans donner celle de `x`.

Si la déclaration et la définition de la fonction sont séparées, les valeurs par défaut doivent être précisées dans la déclaration mais pas dans la définition.

4.3 Utilisation des références

Lors de l'appel d'une fonction, les arguments sont **copiés**, c-à-d que la fonction manipule de nouvelles variables ayant les mêmes valeurs que celles données en arguments, mais que ces variables sont **différentes** des précédentes et locales au bloc d'instruction de la fonction. Exemple: si la fonction `ajoute_un` est définie ainsi:

```
void ajoute_un(int a){
  a++;
}
```

alors l'appel suivant:

```
int x = 5;
ajoute_un(x);
```

est équivalent au bloc:

```
int x = 5;
{
 int a = x;
 a++;
}
```

ce qui fait que la valeur de `x` ne sera pas modifiée !

Si l'on veut changer la valeur de la variable `x`, il faut passer la variable par référence afin que la fonction manipule la même variable et non pas une

copie. La fonction ressemble alors à:

```
void ajoute_un(int& a){
  a++;
}
```

et dans ce cas le bloc précédent devient équivalent à:

```
int x = 5;
{
 int& a = x;
 a++;
}
```

Ici, comme `a` est une référence sur la variable `x`, `a` n'est qu'un autre nom pour la même variable. Modifier `a` affecte donc `x` puisqu'il s'agit de la même variable.

Une fonction peut aussi **retourner** une référence. Cela peut s'avérer utile si l'on cherche à renvoyer une variable que l'on souhaite modifier. Supposons par exemple que nous disposions de deux variables `x` et `y` et que nous voulions manipuler la plus grande des deux. La fonction:

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

retourne une référence vers le plus grand de ses arguments (qui doivent aussi être passés en référence si on ne veut pas retourner une référence vers une simple copie des arguments qui soit locale à la fonction). L'on peut alors appeler la fonction ainsi:

```
int x = 4;
int y = 12;
plusgrande(x, y) += 3;
```

Ce bloc aura de fait le même comportement que celui-ci:

```
int x = 4;
int y = 12;
{
  int& a = x;
  int& b = y;
  if (a>b)
    a += 3;
  else
    b += 3;
}
```

qui lui-même aura le même comportement que:

```
int x = 4;
int y = 12;
if (x>y)
    x += 3;
  else
    y += 3;
```

4.4 Utilisation des références constantes

L'inconvénient de passer une variable par référence est que potentiellement la fonction peut modifier la valeur de cette variable. Si l'on souhaite s'assurer que ce ne sera pas le cas, il suffit de préciser que la référence est constante.

```
void rien(const int& a){
  ...
}
```

Une telle fonction ne pourra pas modifier la variable `a` et ne pourra elle-même qu'appeler des fonctions sur `a` qui ne la modifient pas (c-à-d pas de fonctions qui prennent `a` en référence sans préciser `const`). Tout essai de modifier la valeur de `a` dans la fonction `rien` se soldera par un message d'erreur du compilateur.

Passer des arguments en `const` référence au lieu de les recopier (comme les arguments normaux) permet justement d'économiser ce temps de copie.

4.5 Pointeur vers une fonction

Une fonction peut être passée en argument, comme une variable normale. Pour ce faire on manipule un *pointeur* vers la fonction. Un pointeur `p` vers la fonction suivante:

```
double f(int a, bool b);
```

est du type:

```
double (*p)(int, bool);
```

Ainsi, les lignes suivantes:

```
double (*p)(int, bool);
p = &f;
double x = (*p)(1, false);
```

déclarent un pointeur vers une fonction du même type que `f`, le font pointer sur `f`, puis appellent la fonction pointée.

Pour passer une fonction d'un certain type comme argument, on procèdera donc ainsi:

```
bool test_si_resultat_positif(double (*p)(int, bool), int x){
  double z = (*p)(x, true);
  return (z>0);
}
```

et l'appel se fera simplement:

```
bool b = test_si_resultat_positif(&f, 3);
```

4.6 Champ statique

Les variables déclarées dans une fonction sont locales. A chaque appel de la fonction, elles sont déclarées, utilisées, puis détruites. En conséquence, il n'est pas possible de consulter, lors d'un deuxième appel à une fonction, la valeur qu'avait une des variables locales à la fin du premier appel.

Il est possible cependant de préciser qu'une variable déclarée dans une fonction ne doit pas être détruite et redéclarée à chaque appel, mais qu'il

doit toujours s'agir de la *même* variable. Pour ce faire il suffit de préciser le mot-clé `static` devant la variable lors de sa déclaration et initialisation. L'initialisation n'aura lieu que lors du premier appel à la fonction; au deuxième appel, l'instruction de déclaration/initialisation ne sera pas exécutée.

Ainsi pour compter le nombre de fois qu'une fonction a été appelée:

```
void f(){
  int a = 1;   // variable locale déclarée et effacée à chaque appel
  static int nb = 1;   // variable gardée en mémoire
  nb ++;
}
```

4.7 Arguments de la fonction `main`

Le type habituel de la fonction `main` (celle qui est appelée quand on lance le programme) est:

```
int main(){
 ...
}
```

Il est cependant possible de passer des arguments à cette fonction. Si le programme s'appelle `prog`, on le lance habituellement sous Linux dans un terminal via la commande:

```
./prog
```

Il est possible de passer des arguments en ligne de commande, comme:

```
./prog 2 coucou 3.14
```

Pour récupérer ces arguments, l'on doit utiliser la fonction `main` de la façon suivante:

```
int main(int argc, char* argv[]){
  ...
}
```

A ce moment-là, l'entier `argc` contiendra le nombre d'arguments + 1, et le tableau `argv` contiendra les chaînes de caractères représentant les arguments passés en ligne de commande. Ainsi dans l'exemple précédent, `argc` vaudra 4, `argv[0]` vaudra `"2"` , `argv[1]` vaudra `"coucou"` et `argv[2]`, `"3.14"`.

5 Les classes

5.1 Syntaxe

Une classe est une structure particulière qui permet d'englober plusieurs variables différentes sous un même nom.

Par exemple, si vous manipulez un certain nombre de tableaux de `double` et que vous n'avez pas envie de devoir créer pour chaque tableau une variable qui représente sa taille, vous pouvez créer une classe nommée `Tableau`, qui aura deux *champs*: un nommé `tab`, pour le tableau (un `double*`) et un autre nommé `n`, pour sa taille (un `int`):

```
struct Tableau {
  double* tab;
  int n;
};
```

Attention à **ne pas oublier le point-virgule à la fin de la classe !** De même que les fonctions, la classe `Tableau` doit avoir été déclarée avant d'être utilisée, séparément du reste du code (comme une fonction). De même que pour les fonctions, on peut distinguer la déclaration de la classe de sa définition (qui elle était donnée ci-dessus). La syntaxe de la déclaration, comme pour les fonctions, consiste à troquer le bloc d'instruction pour un point-virgule:

```
struct Tableau;
```

Dans le programme vous pourrez utiliser simplement cette classe de la façon suivante:

```
Tableau A;
A.n = 4;
```

```
A.tab = new double[4];

for (int i=0; i<A.n; i++)
  A.tab[i] = 0.;

Tableau B;
B.n = A.n + 1;
B.tab = new double[B.n];
```

Le nom de la classe est un type comme les autres, de même que `int`, `double` et les autres types de base. Ici on a déclaré une variable `A` (*objet*) de type `Tableau` (*classe de l'objet*). L'ordinateur lui associe automatiquement deux variables propres (*champs*) `n` et `tab`, de type respectif `int` et `double*`. L'accès à ces *champs* se fait en ajoutant derrière l'*objet* un point suivi du nom du champ.

L'exemple précédent est équivalent, d'une certaine façon, à:

```
int An;
double* Atab;

An = 4;
Atab = new double[4];

for (int i=0; i<An; i++)
  Atab[i] = 0.;

int Bn;
double* Btab;

Bn = An + 1;
Btab = new double[Bn];
```

On peut donc déjà voir l'intérêt des classes comme raccourci pour la déclaration des variables ainsi que comme généralisateur qui permet d'avoir plusieurs *objets* qui suivent un même schéma (chacun ayant ses propes champs, de même nom).

Nous verrons que l'intérêt des classes est bien plus vaste.

5.2 Fonctions membres

De même que l'on peut placer des variables dans une classe (qu'on appelle alors *champs*), on peut y placer des fonctions. Exemple:

```
struct Tableau {

  double* tab;
  int n;

  void alloue(int a){
    n = a;
    tab = new double[n];
  }

};
```

Une telle fonction est appelée *fonction membre* de la classe *Tableau*. Remarquez qu'elle peut manipuler directement les champs `n` et `tab` de la classe sans préciser à quel objet ils appartiennent: ils appartiennent à l'objet qui appelle la fonction. On appelle une fonction membre de la façon suivante:

```
Tableau A;
A.alloue(4);
```

Cette dernière ligne signifie donc:

```
{
  int a = 4;
  A.n = a;
  A.tab = new double[A.n];
}
```

La différence entre les fonctions membres d'une classe et les fonctions déclarées en dehors de cette classe est que les premières peuvent accéder directement aux champs de l'objet qui les appelle tandis que les deuxièmes doivent avoir l'objet en question comme argument. Par exemple une fonction qui initialise à 0 tous les champs d'un objet de la classe `Tableau` serait:

```
void initialise(Tableau X){
  for (int i = 0; i<X.n; i++)
    X.tab[i] = 0.;
}
```

et s'appelerait de la façon suivante:

```
Tableau A;
initialise(A);
```

tandis que l'équivalent avec une fonction membre consiste à ajouter **dans la définition de la classe**:

```
void zero(){
  for (int i = 0; i<n; i++)
    tab[i] = 0.;
}
```

et celle-ci s'appelle de la façon:

```
Tableau A;
A.zero();
```

Le code de la page précédente s'écrit maintenant de façon beaucoup plus laconique:

```
Tableau A;
A.alloue(4);

A.zero();

Tableau B;
B.alloue(A.n + 1);
```

De même que la fonction `initialise` peut être découpée en deux, à savoir sa déclaration:

```
void initialise(Tableau X);
```

suivie de sa définition:

```
void initialise(Tableau X){
  for (int i = 0; i<X.n; i++)
    X.tab[i] = 0.;
}
```

on peut également séparer la **déclaration à l'intérieur de la classe** d'une fonction membre:

```
void zero();
```

de sa **définition** alors **à l'extérieur de la classe**:

```
void Tableau::zero(){
  for (int i = 0; i<n; i++)
    tab[i] = 0.;
}
```

L'indicateur `Tableau::` est nécessaire pour préciser à quelle classe appartient cette fonction.

5.3 Champs privés et publiques

Dans une classe, les champs et les fonctions membres peuvent être protégés. Ils sont dans ce cas accessibles seulement par les fonctions membres de cette classe, et on ne peut pas les appeler ailleurs.

L'intérêt est d'obliger le programmeur à passer par certaines fonctions pré-définies afin qu'il ne puisse pas manipuler à sa guise les champs d'un objet. Imaginons par exemple dans l'exemple précédent qu'un programmeur fasse:

```
Tableau A;
A.n = 5;
A.tab = new double[4];
```

Dans ce cas, le champ `n` ne correspond plus à la taille du tableau, ce qui est embêtant puisque toute la classe était construite dans cette optique. Si le programmeur ajoute maintenant:

```
A.zero();
```

il y aura une erreur à cause d'une tentative d'accès à la cinquième case `A.tab[4]` pour lui affecter `0.` alors que le tableau n'a que `4` cases !

L'idéal serait alors d'avoir une classe de la forme (avec la même définition de `alloue` que prédédemment):

```
struct Tableau {

  double* tab;
  int n;

  void alloue(int a);
  int consulte_n();
  double consulte_element(int i);
  void modifie_element(int i, double x);

};



int Tableau::consulte_n(){
  return n;
}

double Tableau::consulte_element(int i){
  return tab[i];
}

void Tableau::modifie_element(int i, double x){
  tab[i] = x;
}
```

où le programmeur en dehors de la classe aurait accès aux quatre fonctions (pour respectivement allouer le tableau avec la taille correspondante, consulter la valeur de `n` sans pouvoir la modifier, consulter et modifier la valeur d'un élément du tableau sans avoir accès directement au champ `tab` afin de ne pas pouvoir changer le réallouer n'importe comment sans changer `n` en conséquence), mais sans pouvoir accéder directement aux champs `tab` et `n` afin de préserver la cohérence de l'objet.

Pour ce faire, il suffit d'ajouter `public:` devant les champs que l'on peut consulter en dehors de la classe, et `private:` devant ceux que l'on veut réserver à l'usage interne de la classe. L'exemple précédent devient alors :

```
struct Tableau {

 private:

  double* tab;
  int n;


 public:

  void alloue(int a);
  int consulte_n();
  double consulte_element(int i);
  void modifie_element(int i, double x);

};
```

Dans une classe définie avec le mot-clé `struct`, les champs sont par défaut `public` si aucun mot-clé de protection n'a été précisé avant. Il existe un autre mot-clé pour définir les classes, il s'agit de... `class`. La seule différence entre `struct` et `class` est que, par défaut, si aucun mot-clé de protection n'a été précisé avant, les champs d'une `class` sont `private`. Ainsi l'exemple précédent se ré-écrit:

```
class Tableau {

  double* tab;
  int n;

 public:

  void alloue(int a);
  int consulte_n();
  double consulte_element(int i);
  void modifie_element(int i, double x);
```

```
};
```

Nous utiliserons désormais `class` au lieu de `struct`.

5.4 Appeler l'objet avec `this`

Lorsque l'on définit une fonction membre, on a naturellement accès aux différents champs de la classe. On peut aussi avoir besoin de l'adresse de l'objet qui a appelé la fonction. Cette adresse est notée `this` à l'intérieur de la classe. Ainsi `*this` sera l'objet lui-même et `(*this).n` sera le champ `n` auquel la fonction membre avait naturellement accès directement sous le nom de `n`.

Il existe une autre syntaxe: `this->` a le même sens que `(*this).` ce qui fait que `this->n` est une troisième façon d'accéder au champ `n`.

5.5 Définir un opérateur

Il existe des fonctions particulières que l'on peut définir sur les objets: des opérateurs, c-à-d des fonctions dont l'un des arguments sera située à leur gauche quand elles sont appelées.

Considérons l'opérateur `+`. On pourrait construire une fonction `somme` du type suivant:

```
Tableau somme(Tableau A, Tableau B);
```

qui prend deux `Tableau` comme arguments et en retourne un troisième. L'appel à cette fonction dans le programme se ferait avec la syntaxe habituelle: `somme(A, B)` .

Une première amélioration apportée par les classes est que l'on peut faire de cette fonction une fonction **membre** en la déclarant **dans la classe** sous la forme:

```
Tableau somme(Tableau B);
```

et en l'appelant par `A.somme(B)` .

Mieux encore, il existe certains noms de fonctions membres (`+`, `-`, `=`, `==`, `+=`, etc) pour lesquels le point précédant le nom de la fonction disparaît, de même que les parenthèses nécessaire pour encadrer le second argument. Il s'agit des *opérateurs*. Ils se déclarent de la façon suivante:

```
Tableau operator+(Tableau B);
```

pour s'utiliser ainsi: `A + B` .

Ainsi dans notre exemple l'opérateur `+` serait défini ainsi:

```
class Tableau {

  double* tab;
  int n;

 public:

  void alloue(int a);
  int consulte_n();
  double consulte_element(int i);
  void modifie_element(int i, double x);

  Tableau operator+(Tableau B){
    Tableau C;
    if (n != B.n) {
      // provoquer une erreur car on ne peut pas sommer deux tableaux de
      //   taille différentes !
    } else {
      C.alloue(n);
      for (int i = 0; i<n; i++)
        C.tab[i] = tab[i] + B.tab[i];
    }
    return C;
  }

};
```

et son utilisation, fort pratique, serait:

```
int taille = 4;

Tableau A;
A.alloue(taille);
```

```
Tableau B;
B.alloue(taille);

for (int i = 0; i<taille; i++) {
  A.modifie_element(i, 3.14);
  B.modifie_element(i, 2.78 - i);
}

Tableau C;
C = A + B;
```

L'addition est ici simple, pratique à appeler, mais la modification d'un élément d'un tableau le semble un peu moins. Pour ce faire, il existe d'autres opérateurs, tels l'opérateur parenthèses `()` et l'opérateur crochets `[]` que l'on peut redéfinir. Exemple: on ajoute l'opérateur parenthèses dans la classe par:

```
double operator()(int i) {
  return tab[i];
}
```

On peut dès lors consulter simplement un élément du tableau:

```
Tableau R;
R.alloue(4);
R.modifie_element(0, 3.14);
double z = R(0);
```

Cependant cela ne permet que de consulter une copie de la valeur de `R.tab[i]`, donc on ne peut pas ainsi modifier un élément du tableau. Pour ce faire il faut retourner l'élément du tableau *lui-même*, c-à-d une **référence** sur la variable stockant la valeur de demandée. L'opérateur doit alors être remplacé par:

```
double& operator()(int i) {
  return tab[i];
}
```

et dès ce moment-là on peut modifier les éléments du tableau:

```
Tableau R;
R.alloue(4);
R(0) = 3.14;
```

Le passage précédent pour construire deux tableaux `A` et `B` et les sommer devient:

```
int taille = 4;

Tableau A;
A.alloue(taille);

Tableau B;
B.alloue(taille);

for (int i = 0; i<taille; i++) {
  A(i) = 3.14;
  B(i) = 2.78 - i;
}

Tableau C;
C = A + B;
```

ce qui est plus lisible et pratique d'emploi.

5.6 Constructeur par défaut

Au moment où un objet de type `Tableau` est déclaré, une partie de la mémoire sera réservée pour stocker ses différents champs. Plus précisément, la déclaration de l'objet appelle une fonction membre particulière, désignée sous le nom de constructeur, qui, par défaut, se contente de déclarer les différents champs de l'objet. Si l'un de ces champs est lui-même un objet d'une certaine classe, le constructeur de cette classe sera appelé pour déclarer et constuire ce champ.

Ce constructeur peut être précisé dans la classe si l'on veut exécuter d'autres instructions à ce moment-là. Par convention, le constructeur porte le nom de la classe et ne renvoie rien. Il est nécessairement `public` puisqu'il

est appelé lors de la déclaration d'un objet (en dehors de la classe). Ainsi dans le cas de la classe `Vecteur` on peut préciser le constructeur de la façon suivante:

```
class Tableau {

  double* tab;
  int n;

 public:

  Tableau(){
    n = 0;
  }

  // autres champs et fonctions membres

};
```

Ainsi maintenant, la simple déclaration:

```
Tableau A;
```

aura également pour effet d'initialiser la valeur de `A.n` à `0`.

Le constructeur n'est appelé qu'au moment de la déclaration d'un objet. On ne peut **pas** l'appeler (manuellement par son nom) dans d'autres circonstances.

5.7 Destructeur par défaut

Il existe de même une fonction membre, le destructeur, qui est appelée automatiquement pour détruire un objet à la fin du bloc dans lequel il a été déclaré (de même que l'on détruit les variables locales à un bloc à la fin du bloc). Par défaut, il se contente de détruire les différents champs. Si l'un des champs est un objet d'une certaine classe, il appelle le destructeur de cette classe sur ce champ. Il peut également être précisé (c-à-d redéfini dans la classe). Par convention, son nom est un tilde (`~`) suivi du nom de la classe, et il ne renvoie rien. Il doit également être `public`. Dans notre cas, il sera très utile pour désallouer le tableau `tab`, car en effet, détruire la variable

`double* tab` ne libère pas le tableau qui est pointé par `tab` si celui-ci a été alloué, mais au contraire se contente d'effacer le pointeur. Préciser la désallocation du tableau dans le destructeur permet de ne pas avoir à le faire manuellement pour chacune des variables de type `Tableau` définie dans le code !

```
class Tableau {

  double* tab;
  int n;

 public:

  Tableau(){
    n = 0;
  }

  ~Tableau(){
    if (n != 0)
      delete[] tab;
  }

  // autres champs et fonctions membres

};
```

Le destructeur est appelé automatiquement quand le bloc dans lequel l'objet a été déclaré se termine. Il est impossible de faire en sorte qu'il ne soit pas appelé à ce moment-là, ou de l'appeler manuellement par son nom.

Dans le cas particulier du retour d'une fonction, l'objet renvoyé par `return` est soit directement transmis là où la fonction a été appelée, sans appel au destructeur (il est alors considéré comme local non pas à la fonction mais à l'endroit du code où la fonction a été appelée), soit il est d'abord recopié dans une nouvelle variable (qui elle est considérée comme locale à l'endroit où la fonction a été appelée) et ensuite détruit par appel au destructeur.

5.8 Constructeurs

Il n'existe qu'un destructeur par classe, mais il peut exister plusieurs constructeurs. Ceux-ci ont alors des arguments différents, mais sont formatés de la même façon (même nom, pas de retour). Par exemple:

```
class Tableau {

  double* tab;
  int n;

 public:

  Tableau(){
    n = 0;
  }

  Tableau(int a){
    n = a;
    tab = new double[a];
  }


  // autres champs et fonctions membres, destructeur

};
```

Au moment de la déclaration d'un objet de cette classe, on a alors le choix entre les deux constructeurs:

```
Tableau A;   // appelle le constructeur sans argument
             // et initialise donc A.n à 0

Tableau B(5);  // appelle le constructeur avec un argument de type int
               // et alloue donc un tableau de taille 5
```

Il existe un constructeur particulier, le **constructeur par copie**. Celui-ci prend comme argument une référence (éventuellement constante) sur un objet de la même classe. C'est ce constructeur qui est appelé lorsque l'on passe à un argument à une fonction et que celle-ci recopie automatiquement

cet argument dans une variable locale à elle. Par défaut, ce constructeur recopie un à un les champs de l'objet en argument dans le nouveau. Dans notre cas, il recopierait la valeur du champ `n` ainsi que la valeur du pointeur `tab`, c-à-d qu'il créérait un nouveau pointeur vers le même tableau que précédemment. Si l'on préfère que le nouvel objet ne partage pas le même tableau, mais qu'un nouveau tableau soit alloué et que l'on recopie les différentes cases de l'ancien dans le nouveau, il faut donc procéder ainsi:

```
class Tableau {

  double* tab;
  int n;

 public:

  Tableau();
  Tableau(int a);

  Tableau(const Tableau& X){
    n = X.n;
    tab = new double[n];
    for (int i = 0; i<n; i++)
      tab[i] = X.tab[i];
  }

  // autres champs et fonctions membres,
  // dont l'opérateur parenthèses précédemment introduit
  double& operator()(int a);

};
```

Ainsi, si une fonction `f` a été déclarée de la façon suivante:

```
void f(Tableau X);
```

alors dans le code suivant il y a un appel caché au constructeur par copie:

```
Tableau A(5);   // appel au constructeur prenant
                // un int comme argument
```

```
for (int i=0; i<A.n; i++)
  A(i) = i+1;

Tableau B(A);  // appel au constructeur par copie
Tableau C = A; // appel au constructeur par copie aussi !
               // équivalent à Tableau C(A);

C = A;  // appel à operator= (à ne pas confondre avec le = ci-dessus)

f(A);  // appel au constructeur par copie afin de recopier l'argument
```

car en effet cette dernière ligne se traduira par:

```
{
  Tableau X = B;    // appel au constructeur par copie
  ... // bloc écrit dans la définition de la fonction f
}
```

Attention ! Le symbole `=` dans une déclaration est un raccourci pour appeler le constructeur par copie, alors que ce même symbole hors des déclarations est l'opérateur `=` !

6 Les templates

6.1 Templater une fonction

Considérons la fonction suivante qui retourne la carré d'un entier passé comme argument:

```
int carre(int x){
  int c = x * x;
  return c;
}
```

Nous voudrions disposer de cette fonction non seulement pour les variables de type `int`, mais aussi pour celles de type `double`, `float`, etc. Pour tous ces types différents, la définition de la fonction resterait la même, à ceci près que chaque occurence du mot `int` serait remplacé par `double`, `float`, etc.

Il est possible de ne pas préciser dans une fonction le type d'un ou plusieurs arguments mais de leur donner un nom générique pour les identifier et les utiliser dans le corps de la fonction. A ce moment, lorsque la fonction sera appelée, ce nom générique sera remplacé par le type adéquat. Pour cela il faut remplacer la fonction précédente par:

```
template <class T>
T carre(T x){
  T c = x * x;
  return c;
}
```

Ici, on a choisi `T` pour désigner un type quelconque. La fonction `carre` prend maintenant un objet `x` d'un type quelconque, et renvoie un objet de même type. Entre temps, une variable `c` a été déclarée avec ce même type.

Il est également possible de passer plusieurs types en template. Par exemple:

```
template <class T, class U>
void f(int a, T b, U c, double d){
  ...
}
```

6.2 Gestion par le compilateur

Dans l'exemple précédent, la fonction `carre` ne sera compilée que pour les valeurs de `T` pour lesquelles elle est effectivement appelée dans le programme. Ainsi, si elle n'est jamais appelée, elle ne sera pas compilée du tout, et si elle est appelée avec seulement les types `int` et `float`, elle ne sera compilée que pour ces deux types, et pas pour le type `double` par exemple. Par conséquent, il n'y aura pas de problème s'il existe un type particulier pour lequel l'opérateur `*` n'existe pas (alors qu'il est appelé dans la fonction `carre`) du moment que la fonction `carre` n'est jamais appelée sur une variable de ce type-là.

Note pour la compilation: comme les fonction templatées ne sont compilées que pour les types pour lesquels elles sont effectivement appelées, si votre programme est divisé en plusieurs fichiers compilés séparément, il

faudra faire attention à mettre les définitions des fonctions templatées dans des .h et non pas dans des .cpp.

6.3 Spécification

Il est possible de spécifier des définitions particulières de la fonction pour certaines valeurs des templates. Ainsi l'on peut ajouter **après** le bloc:

```
template <class T, class U>
void f(int a, T b, U c, double d){
  ... // des instructions
}
```

une redéfinition de `f` pour le cas particulier où `T` et `U` sont `int`:

```
void f(int a, int b, int c, double d){
  // d'autres instructions
}
```

C'est cette fonction-là qui sera désormais appelée si `T` et `U` sont `int`.

Lors d'une spécification, on n'est pas obligé de préciser tous les templates. De même que pour les arguments par défaut des fonctions, il faut spécifier en commençant par les types les plus à droite derrière le mot-clé `template` (c-à-d qu'ici on ne peut pas spécifier `T` sans spécifier `U` en même temps).

6.4 Templater une classe

Une classe peut aussi être templatée. Par exemple, si l'on souhaite que l'un des champs ait un type générique `T` plutôt qu'un type précis:

```
template <class T>
class Vecteur {
  int n;
  T* tableau;
 public:
  Vecteur(const Vecteur<T>& A){}

  // etc.
```

```
};
```

Les différentes classes correspondantes s'appellent `Vecteur<int>` (pour `T = int`), `Vecteur<double>`, `Vecteur<bool>`, etc. Attention, `Vecteur` lui-même n'est pas le nom d'une classe. Cette syntaxe explique que le constructeur par copie indiqué dans l'exemple précédent prend comme argument un `const Vecteur<T>&`.

Ainsi pour déclarer (avec le constructeur vide) un objet `A` de type `Vecteur<T>` pour `T = int` on fait simplement:

```
Vecteur<int> A;
```

7 Héritage

7.1 Syntaxe

7.2 Fonctions virtuelles

7.3 Constructeur

7.4 Destructeur

7.5 Protection: `public, protected et private`

8 Comment compiler

Les lignes de commande données dans ce chapitre sont spécifiques à la plupart des systèmes d'exploitation de type Linux. Pour certains Linux il faudra remplacer `make` par `gmake`. Sous Windows il faudra consulter l'aide du compilateur que vous aurez choisi (par exemple celui intégré dans VisualC++ 2005[1]).

8.1 Fichier simple

Dans le cas où tout le code figure dans un seul fichier `code.cpp` et que l'on veut en faire un exécutable nommé `exec`, il suffit de taper la ligne:
`g++ -o exec code.cpp`

[1] http://msdn.microsoft.com/vstudio/express/visualc/

Si des bibliothèques particulières sont incluses dans le code (comme `#include <cmath>`), il faut indiquer au compilateur où aller chercher ces bibliothèques. En l'occurence pour la bibliothèque standard `<cmath>` il faut taper:
```
g++ -lm -o exec code.cpp
```

Pour utiliser une bibliothèque personnelle `mabibli.h` il faut ajouter `-I` suivi du chemin pour y accéder. Ainsi si le fichier `mabibli.h` est dans le répertoire courant il faudra taper:
```
g++ -I. -o exec code.cpp
```

Pour que TOUS les warnings soient affichés (ce que je vous conseille fortement), l'option `-Wall` doit être ajoutée.

8.2 Plusieurs fichiers

L'organisation générale d'un code en `C++` consiste à séparer les classes et fonctions dans des fichiers différents et d'inclure ces fichiers dans le principal (souvent appelé `main.cpp`). Plus précisément, afin de gagner du temps lors de la compilation de `main.cpp` et de ne pas devoir tout recompiler à chaque fois, on met les déclarations des fonctions qui seront appelées plus tard (dans d'autres fichiers) dans des `fichiers.h` et les définitions dans des `fichiers.cpp` associés. Seuls les `fichiers.h` seront inclus par des `#include`. En conséquence, les décalarations des fonctions utiles, classes, ainsi que les définitions des choses templatées doivent figurer dans les `fichiers.h` inclus.

Pour compiler séparément un fichier auxiliaire `aux.cpp`, il suffit de taper:
```
g++ -I. -c aux.cpp -o aux.o
```

Pour compiler le fichier principal `main.cpp` qui inclut `aux.h`, et en faire un exécutable nommé `exec`, il faudra d'abord compiler `main.cpp` à proprement parler:
```
g++ -I. -c main.cpp -o main.o
```
puis *linker* les fichiers entre eux, c-à-d lier `main.o` et `aux.o` correctement de façon à construire un exécutable nommé `exec`:
```
g++ -I. -o exec main.o aux.o
```

8.3 Makefile

On peut inclure toutes les instructions de compilation (et de linkage) dans un fichier nommé `Makefile` et se contenter de taper la commande `make`. La syntaxe d'un fichier `Makefile` est la suivante:

```
# le # est le symbole du commentaire dans les Makefile

#Préciser que l'on veut tous les warnings
CXXFLAGS+=-Wall

#Au choix:
#1 - Si l'on veut débugguer le programme
CXXFLAGS+=-g -D_DEBUG
#2 - Si l'on veut optimiser l'exécutable (pour qu'il soit le plus rapide possible)
#CXXFLAGS+=-Os -funroll-all-loops -DNDEBUG

#Préciser où chercher les .a et .so
LDLIBS+=-lm -L.

#Où chercher les .h
CXXFLAGS+=-I.

#Compiler les .cpp (les écarts à la ligne sont une tabulation)
%.o: %.cpp
        $(CXX) -c $(CXXFLAGS) -o $@ $*.cpp

#Remarque: CXX est le compilateur, g++ par défaut.
#Vous pouvez le modifier pour préciser une version particulière de g++.

#Que compiler par défaut
PROG=mon_executable
all:: $(PROG)

#Liste de fichiers.o à inclure
OBJ=main.o aux.o

#Quand quels fichiers ont été modifiés faut-il recompiler les .o ?
$(OBJ): aux.h Makefile
```

```
#Comment compiler (linker) mon_executable
$(PROG): $(OBJ)
        $(CXX) -o $(PROG) $(OBJ) $(LDLIBS)

#Faire le ménage quand on tape make clean
clean:
        rm -f $(PROG) $(OBJ)
```

8.4 Instructions au pré-processeur et au compilateur

8.4.1 Commentaires

Le double slash `//` est le symbole du commentaire: le compilateur arrête la lecture d'une ligne à la vue de ce symbole. Ainsi

```
int n = 4; // blablabla... = 5; plutôt ?
// n++;
```

est équivalent à

```
int n = 4;
```

Pour commenter tout un bloc, on peut également utiliser `/*` et `*/`. Tout ce qui figure entre ces symboles est ignoré. Attention à ne pas en imbriquer plusieurs ! Ainsi

```
int n = 4;
/* blablabla... = 5; plutôt ?
   n++;  */
```

est équivalent à la même chose qu'avant.

8.4.2 `#define, #ifdef`

8.4.3 `inline`

On peut faire l'équivalent de `#define` sur des fonctions, en ajoutant le mot-clé `inline` devant leur déclaration-définition.

```
inline int carre(int a){
  return a*a;
}
```

Tout appel à la fonction `carre` sera alors directement remplacé par le code de la fonction avant compilation, ce qui peut permettre dans certains cas un gain de temps à l'éxécution.

9 Les librairies standards

Cette section est très incomplète, elle est là juste pour donner des indices sur comment procéder pour afficher un message, écrire dans un fichier, etc. Pour plus d'informations, vous pouvez par exemple consulter le cours en ligne de Christian Casteyde: `http://casteyde.christian.free.fr/cpp/cours/online/book1.html`

9.1 Les chaînes de caractères

Il existe en `C++` une classe plus pratique que le type de base `char*` pour les manipulation de base des chaînes de caractères, il s'agit de la classe `string` (inclure pour cela `#include <string>`). Exemple d'utilisation:

```
string a = "Bonjour";
string nom = "Marie";
string b = a + " " + nom;  // concaténation de chaînes
char* c = b.c_str();  // transformer un string en char*
a = string(c);            // transformer un char* en string
```

Pour rechercher une sous-chaîne dans un `string a`, on peut utiliser la fonction membre `find` comme dans l'exemple `a.find("jour")`. Pour plus de détails sur les `string`, vous pouvez par exemple consulter `http://casteyde.christian.free.fr/cp`

Pour créer et extraire des chaînes de caractères cependant, les bonnes vieilles fonctions `sprintf` et `sscanf` du `C` sur les `char*` sont toujours aussi pratiques. Ainsi:

```
char* c = new char[200];
sprintf(c, "Bonjour %s pour la %dième fois !", "Marie", 2);
```

écrira, dans `c`, la chaîne `Bonjour Marie pour la 2ième fois !`. En effet, les `%` suivis d'une lettre sont remplacés dans l'ordre par les arguments suivants, la lettre spécifiant le type de l'agument (on peut également éventuellement

spécifier le format): `%s` indique une chaîne de caractères tandis que `%d` indique un entier. Plus de détails dans les pages de `man` (taper `man sprintf` dans un terminal pour avoir l'aide sur cette fonction) !

9.2 Affichage dans le terminal

Pour afficher une chaîne de caractère `chaine` dans le terminal (c-à-d dans la fenêtre depuis laquelle on a lancé le programme), on peut, après avoir inclu la bilbiothèque `iostream` par `#include <iostream>`, utiliser l'instruction suivante:

```
cout << chaine;
```

On peut afficher plusieurs chaînes à la suite les unes des autres:

```
cout << chaine1 << chaine2;
cout << chaine3;
```

Le mot-clé pour revenir à la ligne (et par la même occasion forcer le terminal à se rafraîchir) est `endl`.

```
cout << chaine1 << endl << chaine2;
cout << endl << endl << chaine3 << endl;
```

Pour forcer le rafraîchissement du terminal sans faire un retour à la ligne, l'instruction suivante peut s'avérer utile:

```
cout.flush();
```

Cela marche aussi avec autre chose que des chaînes de caractères (`char*` ou `string`), par exemple `int` et `double`.

```
int n = 4;
double x = 3.14;
cout << "n et x valent respectivement " << n << " et " << x << "." << endl;
```

Je ne rentrerai pas dans les détails, mais il est possible de redéfinir l'opérateur `<<` afin de pouvoir l'utiliser sur de nouvelles classes (pour afficher les champs de la classe). `cout` est en fait un objet (l'unique ?) de type `ostream`.

9.3 Lire une entrée dans le terminal

Il s'agit de l'opposé de `cout`, qui s'appelle `cin`.

```
cout << "Veuillez entrer un entier: ";
cout.flush();
int n;
cin >> n;
cout << endl << "Vous avez rentré l'entier " << n << "." << endl;
```

Ici, après affichage du message `Veuillez entrer un entier:` , il vous faudra entrer un entier dans le terminal avant de pouvoir continuer le programme, c-à-d qu'il faudra taper un entier suivi de `Entrée`. Si vous appuyez sur `Entrée` sans avoir introduit précédemment un entier, le terminal attendra à nouveau que vous rentriez un entier. La valeur entière sera alors affectée à la variable `n`. De même que pour `cout`, la façon de procéder est identique pour lire un `double` ou une chaîne de caractères `string`.

Pour lire une ligne complète depuis le terminal sans se soucier de ce qu'elle contient (c-à-d qu'elle peut même être vide, du moment que la touche `Entrée` a été employée), on peut utiliser la fonction `getline` de la façon suivante:

```
cout << "Veuillez entrer quelque chose: ";
cout.flush();
string a;
getline(cin, a);
```

9.4 `stringstream`

La bibliothèque `stringstream` permet de manipuler des chaînes, en particulier d'en extraire des informations d'un type donné. Exemple:

```
cout << "Veuillez entrer un entier, un mot et une valeur approchée de pi: ";
cout.flush();
string a;
getline(cin, a);
stringstream s;
```

```
s << a;
int n;
string mot;
double vpi;
s >> n;
s >> mot;
s >> vpi;
cout << "Vous avez entré: " << n << ", " << mot << " et " << vpi << "." << endl;
```

9.5 Lecture/écriture dans un fichier

Avec la bibliothèque `fstream`, qui contient en particulier une classe `ifstream` pour lire des fichiers et une autre `ofstream` pour écrire dans un fichier. Utilisation de base:

Pour lire:

```
ifstream f;
f.open("monfichier.txt");
int n;
f >> n; // si je sais que le fichier commence par un entier
string ligne;
getline(f, ligne);
f.close();
```

Pour écrire:

```
ofstream f;
f.open("monfichier.txt");
int n = 42;
string s = "Bonjour";
f << s << " " << n << endl;
f.close();
```

9.6 Les vecteurs

Il existe une classe templatée `vector<T>`...

9.7 Les listes et les itérateurs